Índice general

1. Introducción **1**

1.1. Alfabetos y Cadenas. 1

1.2. Lenguajes. 3

1.3. Introducción a la Técnicas Básicas de Demostración. 6

1.4. El Concepto de Gramática. 8

1.5. Clasificación Jerárquica de las Gramáticas. 11

2. Autómatas Finitos y Expresiones Regulares **15**

2.1. Autómatas Finitos Deterministas. 15

2.2. Autómatas Finitos No Deterministas. 19

2.3. Autómatas Finitos con Movimientos Libres. 24

2.4. Expresiones Regulares. 29

2.5. Otros Tipos de Autómatas Finitos. 36

2.5.1. Autómatas Finitos Bidireccionales 36

2.5.2. Autómatas Finitos con Función de Salida. 38

2.6. Aplicaciones. 39

2.6.1. Aplicación como ayuda al análisis léxico en un compilador. 40

2.6.2. Especificación de parámetros de entrada. 40

2.6.3. Ayuda al diseño de circuitos. 41

3. Propiedades de los Lenguajes Regulares **43**

3.1. El Teorema de Myhill-Nerode. Minimización de Autómatas Finitos. 43
3.2. Lema de Bombeo. 50
3.3. Propiedades de Clausura. 53
3.4. Algoritmos de Decisión para Lenguajes Regulares. 57

4. Gramáticas de Contexto Libre **59**
4.1. Gramáticas y Lenguajes de Contexto Libre. 59
4.2. Árboles de Derivación. 60
4.3. Simplificación de GCL. 63
4.4. Formas Normales. 69
4.4.1. Forma Normal de Chomsky, FNCH. 70
4.4.2. Forma Normal de Greibach, FNG. 73

5. Autómatas de Pila **79**
5.1. Introducción. 79
5.2. Relación entre los Autómatas de Pila y los LCL. 87

6. Propiedades de los Lenguajes de Contexto Libre **91**
6.1. Lema de Bombeo. 91
6.2. Propiedades de Clausura. 96
6.3. Algoritmos de Decisión. 103

7. Máquinas de Turing **109**
7.1. Modelo de Máquina de Turing. 109
7.2. Técnicas para la Construcción de Máquinas de Turing. 115
7.2.1. Almacenamiento en el Control Finito. 115
7.2.2. Cintas con Sectores Múltiples y Marcaje de símbolos. 116
7.2.3. Uso de Subrutinas. 117
7.3. Máquinas de Turing Modificadas. 118

7.3.1. Máquina de Turing con Cinta Infinita en Ambos Sentidos. 118
7.3.2. Máquina de Turing Multicinta. 120
7.3.3. Máquina de Turing con Múltiples Cabezales. 122
7.3.4. Máquina de Turing Offline. 122
7.3.5. Máquina de Turing No Determinista. 123
7.3.6. Máquinas de Turing Restringidas. 125
7.4. Relación de la Máquina de Turing con otros Autómatas. 126

8. Computabilidad **129**
8.1. Lenguajes Recursivos y Funciones Computables. 129
8.1.1. Introducción. 129
8.1.2. La Máquina de Turing como Reconocedor de Lenguajes. 132
8.1.3. La Máquina de Turing como Calculador de Funciones. 133
8.2. La Tesis de Church. 134
8.2.1. El Modelo RAM. 135
8.3. La Máquina de Turing como Generador. 136
8.3.1. Dos Máquinas de Turing Generadoras Básicas. 137
8.4. Caracterización de L.R. y L.R.E. 139
8.4.1. Caracterización de los L.R.E. mediante Generadores. 139
8.4.2. Caracterización de los L.R. mediante Generadores. 141
8.5. Propiedades de L. R. y L. R. E. 144

9. Indecidibilidad **147**
9.1. Concepto de Problema. 147
9.1.1. Introducción. 147
9.1.2. Concepto de Problema. 150
9.2. La Máquina Universal de Turing y Dos Problemas Indecidibles. 152
9.2.1. Codificación de Máquinas de Turing. 152

9.2.2. Ejemplo de un Lenguaje que NO es Recursivamente Enumerable. . 153
9.2.3. La Máquina Universal de Turing. 155
9.2.4. Dos Problemas Indecidibles. 157
9.3. Teorema de Rice. Más Problemas Indecidibles. 162
9.3.1. El Problema de la Vaciedad. 162
9.3.2. El Teorema de Rice. 165
9.4. La Indecidibilidad del Problema de la Correspondencia de Post. 168
9.5. Apéndice 1: Otro Ejemplo de Reducción. El Problema de la Recursividad. . 173
9.6. Apéndice 2: Ejemplo de cómo aplicar Rice fuera de su ámbito. 176

10. Introducción a la Complejidad Computacional 177

10.1. Introducción. 177
10.1.1. El Dilema del Contrabandista. 178
10.2. Definiciones Básicas. 180
10.2.1. Clases de Complejidad. 183
10.3. Relaciones entre las Clases de Complejidad Computacional. 184
10.3.1. Relaciones entre Clases Deterministas y No Deterministas. 185
10.3.2. Las Clases de Complejidad Polinómica. 187
10.4. Introducción a la Teoría de Complejidad Computacional. 190

Capítulo 1

Introducción

Índice General

1.1. Alfabetos y Cadenas. **1**
1.2. Lenguajes. **3**
1.3. Introducción a la Técnicas Básicas de Demostración. **6**
1.4. El Concepto de Gramática. **8**
1.5. Clasificación Jerárquica de las Gramáticas. **11**

1.1. Alfabetos y Cadenas.

Un *alfabeto* es un conjunto *finito y no vacío* de elementos denominados *símbolos*. Los alfabetos se definen bien sea por *extensión*, enumeración de sus símbolos, o bien por *comprensión*, enunciando alguna propiedad que todos los símbolos han de cumplir.

Algunos ejemplos de alfabetos son los siguientes:

1. $\Sigma = \{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z\}$, *el alfabeto latino,*
2. $\Sigma = \{0, 1\}$, *el alfabeto binario,*
3. $\Sigma = \{a_i \mid i \in \mathcal{I}\}$, *donde* $\mathcal{I}$ *es un conjunto finito, como por ejemplo* $\mathcal{I} = \{x \in N \mid x > 0 \wedge x < 50\}$.

Los dos primeros alfabetos se han definido por extensión, mientras que el tercero se ha definido por comprensión.

Dado un alfabeto una *cadena* de ese alfabeto es una combinación de símbolos de ese alfabeto.

Por ejemplo, con el alfabeto (1) una cadena sería "mar", con el alfabeto (2) sería "0111" y con el alfabeto (3) sería "$a_1a_5a_{49}$".

Se define una cadena especial denominada *cadena nula*, denotada por λ, caracterizada por no poseer símbolos.

Se denomina *prefijo* a cualquier secuencia de símbolos de la parte inicial de una cadena, y *sufijo* a cualquier secuencia de símbolos de su parte final.

Por ejemplo, la cadena "0111" tiene como prefijos a las siguientes cadenas: λ, "0", "01", "011" y "0111" mientras que son sus sufijos las cadenas: λ, "1", "11", "111", "0111".

Existe un gran número de operaciones básicas que se pueden realizar con las cadenas. De entre ellas, se destacan:

Longitud de una cadena Se denomina longitud de una cadena *x*, denotado por |x|, a la operación que consiste en contar el número de símbolos que componen a la cadena *x*. Esta operación cumple las siguientes propiedades:

1. $| \lambda |$ = 0
2. |a| = 1, si *a* es un símbolo del alfabeto
3. Si y = $a_1a_2..a_n$, x = $a_1a_2..a_nb$ y $a_i \in \Sigma, b \in \Sigma$, entonces |y| = n $\wedge$ |x| = n + 1.

Dado un alfabeto Σ, se define Σ^n como el conjunto de cadenas de longitud *n* que se pueden construir con ese alfabeto. Así, por ejemplo, siempre se cumple que sea cual sea el alfabeto Σ, $\Sigma^0 = \{\lambda\}$.

Concatenación Sean *x* e *y* dos cadenas, entonces la cadena resultante de concatenar *x* con *y* es la cadena *xy*, cumpliéndose que si $| x |$ = n y $| y |$ = m entonces $| xy |$ = $| x |$ + $| y |$ = n + m. Cuando se concatena una cadena consigo misma varias veces se emplea la potencia para denotarlo; por ejemplo, la cadena *xxxx* se denota como x^4. Además, como consecuencia del uso de esta notación, resulta que $| x^n |$ = n$| x |$.

Estrella de Kleene Se denomina estrella de Kleene, *cierre reflexivo transitivo* u *operación asterisco* sobre un alfabeto Σ al conjunto de todas las cadenas que se pueden construir a partir de los símbolos del alfabeto Σ; en notación matemática esta definición se expresa así: $\Sigma^* = \bigcup_{i \geq 0} \Sigma^i$.

Cierre transitivo El cierre transitivo u *operación más*, se denota como Σ^+ y se define como $\Sigma^+ = \bigcup_{i > 0} \Sigma^i$.

De las definiciones de cierre reflexivo transitivo y de cierre transitivo, se puede deducir que $\Sigma^* = \{\lambda\} \cup \Sigma^+$.

Inversión Dada una cadena $x \in \Sigma^*$, $x = a_1a_2 \ldots a_n$, se define la cadena inversa de *x*, denotada por x^{-1}, a la cadena $a_n \ldots a_2a_1$. Esta operación se puede definir también recursivamente:

1. $\lambda^{-1} = \lambda$
2. $(xa)^{-1} = a(x^{-1})$, con $x \in \Sigma^* \wedge a \in \Sigma$.

De esta definición se desprenden varias propiedades entre las que cabe destacar las siguientes:

1. $a^{-1} = a$ si $a \in \Sigma$, la inversa de un símbolo es el mismo símbolo,
2. $(x^{-1})^{-1} = x$, $x \in \Sigma^*$, la inversa de la inversa es la cadena original,
3. $(x_1 x_2 \dots x_k)^{-1} = x_k^{-1} \dots x_2^{-1} x_1^{-1}$, $x_i \in \Sigma^*$.

1.2. Lenguajes.

Dado un alfabeto Σ se dice que *L* es un *lenguaje* si es un subconjunto de Σ^*, es decir, $L \subseteq \Sigma^*$.

Por ejemplo, sea el alfabeto Σ = {*0,1*}. L_1 = {*0, 01, 011*} *es un lenguaje finito sobre el alfabeto* Σ *y* $L_2 = \{0^{2n} \mid n \geq 0\}$ *es otro lenguaje, esta vez con un número de cadenas infinito, sobre el mismo alfabeto.*

Puesto que se ha definido un lenguaje como un conjunto de cadenas, las operaciones que se pueden realizar sobre lenguajes son las mismas que las que se pueden realizar sobre conjuntos. De entre ellas, se destacan las siguientes:

Unión de lenguajes $L_1 \cup L_2 = \{x \in \Sigma^* \mid x \in L_1 \vee x \in L_2\}$.

Cumple las propiedades

1. Asociativa, $L_1 \cup L_2 \cup L_3 = (L_1 \cup L_2) \cup L_3 = L_1 \cup (L_2 \cup L_3)$,
2. Commutativa, $L_1 \cup L_2 = L_2 \cup L_1$, y
3. Elemento neutro, $L \cup \emptyset = \emptyset \cup L = L$.

Intersección de lenguajes $L_1 \cap L_2 = \{x \in \Sigma^* \mid x \in L_1 \wedge x \in L_2\}$.

Cumple las propiedades

1. Asociativa, $L_1 \cap L_2 \cap L_3 = (L_1 \cap L_2) \cap L_3 = L_1 \cap (L_2 \cap L_3)$,
2. Commutativa, $L_1 \cap L_2 = L_2 \cap L_1$, y
3. Elemento neutro, $L \cap \Sigma^* = \Sigma^* \cap L = L$.

Concatenación $L_1 L_2 = \{x \in \Sigma^* \mid x = yz \wedge y \in L_1 \wedge z \in L_2\}$.

Cumple las propiedades

1. Asociativa, $L_1 L_2 L_3 = (L_1 L_2) L_3 = L_1 (L_2 L_3)$, y
2. Elemento neutro, $L\{\lambda\} = \{\lambda\} L = L$.

En la concatenación de lenguajes también se utiliza la notación potencia tal y como se definió en la concatenación de cadenas, es decir, el lenguaje *LLLL* se denota como L^4.

Estrella de Kleene $L^* = \bigcup_{i \geq 0} L^i$.

Cierre transitivo $L^+ = \bigcup_{i > 0} L^i$.

Al aplicar las operaciones *asterisco* y *más* sobre lenguajes, se puede comprobar que se cumplen las siguientes propiedades:

a) $(L^*)^* = L^*$ b) $L^+ = (L^*)L$
c) $(L^+)^* = L^*$ d) $(L^*)^+ = L^*$
e) $(L^+)^+ = L^+$ f) si $L_1 \subseteq L_2$, entonces $L_1^* \subseteq L_2^* \wedge L_1^+ \subseteq L_2^+$

Dependiendo de si λ pertenece o no pertenece a un lenguaje *L* se establece la siguiente relación entre la operación *más* y la operación *asterisco*:

$$L^+ = L^*, \ si \ \lambda \in L \ y \ L^+ = L^* - \{\lambda\}, \ si \ \lambda \notin L.$$

Diferencia de lenguajes $L_1 - L_2 = \{x \in \Sigma^* \mid x \in L_1 \wedge x \notin L_2\}$.

Se cumple que si $L_1 \subseteq L_2$ entonces $L_1 - L_2 = \emptyset$ puesto que todas las cadenas de L_1 pertenecen a L_2, y que si $L_1 \cap L_2 = \emptyset$ entonces $L_1 - L_2 = L_1$, puesto que ambos lenguajes no tienen cadenas en común.

Complemento Sea *L* un lenguaje, entonces el lenguaje complementario de *L*, denotado por $\bar{L}$, se define como $\bar{L} = \Sigma^* - L = \{x \in \Sigma^* \mid x \notin L\}$.

Cumple las propiedades

1. $\bar{\bar{L}} = L$,
2. $L \cup \bar{L} = \Sigma^*$, y
3. $L \cap \bar{L} = \emptyset$.

Partes de un conjunto Una de las operaciones básicas sobre conjuntos consiste en calcular *el conjunto de todos los conjuntos que se pueden formar con los elementos del conjunto original*. Esta operación se denomina *cálculo de las partes de un conjunto* y, si el conjunto se denomina *A*, se denota como *P(A)* o 2^A.

Por ejemplo, si A es el conjunto A = {a,b,c} entonces el cálculo de las partes del conjunto A, 2^A o P(A), es el siguiente:

$$2^A = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$$

Una vez establecida la operación partes de un conjunto queda por destacar cual es el resultado al aplicarla a un conjunto con un número infinito de elementos. Por ejemplo, dado un alfabeto Σ, considérese 2^{Σ^*}. Con esta operación se denota al conjunto de todos los conjuntos que se pueden formar con los elementos del alfabeto Σ, es decir, todos los posibles lenguajes que se pueden formar con las cadenas de símbolos del alfabeto Σ.

La observación anterior permite realizar una consideración interesante. Sea el alfabeto Σ = {0,1}; en el conjunto 2^{Σ^*} estarían, entre otros muchos conjuntos, el conjunto de cadenas que representan el juego de instrucciones de cualquier microprocesador (68000, pentium, athlon, ...etc.). Más aún, si se concatena en una misma cadena todas las instrucciones de un programa escrito en uno de estos juegos de instrucciones, también se obtiene un elemento de 2^{Σ^*}. Se tiene, entonces, que, de entre los elementos de 2^{Σ^*}, un subconjunto sería el formado por todas cadenas que representan programas correctamente escritos en ese juego de instrucciones, y que, además, un subconjunto de este sería el de los que no sólo son correctos desde el punto de vista sintáctico, sino que también se ejecutan de forma correcta.

Inversión Dado un lenguaje *L*, $L \subseteq \Sigma^*$, se define el lenguaje inverso de *L*, denotado por L^{-1}, como el conjunto de cadenas que resulta de aplicar la operación de inversión a cada una de las cadenas del lenguaje *L*, es decir, $L^{-1} = \{x^{-1} \mid x \in L\}$.

Se puede comprobar que esta operación cumple las siguientes propiedades:

a) $(\Sigma^{-1}) = \Sigma$ b) $(L_1L_2)^{-1} = L_2^{-1}L_1^{-1}$
c) $(L^i)^{-1} = (L^{-1})^i$ d) $(L^*)^{-1} = (L^{-1})^*, (L_1 \cup L_2)^{-1} = L_1^{-1} \cup L_2^{-1}$
e) $(\Sigma^*)^{-1} = \Sigma^*$ f) $(L^{-1})^{-1} = L$

Sustitución Sean Σ y Γ dos alfabetos. La operación de *sustitución* se define como la operación consistente en *asociar a cada símbolo de un alfabeto, un lenguaje sobre el otro alfabeto*, es decir, f es una operación de sustitución si:

$$f : \Sigma \to 2^{\Gamma^*} \mid \forall a \in \Sigma, f(a) \subseteq \Gamma^*.$$

Esta definición se puede extender a cadenas de símbolos de la forma recursiva siguiente:

$$\mathcal{F} : \Sigma^* \to 2^{\Gamma^*}, \; tal\ que \begin{cases} \mathcal{F}(\lambda) = \{\lambda\}, \\ \mathcal{F}(xa) = \mathcal{F}(x)f(a), \; a \in \Sigma \wedge x \in \Sigma^*. \end{cases}$$

Para evitar confusiones se utilizará el mismo símbolo, *f*, para expresar la operación de sustitución sobre símbolos que sobre cadenas.

La última extensión que se puede realizar con esta operación consiste en aplicarla sobre lenguajes. Sea *L* un lenguaje sobre el alfabeto Σ^*, es decir, $L \subseteq \Sigma^*$, entonces la operación de sustitución sobre el lenguaje *L* se define como $f(L) = \bigcup_{x \in L} f(x)$.

Homomorfismo Es un caso particular de la operación de sustitución, que consiste en *asociar a cada símbolo de un alfabeto, una cadena sobre otro alfabeto*,

$$h : \Sigma \to \Gamma^* \mid \forall a \in \Sigma, h(a) \in \Gamma^*.$$

Esta definición se puede extender a cadenas y lenguajes de la forma siguiente (se mantiene el símbolo de función *h* para ambos casos):

$$h : \Sigma^* \to \Gamma^*, \; tal\ que \begin{cases} h(\lambda) = \{\lambda\}, \\ h(xa) = h(x)h(a), \; a \in \Sigma \wedge x \in \Sigma^*. \end{cases}$$

Sea $L \subseteq \Sigma^*$, entonces $h(L) = \bigcup_{x \in L} h(x)$.

Homomorfismo inverso Sea $L \in \Gamma^*$ un lenguaje, entonces la operación de *homomorfismo inverso*, denotada $h^{-1}(L)$ se define como el conjunto de cadenas $h^{-1}(L) = \{x \in \Sigma^* \mid h(x) \in L\}$.

Con estas dos últimas operaciones se cumple que

1. $h(h^{-1}(L)) \subseteq L, L \subseteq \Gamma^*$, y
2. $L \subseteq h^{-1}(h(L)), L \subseteq \Sigma^*$.

Por ejemplo: Sea h(0) = aa y h(1) = bb.
Sea L_1 = {*ab, ba*} $\subseteq$ {*a, b*}*. *Entonces,* $h^{-1}(L_1) = \emptyset$.
Sea L_2 = {*aabb, aab, bba*} $\subseteq$ {*a, b*}*. *Entonces,* $h^{-1}(L_2) = \{01\}$.

1.3. Introducción a la Técnicas Básicas de Demostración.

Aunque cada demostración es diferente, puesto que cada una se aplica a enunciados diferentes, sí que se pueden establecer una serie de técnicas de demostración que se aplican muy frecuentemente en enunciados de teoría de autómatas y lenguajes formales.

La primera de estas técnicas es la denominada *demostración por inducción*. Toda demostración por inducción se compone de tres pasos: el *paso base*, que hay que demostrar cierto, y se suele corresponder con la instancia más sencilla de la propiedad a demostrar, la *hipótesis de inducción*, que se supone cierta, y el *paso de inducción*, que hay que demostrar utilizando la hipótesis de inducción y el paso base.

Como ejemplo de aplicación de esta técnica se va a demostrar el siguiente enunciado:

Proposición 1.1 *Si A es un conjunto finito, entonces el número de elementos del conjunto* 2^A *es* $2^{|A|}$ *siendo* $|A|$ *el número de elementos del conjunto A, es decir, su cardinal.*

Demostración:

Paso Base: *Sea A un conjunto de cardinalidad 0, es decir, A =* $\emptyset$. *Entonces se cumple que* 2^A = {$\emptyset$}, *por lo tanto,* $\mid 2^A \mid$ = *1. También se cumple que* $2^{|A|} = 2^0 = 1$.

Hipótesis de Inducción: *Sea* $n \geq 0$ *y sea A un conjunto de cardinalidad n, se supone cierto que* $\mid 2^A \mid = 2^{|A|}$.

Paso de Inducción: *Sea A un conjunto tal que* $|A|$ = *n + 1. Como* $n \geq 0$ *entonces el conjunto A contiene al menos 1 elemento, denotémoslo por a.*

Sea el conjunto B, B = A - $\{a\}$; entonces $|B| = n$. Por Hipótesis de inducción se sabe que al ser B un conjunto con n elementos se cumple $\mid 2^B \mid = 2^{|B|} = 2^n$.

Se puede dividir el conjunto potencia de A en dos partes, aquellos conjuntos que no contienen al elemento a, es decir, 2^B, y por otra parte aquellos conjuntos que contienen al elemento a, es decir, C = $\{X \cup \{a\} \mid X \in 2^B\}$, que se obtienen introduciendo en cada conjunto de 2^B el elemento a. Esta división particiona el conjunto 2^A en dos partes disjuntas que poseen el mismo número de elementos. Por lo que la cardinalidad del conjunto 2^A será:

$$\mid 2^A \mid = \mid 2^B \mid + \mid 2^C \mid = 2^n + 2^n = 2^{n+1} = 2^{|A|}.$$

c.q.d.

Otra técnica de demostración muy utilizada consiste en la denominada *reducción al absurdo*, en la que se establece una suposición inicial y al ser consecuente con ella se llega a una conclusión que la refuta. Por lo tanto, esa suposición inicial no puede ser correcta puesto que, de ser así, entonces se establecería un enunciado que resulta ser a la vez cierto y falso.

Como ejemplo de aplicación de esta técnica se va a demostrar el siguiente enunciado:

Proposición 1.2 *Dados los lenguajes L_1 y L_2, L_1=L_2 si, y sólo si, $(L_1 \cap \bar{L_2}) \cup (\bar{L_1} \cap L_2) = \emptyset$.*

Demostración:

"⇒" $L_1 = L_2 \Rightarrow (L_1 \cap \bar{L_2}) \cup (\bar{L_1} \cap L_2) = \emptyset$

Por reducción al absurdo. Supóngase que dado cierto que $L_1 = L_2$, entonces $(L_1 \cap \bar{L_2}) \cup (\bar{L_1} \cap L_2) \neq \emptyset$. Si esto ocurre entonces o bien $x \in (L_1 \cap \bar{L_2})$ o bien $x \in (\bar{L_1} \cap L_2)$. En cualquiera de estos casos la cadena x pertenece a uno de los dos lenguajes y no al otro; por lo tanto, ambos lenguajes no pueden ser iguales (difieren en al menos una cadena), lo que contradice la suposición inicial. Por lo tanto, esa suposición no puede ser cierta.

"⇐" $(L_1 \cap \bar{L_2}) \cup (\bar{L_1} \cap L_2) = \emptyset \Rightarrow L_1 = L_2$

Por reducción al absurdo. Supóngase que dado cierto que $(L_1 \cap \bar{L_2}) \cup (\bar{L_1} \cap L_2) = \emptyset$, entonces $L_1 \neq L_2$. Si esto ocurre es porque existe al menos una cadena que pertenece a un lenguaje y no pertenece al otro, de lo que se deduce que $(L_1 \cap \bar{L_2}) \cup (\bar{L_1} \cap L_2) \neq \emptyset$, lo que contradice la suposición inicial. Por lo tanto, esa suposición no puede ser cierta.

c.q.d.

1.4. El Concepto de Gramática.

Informalmente una gramática no es más que un mecanismo para generar las cadenas que pertenecen a un lenguaje. Este mecanismo define un conjunto finito de reglas que, aplicadas a partir de un único símbolo inicial, son capaces de generar todas sus cadenas.

Por ejemplo, sea la gramática

$$S \rightarrow 0S \mid 0A$$
$$A \rightarrow 1A \mid 1$$

En el ejemplo se observan dos tipos de símbolos; por un lado, S *y* A *que se denominan* símbolos auxiliares *y, por otro lado,* 0 *y* 1 *que se denominan* símbolos terminales*. El objetivo es obtener una cadena aplicando* producciones*, que son las reglas que nos indican cómo substituir los símbolos auxiliares. Las cadenas generadas por una gramática están formadas exclusivamente por símbolos terminales.*

Así, para obtener una cadena se parte de S *(símbolo inicial,* start*) y se substituye por una de sus dos producciones. No hay ningún tipo de preferencias entre ellas, se puede elegir cualquiera:*

$$S \Rightarrow 0S$$

y, a continuación, hay que proceder del mismo modo y substituir la nueva aparición de S *por una cualquiera de sus dos producciones:*

$$S \Rightarrow 0\mathbf{S} \Rightarrow 00\mathbf{S} \Rightarrow 000\mathbf{S}$$

En este punto, se podría optar por la segunda producción,

$$S \Rightarrow 0\mathbf{S} \Rightarrow 00\mathbf{S} \Rightarrow 000\mathbf{S} \Rightarrow 0000\mathbf{A}$$

y, por lo tanto, ahora habría que continuar substituyendo el auxiliar A *por una de sus dos producciones,*

$$S \Rightarrow 0\mathbf{S} \Rightarrow 00\mathbf{S} \Rightarrow 000\mathbf{S} \Rightarrow 0000\mathbf{A} \Rightarrow 00001\mathbf{A} \Rightarrow 000011\mathbf{A} \Rightarrow 0000111$$

La cadena 0000111 *está formada exclusivamente por terminales, luego se puede concluir que es una cadena del lenguaje generado por la gramática. A continuación, se formalizan estos conceptos.*

Definición 1.1 (Gramática) *Una gramática es una cuádrupla*

$$G = \langle \Sigma_A, \Sigma_T, P, S \rangle$$

donde:

$\mathbf{\Sigma_A}$ *es el denominado alfabeto de símbolos auxiliares,*

$\mathbf{\Sigma_T}$ *es el denominado alfabeto de símbolos terminales,*

P *es el denominado conjunto de producciones. Es un conjunto de pares ordenados (α, β) finito y no vacío,*

$$P = \{(\alpha_1, \beta_1), (\alpha_2, \beta_2), \ldots, (\alpha_n, \beta_n)\},$$

con $\alpha_i \in (\Sigma_A \cup \Sigma_T)^+ \wedge \beta_i \in (\Sigma_A \cup \Sigma_T)^*, i = 1, 2, \ldots, n.$

S *es un símbolo especial denominado símbolo inicial de la gramática,* $S \in \Sigma_A$.

Se denota por Σ *al conjunto* $\Sigma_A \cup \Sigma_T$ *y se exige que* $\Sigma_A \cap \Sigma_T = \emptyset$.

Las producciones se definen formalmente como pares (α, β) en los que a α se le denomina *antecedente* de la producción y a β su *consecuente*. La notación habitual de una producción (α, β) es la expresión $\alpha \rightarrow \beta$, que ha sido la utilizada en el ejemplo anterior.

Definición 1.2 *Dos cadenas,* θ_1 *y* θ_2 *se encuentran en* relación de derivación directa *dentro de la gramática G, denotado por* $\theta_1 \Rightarrow_G \theta_2$*, cuando*

$$\theta_1 = \gamma\alpha\delta \wedge \theta_2 = \gamma\beta\delta \wedge (\alpha \rightarrow \beta) \in P$$

donde α, β, γ *y* $\delta \in \Sigma^*$.

Es decir, esta relación expresa cómo conseguir de forma directa una cadena θ_2 a partir de otra cadena θ_1: realizando la sustitución de la cadena α por la cadena β (aplicando una producción de la gramática G) dejando el resto de cadenas sin modificar.

Por ejemplo, en la gramática anterior las cadenas $000S$ *y* $0000A$ *están en relación de derivación directa ya que se puede obtener* $0000A$ *de* $000S$ *en un único paso.*

Definición 1.3 *Dos cadenas,* θ_1 *y* θ_2 *se encuentran en* relación de derivación en n pasos *dentro de la gramática G, cuando existen n cadenas* $\mu_1, \mu_2, \ldots, \mu_n \in \Sigma^*$*, tales que* $\theta_1 \Rightarrow_G \mu_1 \Rightarrow_G \mu_2 \Rightarrow_G \ldots \Rightarrow_G \mu_n \Rightarrow_G \theta_2$.

Esta operación se puede generalizar más aún si no se especifica el número de pasos. En este caso se emplea la notación $\theta_1 \overset{*}{\Rightarrow}_G \theta_2$ cuando $n \geq 0$ (es decir, se utilizan cero, una o más derivaciones directas) y se emplea la notación $\theta_1 \overset{+}{\Rightarrow}_G \theta_2$ cuando $n > 0$ (es decir, cuando se utilizan una o más derivaciones directas).

Definición 1.4 *Dada una cadena $\theta \in \Sigma^*$, se dice que θ es una* forma sentencial *cuando $S \overset{*}{\Rightarrow}_G \theta$. Una cadena θ se denomina* sentencia *o* palabra *cuando θ es una forma sentencial y todos los símbolos de θ son símbolos terminales.*

En el ejemplo anterior, las cadenas $000S$, $0000A$ y $00001A$ (entre otras) son formas sentenciales. Sin embargo, 0000111 ya es una palabra, puesto que sólo está formada por terminales.

Definición 1.5 *Se define el* lenguaje asociado a una gramática *como el conjunto de palabras*

$$L(G) = \{x \in \Sigma_T^* \mid S \overset{*}{\Rightarrow}_G x\}.$$

En adelante, y siempre que no exista posibilidad de confusión, se suprimirá la identificación de la gramática como subíndice en las flechas que denotan las relaciones de derivación, tal y como ya se hizo en el desarrollo del ejemplo.

¿Cuáles son los lenguajes generados por las siguientes gramáticas?

(a) $S \to 0S \mid 0A$
$A \to 1A \mid 1$

(b) $S \to 0S \mid 1A$
$A \to 1A \mid \lambda$

(c) $S \to aS \mid bA \mid \lambda$
$A \to bA \mid \lambda$

(d) $S \to aS \mid A$
$A \to bA \mid \lambda$

(e) $S \to 0S1 \mid 01$

(f) $S \to 0S1 \mid \lambda$

(g) $S \to 000S111 \mid 000111$

(h) $S \to 000S111$
$0S1 \to 01$

(i) $S \to X \mid Y$
$Y \to aY \mid A$
$A \to bA \mid \lambda$
$X \to 0B1Y$
$B \to 0B1 \mid \lambda$

Para determinar cuál es el lenguaje generado por una gramática se tiene que calcular cuáles son las cadenas que pueden derivarse a partir de su símbolo inicial. Por ejemplo, en la gramática del apartado a) se puede proceder de la siguiente forma:

$S \Rightarrow 0\mathbf{S} \Rightarrow 00\mathbf{S} \Rightarrow 000\mathbf{S} \Rightarrow \ldots$*, es decir, se puede pensar en aplicar primero n veces la producción $S \to 0S$, obteniéndose*

$S \overset{n}{\Rightarrow} 0^n S \Rightarrow \ldots$, tras lo cual se aplica la producción $S \to 0A$, siendo el resultado
$S \overset{n}{\Rightarrow} 0^n \mathbf{S} \Rightarrow 0^n 0A \Rightarrow \ldots$, tras lo cual se puede aplicar varias veces la producción $A \to 1A$,
$S \overset{n}{\Rightarrow} 0^n S \Rightarrow 0^n 0\mathbf{A} \Rightarrow 0^n 01\mathbf{A} \Rightarrow 0^n 011\mathbf{A} \Rightarrow 0^n 0111\mathbf{A} \ldots$
Si se aplica m veces $A \to 1A$ y ya, por último, $A \to 1$, se llega a
$S \overset{n}{\Rightarrow} 0^n S \Rightarrow 0^n 0A \overset{m}{\Rightarrow} 0^n 01^m \mathbf{A} \Rightarrow 0^n 01^m 1$.

Por lo tanto el lenguaje generado por la gramática del apartado a) es el siguiente:

$$L(G_{(a)}) = \{x \in \{0,1\}^* \mid x = 0^n 1^m, \text{ con } n, m \geq 1\}$$

Se propone como ejercicio la obtención de los demás lenguajes.

Definición 1.6 *Dos gramáticas se denominan* equivalentes *cuando ambas generan el mismo lenguaje. Es decir, dadas dos gramáticas G_1 y G_2 entonces $G_1 \equiv G_2 \Leftrightarrow L(G_1) = L(G_2)$.*

Además, una gramática sólo puede generar un lenguaje mientras que un lenguaje puede ser generado por varias gramáticas (de hecho, hay infinitas gramáticas que son capaces de generar un mismo lenguaje).

Se propone como ejercicio determinar cuáles de las gramáticas anteriores son equivalentes.

1.5. Clasificación Jerárquica de las Gramáticas.

Uno de los posibles criterios para clasificar las gramáticas es hacerlo de acuerdo al formato utilizado para describir al conjunto de producciones. Esta clasificación fue establecida por el lingüista Noam Chomsky.

Gramáticas de tipo 3: También denominadas *regulares*. Las hay de dos tipos, según sean el formato de sus producciones:

- *Lineales a la derecha*, con producciones

$$A \to aB$$
$$A \to a$$

- *Lineales a la izquierda*, con producciones

$$A \to Ba$$
$$A \to a$$

En ambos casos, $A, B \in \Sigma_A,\ a \in \Sigma_T^*$.

Hay que destacar que no se permite mezclar en una misma gramática de tipo 3 producciones lineales a la derecha con producciones lineales a la izquierda.

Gramáticas de tipo 2: También denominadas *de contexto libre*. Son las gramáticas caracterizadas porque su conjunto de producciones presenta el siguiente formato:

$$A \to \beta \; donde \; A \in \Sigma_A,\ \beta \in \Sigma^*.$$

En las gramáticas de tipo 2 el símbolo A siempre se puede sustituir por la cadena β *independientemente* del contexto en el que aparezca.

Gramáticas de tipo 1: También se les conoce como *gramáticas sensibles al contexto*. Son aquellas gramáticas cuyas producciones presentan el siguiente formato:

$$\alpha_1 A \alpha_2 \to \alpha_1 \beta \alpha_2 \; donde \; A \in \Sigma_A,\ \alpha_1, \alpha_2 \in \Sigma^*,\ \beta \in \Sigma^+.$$

Es decir, sólo se permite la sustitución del símbolo A por la cadena β cuando el símbolo A aparezca en el *contexto indicado* por la producción, es decir, con la cadena α_1 a la izquierda de A y por la cadena α_2 a su derecha. Nótese la diferencia con las gramáticas de tipo 2.

Como excepción a este formato se permite que $S \to \lambda$, pero si esto sucede entonces el símbolo inicial S no puede aparecer en el consecuente de ninguna otra producción de la gramática.

Como característica adicional de este tipo de gramáticas se cumple que en cualquier secuencia de derivaciones, por ejemplo $S \overset{*}{\Rightarrow} \theta_1 \overset{*}{\Rightarrow} \theta_2 \overset{*}{\Rightarrow} \theta_3$ siempre se verifica que $\mid \theta_{i+1} \mid \geq \mid \theta_i \mid$. Por este motivo a estas gramáticas también se las denomina *gramáticas crecientes*.

Gramáticas de tipo 0: Son aquellas gramáticas caracterizadas porque en sus producciones no se establece ningún tipo de restricción respecto a su formato.

Se dice que un lenguaje es de tipo *i* (*i* = 3, 2, 1, 0) si, y sólo si, la gramática de índice más alto que puede generarlo es de tipo *i*.

Además, se verifica que si se denomina L_i a la clase de lenguajes generados por gramáticas de tipo *i* entonces se observa que, de la definición de la jerarquía de gramáticas de Chomsky, se deriva el siguiente enunciado

$$L_3 \subseteq L_2 \subseteq L_1 \subseteq L_0.$$

Aún se puede afinar más y, en los temas sucesivos, se demostrará que estas inclusiones no son propias, es decir, existen lenguajes pertenecientes a la clase L_i que no pertenecen a la clase L_{i+1} (i = 0,1,2).

Por lo tanto, se cumple el siguiente enunciado

$$L_3 \subset L_2 \subset L_1 \subset L_0,$$

lo que se podría representar de forma gráfica mediante la siguiente figura:

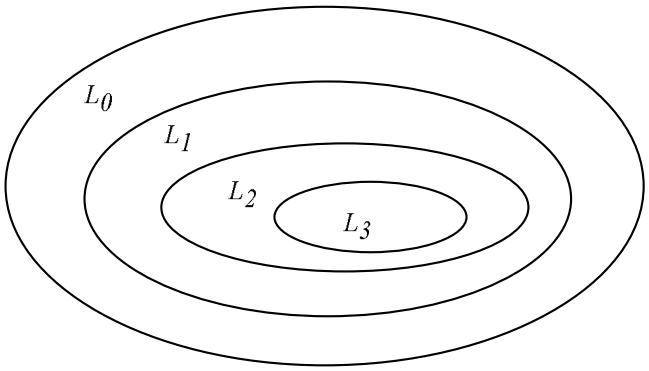


Figura 1.1: Relaciones entre las clases de lenguajes derivadas de la jerarquía de Chomsky.

La jerarquía establecida por Chomsky, además de resultar elegante, permite vertebrar la Teoría de la Computación clasificando las clases de lenguajes en función del número de recursos computacionales necesarios para reconocerlos.

Uno de los objetivos del estudio de los lenguajes formales es asimilar la correspondencia entre lenguajes formales y problemas computacionales, esto es, cualquier lenguaje formal se puede asociar a un problema computacional. Desde este punto de vista, reconocer las cadenas de un lenguaje es equivalente a solucionar un problema. Por lo tanto, la jerarquía de Chomsky realmente lo que permite es clasificar por orden creciente el poder computacional necesario para poder resolver distintas clases de problemas. Y uno de los resultados más espectaculares de esta disciplina lo supone el hecho de que, hoy por hoy, existen problemas que no se pueden resolver aún cuando se utilice el modelo con mayor poder computacional, el correspondiente a los lenguajes de Tipo 0, que modela la capacidad computacional de cualquier computador existente. Esto es, hay problemas *irresolubles*, hay límites para el concepto de computación tal y como lo entendemos.

También resulta interesante comentar cómo ha ido tomando cuerpo esta teoría ya que, en principio, los principales resultados surgieron de campos muy diversos. Del campo de la Lógica Matemática surgió el modelo de Máquina de Turing, que ocupa el lugar más alto de la jerarquía, ya que es el reconocedor de los lenguajes de Tipo 0. Del campo de la

Automática Industrial, surgieron las Autómatas de Control Finito, que ocupan a su vez el lugar más bajo, al corresponderse con los lenguajes de Tipo 3. De avances propios de la Informática, como el desarrollo de los procesadores de lenguaje, surgieron los reconocedores de lenguajes de Tipo 1.

El propio Noam Chomsky no es informático, ni matemático, ni ingeniero. Como ya se ha dicho es un lingüista cuya principal aportación fue el estudio de las gramáticas y su clasificación como parte de la demostración de su teoría *generativa* de las gramáticas [1].

Sin embargo, todos estos elementos tan diversos en su origen, estructuran el modelo existente de la Teoría de la Computación. Los capítulos de este libro se han estructurado de acuerdo a esta jerarquía, estudiando los niveles más trascendentales para nuestros objetivos; así, en la asignatura se verán los diferentes modelos abstractos en tres grandes grupos, correspondientes a los niveles 3, 2 y 0.

[1] La teoría de Chomsky se basa en la existencia en el ser humano de estructuras gramaticales propias, esto es, el ser humano nace con la capacidad de aprender un lenguaje porque los esquemas gramáticales forman parte de su cerebro, de tal forma que, a medida que su experiencia le permite mejorar ese conocimiento, adquiere completamente la capacidad de comunicarse en un idioma. Esta teoría se contrapone a las que asumen que el cerebro humano es una "tabla en blanco" al nacer y que sólo mediante su interacción con el entorno adquiere el lenguaje.

Capítulo 2

Autómatas Finitos y Expresiones Regulares

Índice General

2.1. Autómatas Finitos Deterministas. 15
2.2. Autómatas Finitos No Deterministas. 19
2.3. Autómatas Finitos con Movimientos Libres. 24
2.4. Expresiones Regulares. 29
2.5. Otros Tipos de Autómatas Finitos. 36
2.5.1. Autómatas Finitos Bidireccionales 36
2.5.2. Autómatas Finitos con Función de Salida. 38
2.6. Aplicaciones. 39
2.6.1. Aplicación como ayuda al análisis léxico en un compilador. . . 40
2.6.2. Especificación de parámetros de entrada. 40
2.6.3. Ayuda al diseño de circuitos. 41

2.1. Autómatas Finitos Deterministas.

En esta asignatura se estudiarán distintos tipos de máquinas abstractas con diferente poder computacional. Cada uno de estos tipos de máquinas permiten reconocer lenguajes de distintos tipos y su poder computacional está asociado a cuál es el tipo de lenguajes que pueden reconocer.

El primer tipo de máquinas abstractas que se va a definir son los Autómatas Finitos, que permiten reconocer cadenas pertenecientes a Lenguajes Regulares.

Son los autómatas con menor poder computacional y para reconocer los lenguajes sólo disponen de una cinta de entrada (en la que se escribe una cadena), de un cabezal lector (que lee símbolos de uno en uno y de izquierda a derecha) y de un conjunto finito de reglas

que especifica cómo actuar ante cada símbolo leído de la cinta de entrada. Gráficamente se puede representar por medio de la siguiente figura:

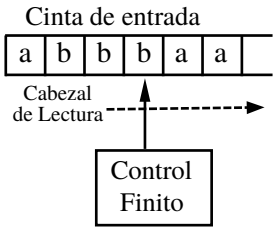


Figura 2.1: Modelo físico de un Autómata Finito.

Definición 2.1 *Un autómata finito determinista, AFD, es una quíntupla* $A = \langle \Sigma, Q, f, q_0, F \rangle$ *donde:*

Σ *es el alfabeto de entrada,*

Q *es un conjunto finito y no vacío de estados,*

f *es una función total definida como*

$$f : Q \times \Sigma \rightarrow Q,$$

q_0 *es el denominado estado inicial,* $q_0 \in Q$*,*

F *es el conjunto de estados denominados finales,* $F \subseteq Q$*.*

La forma más generalizada de representación de un AFD es mediante un grafo dirigido. En este grafo los nodos se corresponden con los estados de Q y los arcos representan valores de la función de transición *f*, de forma que si $f(q_s, a) = q_t$ entonces desde el nodo q_s parte un arco etiquetado con el símbolo a hacia el nodo q_t. Los estados de F se representan mediante nodos con doble círculo y el estado inicial se suele representar con una flecha de entrada al nodo que lo representa.

Ejemplo:

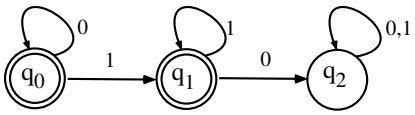
Sea A el siguiente AFD

$$\begin{array}{ll} Q = \{q_0, q_1, q_2\} & F = \{q_0, q_1\} \\ \Sigma = \{0, 1\} & q_0 \in Q,\ es\ el\ estado\ inicial \end{array}$$

en el que la función de transición f se define como:

f	0	1
q_0	q_0	q_1
q_1	q_2	q_1
q_2	q_2	q_2

Este AFD se puede representar mediante el siguiente grafo:



Para poder determinar cuál es el conjunto de cadenas aceptadas por un autómata se necesita estudiar cuál es su comportamiento ante cadenas de símbolos. Se define f' como la extensión de la función de transición que se aplica a cadenas de símbolos del alfabeto,

$$f' : Q \times \Sigma^* \to Q$$
$$a) f'(q, \lambda) = q, \forall\, q \in Q$$
$$b) f'(q, xa) = f(f'(q, x), a), x \in \Sigma^*, a \in \Sigma$$

De esta forma, en el autómata finito del ejemplo anterior

$$f'(q_0, 011) = f(f(f(f'(q_0, \lambda), 0), 1), 1) = q_1.$$

Para simplificar la notación se utiliza el mismo símbolo de función f para referirse tanto a f como a f'.

Definición 2.2 *Se define el lenguaje aceptado por un AFD $A = \langle \Sigma, Q, f, q_0, F \rangle$ como el conjunto de cadenas tales que después de ser analizadas el AFD se encuentra en un estado final; es decir,*

$$L(A) = \{x \in \Sigma^* \mid f(q_0, x) \in F\}$$

Ejemplo:

¿Qué lenguajes reconocen los AFD de la figura 2.2?

EL AFD del caso (a) reconoce las cadenas que tienen el formato $(10)^n$ con $n \geq 1$, es decir $L_a = \{10, 1010, 101010, 10101010, \ldots\}$.

EL AFD del caso (b) acepta las cadenas que presentan el formato bca^n y las cadenas del formato ac^m, con $n, m \geq 0$.

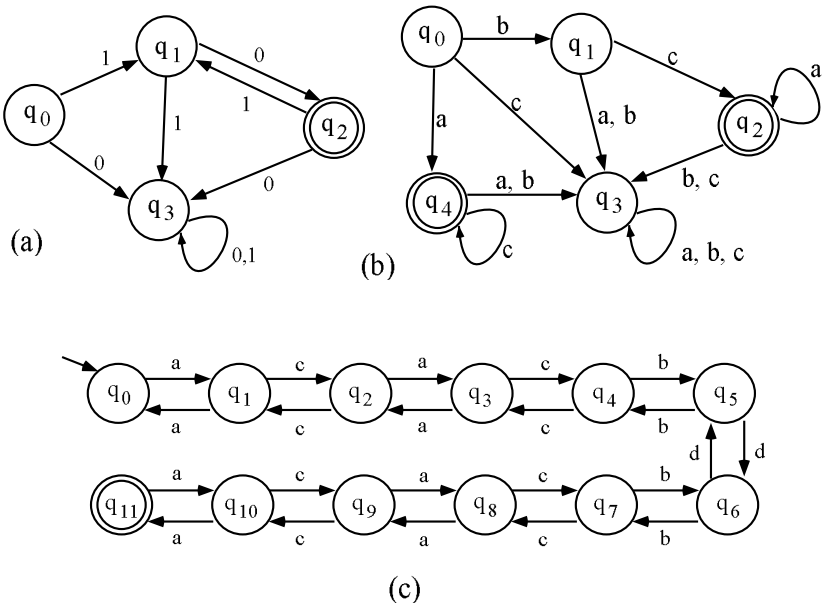


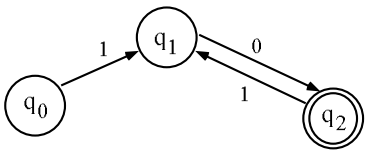
Figura 2.2: ¿Qué lenguajes reconocen estos AFD?

El AFD del caso (c) resulta más difícil de describir de forma general. Algunas de las cadenas que acepta son

$$L_c = \{acacbdbcaca, aaacaaacccbdddddbcaaaca, \ldots\}.$$

Aquellos estados que no sean finales y que, además, una vez que son alcanzados el AFD permanece en ellos, sean cuales sean los símbolos que se lean de la entrada, se denominan *estados sumideros*. Como convenio no suelen mostrarse en el grafo para así poder minimizar el número de arcos del grafo y simplificar su representación gráfica.

Volviendo al ejemplo anterior, en el AFD (a), el estado q_3 es un estado sumidero; por lo tanto, este AFD puede representarse de la forma:



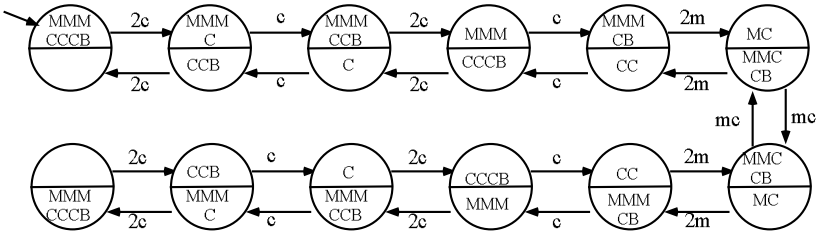
En el AFD del caso (b), el estado q_3 también es un sumidero y el nodo correspondiente y los arcos que se dirigen a él podrían eliminarse.

Para finalizar, se presenta como ejemplo un conocido problema de planificación y cómo resolverlo utilizando un AFD:

Tres caníbales y tres misioneros han de cruzar un río. Sólo disponen de una barca en la que caben como máximo dos personas. ¿Cómo han de cruzar el río de forma que nunca haya más caníbales que misioneros en cualquiera de las orillas?

Para solucionar el problema se identifica cada una de las posibles situaciones viables en las que se pueden encontrar caníbales y misioneros como un estado. En cada estado se representará los caníbales y misioneros que hay en cada orilla, así como la posición de la barca. Así, para indicar que hay 3 misioneros y 1 caníbal en una orilla (`MMMC`*) y 2 caníbales y la barca en la otra orilla (*`CCB`*), se emplea la siguiente notación:*

Para indicar quién ocupa la barca (transiciones entre estados), se emplea la siguiente notación: c *para caníbal,* 2m *dos misioneros,* 2c *dos caníbales y* mc *un misionero y un caníbal. Con esta notación, el conjunto de infinitas soluciones que tiene este problema se puede representar como cualquiera de los caminos que conducen de la situación inicial a la final en el siguiente AFD:*



El AFD diseñado para el problema de los misioneros y los caníbales se puede relacionar con el mostrado en el apartado (c) del ejemplo anterior; el lenguaje L_c se puede asociar a las soluciones de este problema bajo la interpretación *a* por *2c*, *c* por *c*, *b* por *2m* y *d* por *mc*. Por lo tanto, realizando una interpretación adecuada a los símbolos y los estados del AFD se ha podido establecer el lenguaje aceptado por este AFD. En general, el problema de encontrar una interpretación adecuada no es sencillo de solucionar y plantea una serie de cuestiones de orden filosófico/matemático que queda fuera del alcance de este tema.

2.2. Autómatas Finitos No Deterministas.

Un *Autómata Finito No Determinista*, que se denotará como AFN, se caracteriza porque en un estado puede haber más de una transición posible para un mismo símbolo de entrada. Es decir, $\mid f(q,a) \mid \geq 1$ para algún $q \in Q$ y para algún símbolo $a \in \Sigma$.

Definición 2.3 *Un AFN es una quíntupla* $A = \langle \Sigma, Q, f, q_0, F \rangle$ *donde:*

Σ *es el alfabeto de entrada,*

Q *es un conjunto finito y no vacío de estados,*

f *es una función definida de la siguiente forma*

$$f : Q \times \Sigma \to 2^Q,$$

q_0 *es el denominado estado inicial,* $q_0 \in Q$,

F *es el conjunto de estados denominados finales,* $F \subseteq Q$.

Al igual que en el caso de los AFD, para poder determinar cuál es el lenguaje reconocido por este tipo de autómata es preciso conocer cuál es su comportamiento ante conjuntos de estados y ante cadenas de símbolos. Para ello se define una extensión de la función de transición del autómata.

Definición 2.4 *Sea* $P \subseteq Q$, *entonces* $f' : 2^Q \times \Sigma \to 2^Q$ *se define como*

$$f'(P, a) = \bigcup_{q \in P} f(q, a).$$

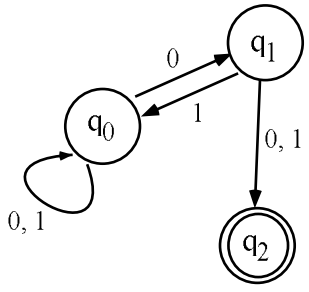
Sea $P \subseteq Q$, *entonces se define* $f'' : 2^Q \times \Sigma^* \to 2^Q$ *mediante la regla recursiva:*

$$1) f''(P, \lambda) = P,$$
$$2) f''(P, ax) = f''(f'(P, a), x),\ x \in \Sigma^*, a \in \Sigma.$$

Para simplificar la notación, en adelante se utilizará el mismo símbolo de función, *f*, para referirse tanto a f como a f' y a f''.

Ejemplo:

Sea A el siguiente AFN



entonces $f(q_0, 01) = f(f(q_0, 0), 1) = f(\{q_0, q_1\}, 1) = \{q_0, q_2\}$.

Una vez descritas estas operaciones, ya se está en condiciones de poder describir cuál es el conjunto de cadenas que este tipo de autómata es capaz de reconocer.

Definición 2.5 *Se define el lenguaje aceptado por un AFN $A = \langle \Sigma, Q, f, q_0, F \rangle$ como el conjunto de cadenas tales que después de ser analizadas, el AFN ha podido alcanzar al menos un estado final, es decir,*

$$L(A) = \{x \in \Sigma^* \mid f(q_0, x) \cap F \neq \emptyset\}.$$

En el autómata del ejemplo anterior, L(A) es el conjunto de cadenas cuyo penúltimo símbolo leído de izquierda a derecha es un 0.

Un sencillo análisis de este tipo de autómata permite establecer que si en un autómata de este tipo no hay ninguna transición tal que para un mismo símbolo se disponga de más de un estado destino diferente, entonces este autómata es determinista. Es decir, los AFD se pueden ver como casos particulares de los AFN. Entonces, si se denota por *L(AFD)* al conjunto de todos los lenguajes que se pueden reconocer con autómatas finitos deterministas y se denota por *L(AFN)* al conjunto de todos los lenguajes que se pueden reconocer con autómatas finitos no deterministas, resulta que $L(AFD) \subseteq L(AFN)$.

La cuestión que se plantea es si la inclusión contraria también se cumple, es decir, si para cualquier AFN existe un AFD que reconozca exactamente el mismo lenguaje que el AFN. El siguiente teorema establece que así es, lo que permite concluir que

$$L(AFD) = L(AFN).$$

Teorema 2.1 *Sea L un lenguaje aceptado por un AFN, entonces existe un AFD que también acepta L.*

Demostración:

Sea $A = \langle \Sigma, Q, f, q_0, F \rangle$ un AFN tal que L(A) = L, entonces se define el siguiente AFD $A' = \langle \Sigma, Q', f', q_0', F' \rangle$ donde

- Q′ es isomorfo al conjunto 2^Q; es decir, el conjunto de estados $\{q_i, \ldots, q_j\} \in 2^Q$ tiene asociado un estado denominado $[q_i, \ldots, q_j]$ en Q′,
- $q_0' = [q_0']$,
- $F' = \{[q_i, \ldots, q_j] \in Q' \mid \{q_i, \ldots, q_j\} \cap F \neq \emptyset\}$, y
- la función de transición f' es

$$f'([q_i, \ldots, q_j], a) = [p_r, \ldots p_s] \Leftrightarrow f(\{q_i, .., q_j\}, a) = \{p_r, \ldots p_s\}.$$

Una vez definido este AFD A′, se va a demostrar que *L(A) = L(A′)*. Para ello se va a demostrar que el comportamiento de ambos autómatas es similar cuando analizan la misma cadena de entrada *x*, es decir,

$$\forall\, x \in \Sigma^* :\ (f(q_0, x) = P) \Leftrightarrow (f'(q'_0, x) = [P]).$$

El método para demostrar este enunciado consiste en aplicar inducción respecto a la longitud de la cadena x.

Paso Base: $\mid x \mid = 0$, es decir, $x = \lambda$. Para esta cadena, por definición de f se cumple $f(q_0, \lambda) = \{q_0\}$, y, por definición de f', también se cumple que $f'(q'_0, \lambda) = q'_0 = [q_0]$. Por lo tanto, se cumple el enunciado.

Hipótesis de Inducción: $\forall\, x \in \Sigma^*$ tal que $\mid x \mid \leq n$ se cumple $f(q_0, x) = P \Leftrightarrow f'(q'_0, x) = [P]$.

Paso de Inducción: Sea $y = xa$, tal que $x \in \Sigma^* \wedge \mid x \mid \leq n \wedge a \in \Sigma$.

Entonces, $f(q_0, y) = f(q_0, xa) = f(f(q_0, x), a) = f(P, a)$. Como por hipótesis de inducción, $[P] = f'(q'_0, x)$, en el AFD A′ se cumple que $f'([P], a) = f'(f'(q'_0, x), a) = f'(q'_0, xa) = f'(q'_0, y)$.

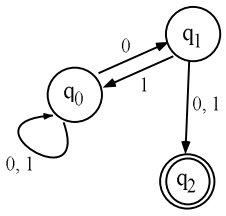
Por lo tanto, ambos autómatas presentan un comportamiento semejante ante las mismas cadenas de entrada. Para completar la demostración basta con observar que reconocen exactamente el mismo conjunto de cadenas:

- Si $x \in L(A) \Rightarrow f(q_0, x) = P, P \cap F \neq \emptyset \Rightarrow f'(q'_0, x) = [P] \in F'$.
- Si $x \in L(A') \Rightarrow f'(q'_0, x) = [P], [P] \in F' \Rightarrow f(q_0, x) = P, P \cap F \neq \emptyset$.

c.q.d.

Ejemplo:

Calcular un AFD que reconozca el mismo lenguaje que el siguiente AFN,



La demostración del teorema anterior proporciona un método para calcular tal dicho AFD. En esta demostración dado un AFN, que en este caso es

$$A = \langle \{0, 1\}, \{q_0, q_1, q_2\}, f, q_0, \{q_2\} \rangle$$

se construye el AFD $A' = \langle \{0,1\}, Q', f', q'_0, F' \rangle$ *donde*

$$Q' = \{[q_0], [q_1], [q_2], [q_0, q_1], [q_0, q_2], [q_1, q_2], [q_0, q_1, q_2], [\emptyset]\},$$
$$q'_0 = [q_0],$$
$$F' = \{[q_2], [q_0, q_2], [q_1, q_2], [q_0, q_1, q_2]\}$$

y tal que la relación entre f y f' *viene determinada por la siguiente condición:*

$$f(P_1, a) = P_2 \Leftrightarrow f'([P_1], a) = [P_2].$$

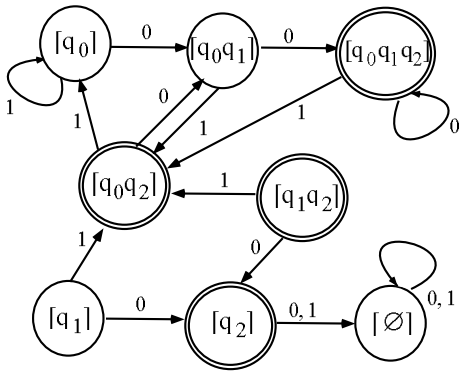
Por lo tanto, las transiciones del AFD se construyen de la siguiente forma:

$$f'([q_0], 0) = [q_0, q_1],\ ya\ que\ f(q_0, 0) = \{q_0, q_1\}$$
$$f'([q_0], 1) = [q_0],\ ya\ que\ f(q_0, 1) = \{q_0\}$$
$$f'([q_1], 0) = [q_2],\ ya\ que\ f(q_1, 0) = \{q_2\}$$
$$f'([q_1], 1) = [q_0, q_2],\ ya\ que\ f(q_1, 1) = \{q_0, q_2\}$$

y, de forma similar, se completaría la definición de f' *,*

$$f'([q_2], 0) = [\emptyset] \qquad f'([q_2], 1) = [\emptyset]$$
$$f'([q_0, q_1], 0) = [q_0, q_1, q_2] \qquad f'([q_0, q_1], 1) = [q_0, q_2]$$
$$f'([q_0, q_2], 0) = [q_0, q_1] \qquad f'([q_0, q_2], 1) = [q_0]$$
$$f'([q_1, q_2], 0) = [q_2] \qquad f'([q_1, q_2], 1) = [q_0, q_2]$$
$$f'([q_0, q_1, q_2], 0) = [q_0, q_1, q_2] \qquad f'([q_0, q_1, q_2], 1) = [q_0, q_2]$$
$$f'([\emptyset], 0) = [\emptyset] \qquad f'([\emptyset], 1) = [\emptyset]$$

Por lo tanto, el AFD equivalente es el siguiente:



Este método es válido pero, además de costoso, produce un gran número de estados inalcanzables desde el estado inicial (en el ejemplo anterior, los estados $[q_1]$, $[q_2]$, $[q_1, q_2]$ y $[\emptyset]$ son inalcanzables) y, por lo tanto, innecesarios.

La operación de calcular un AFD a partir de un AFN es muy común y, por ello, se han desarrollado métodos alternativos. El más sencillo y eficiente es el siguiente:

Mínimo número de pasos para construir un AFD a partir de un AFN

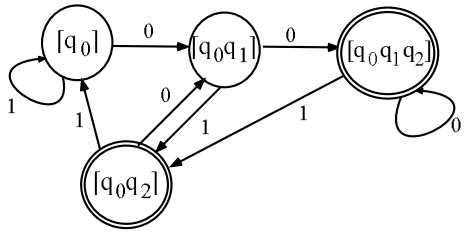
1. Se construye una tabla donde cada columna está etiquetada con un símbolo del alfabeto de entrada y cada fila se etiqueta con un conjunto de estados.
2. La primera fila se etiqueta con $\{q_0\}$, y en cada entrada de la tabla (q_0, a_i) se almacena $f(q_0, a_i) = \{p_1, \ldots, p_n\} = P$.
3. Se etiqueta cada fila con cada uno de los conjuntos P que no tengan asociada una fila en la tabla (es decir, con cada uno de los conjuntos P que aparezcan por primera vez en la tabla) y se completa cada una de estas filas con el correspondiente $f(P, a_i)$.
4. Se realiza el paso (3) hasta que no haya en la tabla conjuntos P sin filas asociadas.
5. Se asocia a cada conjunto P que aparezca en la tabla un estado en el nuevo AFD y aquellos que tengan entre sus componentes algún estado final del AFN se considerarán estados finales en el AFD.

Al aplicar este método al autómata anterior se obtiene el siguiente AFD

f	0	1
$\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_0, q_1\}$	$\{q_0, q_1, q_2\}$	$\{q_0, q_2\}$
$\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2\}$	$\{q_0, q_2\}$
$\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$

Al conjunto $\{q_0\}$ *se le denomina* $[q_0]$*, al* $\{q_0, q_1\}$ *se le denomina* $[q_0, q_1]$*, al* $\{q_0, q_1, q_2\}$ *se le denomina* $[q_0, q_1, q_2]$ *y a* $\{q_0, q_2\}$ *se le denomina* $[q_0, q_2]$.

Como los conjuntos $\{q_0, q_1, q_2\}$ *y* $\{q_0, q_2\}$ *contienen estados finales del AFN entonces los estados* $[q_0, q_1, q_2]$ *y* $[q_0, q_2]$ *serán estados finales en el AFD.*



2.3. Autómatas Finitos con Movimientos Libres.

Un Autómata Finito con Movimientos Libres, que se denotará habitualmente como AFλ, se caracteriza porque cada transición está etiquetada con uno o más símbolos del alfabeto de entrada o bien con la cadena vacía.

Definición 2.6 *Un autómata finito con movimientos libres, AFλ, es una quíntupla* $A = \langle \Sigma, Q, f, q_0, F \rangle$ *donde:*

Σ *es el alfabeto de entrada,*

Q *es un conjunto finito y no vacío de estados,*

f *es una función definida de la siguiente forma*

$$f : Q \times (\Sigma \cup \{\lambda\}) \rightarrow 2^Q,$$

q_0 *es el denominado estado inicial,* $q_0 \in Q$,

F *es el conjunto de estados denominados finales,* $F \subseteq Q$.

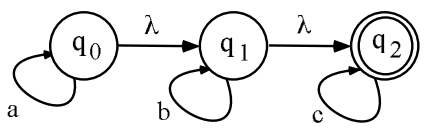
Para poder determinar cuál es el comportamiento de este tipo de autómata ante una cadena de entrada se necesita establecer una serie de definiciones previas.

Definición 2.7 *Se denomina* $\lambda\_clausura(q),\ \forall\ q\ \in\ Q$, *al conjunto de estados accesibles desde el estado* q *utilizando sólo transiciones* λ. *Además, se cumple que* $q \in \lambda\_clausura(q)$.

Definición 2.8 *Se denomina* $\lambda\_CLAUSURA(P)$, *siendo P un conjunto de estados,* $P \in 2^Q$, *al conjunto de estados formado por todas las* λ*_clausuras de todos los estados que forman P,*

$$\lambda\_CLAUSURA(P) = \bigcup_{p \in P} \lambda\_clausura(p).$$

Ejemplo:



$$\lambda\_clausura(q_0) = \{q_0, q_1, q_2\}$$
$$\lambda\_clausura(q_1) = \{q_1, q_2\}$$
$$\lambda\_clausura(q_2) = \{q_2\}$$

Para simplificar la notación se utiliza el mismo símbolo de función λ_clausura para referirse tanto a $\lambda\_clausura$ como a $\lambda\_CLAUSURA$.

Con estas definiciones ya se está en condiciones de poder determinar cuál es el comportamiento de un AFλ ante una cadena de entrada.

Definición 2.9 *Se define* $\hat{f} : Q \times \Sigma^* \rightarrow 2^Q$ *de la forma siguiente:*

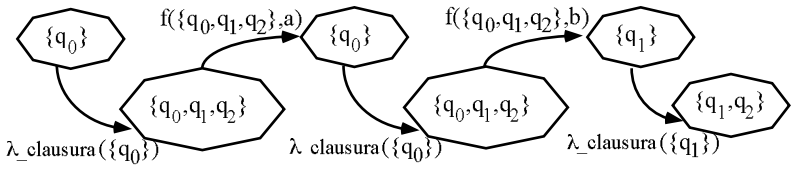
$1) \hat{f}(q, \lambda) = \lambda\_clausura(q)$
$2) \forall\ x \in \Sigma^*,\ \forall\ a \in \Sigma,\ \forall\ q \in Q :$
$\quad \hat{f}(q, xa) = \lambda\_clausura(P) \mid P = \{p \in Q \mid\ \exists\ r \in \hat{f}(q, x) \wedge p \in f(r, a)\}$

Ejemplo:

Dado el AFλ del ejemplo anterior, ¿qué resultado se obtiene al calcular $\hat{f}(q_0, ab)$*?*

$\hat{f}(q_0, ab) = \lambda\_clausura(P_1)$
$\quad P_1 = \{p \in Q \mid \exists\ r \in \hat{f}(q_0, a) \wedge p \in f(r, b)\}$
$\quad\quad \hat{f}(q_0, a) = \lambda\_clausura(P_2)$
$\quad\quad\quad P_2 = \{p \in Q \mid \exists\ r \in \hat{f}(q_0, \lambda) \wedge p \in f(r, a)\}$
$\quad\quad\quad\quad \hat{f}(q_0, \lambda) = \lambda\_clausura(q_0) = \{q_0, q_1, q_2\}, \Rightarrow$
$\quad\quad\quad P_2 = \{p \in Q \mid \exists\ r \in \{q_0, q_1, q_2\} \wedge p \in f(r, a)\} = \{q_0\}, \Rightarrow$
$\quad\quad \hat{f}(q_0, a) = \lambda\_clausura(P_2) = \lambda\_clausura(q_0) = \{q_0, q_1, q_2\}, \Rightarrow$
$\quad P_1 = \{p \in Q \mid \exists\ r \in \{q_0, q_1, q_2\} \wedge p \in f(r, b)\} = \{q_1\}, \Rightarrow$
$\hat{f}(q_0, ab) = \lambda\_clausura(P_1) = \lambda\_clausura(q_1) = \{q_1, q_2\}$

Todo este proceso se refleja en el siguiente esquema:



Otras extensiones para f y $\hat{f}$ son f' y $\hat{f}'$ que se aplican a conjuntos de estados:

$$f' : 2^Q \times (\Sigma \cup \{\lambda\}) \rightarrow 2^Q, \quad f'(P, a) = \bigcup_{p \in P} f(p, a)$$

$$\hat{f}' : 2^Q \times \Sigma^* \rightarrow 2^Q, \qquad \hat{f}'(P, x) = \bigcup_{p \in P} \hat{f}(p, x)$$

Para simplificar la notación se utiliza el mismo símbolo de función para referirse tanto a f como a f', $\hat{f}$ y $\hat{f}'$.

Estas definiciones previas ya permiten establecer cuál es el lenguaje aceptado por un AFλ A.

Definición 2.10 *Se define el lenguaje aceptado por un AFλ A, $A = \langle \Sigma, Q, f, q_0, F \rangle$, como el conjunto de cadenas tales que, después de ser analizadas, el AFλ ha podido alcanzar al menos un estado final, es decir,*

$$L(A) = \{x \in \Sigma^* \mid f(q_0, x) \cap F \neq \emptyset\}$$

De la definición de un AFλ se sigue que un AFN es un caso particular de AFλ en el que no existen transiciones λ. Por lo tanto, si se denota por L(AFλ) al conjunto de todos los lenguajes que pueden ser reconocidos por AFλ, entonces $L(AFN) \subseteq L(AF\lambda)$.

La cuestión que se plantea a continuación es si la inclusión contraria también se cumple; es decir, si para cualquier AFλ existe un AFN que reconozca exactamente el mismo lenguaje que el AFλ.

Teorema 2.2 *Sea L un lenguaje aceptado por un AFλ, entonces existe un AFN que también acepta L.*

Demostración:

Sea $A = \langle \Sigma, Q, f, q_0, F \rangle$ un AFλ tal que L(A) = L, entonces se define el siguiente AFN $A' = \langle \Sigma, Q, f', q_0, F' \rangle$ donde

$$f'(q, a) = \hat{f}(q, a), \forall\, q \in Q, \forall\, a \in \Sigma$$
$$F' = \begin{cases} F \cup \{q_0\} & si\ \lambda\_clausura(q_0) \cap F \neq \emptyset \\ F & en\ otro\ caso \end{cases}$$

Para demostrar que realmente $L(A) = L(A')$ se demostrará que $f'(q_0, x) = \hat{f}(q_0, x)$, por inducción en la longitud de *x*.

Paso Base: $\mid x \mid = 1$, es decir, $x = a \in \Sigma$. Para esta cadena, por definición de f' se cumple $f'(q_0, a) = \hat{f}(q_0, a)$. Por lo tanto, se cumple el enunciado.

Se observa que la inducción se ha comenzado en *i = 1* y no con *i = 0*. Este será un caso particular que se comentará más adelante.

Hipótesis de Inducción: $\forall\, x \in \Sigma^*,\ \mid x \mid \leq n$, se cumple $f'(q_0, x) = \hat{f}(q_0, x)$.

Paso de Inducción: Sea $y = xa$, tal que $x \in \Sigma^* \wedge \mid x \mid \leq n \wedge a \in \Sigma$, entonces $f'(q_0, y) = f'(q_0, xa) = f'(f'(q_0, x), a)$.

Por hipótesis de inducción, $f'(q_0, x) = \hat{f}(q_0, x) = P$, luego

$$f'(f'(q_0, x), a) = f'(P, a) = \bigcup_{p \in P} f'(q_0, a) =$$

$$\bigcup_{p \in P} \hat{f}(q_0, a) = \hat{f}(P, a) = \hat{f}(\hat{f}(q_0, x), a) = \hat{f}(q_0, y).$$

Por lo tanto, $\forall\ x \in \Sigma^*,\ \mid x \mid \geq 1$, se cumple que $f'(q_0, x) = \hat{f}(q_0, x)$ y entonces, si $x \in L(A)$ resulta que $x \in L(A')$ y viceversa.

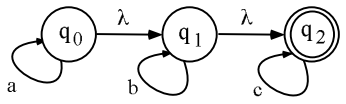
Pero, ¿qué ocurre si $\lambda \in L(A)$? En este caso $q_0 \in F$ y por lo tanto $q_0 \in F'$. ¿Y si $\lambda \in L(A')$? Entonces $q_0 \in F'$ y esto puede ser porque $q_0 \in F$, o bien, porque desde q_0 se puede alcanzar un estado de F mediante transiciones λ. En cualquiera de los dos casos $\lambda \in L(A)$.

c.q.d.

Como consecuencia de este teorema y del teorema 2.1, se obtiene el siguiente resultado:

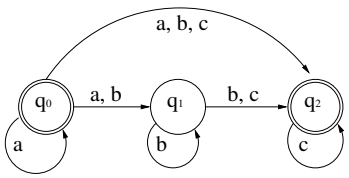
$$\mathbf{L(AFD) = L(AFN) = L(AF\lambda)}.$$

Ejemplo: Calcular un AFN que reconozca el mismo lenguaje que el siguiente AFλ



$f' = \hat{f}$	a	b	c
$\{q_0\}$	$\{q_0, q_1, q_2\}$	$\{q_1, q_2\}$	$\{q_2\}$
$\{q_1\}$	$\emptyset$	$\{q_1, q_2\}$	$\{q_2\}$
$\{q_2\}$	$\emptyset$	$\emptyset$	$\{q_2\}$
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

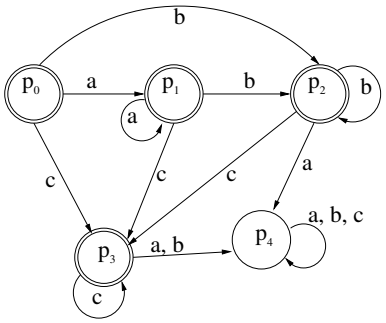
El AFN es el siguiente:



También se puede obtener un AFD directamente de un AFλ sin más que adaptar el método del paso del AFN al AFD para esta nueva situación:

f	a	b	c
$p_0 = \{q_0\}$	$\{q_0, q_1, q_2\}$	$\{q_1, q_2\}$	$\{q_2\}$
$p_1 = \{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2\}$	$\{q_1, q_2\}$	$\{q_2\}$
$p_2 = \{q_1, q_2\}$	$\emptyset$	$\{q_1, q_2\}$	$\{q_2\}$
$p_3 = \{q_2\}$	$\emptyset$	$\emptyset$	$\{q_2\}$
$p_4 = \emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Por lo tanto, el AFD equivalente es el siguiente (el estado p_4 es un estado sumidero):



2.4. Expresiones Regulares.

Las expresiones regulares permiten denotar lenguajes regulares y su estudio resulta de gran interés, tanto por su capacidad de especificación mediante un número reducido de operadores como por sus aplicaciones prácticas en la construcción de analizadores léxicos.

Definición 2.11 *Dado el alfabeto Σ, una expresión regular será el resultado de la aplicación de algunas (o todas) de las siguientes reglas un número finito de veces:*

1. *El símbolo $\emptyset$ es una expresión regular y denota el lenguaje vacío,*
2. *El símbolo λ es una expresión regular y denota el lenguaje $\{\lambda\}$,*
3. *Si* a $\in \Sigma$ *entonces* a *es una expresión regular que denota el lenguaje* $\{$a$\}$,
4. *Si α y β son expresiones regulares entonces*
   a) *$\alpha + \beta$ es una expresión regular que denota la unión de los lenguajes denotados por α y por β.*
   b) *$\alpha\beta$ es una expresión regular que denota la concatenación del lenguaje denotado por α con el denotado por β.*
   c) *α^* es una expresión regular que denota la clausura del lenguaje denotado por α.*

Ejemplos:

Si $\alpha = a^*$ *entonces* $L(\alpha) = L(a^*) = \{a^i \mid i \geq 0\}$.

Si $\alpha = a^*$ *y* $\beta = b^*$ *entonces* $L(\alpha\beta) = L(a^*b^*) = \{a^ib^j \mid i, j \geq 0\}$.

Si $\alpha = a$ *entonces* $L(\alpha^*\alpha) = L(a^*a) = L(a^*)L(a) = \{a^i \mid i \geq 0\}\{a\} = \{a^i \mid i \geq 1\}$.

Si $\alpha = a$ *entonces* $L(\alpha\alpha^*) = L(aa^*) = L(a)L(a^*) = \{a\}\{a^i \mid i \geq 0\} = \{a^i \mid i \geq 1\}$. *Por lo tanto,* $L(a^*a) = L(aa^*)$.

El último ejemplo sirve para ilustrar la cuestión de qué se entiende por expresiones regulares equivalentes: α y β son dos expresiones regulares equivalentes si $L(\alpha) = L(\beta)$.

En general se va a denotar por α tanto a la expresión regular como al lenguaje que denota, y además, para simplificar su escritura se establece una jerarquía de operadores para evaluar una expresión regular: primero se realizan las clausuras, luego las concatenaciones y por último las uniones.

Así, por ejemplo, la expresión regular $a(a)^*$ se puede escribir aa^*, y $(\alpha\beta) + \gamma$ se puede escribir $\alpha\beta + \gamma$.

Teorema 2.3 (Propiedades de las Expresiones Regulares) *Las siguientes relaciones son ciertas:*

1) $\alpha + (\beta + \gamma) = (\alpha + \beta) + \gamma = \alpha + \beta + \gamma$
2) $\alpha + \beta = \beta + \alpha$
3) $\alpha + \emptyset = \emptyset + \alpha = \alpha$
4) $\alpha(\beta\gamma) = (\alpha\beta)\gamma = \alpha\beta\gamma$
5) $\alpha\lambda = \lambda\alpha = \alpha$
6) $\alpha\beta + \alpha\gamma = \alpha(\beta + \gamma); \beta\alpha + \gamma\alpha = (\beta + \gamma)\alpha$
7) $\emptyset\alpha = \alpha\emptyset = \emptyset$
8) $\alpha + \alpha = \alpha; \alpha^* = \lambda + \alpha + \alpha^2 + \ldots + \alpha^n + \alpha^*$
9) $\lambda^* = \lambda$
10) $\emptyset^* = \lambda; \emptyset^+ = \emptyset$
11) $\alpha^*\alpha^* = \alpha^*$
12) $(\alpha^*)^* = \alpha^*$
13) $\alpha\alpha^* = \alpha^*\alpha = \alpha^+$
14) $\alpha^* = \lambda + \alpha + \alpha^2 + \ldots + \alpha^n + \alpha^{n+1}\alpha^*$
15) $\alpha^* = \lambda + \alpha\alpha^* = \lambda + \alpha^+$
16) $\alpha^* = (\lambda + \alpha)^{n-1}(\alpha^n)^*$
17) $(\alpha + \beta)^* = (\alpha^* + \beta^*)^* = (\alpha^*\beta^*)^*$
18) $(\alpha + \lambda)^* = \alpha^* + \lambda = \alpha^*$
19) $(\alpha\beta)^*\alpha = \alpha(\beta\alpha)^*$
20) $(\alpha^*\beta)^*\alpha^* = (\alpha + \beta)^*$
21) $(\alpha^*\beta)^* = (\alpha + \beta)^*\beta + \lambda$
22) $(\beta\alpha^*)^* = \beta(\alpha + \beta)^* + \lambda$

En las propiedades anteriores el signo = debe interpretarse como equivalencia; es decir, al expresar que $\alpha + \beta = \beta + \alpha$ se afirma que el lenguaje denotado por la expresión regular $\alpha + \beta$ es equivalente al denotado por la expresión regular $\beta + \alpha$.

Teorema 2.4 (de Análisis) *Sea L un lenguaje aceptado por un AFD, entonces existe una expresión regular que lo denota.*

Demostración:

Sea $A = \langle \Sigma, Q, f, q_1, F \rangle$ un AFD tal que L(A) = L. Sea $Q = \{q_1, q_2, \ldots, q_n\}$. Asociado a este AFD se pueden definir los siguientes conjuntos de cadenas:

$\forall\, k, 0 \leq k \leq n,$

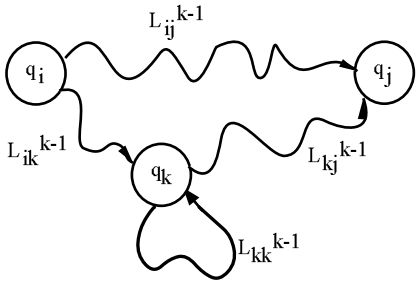
$$L_{ij}^k = \{x \in \Sigma^* \mid f(q_i, x) = q_j \wedge \forall\, y, z \in \Sigma^+ : x = yz, f(q_i, y) = q_s \rightarrow s \leq k\}$$

Es decir, el conjunto L_{ij}^k denota al conjunto de cadenas que partiendo del estado q_i alcanzan el estado q_j después de haber analizado todos sus símbolos y que, además, cualquier estado intermedio por el que se pasa tiene un índice menor que k.

El conjunto L_{ij}^k se puede calcular directamente de la definición, o bien mediante el método propuesto por la siguiente definición recursiva:

1.
$$L_{ij}^0 = \{a \in \Sigma \mid f(q_i, a) = q_j\}\ si\ i \neq j$$
$$L_{ij}^0 = \{a \in \Sigma \mid f(q_i, a) = q_j\} \cup \{\lambda\}\ si\ i = j$$

2. Supóngase construido el conjunto L_{ij}^{k-1}. El conjunto L_{ij}^k estará formado por aquellas cadenas que pasan por estados cuyos índices sean menores que k (L_{ij}^{k-1}) junto con aquellas que pasan por el estado k y por estados cuyos índices son inferiores a k. Y estas cadenas son aquellas que partiendo de q_i van a q_k, L_{ik}^{k-1}, las que desde q_k vuelven a q_k, L_{kk}^{k-1}, y las que desde q_k van a q_j, L_{kj}^{k-1}:



Por lo tanto, $L_{ij}^k = L_{ij}^{k-1} \cup L_{ik}^{k-1}(L_{kk}^{k-1})^* L_{kj}^{k-1}$.

Mediante esta nomenclatura, entonces, se puede caracterizar el conjunto de cadenas aceptadas por un AFD como aquellas cadenas que partiendo de su estado inicial, q_1, llegan a un estado final pasando por cualquier estado intermedio, es decir,

$$L(A) = \bigcup_{qj \in F} L_{1j}^n.$$

Por lo tanto, si a cada conjunto de cadenas L_{ij}^k se le puede asociar una expresión regular, r_{ij}^k, entonces también se puede asociar una expresión regular al lenguaje aceptado por un AFD.

Esta asociación se puede realizar por inducción en el valor de k:

Paso Base: r_{ij}^0 es un conjunto finito de cadenas, cada una de las cuales o es λ o está formada por un único símbolo,

$$r_{ij}^0 = \sum_{f(qi,a)=qj} a \mid a \in \Sigma \; si \; i \neq j$$
$$r_{ij}^0 = \lambda + \sum_{f(qi,a)=qj} a \mid a \in \Sigma \; si \; i = j$$

Hipótesis de Inducción: Supóngase construida la expresión regular r_{ij}^{k-1} asociada al conjunto L_{ij}^{k-1}.

Paso de Inducción: ¿Cómo construir la expresión regular r_{ij}^k asociada al conjunto L_{ij}^k? Se sabe que

$$L_{ij}^k = L_{ij}^{k-1} \cup L_{ik}^{k-1}(L_{kk}^{k-1})^* L_{kj}^{k-1},$$

por lo tanto, como cada L_{ts}^{k-1} tiene asociada la expresión regular r_{ts}^{k-1} entonces

$$r_{ij}^k = r_{ij}^{k-1} + r_{ik}^{k-1}(r_{kk}^{k-1})^* r_{kj}^{k-1}.$$

Como esta expresión es el resultado de unir, concatenar y realizar la clausura de un número finito de expresiones regulares, entonces r_{ij}^k también es una expresión regular.

Por lo tanto,

$$L(A) = \sum_{qj \in F} r_{1j}^n.$$

c.q.d.

Ejemplo:

Calcular la expresión regular asociada al AFD representado en la figura 2.3.

Hay que calcular $r_{12}^3 = r_{12}^2 + r_{13}^2(r_{33}^2)^* r_{32}^2$. *Por observación del AFD se pueden calcular directamente los* r_{ij}^0*:*

$$r_{12}^1 = r_{12}^0 + r_{11}^0(r_{11}^0)^* r_{12}^0 = 0 + \lambda\lambda^* 0 = 0$$
$$r_{13}^1 = r_{13}^0 + r_{11}^0(r_{11}^0)^* r_{13}^0 = 1 + \lambda\lambda^* 1 = 1$$
$$r_{33}^1 = r_{33}^0 + r_{31}^0(r_{11}^0)^* r_{13}^0 = (1 + \lambda) + \emptyset\lambda^* 1 = (1 + \lambda)$$
$$r_{22}^1 = r_{22}^0 + r_{21}^0(r_{11}^0)^* r_{12}^0 = (0 + \lambda) + \emptyset\lambda^* 0 = (0 + \lambda)$$
$$r_{32}^1 = r_{32}^0 + r_{31}^0(r_{11}^0)^* r_{12}^0 = 0 + \emptyset\lambda^* 0 = 0$$
$$r_{23}^1 = r_{23}^0 + r_{21}^0(r_{11}^0)^* r_{13}^0 = 1 + \emptyset\lambda^* 1 = 1$$

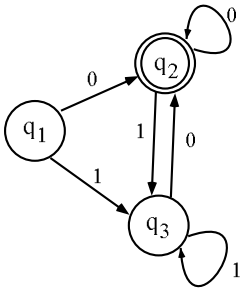


Figura 2.3: Ejemplo del cálculo de expresión regular asociada a un AFD.

Por lo tanto,

$$r_{12}^2 = r_{12}^1 + r_{12}^1(r_{22}^1)^* r_{22}^1 = 0 + 0(0+\lambda)^*(0+\lambda) = 0(\lambda + (0+\lambda)^*(0+\lambda) = 0(\lambda+0)^* = 00^*$$
$$r_{13}^2 = r_{13}^1 + r_{12}^1(r_{22}^1)^* r_{23}^1 = 1 + 0(0+\lambda)^*1 = (\lambda + 0(0+\lambda)^*)1 = (\lambda + 00^*)1 = 0^*1$$
$$r_{33}^2 = r_{33}^1 + r_{32}^1(r_{22}^1)^* r_{23}^1 = (1+\lambda) + 0(0+\lambda)^*1 = \lambda + 1 + 00^*1 = \lambda + (\lambda + 00^*)1 = \lambda + 0^*1$$
$$r_{32}^2 = r_{32}^1 + r_{32}^1(r_{22}^1)^* r_{22}^1 = 0 + 0(0+\lambda)^*(0+\lambda) = 0(\lambda + 0^*) = 00^*$$

Entonces,

$$r_{12}^3 = 00^* + 0^*1(\lambda + 0^*1)^*00^* = (\lambda + 0^*1(\lambda + 0^*1)^*)00^* =$$
$$(\lambda + 0^*1(0^*1)^*)00^* = (0^*1)^*00^* = (0^*1)^*0^*0 = (0+1)^*0$$

Teorema 2.5 (de Síntesis) *Sea r una expresión regular, entonces existe un AFλ A tal que $L(A) = r$.*

Demostración:

Hay que demostrar que para cualquier expresión regular *r* siempre se puede construir un AFλ que sea capaz de reconocer lenguaje que denota *r*. Una forma de calcular el AFλ adecuado a cada expresión regular consiste en aplicar inducción sobre el número de operadores presentes en la expresión regular.

Para simplificar el proceso de demostración se asume que todos los AFλ contienen un único estado final que no posee transiciones de salida, es decir, $F = \{q_f\}\ y\ f(q_f, a) = \emptyset\ \forall\ a \in \Sigma$.

Paso Base: En *r* no hay operadores regulares. Entonces *r* sólo puede ser $\emptyset$, λ, $a \in \Sigma$, a los que se puede asociar, respectivamente, los AFλ que se muestran en al figura 2.4.

Hipótesis de Inducción: Supóngase que para cualquier expresión regular con un número de operadores menor o igual que *n* se cumple que existe un AFλ que la reconoce.

Figura 2.4: AFλs asociados a $\emptyset$, λ y $a \in \Sigma$.

Paso de Inducción: Sea *r* con *n+1* operadores, entonces *r* se puede escribir como $r = r_1 + r_2$, $r = r_1 r_2$ o $r = r_1^*$, donde tanto r_1 como r_2 son expresiones regulares que contienen *n* o menos operadores regulares.

Se estudiará cada una de las tres posibles situaciones:

$\mathbf{r = r_1 + r_2}$**:** Como r_1 y r_2 tienen *n* o menos operadores regulares existen, por H.I., dos AFλ A_1 y A_2 tales que $L(A_1) = r_1$ y $L(A_2) = r_2$. Estos autómatas aparecen esquematizados en la figura 2.5. :

Figura 2.5: $L(A_1) = r_1$ y $L(A_2) = r_2$.

El AFλ de la figura 2.6 reconoce $L(A_1) \cup L(A_2)$, por lo tanto $r = r_1 + r_2$.

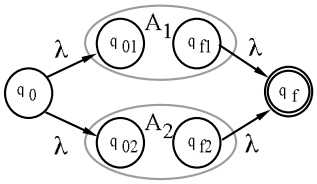


Figura 2.6: AFλ A tal que $L(A) = r_1 + r_2$.

La definición formal de este autómata es la siguiente:
Sea $A_1 = \langle \Sigma_1, Q_1, f_1, q_{01}, \{q_{f1}\} \rangle$ y $A_2 = \langle \Sigma_2, Q_2, f_2, q_{02}, \{q_{f2}\} \rangle$, tales que $Q_1 \cap Q_2 = \emptyset$.
Se define el AFλ

$$A = \langle \Sigma_1 \cup \Sigma_2, Q_1 \cup Q_2 \cup \{q_0, q_f\}, f, q_0, \{q_f\} \rangle$$

en el que la función de transición f se define como:

$$\begin{array}{l} f(q_0, \lambda) = \{q_{01}, q_{02}\} \\ f(q, a) = f_1(q, a), \forall\, q \in Q_1, \forall\, a \in \Sigma_1 \cup \{\lambda\} \\ f(q, a) = f_2(q, a), \forall\, q \in Q_2, \forall\, a \in \Sigma_2 \cup \{\lambda\} \\ f(q_{f1}, \lambda) = f(q_{f2}, \lambda) = \{q_f\} \end{array}$$

$\mathbf{r = r_1 r_2}$**:** Como r_1 y r_2 tienen *n* o menos operadores regulares existen, por H.I., dos AFλ A_1 y A_2 tales que $L(A_1) = r_1$ y $L(A_2) = r_2$. Estos autómatas aparecen esquematizados en la figura 2.7.

Figura 2.7: $L(A_1) = r_1$ y $L(A_2) = r_2$.

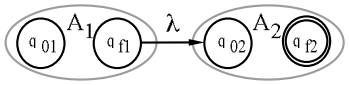


Figura 2.8: AFλ A tal que $L(A) = r_1 r_2$.

Si se permite la transición del estado final final de A_1 al inicial del autómata A_2, se obtiene el AFλ de la figura 2.8, que reconoce $L(A_1)L(A_2)$ y, por lo tanto, $r = r_1 r_2$.
Para definir formalmente este autómata, se parte de la definición de los autómatas $A_1 = \langle \Sigma_1, Q_1, f_1, q_{01}, \{q_{f1}\}\rangle$ y $A_2 = \langle \Sigma_2, Q_2, f_2, q_{02}, \{q_{f2}\}\rangle$ tal que $Q_1 \cap Q_2 = \emptyset$. Entonces, se define el AFλ

$$A = \langle \Sigma_1 \cup \Sigma_2, Q_1 \cup Q_2, f, q_{01}, \{q_{f2}\}\rangle$$

en el que la función de transición f se define como:

$$f(q,a) = f_1(q,a), \forall\, q \in Q_1 - \{q_{f1}\}, \forall\, a \in \Sigma_1 \cup \{\lambda\}$$
$$f(q_{f1}, \lambda) = \{q_{02}\}$$
$$f(q,a) = f_2(q,a), \forall\, q \in Q_2, \forall\, a \in \Sigma_2 \cup \{\lambda\}$$

$\mathbf{r = r_1^*}$: Como r_1 tiene *n* o menos operadores regulares existe, por H.I., un AFλ A_1 tal que $L(A_1) = r_1$:

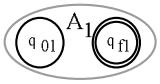


Figura 2.9: AFλ A_1 tal que $L(A_1) = r_1$.

El AFλ de la figura 2.10, reconoce $[L(A_1)]^*$, por lo tanto $r = r_1^*$.
En este caso, para la definición formal, se parte del autómata $A_1 = \langle \Sigma_1, Q_1, f_1, q_{01}, \{q_{f1}\}\rangle$.
Y se define el AFλ

$$A = \langle \Sigma_1, Q_1 \cup \{q_0, q_f\}, f, q_0, \{q_f\}\rangle$$

en el que la función de transición f se define como:

$$f(q_0, \lambda) = f(q_{f1}, \lambda) = \{q_{01}, q_f\}$$
$$f(q,a) = f_1(q,a), \forall\, q \in Q_1 - \{q_{f1}\}, \forall\, a \in \Sigma_1 \cup \{\lambda\}$$

c.q.d.

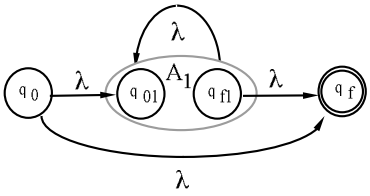


Figura 2.10: AFλ A tal que $L(A) = r_1^*$.

2.5. Otros Tipos de Autómatas Finitos.

Además de las extensiones vistas (no deterministas, deterministas, movimientos libres, expresiones regulares) existen otra serie de modelos de máquinas virtuales con una capacidad de cálculo equivalente a los autómatas finitos. De entre estos autómatas se van a exponer las principales características de los autómatas finitos bidireccionales y de los autómatas con función de salida (Mealy y Moore).

2.5.1. Autómatas Finitos Bidireccionales

Un autómata finito bidireccional determinista es un AFD en el que el cabezal de lectura puede desplazarse tanto una casilla a la derecha como una casilla a la izquierda.

Definición 2.12 *Un autómata finito determinista bidireccional, AF2, es una quíntupla* $A = \langle \Sigma, Q, q_0, F, f \rangle$ *tal que*

Σ *es el alfabeto de entrada,*

Q *es un conjunto finito y no vacío de estados,*

f *es una función total definida como*

$$f : Q \times \Sigma \rightarrow Q \times \{L, R\},$$

q_0 *es el denominado estado inicial,* $q_0 \in Q$*,*

F *es el conjunto de estados denominados finales,* $F \subseteq Q$*.*

La definición de la función de transición no sólo determina las transiciones entre estados, sino también el movimiento del cabezal de forma que si $f(q, a) = (p, L)$ entonces desde el estado *q* con el símbolo *a* se pasa al estado *p* y el cabezal se desplaza una casilla a la izquierda, y si $f(q, a) = (p, R)$ entonces desde el estado *q* con el símbolo *a* se pasa al estado *p* y el cabezal se desplaza una casilla a la derecha.

Para determinar cuál es el comportamiento de un AF2 con respecto a una cadena de entrada, se introduce el concepto de descripción instantánea. Una *descripción instantánea*,

D.I., es una cadena de $\Sigma^* Q \Sigma^*$ de forma que la cadena *wqx* representa la siguiente situación:

- *wx* es la cadena de entrada,
- *q* es el estado actual,
- el cabezal está sobre el primer símbolo de la cadena *x*.

Se define la relación $I_1 \vdash I_2$ como:

1. $a_1 a_2 \ldots a_{i-1} \mathbf{q} a_i \ldots a_n \vdash a_1 a_2 \ldots a_i \mathbf{p} a_{i+1} \ldots a_n$ si $f(q, a_i) = (p, R)$
2. $a_1 a_2 \ldots a_{i-1} \mathbf{q} a_i \ldots a_n \vdash a_1 a_2 \ldots a_{i-2} \mathbf{p} a_{i-1} \ldots a_n$ si $f(q, a_i) = (p, L)$

Se dice que $I_1 \overset{*}{\vdash} I_n$ cuando existen *n* D.I. $I_1, I_2, \ldots, I_n$ tales que se cumple $I_i \vdash I_{i+1}, \forall\, i : 1 \leq i \leq n-1$.

De esta forma, se puede especificar cuál es el lenguaje aceptado por un AF2 A, mediante la expresión

$$L(A) = \{w \in \Sigma^* \mid q_0 w \overset{*}{\vdash} wp, p \in F\}$$

Ejemplo de AF2:

f	0	1
q_0	(q_0, R)	(q_1, R)
q_1	(q_1, R)	(q_2, L)
q_2	(q_0, R)	(q_2, L)

Este AFD2 reconoce todas las cadenas de ceros y unos que no tienen dos unos consecutivos, es decir, $(\lambda + 1)(0 + 01)^*$. *Todos los estados son finales.*

Resulta evidente que si en un AF2 sólo se permite el desplazamiento a la derecha, entonces el comportamiento es similar al de un AFD. Por lo tanto, si se denota por L(AF2) a la clase de todos los lenguajes aceptados por AF2 se verifica $L(AFD) \subseteq L(AF2)$. La relación inversa también se cumple tal y como se enuncia en el siguiente resultado, que no se demuestra:

Teorema 2.6 *Si L es un lenguaje aceptado por un AF2, entonces existe un AFD A tal que L(A) = L.*

De este resultado, se desprende que

$$L(AFD) = L(AF2).$$

2.5.2. Autómatas Finitos con Función de Salida.

Hasta el momento los AF descritos tienen como salida una señal del tipo acepto/no acepto (según se alcance un estado final o no después de consumir toda la cadena de entrada).

Los autómatas finitos con función de salida son modelos que traducen la cadena de entrada en otra cadena de salida. Si esa traducción se realiza en cada estado entonces se tiene una máquina de Moore y si la traducción se realiza en cada transición entonces se tiene una máquina de Mealy.

Máquina de Moore.

Es una séxtupla $A = \langle Q, \Sigma, \Delta, f, \gamma, q_0 \rangle$ donde Q, Σ, f y q_0 se definen como en un AFD. El conjunto Δ representa al alfabeto de salida y γ es una función que asocia a cada estado un símbolo del alfabeto de salida, es decir, $\gamma : Q \rightarrow \Delta$.

La respuesta de A con respecto a la entrada $a_1 a_2 \ldots a_n, n \geq 0$, es

$$\gamma(q_0)\gamma(q_{i1})\gamma(q_{i2}) \ldots \gamma(q_{in})$$

donde $q_0, q_{i1}, q_{i2}, \ldots, q_{in}$ es la secuencia de estados tal que $f(q_{ij}, a_{i+1}) = q_{ij+1}$.

Ejemplo:

Escribir una máquina de Moore que determine el resto de dividir un número binario entre 3.

El resto de dividir un número entre 3 será el entero 0, el 1 o el 2. Por lo tanto, se establece una Máquina de Moore con tres estados, q_0, q_1 y q_2, de forma que $\gamma(q_0) = 0$, $\gamma(q_1) = 1$ y $\gamma(q_2) = 2$.

Para determinar las transiciones entre estados se realiza el siguiente análisis:

- *Si a un número binario* i *se le añade un* 0 *por la derecha se obtiene un nuevo entero,* i0*, cuyo valor es el de* i *multiplicado por* 2*.*
  Por lo tanto, si $i \bmod 3 = r$, entonces $2i \bmod 3 = 2r \bmod 3 = r'$. Cuando se parte de un valor $r = 0$, se obtiene para r' el valor 0*; si $r = 1 \Rightarrow r' = 2$ y si $r = 2 \Rightarrow r' = 1$.*
- *Si al entero* i *se le añade un* 1 *se obtiene como nuevo valor,* i1*, el anterior multiplicado por* 2 *y añadiéndole* 1*.*
  Por lo tanto, si $i \bmod 3 = r$, entonces $(2i + 1) \bmod 3 = (2r + 1) \bmod 3 = r'$. Entonces, si $r = 0 \Rightarrow r' = 1$, si $r = 1 \Rightarrow r' = 0$ y si $r = 2 \Rightarrow r' = 2$.

Este análisis permite diseñar la Máquina de Moore de la figura 2.11.

El comportamiento de esta Máquina de Moore ante la cadena de entrada `110100` *es la cadena de salida $\gamma(q_0)\gamma(q_1)\gamma(q_0)\gamma(q_0)\gamma(q_1)\gamma(q_2)\gamma(q_1)$, es decir,* `0100121`.

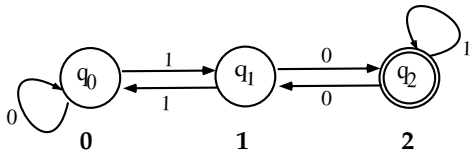


Figura 2.11: Máquina de Moore para calcular el resto de la división entera entre 3.

Al interpretar este resultado se obtiene que `110100` *es el entero 52, y* $52 \bmod 3 = 1$*, que coincide con el último símbolo de salida proporcionado por la máquina de Moore.*

Máquina de Mealy.

Es una séxtupla $A = \langle Q, \Sigma, \Delta, f, \gamma, q_0 \rangle$ donde Q, Σ, f y q_0 se definen como en un AFD. El conjunto Δ representa al alfabeto de salida y γ es una función que asocia a cada transición un símbolo del alfabeto de salida, es decir, $\gamma : Q \times \Sigma \rightarrow \Delta$.

La respuesta de A con respecto a la entrada $a_1 a_2 \ldots a_n, n \geq 0$, es

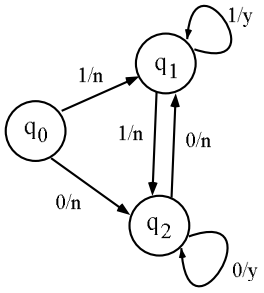
$$\gamma(q_0, a_1)\gamma(q_{i1}, a_2)\gamma(q_{i2}, a_3) \ldots \gamma(q_{in-1}, a_n)$$

donde $q_0, q_{i1}, q_{i2}, \ldots, q_{in-1}$ es la secuencia de estados tal que $f(q_{ij}, a_{i+1}) = q_{ij+1}$.

Ejemplo:

Escribir una máquina de Mealy que produzca salida 'y' *cuando los dos últimos símbolos leídos de la cadena sean iguales y que produzca salida* 'n' *en otro caso.*

Para realizar esta máquina basta con recordar cual ha sido el último símbolo leído de la cadena de entrada, y si su siguiente símbolo coincide con el anterior, entonces se produce salida 'y'*; en cualquier otra situación la salida será* 'n'.



2.6. Aplicaciones.

Los lenguajes regulares son utilizados en aplicaciones muy diversas. Entre ellas, en esta sección se comentan tres de las más usuales,

- análisis léxico de un compilador (exp. regulares, AF),
- especificación de parámetros de entrada (expresiones regulares),
- ayuda al diseño de circuitos (Mealy, Moore).

2.6.1. Aplicación como ayuda al análisis léxico en un compilador.

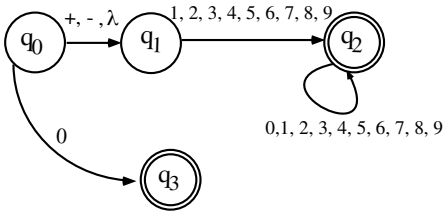
Un compilador es un programa que traduce un código denominado *fuente* en otro código denominado *objeto*. Para realizar esa transformación son necesarias varias etapas. La primera de estas etapas consiste en detectar, de todas las cadenas que componen el código fuente, cuáles son representativas para el compilador. Por ejemplo, qué cadenas representan *constantes* (enteras, reales, lógicas, etc) o bien *palabras reservadas* del lenguaje (`begin`, `end`, $\{, \}$, `procedure`, `function`, `main`, `while`, ... etc.). Estas cadenas se representan por medio de expresiones regulares y su detección se realiza por medio de autómatas finitos.

El siguiente ejemplo muestra cómo representar constantes enteras, con o sin signo, utilizando expresiones regulares y cómo reconocer estas constantes por medio de un autómata finito.

Expresión Regular:

$$\texttt{entero} = ('+' + '-' + \lambda)(\texttt{digitonocero})(\texttt{digito})^* + '0'$$
$$\texttt{digitonocero} = '1' + '2' + '3' + '4' + '5' + '6' + '7' + '8' + '9'$$
$$\texttt{digito} = \texttt{digitonocero} + '0'$$

Autómata Finito:



Esta tarea resulta tan habitual que se encuentran disponibles varias herramientas de programación que automatizan la tarea de diseñar los programas que se comporten como reconocedores léxicos. Por ejemplo, la orden *lex* del sistema operativo `UNIX` establece un mecanismo para generar un programa ejecutable que se comporte como un autómata finito mediante la especificación de su expresión regular asociada.

2.6.2. Especificación de parámetros de entrada.

En algunos programas que requieran de la interacción del usuario para buscar ciertos patrones de caracteres sobre un determinado fichero, suele resultar bastante aconsejable

permitir la introducción de esos datos mediante expresiones regulares. Por ejemplo, en la mayoría de sistemas operativos se incluye la capacidad de buscar ficheros cuyo nombre presente algunas características expresables mediante expresiones regulares.

En el siguiente ejemplo se observa como listar en el sistema operativo UNIX todos los ficheros cuyo nombre empiece por `'p'` *y su tercer carácter sea un* `'5'`*:* `ls p?5*`*. Este comando da lugar al autómata finito que se muestra en la figura 2.12.*

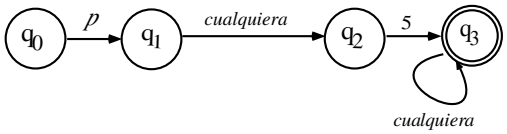


Figura 2.12: Autómata Finito asociado a la expresión `p?5*`

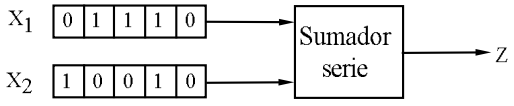
Otro ejemplo de utilización de expresiones regulares en aplicaciones de texto se presenta en el editor de texto `UNIX`, *ed*. Para representar la ejecución de tareas tales como la búsqueda de un texto y su sustitución por otro texto en todo un fichero se utiliza un comando como el siguiente: `s/ bbb*/b`. Al ejecutar este comando se sustituyen las secuencias de dos o más caracteres blancos en el fichero `bbb*` por un único carácter blanco `b`. En este caso el editor *ed* transforma el comando de entrada `s/bbb*/b` en un autómata finito con movimientos libres con el fin de ahorrar memoria al representar el autómata finito.

2.6.3. Ayuda al diseño de circuitos.

Tanto las máquinas de Moore como las máquinas de Mealy se utilizan frecuentemente para diseñar circuitos.

Como ejemplo de este tipo de aplicación se presenta el diseño de un sumador binario serie mediante una máquina de Mealy.

El diagrama de bloques de un sumador binario en serie es el siguiente:



Este diagrama de bloques se puede representar mediante una tabla de estados que contendrá dos estados: q_0 *y* q_1*. El estado* q_0 *representa la condición de acarreo* 0 *mientras que el estado* q_1 *representa la condición de acarreo* 1*.*

$Estado\ actual$	$x_1x_2 = 00$	01	10	11
q_0	$q_0, 0$	$q_0, 1$	$q_0, 1$	$q_1, 0$
q_1	$q_0, 1$	$q_1, 0$	$q_1, 0$	$q_1, 1$

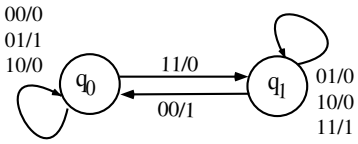


Figura 2.13: Máquina de Mealy para un sumador binario.

La entrada $(q_0, 00) = (q_0, 0)$ *representa que si el dígito* x_1 *es* 0*, el dígito* x_2 *es* 0 *y la máquina de Mealy se encuentra en el estado* q_0*, entonces la máquina de Mealy permanece en el estado* q_0 *y produce el dígito de salida* 0*.*

Capítulo 3

Propiedades de los Lenguajes Regulares

Índice General

3.1. El Teorema de Myhill-Nerode. Minimización de Autómatas Finitos. 43
3.2. Lema de Bombeo. 50
3.3. Propiedades de Clausura. 53
3.4. Algoritmos de Decisión para Lenguajes Regulares. 57

En este tema se presentan las propiedades que cumplen los lenguajes regulares.

El primer resultado a estudiar, el teorema de Myhill-Nerode, es de especial trascedencia puesto que establece la existencia de un AF mínimo para aceptar un determinado lenguaje regular y, además, permite desarrollar un método para calcularlo.

Además, se presentan los resultados que satisfacen los lenguajes regulares. Estos resultados permitirán saber si un lenguaje dado es o no es regular. El interés de estudiar este tipo de herramientas (lema de bombeo, propiedades de clausura) es muy grande, ya que cuando se determina el tipo más restrictivo al que pertenece un lenguaje, se está determinando el mínimo número de recursos computacionales necesario para reconocer sus cadenas.

3.1. El Teorema de Myhill-Nerode. Minimización de Autómatas Finitos.

Sea D un conjunto, una **relación binaria R** es un subconjunto de $D \times D$, es decir, $R \in D \times D$. Si el par $(a, b) \in R$, entonces se denota como aRb; en caso contrario, se denota como $a\not{R}b$.

Definición 3.1 (Relación Binaria de Equivalencia, RBE) *Una relación binaria se dice de equivalencia -r.b.e.- si, y sólo si:*

1. *cumple la propiedad reflexiva:* $\forall\, x \in D, xRx$,
2. *cumple la propiedad simétrica:* $\forall\, x, y \in D$, *si* xRy *entonces* yRx,
3. *cumple la propiedad transitiva:* $\forall\, x, y, z \in D$, *si* $xRy \wedge yRz$ *entonces* xRz.

Ejemplo:

- $iR_{\leq}j$ *si y sólo si* $i \leq j$ *no es r.b.e., puesto que es reflexiva y transitiva, pero no simétrica.*
- $iR_m j$ *si y sólo si* $(i - j) \bmod m = 0$, *es r.b.e.*

Sea R una relación binaria de equivalencia. Se define la *clase de equivalencia* $[x]_R$ como el conjunto $\{y \in D \mid xRy\}$.

El *índice* de una r.b.e. es el número distinto de clases de equivalencia que contiene.

Sea Σ un alfabeto, se dice que una r.b.e. R, $R \subseteq \Sigma^* \times \Sigma^*$, es invariante a la derecha con respecto a la concatenación si $\forall\, z \in \Sigma^*$ se verifica que xRy implica que $xzRyz$.

Teorema 3.1 (Teorema de Myhill-Nerode) *Las tres afirmaciones siguientes son equivalentes:*

1. *El lenguaje* $L \subseteq \Sigma^*$ *es aceptado por algún AF.*
2. *L es la unión de algunas de las clases de equivalencia de una r.b.e. invariante a la derecha de índice finito.*
3. *Sea* R_L *la r.b.e.*

$$xR_L y \text{ si y sólo si } \forall\, z \in \Sigma^*, xz \in L \text{ sólo cuando } yz \in L,$$

entonces R_L *es de índice finito.*

Demostración:

Se realizará una demostración cíclica en la que el establecimiento de cada una de las afirmaciones implica la afirmación de la siguiente:

$(1) \Rightarrow (2)$ Asúmase que L es aceptado por algún AFD $A = \langle \Sigma, Q, q_0, f, F \rangle$.

Se define la r.b.e. R_A como

$$xR_A y \text{ si y sólo si } f(q_0, x) = f(q_0, y).$$

Esta relación acoge a las cadenas que, partiendo del estado inicial del AF, conducen al mismo estado. Y esta relación es invariante a la derecha puesto que

$$\forall\, z \in \Sigma^*, f(q_0, xz) = f(f(q_0, x), z) = f(f(q_0, y), z) = f(q_0, yz).$$

El índice de R_A, es decir, el número de clases de equivalencia es finito, puesto que, como máximo, puede ser igual al número de estados del AF A.

Además, L, es la unión de aquellas clases de equivalencia que incluyen a aquellos estados que son finales en el AFD A.

(2) $\Rightarrow$ (3) Sea R_Σ una r.b.e. que satisface (2).

Sean $x, y \in \Sigma^*$ tales que $xR_\Sigma y$. Como R_Σ es invariante a la derecha, entonces

$$\forall\, z \in \Sigma^*, xzR_\Sigma yz,$$

y como L es la unión de algunas de las clases de equivalencia de R_Σ, se tiene que $xz \in L$ si, y sólo si, $yz \in L$.

Por lo tanto, se cumple la relación R_L, y entonces $xR_L y$.

De esto se deduce $[x]_{R\Sigma} \subseteq [x]_{RL}$, es decir, cada clase de equivalencia de R_Σ está contenida en alguna de R_L. Según esto, R_Σ es un refinamiento de R_L o, lo que es lo mismo, tiene un índice mayor o igual que el índice de R_L.

Y como el índice de R_Σ es finito, también lo debe ser el de R_L.

(3) $\Rightarrow$ (1) Primero se comprueba que R_L es invariante a la derecha.

Supóngase que $xR_L y$, y sea *w* una cadena de Σ^*. Se debe comprobar que $xwR_L yw$, esto es,

$$\forall\, z \in \Sigma^*, xwz \in L \; si, \; y \; sólo \; si, \; ywz \in L.$$

Pero como $xR_L y$ se conoce, por definición de R_L, que

$$\forall\, v \in \Sigma^*, xv \in L \; si, \; y \; sólo \; si, \; yv \in L.$$

Sea $v = wz$ para demostrar que R_L es invariante a la derecha.

Sea Q' el conjunto finito de clases de equivalencia de R_L y sea $[x]$ el elemento de Q' que contiene a *x*. Se define $f'([x], a) = [xa]$. La definición es consistente, pues R_L es invariante a la derecha.

Sea $q'_0 = [\lambda]$ y $F' = \{[x] \mid x \in L\}$.

El AFD $A' = \langle Q', \Sigma, f', q'_0, F' \rangle$ acepta L ya que $f'(q'_0, x) = f'([\lambda], x) = [x]$ y $x \in L(A')$ si, y sólo si $[x] \in F'$.

c.q.d.

En la demostración del teorema de Myhill-Nerode se construye un AFD *A* que reconoce un lenguaje regular L según la relación R_L. También se expone que cualquier otra r.b.e. que se establezca sobre un AFD A′ que reconozca el mismo lenguaje va a tener un número de estados que será mayor o igual al número de estados del AFD proporcionado por la

demostración del teorema de Myhill-Nerode. Por lo tanto, este autómata tendrá el *número mínimo de estados* que pueda tener cualquier autómata que reconozca el mismo lenguaje.

La pregunta que se establece ahora es, ¿puede existir otro AFD que reconozca este mismo lenguaje que tenga el mismo número de estados que el AFD dado por el teorema de Myhill-Nerode pero que tenga una diferente función de transición?

Teorema 3.2 *El AFD de mínimo número de estados de entre todos los AFD que aceptan un mismo lenguaje regular es único, salvo isomorfismos (renombramiento de estados).*

Demostración:

Como ya se ha comentado, el AFD obtenido en el tercer paso de la demostración del teorema de Myhill-Nerode tiene el número mínimo de estados posible.

Sea $A' = \langle \Sigma, Q', q'_0, f', F' \rangle$ este autómata, y sea $A = \langle \Sigma, Q, q_0, f, F \rangle$ otro AFD tal que $L(A) = L(A')$. Supóngase que ambos AFD tienen el mismo número de estados, $\mid Q \mid = \mid Q' \mid$.

Sea *q* un estado de *Q*; entonces, debe existir una cadena *x* tal que $f(q_0, x) = q$ puesto que, en caso contrario, *q* se podría eliminar de *Q* (ya que sería inalcanzable), obteniéndose otro AFD de menor número de estados que el AFD A', lo que se ha demostrado que no es posible.

Identifíquese *q* con el estado al que se llega al analizar la cadena *x* en A', es decir, $f'(q_0, x) = q$. Entonces, si $f(q_0, x) = f(q_0, y) = q$ resulta que ambas cadenas, *x* e *y*, deben estar en la misma clase de equivalencia de R_L, y por lo tanto, $f'(q_0, x) = f'(q_0, y)$.

c.q.d.

Ejemplo:

Sea $L = \{x \in (0+1)^* \mid S(x,0) = \dot{2} \wedge S(x,1) = \dot{2}\}$*, es decir, el lenguaje formado por cadenas que tienen un número par de símbolos '0'y un número par de símbolos '1'.*

Este lenguaje es aceptado por el AFD A'*, que se muestra en la figura 3.1.*

Todos los estados de este autómata son alcanzables, por lo tanto, la relación R_A*, tiene seis clases de equivalencia:*

$$
\begin{array}{ll}
[q_0] = [\lambda] & \textit{representa } \{x \in \Sigma^* \mid f(q_0, x) = q_0\} \\
[q_1] = [0] & \textit{representa } \{x \in \Sigma^* \mid f(q_0, x) = q_1\} \\
[q_2] = [1] & \textit{representa } \{x \in \Sigma^* \mid f(q_0, x) = q_2\} \\
[q_3] = [01] & \textit{representa } \{x \in \Sigma^* \mid f(q_0, x) = q_3\} \\
[q_4] = [11] & \textit{representa } \{x \in \Sigma^* \mid f(q_0, x) = q_4\} \\
[q_5] = [110] & \textit{representa } \{x \in \Sigma^* \mid f(q_0, x) = q_5\}
\end{array}
$$

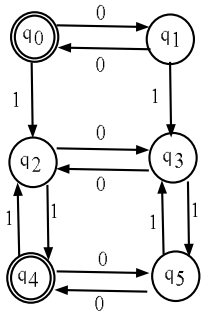


Figura 3.1: Autómata Finito Determinista que reconoce L.

L(A), por lo tanto, es la unión de las clases de equivalencia asociadas a estados finales: $[\lambda] \cup [11]$.

Por lo que respecta a la relación R_L*, esta contendrá cuatro estados:*

- $[q'_0] = [\lambda]$*, representa las cadenas que tienen un número par de símbolos '0'y un número par de símbolos '1',*
- $[q'_1] = [0]$*, representa las cadenas que tienen un número impar de símbolos '0'y un número par de símbolos '1',*
- $[q'_2] = [1]$*, representa las cadenas que tienen un número par de símbolos '0'y un número impar de símbolos '1',*
- $[q'_3] = [01]$*, representa las cadenas que tienen un número impar de símbolos '0'y un número impar de símbolos '1'.*

Aplicando el tercer paso de la demostración del teorema de Myhill-Nerode se obtiene el siguiente AFD mínimo:

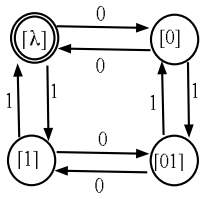


Figura 3.2: Mínimo AFD que reconoce L.

Para obtenerlo, se ha tenido en cuenta que,

- *si desde* $[q'_0] = [\lambda]$ *se lee '0', entonces se pasa a tener un número impar de símbolos '0'(es decir, se pasa a [0] ya que el número de símbolos '1'no ha cambiado, sigue siendo par) y si se lee '1', un número impar de símbolos '1'(es decir, se pasa a [1], número par de símbolos '0'y un número impar de símbolos '1'),*
- *si desde* $[q'_1] = [0]$ *se lee '0', entonces se pasa a tener un número par de símbolos '0'(es decir, se pasa a* $[\lambda]$*), y si se lee '1', un número impar de símbolos '1'(es decir, se pasa a [01]),*

- *si desde* $[q_1'] = [1]$*, se lee '0', entonces se pasa a tener un número impar de símbolos '0'(es decir, se pasa a [01]) y si se lee '1', un número par de símbolos '1'(es decir, se pasa a* $[\lambda]$*),*
- *si desde* $[q_1'] = [01]$*, se lee '0', entonces se pasa a tener un número par de símbolos '0'(es decir, se pasa a [1]) y si se lee '1', un número par de símbolos '1'(es decir, se pasa a [0]).*

Para finalizar el tema, lo que se plantea es cómo obtener un método práctico que permita calcular la relación R_L.

Sea $\equiv$ la r.b.e. sobre los estados de A tal que $p \equiv q$ si, y sólo si, para cada cadena de entrada *x*, $f(p, x)$ es un estado final si, y sólo si, $f(q, x)$ es un estado final.

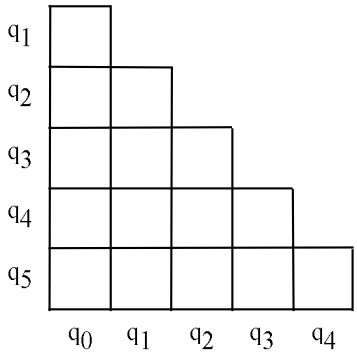
Hay un isomorfismo entre aquellas clases de equivalencia de $\equiv$ que contienen un estado alcanzable desde q_0 para alguna cadena de entrada y los estados del AFD de número mínimo de estados. Por lo tanto, los estados de AFD mínimo se pueden identificar con estas clases.

Se emplea la siguiente notación: si $p \equiv q$ se dice que *p es equivalente a q* y en caso contrario se dice que *p es distinguible de q*.

El cálculo de la relación $\equiv$ se realiza por medio del siguiente método, que se ilustra tomando como ejemplo el AFD presentado en la figura 3.1:

1. Se crea una tabla con $\mid Q \mid -1$ filas y $\mid Q \mid -1$ columnas. Cada fila y cada columna se etiqueta con uno de los estados de *Q*, de forma que si *q* es un estado de *Q* entonces no existen en la tabla entradas correspondientes a parejas *(q,q)*. Asímismo, si *q* y *p* son estados de *Q*, la entrada correspondiente a la pareja *(q,p)* también representa a la pareja *(p,q)*.

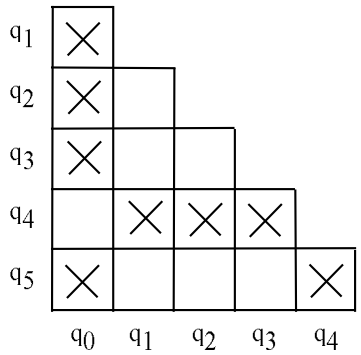
   En el ejemplo, la tabla correspondiente sería de la forma,



2. Se marca cada entrada de la tabla que se corresponde con una pareja *(estado final,*

estado no final), pues todas esas parejas se corresponden con pares de estados distinguibles.

En el ejemplo, se tienen las siguientes parejas de estados final/no final,



3. Para cada par de estados *(p,q)* que no se haya analizado hasta el momento, se consideran los pares de estados *(r,s)* tales que $r = f(q, a)$ y $s = f(p, a)$, para cada símbolo de entrada *a*.

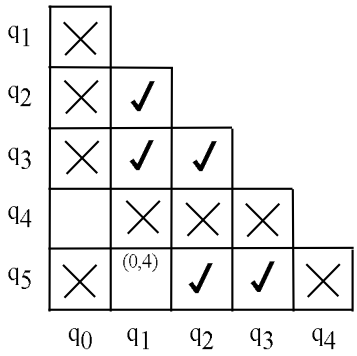
   Si los estados *r* y *s* son distinguibles para alguna cadena *x*, entonces los estados *p* y *q* son distinguibles por la cadena *ax*.

   Por lo tanto, si la entrada *(r,s)* está marcada en la tabla, entonces también se marca la entrada *(p,q)*. Si la entrada *(r,s)* no está marcada, entonces el par *(p,q)* se coloca en una lista asociada con la entrada *(r,s)*. Si posteriormente se marca la entrada *(r,s)*, también se marcarán todas las parejas de la lista asociada.

   En el ejemplo, se obtiene lo siguiente:

   - *El análisis del par* (q_0, q_4) *remite al par* (q_1, q_5)*; por lo tanto, se coloca en la lista asociada a dicha entrada.*
   - *El análisis del par* (q_1, q_2) *remite al par* (q_0, q_3)*; por lo tanto, se marca.*
   - *El análisis del par* (q_1, q_3) *remite al par* (q_0, q_2)*; por lo tanto, se marca.*
   - *El análisis del par* (q_1, q_5) *remite al par* (q_0, q_4)*; por lo tanto, no se puede marcar ninguna de las dos entradas.*
   - *El análisis del par* (q_2, q_3) *remite al par* (q_3, q_4)*; por lo tanto, se marca.*
   - *El análisis del par* (q_2, q_5) *remite al par* (q_3, q_4)*; por lo tanto, se marca.*
   - *El análisis del par* (q_3, q_5) *remite al par* (q_2, q_4)*; por lo tanto, se marca.*

Al finalizar este proceso todas aquellas entradas de la tabla que queden vacías identifican parejas de estados equivalentes.



En el ejemplo, los estados q_1 y q_5 son equivalentes, y lo mismo sucede con los estados q_0 y q_4. Se puede comprobar que el autómata obtenido es el presentado en la figura 3.2.

3.2. Lema de Bombeo.

Este lema proporciona una herramienta muy útil para demostrar que ciertos lenguajes **no son regulares**, es decir, que no pueden ser reconocidos por autómatas finitos. Además, como consecuencias adicionales de su enunciado, también resulta útil como referencia teórica para desarrollar algoritmos que respondan a ciertas cuestiones sobre aspectos determinados de autómatas finitos, como por ejemplo, si el lenguaje aceptado por un autómata finito es finito o infinito.

Lema 3.1 (Lema de Bombeo) *Para todo lenguaje regular L existe una constante* n*, dependiente únicamente de L, tal que si* z *es una cadena de L, y se cumple que* $\mid z \mid \geq n$*, entonces la cadena* z *se puede descomponer como* z = uvw *tal que:*

1. $\mid v \mid \geq 1$,
2. $\mid uv \mid \leq n$,
3. *Para todo* $i \geq 0$ *las cadenas* uv^iw *son, todas, cadenas de L.*

Demostración:

Si un lenguaje es regular, entonces es aceptado por un autómata finito determinista, AFD, $A = \langle Q, \Sigma, f, q_0, F \rangle$. Sea |Q| = n, es decir, el AFD tiene *n* estados.

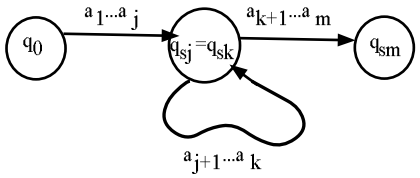
Sea $z = a_1 a_2 \ldots a_m$ una cadena de *m* símbolos, tal que $m \geq n$, y supóngase que $f(q_0, a_1 a_2 \ldots a_i) = q_{si}$, donde q_{si} representa el estado que se alcanza después de analizar los primeros a_i símbolos. Según esto, $f(q_0, a_1) = q_{s1}$, $f(q_0, a_1 a_2) = q_{s2}$, ... etc.

Como la cadena z tiene *m* símbolos y el AFD tiene *n* estados distintos, y $m \geq n$,

entonces si el AFD A comienza a trabajar con la cadena *z*, no puede ser que los primeros *n + 1* estados que se suceden en el análisis de la cadena *z* (q_0, q_{s1}, q_{s2}, $\ldots q_{sn}$) sean todos distintos.

Por lo tanto, existen al menos dos enteros, llamémosles *j* y *k*, $1 \leq j < k \leq n$ tales que $q_{sj} = q_{sk}$, es decir, $f(q_0, a_1 \ldots a_j) = q_{sj}$, $f(q_0, a_1 \ldots a_k) = q_{sk} = q_{sj} \wedge f(q_k, a_{k+1} \ldots a_m) = q_{sm}$. Como $j < k$ entonces la subcadena $a_{j+1} \ldots a_k$ tiene una longitud mayor o igual que *1*, y como $k \leq n$ entonces su longitud no puede ser mayor que *n*.

Gráficamente, la idea se puede expresar de la forma siguiente:



Si la cadena $a_1 \ldots a_m$ pertenece al lenguaje reconocido por el AFD A, entonces $q_{sm} \in F$, y por lo tanto, la cadena $a_1 \ldots a_j a_{k+1} \ldots a_m$ también pertenece a *L(A)*. En este caso, el bucle que reconoce $a_{j+1} \ldots a_k$ no se realiza ninguna vez. Pero también podría darse el caso de realizarlo *i* veces, en cuyo caso la cadena reconocida sería $a_1 \ldots a_j (a_{j+1} \ldots a_k)^i a_{k+1} \ldots a_m$.

¿Qué conclusión se puede sacar de este proceso? Que si se tiene un AFD y una cadena *z* de una longitud lo suficientemente larga que sea aceptada por el AFD, entonces se puede localizar una subcadena de *z* tal que esa subcadena se puede repetir tantas veces como se quiera (es decir, se puede "*bombear*") obteniéndose como resultado de este proceso nuevas cadenas que también serán aceptadas por el AFD.

Con esta idea gráfica, para completar la demostración basta con identificar *u* con $a_1 \ldots a_j$, *v* con $a_{j+1} \ldots a_k$ y *w* con $a_{k+1} \ldots a_n$. Como $j < k$ entonces $\mid v \mid = k - (j + 1) + 1 = k - j \geq 1$ y como $k \leq n$, entonces $\mid uv \mid \leq n$.

c.q.d.

Este lema se aplica para demostrar que un lenguaje *no es regular*, es decir, si un lenguaje no cumple con el lema de bombeo entonces se puede garantizar que no es regular, pero si cumple las condiciones del lema entonces no se puede asegurar si es o no es regular.

Ejemplos:

1. *Demostrar que el lenguaje* $\{a^n b^n \mid n \geq 0\}$ *no es un lenguaje regular.*
   Para poder comprobar si se cumplen las condiciones impuestas por el lema de bombeo, se debe localizar una cadena del lenguaje cuya longitud sea mayor que la constante del lema para este lenguaje. Sea `t` *esa constante. Entonces* $z = a^t b^t$ *pertenece al lenguaje y su longitud es mayor*

que t*, por lo tanto* z *se puede escribir como* uvw*, pero ¿cuales son los símbolos de* z *que componen la cadena* v*? Se analizan todos los casos posibles y si al menos uno de estos casos fuese posible, entonces no se podría demostrar que no se cumple el lema para esta cadena:*

a) *Si* v *está compuesta sólo de símbolos* a*, entonces las cadenas* $uv^iw,\ i \geq 2$ *no pertenecen al lenguaje (contienen más* a*'s que* b*'s).*

$$\underbrace{a\ldots}_{u}\underbrace{\overset{(t)}{\ldots}a\ldots}_{v}\underbrace{abb\overset{(t)}{\ldots}b}_{w}$$

b) *Si en la cadena* v *aparece algún símbolo* b*, deja de cumplirse la segunda condición, ya que entonces* $\mid uv \mid > t$*, ya que en la subcadena* u *deberían de estar, al menos, todas los* t *símbolos* a*.*

$$\underbrace{a\ldots}_{u}\underbrace{\overset{(t)}{a}\ldots b}_{v}\underbrace{\ldots b\overset{(t)}{b}\ldots b}_{w}$$

$$\underbrace{a\overset{(t)}{\ldots}ab}_{u}\underbrace{\ldots b\overset{(t)}{\ldots}\ldots}_{v}\underbrace{\ldots b}_{w}$$

Como no hay más opciones posibles para la asignación de símbolos a la cadena v*, entonces este lenguaje no cumple el lema de bombeo y, por lo tanto, no puede ser un lenguaje regular.*

2. *Demostrar que el lenguaje* $L = \{0^p \mid p\ es\ un\ número\ primo\}$ *no es un lenguaje regular.*

   Este lenguaje está formado por las cadenas de 0's cuya longitud es un número primo. Sea n *la constante del lema de bombeo y sea* $z = 0^k$ *tal que* k *es un número primo mayor que* n*.*

   Como el conjunto de números primos es un conjunto de infinitos elementos se garantiza que ese número primo k *existe, sea cual sea el valor de* n*. Por lo tanto* $z \in L$ *y si L fuese un lenguaje regular entonces deberían de cumplirse las condiciones expuestas por el lema de bombeo, en particular, que*

   $$uv^iw \in L\ \forall i \geq 0.$$

   Sea $i = k + 1$*; la cadena* $uv^{k+1}w$ *debería pertenecer a L. Pero,*

   $$\mid uv^{k+1}w \mid = \mid uv^kvw \mid = \mid uvw \mid + \mid v^k \mid = k + k \mid v \mid = k(1+ \mid v \mid).$$

   Es decir, $\mid uv^{k+1}w \mid$ *no es un número primo, puesto que es divisible por* k *y por* $(1+ \mid v \mid)$*. Como consecuencia, el lenguaje L no es un lenguaje regular puesto que no cumple el lema de bombeo.*

3.3. Propiedades de Clausura.

Existen numerosas operaciones que aplicadas a lenguajes regulares dan como resultado otro lenguaje regular. Por ejemplo, la unión de dos lenguajes regulares es un lenguaje regular puesto que si r_1 y r_2 denotan a los lenguajes regulares L_1 y L_2, entonces la expresión regular $r_1 + r_2$ denota al lenguaje $L_1 \cup L_2$. Este tipo de operaciones reciben el nombre de *operaciones de clausura*.

Teorema 3.3 *Los lenguajes regulares son cerrados bajo las operaciones de unión, concatenación, y estrella de Kleene.*

La demostración del teorema anterior se deja propuesto como ejercicio.

Teorema 3.4 *Los lenguajes regulares son cerrados bajo la operación de complementación.*

Demostración:

Sea $A = \langle Q, \Sigma, q_0, f, F\rangle$ un Autómata Finito Determinista tal que L(A) = L. Se construye un AFD $A' = \langle Q, \Sigma, q_0, f, Q - F\rangle$, es decir, A' es un AFD que se diferencia del AFD A en que en A' serán estados finales los que no lo eran en el AFD A y viceversa.

Por lo tanto,

$L(A') = \{x \in \Sigma^* \mid f(q_0, x) \in Q - F\} = \{x \in \Sigma^* \mid f(q_0, x) \notin F\} = \Sigma^* - \{x \in \Sigma^* \mid f(q_0, x) \in F\} = \Sigma^* - L = \bar{L}$.

c.q.d.

Resulta interesante destacar que la condición de que el autómata finito de partida sea determinista es muy importante, puesto que de no ser así, entonces la demostración no sería correcta.

Por ejemplo, si A fuese un AFN tal que $f(q, a) = \{q_j, q_k\}$ y $q_j \in F$, pero $q_k \notin F$, al construir el AF A' se tendría que $q_k \in Q - F$ y por lo tanto sería un estado final de A'. Y como $f(q, a) = \{q_j, q_k\}$ resulta que la cadena a es reconocida por el AF A y por A'. Esto es imposible si el AF A' reconoce el complementario de L(A).

Teorema 3.5 *Los lenguajes regulares son cerrados bajo la operación de intersección.*

Demostración:

Como los lenguajes regulares son cerrados bajo las operaciones de complementación y de unión, entonces si L_1 y L_2 son lenguajes regulares también lo será $\overline{\overline{L_2} \cup \overline{L_1}} = L_1 \cap L_2$.

c.q.d.

¿Cómo se podría construir un AFD que reconociese la intersección de dos lenguajes regulares L_1 y L_2? Un método consistiría en construir el AFD A_1 que reconociese L_1, y el AFD A_2, que reconociese L_2; a partir de ellos se puede construir el AFD A'_1 que reconoce el complementario de L_1 y, posteriormente, el AFD A'_1 que reconoce el complemento de L_1. Con estos, se puede construir el AFD A'_U para reconocer la unión de $L(A'_1)$ y $L(A'_2)$. Para finalizar, se construye el AFD que reconoce el complementario de $L(A'_U)$. Pero este método resulta muy largo.

Este AFD se puede calcular de forma más sencilla aplicando el siguiente método. Sea $A_1 = \langle Q_1, \Sigma, q_0, f_1, F_1 \rangle$ un Autómata Finito Determinista tal que $L(A_1) = L_1$, y sea $A_2 = \langle Q_{12}, \Sigma, p_0, f_2, F_2 \rangle$ otro Autómata Finito Determinista tal que $L(A_2) = L_2$.

Se construye el siguiente AFD $A'' = \langle Q_1 \times Q_2, \Sigma, [q_0, p_0], f, F_1 \times F_2 \rangle$ tal que la función de transición f se define de la forma siguiente:

$$f([q,p],a) = [f_1(q,a), f_2(p,a)].$$

De esta forma el lenguaje reconocido por el AFD A'' es el siguiente,

$$L(A'') = \{x \in \Sigma^* \mid f([q_0,p_0],x) \in F_1 \times F_2\} =$$

$$\{x \in \Sigma^* \mid f_1(q_0,x) \in F_1 \wedge f_2(p_0,x) \in F_2\} = L_1 \cap L_2.$$

Ejemplo de aplicación del teorema:

Sea $L = \{x \in (0+1)^* \mid S(0,x) = S(1,x)\}$. *¿Es L un lenguaje regular?*

El lenguaje L está formado por cadenas que tienen el mismo número de 0's que de 1's dispuestos en cualquier posición.

Si L fuese un lenguaje regular entonces al intersectarlo con un lenguaje regular debería dar como resultado un lenguaje regular. El lenguaje 0^*1^* *es un lenguaje regular, puesto que es una expresión regular. Por lo tanto si L fuese regular también lo debería ser el lenguaje* $L \cap 0^*1^*$*, pero este lenguaje es el lenguaje* 0^n1^n *que ya se ha visto que no es un lenguaje regular.*

Por lo tanto, L tampoco puede ser regular.

Teorema 3.6 *Los lenguajes regulares son cerrados bajo la operación de sustitución.*

Demostración:

Se define la operación de sustitución $f : \Sigma \rightarrow 2^{\Delta*}$ de forma que $\forall\ a \in \Sigma$, $f(a)$ es un lenguaje regular.

Además, se conoce que si r_1 y r_2 son expresiones regulares entonces también son expresiones regulares las expresiones $r_1 + r_2$, $r_1 r_2$ y r_1^*.

Por lo tanto, como la operación de sustitución de una unión, de una concatenación o de una clausura es la unión, concatenación o clausura de la sustitución y al ser estas operaciones de clausura para lenguajes regulares, resulta que la operación de sustitución también es una operación de clausura.

c.q.d.

Ejemplo:
Sea $\Sigma = \{a, b\}$ *y* $\Delta = \{0, 1, 2\}$ *tal que* $f(a) = r_a = 0^*1^*$ *y* $f(b) = r_b = 0^*2 + 2$.
Sea L *el siguiente lenguaje regular,* $L = a^*(b + ab)^*$. *Calcular* $f(L)$.

$$f(L) = f(a^*(b + ab)^*) = f(a^*)f((b + ab)^*) = f(a)^* f(b + ab)^* = f(a)^*(f(b) + f(ab))^* = f(a)^*(f(b) + f(a)f(b))^* = (0^*1^*)((0^*2 + 2) + (0^*1^*)(0^*2 + 2))^*.$$

Como consecuencia de este teorema, y como la operación de homomorfismo es un caso particular de la operación de sustitución, resulta el siguiente corolario.

Corolario 3.1 *Los lenguajes regulares son cerrados bajo la operación de homomorfismo.*

Teorema 3.7 *Los lenguajes regulares son cerrados bajo la operación de homomorfismo inverso.*

Demostración:

Sea $h : \Sigma \rightarrow \Delta^*$. Sea L un lenguaje regular de Δ^*. Se demostrará que $h^{-1}(L)$ es otro lenguaje regular. El método consiste en construir un AFD, A', que reconozca $h^{-1}(L)$ a partir del AFD que reconozca L, A.

Sea $A = \langle \Delta, Q, f, q_0, F \rangle$ un AFD tal que $L(A) = L$. Se define el AFD $A' = \langle \Sigma, Q, f', q_0, F \rangle$ tal que la función de transición f' se define como

$$f'(q, a) = f(q, h(a))\ \forall\ q \in Q, \forall\ a \in \Sigma.$$

Aplicando inducción sobre la longitud de la cadena x se puede demostrar que si $f'(q_0, x) = p$, entonces en el AFD A también se cumple que $f(q_0, h(x)) = p$.

Por lo tanto,

$L(A') = \{x \in \Sigma^* \mid f'(q_0, x) \in F\} = \{x \in \Sigma^* \mid f(q_0, h(x)) \in F\} = \{x \in \Sigma^* \mid h(x) \in L\} = h^{-1}(L)$.

c.q.d.

Teorema 3.8 *Los lenguajes regulares son cerrados bajo la operación de inversión.*

Demostración:

Sea $A = \langle \Delta, Q, f, q_0, F\rangle$ un autómata finito arbitrario y sea el autómata finito $A' = \langle \Sigma, Q', f', q_0', F'\rangle$, definido como $Q' = Q \cup \{q_0'\}$ y $F' = \{q_0\}$, tal que $q_0' \notin Q$, es decir, q_0' es un estado nuevo.

La función de transición f' se define en dos pasos:

1. $f'(q_0', \lambda) = F$,
2. $q \in f(p, a) \Leftrightarrow p \in f'(q, a), a \in (\Sigma \cup \{\lambda\})$.

Con esta construcción se garantiza que $L(A') = [L(A)]^{-1}$.

Adicionalmente, hay que comentar que el método de construcción de A' se puede simplificar *si en el conjunto de estados finales F sólo hay un estado*, es decir, $F = \{q_f\}$. Entonces, la construcción del autómata finito A' se puede realizar de esta otra forma:

$$
\begin{array}{l}
Q = Q' \\
q_0' = q_f \\
F' = \{q_0\} \\
q \in f(p, a) \Leftrightarrow p \in f'(q, a), a \in (\Sigma \cup \{\lambda\})
\end{array}
$$

c.q.d.

Ejemplo:

Sea $L = \{x \in 0^m 1^n \mid m \leq n\}$. ¿Es L un lenguaje regular?.

Supóngase que L es regular; entonces, también lo debe de ser $L^{-1} = \{1^n 0^m \mid m \leq n\}$.

Se define el homomorfismo $g : \{0, 1\} \rightarrow \{0, 1\}^*$ tal que $g(0) = 1$ y $g(1) = 0$.

Como la operación de homomorfismo es una operación de clausura para los lenguajes regulares, entonces el lenguaje $g(L^{-1})$ también debe ser un lenguaje

regular . Y, por lo tanto, el lenguaje $L \cap g(L^{-1})$ debe ser regular, pero este lenguaje es

$$L \cap g(L^{-1}) = \{0^m 1^n \mid m \leq n\} \cap \{0^n 1^m \mid m \leq n\} = \{0^n 1^n \mid n \geq 0\}.$$

que no es un lenguaje regular. Se deduce entonces que la suposición inicial, L es un lenguaje regular, no puede ser cierta.

3.4. Algoritmos de Decisión para Lenguajes Regulares.

Un gran número de cuestiones sobre lenguajes regulares se pueden resolver mediante el uso de algoritmos de decisión como, por ejemplo, determinar si un lenguaje regular es vacío, finito o infinito, o bien determinar si una determinada cadena puede ser generada por una determinada gramática regular.

Como se verá en temas siguientes, estas cuestiones no siempre serán resolubles para lenguajes de otros tipos.

Teorema 3.9 *El conjunto de cadenas aceptadas por un autómata finito*

$$A = \langle \Sigma, Q, q_0, f, F \rangle$$

tal que $\mid Q \mid = n$ *es:*

1. *No vacío* $\Leftrightarrow \exists\, x \in L(A) / \mid x \mid < n$.
2. *Infinito* $\Leftrightarrow \exists\, x \in L(A) / n \leq \mid x \mid < 2n$.

Demostración:

1. No vacío $\Leftrightarrow \exists\, x \in L(A) / \mid x \mid < n$.

   "$\Leftarrow$": Obviamente, si existe una cadena que pertenece a L(A), entonces L(A) no es el conjunto vacío.

   "$\Rightarrow$": Si L(A) es no vacío es porque existe al menos una cadena que pertenece a L(A). Sea x esa cadena. Si $\mid x \mid < n$ entonces ya está demostrado el teorema.
   Si $\mid x \mid \geq n$ entonces, por el lema de bombeo, resulta que $x = uvw$, cumpliéndose también que $\forall\, i \geq 0,\ uv^i w \in L(A)$. En particular, si se selecciona $i = 0$, se obtiene la cadena $uv^0 w = uw$ que, como $\mid v \mid \geq 1$, cumple que $\mid x \mid = \mid uvw \mid > \mid uw \mid$ y $uw \in L(A)$. Si $\mid uw \mid < n$, entonces ya se ha conseguido demostrar el teorema. En caso contrario, se vuelve a aplicar este razonamiento.

Como cada vez que se aplica este razonamiento se obtienen cadenas de menor longitud, tiene que llegar un momento en que se obtenga una cadena cuya longitud sea menor que *n*.

Como consecuencia, un posible algoritmo para poder afirmar si el lenguaje que reconoce un AF es o no es vacío, consistiría en determinar si hay alguna cadena de longitud menor que *n*, siendo $\mid Q \mid = n$, que pertenezca al lenguaje (nótese que como *n* es un valor fijo, hay un número finito de posibles cadenas que pudieran pertenecer al lenguaje; por lo tanto, se asegura el fin del proceso).

2. Infinito $\Leftrightarrow \exists\, x \in L(A) / n \leq \mid x \mid < 2n$.

   "$\Leftarrow$": Como $\mid x \mid \geq n$, entonces, por el lema de bombeo, se cumple que $\forall\ i \geq 0, uv^iw \in L(A)$. Por lo tanto, se puede afirmar que hay un número infinito de cadenas que pertenecen a $L(A)$.

   "$\Rightarrow$": Se sigue un proceso de razonamiento similar al realizado en el caso "$\Rightarrow$" del apartado anterior, teniendo en cuenta que $\mid v \mid \geq 1$ y $\mid uv \mid \leq n$.

   Como consecuencia, un posible algoritmo para poder afirmar si el lenguaje que reconoce un AF es o no es infinito, consistiría en determinar si hay alguna cadena de longitud menor que *2n* y menor o igual a *n*, siendo $\mid Q \mid = n$, que pertenezca al lenguaje.

c.q.d.

Teorema 3.10 *Existe un algoritmo para determinar si dos autómatas finitos deterministas reconocen el mismo lenguaje.*

Demostración:

Sea A_1 un AFD $\mid L(A_1) = L_1$ y sea A_2 otro AFD $\mid L(A_2) = L_2$. Se construye un AFD A′ que reconozca el lenguaje,

$$L(A') = (L_1 \cap \overline{L}_2) \cup (\overline{L}_1 \cap L_2).$$

Entonces $L_1 = L_2 \Leftrightarrow L(A') = \emptyset$, que, según el teorema 3.9, es una cuestión decidible.

c.q.d.

Capítulo 4

Gramáticas de Contexto Libre

Índice General

4.1. Gramáticas y Lenguajes de Contexto Libre. 59
4.2. Árboles de Derivación. 60
4.3. Simplificación de GCL. 63
4.4. Formas Normales. 69
4.4.1. Forma Normal de Chomsky, FNCH. 70
4.4.2. Forma Normal de Greibach, FNG. 73

4.1. Gramáticas y Lenguajes de Contexto Libre.

Tal y como se expuso en el tema anterior existen ciertos lenguajes muy usuales que no pueden ser generados por medio de gramáticas regulares. Algunos de estos lenguajes sí pueden ser generados por otro tipo de gramáticas, las denominadas *gramáticas de contexto libre* (también denominadas gramáticas incontextuales, o independientes del contexto). Uno de los ejemplos más típicos consiste en la gramática de contexto libre que genera el lenguaje formado por aquellas cadenas que tengan igual número de paréntesis abiertos que cerrados.

Definición 4.1 *Una Gramática de Contexto Libre (GCL) es una cuádrupla* $G = \langle \Sigma_A, \Sigma_T, P, S \rangle$ *donde*

Σ_A *es un conjunto finito y no vacío de símbolos denominados auxiliares,*

Σ_T *es un conjunto finito de símbolos denominados terminales,* $\Sigma_A \cap \Sigma_T = \emptyset$*,*

S *es el símbolo inicial de la Gramática,* $S \in \Sigma_A$*,*

P *es el conjunto de producciones, donde cada producción es de la forma* $A \rightarrow \beta$ *con* $A \in \Sigma_A$ *y* $\beta \in (\Sigma_T \cup \Sigma_A)^*$.

Un lenguaje es *de tipo 2*, también denominado *lenguaje de contexto libre*, si la gramática más restrictiva que puede generarlo, según la jerarquía de Chomsky, es de tipo 2.

Ejemplos de GCL:

- *El lenguaje* $L_1 = \{0^n 1^n \mid n \geq 0\}$ *puede ser generado por medio de la siguiente GCL*

$$S \rightarrow \lambda \mid 0S1$$

Efectivamente, para comprobar que esta gramática genera el lenguaje L_1 *se realiza un análisis parecido al proceso de demostración por inducción:*

Paso base *la cadena de menor longitud que pertenece al lenguaje* L_1 *es* λ*;*

Hipótesis de inducción *supóngase que desde S se puede generar cualquier cadena de longitud* k *que pertenezca a* L_1*;*

Paso de inducción *la siguiente cadena que pertenece al lenguaje* L_1 *de mayor longitud a la de S es la cadena 0S1.*

- *El lenguaje* $L_2 = \{0^n 1^m \mid n \geq m\}$ *puede ser generado por medio de la siguiente GCL*

$$S \rightarrow \lambda \mid 0S \mid 0S1$$

- *El lenguaje* $L_3 = \{w \in (0+1)^* \mid w = w^{-1}\}$ *puede ser generado por medio de la siguiente GCL*

$$S \rightarrow \lambda \mid 0S0 \mid 1S1 \mid 0 \mid 1$$

4.2. Árboles de Derivación.

Definición 4.2 *Un árbol es un conjunto finito de nodos unidos mediante arcos que forman un grafo acíclico que cumple las tres condiciones siguientes:*

1. *Existe un único nodo denominado* raíz *al que no llega ningún arco.*
2. *Para todo nodo* n *existe una única secuencia de arcos que lleva desde el nodo* raíz *hasta el nodo* n.
3. *En cada nodo, excepto el nodo* raíz*, entra un único arco.*

Sean *n* y *m* dos nodos, se dice que *m es descendiente directo de n* cuando del nodo *n* se puede ir al nodo *m* por un sólo arco. Un nodo *m es descendiente de un nodo n* cuando existe una secuencia de nodos $n_0 n_1 n_2 \ldots n_k$ tal que $n_0 = n, n_k = m$ y se cumple que n_{i+1} es descendiente directo del nodo $n_i, \forall\, i,\ 0 \leq i < k$.

A los nodos que no tienen descendientes directos se les denomina *hojas*.

Definición 4.3 *Dada una GCL $G = \langle \Sigma_A, \Sigma_T, P, S \rangle$, un árbol de derivación en G es un árbol que cumple las siguientes condiciones:*

1. *Cada nodo tiene asignado una etiqueta de $\{\lambda\} \cup \Sigma_A \cup \Sigma_T$,*
2. *La etiqueta del nodo* raíz *es S,*
3. *Si el nodo* n *no es una hoja, entonces la etiqueta del nodo* n *es un símbolo de Σ_A,*
4. *Si $n_1 n_2 \ldots n_k$ son todos descendientes directos de un nodo* n*, ordenados de izquierda a derecha, y si la etiqueta del nodo* n *es el símbolo A y la etiqueta de cada nodo n_i es el símbolo l_i, entonces $(A \to l_1 l_2 \ldots l_k)$ es una producción de G,*
5. *Si el nodo* n *es descendiente directo del nodo n' y la etiqueta del nodo n es λ, entonces* n *es el único descendiente directo del nodo n'.*

El *resultado* de un árbol de derivación es la cadena formada por la concatenación de los símbolos que etiquetan sus hojas tomados de izquierda a derecha.

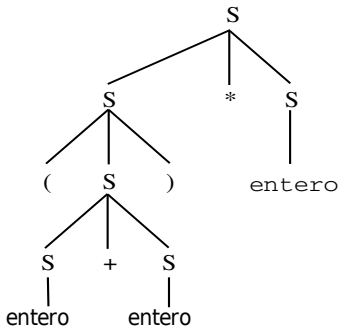
El árbol de derivación contiene todas las posibles derivaciones asociadas a una forma sentencial.

Ejemplo:

Sea G_1 la GCL definida por el siguiente conjunto de producciones

$$\begin{array}{ll} S \to S + S & S \to S \texttt{ DIV } S \\ S \to S - S & S \to S \texttt{ MOD } S \\ S \to S * S & S \to S \uparrow S \\ S \to (S) & S \to \texttt{entero} \end{array}$$

La forma sentencial `(entero + entero) * entero` *es el resultado del siguiente árbol de derivación*



Teorema 4.1 *Sea $G = \langle \Sigma_A, \Sigma_T, P, S \rangle$ una GCL, entonces α es una forma sentencial si, y sólo si, existe un árbol de derivación en la gramática G cuyo resultado es α.*

Se propone como ejercicio demostrar el resultado anterior por inducción en la profundidad del árbol.

Definición 4.4 *Una GCL es ambigua si $\exists\, x \in L(G)$ tal que existen dos o más árboles de derivación diferentes cuyo resultado es x. Un lenguaje de contexto libre es inherentemente ambiguo si toda GCL que lo genere es ambigua.*

Otra cuestión asociada a la derivación de cadenas en una gramática es como seleccionar el símbolo auxiliar a substituir en cada paso de derivación de una forma sentencial. De todas las opciones posibles se destacan dos. Si se selecciona siempre el símbolo auxiliar que se encuentra más a la izquierda entonces se denomina proceso de *derivación a izquierdas*. Si siempre se selecciona el auxiliar más a la derecha, entonces se denomina *derivación a derechas*.

Cada árbol de derivación define una, y sólo una, derivación a izquierdas y una, y sólo una, derivación a derechas.

Siguiendo con el ejemplo anterior, la forma sentencial `(entero + entero) * entero` *se puede obtener mediante la siguiente derivación a izquierdas:*

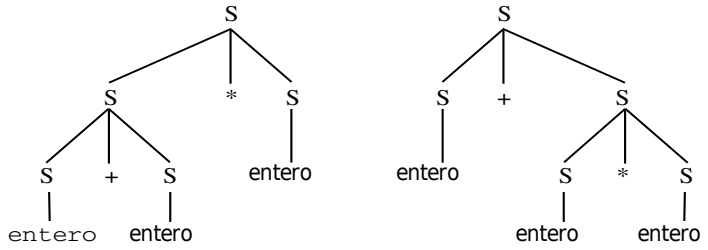
$$\mathbf{S} \Rightarrow \mathbf{S} * S \Rightarrow (\mathbf{S}) * S \Rightarrow (\mathbf{S} + S) * S \Rightarrow (\texttt{entero} + \mathbf{S}) * S \Rightarrow (\texttt{entero} + \texttt{entero}) * \mathbf{S} \Rightarrow (\texttt{entero} + \texttt{entero}) * \texttt{entero},$$

o mediante esta otra, que será una derivación a derechas

$$\mathbf{S} \Rightarrow S * \mathbf{S} \Rightarrow S * \texttt{entero} \Rightarrow (S + \mathbf{S}) * \texttt{entero} \Rightarrow (\mathbf{S} + \texttt{entero}) * \texttt{entero} \Rightarrow (\texttt{entero} + \texttt{entero}) * \texttt{entero}.$$

Esta concepto permite exponer otra definición de gramática ambigua: *Una gramática es ambigua si existe una cadena para la que haya dos o más derivaciones a izquierda o a derecha*.

Siguiendo con el ejemplo, la gramática anterior es ambigua, puesto que para la forma sentencial `entero + entero * entero` *existen los dos siguientes árboles de derivación,*



y las dos derivaciones a izquierdas,

D.I. 1: **S** ⇒ **S** * *S* ⇒ **S** + *S* * *S* ⇒ `entero` + **S** * *S* ⇒ `entero + entero` * **S** ⇒ `entero + entero * entero`,

D.I. 2: **S** ⇒ **S** + *S* ⇒ `entero` + **S** ⇒ `entero` + **S** * *S* ⇒ `entero + entero)` * **S** ⇒ `entero + entero * entero`.

4.3. Simplificación de GCL.

Una de las primeras tareas que hay que realizar con una GCL es eliminar todas aquellas producciones que no aporten ningún tipo de información válida en la generación de alguna de las cadenas del lenguaje. Es decir, hay que simplificar la GCL sin alterar el conjunto de cadenas que es capaz de generar.

Cualquier lenguaje de contexto libre, L, puede ser generado por medio de una GCL, G, que cumpla las siguientes condiciones:

1. Cada símbolo (terminal o auxiliar) de G se emplea en la derivación de alguna cadena de L.
2. En el conjunto de producciones de G no existen *producciones unitarias*, es decir, producciones de la forma $A \rightarrow B$ donde $A, B \in \Sigma_A$.
3. Si $\lambda \notin L$ entonces en el conjunto de producciones de G no existen *producciones vacías*, es decir, producciones de la forma $A \rightarrow \lambda$.

El objetivo de estas condiciones es determinar una GCL en la cual, en cada paso de derivación de una cadena, siempre se introduce información relevante.

Dada una GCL, G, se puede construir una GCL, G', tal que $L(G) = L(G')$ y en G' no hay símbolos inútiles, es decir, que cumple la primera de las condiciones establecidas previamente.

Para construir esta GCL G' se aplican dos lemas. El primero de ellos determina el conjunto de símbolos a partir de los cuales se puede obtener una cadena del lenguaje; el segundo lema determina el conjunto de símbolos que pueden aparecer en una forma sentencial de la

gramática, es decir, los símbolos que pueden ser alcanzados desde el símbolo inicial de la gramática.

Lema 4.1 (de la derivabilidad) *Dada una GCL,* $G = \langle \Sigma_A, \Sigma_T, P, S \rangle$, $L(G) \neq \emptyset$, *puede construirse una GCL equivalente* $G' = \langle \Sigma'_A, \Sigma_T, P', S \rangle$ *en la que se cumple que* $\forall\ A \in \Sigma'_A\ \exists\ w \in \Sigma_T^*$ *tal que* $A \stackrel{*}{\Rightarrow} w$.

Demostración:

Para construir la GCL G′ se calcula el conjunto de símbolos derivables por medio de la siguiente fórmula:

Paso Base: $\forall\ A \in \Sigma_A,\ \forall\ w \in \Sigma_T^*$ tal que $A \rightarrow w \in P$, entonces se sabe que A es derivable.

Paso Recursivo: Si $(A \rightarrow \alpha) \in P$ y si todos los símbolos auxiliares de α son derivables, entonces el símbolo A también es derivable.

El siguiente algoritmo iterativo calcula la fórmula anterior:

$\Sigma_{Aaux} = \emptyset$
$\Sigma'_A = \{\mathtt{A} \in \Sigma_A \mid \exists\ w \in \Sigma_T^* : (\mathtt{A} \rightarrow w) \in \mathtt{P}\}$
`while` $(\Sigma_{Aaux} \neq \Sigma'_A)$`:`
    $\Sigma_{Aaux} = \Sigma'_A$
    $\Sigma'_A = \Sigma_{Aaux} \cup \{\mathtt{A} \in \Sigma_A \mid \exists\ \alpha \in (\Sigma_T \cup \Sigma_{Aaux})^* : (\mathtt{A} \rightarrow \alpha) \in \mathtt{P}\}$

En este algoritmo se inicializa el conjunto de símbolos derivables de acuerdo al paso base de la fórmula anterior y, posteriormente, se calcula el paso recursivo de forma iterativa hasta que no se detecten nuevos símbolos derivables en el conjunto Σ'_A.

Una vez obtenido Σ'_A se define *P'* como el siguiente conjunto

$$P' = \{(A \rightarrow \alpha) \in P \mid A \in \Sigma'_A \wedge \alpha \in (\Sigma_T \cup \Sigma'_A)^*\}$$

Para finalizar, hay que demostrar que L(G′) = L(G). Para ello se demostrará que $L(G') \subseteq L(G)$ y que $L(G) \subseteq L(G')$.

$L(G') \subseteq L(G)$ **:** La construcción de G′ implica que $\Sigma'_A \subseteq \Sigma_A$ y $P' \subseteq P$, por lo tanto si $S \stackrel{*}{\Rightarrow} x$ en G′, entonces aplicando las mismas producciones en G se obtiene la misma cadena *x*.

$L(G) \subseteq L(G')$ **:** Supóngase que $\exists\ x \mid x \in L(G) \wedge x \notin L(G')$. Si esto es así es porque en la obtención de *x* se utiliza algún símbolo auxiliar de Σ_A que no está en Σ'_A, pero

esto no puede ser puesto que los símbolos de $\Sigma_A - \Sigma'_A$ no son derivables. Entonces, x debe pertenecer a L(G′).

c.q.d.

Lema 4.2 (de la alcanzabilidad) *Dada una GCL,* $G = \langle \Sigma_A, \Sigma_T, P, S \rangle$ *tal que* $L(G) \neq \emptyset$*, puede construirse una GCL equivalente* $G' = \langle \Sigma'_A, \Sigma'_T, P', S \rangle$ *tal que* $\forall\, A \in (\Sigma'_A \cup \Sigma'_T)\ \exists\, \alpha, \beta \in (\Sigma'_A \cup \Sigma'_T)^* \mid S \stackrel{*}{\Rightarrow} \alpha A \beta$.

Demostración:

Para construir la GCL G′ se calcula el conjunto de símbolos alcanzables por medio del proceso iterativo:

- $\Sigma'_A = \{S\}, \Sigma'_T = \emptyset$,
- $\forall\ (A \rightarrow \alpha) \in P$: si $A \in \Sigma'_A$, todos los símbolos auxiliares de α se añaden a Σ'_A y todos sus símbolos terminales se añaden a Σ'_T.

Se define P' como el siguiente conjunto

$$P' = \{(A \rightarrow \alpha) \in P \mid A \in \Sigma'_A \wedge \alpha \in (\Sigma'_T \cup \Sigma'_A)^*\}$$

Para finalizar, hay que demostrar que L(G′) = L(G). Queda propuesto como ejercicio.

c.q.d.

Teorema 4.2 *Todo LCL L,* $L \neq \emptyset$*, puede ser generado por una GCL sin símbolos inútiles.*

Demostración:

Sea G una GCL que genera el LCL L.

Sea G′ la GCL resultado de aplicar a la GCL G el lema 4.1 (al eliminar los símbolos no derivables).

Sea G′′ la GCL resultado de aplicar a la GCL G′ el lema 4.2 (al eliminar los símbolos no alcanzables). Entonces G′′ no contiene símbolos inútiles y L(G′′) = L(G′) = L(G).

Para demostrar que G′′ no contiene símbolos inútiles supóngase que contiene un símbolo inútil X. Como $X \in \Sigma'_A$ entonces X es derivable, y como $X \in \Sigma''_A$ entonces X es derivable y alcanzable, por lo tanto, $S \stackrel{*}{\Rightarrow} \alpha X \beta \stackrel{*}{\Rightarrow} w \in \Sigma_T^*$, y entonces X es útil, lo que contradice la suposición de que en G′′ había un símbolo inútil. Por lo tanto, en G′′ no hay símbolos inútiles.

c.q.d.

La aplicación de los lemas a cada GCL debe realizarse según el orden indicado, puesto que en caso contrario se podría obtener una GCL con símbolos inútiles.

Ejemplo:
Sea G la GCL definida por medio de las siguientes producciones:

$$\{S \to aABC \mid a; A \to a; B \to b; E \to b\}.$$

Si se aplica primero el lema 4.1 se obtiene la GCL

$$\{S \to a; A \to a; B \to b; E \to b\},$$

y al aplicar a continuación el lema 4.2 se obtiene la GCL

$$\{S \to a\}.$$

Si a G se le aplicase primero el lema 4.2 se obtendría la GCL siguiente

$$\{S \to aABC \mid a; A \to a; B \to b\},$$

y al aplicar posteriormente el lema 4.1 la GCL resultante sería la siguiente

$$\{S \to a; A \to a; B \to b\}$$

que contiene los símbolos inútiles {A, B}.

El siguiente paso para simplificar una GCL consiste en eliminar los símbolos anulables.

Si $A \to \lambda \in P$, entonces la aparición del símbolo A en alguna forma sentencial intermedia del proceso de generación de una cadena, x, del lenguaje supone que, más pronto o más tarde, el auxiliar A será sustituido por λ, y que, por lo tanto, dicho símbolo no habrá reportado ninguna utilidad en el proceso de generación de la cadena x.

Teorema 4.3 *Para toda GCL G | L(G) = L, existe una GCL G′ tal que L(G′) = L(G) - $\{\lambda\}$ y G′ no contiene símbolos inútiles ni anulables.*

Demostración:

Sea $G = \langle \Sigma_A, \Sigma_T, P, S \rangle$ una GCL. El primer paso consiste en calcular los símbolos anulables, es decir, el conjunto $\mathcal{A} = \{X \in \Sigma_A \mid X \stackrel{*}{\Rightarrow} \lambda\}$. Este conjunto se puede calcular por medio del siguiente proceso iterativo:

- $A_1 = \{X \in \Sigma_A \mid X \to \lambda) \in P\}$,

- $A_{i+1} = A_i \cup \{X \in \Sigma_A \mid \exists\, \alpha \in A_i^+ : (X \to \alpha) \in P\}, \forall\, i \geq 1.$

El proceso finaliza cuando para algún valor de *k* se cumple que $A_{k+1} = A_k$.

Sea $G_2 = \langle \Sigma_A, \Sigma_T, P_2, S \rangle$ la GCL tal que el conjunto de producciones P_2 se construye a partir de P y del conjunto A de la siguiente forma:

$(X \to \beta) \in P_2 \Leftrightarrow \beta \neq \lambda \wedge \exists\, \alpha \in (\Sigma_A \cup \Sigma_T)^+ : (X \to \alpha) \in P \ \wedge \beta$ se puede obtener de α eliminando ninguna, una o más apariciones de ninguno, uno o más símbolos del conjunto de símbolos anulables $\mathcal{A}$.

Falta por demostrar que $L(G_2) = L(G) - \{\lambda\}$. Para ello se demostrará la inclusión en ambos sentidos.

$L(G_2) \subseteq L(G) - \{\lambda\}$ **:** Cada producción de P_2 se ha obtenido realizando una secuencia de derivaciones con producciones de P; por lo tanto, si $X \to \gamma \in P_2$ entonces $X \overset{*}{\Rightarrow} \gamma$ aplicando producciones de P.

Sea $w \in L(G_2)$, luego $S \Rightarrow \theta_1 \Rightarrow \theta_2 \Rightarrow \ldots \Rightarrow w \neq \lambda$ aplicando producciones de P_2. Cada una de esas derivaciones $\theta_i \Rightarrow \theta_{i+1}$ puede realizarse en uno o más pasos en la gramática G, y por lo tanto, se puede obtener la misma cadena *w*: $S \overset{*}{\Rightarrow} \theta_1 \overset{*}{\Rightarrow} \theta_2 \overset{*}{\Rightarrow} \ldots \overset{*}{\Rightarrow} w \neq \lambda$.

$L(G) - \{\lambda\} \subseteq L(G_2)$ **:** Sea $w \in L(G), w \neq \lambda$, luego $S \Rightarrow \theta_1 \Rightarrow \theta_2 \Rightarrow \ldots \Rightarrow w$. Supóngase que $\theta_i \overset{*}{\Rightarrow} \theta_j$ aplicando únicamente producciones λ.

Para obtener P_2 lo que se ha hecho ha sido eliminar cero o más apariciones de cero o más símbolos anulables de la forma sentencial θ_i, obteniendo la forma sentencial θ_j. Así pues, en P_2 existe una producción que permite realizar esa transformación directamente, y entonces en G_2 se puede derivar directamente θ_j de θ_i, $\theta_i \Rightarrow \theta_j$.

Una vez obtenida G_2 se le aplica el teorema 4.2 para eliminar los símbolos inútiles[1], obteniéndose la GCL buscada.

c.q.d.

Ejemplo:

Sea G la GCL dada por el siguiente conjunto de producciones:

$$\begin{aligned} S &\to ABC \mid Da \\ A &\to aAb \mid \lambda \\ B &\to bBc \mid \lambda \\ C &\to cCa \mid ca \\ D &\to AB \mid a \end{aligned}$$

[1] Si bien en la práctica suele aplicarse este paso primero, se debe tener en cuenta que tras eliminar las transiciones λ es posible que vuelvan a aparecer símbolos inútiles.

El conjunto de símbolos anulables, $\mathcal{A}$, se calcula de acuerdo al método indicado:

$$A_1 = \{A, B\},\ A_2 = \{A, B, D\},\ A_3 = A_2 = \mathcal{A}$$

El conjunto de producciones P_2 será el siguiente:

$$\begin{aligned}
&S \to ABC \mid AC \mid BC \mid C \mid Da \mid a\\
&A \to aAb \mid ab\\
&B \to bBc \mid bc\\
&C \to cCa \mid ca\\
&D \to AB \mid A \mid B \mid a
\end{aligned}$$

Corolario 4.1 *Dada una GCL G puede construirse una GCL G′ equivalente a G tal que no tenga producciones λ excepto cuando $\lambda \in L(G)$ en cuyo caso $S' \to \lambda$ es la única producción en la que aparece λ y además S′ no aparece en el consecuente de ninguna otra regla de producción.*

Efectivamente, si $G = \langle \Sigma_A, \Sigma_T, P, S\rangle$ es una GCL, tal que $\lambda \in L(G)$, entonces se construye la gramática $G_2 = \langle \Sigma_A, \Sigma_T, P_2, S\rangle$ siguiendo el método expuesto en la demostración del teorema 4.3.

La relación entre G y G_2 es $L(G_2) = L(G) - \{\lambda\}$. Para construir la gramática G′ del corolario se define G′ como $\langle \Sigma'_A, \Sigma_T, P', S'\rangle$ tal que $\Sigma'_A = \Sigma_A \cup \{S'\},\ S' \in \Sigma_A,\ y\ P' = P_2 \cup \{S' \to \lambda \mid S\}$.

El último paso para simplificar una GCL consiste en eliminar las producciones unitarias, es decir aquellas producciones de la forma $A \to B$, ya que cuando aparecen en una derivación lo único que introducen es un cambio de nombre del auxiliar.

Teorema 4.4 *Todo LCL L, tal que $\lambda \notin L$, se puede generar por una GCL G que no contiene producciones unitarias ni símbolos inútiles ni anulables.*

Demostración:

Sea $G = \langle \Sigma_A, \Sigma_T, P, S\rangle$ una GCL tal que L(G) = L y en G no existen producciones λ. El teorema 4.3 asegura que existe. El primer paso para obtener la gramática sin producciones unitarias, consiste en calcular los símbolos que aparecen en dichas producciones, es decir, los conjuntos $U(X) = \{Y \in \Sigma_A \mid Y \overset{*}{\Rightarrow} X\}, \forall\, X \in \Sigma_A$.

Estos conjuntos se pueden calcular por medio del siguiente proceso iterativo:

- $U_1(X) = \{X\}$
- $U_{i+1}(X) = U_i(X) \cup \{Z \in \Sigma_A \mid \exists\, Y \in U_i(X) : (Z \to Y) \in P\}, \forall\, i \geq 2.$

En el cálculo anterior, la condición de parada es $\exists\, k \mid U_{k+1}(X) = U_k(X)$.

Sea $G_1 = \langle \Sigma_A, \Sigma_T, P_1, S \rangle$ la GCL tal que el conjunto de producciones P_1 se construye a partir de P y del conjunto U(X) de la siguiente forma:

$$(Z \to \alpha) \in P_1 \Leftrightarrow \alpha \notin \Sigma_A \wedge \exists\, X \in \Sigma_A \mid Z \in U(X) \wedge (X \to \alpha) \in P.$$

Resta por demostrar que $L(G_1) = L(G)$, comprobando la doble inclusión.

$L(G_1) \subseteq L(G)$ **:** Cada producción de P_1 se ha obtenido realizando una secuencia de derivaciones con producciones de P; por lo tanto, si $X \to \beta \in P_1$ entonces $X \overset{*}{\Rightarrow} \beta$ aplicando producciones de P.

Sea $w \in L(G_1)$, es decir, $S \Rightarrow \theta_1 \Rightarrow \theta_2 \Rightarrow \ldots \Rightarrow w$, aplicando producciones de P_1; entonces, en la gramática G se puede realizar la siguiente derivación: $S \overset{*}{\Rightarrow} \theta_1 \overset{*}{\Rightarrow} \theta_2 \overset{*}{\Rightarrow} \ldots \overset{*}{\Rightarrow} w$. Por lo tanto, $w \in L(G)$.

$L(G) \subseteq L(G_1)$ **:** Sea $w \in L(G)$, y sea la secuencia de producciones que la producen la siguiente, $S \Rightarrow \theta_1 \Rightarrow \theta_2 \Rightarrow \ldots \Rightarrow w$.

Supóngase que hasta θ_i no se aplican producciones unitarias, y que $\theta_i = \alpha X_i \beta \Rightarrow \theta_{i+1} = \alpha X_{i+1} \beta \overset{*}{\Rightarrow} \theta_{j-1} = \alpha X_{j-1} \beta$ aplicando únicamente producciones unitarias y que de $\theta_{j-1} = \alpha X_{j-1} \beta \Rightarrow \theta_j = \alpha \gamma \beta$ por medio de una producción no unitaria, $X_{j-1} \to \gamma$.

Lo que se ha hecho ha sido sustituir en cada derivación un símbolo auxiliar, X_i, por otro símbolo auxiliar, X_{i+1}, hasta llegar a la forma sentencial $\theta_{j-1} = \alpha X_{j-1} \beta$ y, posteriormente, aplicar la producción $X_{j-1} \to \gamma$.

En P_1 existe una producción que permite realizar esa transformación directamente, $X_i \to \gamma$; entonces en G_1 se puede derivar directamente la forma sentencial θ_j desde la forma sentencial θ_i, $\theta_i \Rightarrow \theta_j$.

Una vez obtenida G_1 se eliminan los símbolos inútiles[2] obteniéndose la GCL buscada.

c.q.d.

4.4. Formas Normales.

Una forma normal para una clase de lenguajes es un formato en el que se pueden escribir las producciones de una gramática que genere un lenguaje de esa clase.

De entre las distintas (y múltiples) formas normales existentes, en este tema se estudiarán dos de las más útiles: la forma normal de Chomsky (FNCH) y la forma normal de Greibach (FNG).

[2] Recuérdese lo comentado en el pie de página 1.

4.4.1. Forma Normal de Chomsky, FNCH.

Teorema 4.5 (Formal Normal de Chomsky) *Todo LCL L, $\lambda \notin L$, se puede generar por una GCL en la que todas sus producciones tienen el siguiente formato:*

$$A \to BC$$
$$A \to a$$

donde A, B, C son símbolos auxiliares y a es un símbolo terminal.

Demostración:

Sea L un LCL, $\lambda \notin L$, y sea $G = \langle \Sigma_A, \Sigma_T, P, S \rangle$ una GCL tal que L(G) = L y en G no existan producciones unitarias, símbolos inútiles ni producciones λ.

El primer paso para conseguir una GCL equivalente a G en la que todas sus producciones estén en forma normal de Chomsky consiste en conseguir que los consecuentes de todas las producciones de longitud mayor que dos estén compuestos exclusivamente por símbolos auxiliares.

Para ello se define el conjunto $C = \{C_a \mid a \in \Sigma_T\} : \ C \cap \Sigma_A = \emptyset$ (es decir, los elementos de C son todos símbolos que no pertenecen a Σ_A).

Sea $f : \Sigma_A \cup \Sigma_T \longrightarrow \Sigma_A \cup C$ la función definida de la forma siguiente:

$$f(X) = X, \ si \ X \in \Sigma_A$$
$$f(X) = C_X, \ si \ X \in \Sigma_T$$

Sea $G' = \langle \Sigma'_A, \Sigma_T, P', S \rangle$ la GCL en la que

$$\begin{aligned} \Sigma'_A &= \Sigma_A \cup C \\ P' = \ & \{A \to f(\alpha) \mid (A \to \alpha) \in P \wedge \mid \alpha \mid \geq 2\} \cup \\ & \{A \to a \in P \mid a \in \Sigma_T\} \cup \\ & \{C_a \to a \mid C_a \in C\} \end{aligned}$$

Con esta construcción se garantiza que L(G′) = L(G).

$L(G') \subseteq L(G)$ **:** Por inducción en el número de pasos que intervienen en la derivación de una cadena, se demostrará que si $A \stackrel{*}{\Rightarrow} w$ en la GCL G′ (donde $A \in \Sigma_A$ y $w \in \Sigma_T^*$), entonces en G se puede derivar $A \stackrel{*}{\Rightarrow} w$.

Paso Base (n = 1): si en G′ se obtiene $A \Rightarrow w$ en un paso es porque $A \to w \in P'$ y como $A \in \Sigma_A$, entonces $w \in \Sigma_T$, y, por lo tanto, $(A \to w) \in P$.

Hipótesis de inducción: Si $A \stackrel{k}{\Rightarrow} w$ en la gramática G′, entonces $A \stackrel{*}{\Rightarrow} w$ en la GCL G.

Paso de inducción: Sea $A \overset{k+1}{\Rightarrow} w$ una derivación en G′ de *k+1* pasos.

El primer paso debe de ser de la forma $A \to B_1 B_2 \dots B_m,\ m \geq 2$, y entonces se puede escribir $w = w_1 w_2 \dots w_m$, donde $B_i \overset{*}{\Rightarrow} w_i$. Si $B_i = C_a$ entonces $w_i = a$.

Por la construcción de P′, existe una producción $A \to X_1 X_2 \dots X_m$ de P en la que $X_i = B_i,\ si\ B_i \in \Sigma_A\ y\ X_i = a,\ si\ B_i \in C$.

Para aquellos $B_i \in \Sigma_A$, se sabe que la derivación $B_i \overset{*}{\Rightarrow} w_i$ no lleva más de *k* pasos, y, por hipótesis de inducción, en la GCL G se puede derivar $X_i \overset{*}{\Rightarrow} w_i$. Por lo que $A \overset{*}{\Rightarrow} w$.

$L(G) \subseteq L(G')$ **:** Por construcción de P′, si $A \to \alpha$ es cualquier producción de G, entonces en G′ se puede derivar $A \Rightarrow f(\alpha) \overset{*}{\Rightarrow} \alpha$.

Todas las producciones de la GCL G′ tienen el siguiente formato: $A \to \beta$, donde β está compuesta únicamente por símbolos auxiliares y tiene una longitud mayor o igual que dos, o bien $A \to \alpha,\ \alpha \in \Sigma_T$.

Por lo tanto, para obtener la GCL en Forma Normal de Chomsky, a la que se denominará G″, $G'' = \langle \Sigma'_A \cup \Sigma''_A, \Sigma_T, P'', S \rangle$, basta con realizar el siguiente proceso:

- $\forall\ (A \to \alpha)\ |\ |\ \alpha\ | \leq 2$, entonces $(A \to \alpha) \in P''$.
- $\forall\ (A \to A_1 A_2 \dots A_n) \in P'$ tal que $n > 2$, entonces $\{A \to A_1 D_1\ ;\ D_1 \to A_2 D_2\ ;\ \dots\ ;\ D_{n-2} \to A_{n-1} A_n\} \in P'' \wedge D_1, D_2, \dots, D_{n-2} \in \Sigma''_A$, donde $D_1, D_2, \dots, D_{n-2}$ son nuevos símbolos auxiliares.

c.q.d.

Ejemplo:

Calcular una GCL en FNCH equivalente a la GCL G determinada por medio del siguiente conjunto de producciones:

$$
\begin{aligned}
&S \to aAb \\
&A \to aAb \mid aC \mid AB \mid \lambda \\
&B \to bBa \mid \lambda \\
&C \to CD \mid AB \\
&D \to bD \mid \lambda \\
&E \to aEb \mid aE \mid aS
\end{aligned}
$$

El primer paso consiste en eliminar los símbolos inútiles. Para ello se eliminan primero los símbolos no derivables (si bien, en este ejemplo, se obtiene que todos los símbolos son derivables) y todos los símbolos no alcanzables; en este caso se obtiene que el símbolo E es no alcanzable.

La GCL equivalente sin símbolos inútiles es:

$$
\begin{aligned}
&S \to aAb \\
&A \to aAb \mid aC \mid AB \mid \lambda \\
&B \to bBa \mid \lambda \\
&C \to CD \mid AB \\
&D \to bD \mid \lambda
\end{aligned}
$$

El siguiente paso consiste en eliminar las producciones λ. Para ello se calcula el conjunto de símbolos anulables, el conjunto $\mathcal{A}$:

$$A_1 = \{A, B, D\};\ A_2 = \{A, B, C, D\} = A_3 = \mathcal{A}$$

Por lo tanto, como $S \notin \mathcal{A}$ resulta que $\lambda \notin L(G)$ y, por lo tanto, se puede escribir una GCL equivalente sin producciones λ:

$$
\begin{aligned}
&S \to aAb \mid ab \\
&A \to aAb \mid ab \mid aC \mid a \mid AB \mid B \\
&B \to bBa \mid ba \\
&C \to CD \mid D \mid AB \mid A \mid B \\
&D \to bD \mid b
\end{aligned}
$$

Al eliminar las producciones λ no han aparecido símbolos inútiles. El último paso para conseguir una GCL simplificada consiste en eliminar las producciones unitarias.

La GCL G' equivalente a la GCL inicial y sin símbolos inútiles, producciones λ, ni producciones unitarias es:

$$
\begin{aligned}
&S \to aAb \mid ab \\
&A \to aAb \mid ab \mid aC \mid a \mid AB \mid bBa \mid ba \\
&B \to bBa \mid ba \\
&C \to CD \mid bD \mid b \mid AB \mid bBa \mid ba \mid aAb \mid ab \mid aC \mid a \\
&D \to bD \mid b
\end{aligned}
$$

La GCL G'' en Forma Normal de Chomsky tal que L(G'') = L(G') es la siguiente:

$$
\begin{aligned}
&S \to C_aX_1 \mid C_aC_b \\
&A \to C_aX_1 \mid C_aC_b \mid C_aC \mid a \mid AB \mid C_bX_2 \mid C_bC_a \\
&B \to C_bX_2 \mid C_bC_a \\
&C \to CD \mid C_bD \mid b \mid AB \mid C_bX_2 \mid C_bC_a \mid C_aX_1 \mid C_aC_b \mid C_aC \mid a \\
&D \to C_bD \mid b \\
&C_a \to a \\
&C_b \to b \\
&X_1 \to AC_b \\
&X_2 \to BC_a
\end{aligned}
$$

Corolario 4.2 *Dada una GCL G puede construirse una GCL G′ equivalente a G tal que todas sus producciones estén en Forma Normal de Chomsky, excepto cuando $\lambda \in L(G)$, en cuyo caso $S' \to \lambda$ es la única producción en la que aparece λ, S′ no aparece en el consecuente de ninguna otra regla de producción, y el resto de producciones de P′ están en Forma Normal de Chomsky.*

Efectivamente, si $G = \langle \Sigma_A, \Sigma_T, P, S \rangle$ es una GCL, tal que $\lambda \in L(G)$, entonces se construye $G'' = \langle \Sigma''_A, \Sigma_T, P'', S \rangle$ tal que $L(G'') = L(G) - \{\lambda\}$ y G′′ está en Forma Normal de Chomsky. Resulta suficiente con construir la GCL $G' = \langle \Sigma'_A, \Sigma_T, P', S' \rangle$ donde

- $\Sigma'_A = \Sigma''_A \cup \{S'\}$,
- $S' \in \Sigma''_A$ y
- P" $= \{S' \to \lambda\} \cup \{S' \to \gamma \mid (S \to \gamma) \in P''\} \cup P''$.

4.4.2. Forma Normal de Greibach, FNG.

Para poder realizar la transformación de una GCL en otra que cumpla con las condiciones expuestas por la forma normal de Greibach se precisa de unas herramientas básicas que son el *lema de la sustitución* y el de *la eliminación de la recursividad a izquierdas* en una GCL.

Ambos lemas proporcionan métodos que permiten realizar ciertas transformaciones en una GCL, sin que se vea afectado el lenguaje generado por la misma.

Lema 4.3 (de la Sustitución) *Sea $G = \langle \Sigma_A, \Sigma_T, P, S \rangle$ una GCL. Sea $(A \to \alpha_1 B \alpha_2) \in P$ y sea $(B \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_r)$ el conjunto de todas las producciones de G que tienen al símbolo auxiliar B como antecedente.*
Sea $G' = \langle \Sigma_A, \Sigma_T, P', S \rangle$ la GCL tal que P′ se ha obtenido del conjunto P al eliminar de este la producción $(A \to \alpha_1 B \alpha_2)$ y añadir las producciones $(A \to \alpha_1\beta_1\alpha_2 \mid \alpha_1\beta_2\alpha_2 \mid \ldots \mid \alpha_1\beta_r\alpha_2)$, entonces L(G) = L(G′).

Demostración:

Para demostrar que ambas GCL generan el mismo lenguaje, se demostrará la inclusión en los dos sentidos: $L(G) \subseteq L(G')$ y $L(G') \subseteq L(G)$.

$L(G') \subseteq L(G)$ **:** Cada producción de P′ que no sea de P puede obtenerse por medio de dos derivaciones de producciones del conjunto P; es decir, si $A \to \alpha_1\beta_i\alpha_2$ es una producción de P′ que no es de P, entonces se pueden realizar las siguientes derivaciones con producciones de P: $A \Rightarrow \alpha_1 B \alpha_2 \Rightarrow \alpha_1\beta_i\alpha_2$.

$L(G) \subseteq L(G')$ **:** Sea $x \in L(G)$; entonces, si en el proceso de derivación de x se ha utilizado la producción $A \rightarrow \alpha_1 B \alpha_2$, posteriormente se ha debido sustituir el símbolo B por alguno de los consecuentes asociados a las producciones de B, que son los siguientes: $(B \rightarrow \beta_1 \mid \beta_2 \mid \ \ldots \ \mid \beta_r)$, obteniéndose, entonces, $\alpha_1 \beta_i \alpha_2$.

En G′ se puede obtener la misma cadena, x, realizando la misma secuencia de derivación que en G′ hasta llegar a la sustitución del símbolo A. En este momento se sustituye el símbolo A por la producción de P′, $A \rightarrow \alpha_1 \beta_i \alpha_2$.

c.q.d.

Lema 4.4 (de la Eliminación de la Recursividad a Izquierdas) *Sea* $G = \langle \Sigma_A, \Sigma_T, P, S \rangle$ *una GCL y sea* $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ \ldots \ \mid A\alpha_r$ *el conjunto de todas las producciones de P que tienen al símbolo A como antecedente y que son recursivas a izquierdas. Sea* $A \rightarrow \beta_1 \mid \beta_2 \mid \ \ldots \ \mid \beta_m$ *el resto de producciones de P que tienen al símbolo A como antecedente.*
Sea $G' = \langle \Sigma'_A, \Sigma_T, P', S \rangle$ *la GCL obtenida de G tal que* $\Sigma'_A = \Sigma_A \cup \{B\},\ B \notin \Sigma_A$, *y tal que en P′ se han eliminado todas las producciones que tienen al símbolo A como antecedente y en su lugar se han añadido las siguientes producciones:*

1. $A \rightarrow \beta_i \mid \beta_i B,\ 1 \leq i \leq m$,
2. $B \rightarrow \alpha_i \mid \alpha_i B,\ 1 \leq i \leq r$.

Entonces L(G) = L(G′).

Demostración:

También en este lema, para ver que ambas GCL generan el mismo lenguaje, se demostrará la inclusión en los dos sentidos: $L(G) \subseteq L(G')$ y $L(G') \subseteq L(G)$.

$L(G) \subseteq L(G')$ **:** Sea $x \in L(G)$. Supóngase que en el proceso de derivación de la cadena x no se han utilizado producciones recursivas a izquierdas hasta la forma sentencial θ_i, y que, desde esta forma sentencial hasta la forma sentencial θ_{j-1}, sólo se aplican producciones recursivas a izquierdas.

Desde θ_{j-1} se genera la forma sentencial θ_j utilizando una producción no recursiva a izquierdas.

En resumen, todo este proceso se puede expresar de la forma siguiente, $S \stackrel{*}{\Rightarrow} \theta_i$, y desde esta forma sentencial se realizan las derivaciones recursivas a izquierdas:

$$\theta_i = \gamma A \delta \Rightarrow \gamma A \alpha_i \delta \Rightarrow \gamma A \alpha_{i2} \alpha_{i1} \delta \Rightarrow \ \ldots \ \Rightarrow \gamma A \alpha_{ip} \ldots \alpha_{i2} \alpha_{i1} \delta = \theta_{j-1}.$$

La siguiente derivación que pasa de θ_{j-1} a θ_j no es recursiva a izquierdas:

$$\gamma A \alpha_{ip} \ldots \alpha_{i2} \alpha_{i1} \delta \Rightarrow \gamma \beta_j \alpha_{ip} \ldots \alpha_{i2} \alpha_{i1} \delta = \theta_j.$$

En G′ se puede realizar la misma secuencia de derivaciones hasta generar θ_i (ya que todas las producciones utilizadas, al no ser recursivas a izquierdas, también pertenecen a P′) y, a partir de esta forma sentencial se pueden realizar las siguientes secuencias de derivaciones,

$$\theta_i = \gamma A\delta \Rightarrow \gamma\beta_j B\delta \Rightarrow \gamma\beta_j\alpha_{ip}B\delta \Rightarrow \gamma\beta_j\alpha_{ip}\alpha_{ip-1}B\delta \Rightarrow$$
$$\ldots \Rightarrow \gamma\beta_j\alpha_{ip}\ldots\alpha_{i2}B\delta \Rightarrow \gamma\beta_j\alpha_{ip}\ldots\alpha_{i2}\alpha_{i1}\delta = \theta_j.$$

Por lo tanto, se puede llegar a la misma forma sentencial, y, si desde θ_j se deriva *x* en G, entonces también se puede derivar *x* en G′.

$L(G') \subseteq L(G)$ **:** Esta inclusión se demuestra de forma similar a la inclusión previa. Queda propuesta como ejercicio.

c.q.d.

Teorema 4.6 (Formal Normal de Greibach) *Todo LCL L, $\lambda \notin L$, se puede generar por una GCL en la que todas sus producciones tienen el siguiente formato:*

$$A \rightarrow a\alpha$$

donde A es un símbolo auxiliar, α es una cadena de símbolos auxiliares y a es un símbolo terminal.

Demostración:

Sea $G = \langle \Sigma_A, \Sigma_T, P, S\rangle$ una GCL tal que $\lambda \notin L$.

Sea $G' = \langle \Sigma'_A, \Sigma_T, P', S\rangle$ la GCL en Forma Normal de Chomsky tal que L(G′) = L(G).

El primer paso para conseguir transformar G′ en otra GCL en Forma Normal de Greibach, consiste en establecer un orden arbitrario entre los símbolos auxiliares de G′, $\Sigma'_A = \{A_1, A_2, \ldots, A_m\}$. Este orden se establecerá con la intención de que el mayor número posible de producciones de P′ cumpla con la siguiente propiedad

$$\forall\ A_k, A_j \in \Sigma'_A,\ si\ A_k \rightarrow A_j\beta \in P',\ entonces\ j > k. \tag{4.1}$$

Generalmente, no será posible que todas las producciones de P′ cumplan esta propiedad.

Se observa que al estar G′ en FNCH entonces cualquier producción de P′ susceptible de ser transformada para que cumpla la anterior propiedad (es decir, aquellas producciones de P′ en las que el primer símbolo del consecuente sea un símbolo de Σ'_A), tiene su consecuente formado por dos símbolos auxiliares.

Supóngase que $\forall\ i, 1 \leq i < k$ se cumple que $si\ A_i \rightarrow A_j\beta,\ entonces\ j > i$. La producción asociada al símbolo A_k puede ser de una de estas cuatro formas:

1. $A_k \rightarrow a$,
2. $A_k \rightarrow A_j\beta,\ j > k$,
3. $A_k \rightarrow A_j\beta,\ j < k$,
4. $A_k \rightarrow A_k\beta$.

Las dos primeras situaciones identifican formatos que cumplen con la propiedad (4.1).

La tercera situación no cumple con esta propiedad, por lo tanto, se debería aplicar algún método para que lo cumpliese. Este método consiste en aplicar el *lema de la sustitución* (lema 4.3) tantas veces como sea necesario hasta obtener una producción que, o bien cumpla la propiedad (4.1), o bien esté en el formato 4:

> Sea $A_k \rightarrow A_j\beta$, una producción de la forma 3, es decir $j < k$.
>
> Las producciones asociadas al símbolo A_j deben cumplir con la propiedad (4.1). Sea $A_j \rightarrow A_p\beta_1,\ p > j$; luego, sustituyendo A_j en la producción de A_k se obtiene $A_k \rightarrow A_p\beta_1\beta$.
>
> En esta situación *p* puede ser mayor (en cuyo caso, ya se cumple con la propiedad), menor o igual a *k*.
>
> Supóngase que $p < k$; entonces, las producciones asociadas al símbolo A_p deben cumplir con la propiedad (4.1).
>
> Sea $A_p \rightarrow A_q\beta_2,\ q > p$. Sustituyendo A_p en la producción de A_k se obtiene $A_k \rightarrow A_q\beta_2\beta_1\beta$.
>
> De esta manera, como máximo en *k-1* pasos, se obtendrá una producción que cumpla con la propiedad (4.1) o que esté en el formato 4.

Las producciones de tipo 4 se pueden modificar aplicando el *lema de eliminación de la recursividad a izquierdas* (lema 4.4).

Cuando todas las producciones iniciales cumplan con la propiedad (4.1), entonces el conjunto de producciones contendrá producciones con los siguientes formatos:

1. $A_i \rightarrow A_j\gamma,\ j > i$,
2. $A_i \rightarrow a\gamma,\ a \in \Sigma_T,\ \gamma \in \Sigma_A^*$, ya que G′ estaba inicialmente en FNCH,
3. $B_i \rightarrow \gamma$.

Sea A_m el símbolo auxiliar de mayor peso en la ordenación de los símbolos de Σ'_A.

Las producciones de A_m no pueden ser de la forma 1, ya que no hay ningún símbolo auxiliar superior en el orden a A_m. Tampoco pueden ser de la forma 3, ya que estas producciones son las asociadas a la introducción de nuevos símbolos auxiliares en la eliminación de la recursividad a izquierdas. Por lo tanto, las producciones de A_m son de las forma,

$$A_m \rightarrow a_1\gamma_1 \mid a_2\gamma_2 \mid \ \ldots\ \mid a_t\gamma_t.$$

y ya están en FNG.

Para convertir el resto de producciones asociadas a los símbolos de Σ'_A a la FNG se aplica el siguiente algoritmo:

```
for in range(m-1,0, -1):
     Sea A_i → A_jγ
     # j > i; por lo tanto, las producciones de A_j están en FNG
     Substituir A_j por sus consecuentes en A_i
     # se obtienen producciones en FNG
```

Una vez realizado este proceso sólo resta convertir el formato de las producciones de tipo 3, $B_i \rightarrow \gamma$, a FNG. Para ello, basta con fijarse en que el primer símbolo de γ es un símbolo de Σ'_A y, por lo tanto, realizando su sustitución por los consecuentes de sus producciones asociadas se tiene un nuevo conjunto de producciones en FNG.

c.q.d.

Corolario 4.3 *Dada una GCL G puede construirse una GCL G′ equivalente a G tal que todas sus producciones estén en Forma Normal de Greibach, excepto cuando $\lambda \in L(G)$, en cuyo caso se cumple que*

1. *$S' \rightarrow \lambda$ es la única producción en la que aparece λ,*
2. *S' no aparece en el consecuente de ninguna otra regla de producción, y*
3. *el resto de producciones de P′ están en Forma Normal de Greibach.*

Efectivamente, si $G = \langle \Sigma_A, \Sigma_T, P, S\rangle$ es una GCL, tal que $\lambda \in L(G)$, entonces se construye $G'' = \langle \Sigma''_A, \Sigma_T, P'', S\rangle$ tal que $L(G'') = L(G) - \{\lambda\}$ y G′′ esté en Forma Normal de Greibach. Es suficiente con construir la GCL $G' = \langle \Sigma'_A, \Sigma_T, P', S\rangle$ donde

$$\Sigma'_A = \Sigma''_A \cup \{S'\},\ S' \in \Sigma''_A \text{ y } P' = \{S' \rightarrow \lambda\} \cup \{S' \rightarrow \gamma \mid S \rightarrow \gamma \in P''\} \cup P''.$$

Ejemplo:

Sea G la siguiente GCL:

$$S \rightarrow AS \mid AB$$
$$A \rightarrow BA \mid a$$
$$B \rightarrow AB \mid b$$

Se establece el siguiente orden respecto a los símbolos auxiliares: $B < S < A$.

Con este orden[3] *las producciones de S cumplen con la propiedad (4.1), las del símbolo B también, pero la producción del símbolo A, $A \rightarrow BA$, no la cumple.*

[3]Se ha elegido éste, pero también se hubiera podido optar por el orden S, A, B y, en ese caso, el desarrollo sería similar teniendo en cuenta que son las producciones del símbolo B las que no cumplen la propiedad (4.1) y que sobre ellas habría que aplicar los lemas 4.3 y 4.4.

Por lo tanto, se aplicará tantas veces como sea necesario el lema 4.3, de la sustitución, hasta conseguir una producción que o bien cumpla la propiedad o bien sea recursiva a izquierdas. En este caso si se sustituye B por sus consecuentes, de la producción $A \to BA$, se obtienen las producciones $A \to ABA \mid bA$.

Si, además, se elimina la recursividad a izquierdas que se ha introducido en las producciones del símbolo A, queda el siguiente conjunto de producciones:

$$\begin{aligned} &S \to AS \mid AB \\ &B \to AB \mid b \\ &A \to bA \mid a \mid bAX_1 \mid aX_1 \\ &X_1 \to BA \mid BAX_1 \end{aligned}$$

En esta situación, las producciones del símbolo A ya están en FNG. Ahora hay que realizar un proceso de sustitución en los consecuentes de los símbolos de menor orden al símbolo A.

$$\begin{aligned} &S \to bAS \mid aS \mid bAX_1S \mid aX_1S \mid bAB \mid aB \mid bAX_1B \mid aX_1B \\ &B \to bAB \mid aB \mid bAX_1B \mid aX_1B \mid b \end{aligned}$$

Sólo falta sustituir el primer símbolo del consecuente asociado a cada producción de X_1 para tener toda la gramática en FNG:

$$\begin{aligned} &S \to bAS \mid aS \mid bAX_1S \mid aX_1S \mid bAB \mid aB \mid bAX_1B \mid aX_1B \\ &B \to bAB \mid aB \mid bAX_1B \mid aX_1B \mid b \\ &A \to bA \mid a \mid bAX_1 \mid aX_1 \\ &X_1 \to bABA \mid aBA \mid bAX_1BA \mid aX_1BA \mid bA \mid \\ &\qquad bABAX_1 \mid aBAX_1 \mid bAX_1BAX_1 \mid aX_1BAX_1 \mid bAX_1 \end{aligned}$$

Capítulo 5

Autómatas de Pila

Índice General

5.1. Introducción. 79

5.2. Relación entre los Autómatas de Pila y los LCL. 87

5.1. Introducción.

La tarea de reconocimiento de los LCL se realiza por medio de un tipo de autómata denominado *autómata de pila*, AP.

Definición 5.1 *Un Autómata de Pila es una séptupla*

$$A = \langle \Sigma, Q, \Gamma, f, q_0, Z_0, F \rangle$$

donde

Σ *es el alfabeto de entrada,*

Q *es el conjunto de estados, que es finito y no vacío,*

Γ *es el alfabeto de la pila,*

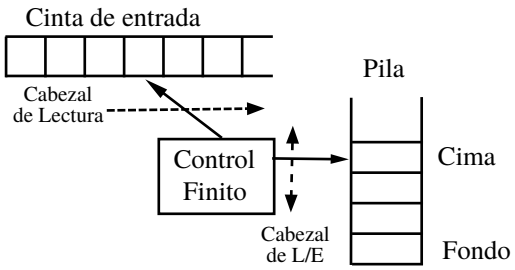
q_0 *es el estado inicial,*

Z_0 *es un símbolo especial, denominado* fondo de pila*,* $Z_0 \in \Gamma$*,*

F *es el conjunto de estados finales,* $F \subseteq Q$*,*

f *es la función de transición,* $f : Q \times (\Sigma \cup \{\lambda\}) \times \Gamma \longrightarrow 2^{Q \times \Gamma^*}$.

Una descripción informal de tal autómata es la representada en la siguiente figura:

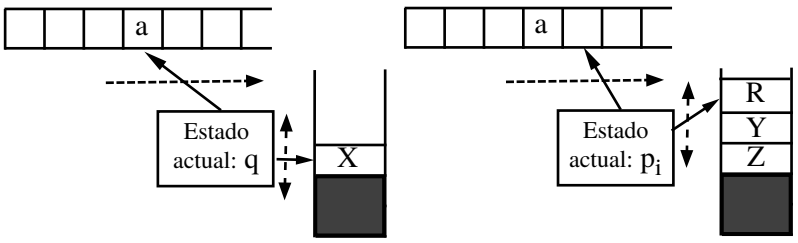


El autómata de pila consta de una cinta de entrada dividida en celdas, cada una de las cuales puede almacenar un sólo símbolo. El acceso a cada celda se realiza por medio del cabezal lector, cuyo movimiento siempre es de izquierda a derecha. El autómata posee un cabezal adicional de lectura/escritura que sólo puede leer o escribir sobre la cima de una pila. Por eso en este autómata se dispone de dos alfabetos de símbolos, el alfabeto de la cinta de entrada, Σ, y el de la pila, Γ.

El funcionamiento de este autómata es el siguiente: dado un *símbolo de la cinta de entrada* o bien la cadena vacía[1] (que equivale a no leer el símbolo que se encuentra bajo el cabezal de lectura), el *estado actual* del autómata (especificado en el control finito) y el símbolo que esté en la *cima de la pila*, entonces este autómata

1. cambia de estado,
2. elimina el símbolo de la cima de la pila y apila ninguno, uno o varios símbolos en la misma, y
3. mueve el cabezal de lectura sobre la cinta de entrada una celda a la derecha, siempre y cuando se hubiese leído un símbolo de la misma.

Es decir, dada la transición $f(q, a, X) = \{(p_1, \gamma_1), (p_2, \gamma_2), \ldots, (p_m, \gamma_m)\}$ el autómata de pila, en el supuesto que se seleccione para ejecución la acción $f(q, a, X) = (p_i, \gamma_i)$, con $\gamma_i = RYZ$, realiza los siguientes cambios:

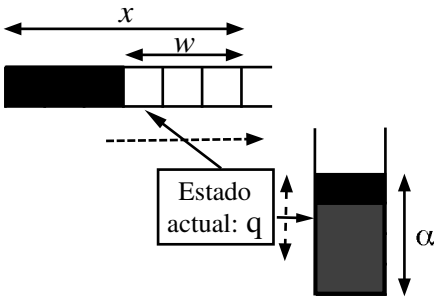


Como se observa en la figura, se transita desde el estado *q* al estado p_i (que pasa a ser el estado actual) el cabezal de lectura se desplaza una celda a la derecha y en la pila desaparece el símbolo *X* de la cima y en su lugar se introducen los símbolos *Z*, *Y* y *R*, en este orden.

[1] Más adelante, se remarcará esta cuestión, pero nótese que esta posibilidad de no leer el símbolo bajo el cabezal hace que el comportamiento de unl autómata de pila sea no determinista por propia definición.

El comportamiento es similar en el caso de ejecutar una transición del tipo $f(q, \lambda, X) = \{..., (p_i, \gamma_i), ...\}$, con la salvedad de que, en este caso, el cabezal de lectura de la cinta *no* se hubiera desplazado hacia la derecha y permanecería sobre el símbolo *a*.

Para poder describir el comportamiento del AP sin necesidad de tener que especificar de forma gráfica su evolución, se suelen utilizar las *descripciones instantáneas*; en el caso de un AP, una descripción instantánea es un elemento del conjunto $Q \times \Sigma^* \times \Gamma^*$. Así, por ejemplo, la descripción instantánea (q, w, α) $(\in Q \times \Sigma^* \times \Gamma^*)$ representa la siguiente situación:



La cadena de entrada es *x*, de la cual falta todavía por analizar el sufijo *w*. El cabezal de lectura está sobre el primer símbolo de *w*, el estado actual del autómata es *q* y el contenido de la pila es α.

Se dice que una descripción instantánea I_1 *alcanza* a otra descripción instantánea I_2 *en un sólo paso*, y se denota como $I_1 \vdash I_2$, cuando se cumplen las siguientes condiciones:

$$\begin{array}{l} I_1 = (q, aw, X\gamma) \wedge \\ I_2 = (p, w, \delta\gamma) \wedge \\ (p, \delta) \in f(q, a, X) \mid \ a \in (\Sigma \cup \{\lambda\}) \wedge X \in \Gamma \end{array}$$

Se dice que una descripción instantánea I_1 *alcanza* a otra descripción instantánea I_2, y se denota como $I_1 \overset{*}{\vdash} I_2$, si $\exists$ *n* descripciones instantáneas auxiliares $DI_1, DI_2, \ldots, DI_n$ tal que a través de ellas se pueda alcanzar I_2 desde I_1, es decir,

$$I_1 = DI_1 \vdash DI_2 \vdash \ldots \vdash DI_n = I_2.$$

Con estas definiciones ya se está en condiciones de poder establecer cuál es el lenguaje aceptado por un AP, en el que hay que diferenciar dos casos.

Definición 5.2 *Se define el* **lenguaje aceptado por estado final** *de un AP A al conjunto*

$$L(A) = \{x \in \Sigma^* \mid (q_0, x, Z_0) \overset{*}{\vdash} (p, \lambda, \gamma) \ tal \ que \ p \in F\}.$$

Se define el **lenguaje aceptado por pila vacía** *de un AP A al conjunto*

$$N(A) = \{x \in \Sigma^* \mid (q_0, x, Z_0) \overset{*}{\vdash} (p, \lambda, \lambda)\}.$$

Se observa que en la definición de N(A), no importa el estado al que se llegue al analizar la cadena x. Es suficiente con que, al finalizar su análisis, la pila quede vacía.

Ejemplo:

Sea A el siguiente AP:

$$A = \langle \{a, b\}, \{q_0\}, \{S, A, B\}, f, q_0, S, \emptyset \rangle$$

donde la función de transición f se define de la forma siguiente:

$$\begin{aligned} f(q_0, a, S) &= \{(q_0, SB), (q_0, ASB)\} \\ f(q_0, \lambda, S) &= \{(q_0, \lambda)\} \\ f(q_0, b, B) &= \{(q_0, \lambda)\} \\ f(q_0, a, A) &= \{(q_0, \lambda)\} \end{aligned}$$

Como se observa, en este AP el conjunto de estados finales es $\emptyset$. Por lo tanto, este AP reconoce el lenguaje $\emptyset$ por el criterio de estado final.

El lenguaje reconocido por este autómata por el criterio de pila vacía es $N(A) = \{a^m b^n \mid m \geq n\}$.

Para comprobar que este AP reconoce la cadena aaabb *se van a especificar todas las posibles transiciones que se pueden realizar con este autómata; el resultado se muestra en la figura 5.1. Como al menos una secuencia de transiciones consigue vaciar la pila y consumir toda la cadena de entrada, entonces la cadena* aaabb *será aceptada.*

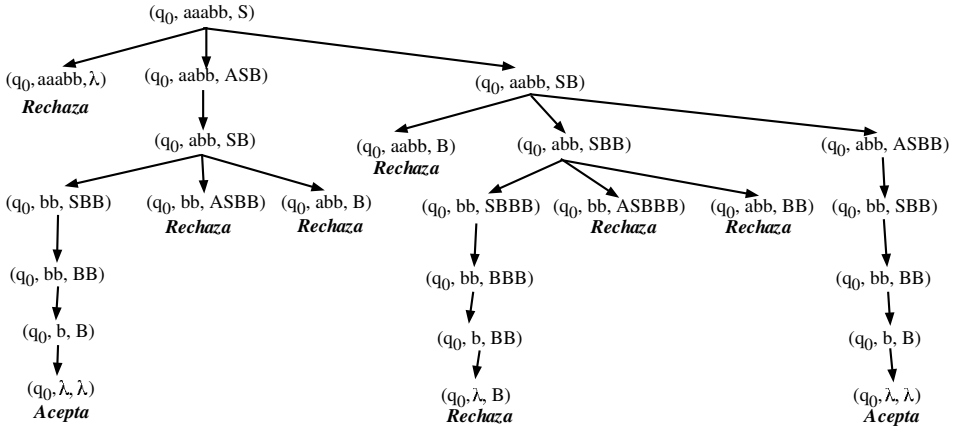


Figura 5.1: Análisis de todas las posibles transiciones que puede realizar el autómata A sobre la cadena *aaabb*.

La definición de un autómata de pila implica un *comportamiento no determinista*, que se pone de manifiesto en el ejemplo anterior.

De forma general, un autómata de pila determinista es un AP en el que es aplicable una, y sólo una, transición en cada instante. Sin embargo, esta posibilidad no basta para excluir el comportamiento no determinista, ya que puede darse la situación de que, desde

un mismo estado y con un mismo símbolo en la cima de la pila, se puedan realizar acciones diferentes si existen transiciones con lectura de símbolo o sin lectura de símbolo, es decir, $f(q, a, A) \neq \emptyset \wedge f(q, \lambda, A) \neq \emptyset$. Esto representa una situación no determinista tal y como se pone de relieve en la figura 5.1 en, por ejemplo, las *tres* posibles transiciones que pueden ejecutarse desde la situación inicial $(q_0, aaabb, S)$.

Por lo tanto, *para que un AP sea determinista* se han de cumplir las siguientes condiciones:

1. $\forall\, q \in Q,\ \forall\, a \in (\Sigma \cup \{\lambda\}),\ \forall\, A \in \Gamma,\ se\ cumple\ que\ \mid f(q, a, A) \mid \leq 1,$
2. $\forall\, q \in Q,\ \forall\, A \in \Gamma,\ si\ f(q, \lambda, A) \neq \emptyset\ entonces\ \forall\, a \in \Sigma,\ f(q, a, A) = \emptyset.$

Ejemplo:

Construir un AP determinista que reconozca el lenguaje $L = \{0^n 1^n \mid n \geq 0\}$.

$$
\begin{aligned}
&f(q_0, \lambda, Z) = \{(q_0, X)\}\\
&f(q_0, 0, X) = \{(q_1, AX)\}\\
&f(q_1, 0, A) = \{(q_1, AA)\}\\
&f(q_1, 1, A) = \{(q_2, \lambda)\}\\
&f(q_2, 1, A) = \{(q_2, \lambda)\}\\
&f(q_2, \lambda, X) = \{(q_0, \lambda)\}
\end{aligned}
$$

El AP construido reconoce el lenguaje L por estado final. Efectivamente, por cada símbolo '0' leído de la cinta de entrada, se introduce en la pila otro símbolo A. En la situación en que se lea el primer '1' se pasa al estado q_2, cuyo fin es eliminar un símbolo A de la pila por cada '1' leído.

Si al acabar de leer la cadena de entrada el AP está en el estado q_0 (que es el único estado final) es porque había el mismo número de '0' que de '1'. Cualquier otra transición que no esté especificada en la anterior función de transición produce que, si esa situación se presenta, entonces el AP se para y rechaza la cadena.

La definición de Autómatas de Pila determinista y no deterministas, permite plantear si el requisito del determinismo reduce el poder reconocedor de un AP.

Teorema 5.1 *Existen LCLs que no son aceptados por ningún AP determinista.*

Demostración

Sea $L = \{a^n b^n : n > 0\} \cup \{a^n b^{2n} : n > 0\}$. Este lenguaje es un LCL pues es generado por la siguiente GCL:

$$
\begin{aligned}
&S \rightarrow aAb \mid aBbb\\
&A \rightarrow aAb \mid \lambda\\
&B \rightarrow aBbb \mid \lambda
\end{aligned}
$$

Para demostrar que L no puede ser reconocido por ningún AP determinista es preciso saber que el lenguaje $\{a^n b^n c^n \mid n > 0\}$ no es un LCL[2].

La demostración se hará por reducción al absurdo: es decir, si se supone que existe un AP determinista que reconoce L, entonces se llega a un resultado imposible.

Sea *A* un AP determinista tal que *L(A) = L*. Se realiza la siguiente construcción:

1. Sean A_1 y A_2 copias del AP *A*. Se dice que dos estados, uno de A_1 y otro de A_2, son *primos entre sí*, si ambos estados son copias del mismo estado del AP original *A*.
2. Eliminar la característica de aceptación de los estados de aceptación de A_1 y la característica de estado inicial del estado inicial de A_2.
3. Modificar el estado destino *p* de cada una de las transiciones que se originan en un antiguo estado de aceptación de A_1, para que el estado destino sea el primo de *p* en A_2.
4. Modificar todas aquellas transiciones que lean una *b* de la entrada y cuyos estados destino se encuentren en A_2 para que lean una *c* en su lugar.

Con esta construcción el conjunto de cadenas aceptado por este AP A' será el formado por aquellas cadenas que comenzando en el estado inicial de A_1 reconocen *n* símbolos *a* y, posteriormente, reconocen *n* símbolos *b*. En este instante, el estado en el que se encuentra el AP es un antiguo estado final de A_1. Sus transiciones se han modificado para que reconozca *n* símbolos *c* (pasando al antiguo AP A_2) en vez de *n* símbolos *b*, y se llega de esta forma, a uno de los estados finales de A_2, que son los únicos estados finales de A'. Por lo tanto, se ha reconocido una cadena de la forma $a^n b^n c^n, n > 0$.

Es decir, se ha podido construir un AP que permite reconocer un lenguaje que no es un LCL. Como la construcción realizada ha sido correcta, el único fallo en la demostración ha sido la suposición de que pueda existir ese AP determinista que reconozca L.

Como consecuencia, el lenguaje L no puede ser reconocido por ningún AP determinista.

c.q.d.

La consecuencia directa de este teorema es que, si se denomina L(APND) al conjunto de lenguajes aceptados por AP no deterministas, mediante el criterio de alcanzabilidad del estado final, y L(APD) al conjunto de lenguajes aceptados por AP deterministas, mediante el criterio de alcanzabilidad del estado final, entonces

$$L(APD) \subset L(APND).$$

[2]Se demostrará en el capítulo 6.

Otra cuestión relacionada con el estudio de los lenguajes aceptados por los AP, consiste en determinar si, dado que a cada AP se le pueden asociar dos lenguajes (el aceptado por *estado final* y el aceptado por *pila vacía*), existe alguna diferencia respecto al conjunto de lenguajes que pueden reconocerse por ambos criterios.

Es decir, si se denomina *N(APND)* al conjunto de lenguajes aceptados por APND mediante el criterio de pila vacía y *L(APND)* al conjunto de lenguajes aceptados por APND mediante el criterio de estado final, ¿cuál es la relación existente entre L(APND) y N(APND)?.

Teorema 5.2 *Si L = L(A) para algún AP A, entonces existe un AP A' tal que $N(A') = L$.*

Demostración:

Sea $A = \langle \Sigma, Q, \Gamma, f, q_0, Z_0, F \rangle$ un AP tal que $L(A) = L$.

Para construir un AP que reconozca L por pila vacía, el método consiste en introducir un nuevo estado, q_e, al que se accede desde cualquiera de los estados de F y cuya única finalidad consiste en vaciar la pila.

Además, se debe evitar la situación en la que en el AP A se vacie la pila, sin que el estado del autómata sea final, ya que esto podría provocar que una cadena no aceptada por A, sí que fuera reconocida por por A'. Para evitar esta situación, se precisa disponer de un nuevo símbolo fondo de pila que sólo se elimine cuando se esté en el nuevo estado q_e; es decir, cuando se haya pasado obligatoriamente por un estado final, habiendo consumido además toda la cadena de entrada.

Sea el AP $A' = \langle \Sigma, Q', \Gamma', f', q_0', X_0, \emptyset \rangle$ tal que

- $Q' = Q \cup \{q_0', q_e\}\ tal\ que\ q_0', q_e \notin Q$,
- $\Gamma' = \Gamma \cup \{X_0\}\ tal\ que\ X_0 \notin \Gamma$,

y en el que las reglas que definen a la función de transición f' son las siguientes:

1. $f'(q_0', \lambda, X_0) = \{(q_0, Z_0 X_0)\}$,
2. $f(q, a, B) \subseteq f'(q, a, B),\ \forall\, q \in Q,\ \forall\, a \in (\Sigma \cup \{\lambda\}),\ \forall\, B \in \Gamma$,
3. $(q_e, \lambda) \in f'(q, \lambda, B),\ \forall\, q \in F,\ \forall\, B \in \Gamma'$,
4. $\{(q_e, \lambda)\} = f'(q_e, \lambda, B),\ \forall\, B \in \Gamma'$.

Hay que demostrar que $L(A) = N(A') = L$. Para ello, se demostrará que $L(A) \subseteq N(A')$ y que $N(A') \subseteq L(A)$.

$L(A) \subseteq N(A')$ **:** Sea $x \in L(A)$; entonces $(q_0, x, Z_0) \overset{*}{\vdash} (p, \lambda, \alpha), p \in F \wedge \alpha \in \Gamma^*$. Por lo tanto, en el AP A' se puede obtener la siguiente secuencia de descripciones instantáneas:

$$(q'_0, x, X_0) \vdash (q_0, x, Z_0X_0) \overset{*}{\vdash} (p, \lambda, \alpha X_0) \overset{*}{\vdash} (q_e, \lambda, \lambda),$$

y, entonces, $x \in N(A')$.

$N(A') \subseteq L(A)$ **:** Sea $x \in N(A')$; entonces desde la descrición instantánea inicial, (q'_0, x, X_0), sólo se puede realizar una única transición a la descrición instantánea (q_0, x, Z_0X_0) y, a partir de esta, todas las transiciones realizadas son también transiciones de A.

Para poder aceptar la cadena *x* el AP A' debe tener la pila vacía y eso sólo se consigue pasando previamente por un estado final de A_2; por lo tanto, de la descrición instantánea (q_0, x, Z_0X_0) se ha de llegar a la descrición instantánea $(p, \lambda, \alpha X_0)$, siendo p un estado final de A, para, posteriormente, vaciar la pila mediante las transiciones de tipo 4.

En el AP A se pueden realizar las mismas transiciones desde (q_0, x, Z_0) hasta (p, λ, α) aceptando, por lo tanto, la cadena *x*.

c.q.d.

Queda demostrado entonces que $L(APND) \subseteq N(APND)$. A continuación se verá que la inclusión también se cumple a la inversa.

Teorema 5.3 *Si $L = N(A)$ para algún AP A, entonces existe un AP A' tal que $L(A') = L$.*

Demostración

Sea $A = \langle \Sigma, Q, \Gamma, f, q_0, Z_0, \emptyset \rangle$ un AP tal que $N(A) = L$.

Para construir un AP que reconozca L por estado final, el método consiste en introducir un nuevo estado final, q'_f, al que se accede cuando en el AP A la pila está vacía.

Para poder determinar cuando se vaciaría la pila, sin que esa situación llegue a producirse, se precisa de un nuevo símbolo fondo de pila que actúe como señal.

Sea el AP $A' = \langle \Sigma, Q', \Gamma', f', q'_0, X_0, F' \rangle$ tal que

- $Q' = Q \cup \{q'_0, q'_f\},\ tal\ que\ q'_0, q'_f \notin Q$,
- $\Gamma' = \Gamma \cup \{X_0\},\ tal\ que\ X_0 \notin \Gamma$,
- $F' = \{q'_f\}$,

y en el que las reglas que definen a la función de transición f' son las siguientes:

1. $f'(q'_0, \lambda, X_0) = \{(q_0, Z_0X_0)\}$,
2. $f'(q, a, B) = f(q, a, B),\ \forall\ q \in Q,\ \forall\ a \in (\Sigma \cup \{\lambda\}),\ \forall\ B \in \Gamma$,
3. $f'(q, \lambda, X_0) = \{(q'_f, \lambda)\},\ \forall\ q \in Q$.

Falta demostrar que $L(A') = N(A) = L$, y para ello se seguirá el método habitual, demostrando que $L(A') \subseteq N(A)$ y que $N(A) \subseteq L(A')$.

$N(A) \subseteq L(A')$ **:** Sea $x \in N(A)$; entonces, $(q_0, x, Z_0) \overset{*}{\vdash} (p, \lambda, \lambda)$. Por lo tanto, en el AP A' se puede obtener la siguiente secuencia de descripciones instantáneas:

$$(q'_0, x, X_0) \vdash (q_0, x, Z_0X_0) \overset{*}{\vdash} (p, \lambda, X_0) \vdash (q'_f, \lambda, \lambda),$$

y, por lo tanto, $x \in L(A')$.

$L(A') \subseteq N(A)$ **:** Sea $x \in L(A')$; entonces, desde la descripción instantánea inicial (q'_0, x, X_0) sólo se puede realizar una única transición a la descripción instantánea (q_0, x, Z_0X_0) y, a partir de esta, todas las transiciones realizadas son también de A.

Para poder aceptar la cadena *x*, el AP A' debe alcanzar un estado final y eso sólo se consigue pasando previamente por una pila cuyo único símbolo sea X_0. Por lo tanto, de la descripción instantánea (q_0, x, Z_0X_0) se ha de llegar a la descripción instantánea (p, λ, X_0), para, posteriormente, pasar al estado final mediante la transición número 3; es decir, se alcanza la descripción instantánea (q'_f, λ, λ).

En el AP A se pueden realizar las mismas transiciones desde (q_0, x, Z_0) hasta (p, λ, λ) aceptando, por lo tanto, la cadena *x*.

c.q.d.

Como consecuencia de ambos teoremas se obtiene que *L(APND) = N(APND)*.

5.2. Relación entre los Autómatas de Pila y los LCL.

En esta sección se establecerán formalmente las relaciones que existen entre los lenguajes de contexto libre y los autómatas de pila no deterministas.

Teorema 5.4 *Si L es un LCL, entonces existe un APND A | L = N(A).*

Demostración

Sea L un LCL, tal que $\lambda \notin L$, entonces existe una GCL $G = \langle \Sigma_A, \Sigma_T, P, S \rangle$ en Forma Normal de Greibach, tal que L(G) = L.

Se construye el siguiente AP, $A = \langle \Sigma, Q, \Gamma, q_0, Z_0, f, \emptyset \rangle$ donde,

$$\begin{array}{l} \Sigma = \Sigma_T \\ Q = \{q_0\} \\ \Gamma = \Sigma_A \\ Z_0 = S \\ (q_0, \gamma) \in f(q_0, a, B) \Leftrightarrow (B \rightarrow a\gamma) \in P \end{array}$$

Resta por demostrar que *N(A) = L*. Para ello, se demostrará que si se emplean siempre derivaciones a izquierda en la GCL G, entonces

$$(S \stackrel{*}{\Rightarrow} w\alpha) \Leftrightarrow ((q_0, w, S) \stackrel{*}{\vdash} (q_0, \lambda, \alpha)),$$

donde $w \in \Sigma_T^* \wedge \alpha \in \Sigma_A^*$.

$((q, w, S) \stackrel{*}{\vdash} (q, \lambda, \alpha)) \Rightarrow (S \stackrel{*}{\Rightarrow} w\alpha)$ **:** Por inducción sobre el número de descripciones instantáneas utilizadas:

Paso Base: Con i = 0, $(q, \lambda, S) \stackrel{0}{\vdash} (q, \lambda, S)$, entonces $S \stackrel{*}{\Rightarrow} S$.

Hipótesis de Inducción: $(q, w, S) \stackrel{n}{\vdash} (q, \lambda, \alpha) \Rightarrow S \stackrel{*}{\Rightarrow} w\alpha$.

Paso de Inducción: Sea $(q, wa, S) \stackrel{n+1}{\vdash} (q, \lambda, \alpha)$ una derivación de *n+1* pasos; entonces, $(q, wa, S) \stackrel{n}{\vdash} (q, a, \beta) = (q, a, B\theta) \vdash (q, \lambda, \alpha)$.
Las primeras *n* derivaciones implican, por H.I., que $S \stackrel{*}{\Rightarrow} w\beta$, y la última derivación sólo se puede realizar si $(q, \gamma) \in f(q, a, B)$.
Por lo tanto, $\exists\, (B \rightarrow a\gamma) \in P$ y, además, $\beta = B\theta$, por lo que se obtiene que,

$$S \stackrel{*}{\Rightarrow} wB\theta \Rightarrow wa\gamma\theta = wa\alpha.$$

$(S \stackrel{*}{\Rightarrow} w\alpha) \Rightarrow ((q, w, S) \stackrel{*}{\vdash} (q, \lambda, \alpha))$ **:** Por inducción sobre el número de pasos de derivación y sabiendo que en la gramática se emplean siempre derivaciones a izquierda:

Paso Base: Con i = 0, $S \stackrel{0}{\Rightarrow} S$, entonces $(q, \lambda, S) \stackrel{*}{\vdash} (q, \lambda, S)$.

Hipótesis de Inducción: $S \stackrel{n}{\Rightarrow} w\alpha \Rightarrow (q, w, S) \stackrel{*}{\vdash} (q, \lambda, \alpha)$.

Paso de Inducción: Sea $S \stackrel{n+1}{\Rightarrow} wa\alpha$ una derivación de *n+1* pasos. Esta derivación se ha obtenido por medio de la siguiente secuencia, $S \stackrel{n}{\Rightarrow} wB\beta \Rightarrow wa\gamma\beta = wa\alpha$.
Las primeras *n* derivaciones implican, por H.I., que $(q, wa, S) \stackrel{n}{\vdash} (q, a, B\beta)$, y la última derivación sólo se puede realizar si $\exists\, (B \rightarrow a\gamma) \in P$, y, por lo tanto, $(q, \gamma) \in f(q, a, B)$.
Entonces,

$$(q, wa, S) \stackrel{n}{\vdash} (q, a, B\beta) \vdash (q, \lambda, \gamma\beta) = (q, \lambda, \alpha).$$

c.q.d.

Por último, queda por establecer el teorema recíproco, cuya demostración se puede encontrar en el libro "*Introduction to Automata Theory, Languages and Computation*" de Hopcroft y Ullman.

Teorema 5.5 *Si L = N(A) para algún AP A, entonces L es un LCL.*

Capítulo 6

Propiedades de los Lenguajes de Contexto Libre

Índice General

6.1. Lema de Bombeo. **91**
6.2. Propiedades de Clausura. **96**
6.3. Algoritmos de Decisión. **103**

6.1. Lema de Bombeo.

Al igual que al estudiar los lenguajes regulares, resulta interesante disponer de ciertas herramientas que permitan determinar cuando un lenguaje es de contexto libre, o bien qué conjunto de operaciones pueden ser aplicadas a un lenguaje de contexto libre para obtener como resultado otro lenguaje de contexto libre.

Una de las herramientas más poderosas para determinar que un lenguaje *no es de contexto libre* consiste en verificar que no cumple con las propiedades establecidas por el *lema de bombeo* para LCL.

Lema 6.1 (Lema de Bombeo) *Para todo LCL L, existe una constante* n*, dependiente únicamente de L, tal que si* z *es una cadena de L,* $\mid z \mid \geq n$*, entonces la cadena* z *se puede descomponer como* z = uvwxy *tal que:*

1. $\mid vx \mid \geq 1$,
2. $\mid vwx \mid \leq n$,
3. *Para todo* $i \geq 0$*, las cadenas* uv^iwx^iy *son todas cadenas de L.*

Demostración:

Sea L un LCL, tal que $\lambda \notin L$; se sabe que entonces existe una GCL en Forma Normal de Chomsky, G, tal que L(G) = L.

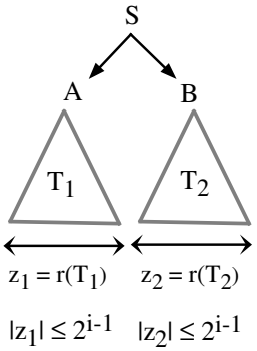
El primer paso en la demostración consiste en determinar que, si un árbol de derivación asociado a una cadena de G no tiene un camino (desde la raíz hasta una hoja) de longitud mayor que *i*, entonces la longitud de la cadena resultado de ese árbol es menor o igual que 2^{i-1}. Esto se puede demostrar por inducción respecto a la longitud del camino asociado a la cadena de L(G).

Paso base: Si la altura del árbol es 1, entonces sólo hay una posibilidad (debido a que todas las producciones de G están en FNCH) y es que ese árbol sea el siguiente :

Por lo tanto, $\mid a \mid = 1 \leq 2^{1-1} = 2^0 = 1$.

Hipótesis de Inducción: El enunciado es cierto para caminos de longitud *i*.

Paso de Inducción: ¿Qué sucede para caminos de longitud *i+1*? En este caso, la primera derivación realizada a partir de S debió ser de la forma $S \rightarrow AB$ (ya que todas las producciones de G están en FNCH) y, entonces, en cada uno de los dos nodos del primer nivel cuelgan subárboles de derivación de altura máxima *i*. Por lo tanto, por hipótesis de inducción, cada uno de estos árboles tiene como resultado una cadena de longitud máxima 2^{i-1}.



El resultado del árbol cuya raíz es S, tiene como resultado una cadena que será la concatenación de las cadenas generadas por esos dos subárboles, y, por lo tanto, su longitud máxima será $2^{i-1} + 2^{i-1} = 2^i$.

Una vez obtenido este resultado intermedio, considérese la GCL G, en FNCH, con *k* símbolos auxiliares, $\mid \Sigma_A \mid = k$, y sea $n = 2^k$.

Sea z una cadena de L(G) cuya longitud es mayor o igual que *n*; entonces, como $\mid z \mid > 2^{k-1}$, cualquier árbol de derivación para z debe tener, como mínimo, una longitud *k+1*.

Este camino tiene como mínimo *k+2* nodos, de los cuales todos, excepto el último, están etiquetados por símbolos auxiliares, por lo que, como sólo existen k símbolos auxiliares distintos, al menos un símbolo auxiliar debe aparecer dos veces en el camino.

Es decir, si P es uno de los caminos de mayor longitud en el árbol, entonces debe contener al menos dos nodos v_1 y v_2 que satisfagan las siguientes condiciones:

1. v_1 y v_2 tienen la misma etiqueta, A,
2. el nodo v_1 está más próximo a la raíz que el nodo v_2,
3. el camino desde v_1 hasta la hoja tiene una longitud máxima de *k+1*.

Si se denomina T_1 al árbol de derivación que cuelga del nodo v_1, entonces la longitud de la cadena resultado está acotada por el valor 2^k (puesto que la altura de T_1 es, como máximo, *k+1*). Sea z_1 esa cadena.

Sea T_2 el árbol que cuelga de v_2 y z_2 su cadena resultado; entonces, se puede escribir z_1 como $z_3 z_2 z_4$.

Además, z_3 y z_4 no pueden ser λ simultáneamente, puesto que la primera derivación utilizada para obtener z_1 debe estar en FNCH, es decir, será de la forma $A \rightarrow BC$.

Por lo tanto, el árbol T_2 debe pertenecer *totalmente* o bien al árbol que cuelga de B o bien al que cuelga de C.

En resumen, se conoce que $A \stackrel{*}{\Rightarrow} z_3 A z_4$ y que $A \stackrel{*}{\Rightarrow} z_2$, tal que $\mid z_3 z_2 z_4 \mid \leq n = 2^k$.

Entonces, $A \stackrel{*}{\Rightarrow} z_3^i z_2 z_4^i$, para cualquier valor positivo de *i*. Por lo tanto, la cadena z_1 puede escribirse como $u z_3 z_2 z_4 y$, siendo *u* e *y* cadenas cualesquiera.

Identificando z_3 con *v*, z_2 con *w* y z_4 con *x* se demuestran todos los enunciados del lema.

c.q.d.

Ejemplo:

Sea la GCL G,

$$G = \langle \{S, A, B\}, \{a, b\}, S, \{S \rightarrow AA; A \rightarrow AB \mid a; B \rightarrow AS \mid b\} \rangle.$$

La cadena abaaaa *tiene asociado el árbol de derivación mostrado en la figura 6.1.*

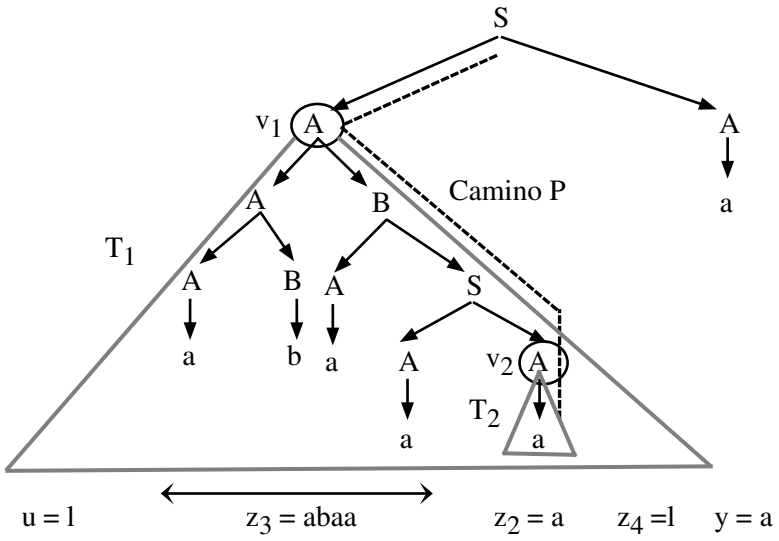


Figura 6.1: Árbol de derivación de la cadena *abaaaa*.

Como la altura del árbol es 5 y el número de símbolos auxiliares es 3, entonces deben existir, como mínimo, dos nodos que cumplan las tres condiciones expuestas en la demostración del lema de bombeo.

Sean v_1 y v_2 dos de esos nodos. Su etiqueta es el símbolo A.

Desde v_2 se genera la cadena a*, pero también podría generarse de nuevo, y repetidas veces, todo el subárbol T_1, obteniéndose las cadenas $uz_3^i z_2 z_4^i y$. Todas estas cadenas pertenecen a L(G).*

El lema de bombeo resulta de utilidad para demostrar que un lenguaje *no es de contexto libre*, si no lo cumple.

Ejemplo:

Demostrar que el lenguaje $\{a^n b^n c^n \mid n \geq 0\}$ no es un lenguaje de contexto libre.

Para poder comprobar si se cumplen las condiciones impuestas por el lema de bombeo, se debe localizar una cadena del lenguaje cuya longitud sea mayor que la constante del lema para este lenguaje.

Sea k *esa constante. La cadena $z = a^k b^k c^k$, pertenece al lenguaje y su longitud es mayor que* k, $\mid z \mid = 3k$*; por lo tanto,* z *se puede escribir como* uvwxy*, pero ¿cuáles son los símbolos de* z *que forman parte de la cadena* v *y de la cadena* x*?.*

Se deben analizar todos los casos posibles: si al menos uno de estos casos satisficiera las tres condiciones del lema, no se podría demostrar que no se cumple el lema para esta cadena.

1. *Si* v *y* x *están compuestas sólo de símbolos* a*'s, entonces las cadenas* $uv^iwx^iy,\ con\ i \geq 2$ *no pertenecen al lenguaje (contienen más* a*'s que* b*'s y* c*'s).*

$$\underbrace{\underbrace{aa\ldots a}^{(k)}}_{uvwx}\underbrace{\underbrace{ab\ldots b}^{(k)}\underbrace{bcc\ldots cc}^{(k)}}_{y}$$

2. *Si* v *y* x *están compuestas sólo de símbolos* b*'s, entonces las cadenas* $uv^iwx^iy,\ con\ i \geq 2$ *no pertenecen al lenguaje (contienen más* b*'s que* a*'s y* c*'s).*

$$\underbrace{\underbrace{aa\ldots a}^{(k)}}_{u}\underbrace{\underbrace{abb\ldots b}^{(k)}}_{vwx}\underbrace{\underbrace{bcc\ldots cc}^{(k)}}_{y}$$

3. *Si* v *y* x *están compuestas sólo de símbolos* c*'s, entonces las cadenas* $uv^iwx^iy,\ con\ i \geq 2$ *no pertenecen al lenguaje (contienen más* c*'s que* a*'s y* b*'s).*

$$\underbrace{\underbrace{aa\ldots a}^{(k)}\underbrace{aabb\ldots b}^{(k)}}_{u}\underbrace{\underbrace{bccc\ldots c}^{(k)}}_{vwxy}$$

4. *Si* v *está compuesta por* a*'s y* x *por* b*'s, entonces las cadenas* uv^iwx^iy*, con i* $\geq$ *2 no pertenecen al lenguaje (contienen más* a*'s y* b*'s que* c*'s).*

$$\underbrace{\underbrace{aa\ldots a}^{(k)}}_{uv}\underbrace{\underbrace{abb\ldots b}^{(k)}}_{wx}\underbrace{\underbrace{bcc\ldots cc}^{(k)}}_{y}$$

5. *Si* v *está compuesta por* b*'s y* x *por* c*'s, entonces las cadenas* uv^iwx^iy*, con i* $\geq$ *2 no pertenecen al lenguaje (contienen más* b*'s y* c*'s que* a*'s).*

$$\underbrace{\underbrace{aa\ldots a}^{(k)}}_{u}\underbrace{\underbrace{abb\ldots b}^{(k)}}_{v}\underbrace{\underbrace{bcc\ldots cc}^{(k)}}_{wxy}$$

6. *No puede darse el caso que* v *sean* a*'s y* x *sean* c*'s, pues entonces* $\mid vwx \mid > k$ *(en* w *hay como mínimo* k *símbolos* b*'s).*

$$\underbrace{\underbrace{aa\ldots a}^{(k)}}_{uv}\underbrace{\underbrace{abb\ldots b}^{(k)}}_{w}\underbrace{\underbrace{bcc\ldots cc}^{(k)}}_{xy}$$

7. *No pueden mezclarse símbolos en* v *ni en* x *pues si se hiciese entonces las cadenas* $uv^iwx^iy,\ con\ i \geq 2$ *no pertenecerían al lenguaje (habría símbolos descolocados con respecto al formato de las cadenas del lenguaje)*

Como no hay más opciones posibles para la asignación de símbolos a las cadenas v *y* x *y ninguna de ellas es válida, entonces este lenguaje no cumple el lema de bombeo y, por lo tanto, el lenguaje*

$$\{a^nb^nc^n \mid n \geq 0\}$$

no es un lenguaje de contexto libre.

6.2. Propiedades de Clausura.

Las propiedades de clausura para LCL, son aquellas operaciones que, aplicadas sobre LCL, dan como resultado otro LCL. Tal y como ocurría al estudiar los lenguajes regulares, dichas propiedades pueden ayudar a determinar si un lenguaje es, o no es, de contexto libre.

Teorema 6.1 *Los LCL son cerrados bajo las operaciones de unión, concatenación, clausura e inversión.*

La demostración de este teorema se deja propuesto como ejercicio.

Teorema 6.2 *Los LCL son cerrados bajo la operación de sustitución.*

Demostración:

Sea $G = \langle \Sigma_A, \Sigma_T, P, S \rangle$ una GCL tal que L(G) = L.

Se define la operación de sustitución $f : \Sigma \longrightarrow 2^{\Delta^*}$, de forma que, $\forall\ a \in \Sigma, f(a) = L(G_a)$; es decir, la operación de sustitución, aplicada a cada símbolo de Σ, proporciona como resultado una GCL $G_a = \langle \Sigma_{Aa}, \Sigma_{Ta}, P_a, S_a \rangle$ sobre el alfabeto Δ.

Además, se cumple que

$\forall\ a, b \in \Sigma$, con $G_a = \langle \Sigma_{Aa}, \Sigma_{Ta}, P_a, S_a \rangle$ y $G_b = \langle \Sigma_{Ab}, \Sigma_{Tb}, P_b, S_b \rangle$, entonces $(\Sigma_A \cap \Sigma_{Aa}) \cap \Sigma_{Ab} = \emptyset$.

La tarea consiste en construir una GCL que reconozca *f(L)*.

Sea $G' = \langle \Sigma'_A, \Sigma'_T, P', S' \rangle$ tal que,

$$\begin{aligned} \Sigma'_A &= \Sigma_A \cup (\textstyle\bigcup_{a \in \Sigma} \Sigma_{Aa}) \\ \Sigma'_T &= \textstyle\bigcup_{a \in \Sigma} \Sigma_{Ta} \\ S' &= S \\ P' &= P'' \cup (\textstyle\bigcup_{a \in \Sigma} P_a) \end{aligned}$$

donde P$''$ se obtiene de P reemplazando, en cada producción de P, cada símbolo terminal del consecuente por el símbolo inicial de la GCL asociada a ese símbolo terminal mediante la operación de sustitución *f*. Por ejemplo, si $(A \rightarrow abS) \in P$, entonces $(A \rightarrow S_a S_b S) \in P''$.

Se demostrará que *L(G') = f(L)*, estudiando las dos inclusiones.

$L(G') \subseteq f(L)$ **:** Sea $x = f(a_1)f(a_2)\ldots f(a_n) \in L(G')$; desde S', y aplicando únicamente producciones del conjunto P$''$, se obtiene una forma sentencial $\theta = S_{a_1} S_{a_2} \ldots S_{an}$ y, a partir de ésta y aplicando las producciones de $\bigcup_{a \in \Sigma} P_a$, se obtiene una cadena de terminales *x*, es decir, $S' \stackrel{*}{\Rightarrow} \theta \stackrel{*}{\Rightarrow} x$.

En la GCL G se puede obtener la cadena $S \stackrel{*}{\Rightarrow} y = a_1a_2 \dots a_n$ que pertenece a L, luego $f(y) = f(a_1)f(a_2) \dots f(a_n) = x$, pertenece a f(L).

$f(L) \subseteq L(G')$**:** Sea $x \in f(L)$, entonces $x = f(a_1)f(a_2) \dots f(a_n)$, tal que $y = a_1a_2 \dots a_n$ es una cadena de L. Por lo tanto, la cadena *y* se puede obtener a partir de S aplicando producciones del conjunto P$''$, y, a partir de esta forma sentencial, se pueden aplicar producciones de $\bigcup_{a\in\Sigma} P_a$, obteniéndose la cadena *x*.

c.q.d.

Ejemplo:

Sea $\Sigma = \{a, b\}$ *y* $\Delta = \{0, 1, 2\}$*, tal que*

$$f(a) = L_a = \{0^n1^n \mid n \geq 1\},\ y$$
$$f(b) = L_b = \{xx^{-1} \mid x \in (0 + 2)^*\}.$$

Sea $L = \{x \in (a + b)^* \mid S(x, a) = S(x, b)\}$*. Calcular una GCL que genere f(L).*

Sea G_a *la GCL definida por las siguientes producciones:*

$$S_a \rightarrow 0S_a1 \mid 01$$

Sea G_b *la GCL definida por las siguientes producciones:*

$$S_b \rightarrow 0S_b0 \mid 2S_b2 \mid \lambda$$

Sea G la GCL definida por las siguientes producciones:

$$S \rightarrow aSbS \mid bSaS \mid SS \mid \lambda$$

Entonces $G' = \langle \Sigma'_A, \Sigma'_T, P', S\rangle$*, donde*

$$\Sigma'_A = \{S, S_a, S_b\}$$
$$\Sigma'_T = \{0, 1, 2\}$$
$$P' = \{S \rightarrow S_aSS_bS \mid S_bSS_aS \mid SS \mid \lambda;$$
$$S_a \rightarrow 0S_a1 \mid 01; S_b \rightarrow 0S_b0 \mid 2S_b2 \mid \lambda\}$$

Con esta definición, sea y=abab *(por lo tanto,* f(y)=f(a)f(b)f(a)f(b)*) y sea* $x \in f(y)$*, tal que* x=001102200122 *con* $0011 \in f(a)$*,* $0220 \in f(b)$*,* $01 \in f(a)$ *y* $22 \in f(b)$*. La cadena* x *se puede generar en* G' *mediante la secuencia*

$$S \rightarrow S_aSS_bS \rightarrow S_aS_bS \rightarrow S_aS_bS_aSS_bS \rightarrow S_aS_bS_aS_bS$$
$$\rightarrow S_aS_bS_aS_b = \theta \stackrel{*}{\Rightarrow} 001102200122$$

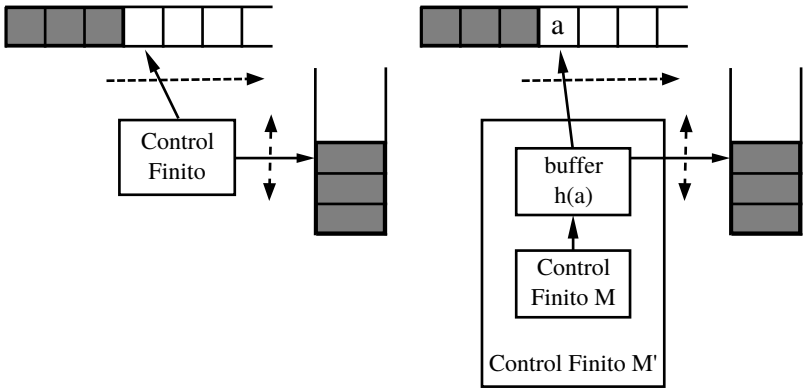
Corolario 6.1 *Los LCL son cerrados bajo la operación de homomorfismo (caso particular de la sustitución).*

Teorema 6.3 *Los LCL son cerrados bajo la operación de homomorfismo inverso.*

Demostración:

Sea $h : \Sigma \longrightarrow \Delta^*$ un homomorfismo. Sea L un LCL de Δ^*. Se demostrará que $h^{-1}(L)$ es otro LCL.

El método consiste en construir un AP, M′, que reconozca $h^{-1}(L)$ a partir del AP M que reconoce L, de forma que, para cada símbolo leído de la cinta de entrada, $a \in \Sigma$, el AP M′ almacene en un buffer el valor $h(a) \in \Delta^*$, analizando el contenido de este buffer según se indica en el AP M. Es decir, el AP M′, ante una entrada de la forma $a_1 a_2 \ldots a_n$, almacena en el buffer $h(a_1)h(a_2)\ldots h(a_n)$.



La definición formal de M′ será la siguiente: dado $M = \langle \Sigma, Q, \Gamma, f, q_0, Z_0, F \rangle$, entonces se construye el AP $M' = \langle \Sigma, Q', \Gamma, f', [q_0, \lambda], Z_0, F' \rangle$, donde

$$Q' = \{[q, x] \mid q \in Q,\ \exists\, a \in \Sigma \mid x\ es\ sufijo\ de\ h(a)\},\ y$$
$$F' = \{[q, \lambda] \mid q \in F\}.$$

La función de transición, f', se define como:

1. $f'([q_0, \lambda], a, Z_0) = \{([q_0, h(a)], Z_0)\}$. Se inicializa la simulación del AP M en el control finito del AP M′,
2. $([p, x], \gamma) \in f'([q, x], \lambda, Y),\ tal\ que\ (p, \gamma) \in f(q, \lambda, Y)$. Se simulan los movimientos λ del AP M,
3. $([p, x], \gamma) \in f'([q, ax], \lambda, Y),\ tal\ que\ (p, \gamma) \in f(q, a, Y)$. Se simulan, en M′, los movimientos del AP M,

4. $f'([q,\lambda],\lambda,Y) = \{([q,\lambda],\lambda)\}, Y \neq Z_0$. Cuando se vacía el buffer y se ha llegado a un estado $q \in F$, entonces se ha de vaciar la pila.
5. $f'([q,\lambda],a,Z_0) = \{([q_0,h(a)],Z_0)\}$. Cuando se vacía el buffer y se ha llegado a un estado $q \in F$, y en la pila sólo está el símbolo Z_0, entonces se prepara para analizar *h(a)*.

Falta por demostrar que $L(M') = h^{-1}(L)$:

$h^{-1}(L) \subseteq L(M')$ **:** Sea $x \in h^{-1}(L)$, entonces $h(x) \in L$, y, por lo tanto, en el AP M se realiza la siguiente secuencia,

$$(q_0,h(x),Z_0) = (q_0,h(a_1)h(a_2)\ldots h(a_n),Z_0) \overset{*}{\vdash} (q_1,h(a_2)\ldots h(a_n),\alpha_1) \overset{*}{\vdash}$$
$$(q_2,h(a_3)\ldots h(a_n),\alpha_2) \overset{*}{\vdash} (q_n,\lambda,\alpha_n),\ q_n \in F \wedge \alpha_n \in \Gamma^*.$$

Al aplicarle la misma cadena al autómata M′ se obtiene la siguiente secuencia,

$$([q_0,\lambda],a_1a_2\ldots a_n,Z_0) \vdash ([q_0,h(a_1)],a_2\ldots a_n,Z_0) \overset{*}{\vdash}$$
$$([q_1,\lambda],a_2\ldots a_n,Z_0) \overset{*}{\vdash} ([q_n,\lambda],\lambda,Z_0),\ q_n \in F.$$

Por lo tanto, $x \in L(M')$.

$L(M') \subseteq h^{-1}(L)$ **:** Sea $x \in L(M')$, entonces $x = a_1a_2\ldots a_n$ y en el análisis de cada símbolo a_i, el AP M'reconoce $h(a_i)$ mediante las transiciones del AP M. Por lo tanto, la cadena que está reconociendo implícitamente M′ es $h(a_1)h(a_2)\ldots h(a_n)$ y, entonces, $x = a_1a_2\ldots a_n \in h^{-1}(L)$.

c.q.d.

Teorema 6.4 *Los LCL no son cerrados bajo la operación de intersección.*

Demostración:

Es suficiente con encontrar un contraejemplo.

Sea $L_1 = \{a^nb^nc^m \mid n \geq 1, m \geq 1\}$, el LCL generado por el siguiente conjunto de producciones de contexto libre

$$P_1 = \{S \rightarrow AB; A \rightarrow aAb \mid ab; B \rightarrow cB \mid c\}.$$

Sea $L_2 = \{a^nb^mc^m \mid n \geq 1, m \geq 1\}$, el LCL generado por el siguiente conjunto de producciones de contexto libre

$$P_2 = \{S \rightarrow CD; C \rightarrow aC \mid a; D \rightarrow bDc \mid bc\}.$$

El resultado de la intersección es el lenguaje $L_1 \cap L_2 = \{a^nb^nc^n \mid n \geq 1\}$, que por el lema de bombeo se conoce que no es un LCL.

c.q.d.

Corolario 6.2 *Los LCL no son cerrados bajo la operación de complementación.*

Demostración:

Sean L_1 y L_2 LCL; entonces, si la complementación fuese una operación de clausura, también serían LCL los lenguajes $\overline{L_1}$ y $\overline{L_2}$.

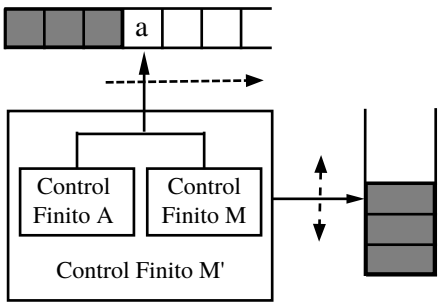
Por lo tanto, como la unión de LCL es una operación de clausura, también $\overline{L_1} \cup \overline{L_2}$ es un LCL y, por lo tanto, también lo sería el lenguaje $\overline{\overline{L_1} \cup \overline{L_2}} = L_1 \cap L_2$, lo que se sabe que es falso.

c.q.d.

Teorema 6.5 *Los LCL son cerrados bajo la operación de intersección con lenguajes regulares.*

Demostración:

Sea M un AP tal que L(M) = L. Sea A un AFD tal que R = L(A). El método de demostración se centra en construir un AP que reconozca $L \cap R$. Informalmente este AP, M′, se puede describir por medio de la siguiente figura:



La cadena de entrada será aceptada si, una vez consumida, tanto el control finito del AFD A como el control finito del AP M se encuentran en un estado final, o pueden llegar a uno de ellos mediante transiciones λ.

Formalmente esta construcción se realiza de la forma siguiente:

Sea $M = \langle \Sigma, Q_M, \Gamma, f_M, q_0, Z_0, F_M \rangle$ un AP y sea $A = \langle \Sigma, Q_A, f_A, p_0, F_A \rangle$, entonces se construye el AP M′, $M' = \langle \Sigma, Q_A \times Q_M, \Gamma, f, [q_0, p_0], Z_0, F_A \times F_M \rangle$, donde la función de transición se define de la forma siguiente:

$$([p', q'], \gamma) \in f([p, q], a, X) \Leftrightarrow f_A(p, a) = p' \wedge (q', \gamma) \in f_M(q, a, X).$$

Falta demostrar que $L(M') = L \cap R$, lo que se hará por inducción sobre el número de transiciones realizadas en la expresión,

$$([p_0, q_0], x, Z_0) \overset{i}{\vdash} ([p, q], \lambda, \gamma) \Leftrightarrow f_A(p_0, x) = p \wedge (q_0, x, Z_0)) \overset{i}{\vdash} (q, \lambda, \gamma)$$

"⇒": Hay que demostrar que

$$([p_0, q_0], x, Z_0) \overset{i}{\vdash} ([p, q], \lambda, \gamma) \Rightarrow f_A(p_0, x) = p \wedge (q_0, x, Z_0)) \overset{i}{\vdash} (q, \lambda, \gamma).$$

Paso Base, i = 0: Entonces, $p = p_0,\ q = q_0, x = \lambda,\ \gamma = Z_0$.

Hipótesis de Inducción: $([p_0, q_0], x, Z_0) \overset{i-1}{\vdash} ([p, q], \lambda, \gamma) \Leftrightarrow f_A(p_0, x) = p \wedge (q_0, x, Z_0)) \overset{i-1}{\vdash} (q, \lambda, \gamma)$

Paso de Inducción: Sea *y* = *xa*, entonces,

$$([p_0, q_0], xa, Z_0) \overset{i-1}{\vdash} ([p, q], a, \gamma) \vdash ([p', q'], \lambda, \beta).$$

La derivación de los *i-1* primeros pasos, por hipótesis de inducción, implica que

$$(q_0, xa, Z_0) \overset{i-1}{\vdash} (q, a, \gamma)$$

y que $f_A(p_0, xa) = f_A(f_A(p_0, x), a) = f_A(p, a)$.
Además, el paso $([p, q], a, \gamma)) \vdash ([p', q'], \lambda, \beta)$ implica que $\gamma = X\gamma' \wedge \beta = \beta'\gamma'$, existiendo las transiciones $(q', \beta') \in f_M(q, a, X)$ y que, también, $f_A(p, a) = p'$.
Por lo tanto, se cumple que

$$(q_0, xa, Z_0) \overset{i-1}{\vdash} (q, a, \gamma) = (q, a, X\gamma') \vdash (q', \lambda, \beta) = (q', \lambda, \beta'\gamma')$$

y que $f_A(p_0, xa) = f_A(f_A(p_0, x), a) = f_A(p, a) = p'$.

"⇐": Hay que demostrar que

$$f_A(p_0, x) = p \wedge (q_0, x, Z_0)) \overset{i}{\vdash} (q, \lambda, \gamma) \Rightarrow ([p_0, q_0], x, Z_0) \overset{i}{\vdash} ([p, q], \lambda, \gamma).$$

La demostración es similar a la del caso anterior y se deja propuesta como ejercicio.

c.q.d.

Este teorema resulta de gran utilidad para determinar que un lenguaje no es de contexto libre.

Ejemplo:

Determinar si el lenguaje $L = \{xx \mid x \in (a+b)^*\}$ *es un lenguaje de contexto libre.*

El lenguaje $a^+b^+a^+b^+$ *es un lenguaje regular. Si L fuese un lenguaje de contexto libre, también lo sería el lenguaje* $L \cap a^+b^+a^+b^+$*. Pero este lenguaje es,*

$$L \cap a^+b^+a^+b^+ = \{a^nb^ma^nb^m \mid n, m \geq 1\}$$

que se demuestra que no es de contexto libre al no cumplir el lema de bombeo para lenguajes de contexto libre.

Sea k *la constante del lema para L. Entonces* $z = a^kb^ka^kb^k$ *pertenece a L y su longitud es mayor que* k*; por lo tanto,* z *se puede escribir como* uvwxy*, pero ¿cuáles son los símbolos de* z *que componen la subcadena* x *y cuáles lo que componen la subcadena* y*?. Se analizan todos los casos posibles:*

1. *Si* v *y* x *están compuestas sólo de símbolos* a*'s del principio de* z*, entonces las cadenas* $uv^iwx^iy,\ i \geq 2$ *no pertenecen al lenguaje (contienen más* a*'s en el primer bloque que en el segundo).*

$$\underbrace{\underbrace{aa\overset{(k)}{\ldots}aa}_{uvwx}b\ldots bbb\ldots b}_{y}\underbrace{aaab\overset{(k)}{\ldots}bb}$$

   Algo similar sucedería si se escogieran a*'s del segundo bloque.*

2. *Si* v *y* x *están compuestas sólo de símbolos* b*'s del principio de* z*, entonces las cadenas* $uv^iwx^iy,\ i \geq 2$ *no pertenecen al lenguaje (contienen más* b*'s en el primer bloque que en el segundo).*

$$\underbrace{\underbrace{aa\ldots aa\overset{(k)}{bb}\ldots bb}_{uvwx}aa\ldots abb\ldots bb}_{y}$$

   Algo similar sucedería si se escogieran b*'s del segundo bloque.*

3. *Si* v *está compuesta por* a*'s y* x *por* b*'s, ambas en la primera mitad de* z*, entonces las cadenas* $uv^iwx^iy,\ i \geq 2$ *no pertenecen al lenguaje (contienen más* a*'s y* b*'s en la primera mitad que en la segunda).*

$$\underbrace{aa}_{u}\underbrace{\overset{(k)}{\ldots}a}_{v}\underbrace{abb}_{w}\underbrace{\overset{(k)}{\ldots}bb}_{x}\underbrace{aa\ldots abb\overset{(k)}{\ldots}bb}_{y}$$

   Algo similar sucedería si se escogieran a*'s y* b*'s de la segunda mitad; o* b*'s de la primera mitad y* a*'s de la segunda mitad.*

4. *No puede darse el caso que* v *esté formada por* a*'s de la primera mitad y* x *por* a*'s de la segunda mitad, pues entonces* $\mid vwx \mid > k$*, ya que en* w *hay, como mínimo,* k *símbolos* b*'s.*

$$\underbrace{aa}_{u}\underbrace{\overset{(k)}{\ldots}}_{v}\underbrace{aabb\overset{(k)}{\ldots}bbaa}_{w}\underbrace{\overset{(k)}{\ldots}}_{x}\underbrace{aabb\overset{(k)}{\ldots}bb}_{y}$$

Algo similar sucedería si se escogieran b*'s de la primera mitad y de la segunda mitad.*

5. *No pueden mezclarse símbolos en* v *ni en* x*, pues, si se hiciese, entonces las cadenas* $uv^iwx^iy,\ i \geq 2$ *no pertenecerían al lenguaje (habría símbolos descolocados respecto al formato de las cadenas del lenguaje)*

Como no hay más opciones posibles para la asignación de símbolos a las cadenas v *y* x*, este lenguaje no cumple el lema de bombeo y, por lo tanto, el lenguaje* $\{a^nb^ma^nb^m \mid n, m \geq 1\}$ *no es un lenguaje de contexto libre. Tampoco lo será el lenguaje* $L = \{xx \mid x \in (a+b)^*\}$.

6.3. Algoritmos de Decisión.

Un gran número de cuestiones sobre LCL resultan solucionables mediante el uso de algoritmos de decisión como, por ejemplo, determinar si un LCL es vacío, finito o infinito, o bien determinar si una determinada cadena puede ser generada por una determinada GCL.

Existen, sin embargo, otras cuestiones sobre LCL que no pueden ser resueltas algorítmicamente; es decir, no existen algoritmos que resuelvan cada una de estas cuestiones para cualquier GCL que reciban como entrada.

Ejemplos de tales cuestiones son el determinar si dos GCL son equivalentes, si una GCL es ambigua, o bien si el complementario de un LCL es otro LCL.

En esta sección se introducen una serie de algoritmos de decisión para algunas de las cuestiones resolubles enunciadas anteriormente.

Teorema 6.6 *Existe un algoritmo de decisión para determinar si un LCL es*

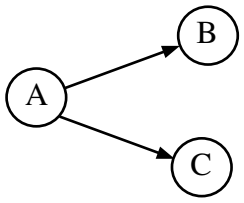
1. *vacío,*
2. *finito o infinito.*

Demostración:

El primer apartado del teorema resulta sencillo de demostrar: basta con aplicar el algoritmo de eliminación de símbolos inútiles y comprobar si el símbolo inicial, S, es o no inútil. Si S es inútil entonces $L(G) = \emptyset$, y en caso contrario $L(G) \neq \emptyset$.

Para determinar si L = L(G) es o no es finito se utiliza el siguiente método: Sea $G = \langle \Sigma_A, \Sigma_T, P, S \rangle$ una GCL tal que L(G) = L; entonces, se construye la gramática $G' = \langle \Sigma'_A, \Sigma'_T, P', S \rangle$, en Forma Normal de Chomsky, tal que $L(G') = L(G) - \{\lambda\}$.

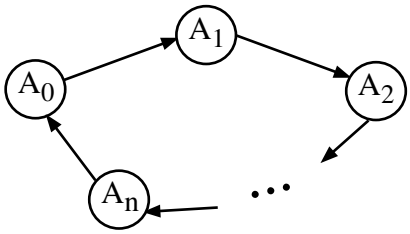
A partir de esta gramática se construye un grafo dirigido donde cada nodo representa a un elemento de Σ'_A y cada producción de la forma $A \rightarrow BC$, o bien $A \rightarrow CB$, da lugar a dos arcos, uno que conecta el nodo A con el nodo B y otro que conecta al nodo A con el nodo C.



Se demostrará que

El lenguaje L(G) es finito $\Leftrightarrow$ en el grafo asociado a G′ no existen caminos cerrados.

Para la comprobación del enunciado anterior, primero se supondrá que existe un camino cerrado, por ejemplo, $A_0A_1A_2 \ldots A_nA_0$



Nota: como el arco que conecta el nodo A_i con el nodo A_{i+1} puede representar tanto a la producción $A_i \rightarrow A_{i+1}B$ como a la producción $A_i \rightarrow BA_{i+1}$, entonces

- si la producción que representa el arco es $A_i \rightarrow BA_{i+1}$, este arco expresa la derivación $\alpha_iA_i\beta_i \Rightarrow \alpha_{i+1}A_{i+1}\beta_{i+1}$ con $\alpha_{i+1} = \alpha_iB \wedge \beta_{i+1} = \beta_i$;
- si la producción que represente ese arco es $A_i \rightarrow A_{i+1}B$, este arco expresa la derivación $\alpha_iA_i\beta_i \Rightarrow \alpha_{i+1}A_{i+1}\beta_{i+1}$ con $\alpha_{i+1} = \alpha_i \wedge \beta_{i+1} = B\beta_i$.

Inicialmente, $\alpha_0 = \beta_0 = \lambda$.

Al camino $A_0A_1A_2 \ldots A_nA_0$ se le puede asociar la siguiente secuencia de derivaciones,

$$A_0 \Rightarrow \alpha_1A_1\beta_1 \Rightarrow \alpha_2A_2\beta_2 \stackrel{*}{\Rightarrow} \alpha_nA_n\beta_n \Rightarrow \alpha_{n+1}A_0\beta_{n+1}$$

donde $\mid \alpha_{n+1}\beta_{n+1} \mid = n + 1 \wedge \alpha_{n+1}, \beta_{n+1} \in {\Sigma'_A}^*$.

Además, como todos los símbolos de G′ están en FNCH, $\alpha_{n+1} \stackrel{*}{\Rightarrow} v,\ v \in {\Sigma'_T}^*$ y $\beta_{n+1} \stackrel{*}{\Rightarrow} x,\ x \in {\Sigma'_T}^*$.

Como el símbolo A_0 es alcanzable, entonces $S \stackrel{*}{\Rightarrow} \gamma A_0 \delta \stackrel{*}{\Rightarrow} uA_0y,\ u, y \in {\Sigma'_T}^*$.

El símbolo A_0 también resulta ser derivable y, entonces, $A_0 \stackrel{*}{\Rightarrow} w,\ w \in {\Sigma'_T}^*$.

Reuniendo todos estos resultados se obtiene:

$$S \stackrel{*}{\Rightarrow} uA_0y \stackrel{*}{\Rightarrow} uvA_0xy \stackrel{*}{\Rightarrow} uv^2A_0x^2y \stackrel{*}{\Rightarrow} uv^iA_0x^iy \stackrel{*}{\Rightarrow} uv^iwx^iy,\ \forall\, i \geq 0$$

por lo que, variando el valor de la *i* se obtienen diferentes, e infinitas, cadenas del lenguaje. Por lo tanto, L(G) es infinito.

Falta demostrar que si en el grafo no existen caminos cerrados, entonces el lenguaje es finito.

Para ello hay que definir previamente el concepto de *Rango(A)*, como la longitud del camino más largo que existe en el grafo partiendo del nodo A, $\forall\ A \in \Sigma'_A$.

Nótese que esta definición de *Rango(A)* sólo tiene sentido si en el grafo no existen caminos cerrados (si existiese un camino cerrado en el nodo A no se podría hablar de la longitud máxima de ese camino, puesto que se puede recorrer tantas veces como se desee).

Con este definición se cumple que,

- si $(A \rightarrow a) \in P'$, entonces *Rango(A) = 0*, y,
- si $(A \rightarrow BC) \in P'$, entonces $Rango(B) < Rango(A) \wedge rango(C) < Rango(A)$.

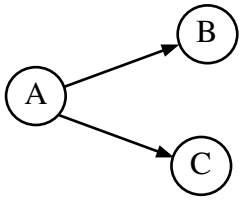
Se demostrará por inducción que

$$\forall\, x \in {\Sigma'_T}^*,\ si\ A \stackrel{*}{\Rightarrow} x,\ entonces\ \mid x \mid \leq 2^{Rango(A)}.$$

Paso Base : *Rango(A) = 0*; esto ocurre cuando $A \rightarrow a \in P'$ y, entonces, $\mid a \mid = 1 \leq 2^{Rango(A)} = 2^0 = 1$.

Hipótesis de Inducción : *Rango(A) = r*, entonces si $A \stackrel{*}{\Rightarrow} x,\ resulta\ que\ \mid x \mid \leq 2^r$.

Paso de Inducción : Sea *Rango(A) = r+1*. Entonces, en el grafo existe una relación del tipo



es decir, $A \Rightarrow BC \stackrel{*}{\Rightarrow} z = xy$, donde $Rango(B) < Rango(A)$ y $Rango(C) < Rango(A)$; por lo tanto, si $B \stackrel{*}{\Rightarrow} x$, entonces $\mid x \mid \leq 2^{Rango(B)} < 2^{Rango(A)}$, o, lo que es lo mismo, $\mid x \mid \leq 2^r$.

Aplicando un razonamiento similar con el símbolo C, se obtiene que si $C \stackrel{*}{\Rightarrow} y$, se cumple entonces $\mid y \mid \leq 2^r$, por lo que resulta que

$$\mid z \mid = \mid xy \mid = \mid x \mid + \mid y \mid \leq 2^r + 2^r = 2^{r+1}.$$

Si se denomina $Rango(S) = r_0$, entonces $\forall\ x \in L(G')$ se verifica que $\mid x \mid \leq 2^{Rango(S)} = 2^{r_0}$. Como r_0 es una constante, entonces el número de cadenas que pueden pertenecer a L(G') tiene que ser finito, puesto que su longitud está acotada por una constante.

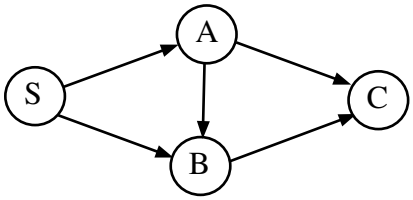
c.q.d.

Ejemplo:

Sea G la siguiente GCL en Forma Normal de Chomsky

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow BC \mid a \\ B &\rightarrow CC \mid b \\ C &\rightarrow a \end{aligned}$$

determinar si L(G) es finito o infinito.



La primera tarea a realizar es construir su grafo asociado. Como en el grafo no hay caminos cerrados entonces L(G) es finito; más aún, el número máximo de cadenas de L(G) está limitado a cadenas de longitud máxima $2^{Rango(S)} = 2^3 = 8$.

El siguiente algoritmo que se va a exponer trata de determinar la *pertenencia de una cadena* x *a un lenguaje*.

Se podría pensar que la cuestión puede quedar resuelta, simplemente, introduciendo la cadena *x* como entrada al AP que reconoce un determinado lenguaje, L(G). Pero, si se permite que el AP sea arbitrario, podría contener producciones del tipo

$$(q, X) \in f(q, \lambda, X)\ ,$$

o bien del tipo

$$(p, Y) \in f(q, \lambda, X)\ \wedge\ (q, X) \in f(p, \lambda, Y)\ ;$$

es decir, transiciones que provocarían en el AP una ejecución infinita, con lo que sería posible no saber si la cadena será, o no, aceptada por el AP. Este método sólo sería válido si se pudiera garantizar que el AP no contiene transiciones que provoquen una ejecución infinita.

En el capítulo 4 se demostró que cualquier GCL G, tal que $\lambda \notin L(G)$, tiene una GCL, G′, equivalente en Forma Normal de Greibach. Basado en esta forma normal, hay un método básico para determinar si una cadena *x*, $\mid x \mid = n$, pertenece al conjunto de cadenas generables por G. Consiste en construir todas las derivaciones de *n* pasos en la GCL G′; si alguna de estas derivaciones genera la cadena *x*, entonces $x \in L(G)$, y si ninguna de estas derivaciones permite obtener *x*, entonces $x \notin L(G)$. El inconveniente de este método es que presenta un alto coste computacional, de tipo exponencial, $O(k^n)$.

Un método alternativo consistiría en construir el AP asociado a G′, mediante el método expuesto en el capítulo 5. Una vez construido, bastaría con simular el comportamiento de este AP no determinista con todas las posibles transiciones. Si en alguna de estas secuencias se llegase a consumir la cadena, resultando la pila vacía, entonces la cadena sería aceptada; en caso contrario, la cadena sería rechazada. Este método también presenta un coste computacional exponencial, $O(k^n)$.

Existen otros métodos con menor coste que permiten solucionar este problema. De entre estos, uno de los más usados es el método de *Cocke-Younger-Kasami*, más conocido como *algoritmo CYK*.

Este algoritmo se basa en el siguiente razonamiento. Sea $G = \langle \Sigma_A, \Sigma_T, P, S \rangle$ una GCL en Forma Normal de Chomsky y sea $x, \mid x \mid = n$. Para cada símbolo $A \in \Sigma_A$, se definen las cadenas que puede derivar como $A \overset{*}{\Rightarrow} x_{ij}$, tal que x_{ij} es el segmento de la cadena *x* que comienza en la posición *i* y que tiene longitud *j*. Con esta notación, la cadena *x* se puede denotar como $x = x_{1|x|} = x_{1n}$.

El algoritmo calcula los conjuntos $V_{ij} = \{A \in \Sigma_A \mid A \overset{*}{\Rightarrow} x_{ij}\}$. La condición de pertenencia es la siguiente:

$$x \in L(G) \Leftrightarrow S \overset{*}{\Rightarrow} x = x_{1|x|} \Leftrightarrow S \in V_{1|x|}.$$

Para calcular estos conjuntos se procede por inducción sobre *j*:

$j = 1$ **:** Entonces $x_{i1} \in \Sigma_T$, y, por lo tanto, $V_{i1} = \{A \in \Sigma_A \mid A \to x_{i1} \in P\}$,

$j > 1$ **:** Entonces es condición necesaria y suficiente para que $A \overset{*}{\Rightarrow} x_{ij}$ que

$$\exists\, (A \to BC) \in P \mid \exists\, k,\ 1 \leq k < j,\ tal\ que\ B \overset{*}{\Rightarrow} x_{ik} \ \wedge\ C \overset{*}{\Rightarrow} x_{i+k,j-k},$$

por lo que, como $k < j$ y $j - k < j$, bastaría con comprobar que $B \in V_{ik} \ \wedge\ C \in V_{i+k,j-k}$.

El algoritmo, cuyo coste computacional para el caso peor es $O(n^3)$, es el siguiente:

```
# Algoritmo CYK
for i in range(1,n+1):
V_i1 = {A ∈ Σ_A | (A → x_i1) ∈ P}
for j in range(2,n+1):
    for i in range(1,n-j+2):
      V_ij = ∅
      for k in range(1,j):
        V_ij = V_ij ∪ {A ∈ Σ_A| (A → BC) ∈ P ∧ B ∈ V_ik ∧ C ∈ V_i+k,j-k}
```

Ejemplo:

Determinar si la cadena x = baaba *pertenece al lenguaje generado por la siguiente GCL:*

$$S \to AB \mid BC$$
$$A \to BA \mid a$$
$$B \to CC \mid b$$
$$C \to AB \mid a$$

Se construye iterativamente la siguiente tabla:

$j \setminus i$	*b*	*a*	*a*	*b*	*a*
1	*B*	*A, C*	*A, C*	*B*	*A, C*
2	*S, A*	*B*	*S, C*	*S, A*	
3	$\emptyset$	*B*	*B*		
4	$\emptyset$	*S, A, C*			
5	*S, A, C*				

Cada entrada (i,j) *de la tabla se rellena con el valor de* V_{ij}. *Como* $S \in V_{15}$, *se verifica que* $x \in L(G)$.

Capítulo 7

Máquinas de Turing

Índice General

7.1. Modelo de Máquina de Turing. 109
7.2. Técnicas para la Construcción de Máquinas de Turing. 115
7.2.1. Almacenamiento en el Control Finito. 115
7.2.2. Cintas con Sectores Múltiples y Marcaje de símbolos. 116
7.2.3. Uso de Subrutinas. 117
7.3. Máquinas de Turing Modificadas. 118
7.3.1. Máquina de Turing con Cinta Infinita en Ambos Sentidos. . . . 118
7.3.2. Máquina de Turing Multicinta. 120
7.3.3. Máquina de Turing con Múltiples Cabezales. 122
7.3.4. Máquina de Turing Offline. 122
7.3.5. Máquina de Turing No Determinista. 123
7.3.6. Máquinas de Turing Restringidas. 125
7.4. Relación de la Máquina de Turing con otros Autómatas. 126

7.1. Modelo de Máquina de Turing.

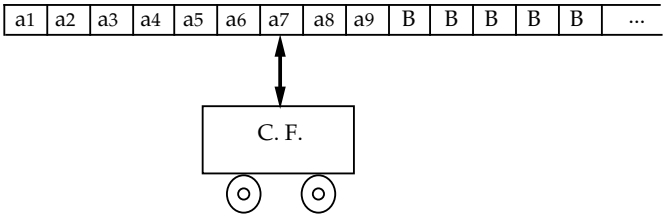
La Máquina de Turing es el último modelo de máquina abstracta que se estudiará. Es un autómata que, a pesar de su simplicidad, es capaz de realizar cualquier cálculo que pueda ser realizado por un computador y no existe ningún otro modelo con mayor poder computacional.

Es por ello que se considera como modelo formal del concepto de Algoritmo, si bien es cierto que existen otros modelos que definen la misma clase de problemas, pudiéndose establecer equivalencias entre todos ellos.

Hay dos puntos de vista para estudiarla:

1. La clase de lenguajes que define: es un *reconocedor de lenguajes de tipo 0*, según la jerarquía de Chomsky.
2. La clase de funciones que computa: es el *solucionador de problemas más potente que hay*.

Como autómata, la Máquina de Turing responde al siguiente modelo mecánico:



1. Tiene una cinta *infinita*, pero limitada a la izquierda. Sus celdas siempre están llenas: o bien por caracteres formando un secuencia de entrada/salida o bien por el carácter especial blanco (`B`).
2. Tiene un cabezal de *lectura/escritura*, que puede desplazarse tanto a la derecha como a la izquierda, con el único límite de que no es posible moverse a la izquierda de la primera celda.
3. Su funcionamiento está basado en un paso elemental, *transición*, que se compone siempre de tres acciones:
   a) Cambio de estado.
   b) Escritura de un símbolo en la celda de la cinta que examina, reemplazando al que hubiera antes.
   c) Desplazamiento a izquierda (L) o derecha (R) una posición.
4. En el *control finito* se controla el funcionamiento: cuál es el estado actual de la máquina y cuáles son las posibles transiciones. El número de estados siempre es *finito*.

La capacidad de escribir es su principal diferencia con otros autómatas, como el autómata finito o el autómata de pilas, y es la que la dota de potencia para reconocer los lenguajes de tipo 0 o para calcular funciones.

Ejemplo 1:

Diseñar una Máquina de Turing para calcular la suma de dos números naturales.

Una forma simple de representar los números naturales es utilizando tantos símbolos como indique su valor numérico. Es decir, si hay que sumar 2 y 3, se podrían representar, respectivamente, como 11 y 111. Para separar ambos sumandos se podría utilizar un 0. Así, la cinta de entrada del autómata podría tener el siguiente aspecto,

`0011011100000BBBBBBBBBB...`

Puesto que 11+111 = 11111, una posible forma de realizar el cálculo si el cabezal estuviera sobre la primera celda sería: primero, localizar el primer sumando (es decir, localizar el primer 1) y, segundo, eliminar ese primer 1 del primer sumando con un 0 y avanzar hasta encontrar el 0 de la separación y sustituirlo por un 1. Se obtendría como cadena de salida,

`0001111100000BBBBBBBBBB...`

Para ello, el autómata debe obedecer a la función de transición:

	0	*1*
q_0	$(q_0,\ 0,\ R)$	$(q_1,\ 0,\ R)$
q_1	$(q_2,\ 1,\ R)$	$(q_1,\ 1,\ R)$
q_2		

En esta tabla, hay tantas columnas como símbolos pueda leer el autómata y tantas filas como estados. Para interpretarla, se asume que cada entrada es la transición asociada al estado indicado por la fila cuando se lee el símbolo asociado a la columna. La tabla anterior es equivalente a la función

$$\begin{aligned}
\mathtt{f}(\mathtt{q}_0, \mathtt{0}) &= (\mathtt{q}_0, \mathtt{0}, \mathtt{R}) \\
\mathtt{f}(\mathtt{q}_0, \mathtt{1}) &= (\mathtt{q}_1, \mathtt{0}, \mathtt{R}) \\
\mathtt{f}(\mathtt{q}_1, \mathtt{1}) &= (\mathtt{q}_1, \mathtt{1}, \mathtt{R}) \\
\mathtt{f}(\mathtt{q}_1, \mathtt{0}) &= (\mathtt{q}_2, \mathtt{1}, \mathtt{R})
\end{aligned}$$

La primera transición, por ejemplo, se lee: "en el estado q_0, cuando se lee un 0, se transita a q_0, se escribe 0 y se mueve el cabezal a la derecha". El estado q_2 es un estado final y no se le asocia ninguna transición.

Ejemplo 2:

Diseñar una Máquina de Turing para determinar si dos números naturales son iguales.

En este caso, se puede representar el primer número mediante 0's y el segundo mediante 1's. El problema se reduciría a comprobar que haya tantos ceros como unos. La cinta de entrada podría tener este aspecto:

`0000011111BBBBBB...`

Se asume que el cabezal está ajustado a la izquierda del primer número. A partir de esta posición, moviéndose hacia la derecha, es fácil localizar el primer 1. Para poder comprobar que hay tantos ceros como unos, se deberían

marcar de alguna forma las parejas que ya se hayan estudiado. Para ello se usará el símbolo X para marcar los ceros y el símbolo Y para marcar los unos. Así, el comportamiento del autómata, en etapas sucesivas, debería producir el siguiente contenido de la cinta:

Inicial:	`0000011111BBBBBB...`
Marcar primera pareja:	`X0000Y1111BBBBBB...`
Marcar segunda pareja:	`XX000YY111BBBBBB...`
Marcar tercera pareja:	`XXX00YYY11BBBBBB...`
Marcar cuarta pareja:	`XXXX0YYYY1BBBBBB...`
Marcar quinta pareja:	`XXXXXYYYYYBBBBBB...`

De esta forma, si al acabar de marcar parejas, sólo quedan X's e Y's en la cinta, seguro que los números eran iguales. Este comportamiento queda descrito por la siguiente función de transición:

	0	*1*	*X*	*Y*	`B`
q_0	(q_1, X, R)			(q_3, Y, R)	
q_1	$(q_1, 0, R)$	(q_2, Y, L)		(q_1, Y, R)	
q_2	$(q_2, 0, L)$		(q_0, X, R)	(q_2, Y, L)	
q_3				(q_3, Y, R)	$(q_4, \mathtt{B}, L)$
q_4					

Como en el ejemplo anterior, no hay transición asociada al estado q_4, que es el final (es al que se llega cuando efectivamente se ha comprobado que ambos números son iguales). Nótese que en la tabla hay entradas, correspondientes a estados no finales, sin transición asociada. En el caso de llegar a alguno de estos huecos, la máquina parará, pero al no parar en el estado q_4, se debe concluir que los números no eran iguales.

Este último ejemplo, además, permite comprender el doble punto de vista del estudio de las Máquinas de Turing. Al fin y al cabo, el segundo ejemplo es una Máquina de Turing que reconoce el lenguaje $0^n 1^n$.

Definición 7.1 *Una Máquina de Turing es una séptupla,*

$$M = \langle \Sigma,\ Q,\ \Gamma,\ f,\ q_0,\ \mathtt{B},\ F \rangle$$

donde,

Σ *es el alfabeto de entrada,*

Q *es el conjunto de estados. Es finito y no vacío. Se ubica en el Control Finito.*

Γ *es el alfabeto de la cinta,* $\Sigma \subseteq \Gamma - \{B\}$.

f *es la función de transición,*

$$f : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L,\ R\}.$$

Partiendo de un estado y de un símbolo, indica la transición a otro estado, el símbolo a escribir en la cinta y el movimiento del cabezal.

q_0 *es el estado inicial,* $q_0 \in Q$.

B *es un símbolo especial que se denomina BLANCO,* $\mathtt{B} \in \Gamma$.

F *es el conjunto de estados finales (aceptadores),* $F \subseteq Q$.

En el primer ejemplo se tendría:

$\Sigma = \{0, 1\}$, $\Gamma = \{0, 1, \mathtt{B}\}$, *Q =* $\{q_0, q_1, q_2\}$, *F =* $\{q_2\}$ *y f, función de transición, según queda descrita en la tabla,*

y, en el segundo,

$\Sigma = \{0, 1\}$, $\Gamma = \{0, 1, X, Y, \mathtt{B}\}$, *Q =* $\{q_0, q_1, q_2, q_3, q_4\}$, *F =* $\{q_4\}$ *y f, función de transición, según queda descrita en la tabla.*

Definición 7.2 (Descripción Instantánea) *Una descripción instantánea es una cadena que pertenece a* $\Gamma^* Q \Gamma^*$.

Las descripciones instantáneas, DI, se utilizan para describir el estado de la máquina. En el contexto del modelo de Máquina de Turing una DI representa la cadena que en un momento dado se encuentra en la cinta y un identificador de estado que, además de indicar en qué estado se halla la máquina, indica la posición del cabezal de L/E: el cabezal siempre está sobre el símbolo inmediatamente a la derecha del identificador de estado. Como la cinta es infinita, sólo se representarán los símbolos significativos.

Convenios para interpretar las DI:

1. Si la transición es del tipo $f(q, x_i) = (p, Y, L)$, la evolución se representa

$$x_1 x_2 \cdots x_{i-1} \mathbf{q} x_i x_{i+1} \cdots x_n \vdash x_1 x_2 \cdots x_{i-2} \mathbf{p} x_{i-1} Y x_{i+1} \cdots x_n$$

2. Si la transición es del tipo $f(q, x_i) = (p, Y, R)$, la evolución se representa

$$x_1 x_2 \cdots x_{i-1} \mathbf{q} x_i x_{i+1} \cdots x_n \vdash x_1 x_2 \cdots x_{i-2} x_{i-1} Y \mathbf{p} x_{i+1} \cdots x_n$$

En el primer ejemplo, la evolución de la MT diseñada (con la entrada indicada) queda descrita por la siguiente secuencia de DIs,

$$q_0 00110111000000\mathtt{B} \vdash 0 q_0 0110111000000\mathtt{B} \vdash 00 q_0 110111000000\mathtt{B} \vdash$$
$$000 q_1 10111000000\mathtt{B} \vdash 0001 q_1 0111000000\mathtt{B} \vdash 00011 q_2 1110000 0\mathtt{B}$$

Definición 7.3 *Dos Descripciones Instantáneas de la Máquina de Turing M, I_1 e I_2, están en relación si desde I_1 se puede alcanzar I_2 en un sólo paso ($I_1 \vdash I_2$).*

El cierre reflexivo-transitivo de una relación se representa como $I_1 \overset{*}{\vdash} I_2$; el cierre transitivo, como $I_1 \overset{+}{\vdash} I_2$.

Las definiciones anteriores permiten definir formalmente qué lenguaje reconoce una Máquina de Turing:

Definición 7.4 *Dada la Máquina de Turing,*

$$M = \langle \Sigma,\ Q,\ \Gamma,\ f,\ q_0,\ \mathtt{B},\ F \rangle$$

el lenguaje asociado a esta máquina, al que se denomina $L(M)$ se define como,

$$L(M) \ = \ \{x \in \Sigma^* \mid q_0 x \ \overset{*}{\vdash} \ \alpha_1 p \alpha_2,\ p \in F,\ \alpha_1, \alpha_2 \in \Gamma^*\}$$

Es decir, $L(M)$ es el conjunto de cadenas de entrada que llevan a la máquina a un estado final, independientemente de la posición que ocupe el cabezal.

7.2. Técnicas para la Construcción de Máquinas de Turing.

El objetivo de esta sección es estudiar técnicas que facilitan la construcción de Máquinas de Turing, pero que no afectan a la potencia computacional del modelo ya que siempre se puede simular la solución obtenida mediante el modelo formal.

7.2.1. Almacenamiento en el Control Finito.

Consiste en asociar a determinados estados (o a todos) campos finitos de información adicional sobre las transiciones en el control finito.

Por lo tanto, un estado no sólo queda definido por su identificador, sino también por un número finito de campos de información (normalmente asociado a símbolos del alfabeto de la cinta). La idea es frenar un crecimiento excesivo de identificadores de estado, que pueda dificultar el entendimiento del comportamiento de la máquina. Suele ser útil en aquellos casos en los que se debe "recordar" qué símbolo se ha leído (y en los que la alternativa a usar esta técnica es realizar transiciones a distintos estados, dependiendo de que se lean distintos símbolos).

Ejemplo 3: Construir una Máquina de Turing capaz de reconocer el lenguaje

$$L = \{x \in \Sigma^+ \mid x = ay,\ a \notin y,\ a \in \Sigma\}$$

es decir, un lenguaje en el que el primer símbolo de la cadena no puede volver a aparecer en el resto. Supóngase que Σ = {0,1}*. Una posible idea es diseñar una MT tal que del estado inicial al leer un* 0 *pase al estado* q_1 *y a leer un* 1 *pase a* $q_2 \ldots$*, pero lo único que motiva esta diferencia entre los estados es el símbolo leído, es decir, la distinción de estados está motivada por la necesidad de poder distinguir de alguna forma entre leer primero un* 1 *o un* 0 *para saber que posteriormente no se va a poder repetir. Entonces, ¿por qué no realizar una transición a un estado que sirva para "recordar" cuál fue el símbolo leído en primer lugar? Por ejemplo,*

$$\begin{aligned}
\mathtt{f}([\mathtt{q}_0, \mathtt{B}], \mathtt{0}) &= ([\mathtt{q}_1, \mathtt{0}], \mathtt{0}, \mathtt{R}) \\
\mathtt{f}([\mathtt{q}_0, \mathtt{B}], \mathtt{1}) &= ([\mathtt{q}_1, \mathtt{1}], \mathtt{1}, \mathtt{R}) \\
\mathtt{f}([\mathtt{q}_1, \mathtt{0}], \mathtt{1}) &= ([\mathtt{q}_1, \mathtt{0}], \mathtt{1}, \mathtt{R}) \\
\mathtt{f}([\mathtt{q}_1, \mathtt{1}], \mathtt{0}) &= ([\mathtt{q}_1, \mathtt{1}], \mathtt{0}, \mathtt{R}) \\
\mathtt{f}([\mathtt{q}_1, \mathtt{0}], \mathtt{B}) &= ([\mathtt{q}_1, \mathtt{B}], \mathtt{B}, \mathtt{R}) \\
\mathtt{f}([\mathtt{q}_1, \mathtt{1}], \mathtt{B}) &= ([\mathtt{q}_1, \mathtt{B}], \mathtt{B}, \mathtt{R})
\end{aligned}$$

Con esto se tendría que Q = {$[\mathtt{q}_0, \mathtt{B}]$, $[\mathtt{q}_1, \mathtt{0}]$, $[\mathtt{q}_1, \mathtt{1}]$, $[\mathtt{q}_1, \mathtt{B}]$}*; el estado inicial es* $[\mathtt{q}_0, \mathtt{B}]$ *y el estado final es* $[\mathtt{q}_1, \mathtt{B}]$.

Este ejemplo puede parecer muy simple; para apreciar realmente la utilidad de esta técnica, se aconseja resolver el mismo problema cuando Σ *coincide con el alfabeto latino, por ejemplo.*

En un Máquina de Turing así diseñada, el conjunto de estados Q se puede ver como el resultado de realizar el producto cartesiano entre un conjunto de identificadores de estados y el conjunto de símbolos que sean significativos a la hora de realizar transiciones. Si se guarda un único campo de información se obtendrían pares [id estado, inf]; si se guardan dos campos de información se obtendrían triplas [id estado, inf1, inf2], y así sucesivamente.

Si se quiere obtener una Máquina de Turing estándar bastará con renombrar los pares (o triplas, etc.) obtenidos al realizar el diseño mediante la técnica descrita.

En el ejemplo anterior, de $Q = \{[\mathtt{q}_0, \mathtt{B}], [\mathtt{q}_1, \mathtt{0}], [\mathtt{q}_1, \mathtt{1}], [\mathtt{q}_1, \mathtt{B}]\}$ *se puede obtener el conjunto* $Q' = \{\mathtt{q}'_0, \mathtt{q}'_1, \mathtt{q}'_2, \mathtt{q}'_3\}$ *en el que el estado inicial es* $\mathtt{q}'_0$ *y el estado final es* $\mathtt{q}'_3$.

Es importante tener en cuenta que la información que se almacena en el control finito siempre será **finita**.

7.2.2. Cintas con Sectores Múltiples y Marcaje de símbolos.

En el diseño de algunas Máquinas de Turing puede resultar interesante considerar que la cinta está dividida en varios sectores; nótese que se tiene una cinta, pero es como si cada celda tuviera distintos "compartimentos". En este caso se asume que el cabezal tiene capacidad para leer/escribir todos los sectores a la vez: cada sector puede tener un símbolo y el cabezal los interpreta todos juntos (sería similar a interpretar que `000000101` es `5`). La ventaja de esta técnica es disminuir el número de símbolos del alfabeto de la cinta: con sólo dos símbolos y k sectores se podría representar lo mismo que con 2^k símbolos, por ejemplo. Dependiendo del sector, puede variar la interpretación.

En general, para describir las Máquinas de Turing así diseñadas se realizan tantos productos cartesianos del alfabeto como sectores aparezcan. El cabezal lee una k_tupla, por lo que el alfabeto de la cinta toma la forma,

$$\Gamma \;=\; \{[\alpha_1, \alpha_2, \ldots, \alpha_k] \mid [\alpha_1, \alpha_2, \ldots, \alpha_k] \in \Gamma_1 \times \Gamma_2 \times \ldots \times \Gamma_k\}$$

lo que lleva a que escribir las transiciones como

$$f(q_i, [\alpha_1, \alpha_2, \ldots, \alpha_k] = (q_j, [\alpha'_1, \alpha'_2, \ldots, \alpha'_k], L/R).$$

Evidentemente, para obtener una Máquina de Turing estándar, basta con renombrar cada una de las k_tuplas, eligiendo el número adecuado de símbolos.

Hay un caso particular de esta técnica, que suele utilizarse muy frecuentemente: la Máquina de Turing tiene dos sectores, de los cuales uno se utiliza para almacenar la cadena de entrada (y obtener la cadena de salida, si es el caso) y el segundo sólo admite una marca especial ($\surd$) ó `B`. La técnica permite marcar símbolos sin borrarlos.

$\surd$	B	$\surd$	B	B	$\cdots$
0	0	1	0	1	$\cdots$

Al utilizar esta técnica hay que redefinir el alfabeto de entrada y el de cinta, como el producto cartesiano entre el alfabeto original y el conjunto $\{\mathtt{B}, \surd\}$.

Como ejemplo de aplicación, considérese el problema propuesto en el segundo ejemplo, pero, en este caso, se marcarán los símbolos en lugar de sobreescribirlos con X e Y. De acuerdo a lo visto, se tendría

$$\Sigma = \{[\mathtt{B}, d] \mid d \in \{0,1\}\},\ \Gamma = \{[X, d] \mid X \in \{\mathtt{B}, \surd\},\ d \in \{0,1,\mathtt{B}\}\}$$

	[B,0]	[B,1]	[$\surd$,0]	[$\surd$,1]	[B,B]
q_0	(q_1, [$\surd$,0], R)			(q_3, [$\surd$,1], R)	
q_1	(q_1, [B,0], R)	(q_2, [$\surd$,1], L)		(q_1, [$\surd$,1], R)	
q_2	(q_2, [B,0], L)		(q_0, [$\surd$,0], R)	(q_2, [$\surd$,1], L)	
q_3				(q_3, [$\surd$,1], R)	(q_4, [B,B], L)
q_4					

7.2.3. Uso de Subrutinas.

La idea es la misma que cuando se trabaja en un lenguaje de alto nivel: aprovechar las ventajas del diseño modular para facilitar el diseño de la Máquina de Turing.

La base será la descomposición de la tarea a realizar en tareas más simples; cada una de estas subtareas se describirá en una tabla de transición propia (la "subrutina"). En la tabla de transición que describe a la Máquina de Turing que resuelve el problema completo habrá estados de "llamada a subrutina", q_{ll}, caracterizados por que suponen la transición al estado inicial de una "subrutina". El estado final de ésta será realmente un "estado de salida" que permite transitar hacia un estado de "return" en la Máquina de Turing "principal".

"Subrutina":

	α	β	$\cdots$
q_i	(q_j,...)		
$\cdots$			$\cdots$
q_f		(q_r,...)	$\cdots$

MT "Principal":

	γ	ω	δ	$\cdots$
q_r	(q_s,...)			$\ldots$
$\cdots$				$\cdots$
q_{ll}		(q_j,...)	(q_i,...)	$\ldots$

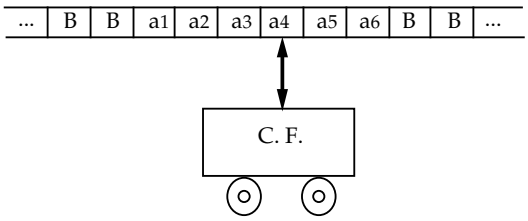
Para obtener una Máquina de Turing estándar, bastaría con reescribir todas las funciones de transición como una única función de transición más grande.

7.3. Máquinas de Turing Modificadas.

La Máquina de Turing es un autómata de definición y funcionamiento muy sencillo. El objetivo de esta sección es estudiar si es posible aumentar su poder computacional añadiéndole nuevos elementos o si esto simplemente reportará ventajas desde el punto de vista de una mayor rapidez en los cálculos o una mayor facilidad para realizarlos.

7.3.1. Máquina de Turing con Cinta Infinita en Ambos Sentidos.

La diferencia con el modelo general radica en que la cinta es infinita en ambos sentidos.



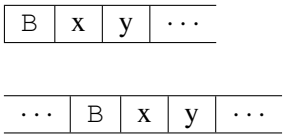
La definición es la misma,

$$M = \langle \Sigma,\ Q,\ \Gamma,\ f,\ q_0,\ \mathtt{B},\ F \rangle$$

pero, por ejemplo, se puede tener la transición

$$f(q_0, x)\ =\ (q_1, y, L)$$

que bloquearía a una Máquina de Turing limitada a la izquierda, o puede haber matices de interpretación:



En el modelo general ese B es significativo y formará parte de una DI, qBxy $\vdash$ Bpxy. En este modelo, el B no es significativo: formará parte de los infinitos que hay en la cinta

Teorema 7.1 *El lenguaje L es reconocido por una Máquina de Turing con cinta infinita en ambos sentidos $\Leftrightarrow$ L es reconocido por una Máquina de Turing con cinta infinita por la derecha.*
(Ambos modelos tienen el mismo poder computacional)

Demostración:[1]

1. *Si L es reconocido por una Máquina de Turing con cinta infinita por la derecha, L es reconocido por una Máquina de Turing con cinta infinita*. Para ello se define un símbolo especial (SE) que no pertenezca al alfabeto de la cinta. Si se detecta en una celda, la máquina se debería desconectar puesto que se ha realizado un movimiento a la izquierda de la primera celda. Hacia la derecha no se pone ninguna limitación.

		x_1	x_2	$\cdots$

(M_1)

$\cdots$	SE	x_1	x_2	$\cdots$

(M_2)

2. *Si L es reconocido por una Máquina de Turing con cinta infinita, L es reconocido por una Máquina de Turing con cinta infinita por la derecha*. La idea intuitiva de la demostración sería obtener una cinta limitada al "cortar" la cinta infinita en ambos sentidos y "pegar" ambos trozos haciendo coincidir el corte:

$\cdots$	z_{-3}	z_{-2}	z_{-1}	z_0	z_1	z_2	z_3	$\cdots$

(M_1)

z_0	z_1	z_2	z_3	$\cdots$
SE	z_{-1}	z_{-2}	z_{-3}	$\cdots$

(M_2)

Se ha denominado M_1 a la máquina con cinta infinita en ambos sentidos, y M_2 a la máquina limitada por la izquierda. La máquina M_2 tiene dos sectores, pero en un momento dado sólo "trabaja" con uno de ellos (es decir, en un sector siempre se escribe lo mismo que se lee, y sólo en el otro la escritura es realmente significativa). Para demostrar el enunciado basta con ver cómo "traducir" los movimientos de M_1 en la máquina M_2; la idea es la siguiente:

- Cualquier movimiento a la dcha. de $\mathbf{z_0}$, hacia R ó L en $\mathbf{M_1}$, es un movimiento a la dcha. de z_0, hacia R ó L, "trabajo" en sector superior en M_2.

Por ejemplo,

- *el movimiento de $\mathbf{z_1}$ a $\mathbf{z_2}$ (R) en $\mathbf{M_1}$, es un movimiento a la dcha. (de z_1 a z_2), "trabajo" en sector superior en M_2,*
- *el movimiento de $\mathbf{z_2}$ a $\mathbf{z_1}$ (L) en $\mathbf{M_1}$, es un movimiento a la izqda. (de z_2 a z_1), "trabajo" en sector superior en M_2.*

- El movimiento de $\mathbf{z_0}$ a $\mathbf{z_{-1}}$ en $\mathbf{M_1}$, (L), es un movimiento a la dcha. (de SE a z_{-1}), "trabajo" en sector superior enM_2.

- Cualquier movimiento a la izqda. de $\mathbf{z_{-1}}$, hacia R ó L en $\mathbf{M_1}$, es un movimiento a la dcha. de z_0, hacia L ó R, "trabajo" en sector inferior en M_2.

Por ejemplo,

[1] A lo largo de este tema, no se estudiarán las demostraciones formales, sino que se darán sus ideas básicas. Las demostraciones formales supondrían el establecimiento de un método constructivo para definir la función de transición de una MT, M_1, a partir de otra MT, M_2. Ello es demasiado farragoso y no contribuiría a una mejor comprensión de los enunciados

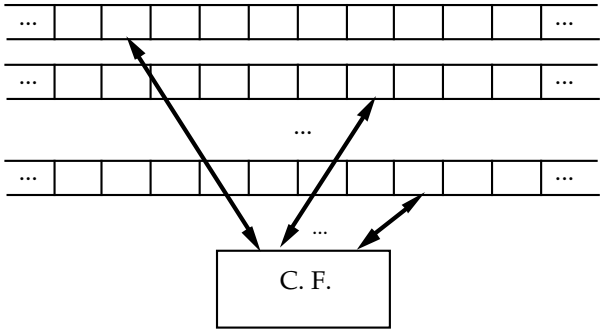
- *el movimiento de* $\mathbf{z_{-1}}$ *a* $\mathbf{z_{-2}}$ *(L) en* $\mathbf{M_1}$*, es un movimiento a la dcha. (de* z_{-1} *a* z_{-2}*), "trabajo" en sector inferior en* M_2*,*
- *el movimiento de* $\mathbf{z_{-2}}$ *a* $\mathbf{z_{-1}}$ *(R) en* $\mathbf{M_1}$*, es un movimiento a la izqda. (de* z_{-2} *a* z_{-1}*), "trabajo" en sector inferior en* M_2*.*

- El movimiento de $\mathbf{z_{-1}}$ a $\mathbf{z_0}$ en $\mathbf{M_1}$, (R), es un movimiento a la izqda. (de z_{-1} a SE), "trabajo" en sector inferior en M_2.

♣

7.3.2. Máquina de Turing Multicinta.

En este modelo, la máquina de Turing tiene k cintas, infinitas en ambos sentidos, y k cabezales de L/E,



Sólo hay una entrada de información, en la primera cinta. Los tres pasos asociados a cada transición son ahora:

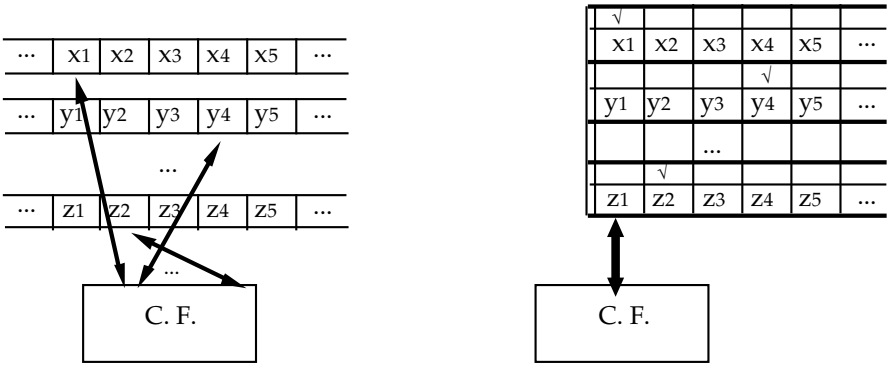
1. transición de estado,
2. escribir un símbolo en cada una de las celdas sobre las que están los cabezales de L/E,
3. el movimiento de cada cabezal es independiente y será R, L ó NADA (Z).

Teorema 7.2 *El lenguaje L es reconocido por una Máquina de Turing Multicinta* $\Leftrightarrow$ *L es reconocido por una Máquina de Turing de una sola cinta.*
(Ambos modelos tienen el mismo poder computacional)

Demostración:(Idea Intuitiva)

1. *Si L es reconocido por una Máquina de Turing de una sola cinta, L es reconocido por una Máquina de Turing Multicinta*. Basta con hacer funcionar una sola cinta de la Máquina de Turing multicinta, la que se utiliza para realizar la entrada de información.

2. *Si L es reconocido por una Máquina de Turing Multicinta, L es reconocido por una Máquina de Turing de una sola cinta*. Gracias al teorema 7.1 el hecho de que las cintas sean o no infinitas en ambos sentidos es indiferente. La idea básica, entonces, será pasar de un Máquina de Turing multicinta con k cintas a una Máquina de Turing de una cinta pero con 2k sectores. Así, por cada cinta de la Máquina de Turing multicinta se tendrá en un sector, la información, y en el otro, una marca que indique la posición del cabezal de L/E de la máquina original.



M1 M2

La máquina M_2 simula el funcionamiento de la máquina M_1 por barridos. Se necesitará un contador de valor inicial k y, suponiendo que el cabezal de M_2 está ajustado sobre la marca situada más a la izquierda, se comienza un movimiento hacia la derecha. Cada vez que se detecte una marca de cabezal de L/E, se decrementa el contador y se almacena el símbolo asociado en el control finito. El barrido acaba cuando el contador valga 0. Habrá k símbolos almacenados en el CF de M_2. Se realizaría la misma transición que se realizaría en M_1. Para poder llevar a cabo los movimientos, habrá que realizar un nuevo barrido hacia la izquierda, trasladando las marcas hacia donde haya que trasladarlas (L, R ó Z) y sobreescribiendo los símbolos que haya que reescribir; una vez completado el barrido, se cambia de estado.

♣

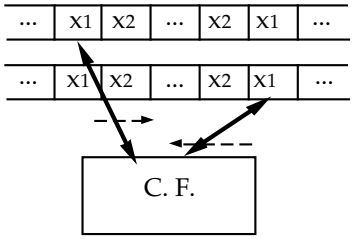
Nótese que un movimiento de M_1 equivale a varios movimientos de M_2. Este modelo tiene el mismo poder computacional, pero suele ser más eficiente que el modelo con una sola cinta.

Ejemplo:

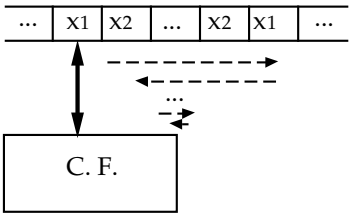
Si se calcula el número de movimientos necesarios para reconocer una cadena del lenguaje

$$L = \{ww^{-1} \mid w \in \Sigma^*\}$$

con respecto a su longitud, en una máquina con 2 cintas el resultado sería lineal y en una máquina con una cinta, cuadrático.



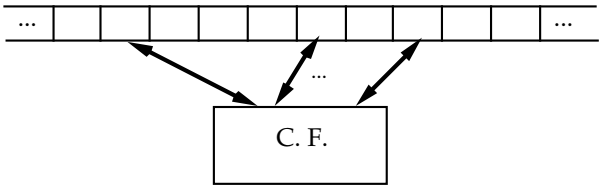
M1: lineal



M2: cuadrático

7.3.3. Máquina de Turing con Múltiples Cabezales.

Tiene k cabezales de L/E, como la multicinta, pero con una sola cinta. Los cabezales operan todos de forma independiente. Como en las Máquinas de Turing multicinta, se admiten movimientos L, R ó Z.



Teorema 7.3 *Un lenguaje L es reconocido por una Máquina de Turing de múltiples cabezales ⇔ L es reconocido por una Máquina de Turing de un cabezal.*
(Ambos modelos tienen el mismo poder computacional)

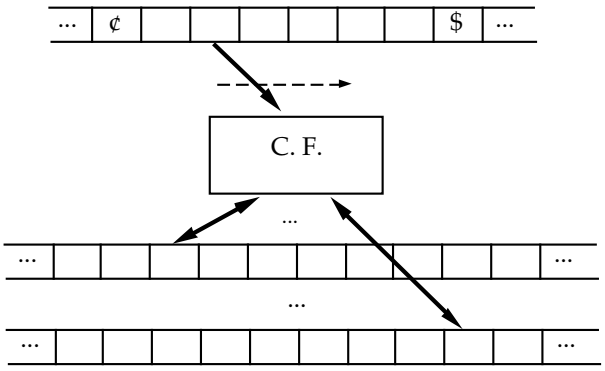
Demostración:(Idea Intuitiva)

1. *Si L es reconocido por una Máquina de Turing unicabezal, L es reconocido por una Máquina de Turing multicabezal*. Basta con trabajar con un sólo cabezal en la de múltiples cabezales.
2. *Si L es reconocido por una Máquina de Turing multicabezal, L es reconocido por una Máquina de Turing unicabezal*. La simulación es similar a la realizada para las Máquinas de Turing multicinta: se utiliza una cinta con k+1 sectores, k para marcar las posiciones de los distintos cabezales y uno para la información de la cinta.

♣

7.3.4. Máquina de Turing Offline.

Es un caso particular de las Máquinas de Turing multicinta: tienen una cinta especial de sólo lectura en la que el cabezal, que sólo puede moverse hacia la derecha, no puede moverse de la zona delimitada por un par de símbolos especiales.



Para simularla, no hace falta más que construir una Máquina de Turing multicinta con una cinta especial sobre la que se copia la información de entrada. El cabezal de esa cinta sólo lee (sobreescribe siempre el mismo símbolo que lee) y para al encontrar el símbolo \$. A partir de la Máquina de Turing multicinta se puede aplicar el teorema 7.2 para demostrar que su poder computacional es el mismo que el de una Máquina de Turing estándar.

7.3.5. Máquina de Turing No Determinista.

Es una Máquina de Turing con cinta limitada a la izquierda, que se caracteriza por tener asociada más de una transición desde algún estado con el mismo símbolo,

$$\begin{aligned}
&\mathtt{f}(\mathtt{q_i},\mathtt{a}) = \{(\mathtt{q_i},\mathtt{b},\mathtt{L}),(\mathtt{q_i},\mathtt{a},\mathtt{R}),(\mathtt{q_j},\mathtt{a},\mathtt{R}),(\mathtt{q_j},\mathtt{b},\mathtt{R})\}\\
&\mathtt{f}(\mathtt{q_i},\mathtt{b}) = \{(\mathtt{q_i},\mathtt{b},\mathtt{R})\}\\
&\mathtt{f}(\mathtt{q_j},\mathtt{a}) = \{(\mathtt{q_j},\mathtt{b},\mathtt{L}),(\mathtt{q_j},\mathtt{a},\mathtt{R})\}\\
&\vdots
\end{aligned}$$

El número de transiciones asociado a cada par estado/símbolo *siempre es finito*.

Teorema 7.4 *El lenguaje L es reconocido por una Máquina de Turing No Determinista ⇔ L es reconocido por una Máquina de Turing Determinista.*
(Ambos modelos tienen el mismo poder computacional)

Demostración:(Idea Intuitiva)

1. *Si L es reconocido por una Máquina de Turing Determinista, L es reconocido por una Máquina de Turing No Determinista*. Las Máquinas de Turing deterministas son Máquinas de Turing no deterministas en las que sólo hay una transición por cada par estado/símbolo.

2. *Si L es reconocido por una Máquina de Turing No Determinista, L es reconocido por una Máquina de Turing Determinista*. La demostración consiste en determinar cómo una Máquina de Turing determinista puede simular el comportamiento de una

Máquina de Turing no determinista. Para ello, en primer lugar, y ya que el número de transiciones asociadas a cada par estado/símbolo es finito, se determina **r**, el número máximo de opciones asociadas a las transiciones (en el ejemplo anterior, r=4). Además, se necesita disponer de una Máquina de Turing determinista con 3 cintas, limitadas a la izquierda.

La **primera cinta**, recoge la información de entrada, la cadena a reconocer. La **segunda cinta** sirve para llevar la cuenta de qué opciones se van tomando. Para ello, sobre esa cinta hay que ir generando cadenas del alfabeto {1, 2, .., r} por orden numérico:

- primero, todas las cadenas de longitud 1,

  1, 2, 3, ..., r

- segundo, todas las cadenas de longitud 2,

  11, 12, 13, ..., 1r, 21, 22, 23, ..., 2r, ..., r1, r2, r3, ..., rr

- tercero, todas las cadenas de longitud 3,

  111, 112, 113, ..., 11r, 121, 122, 123 ..., 12r, ..., 1r1, 1r2, ..., 1rr, ..., rr1, rr2, rr3, ..., rrr

- y, en general, en el paso i-ésimo se generarían todas las cadenas de longitud i,

  11..1, 11..2, ..., 11..r, 11..21, 11..22, ..., 11..r1, .. etc.

Sobre la **tercera cinta** se desarrolla la simulación propiamente dicha. Cada vez que se genera una secuencia en la cinta 2, se copia la cadena de entrada en la cinta 3. La secuencia de la cinta 2 indica qué transición concreta se elige cada vez. Si, por ejemplo, en la cinta 2 está la secuencia

11231..

quiere decir que la primera vez que aplique la función de transición se aplica la primera transición de entre las posibles. La segunda también se aplica la primera; la tercera, se debe aplicar la segunda, la cuarta se debe aplicar la tercera, la quinta se debe aplicar la primera[2]...

Con este "chivato", la máquina operaría sobre la cinta 3. Cada vez que se prueba una secuencia y no se llega a un estado de aceptación, se genera la siguiente y se vuelve a comenzar la simulación. Cuando se encuentra una secuencia que permite aceptar la cadena, la máquina para y acepta.

Si la cadena es aceptada por la Máquina de Turing no determinista, es porque existe una secuencia de aplicaciones de la función de transición que conducen a un estado final. Como en la cinta 2 se van generando todas las posibles secuencias, esta nueva Máquina de Turing determinista alguna vez tendrá que encontrar la correcta y parar aceptando.

[2]Nótese que se producirán secuencias imposibles. Se debe tener en cuenta que r es un máximo y no todas las transiciones tendrán tantas opciones. Al generar todas las secuencias del alfabeto {1,2,..,r}, es posible que una secuencia obligue, por ej., a tomar la tercera opción en un par estado/símbolo que sólo tiene dos. Entonces, se desecha la secuencia y se pasa a la siguiente.

7.3.6. Máquinas de Turing Restringidas.

El objetivo de las subsecciones anteriores era poner de relieve que diferentes mejoras físicas sobre el modelo de Máquina de Turing, no se traducen en un mayor poder computacional, sino en una mayor eficiencia en el cálculo o en una mayor simplicidad de diseño.

De la misma forma que podría creerse que introduciendo mejoras físicas en el modelo tendría que aumentar su poder computacional, puede parecer que si se considera un modelo más restringido, éste debe disminuir. Sin embargo, es posible establecer restricciones sobre el número de estados o el número de símbolos del alfabeto de la Máquina de Turing, sin que esto ocurra[3].

En concreto, una Máquina de Turing sin restricciones sobre el alfabeto, con una cinta y sólo tres estados puede simular el comportamiento de cualquier Máquina de Turing. Si lo que se restringe es el número de símbolos, se puede probar el siguiente resultado:

Teorema 7.5 *Si un lenguaje L, $L \subseteq (0+1)^*$, es reconocido por una Máquina de Turing, entonces L es aceptado por una Máquina de Turing con una sola cinta y alfabeto de cinta restringido a los símbolos* $\{0, 1, \mathtt{B}\}$.

Demostración:(Idea intuitiva)

Sea $L = L(M_1)$, con $M_1 = \langle \{0,1\}, Q, \Gamma, f, q_0, B, F \rangle$. Bajo el supuesto de que Γ tiene entre $2^{k-1}+1$ y 2^k símbolos, son suficientes *k* bits para codificar cualquier símbolo[4] de Γ. La Máquina de Turing M_2, con alfabeto de cinta restringido a $\{0,1,\mathtt{B}\}$, debe simular el comportamiento de M_1.

Para ello, la cinta de M_2 consistirá en una secuencia de códigos representando los símbolos de M_1. El control finito de M_2 debe recordar tanto el estado de M_1 como la posición de su cabezal de L/E, módulo *k*, de forma que sepa cuando se encuentra al principio de un código de un símbolo de M_1.

Al comenzar la simulación de un movimiento de M_1, el cabezal de M_2 debe estar situado al comienzo del código binario de un símbolo de M_1. Moviéndose hacia la derecha, M_2 lee los *k-1* símbolos siguientes para determinar la transición que debe realizar M_1.

Una vez que se sabe qué símbolo escribiría M_1, M_2 reemplaza los *k* símbolos leídos para reemplazarlos por el código del nuevo símbolo, moviéndose hacia la izquierda; se colocaría sobre el comienzo del siguiente código binario a interpretar (dependiendo de que el movimiento de M_1 sea L ó R) y realizaría la misma transición de estado. Si el estado es

[3] No simultáneamente: si se restringe el alfabeto de cinta, el número de cintas y el número de estados simultáneamente, el resultado sería un número finito de posibles MT que, evidentemente, no pueden tener el mismo poder que el modelo general, con un número infinito de posibles MT.

[4] Hay que tener en cuenta que $\Sigma \subset \Gamma$; es decir, sobre 0, 1 y B también se realizará la codificación.

final, M_2 acepta y, si no, pasa a simular el siguiente movimiento de M_1.

Hay un caso especial en esta simulación, que sería el caso en el que M_1 alcanzara un blanco en la cinta (por ejemplo, llegar a la posición donde se acaba la cadena). La máquina M_2 también alcanzaría una celda en blanco, pero para realizar fielmente la simulación, debe escribir en esa celda y en las *k-1* siguientes el código binario correspondiente al símbolo `B` de M_1.

Es preciso puntualizar, además, que la entrada de M_2 y M_1 no pueden ser la misma cadena. Es decir, la cadena *w*, $w \in (0+1)^*$, que es la entrada de M_1, debe codificarse también para ser una cadena de entrada que M_2 acepte. Por lo tanto, antes de la simulación se debe proceder a reescribir *w*, codificando cada uno de sus símbolos en *k* bits.

♣

Puesto que la misma técnica de codificación se puede aplicar sobre cualquier alfabeto, se establece el siguiente corolario:

Corolario 7.1 *Si un lenguaje L, sobre cualquier alfabeto, es reconocido por una Máquina de Turing, entonces L es aceptado por una Máquina de Turing offline con una sola cinta, además de la de entrada, y alfabeto de cinta restringido a los símbolos* $\{0, 1, \mathtt{B}\}$.

7.4. Relación de la Máquina de Turing con otros Autómatas.

Tal y como se ha visto en la sección anterior, las distintas variaciones del modelo de Máquina de Turing tienen **todas** el **mismo poder computacional**. El objetivo de esta sección es evidenciar que hay otros modelos que también tienen el mismo poder computacional que las Máquinas de Turing. Cabe destacar las variaciones del modelo de Autómata de Pilas, ya que de los autómatas estudiados, es el único que dispone de un elemento, la pila, en el que es posible escribir símbolos.

De hecho, un Autómata de Pila se puede simular con una Máquina de Turing, en general no determinista, con una cinta de sólo lectura (que sería la cinta del autómata), en la cual el movimiento se restringe a la derecha, y una cinta de trabajo, limitada a la izquierda, que simulará la pila, de forma que un movimiento a la derecha siempre supondrá escribir un símbolo de $\Gamma - \{\mathtt{B}\}$ y se asociará a *apilar* y un movimiento a la izquierda siempre supondrá escribir `B` y se asociará a *desapilar*. La función de transición se construye traduciendo la original bajo estas premisas.

La figura 7.1 esquematiza las ideas básicas de la simulación.

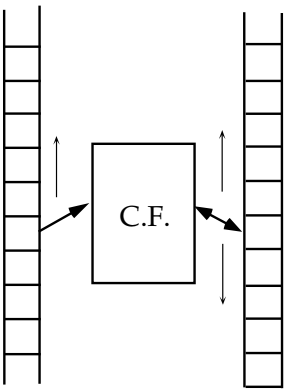


Figura 7.1: Una Máquina de Turing con dos cintas, una de sólo lectura, permite simular el comportamiento de un Autómata de Pilas.

Lema 7.1 *Una Máquina de Turing con una sola cinta puede ser simulada por un Autómata de Pila determinista con dos pilas.*

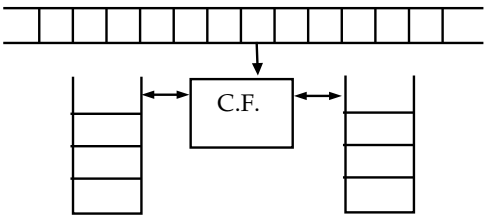
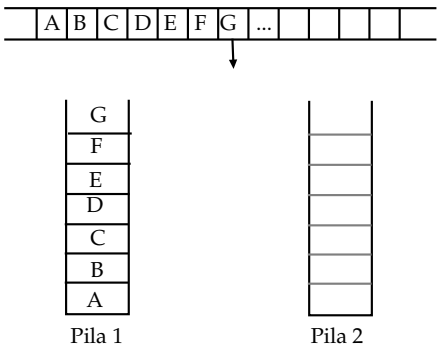


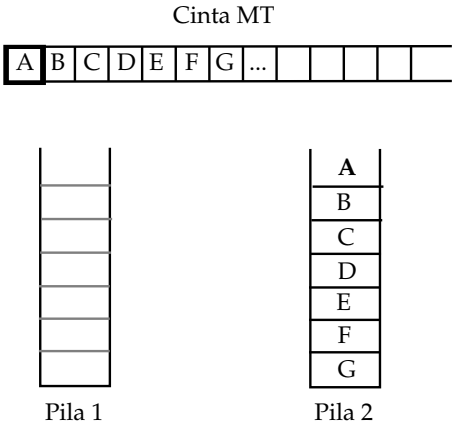
Figura 7.2: Autómata de Pilas con dos pilas.

Demostración:(Idea Intuitiva)

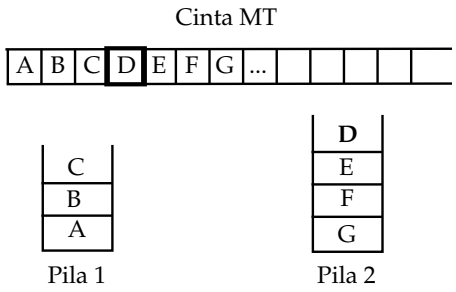
Para realizar la simulación, primero se copia en la cinta de entrada del Autómata de Pila el contenido de la cinta de entrada de la Máquina de Turing; se introduce en una pila dicha cadena y se invierte sobre la otra.



A continuación, comenzaría realmente la simulación.



Para realizarla no hay más que conseguir que en el tope de la Pila 2 esté el símbolo que estaría debajo del cabezal de la Máquina de Turing:



Para simular un movimiento a la derecha, se desapila de la Pila 2 y se apila en la Pila 1 el símbolo a escribir en la cinta. Para simular un movimiento a la izquierda, se sobreescribe el símbolo que está en el tope de la Pila 2 con el símbolo que se escribiría en la cinta y se desapila de la Pila 1 un símbolo que se apila en la Pila 2.

Capítulo 8

Computabilidad

Índice General

8.1. Lenguajes Recursivos y Funciones Computables. 129
8.1.1. Introducción. 129
8.1.2. La Máquina de Turing como Reconocedor de Lenguajes. 132
8.1.3. La Máquina de Turing como Calculador de Funciones. 133
8.2. La Tesis de Church. 134
8.2.1. El Modelo RAM. 135
8.3. La Máquina de Turing como Generador. 136
8.3.1. Dos Máquinas de Turing Generadoras Básicas. 137
8.4. Caracterización de L.R. y L.R.E. 139
8.4.1. Caracterización de los L.R.E. mediante Generadores. 139
8.4.2. Caracterización de los L.R. mediante Generadores. 141
8.5. Propiedades de L. R. y L. R. E. 144

8.1. Lenguajes Recursivos y Funciones Computables.

8.1.1. Introducción.

En 1900, el matemático David Hilbert propuso el llamado *Problema de la Decisión* (*Entscheidungsproblem*), formulado como *"Descubrir un método general para determinar si una fórmula de lógica formal puede o no satisfacerse"*. El interés que podía tener tal problema se debía al trabajo desarrollado desde 1879 por Gottlob Frege, para reducir los enunciados matemáticos a enunciados de la lógica formal: si los enunciados matemáticos se reducen a fórmulas lógicas y hay un método general para determinar si una fórmula se puede satisfacer o no, entonces habría un método general para determinar si un enunciado (teorema) es cierto o no. Enunciados como la Conjetura de Goldbach o la Conjetura de los Números Perfectos, dejarían de ser conjeturas: al aplicarles este método, pasarían a ser

teoremas. Bastaría con aplicar tal método "mecánicamente" para determinar si un enunciado es o no es un teorema. Ese fue el inicio del *"formalismo"*, escuela matemática que pretendía convertir todas las matemáticas en un gran *sistema formal*, un sistema formado por *axiomas* y *reglas*, en el que las expresiones se representan mediante símbolos y sólo se puede operar usando las reglas del sistema.

Se abrió así una vía de investigación que, 36 años más tarde, llevó a la demostración de que tal problema, la determinación de ese método "general", era imposible de resolver. Sin embargo, los resultados obtenidos hasta llegar a esta conclusión son de gran importancia.

El primer resultado de interés surge en 1901, cuando Bertrand Russell descubrió una paradoja irrefutable en la teoría elemental de los conjuntos, que se puede formular como *"El conjunto de los conjuntos que no pertenecen a sí mismos, ¿pertenece a sí mismo?"*. Esta paradoja consiguió anular una de las reglas básicas instituidas por Frege para reducir los enunciados matemáticos a enunciados lógicos. Russell era un gran admirador del trabajo de Frege, y consiguió salvar su trabajo cuando publicó, junto a Whitehead, su obra *"Principia Mathematica"* en la que introduce una *teoría de conjuntos axiomática* que elimina su paradoja (introduce restricciones en cuanto a cómo definir un conjunto, en concreto a conjuntos que se definan a sí mismos).

En 1928, Hilbert y Ackermann definieron el cálculo de predicados de primer orden y en 1931, Kurt Gödel demostró en su tesis doctoral el llamado *Teorema de la Completitud*: el cálculo de predicados de primer orden es completo, es decir, cada predicado válido o es cierto o es falso. Como consecuencia de su trabajo, también pudo demostrar que *un sistema axiomático no es completo* ya que no puede contener todos los enunciados verdaderos de la teoría que pretende formalizar.

> *Cualquier sistema consistente de lógica formal lo bastante potente para abarcar los enunciados de toda la aritmética ordinaria tiene que contener enunciados verdaderos que no pueden ser demostrados dentro del sistema.*

Exponer, y entender, el trabajo de Gödel no es sencillo. Una idea aproximada de sus resultados sería saber que llegó en el sistema a construir una cadena que era la formulación equivalente a "*este teorema no se puede demostrar*". Pero Gödel pudo demostrarlo saliendo *fuera* del sistema.

La consecuencia inmediata del trabajo de Gödel fue que la propuesta de Hilbert se eliminó de un plumazo: no puede existir un método que permita decidir si un enunciado arbitrariamente dado es cierto o falso, ya que si lo hubiera demostraría todos los enunciados ciertos y tal demostración es imposible en un sistema incompleto.

A partir de ahí, la investigación continuó por un problema menos ambicioso, el *Problema de la Demostrabilidad*. No es posible encontrar un método que permita decidir si un enunciado es cierto o no, pero *¿existe un único método que permita demostrar, a partir de un sistema de axiomas lógicos, los enunciados matemáticos demostrables?*.

Esta línea de trabajo llevó a Alonzo Church, junto con Kleene y Rosser a definir el $\lambda - calculus$, un lenguaje formal que permite expresar funciones matemáticas (y en el que

se basa el lenguaje LISP). Kleene demostró que todas las funciones utilizadas por Gödel, las *funciones recursivas*, pueden expresarse en $\lambda - calculus$. Esto llevó a Church a enunciar que si una función matemática es computable (se puede calcular su valor para todo número perteneciente a su dominio de definición), se puede expresar en $\lambda - calculus$. Esta suposición se conoce como la *Tesis de Church* y se suele enunciar como

> *Toda función efectivamente calculable es recursiva.*

Church, además, demostró que en el caso de que hubiera una función matemática expresable en $\lambda - calculus$ pero no computable, entonces no habría método alguno para determinar si un enunciado matemático dado es o no demostrable (y, si no sabemos si un enunciado es o no demostrable, ¿sobre qué se aplicaría el método único de demostración de los enunciados demostrables? ;-). En 1936 publicó una fórmula con esas características, poniendo punto final a la propuesta de Hilbert.

Estos resultados se produjeron de forma paralela al trabajo desarrollado por Alan Turing. Familiarizado con el Problema de la Decisión, lo atacó desde otro punto de vista, intentando formalizar el concepto de *método*. Su definición de método coincide con el concepto actual de *algoritmo*, un *procedimiento que puede ser ejecutado mecánicamente sin intervención creativa alguna*.

De esta idea surgió el modelo de computación que se conoce como Máquina de Turing, que permite la descomposición de cualquier algoritmo en una secuencia de pasos muy simples. Del modelo se desprende la definición formal de computabilidad dada por Turing,

> *Una función es computable si existe una Máquina de Turing que la calcule en un número finito de pasos.*

Trabajando de forma independiente a otros autores, Turing comprendió que había una relación entre el problema de Hilbert y el hecho de asegurar que una función es computable. Gracias a su modelo, Turing pudo llegar a las mismas conclusiones que Church, de una forma más directa. Para ello se basó en los resultados de Georg Cantor sobre conjuntos contables. Cantor había definido un *conjunto contable*, como un conjunto infinito en el que cada objeto se puede asociar, de forma biunívoca, a un elemento del conjunto de los enteros positivos.

Abstrayendo el modelo mecánico, la Máquina de Turing se puede identificar con la función que calcula, lo que permite definirla como una *función de enteros sobre enteros*. Una Máquina de Turing acepta un *conjunto finito* de caracteres de entrada, tiene un *número finito* de estados y, *si la función es computable*, acaba sus cálculos en un *número finito* de pasos. Por lo tanto, cualquier Máquina de Turing se puede describir mediante una *cadena finita de caracteres*, lo que lleva a la conclusión de que el número de máquinas de Turing (por lo tanto, el número de funciones computables) es infinito pero *contable*.

Pero del trabajo de Cantor también se deduce que hay infinitos conjuntos no contables (el de los números reales, por ejemplo). Entre estos, está el conjunto de todas las funciones de enteros sobre enteros. Por lo tanto, si el número de funciones de enteros sobre enteros es

no contable y el número de máquinas de Turing es contable, es evidente que es imposible que existan suficientes máquinas de Turing para calcularlas todas. Y sólo son funciones computables las que se pueden calcular mediante una Máquina de Turing.

El objetivo de este tema es, precisamente, formalizar el concepto de función computable. Para ello, primero se definirán las clases de lenguajes (y de funciones) que define la Máquina de Turing según su comportamiento. A continuación, se enunciará la Tesis de Church (ó de Turing-Church), además de presentar otros modelos equivalentes tanto al de Máquina de Turing como al de función recursiva. Para finalizar, se verá la caracterización de estas clases mediante Máquinas de Turing Generadoras, además de ver algunas propiedades de clausura.

8.1.2. La Máquina de Turing como Reconocedor de Lenguajes.

Se recuerda la definición de L(M), lenguaje reconocido por una Máquina de Turing, M:

Definición 8.1 *Dada la Máquina de Turing,*

$$M = \langle \Sigma,\ Q,\ \Gamma,\ f,\ q_0,\ \mathtt{B},\ F \rangle$$

el lenguaje asociado a esta máquina, al que llamaremos $L(M)$ se define como,

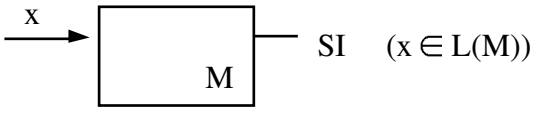
$$L(M) \ = \ \{x \in \Sigma^* \mid q_0 x \overset{*}{\vdash} \alpha_1 p \alpha_2,\ p \in F,\ \alpha_1, \alpha_2 \in \Gamma^*\}$$

(L(M) es el conjunto de cadenas de entrada que llevan a la máquina a un estado final, independientemente de la posición que ocupe el cabezal).

Si la cadena de entrada en una máquina M pertenece a *L(M)*, la máquina M siempre se detiene. Pero lo que ocurre cuando la cadena no pertenece al lenguaje da pie a la clasificación de los lenguajes en *Recursivos* y *Recursivamente Enumerables*.

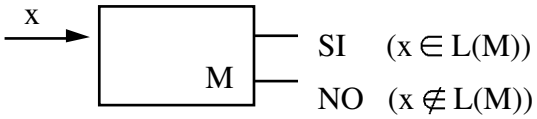
Definición 8.2 (Lenguaje Recursivamente Enumerable) *Sea M una Máquina de Turing; se dice que L = L(M) es un Lenguaje Recursivamente Enumerable si*

- $\forall\ x \in L$, *M se detiene en* $q \in F$,
- $\forall\ x \notin L$, *M se detiene en* $q \notin F$ *ó bien M no se detiene.*

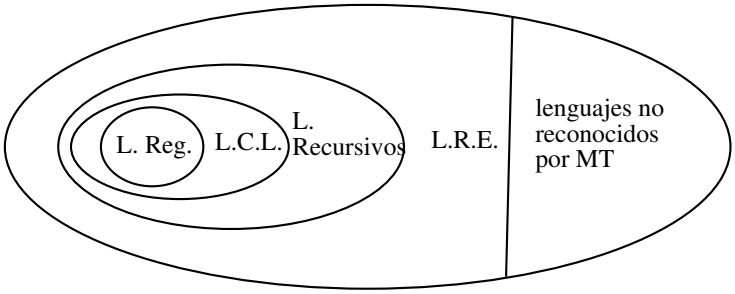


Definición 8.3 (Lenguaje Recursivo) *Se dice que L es un Lenguaje Recursivo si* existe al menos una *Máquina de Turing M, tal que L = L(M) y*

- $\forall\, x \in L$, *M se detiene en* $q \in F$,
- $\forall\, x \notin L$, *M se detiene en* $q \notin F$.



Con esta definición, el conjunto de todos los lenguajes queda de la siguiente forma



8.1.3. La Máquina de Turing como Calculador de Funciones.

Formalmente, la definición de la Máquina de Turing como calculador de funciones, toma la forma de una función k-aria *f*, con argumentos enteros y que devuelve un valor entero,

$$f : Z^k \longrightarrow Z,\ f(i_1, i_2, \ldots, i_k) = i.$$

Para normalizar su funcionamiento como calculador, se toma el convenio de representar los argumentos enteros como cadenas de 0's ($n \rightarrow 0^n$), utilizando el 1 como separador entre los argumentos:

$$f(i_1, i_2, \ldots, i_k) \hookrightarrow f(0^{i_1}10^{i_2}1 \ldots 0^{ik}).$$

El mismo convenio sirve para representar el resultado de la computación.

Definición 8.4 (Función Parcial) *Una función f,* $f : A \longrightarrow B$ *se dice que es una Función Parcial si* $\exists\, C \subseteq A,\ C \neq \emptyset$, *tal que* $\forall\, x \in C,\ \exists\, f(x)$ *(existe un subconjunto no vacío de A en el que todos los elementos tienen imagen calculable).*

Definición 8.5 (Función Total) *Una función f,* $f : A \longrightarrow B$ *se dice que es una Función Total si* $\forall\ x \in A,\ \exists\ f(x)$ *(se puede calcular la imagen para cualquier elemento de A).*

El modelo de Máquina de Turing define las *funciones recursivas parciales* ya que, es posible suministrarle argumentos con los cuales la Máquina de Turing no se detenga. De ahí que se establezca un paralelismo entre la definición de función parcial y la de lenguaje recursivamente enumerable: ambos establecen un funcionamiento de la Máquina de Turing que queda indeterminado, bien sea porque un argumento no tiene imagen, bien sea porque se proporciona a la máquina una cadena que no pertenece al lenguaje. .

Una Máquina de Turing que se detiene ante cualquier entrada define una *función recursiva total*, ya que para cualquier argumento hay una imagen. En términos de lenguajes, una función total se puede asimilar a un lenguaje recursivo.

Nota: todas las operaciones aritméticas habituales (+, *, /, n!, $\log(n)$, 2^n) son funciones recursivas totales.

8.2. La Tesis de Church.

Al estudiar en el capítulo 7 el modelo de Máquina de Turing y sus posibles extensiones, se demostró que cualquier posible mejora introducida no aumenta el poder computacional del modelo básico. Lo mismo se puede aplicar al estudio de otros autómatas que trabajan sobre el mismo principio: el desarrollo de un cálculo mediante la aplicación repetida de pasos finitos y completamente definidos.

De ahí que la Máquina de Turing sea un modelo formal de la noción de algoritmo: sólo se puede considerar como un algoritmo lo que sea posible realizar mediante una Máquina de Turing. Lo que Gödel demostró en 1931 es que en cualquier sistema de axiomas matemático hay enunciados que son ciertos pero cuya veracidad no se puede probar dentro del sistema. Church demostró en 1936 que no hay un método mecánico general para decidir si un enunciado es o no es demostrable. Ello hizo necesario disponer de un substituto matemático y exacto para la noción, intuitiva e informal, de computabilidad mecánica; Church utilizó su propio sistema, el $\lambda - calculus$. Turing llegó de forma independiente a los mismos resultados al mismo tiempo. Aún más: su explicación lógica del concepto de función computable en términos de una máquina abstracta, es superior y más simple que la de Church. Este modelo ha permitido probar los resultados sobre la incomputabilidad y desarrollar el teorema de incompletitud de Gödel, y sus consecuencias, en su totalidad.

El principio de que las Máquinas de Turing son versiones formales de algoritmos y que ningún procedimiento computacional se puede considerar un algoritmo salvo que sea posible presentarlo como una Máquina de Turing, es la versión informática de la Tesis de Church, o Tesis de Turing-Church, que ya se comentó en la Introducción:

> *La noción intuitiva de función computable puede identificarse con las funciones recursivas parciales. Las Máquinas de Turing tienen el poder suficiente para calcular cualquier función recursiva parcial.*

En esta tesis, la noción de función computable no pone límites ni al número de pasos necesarios, ni a la cantidad de espacio de almacenamiento necesario, para desarrollar la computación. No hay que confundir *computable* con *efectivamente computable*.

Esta es una tesis, no un teorema ni un resultado matemático: sólo establece la correspondencia entre un concepto informal (la computabilidad) y un concepto matemático (las funciones recursivas parciales). Es, por lo tanto, teóricamente posible que la Tesis de Church pueda quedar obsoleta en el futuro, si se propone un modelo alternativo para la noción de computabilidad que pueda desarrollar cálculos que no pueden realizarse mediante la Máquina de Turing. Pero no se considera probable.

De hecho, hay una serie de modelos lógicos desarrollados ($\lambda - calculus$, Sistemas de Post, Máquinas de Turing,...) y todos definen la misma clase de funciones, las funciones recursivas parciales. De entre estos modelos, hay uno más cercano al punto de vista de los programadores, el modelo RAM.

8.2.1. El Modelo RAM.

El modelo RAM (*Random Access Machine*), consiste en un *número infinito* de palabras de memoria, *numeradas* (0, 1, 2, ...), cada una de las cuales puede almacenar cualquier número entero, y un *número finito de registros aritméticos* capaces de almacenar cualquier entero. Los enteros podrían entenderse como las instrucciones (codificadas en binario) de un computador.

Eligiendo correctamente el conjunto de instrucciones y el tamaño del máximo entero representable en memoria, el modelo RAM puede simular cualquier computador existente. Veremos a continuación, que la Máquina de Turing tiene el mismo poder computacional que el modelo RAM.

Teorema 8.1 *Una Máquina de Turing puede simular una RAM, supuesto que las instrucciones RAM pueden ser simuladas (las conocemos y sabemos interpretar cada una de ella) por una Máquina de Turing.*

Demostración: (Idea Intuitiva)

La Máquina de Turing necesaria para la simulación será una Máquina de Turing multicinta. En una cinta, se almacenan las palabras de memoria de la RAM que tienen valores. El formato para almacenarlas podría ser

$$\mathtt{B}\#0 * v_0\#1 * v_1\#10 * v_2\#...\#i * v_i\#...$$

donde con v_i se representa el contenido, en binario, de la palabra *i* de memoria de la RAM. Nótese que, en un momento dado, sólo se está usando un número finito de palabras de

memoria; por lo tanto, el número de celdas ocupadas en la Máquina de Turing también será finito.

Para almacenar los contenidos de los registros de la RAM, la Máquina de Turing utilizará tantas cintas como registros haya (el número de registros es finito) . Además, en otras dos cintas se almacenará el valor del Contador de Posición, que contiene el número de la palabra en la que está la próxima instrucción, y el valor del Registro de Direcciones de Memoria, en el cual puede colocarse el número de una palabra de memoria.

¿Cómo podría funcionar? Se asume, por ejemplo, que en cada palabra 10 bits indican una instrucción standard (`LOAD`, `STORE`, `ADD`,...) y los demás son la dirección de un operando.

En un momento dado, el contador de posición contiene el valor *i*, en binario. Habría que buscar, en la primera cinta, la secuencia $\#i*$. Si no la encuentra, no hay instrucción y la Máquina de Turing para (la RAM también pararía, en este caso).

Si la encuentra, se examina la información entre el símbolo * y el siguiente símbolo #. Supóngase que es el equivalente a "`ADD reg 2`" y un número *j* en binario (sumar al registro 2 el contenido de la posición j). Se copia el valor de *j* en la cinta usada como registro de direcciones de memoria. A continuación, se busca la secuencia $\#j*$ en la primera cinta. Si no se encuentra, se asume el valor 0; si se encuentra se suma el valor de v_j a los contenidos de la cinta que se usa como registro 2. Se incrementa el valor de la cinta usada como contador de posición. Y se pasaría a la siguiente instrucción.

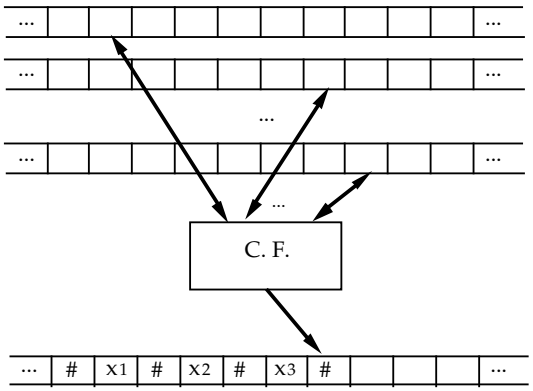
8.3. La Máquina de Turing como Generador.

Las Máquinas de Turing no sólo tienen capacidad para reconocer las cadenas de un lenguaje, sino que también tienen capacidad para generar lenguajes.

La definición formal de la máquina es la misma, con dos salvedades. En un generador el alfabeto Σ debe entenderse como el alfabeto sobre el que se formarán las cadenas del lenguaje. Y, al hablar de las Máquinas de Turing como generadoras, se suele considerar una máquina multicinta con una cinta especial, la *Cinta de Salida* (lo cual es lógico: en una máquina *reconocedora* cobra especial importancia la *entrada*, la cadena a reconocer; en una máquina *generadora*, lo importante es la *salida*, las cadenas que genera).

En la cinta de salida, la máquina irá produciendo todas las cadenas que pertenecen al lenguaje que tiene que generar. Sobre esta cinta sólo se puede escribir. Y, una vez escrito un símbolo, *no se puede reescribir*. El cabezal asociado tiene restringido el movimiento en un sentido, de izquierda a derecha. La imagen más gráfica de esta cinta sería una impresora.

Como ya se ha dicho, las cadenas se formarán a partir del alfabeto Σ, y aparecen en la cinta de salida separadas por un símbolo especial que no pertenezca a dicho alfabeto (normalmente, se utilizará el símbolo #).



Sea M la Máquina de Turing que genera el lenguaje L, L = $\{x_1, x_2, x_3, ...\}$. A M se le denomina *Generador* de L y se utiliza la notación *L = G(M)* para representar a dicho lenguaje.

Cuando se estudia la Máquina de Turing como generador las siguientes afirmaciones son ciertas:

- Si M para, *G(M)* es *finito*. El recíproco no es cierto, en general.
- Las cadenas pueden aparecer repetidas varias veces en la cinta de salida.
- No se asume ningún tipo de orden en la generación de cadenas.
- Una cadena pertenece a G(M) si antes o después aparece en la cinta de salida entre dos símbolos $\#$.

8.3.1. Dos Máquinas de Turing Generadoras Básicas.

Se describen, a continuación, dos generadores básicos para la asignatura y que intervienen en varias construcciones y demostraciones.

El Generador Canónico Se llama Generador Canónico del alfabeto Σ, $G_c(\Sigma^*)$, a la Máquina de Turing capaz de generar todas las cadenas de Σ^*, siguiendo el *Orden Canónico*, que se define de la siguiente forma[1]:

1. Se generan las cadenas por orden creciente de tamaño,
2. Las cadenas del mismo tamaño se generan en orden numérico creciente; para ello, si
$$\Sigma = \{a_0, a_1, a_2, ..., a_{k-1}\},$$
se supone que el símbolo a_i es el *i-ésimo* dígito en base *k*. Por lo tanto, las cadenas de longitud n son los números del 0 al $k^n - 1$.

[1]Este sería el generador necesario para poder realizar la simulación de una Máquina de Turing No Determinista, tal y como se vio en el teorema 7.4 .

Ejemplo:
Si $\Sigma = \{0, 1\}$, entonces Σ^* se irá generando en el siguiente orden: λ, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111, 0000, ... etc.
Nótese que 0, 00, 000, 0000, ... son distintas cadenas (igual que lo serían a, aa, aaa, aaaa, ... si $\Sigma = \{a, b\}$).

En el orden canónico, cada cadena ocupa una posición determinada, pudiéndose caracterizar cada cadena por dicha posición.

El Generador de Pares Se llama Generador de Pares a la Máquina de Turing capaz de generar *todos* los pares $(i, j) \in N \times N$,

$$G(M) = \{(i, j) \mid i, j \in N\}.$$

Si hay que generar todos los pares, la política de generación no puede ser del tipo

```
for i in range(1, ∞):
    for j in range(1, ∞):
        generar (i,j)
```

ya que nunca se llegaría a generar el par (2,1). Para asegurar que todos los pares se generan en tiempo finito, se recurre al orden que proporciona el Triángulo de Tartaglia:

$$\begin{gathered}(1,1)\\(1,2)(2,1)\\(1,3)(2,2)(3,1)\\(1,4)(2,3)(3,2)(4,1)\\(1,5)(2,4)(3,3)(4,2)(5,1)\\\ldots\end{gathered}$$

En este triángulo, los pares *(i,j)* se generan por orden creciente de sumas, y se podría utilizar la secuencia algorítmica,

```
for s in range(2, ∞):
    i = 1
    j =  s - i
    while (i<s):
         generar (i,j)
         i++
         j--
```

De esta forma, se puede asegurar que el número de pares generado antes de generar el par *(i,j)* es

$$(\sum_{k=1}^{i+j-2} k) + i = \frac{(i+j-2)(i+j-1)}{2} + i\,,$$

por lo tanto, finito. Por lo tanto, el número de pasos necesario para calcular cualquier par *(i,j)* también es *finito*.

8.4. Caracterización de L.R. y L.R.E.

Una función computable se identifica con una función recursiva parcial o, lo que es lo mismo, queda descrita por un lenguaje recursivamente enumerable. de ahí, la importancia de poder clasificar a un lenguaje dentro de esta clase. Para demostrar que un lenguaje es recursivamente enumerable, basta con construir una Máquina de Turing capaz de reconoce sus cadenas; si, además, dicha Máquina tiene la parada garantizada, es decir, rechaza las cadenas que no pertenecen al lenguaje, se habrá demostrado que el lenguaje es recursivo.

Existe otra forma de caracterizar el carácter recursivo o recursivamente enumerable de un lenguaje, utilizando generadores. Al fin y al cabo, los lenguajes recursivamente enumerables reciben este nombre por influencia del idioma inglés (en este idioma, un generador se conoce como *enumerator*).

8.4.1. Caracterización de los L.R.E. mediante Generadores.

El objetivo de este apartado es llegar a demostrar que se puede afirmar que todo lenguaje que se pueda generar mediante una Máquina de Turing es un L.R.E.; además, cualquier L.R.E. podrá ser generado por una Máquina de Turing.

Lema 8.1 *Si un lenguaje L es generado por una Máquina de Turing M_1, L = G(M_1), entonces L es un L.R.E. ($\exists\ M_2 \mid L = L(M_2)$).*

Demostración:

Idea básica: si se puede garantizar la construcción de M_2 que reconozca L, entonces L es un L.R.E.

Para demostrar que L es L.R.E., hay que construir una Máquina de Turing M_2 con una cinta más que M_1, que será una *cinta de entrada*. Para reconocer una cadena introducida en la cinta de entrada, el comportamiento de M_2 sería el siguiente: M_2 se construye para simular el comportamiento de M_1; además, llegado el momento en que M_1 imprime el símbolo # sobre la cinta de salida (es decir, después de producir una nueva cadena de L), se compara la cadena generada con la cadena de entrada. Si son iguales, M_2 acepta; si no, continúa la simulación, pasando a generar la siguiente cadena. El comportamiento de M_2 se esquematiza en la figura 8.1.

Esta construcción asegura que M_2 sólo acepta la cadena de entrada cuando es igual a una cadena generada por M_1, es decir, una cadena que pertenece a G(M_1); por lo tanto, se sigue que L(M_2) = G(M_1). Además, el lenguaje reconocido por M_2 es recursivamente enumerable, ya que sólo se puede asegurar que *la máquina acepta la cadena si pertenece al lenguaje*.

c.q.d.

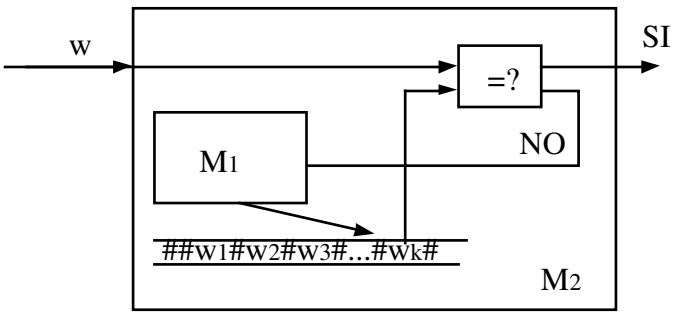


Figura 8.1: Esquema de la construcción de una Máquina de Turing capaz de reconocer un lenguaje, si se conoce el generador de dicho lenguaje.

También es posible demostrar el recíproco, es decir, si L es reconocido por alguna Máquina de Turing, entonces L se puede generar mediante una Máquina de Turing. La idea básica, en este caso, es la siguiente: conocido el alfabeto sobre el que se forman las cadenas de L, se podrían generar todas las cadenas sobre ese alfabeto. Puesto que se dispone de una Máquina de Turing que reconoce, entre todas, las que pertenecen al lenguaje, se puede generar éste completamente.

Para desarrollar esta idea, se utiliza el Generador Canónico, ya que, dado un alfabeto Σ, genera todas las cadenas que pertenecen a Σ^*. Pero hay que tener en cuenta que el lenguaje es recursivamente enumerable: por lo tanto, si w_i no perteneciera a L, es posible que la Máquina de Turing que reconoce L no pare nunca al trabajar sobre dicha cadena. Esto supondría que nunca se llegaría a trabajar con w_{i+1}, w_{i+2}, ... que puede que sí pertenezcan. Esta dificultad se puede solventar gracias al Generador de Pares, tal y como se verá a continuación.

Teorema 8.2 *Un lenguaje L es L.R.E.* ($\exists\ M_2 \mid L = L(M_2)$) $\Leftrightarrow$ *L se puede generar* ($\exists\ M_1 \mid L = G(M_1)$).

Demostración:

"⇐": Lema 8.1

"⇒": Por construcción de la Máquina de Turing M_1 de acuerdo al esquema mostrado en la figura 8.2: cada vez que el Generador de Pares produce un par *(i,j)*, el valor de *i* se introduce como entrada al Generador Canónico para que produzca w_i, la *i-ésima* cadena en orden canónico. Esta cadena se suministra a M_2, junto con el valor de *j*. Si M_2 reconoce la cadena w_i en exactamente *j* pasos, entonces M_1, genera la cadena w_i (la imprime en la cinta de salida).

Si una determinada cadena w pertenece a L, ocupará una determinada posición en orden canónico en el lenguaje Σ^* y será reconocida en un número determinado y

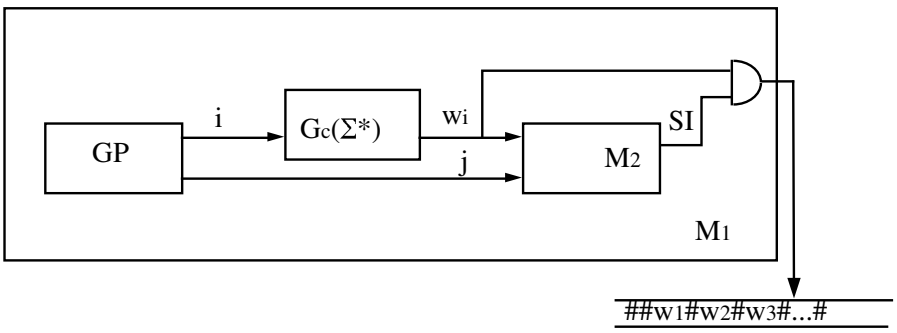


Figura 8.2: Esquema de la Máquina de Turing que permite garantizar que es posible generar cualquier lenguaje recursivamente enumerable.

finito de pasos; es decir, w se corresponderá con algún par *(i,j)*. Como cualquier par es generado en un número finito de pasos por el generador de pares, si $w \in L$ se imprimirá en la cinta de salida (se generará), en un número finito de pasos. Por lo tanto, cualquier cadena del lenguaje puede ser generada por M_1 en tiempo finito.

c.q.d.

Corolario 8.1 *Si L es un L.R.E., entonces hay un generador de L que produce cada cadena exactamente una vez (M_1, por ejemplo).*

8.4.2. Caracterización de los L.R. mediante Generadores.

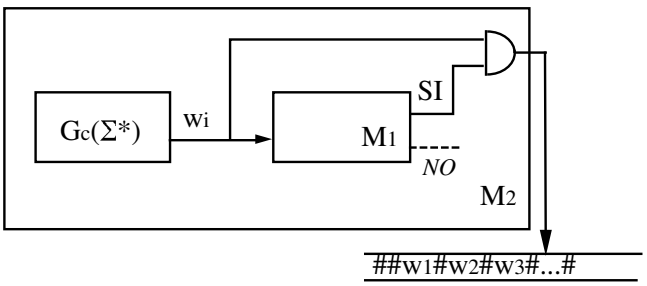
Tal y como se ha visto, cualquier lenguaje que se puede generar mediante una Máquina de Turing es un L.R.E. (y viceversa). A continuación se verá que si, además, se puede generar en orden canónico entonces es un L.R. (y viceversa: cualquier L.R. se puede generar en orden canónico).

Lema 8.2 *Si L es recursivo ($\exists\ M_1 \mid L = L(M_1)\ y\ M_1\ siempre\ para$), entonces existe un generador de L ($\exists\ M_2 \mid L = G(M_2)$) que imprime las cadenas de L en orden canónico.*

Demostración:

Idea básica: Construcción de M_2 a partir de la MT M_1.

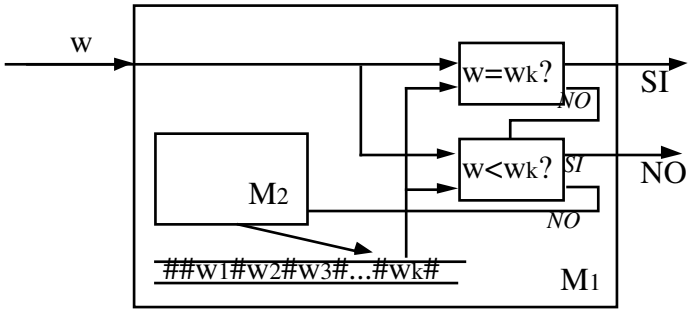
El lenguaje L será un subconjunto de todas las cadenas que se puedan construir a partir de un cierto alfabeto, Σ. Se construye M_2, cuyo comportamiento queda descrito en el siguiente diagrama de bloques:



Se generan las cadenas de Σ^* en orden canónico. Después de generar cada cadena, M_2 simula el comportamiento de M_1 con la cadena generada. Si M_1 la acepta, M_2 la escribe en la cinta de salida. Ya que M_1 siempre para, M_2 podrá generar cada cadena en tiempo finito. Y puesto que las cadenas se producen en orden canónico, aparecerán también en orden canónico en la cinta de salida.

c.q.d.

Para demostrar el recíproco, es decir, "*dado un lenguaje L, tal que L es generado por una Máquina de Turing M_2 en orden canónico, L=G(M_2), entonces L es un L.R.*", se podría seguir la misma pauta que en las demostraciones anteriores: construir una Máquina de Turing M_1, basada en M_2, que acepte las cadenas de L y rechace las cadenas que no pertenezcan a dicho lenguaje. La construcción se podría basar en el esquema mostrado en la figura:



La máquina M_1 simularía el comportamiento de M_2: cada cadena que se genere en la cinta de salida de M_2, se compara con la cadena que hay en la cinta de entrada de M_1. Si ambas son iguales, la cadena es aceptada. Si son distintas, se comprueba que la última cadena generada no sea superior, en orden canónico, a la cadena de la cinta de entrada; en este caso, dicha cadena pude ser rechazada, ya que se tiene la seguridad de que no pertenece al lenguaje puesto que, si así fuera, ya habría aparecido en la cinta de M_2.

Pero esta construcción presenta un problema

- Si L es un lenguaje infinito, está garantizado que M_1 para, tal y como se ha hecho la construcción.

- Pero si L es finito, no se sabe cuál será el comportamiento de M_2 cuando acabe de generar cadenas, puede parar o puede seguir funcionando. En este último caso, la construcción no permite garantizar que M_1 pare siempre.

Pero cualquier lenguaje finito tiene asociado una Máquina de Turing con parada asegurada (un Autómata Finito). Por lo tanto, se garantiza que, en cualquiera de los dos casos, tanto si L es finito como infinito, existe una Máquina de Turing que siempre para (aunque no se pueda dar una "receta" única para su construcción). Según esto, se puede enunciar el siguiente teorema,

Teorema 8.3 *Un lenguaje L es L.R.* ($\exists\ M_1 \mid L = L(M_1)\ y\ M_1\ siempre\ para$) $\Leftrightarrow$ *L se puede generar* ($\exists\ M_2 \mid L = G(M_2)$) *en orden canónico.*

Demostración:

"⇒": Lema 8.2

"⇐": Si L es infinito, entonces L es reconocido por la Máquina de Turing M_1 descrita anteriormente. Si L es finito, entonces hay algún autómata finito que acepta L; por lo tanto, en cualquiera de los dos casos, existe una Máquina de Turing que acepta L y siempre para.

c.q.d.

8.5. Propiedades de Lenguajes Recursivos y Lenguajes Recursivamente Enumerables.

El objetivo de esta sección es estudiar propiedades que pueden ayudar a determinar cuándo un lenguaje es o no es un lenguaje recursivo o recursivamente enumerable. El primer teorema muestra cómo estudiar una propiedad de clausura. Siguiendo el mismo modelo se pueden estudiar la intersección, la diferencia, la clausura transitiva, etc.

Teorema 8.4

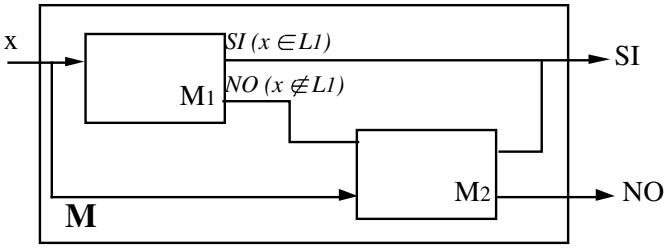
1. *La unión de dos lenguajes recursivos es un lenguaje recursivo.*
2. *La unión de dos lenguajes recursivamente enumerables es un lenguaje recursivamente enumerable.*

Demostración:

1. Sean L_1 y L_2, L.R. $\Rightarrow \exists\ M_1, M_2 \mid L_1 = L(M_1), L_2 = L(M_2)$ y M_1 y M_2 siempre paran:

   - si $x \in L_1$, M_1 acepta la cadena *x* y si $x \notin L_1$, M_1 rechaza la cadena *x*.
   - si $x \in L_2$, M_2 acepta la cadena *x* y si $x \notin L_2$, M_2 rechaza la cadena *x*.

   Sea $L = L_1 \cup L_2$, ¿$\exists\ M \mid L = L(M)$ y M siempre para? Sí, construyendo la máquina que se presenta en la figura:



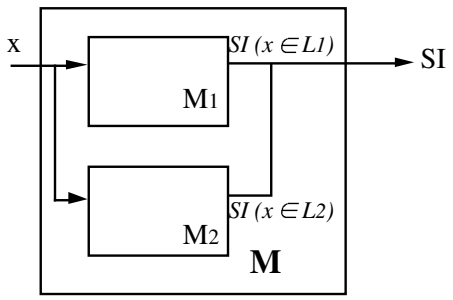
   La construcción es correcta puesto que

   - si $x \in L_1\ OR\ x \in L_2 \Rightarrow x \in L_1 \cup L_2 = L$ (M_1 ó M_2 aceptan $\Rightarrow$ M acepta)
   - si $x \notin L_1\ AND\ x \notin L_2 \Rightarrow x \notin L_1 \cup L_2 = L$ (M_1 y M_2 rechazan $\Rightarrow$ M rechaza).

   Por lo tanto, M siempre para y si $x \in L$, M acepta y si $x \notin L$, M rechaza.

2. Sean L_1 y L_2, L.R.E. $\Rightarrow \exists\ M_1, M_2 \mid L_1 = L(M_1), L_2 = L(M_2)$, es decir, si $x \in L_1$, M_1 acepta y si $x \in L_2$, M_2 acepta.

   Sea $L = L_1 \cup L_2$, ¿$\exists\ M \mid L = L(M)$? Sí, construyendo M de la siguiente forma:



La construcción es correcta puesto que si $x \in L_1$ OR $x \in L_2 \Rightarrow x \in L_1 \cup L_2 = L$ (M_1 ó M_2 aceptan $\Rightarrow$ M acepta) por lo tanto, si $x \in L$, M acepta. El comportamiento de M queda indeterminado si $x \notin L_1$ AND $x \notin L_2 \Rightarrow x \notin L_1 \cup L_2 = L$, pero se intenta caracterizar un lenguaje recursivamente enumerable.

c.q.d.

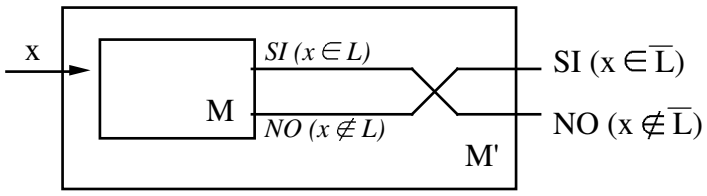
Los dos siguientes teoremas se refieren a la operación complemento:

Teorema 8.5 *El complemento de un lenguaje recursivo es un lenguaje recursivo.*

Demostración:

Sea L un L.R. $\Rightarrow \exists$ M | L = L(M) y M siempre para: si $x \in L$, M acepta la cadena *x* y si $x \notin L$, M rechaza la cadena *x*.

Sea $\bar{L}$ = Σ^* - L, ¿$\exists$ M′ | $\bar{L}$ = L(M′) y M′ siempre para? Sí, construyendo la máquina M′ de la siguiente forma:



La construcción es correcta puesto que

- si $x \in L \Rightarrow x \notin \bar{L}$ (M acepta $\Rightarrow$ M′ rechaza)
- si $x \notin L \Rightarrow x \in \bar{L}$ (M rechaza $\Rightarrow$ M′ acepta)

por lo tanto, M′ siempre para y si $x \in \bar{L}$, M′ acepta y si $x \notin \bar{L}$, M′ rechaza. Y dado que es posible construir M′, $\bar{L}$ es un L.R.

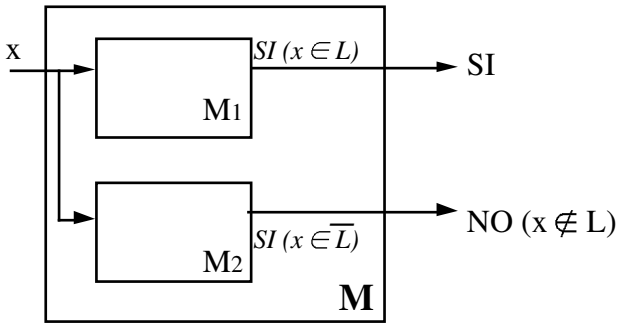
c.q.d.

Teorema 8.6 *Si un lenguaje L y su complementario $\bar{L}$ son L.R.E., entonces L y $\bar{L}$ son lenguajes recursivos.*

Demostración:

Sea L un L.R.E. $\Rightarrow \exists\ M_1 \mid$ L=L(M), es decir, si $x \in L$, M_1 acepta. Sea $\bar{L} = \Sigma^*$ - L, un L.R.E. $\Rightarrow \exists\ M_2 \mid \bar{L} = L(M)$, es decir, si $x \in \bar{L}$, M_2 acepta.

¿$\exists$ M | L = L(M) y M siempre para? Sí, construyendo la máquina M de la siguiente forma:



La construcción es correcta puesto que

- $x \in L \Rightarrow M_1$ acepta $\Rightarrow$ M acepta,
- $x \notin L \Rightarrow x \in \bar{L} \Rightarrow M_2$ acepta $\Rightarrow$ M rechaza.

por lo tanto, M siempre para y si $x \in L$, M acepta y si $x \notin L$, M rechaza. Por lo tanto, L es un lenguaje recursivo; y, por el teorema 8.5, $\bar{L}$ también es un lenguaje recursivo.

c.q.d.

Además, de los teoremas 8.5 y 8.6 se desprenden las siguientes relaciones: Dado el lenguaje L y su complementario $\bar{L}$, sólo se puede cumplir una de estas cuatro afirmaciones,

1. L y $\bar{L}$ son lenguajes recursivos,
2. L es recursivamente enumerable (no recursivo) y $\bar{L}$ NO es recursivamente enumerable,
3. L NO es recursivamente enumerable y $\bar{L}$ es recursivamente enumerable (no recursivo),
4. L y $\bar{L}$ NO son lenguajes recursivamente enumerables.

Capítulo 9

Indecidibilidad

Índice General

9.1. Concepto de Problema. 147
9.1.1. Introducción. 147
9.1.2. Concepto de Problema. 150
9.2. La Máquina Universal de Turing y Dos Problemas Indecidibles. . . 152
9.2.1. Codificación de Máquinas de Turing. 152
9.2.2. Ejemplo de un Lenguaje que NO es Recursivamente Enumerable. 153
9.2.3. La Máquina Universal de Turing. 155
9.2.4. Dos Problemas Indecidibles. 157
9.3. Teorema de Rice. Más Problemas Indecidibles. 162
9.3.1. El Problema de la Vaciedad. 162
9.3.2. El Teorema de Rice. 165
9.4. La Indecidibilidad del Problema de la Correspondencia de Post. . . 168
9.5. Apéndice 1: Otro Ejemplo de Reducción. El Problema de la Recursividad. 173
9.6. Apéndice 2: Ejemplo de cómo aplicar Rice fuera de su ámbito. . . . 176

9.1. Concepto de Problema.

9.1.1. Introducción.

Como se ha visto en los capítulos anteriores, una Máquina de Turing se puede estudiar como un calculador de funciones o como un reconocedor de lenguajes. Así, la clase de los lenguajes recursivos se puede identificar con la clase de las funciones recursivas totales, mientras que la clase de los lenguajes recursivamente enumerables se puede identificar con la clase de las funciones recursivas parciales.

Cualquier Máquina de Turing reconocedora de cadenas es una Máquina de Turing calculadora de la *función característica* del lenguaje que reconoce: es una función que asocia el valor 1 a las cadenas que pertenecen al lenguaje y 0 a las cadenas que no pertenecen al lenguaje. Cualquier Máquina de Turing calculadora de funciones es una Máquina de Turing reconocedora de lenguajes, ya que se puede representar cada función por el lenguaje formado por las tuplas que se pueden formar con sus parámetros de entrada y de salida.

Ejemplo:

La función suma *se podría representar como el conjunto*

{(0, 0, 0), (0, 1, 1), (0, 2, 2), .., (1, 0, 1), (1, 1, 2), .., (1, 9, 10), .., (325, 16, 341), ..}.

Este conjunto de cadenas, este lenguaje, estará formado por tripletas de números tales que el tercero representa al suma de los dos primeros.

Lo anterior pretende solamente incidir en lo que se viene repitiendo desde el primer tema: las dos visiones son completamente equivalentes, y una función total puede ser representada mediante un lenguaje recursivo y una función parcial mediante un lenguaje recursivamente enumerable.

Además, la hipótesis de Church permite identificar las funciones computables con las funciones recursivas parciales. Es decir, se pueden diseñar Máquinas de Turing para calcular las funciones recursivas parciales. O, desde el punto de vista del reconocimiento de lenguajes, se pueden diseñar Máquinas de Turing que aceptan una cadena si forma parte de un lenguaje recursivamente enumerable (de ahí que algunos autores denominen indistintamente a las funciones computables, como funciones *Turing-computables* o funciones *Turing-aceptables*).

Si una función es computable, se puede calcular su solución *cuando exista*. Y esa solución se puede calcular mediante una Máquina de Turing. Cuestión aparte es garantizar a priori qué ocurrirá con el proceso para cualquier parámetro. Está garantizado que será finito cuando la función es total, ya que se conoce su dominio de definición (para estos valores finalizará con éxito, para los demás finalizará con un error). Pero en una función parcial, el dominio de definición no está determinado. Por lo tanto, no siempre se podrá garantizar la finitud y corrección del proceso, sólo para ciertos valores de los parámetros. Como un ejemplo, se considera el siguiente algoritmo:

```
boole maravilloso(int n){
/* pre: n=N, entero positivo */

   boole loEs;

   if (n==1)
      loEs=cierto;
   else
```

```
      if (n%2==0)
         loEs=maravilloso(n/2);
      else
         loEs=maravilloso(3*n+1);

   return loEs;
}
/* post: cierto, si n es maravilloso, si no lo es... :-( */
```

Este algoritmo se ha construido siguiendo la definición de Número Maravilloso: *"Un número n es maravilloso si es el 1, o puede alcanzarse el 1 a través del siguiente proceso: si es par, se considera el valor de n/2; si es impar, el valor de 3*n+1"*. Se *conjetura* que todos los números enteros son maravillosos, pero no ha podido demostrarse. De ahí que el Dominio de Definición de esta algoritmo no esté determinado; y, por lo tanto, se esté hablando de una función parcial, en la que no es posible garantizar que el proceso finalice para *cualquier* valor entero que se considere.

Si una función es total, existe al menos una Máquina de Turing que siempre se detiene, bien dando el resultado del cálculo, bien indicando la existencia de un error. Ese error sólo se dará si *los parámetros no pertenecen al dominio de definición* de la función. La situación equivalente, desde el punto de vista del reconocimiento de un lenguaje, es *la cadena no pertenece al lenguaje*.

La MT calcula funciones:	La MT reconoce lenguajes:
Si la función es total	Si el lenguaje es recursivo
la MT siempre para.	la MT siempre para.
Devuelve el resultado si los parámetros $\in$ Dominio Definición.	Acepta la cadena si la cadena $\in$ L(M).
Devuelve error si los parámetros $\notin$ Dominio Definición.	Rechaza la cadena si la cadena $\notin$ L(M).

Si una función es parcial existirá una Máquina de Turing de la que sólo se asegura que parará devolviendo el resultado del cálculo cuando *los parámetros pertenecen al dominio de definición* de la función. No se puede garantizar qué ocurre en caso contrario. De nuevo, se establece un paralelismo evidente entre este comportamiento y lo que ocurre desde el punto de vista de reconocimiento del lenguaje,

La MT calcula funciones:	La MT reconoce lenguajes:
Si la función es parcial	Si el lenguaje es rec. enumerable
la MT siempre para devolviendo el resultado si los parámetros $\in$ Dominio Definición.	la MT siempre para aceptando la cadena si la cadena $\in$ L(M).
Pero el comportamiento está indefinido si los parámetros $\notin$ Dominio Definición.	Pero el comportamiento está indefinido si la cadena $\notin$ L(M).

Por lo tanto, ¿cuál es el interés en determinar si un lenguaje es o no es recursivo?. En este tema, mediante el *concepto de problema* y la definición de la *Máquina Universal de Turing*, el objetivo será estudiar los mecanismos que permiten establecer cuáles son los problemas en los que se puede garantizar la existencia de una computación finita por medio de un computador. En este tema se demostrará que existen problemas para los cuales esa computación finita no existe. Por lo tanto, se demostrará que la computación tiene límites.

9.1.2. Concepto de Problema.

Definición 9.1 (Problema) *Un Problema es un enunciado cierto o falso dependiendo de los valores de los parámetros que aparecen en su definición.*

Definición 9.2 (Solución) *Una Solución a un Problema es una aplicación entre el conjunto de instancias de los parámetros del problema y el conjunto* $\{cierto, falso\}$.

Definición 9.3 (Algoritmo) *Un Algoritmo es un conjunto de pasos cuyo objetivo es resolver un problema.*

Es posible identificar un algoritmo con una función,

$$f : A_1 \times A_2 \times \ldots \times A_n \longrightarrow A,$$

de forma que un algoritmo obtiene un valor de salida a partir de unos valores de entrada si ese valor de salida existe, es decir, si hay solución al problema. Mientras que la solución a un problema se asimilaría al establecimiento de una aplicación

$$P(f) : A_1 \times A_2 \times \ldots \times A_n \times A \longrightarrow \{cierto, falso\}.$$

Sólo si se puede establecer esa aplicación entre parámetros y el conjunto {cierto, falso} hay una solución del problema; y sólo si esa aplicación es una *función total* existe la seguridad de establecer el algoritmo: para cualquier instancia se puede *decidir* si el enunciado es cierto o falso. Y esa decisión se puede obtener aplicando el algoritmo, de forma que se cumple la relación,

$$\mathbf{P(f)(a_1, a_2, \ldots, a_n, a) = cierto \Leftrightarrow f(a_1, a_2, \ldots, a_n) = a}.$$

Por ejemplo,

$$f : \mathcal{N} \times \mathcal{N} \rightarrow \mathcal{N},\ f(x,y) = x + y$$
P(f)(3,5,7)=falso, P(f)(3,5,8)=cierto, ...

En este problema siempre es posible decidir, por lo tanto, se sabe que siempre será posible establecer la solución y se puede estar seguro de que habrá un algoritmo que permita solucionar el problema.

En aquellos problemas para los que exista algún valor en el conjunto de instancias para el que la aplicación no esté definida, NO se puede DECIDIR siempre: no se puede asegurar cuál será el comportamiento ante dichas instancias. Son problemas en los que la aplicación entre el conjunto de parámetros y el conjunto $\{$*cierto, falso*$\}$ es una función parcial. Por lo tanto, se establece la relación,

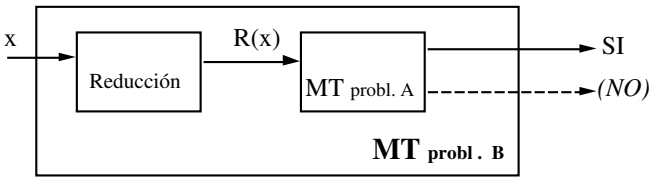
PROBLEMA DECIDIBLE $\equiv$ LENGUAJE RECURSIVO.

Así, estudiando si el lenguaje asociado a un determinado problema es o no es recursivo, se puede saber si el problema es o no es decidible y si se puede o no garantizar la existencia de un algoritmo de ejecución finita. Es decir, se asocia la existencia de un *algoritmo* a la existencia de una *Máquina de Turing que siempre pare*, diciendo SI o NO. Y se usarán indistintamente ambos términos:

Existe un algoritmo $\equiv$ Existe una MT que siempre para.

Resulta una práctica habitual que los problemas se enuncien como *problemas de decisión*, de respuesta SI o NO y en los que, por lo tanto, resulta fácil transformar el enunciado en la cuestión de si un lenguaje (el asociado al enunciado) es o no recursivo y, por lo tanto, si existe o no un algoritmo que permita resolverlos. Centrarse en este tipo de problemas no supone restringir el campo de estudio, ya que normalmente cualquier problema se puede plantear como un problema de decisión y viceversa.

El objetivo de este tema será, por lo tanto, establecer resultados que permitan afirmar si un problema es o no *decidible*. O, lo que es lo mismo, si el lenguaje asociado a dicho problema es o no *recursivo*. Para ello, una técnica habitual será la *reducción* de un problema A (o de su lenguaje equivalente) a otro problema B,



de tal forma que el conocimiento que se tenga del problema A permita llegar a alguna conclusión sobre el problema B.

9.2. La Máquina Universal de Turing y Dos Problemas Indecidibles.

9.2.1. Codificación de Máquinas de Turing.

Para estudiar la codificación de una Máquina de Turing, se recuerda el siguiente resultado, que ya se comentó en el tema 7:

Teorema 7.5 Si L $\in (0+1)^*$ es aceptado por alguna Máquina de Turing $\Rightarrow$ L es aceptado por una Máquina de Turing con alfabeto restringido a $\{0, 1, B\}$.

Este teorema sostiene que cualquier lenguaje sobre el alfabeto $\{0,1\}$ se puede reconocer en una Máquina de Turing cuyo alfabeto de cinta sea $\{0,1,\mathrm{B}\}$; por lo tanto, sólo son necesarios tres símbolos en dicha Máquina de Turing.

Por otro lado, también es posible afirmar que no hay necesidad de más de un estado final en cualquier Máquina de Turing.

Con estas premisas, se propone la siguiente codificación para cualquier Máquina de Turing con alfabeto restringido: Sea

$$M = \langle \{0,1\},\ Q,\ \{0,1,\mathtt{B}\},\ f,\ q_1,\ \mathtt{B},\ \{q_2\} \rangle$$

una Máquina de Turing con alfabeto de entrada $\{0,1\}$ y el B como único símbolo adicional en el alfabeto de la cinta. Se asume que Q = $\{q_1, q_2, q_3, .., q_n\}$ es el conjunto de estados y que sólo hay un estado final q_2 (y, por supuesto, q_1 es el estado inicial).

Se renombra el alfabeto:

Símbolo					Codificación
0	$\rightarrow$	x_1	(Primer símbolo)	$\rightarrow$	$0^1 = 0$
1	$\rightarrow$	x_2	(Segundo símbolo)	$\rightarrow$	$0^2 = 00$
B	$\rightarrow$	x_3	(Tercer símbolo)	$\rightarrow$	$0^3 = 000$

Se renombran las direcciones:

Dirección					Codificación
L	$\rightarrow$	D_1	(Primera dirección)	$\rightarrow$	$0^1 = 0$
R	$\rightarrow$	D_2	(Segunda dirección)	$\rightarrow$	$0^2 = 00$

Y, si cada identificador de estado q_i se representa como 0^i, entonces se puede codificar cada una de las transiciones:

$$f(q_i, x_j) = (q_k, x_l, D_m) \ \hookrightarrow\ 0^i 10^j 10^k 10^l 10^m$$

A cada una de estas transiciones así codificadas se le denominará *código* y se le asigna un orden, por lo que se puede codificar una Máquina de Turing M como

$$111código_1 11código_2 11código_3 11 \ldots 11código_r 111$$

Los códigos siempre estarán entre parejas de 11. El orden de los códigos es irrelevante; una misma Máquina de Turing puede tener distintas codificaciones[1]. Pero cada codificación sólo puede estar asociada a una Máquina de Turing.

Por ejemplo: Sea la Máquina de Turing

$$M = \langle \{0,1\},\ \{q_1, q_2, q_3\},\ \{0,1,B\},\ f,\ q_1,\ \mathtt{B},\ \{q_2\} \rangle$$

con

$$\begin{aligned} \mathtt{f}(\mathtt{q}_1, \mathtt{1}) &= (\mathtt{q}_3, \mathtt{0}, \mathtt{R}) \\ \mathtt{f}(\mathtt{q}_3, \mathtt{0}) &= (\mathtt{q}_1, \mathtt{1}, \mathtt{R}) \\ \mathtt{f}(\mathtt{q}_3, \mathtt{1}) &= (\mathtt{q}_2, \mathtt{0}, \mathtt{R}) \\ \mathtt{f}(\mathtt{q}_3, \mathtt{B}) &= (\mathtt{q}_3, \mathtt{1}, \mathtt{L}) \end{aligned}$$

Una posible codificación de M es la cadena:

```
11101001000101001100010101001001100010010010100110001000
10010111
```

Con esta codificación, *una Máquina de Turing es un número en binario*. Por lo tanto, *cualquier número en binario* podrá ser considerado, inicialmente, un *código de Máquina de Turing*. Evidentemente, habrá cadenas binarias que representen Máquina de Turing y cadenas binarias que no representen ninguna Máquina de Turing: las cadenas que no comiencen por 111, o no acaben en 111 o que no tengan parejas de 1's separando 5 bloques de 0's que, a su vez, están separados por 1, no codifican Máquinas de Turing.

Se denota $\langle$M,$w\rangle$ a la cadena formada al concatenar la codificación de M con la cadena w. Se interpreta que la cadena w sería la cadena de entrada a la Máquina de Turing M.

En el ejemplo anterior, la cadena $\langle M,$ `1011`$\rangle$ *se codifica como:*

```
11101001000101001100010101001001100010010010100110001000
10001001011110 11
```

El establecimiento de esta codificación es básico para la definición de la *Máquina Universal de Turing*, como aquella Máquina de Turing cuyas cadenas de entrada son de la forma $\langle$M,$w\rangle$ y cuyo comportamiento consiste en la simulación del comportamiento de la máquina M cuando su entrada es la cadena w.

9.2.2. Ejemplo de un Lenguaje que NO es Recursivamente Enumerable.

A continuación se presenta un lenguaje que no es recursivamente enumerable; su interés consiste en poder utilizarlo como una herramienta para caracterizar el carácter recursivo o

[1] Ya que una misma MT puede tener distintas codificaciones según el orden en el que se representen los códigos, cuando se utilice la notación $\langle$M$\rangle$ para representar un código de MT, lo que realmente se representará es el conjunto de cadenas que codifican a la MT M.

recursivamente enumerable de otros lenguajes, utilizando las relaciones vistas en la sección 8.5.

Es un lenguaje construido con el propósito de disponer de un lenguaje no recursivamente enumerable.

Se construye mediante una tabla. La numeración de las filas sirve para representar las cadenas de $(0+1)^*$ en orden canónico: la primera fila, representa la primera cadena; la segunda fila, a la segunda cadena en orden canónico, etc. Las columnas se numeran sucesivamente a partir del 1 y cada índice de columna se interpreta en binario, de forma que dichos números en binario se interpretan como codificaciones de Máquina de Turing (*j* en binario es la Máquina de Turing M_j). Se construye la siguiente tabla infinita[2],

	1	2	3	4	...
1	**0**	1	1	0	...
2	0	**0**	1	1	...
3	1	1	**1**	0	...
4	1	0	0	**1**	...
...	...	...	...	...	...

en la que cada entrada es 1 ó 0 de acuerdo al siguiente convenio: si $w_i \in L(M_j)$, entonces el elemento *(i,j)* es 1, si no es 0,

$$(i,j) = 1 \Leftrightarrow w_i \in L(M_j),$$
$$(i,j) = 0 \Leftrightarrow w_i \notin L(M_j).$$

Se construye el lenguaje $\mathcal{L}_d$, el *lenguaje diagonal*, formado por los elementos diagonales nulos; es decir, en $\mathcal{L}_d$ está la cadena w_i si *(i,i)*=0; por lo tanto, M_i NO acepta la cadena w_i:

$$\mathcal{L}_d = \{w_i \mid M_i \text{ NO acepta } w_i\} = \{w_i \mid w_i \notin L(M_i)\}$$

Lema 9.1 *El lenguaje $\mathcal{L}_d$ NO es recursivamente enumerable.*

Demostración:

Supóngase, como hipótesis de partida, que existe alguna Máquina de Turing, M_j que acepte el lenguaje $\mathcal{L}_d$, es decir, $\mathcal{L}_d = L(M_j)$. Sea w_j la cadena que ocupa la *j-ésima* posición en orden canónico y el valor *j* codificado en binario, el código de M_j.

Si $\mathcal{L}_d = L(M_j)$ se debe cumplir que si $w_j \in \mathcal{L}_d$, entonces $w_j \in L(M_j)$. Y esto es imposible:

$$w_j \in \mathcal{L}_d \Rightarrow (j,j) = 0 \Rightarrow w_j \notin L(M_j),$$
$$w_j \notin \mathcal{L}_d \Rightarrow (j,j) = 1 \Rightarrow w_j \in L(M_j).$$

[2]No todas las codificaciones de las columnas tendrán sentido como códigos de MT; de hecho, el ejemplo que muestra cómo se llenaría es ficticio, puesto que todas esas entradas deberían ser ceros, ya que tanto 1, como 10, 11 ó 100 no son códigos válidos de MT.

Por lo tanto, $\mathcal{L}_d \neq L(M_j)$. La hipótesis es falsa y no existe ninguna Máquina de Turing que acepte $\mathcal{L}_d$.

c.q.d

9.2.3. La Máquina Universal de Turing.

Definición 9.4 (Lenguaje Universal, $\mathcal{L}_U$) *Se denomina Lenguaje Universal al conjunto de cadenas*

$$\mathcal{L}_U = \{\langle M,w\rangle \mid M \text{ acepta } w\}\ .$$

En esta definición se asume que M admite la codificación anteriormente descrita, dado que si su alfabeto no está restringido al alfabeto $\{0,1,\mathtt{B}\}$ siempre se podrá encontrar la máquina equivalente con alfabeto restringido.

Teorema 9.1 *$\mathcal{L}_U$ es un lenguaje recursivamente enumerable.*

Demostración:

La demostración se realiza mediante la construcción de una Máquina de Turing, M1, que acepta $\mathcal{L}_U$. Puesto que para determinar si la cadena $\langle$M,$w\rangle \in \mathcal{L}_U$ hay que determinar si la máquina M acepta la cadena w, se construirá para permitir la simulación del comportamiento de M con w.

Dicha máquina tendrá tres cintas. La primera cinta es de *entrada*; en ella irá la cadena $\langle$M,$w\rangle$ y, puesto que en ella estará la codificación de M, su papel en la simulación de su comportamiento con la cadena w sería similar al de la memoria de un ordenador. Todas las transiciones (todos los códigos, entre pares de 11) están en el primer bloque (entre los dos 111).

La segunda cinta de M1, simulará a la cinta de entrada de M y, por lo tanto, es la que se realiza realmente la simulación del comportamiento de M. En ella se copiará la cadena w.

La tercera cinta de M1, servirá para llevar cuenta del estado en que estaría M; para ello, q_i se codificará como 0^i. Su papel en la simulación es similar al del contador de programa de un ordenador.

El comportamiento de M1 es el siguiente:

1. Se comprueba que la cadena que está en la cinta de entrada es correcta, es decir, que el formato es adecuado (bloques 111 y bloques 11) y que no hay dos códigos distintos

que comiencen con 0^i10^j para un mismo par *i, j*. También se comprueba que en cada código de la forma $0^i10^j10^k10^l10^m$, $1 \leq j \leq 3$, $1 \leq l \leq 3$ y $1 \leq m \leq 2$ (para ello se pueden usar las otras cintas como auxiliares si hiciera falta).

2. Se inicializa la cinta 2 con la cadena w. Se inicializa la cinta 3 con el valor 0 que codifica a q_1. Los 3 cabezales de lectura/escritura se colocan en el símbolo situado más a la izquierda en las 3 cintas.

3. **Repetir el siguiente proceso**,

   Sea x_j el símbolo que se está leyendo en la cinta 2 y sea 0^i el estado representado en la cinta 3. Se recorre la cinta 1 de izquierda a derecha (parando en el segundo 111) buscando una subcadena (entre 11's) que comience por $0^i10^j\ldots$; si no se encuentra M1 para y rechaza ya que M no tiene transición asociada. Si el código se encuentra, por ejemplo $0^i10^j10^k10^l10^m$, se escribe 0^k en la cinta 3, se escribe x_l en la celda bajo el cabezal en la cinta 2 y en esa cinta el cabezal se desplaza en la dirección D^m.

   hasta que en la cinta 3 se llega al código 00 **o** M1 para y rechaza.

Si en la cinta 3 se llega al código 00, quiere decir que M trabajando con la cadena w llega a q_2, estado final; en este caso la máquina M1 para y acepta la cadena $\langle$M,$w\rangle$. Por lo tanto, M1 acepta $\langle$M,$w\rangle$ si y sólo si M acepta la cadena w. También se cumple que si M para sin aceptar w, entonces M1 para sin aceptar $\langle$M,$w\rangle$ y que si M no para con w, entonces M1 tampoco para con $\langle$M,$w\rangle$. De la existencia de M1 se sigue que $\mathcal{L}_U$ es un Lenguaje Recursivamente Enumerable.

c.q.d

Aplicando resultados anteriores (ver capítulo 7), se podría construir una máquina equivalente a M1, que tuviera una sola cinta, limitada a la izquierda y el alfabeto restringido a $\{0, 1, \texttt{B}\}$. Esta máquina es la definición formal de la Máquina Universal de Turing,

Definición 9.5 (Máquina Universal de Turing, $\mathcal{M}_U$) *Máquina de Turing con una sola cinta, limitada a la izquierda, y con alfabeto $\{0, 1, \texttt{B}\}$ que acepta el lenguaje $\mathcal{L}_U$.*

La culminación del modelo de computación establecido por Alan Turing con esta Máquina Universal ha tenido, y sigue teniendo, mucha más trascendencia del resultado puramente lógico.

Este resultado establece que es posible *enumerar* todas las cadenas que representan computaciones válidas. Desde el punto de vista lógico permitió asegurar que tiene que haber funciones *no computables*, tal y como se estableció en la introducción del tema 8.

Pero, además, cuando Von Neumann estableció la arquitectura que hoy se conoce con su nombre, lo hizo influenciado por el resultado de Turing: hasta que Von Neumann introdujo el concepto de *control por programa almacenado* los computadores se "programaban" (si así se puede denominar) reconfigurando completamente el *hardware* para cada cálculo a realizar. Este modelo inspiró a Von Neumann la idea de que sería más factible disponer de una circuitería simple (un autómata) gobernado por un conjunto de instrucciones, más o menos complejas, y fue, en definitiva, el punto de arranque del desarrollo de la programación.

Se ha comentado que la Máquina de Turing se estudia como modelo formal de algoritmo; se puede establecer la analogía siguiente:

> *Una máquina de Turing procesando una cadena de entrada, es un modelo de un algoritmo procesando sus datos de entrada.*

El concepto de Máquina Universal de Turing permite establecer una analogía que resulta más familiar todavía,

> *Una Máquina Universal de Turing recibe como cadena de entrada el código de una Máquina de Turing y la cadena con la que esta trabajaría; un computador de propósito general recibe como cadena de entrada el código de un programa y los datos con los que el programa trabaja.*

9.2.4. Dos Problemas Indecidibles.

La existencia del lenguaje $\mathcal{L}_d$ permite establecer que existe, al menos, un lenguaje que no pertenece a la clase de los lenguajes recursivamente enumerables.

Es, además, un lenguaje que permitirá establecer resultados sobre la indecidibilidad. No existe una herramienta como el Lema de Bombeo para lenguajes recursivos y recursivamente enumerables. Por lo tanto, para poder estudiar que lenguajes pertenecen o no a dichas clases, sólo se dispone, como herramientas de trabajo, de las propiedades de clausura y de la reducción de problemas.

Si se puede diseñar una Máquina de Turing que acepte las cadenas de un lenguaje, se demuestra que éste es computable. Si, además, la construcción garantiza la parada, el lenguaje es decidible. Y, entonces, existe un algoritmo para resolver el problema asociado.

Ya se ha establecido que $\mathcal{L}_U$ es un lenguaje computable; si, además, fuera decidible, existiría un algoritmo de algoritmos, la posibilidad de determinar automáticamente el éxito o el fracaso de una computación. Algo parecido al teorema de los teoremas que buscaba Hilbert.

Para estudiar si $\mathcal{L}_U$ es o no un lenguaje recursivo, se utiliza como herramienta el lenguaje $\mathcal{L}_d$. En el lema 9.1, quedó establecido que $\mathcal{L}_d$ es un lenguaje no recursivamente enumerable. Por lo tanto, $\mathcal{L}_d$ tampoco es un lenguaje recursivo y, por el teorema 8.5, su complementario, $\bar{\mathcal{L}}_d$,

$$\bar{\mathcal{L}}_d = \{w_i \mid M_i \text{ acepta } w_i\} = \{w_i \mid w_i \in L(M_i)\},$$

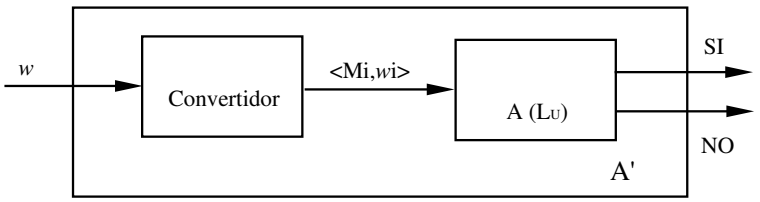
tampoco puede ser un lenguaje recursivo. Este resultado permitirá demostrar que $\mathcal{L}_U$ es un lenguaje recursivamente enumerable no recursivo, puesto que es posible reducir el lenguaje $\bar{\mathcal{L}}_d$ al lenguaje $\mathcal{L}_U$.

Teorema 9.2 *$\mathcal{L}_U$ es un lenguaje recursivamente enumerable no recursivo.*

Demostración:

Se parte de la suposición de que existe un algoritmo A (una Máquina de Turing que siempre para) para reconocer $\mathcal{L}_U$. Si esto fuera cierto, entonces se podría establecer el siguiente procedimiento para reconocer $\bar{\mathcal{L}}_d$:

Se realiza una reducción utilizando el *Convertidor de Cadenas*. El convertidor, dada una cadena $w \in (0+1)^*$, determina el valor de *i* tal que $w = w_i$, la *i-ésima* cadena en orden canónico. Este valor *i*, expresado en binario, es el código de alguna Máquina de Turing, M_i. La salida del convertidor es la cadena $\langle \mathrm{M}_i, w_i \rangle$ que se suministra al algoritmo A. Así se consigue una Máquina de Turing que acepta w si y sólo si M_i acepta w_i.



Es decir, se ha construido un algoritmo A' que indica si la cadena w pertenece o no al lenguaje $\bar{\mathcal{L}}_d$. Puesto que esto es imposible, ya que $\bar{\mathcal{L}}_d$ no es recursivo, entonces la suposición de que existe el algoritmo A debe ser falsa.

Por lo tanto, $\mathcal{L}_U$ es un lenguaje recursivamente enumerable no recursivo.

c.q.d

Es decir, el problema

"Dada una Máquina de Turing M y una cadena w, ¿acepta M la cadena w?"

es un problema *indecidible*.

El Problema de la Parada.

Un problema tan importante como el anterior y también indecidible es el *Problema de la Parada*:

"Sea M una Máquina de Turing y una cadena w. ¿Para M con entrada w?".

Es mucha la importancia de este problema: si se pudiera predecir la parada de M, si se pudiera predecir *de forma automática* la parada de cualquier proceso en ejecución, también se podría decidir automáticamente el éxito o el fracaso de esa ejecución.

Supóngase que fuera posible predecir la parada de un proceso; es decir, que se dispone de una función llamada `Halts`, tal que

```
int Halts(char *P, char *I) {
/* Pre: P e I son cadenas de caracteres, siendo P */
/* el código fuente de un programa e I los datos */

   /* (1) se determina si P es un programa correcto */
                    /* sintácticamente (compilación)*/
   /* (2) se determina si P finaliza su ejecución */
               /* cuando lee la cadena de entrada I */

   return halt;
}
/* Post: devuelve 1 si P para con I, 0 en caso contrario */
```

Sabiendo esto, se escribe el programa siguiente:

```
int main() {
   char I[100000000];
   /* hacer I tan grande como se quiera, o usar malloc */

   read_a_C_program_into(I);

   if (Halts(I,I)) {
       while(1){}    /* bucle infinito */
   }
   else
       return 1;
}
```

Este programa se almacena en el fichero `Diagonal.c`. Una vez compilado y montado, se obtiene el código ejecutable, `Diagonal`. A continuación se ejecuta,

```
Diagonal<Diagonal.c
```

Sólo hay dos casos posibles al ejecutar esta orden, y son mutuamente excluyentes:

Caso 1: `Halts(I,I)` devuelve 1.

Según la definición de `Halts`, esto significa que `Diagonal.c` finaliza su ejecución cuando recibe como entrada `Diagonal.c`. Pero, según la definición de `Diagonal.c`, que `Halts(I,I)` devuelva `1`, significa que en el condicional se ejecutará la rama "if", que contiene el bucle infinito; es decir, la ejecución de `Diagonal.c` no finaliza NUNCA.

Se llega a una contradicción.

Caso 2: `Halts(I,I)` devuelve 0.

Según la definición de `Halts`, esto significa que `Diagonal.c` NUNCA finaliza su ejecución cuando recibe como entrada `Diagonal.c`. PERO, según la definición de `Diagonal.c`, que `Halts(I,I)` devuelva `0`, significa que en el condicional se ejecutará la rama "else", por lo que finaliza la ejecución de `Diagonal.c`.

Se llega, de nuevo, a una contradicción.

Puesto que no hay más casos posibles la suposición inicial debe ser falsa: no es posible que exista la función `Halts`.

La demostración formal de que el problema de la parada es un problema indecidible, se basa en que el lenguaje asociado,

$$\mathcal{L}_H = \{\langle \text{M}, w\rangle \mid \text{M para con } w\}\ ,$$

es un lenguaje recursivamente enumerable (modificando el comportamiento de $\mathcal{M}_U$, por ejemplo, se obtiene una Máquina de Turing que indica si M para con la cadena w), pero no es recursivo.

Para demostrarlo, se reduce $\mathcal{L}_d$ a $\mathcal{L}_H$, sabiendo que $\mathcal{L}_d$ es un lenguaje no recursivamente enumerable.

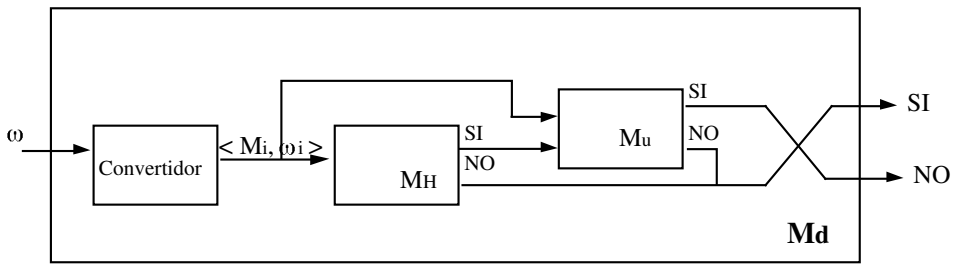
Lema 9.2 *$\mathcal{L}_H$ es un lenguaje recursivamente enumerable no recursivo.*

Demostración:

Supóngase que el problema de parada es decidible; por lo tanto, la Máquina de Turing que reconoce el lenguaje $\mathcal{L}_H$, $\mathcal{M}_H$, ante una entrada $\langle \text{M}, w\rangle$ respondería SI o NO dependiendo de si M para o no para al trabajar sobre w.

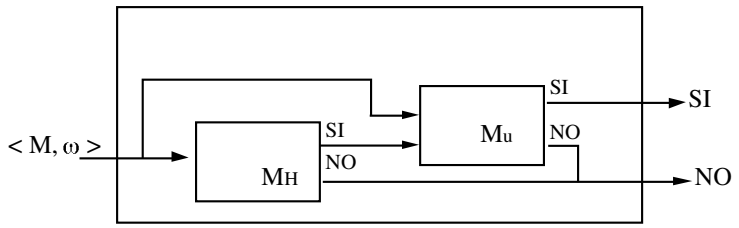
Si esto fuera cierto, se podría construir el siguiente algoritmo para reconocer $\mathcal{L}_d$:

Esta Máquina de Turing, a la que se llamaría $\mathcal{M}_d$, determinaría si una cadena w_i es aceptada por M_i, es decir, si la cadena w pertenece o no al lenguaje $\mathcal{L}_d$. Puesto que es imposible, $\mathcal{M}_H$ no puede existir tal y como se ha descrito y $\mathcal{L}_H$ no puede ser un lenguaje recursivo.



c.q.d

En la construcción anterior, nótese que si se sabe que M_i para al trabajar sobre w_i, entonces seguro que al suministrar esa información a $\mathcal{M}_U$, su respuesta será del tipo SI/NO.



Por lo tanto, si $\mathcal{L}_H$ fuera un lenguaje recursivo, entonces también $\mathcal{L}_U$ sería un lenguaje recursivo. Y existiría la posibilidad de desarrollar herramientas automáticas que permitieran comprobar si un algoritmo es o no es correcto. Ello imposibilita que se pueda disponer de un método automático de "análisis semántico[3]", similar a las herramientas de "análisis léxico" o de "análisis sintáctico".

En el libro de Douglas R. Hofstadter, "Gödel, Escher, Bach: Un eterno y grácil bucle", se hace la siguiente reflexión sobre las dificultades de tal análisis semántico: el autor comienza la reflexión sobre la diferencia entre apreciar los aspectos sintácticos de una forma (reconocer una fórmula bien construida, apreciar las líneas y colores de una pintura o ser capaces de leer una partitura de música) y los aspectos semánticos asociados a dicha forma (reconocer la verdad o falsedad de una fórmula o los sentimientos que puedan inspirar una pintura o una determinada pieza de música):

> *"Subjetivamente, se percibe que los mecanismos de extracción de la significación interior carecen por completo de parentesco con los procedimientos de decisión que verifican la presencia o ausencia de una cualidad específica, como, por ejemplo, el carácter de una fórmula bien formada. Tal vez sea por ello que la significación interior sea algo que descubre más cosas de sí misma a medida que el tiempo pasa. A este respecto jamás se puede estar seguro, de la misma forma en que sí es posible estarlo a propósito de lo bien formado, de que uno ha finiquitado el tema.*

[3]No debe confundirse el concepto que aquí se pretende describir con la fase habitualmente denominada de análisis semántico en la compilación; el concepto que se pretende desarrollar aquí es el "entender el significado de".

> *Esto nos propone la posibilidad de trazar una distinción entre los dos tipos sentidos de "forma" de las pautas que hemos comentado. Primero, tenemos cualidades [...] que pueden ser aprehendidas mediantes* verificaciones predictiblemente finalizables*. Propongo llamar a estas las* cualidades sintácticas *de la forma. [...] los aspectos* semánticos *de la forma son aquellos que no pueden ser verificados dentro de un lapso predictible: requieren* verificaciones de finalización imprevisible *[...]. Por lo tanto, las propiedades "semánticas" están vinculadas con búsquedas no finalizables ya que, en un sentido importante, la significación de un objeto no está situada en el objeto mismo. Esto no equivale a sostener que no es posible captar la significación de ningún objeto [...]. Con todo siempre quedan aspectos de aquella que siguen ocultos durante lapsos no previsibles. [...]*
>
> *Así, otra manera de caracterizar la diferencia entre propiedades "sintácticas" y "semánticas", es la observación de que las primeras residen en el objeto bajo examen, en tanto que las segundas dependen de sus relaciones con una clase potencialmente infinita de otros objetos."*

9.3. Teorema de Rice. Más Problemas Indecidibles.

Hasta este momento se ha asociado la decidibilidad de un determinado problema a la existencia de un lenguaje recursivo que lo describa.

El Teorema de Rice da una caracterización más simple, pero sólo se puede aplicar a los problemas de decisión que están relacionados con *propiedades de lenguajes recursivamente enumerables*. Estos problemas comparten un enunciado genérico:

> *"Sea una Máquina de Turing M. ¿L(M) cumple la propiedad X?"*

y los lenguajes asociados son de la forma,

$$\mathcal{L}_X = \{\langle \mathrm{M} \rangle \mid \mathrm{L(M)} \text{ tiene la propiedad X}\} .$$

La siguiente subsección muestra un ejemplo de cómo estudiar este tipo de problemas y los lenguajes asociados. En el apéndice 1, sección 9.5, se puede encontrar otro ejemplo.

9.3.1. El Problema de la Vaciedad.

> *"Sea una Máquina de Turing M. ¿L(M) $\neq \emptyset$?".*

Es un problema indecidible. Para estudiarlo, se trabaja con el lenguaje asociado

$$\mathcal{L}_{ne} = \{\langle \mathrm{M} \rangle \mid \mathrm{L(M)} \neq \emptyset\} ,$$

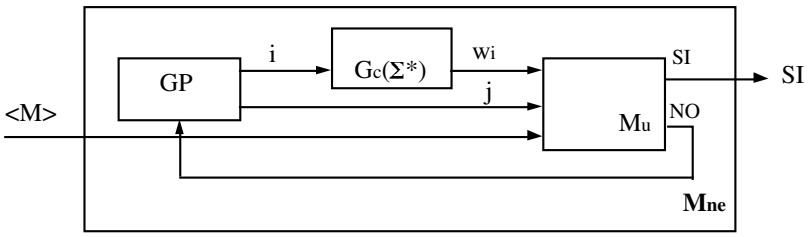
que es el conjunto de los códigos de todas las Máquinas de Turing que aceptan al menos una cadena. También se estudiará su lenguaje complementario,

$$\mathcal{L}_e = \{\langle \mathrm{M} \rangle \mid \mathrm{L(M)} = \emptyset\} ,$$

que incluye tanto a las cadenas binarias que no representan Máquinas de Turing (no se ajustan al formato elegido para codificarlas), como a las cadenas binarias que representan Máquinas de Turing que reconocen el lenguaje vacío. Se demostrará que $\mathcal{L}_{ne}$ es un lenguaje recursivamente enumerable no recursivo y que $\mathcal{L}_e$ no es recursivamente enumerable.

Lema 9.3 *$\mathcal{L}_{ne}$ es un lenguaje recursivamente enumerable.*

Demostración:



Basta con construir una Máquina de Turing, $\mathcal{M}_{ne}$, que, dada la cadena $\langle \mathrm{M} \rangle$ como entrada, simule el comportamiento del generador de pares, $G_p(i, j)$, de forma que por cada par *(i,j)* generado, $\mathcal{M}_U$ simule el comportamiento de M sobre la cadena *i-ésima* en orden canónico en *j* pasos. En el momento en que $\mathcal{M}_U$ acepte una cadena, $\mathcal{M}_{ne}$ aceptará la cadena $\langle \mathrm{M} \rangle$.

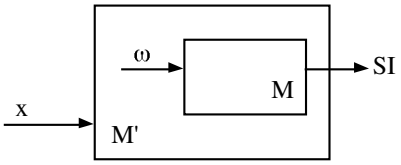
c.q.d.

Lema 9.4 *$\mathcal{L}_e$ NO es un lenguaje recursivo.*

Demostración:

La demostración se realiza por contradicción: si $\mathcal{L}_e$ fuera recursivo, habría un algoritmo, la Máquina de Turing $\mathcal{M}_e$, que aceptaría el lenguaje $\mathcal{L}_e$ y que siempre pararía. Si esta Máquina de Turing existiera entonces existiría también un algoritmo para reconocer $\mathcal{L}_U$ y eso es imposible. Por lo tanto, mediante la reducción de $\mathcal{L}_e$ a $\mathcal{L}_U$, se demostrará que $\mathcal{L}_e$ no puede ser recursivo.

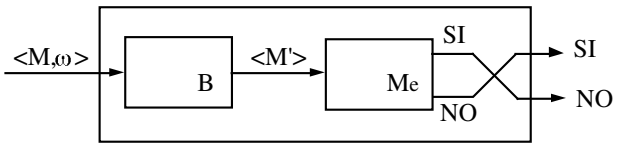
La reducción utiliza la siguiente Máquina de Turing, M'. Su funcionamiento es el siguiente: dada la cadena $\langle$M,$w\rangle$, la máquina ignora su propia entrada y simula el comportamiento de la máquina M con la cadena w. Si M acepta w, entonces M$'$ llega a un estado de aceptación.



Con este comportamiento, el lenguaje reconocido por M$'$ es el siguiente:

$$L(M') = \begin{cases} \Sigma^* \; si \; M \; acepta \; w \\ \emptyset \; si \; M \; no \; acepta \; w \end{cases} .$$

Para la demostración no se utiliza directamente la máquina M$'$, sino un algoritmo B tal que si su entrada es la cadena $\langle$M,$w\rangle$, produce como salida la cadena $\langle$M$'\rangle$, siendo M$'$ la Máquina de Turing descrita anteriormente. Con este algoritmo se puede construir la siguiente Máquina de Turing, $\mathcal{C}$:



$$Si \; M \; acepta \; w \Rightarrow L(M') \neq \emptyset \Rightarrow \mathcal{M}_e \; dice \; NO \Rightarrow \mathcal{C} \; dice \; SI,$$
$$Si \; M \; no \; acepta \; w \Rightarrow L(M') = \emptyset \Rightarrow \mathcal{M}_e \; dice \; SI \Rightarrow \mathcal{C} \; dice \; NO.$$

Es decir, $\mathcal{C}$ acepta aquellas cadenas $\langle$M,$w\rangle$ tales que M acepta w y rechaza las cadenas $\langle$M,$w\rangle$ tales que M no acepta $w \Rightarrow \mathcal{L}_U$ es un lenguaje recursivo.

Se ha llegado a una contradicción, por lo que la suposición de que existe el algoritmo $\mathcal{M}_e$ (la suposición de que $\mathcal{L}_e$ es recursivo) tiene que ser falsa.

c.q.d.

Como consecuencia de los lemas 9.3 y 9.4 se tiene que,

- $\mathcal{L}_e$ no es recursivo,
- $\mathcal{L}_{ne}$ es recursivamente enumerable,

$\Rightarrow$ $\mathcal{L}_e$ NO es un lenguaje recursivamente enumerable (ya que entonces tanto él como $\mathcal{L}_{ne}$ serían recursivos).

Por lo tanto, *el problema de la Vaciedad es indecidible*.

9.3.2. El Teorema de Rice.

En la subsección anterior se ha visto que no se puede decidir si el lenguaje aceptado por una Máquina de Turing es vacío o no. Utilizando técnicas de construcción similares se llegaría a la misma conclusión sobre cuestiones tales como si un lenguaje es o no recursivo, es o no infinito, es o no finito, es o no regular, es o no de contexto libre, si tiene o no un número par de cadenas...

Todos estos resultados están basados el Teorema de Rice e indican qué se puede decidir sobre el lenguaje aceptado por una Máquina de Turing. Para estudiarlos, hay que definir previamente el concepto de propiedad de lenguajes recursivamente enumerables.

Definición 9.6 (Propiedad) *Una propiedad, $\mathcal{P}$, es un conjunto de lenguajes recursivamente enumerables, siendo cada uno de ellos un subconjunto de $(0+1)^*$. Un lenguaje L tiene la propiedad $\mathcal{P}$ si L es un elemento de $\mathcal{P}$.*

Ejemplo:

El conjunto $\{L \mid L$ es infinito$\}$ representa la propiedad de infinitud.

Definición 9.7 (Propiedad Trivial) *Una propiedad es trivial si está vacía o si está formada por todos los lenguajes recursivamente enumerables.*

Definición 9.8 *Dada una propiedad P, se llama $\mathcal{L}_\mathcal{P}$ al lenguaje formado por cadenas que son los códigos de las Máquinas de Turing que reconocen lenguajes que tienen la propiedad $\mathcal{P}$,*

$$\mathcal{L}_\mathcal{P} = \{\langle M\rangle \mid L(M) \in \mathcal{P}\}.$$

Los resultados estudiados hasta el momento, permiten enunciar la siguiente caracterización:

"Una propiedad $\mathcal{P}$ es *decidible* $\Leftrightarrow$ $\mathcal{L}_\mathcal{P}$ es un lenguaje *recursivo*".

El Teorema de Rice brinda otra caracterización, más sencilla. Se aplica *sólo a lenguajes que representan propiedades de los lenguajes recursivamente enumerables*, es decir, propiedades de lenguajes que son aceptados por alguna Máquina de Turing. El principal resultado de Rice afirma que *sólo son decidibles las propiedades que son triviales*, es decir, aquellas

propiedades que o bien las cumplen todos los lenguajes R.E., o bien no las cumple ningún lenguaje R.E.

Teorema 9.3 (Teorema de Rice) *Cualquier propiedad no trivial $\mathcal{P}$ es INDECIDIBLE.*

Es decir, $\mathcal{L}_{\mathcal{P}}$ es NO recursivo si $\mathcal{P}$ es NO trivial[4].

De este teorema, se puede deducir, por ejemplo, el siguiente resultado:

Corolario 9.1 *Las siguientes propiedades de lenguajes recursivamente enumerables son no decidibles:*

1. *ser o no ser vacío (*emptiness*),*
   $\mathcal{P} = \{L \mid L = \emptyset\},\ \mathcal{L}_{\mathcal{P}} = \{\langle M\rangle \mid L(M) = \emptyset\}$.
2. *ser o no ser finito (*finiteness*),*
   $\mathcal{P} = \{L \mid L\ es\ finito\},\ \mathcal{L}_{\mathcal{P}} = \{\langle M\rangle \mid L(M)\ es\ finito\}$.
3. *ser o no ser regular (*regularity*),*
   $\mathcal{P} = \{L \mid L\ es\ regular\},\ \mathcal{L}_{\mathcal{P}} = \{\langle M\rangle \mid L(M)\ es\ regular\}$.
4. *ser o no ser de contexto libre (*context-freedom*),*
   $\mathcal{P} = \{L \mid L\ es\ lcl\},\ \mathcal{L}_{\mathcal{P}} = \{\langle M\rangle \mid L(M)\ es\ lcl\}$.

Además del resultado que caracteriza si una propiedad $\mathcal{P}$ es o no es decidible y, por lo tanto, si $\mathcal{L}_{\mathcal{P}}$ es o no es recursivo, Rice establece en otro teorema las tres condiciones necesarias y suficientes para establecer cuando $\mathcal{L}_{\mathcal{P}}$ sería o no un lenguaje recursivamente enumerable.

[4]La demostración de este teorema se puede encontrar en la bibliografía.

Teorema 9.4 (Segundo Teorema de Rice) $\mathcal{L}_\mathcal{P} = \{\langle M\rangle \mid L(M) \in \mathcal{P}\}$ *es L.R.E.* $\Leftrightarrow$ $\mathcal{P}$ *satisface las tres condiciones siguientes:*

1. *Condición de Inclusión: Si* $L_1 \in \mathcal{P}$ *y* $L_1 \subseteq L_2$*, siendo* L_2 *un L.R.E., entonces* $L_2 \in \mathcal{P}$.
2. *Si L es un lenguaje infinito que está en* $\mathcal{P}$*, entonces hay un subconjunto finito de L en* $\mathcal{P}$.
3. *El conjunto de lenguajes finitos que pertenecen a* $\mathcal{P}$ *es* enumerable *(es decir, hay una Máquina de Turing que puede generar todas las cadenas de todos los lenguajes finitos de* $\mathcal{P}$*).*

El enunciado, discusión y demostración completa del segundo teorema de Rice se puede encontrar en la bibliografía. Aquí se destacarán únicamente los siguientes corolarios:

Corolario 9.2 *Las siguientes propiedades de lenguajes R.E. son indecidibles y los lenguajes asociados son lenguajes no recursivamente enumerables:*

1. *ser vacío,*
   $\mathcal{P} = \{L \mid L = \emptyset\}$, $\mathcal{L}_\mathcal{P} = \{\langle M\rangle \mid L(M) = \emptyset\}$ *(=* $\mathcal{L}_e$*).*
2. *ser recursivo,*
   $\mathcal{P} = \{L \mid L\ es\ recursivo\}$, $\mathcal{L}_\mathcal{P} = \{\langle M\rangle \mid L(M)\ es\ recursivo\}$ *(=* $\mathcal{L}_r$*).*
3. *no ser recursivo,*
   $\mathcal{P} = \{L \mid L\ no\ es\ recursivo\}$,
   $\mathcal{L}_\mathcal{P} = \{\langle M\rangle \mid L(M)\ no\ es\ recursivo\}$ *(=* $\mathcal{L}_{nr}$*).*
4. *ser regular,*
   $\mathcal{P} = \{L \mid L\ es\ regular\}$, $\mathcal{L}_\mathcal{P} = \{\langle M\rangle \mid L(M)\ es\ regular\}$.
5. *tener una única cadena,*
   $\mathcal{P} = \{L \mid L\ sólo\ tiene\ una\ cadena\}$,
   $\mathcal{L}_\mathcal{P} = \{\langle M\rangle \mid L(M)\ sólo\ tiene\ una\ cadena\}$.

Todos los casos expuestos en el corolario 9.2 no cumplen la condición de inclusión.

Corolario 9.3 *Las siguientes propiedades de lenguajes R.E. son indecidibles y los lenguajes asociados son lenguajes recursivamente enumerables:*

1. *no ser vacío,*
   $\mathcal{P} = \{L \mid L \neq \emptyset\},\ \mathcal{L}_{\mathcal{P}} = \{\langle M\rangle \mid L(M) \neq \emptyset\}$ *(=*$\mathcal{L}_{ne}$*).*
2. *contener al menos 10 cadenas,*
   $\mathcal{P} = \{L \mid L\ tiene\ 10\ o\ más\ cadenas\}$,
   $\mathcal{L}_{\mathcal{P}} = \{\langle M\rangle \mid L(M)\ tiene\ 10\ o\ más\ cadenas\}$.
3. $w \in L$*, para una cadena dada* w*,*
   $\mathcal{P} = \{L \mid w \in L\},\ \mathcal{L}_{\mathcal{P}} = \{\langle M\rangle \mid w \in L(M)\}$.
4. *tener intersección no vacía con* $\mathcal{L}_U$*,*
   $\mathcal{P} = \{L \mid L \cap \mathcal{L}_U \neq \emptyset\},\ \mathcal{L}_{\mathcal{P}} = \{\langle M\rangle \mid L(M) \cap \mathcal{L}_U \neq \emptyset\}$.

Los teoremas de Rice son de fácil aplicación, pero su ámbito se restringe al estudio de propiedades de los lenguajes R.E. Así, no se pueden aplicar tal cual, por ejemplo, al estudio de las propiedades de la Máquina de Turing que reconoce el lenguaje[5].

Lo que sí se puede intentar es relacionar cuestiones de decidibilidad que caen fuera de dicho ámbito con una propiedad de lenguajes R.E., a la que sí se pueda aplicar el teorema de Rice. En el apéndice 2, sección 9.6 se muestra un ejemplo.

9.4. La Indecidibilidad del Problema de la Correspondencia de Post.

Los Sistemas de Correspondencia de Post fueron formulados por Emil Post en 1931, la misma época en la que Turing formuló su modelo de computación; la idea básica era la misma en ambos modelos formales y, de hecho, está demostrado que son equivalentes en cuanto a poder computacional.

Al margen de su propio interés como modelo de computación, este problema sirve de conexión entre los resultados sobre Indecidibilidad en el Problema de Aceptación, "*¿M acepta w?*" (y en el Problema de la Parada, "*¿M para con w?*"), con cuestiones indecidibles en el ámbito de las gramáticas y lenguajes de contexto libre.

[5]En este caso concreto, habría que distinguir entre cuestiones referidas a "características técnicas" de la Máquina de Turing, que normalmente son claramente decidibles, como decidir si una MT M tiene o no un número par de estados (siempre se puede diseñar una MT M'que cuente el número de estados de M, dada su codificación ⟨M⟩), y entre cuestiones referidas al lenguaje reconocido por una MT, como la planteada en el ejemplo.

Definición 9.9 (Sistemas de Correspondencia de Post) *Una instancia del Problema de la Correspondencia de Post (PCP), se denomina un Sistema de Correspondencia de Post (SCP) y consta de tres elementos: un alfabeto Σ y dos conjuntos A y B de cadenas de Σ^+, tales que A y B contienen el mismo número de cadenas. Si*

$$A = \{u_1, u_2, \ldots, u_k\} \text{ y } B = \{v_1, v_2, \ldots, v_k\},$$

una solución para esta instancia del PCP es una secuencia de índices $i_1, i_2, \ldots, i_n$ tal que

$$u_{i_1} u_{i_2} \ldots u_{i_n} = v_{i_1} v_{i_2} \ldots v_{i_n}.$$

Por ejemplo:

A = {a, abaaa, ab},

B = {aaa, ab, b}

Solución: i_1 = 2, $i_2 = i_3$ = 1, i_4 = 3, $abaaa|a|a|ab = ab|aaa|aaa|b$.

Normalmente se obtienen una visión más clara del problema si se ve como una colección de bloques

$$\boxed{\frac{u_1}{v_1}}\boxed{\frac{u_2}{v_2}}\cdots\boxed{\frac{u_k}{v_k}}$$

y se busca una secuencia de bloques tal que la cadena superior es igual a la inferior.

El ejemplo anterior puede verse, entonces, como:

$$\boxed{\frac{a}{aaa}}, \boxed{\frac{abaaa}{ab}}, \boxed{\frac{ab}{b}}$$

y la solución la secuencia

$$\boxed{\frac{abaaa}{ab}}\boxed{\frac{a}{aaa}}\boxed{\frac{a}{aaa}}\boxed{\frac{ab}{b}}$$

siendo iguales la cadena superior y la inferior.

El PCP consiste en el problema de determinar si un SCP arbitrario tiene o no una solución. El PCP es un problema INDECIDIBLE. Se puede demostrar a través del PCPM, Problema de la Correspondencia de Post Modificado, en el cual la secuencia de índices debe ser

$$1, i_2, \ldots, i_n \text{ tal que } u_1 u_{i_2} \ldots u_{i_n} = v_1 v_{i_2} \ldots v_{i_n}.$$

El siguiente resultado establece la conexión entre el PCP y el PCPM:

Lema 9.5 *Si el PCP fuese DECIDIBLE, lo sería también el PCPM.*

La idea básica para demostrar este lema sería similar al siguiente razonamiento: si se conoce una solución del PCP, con "cambiar dos bloques de sitio" en la instancia se obtiene una solución del PCPM. Si este lema es cierto, también lo es su contrarrecíproco:

Lema 9.6 *Si el PCPM es INDECIDIBLE, entonces también lo es el PCP.*

Según esto, para establecer la indecidibilidad del PCP, basta con establecer la indecidibilidad del PCPM. Esto se puede hacer por reducción del Problema de la Aceptación, "*¿M acepta w?*", al PCPM. Puesto que este problema es indecidible, entonces el PCPM también lo es.

Teorema 9.5 *El PCPM es INDECIDIBLE.*

La demostración de este teorema consiste en, primero, establecer la reducción entre el PCPM y el Problema de la Aceptación, para poder concluir entonces que el PCPM es indecidible. La reducción se puede hacer mediante la siguiente construcción,

Sea M, con alfabetos Σ y Γ, y sea $w \in \Sigma^*$. Para ver si $w \in$ L(M), se estudia si la siguiente instancia del PCPM

$$\text{A} = \{u_1, u_2, \ldots, u_k\}, \text{B} = \{v_1, v_2, \ldots, v_k\},$$

tiene solución, sabiendo que los conjuntos A y B se forman a partir de la función de transición de la Máquina de Turing M, mediante la construcción de cinco grupos de fichas: Sea \$ $\notin \Gamma$, sea q_1 el estado inicial de M,

Grupo 1: $\boxed{\frac{\$}{\$q_1 w \$}}$, siendo q_1 el estado inicial de M.

Grupo 2: $\boxed{\frac{\$}{\$}}$, $\boxed{\frac{\gamma}{\gamma}}, \forall\, \gamma \in \Gamma, \gamma \neq \mathtt{B}$

Grupo 3: Formado a partir de f, distinguiendo entre cuatro tipos de transiciones,

1. $f(q, \sigma) = (p, \tau, R) \longrightarrow \boxed{\frac{q\sigma}{\tau p}}$
2. $f(q, \mathtt{B}) = (p, \tau, R) \longrightarrow \boxed{\frac{q\$}{\tau p\$}}$
3. $f(q, \sigma) = (p, \tau, L) \longrightarrow \boxed{\frac{\gamma q \sigma}{p \gamma \tau}}, \forall\, \gamma \in \Gamma, \gamma \neq \mathtt{B}$

4. $f(q, \mathtt{B}) = (p, \tau, L) \longrightarrow \boxed{\frac{\gamma q\$}{p\gamma\tau\$}}, \forall\, \gamma \in \Gamma, \gamma \neq \mathtt{B}$

Grupo 4: A partir de los estados finales, $\forall\, q \in F,\ \forall\, \sigma, \tau \in \Gamma - \{\mathtt{B}\}$,

$$\boxed{\frac{\sigma q\tau}{q}}, \boxed{\frac{\sigma q\$}{q\$}}, \boxed{\frac{\$q\tau}{\$q}}$$

Grupo 5: $\forall\, q \in F$,

$$\boxed{\frac{q\$\$}{\$}}$$

Además de establecer la reducción anterior, para completar la demostración del teorema 9.5, se debe probar el siguiente resultado:

Lema 9.7 *M acepta $w \Leftrightarrow$ hay una solución a la instancia derivada del PCPM.*

La demostración, que no se verá, es bastante intuitiva. Se puede encontrar en el libro de Dean Kelley, "Teoría de Autómatas y Lenguajes Formales", capítulo 6.

Del teorema 9.5 y de los lemas 9.5 y 9.6 se concluye que:

Teorema 9.6 *El PCP es INDECIDIBLE.*

Una de las áreas de interés del PCP es que sirve como herramienta para establecer la decidibilidad de cuestiones relacionadas con los LCL. Sirvan las dos siguientes como ejemplo:

1. **El problema de la intersección vacía de las gramáticas de contexto libre es INDECIDIBLE.**

   La idea es construir a partir de una instancia del PCP,

   $$\text{A} = \{u_1, u_2, \ldots, u_k\},\ \text{B} = \{v_1, v_2, \ldots, v_k\},\ u_i, v_i \in \Sigma^+,$$

   y del conjunto de símbolos C = $\{a_1, a_2, \ldots, a_k\}$, $a_i \notin \Sigma$, las gramáticas de contexto libre G_A y G_B,

   $$G_A = \langle \{S_A\}, \Sigma \cup C, S_A, P_A \rangle,\ \ G_B = \langle \{S_B\}, \Sigma \cup C, S_B, P_B \rangle$$

   con

   $$\begin{array}{lll} P_A\text{:} & S_A \rightarrow u_i S_A a_i \mid u_i a_i, & \text{i=1,2,...,k ,} \\ P_B\text{:} & S_B \rightarrow v_i S_B a_i \mid v_i a_i, & \text{i=1,2,...,k .} \end{array}$$

Por lo tanto, las cadenas generadas por G_A serán del tipo

$$u_{i_1} u_{i_2} \dots u_{i_{n-1}} u_{i_n} a_{i_n} a_{i_{n-1}} \dots a_{i_2} a_{i_1},$$

y las cadenas generadas por G_B serán del tipo

$$v_{i_1} v_{i_2} \dots v_{i_{n-1}} v_{i_n} a_{i_n} a_{i_{n-1}} \dots a_{i_2} a_{i_1}.$$

Si $L(G_A) \cap L(G_B) \neq \emptyset$, hay alguna cadena que pertenece a ambos lenguajes. Para que esto se cumpla, tiene que haber alguna cadena tal que

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

ya que la segunda parte es la misma en ambas.

De ahí se sigue que $L(G_A) \cap L(G_B)$ es distinto de vacío si el PCP tiene solución. Luego es indecidible.

2. **El problema de la ambigüedad de las gramáticas de contexto libre es INDECIDIBLE.**

   Para demostrarlo, basta con construir

   $$G_A = \langle \{S, S_A, S_B\}, \Sigma \cup C, S, P \rangle$$

   con $P: \ P_A \cup P_B \cup \{S \to S_A \mid S_B\}$.

   Con un razonamiento similar al anterior, se ve que para que la gramática sea ambigüa, debe existir una cadena que pueda derivarse de S_A o de S_B, indistintamente, y eso ocurre cuando la instancia asociada del PCP tiene solución. Por lo tanto, el problema es indecidible.

9.5. Apéndice 1: Otro Ejemplo de Reducción. El Problema de la Recursividad.

"Sea una Máquina de Turing M. ¿L(M) es recursivo?".

Es un problema indecidible. Para estudiarlo, se trabaja con el lenguaje asociado

$$\mathcal{L}_r = \{\langle \text{M} \rangle \mid \text{L(M) es recursivo}\},$$

que es el conjunto de los códigos de todas las Máquinas de Turing que aceptan un lenguaje recursivo[6] y su lenguaje complementario,

$$\mathcal{L}_{nr} = \{\langle \text{M} \rangle \mid \text{L(M) no es recursivo}\},$$

que es el conjunto de los códigos de todas las Máquinas de Turing que aceptan lenguajes no recursivos. Se demostrará que ni $\mathcal{L}_r$ ni $\mathcal{L}_{nr}$ son lenguajes recursivamente enumerables.

Lema 9.8 *$\mathcal{L}_r$ no es un lenguaje recursivamente enumerable.*

Demostración:

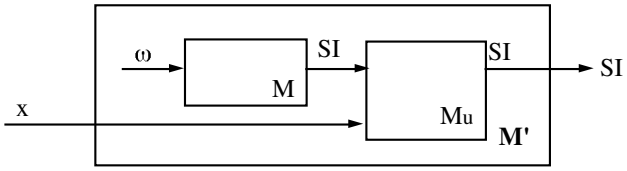
La demostración se realiza por contradicción: Si $\mathcal{L}_r$ fuera recursivamente enumerable, habría una Máquina de Turing, $\mathcal{M}_r$, que aceptaría aquellas cadenas que codifiquen Máquinas de Turing cuyos lenguajes son recursivos. Se probará que la existencia de esa Máquina de Turing es imposible, ya que si existiera se podría reconocer el complementario de $\mathcal{L}_U$, $\bar{\mathcal{L}}_U$, y eso es imposible. Por lo tanto, mediante la reducción de $\mathcal{L}_r$ a $\bar{\mathcal{L}}_U$, se demostrará que $\mathcal{L}_r$ no puede ser recursivamente enumerable.

La reducción utiliza la siguiente Máquina de Turing, M′. Su funcionamiento es el siguiente: dada la cadena $\langle$M,$w\rangle$, la máquina ignora, en principio, su propia entrada y simula el comportamiento de la máquina M con la cadena w. Si M acepta w, entonces M′ trabaja con su propia entrada, llegando a un estado de aceptación si la cadena suministrada pertenece a $\mathcal{L}_U$.

Con este comportamiento, el lenguaje reconocido por M′ es el siguiente:

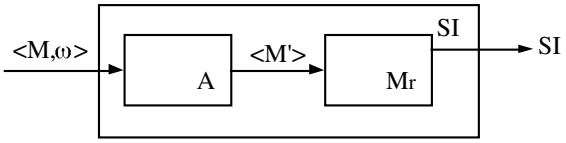
$$L(M') = \begin{cases} \mathcal{L}_U \; si \; M \; acepta \; w \\ \emptyset \; si \; M \; no \; acepta \; w \end{cases}.$$

[6]Nótese que $\mathcal{L}r$ no coincide con el conjunto $\{\langle$M$\rangle \mid$ M para con todas las cadenas$\}$ ya que el hecho de que un lenguaje sea recursivo sólo implica que existe al menos UNA MT que siempre para, pero puede ser que haya otras que también lo reconozcan y no tengan asegurada la parada con cadenas que no pertenezcan al lenguaje.



Por lo tanto, el lenguaje que acepta M′ es recursivo (∅, que es una expresión regular y, por lo tanto, un lenguaje recursivo) sólo si M no acepta w.

Sea un algoritmo A tal que si la entrada es la cadena ⟨M,w⟩, produce como salida la cadena ⟨M′⟩, siendo M′ la Máquina de Turing descrita anteriormente. Con este algoritmo se puede construir la siguiente Máquina de Turing:



$$Si\ M\ no\ acepta\ w \Rightarrow L(M')(=\emptyset)\ es\ recursivo \Rightarrow \mathcal{M}_r\ dice\ SI \Rightarrow La\ MT\ dice\ SI.$$

Es decir, esta Máquina de Turing acepta aquellas cadenas ⟨M,w⟩ tales que M no acepta $w \Rightarrow \bar{\mathcal{L}}_U$ es un lenguaje recursivamente enumerable.

Se ha llegado a una contradicción, por lo que la suposición de que existe $\mathcal{M}_r$ (la suposición de que $\mathcal{L}_r$ es recursivamente enumerable) tiene que ser falsa.

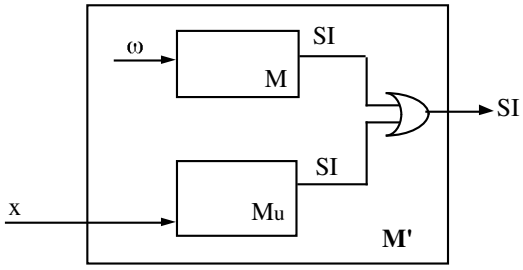
c.q.d.

Lema 9.9 *$\mathcal{L}_{nr}$ NO es un lenguaje recursivamente enumerable.*

Demostración:

También se demostrará por contradicción. Si esta hipótesis fuera cierta, si $\mathcal{L}_{nr}$ fuera R.E., entonces existiría una Máquina de Turing, $\mathcal{M}_{nr}$, que reconocería sus cadenas. Si esta máquina existiera se podría probar que $\bar{\mathcal{L}}_U$ es R.E., mediante la reducción de $\mathcal{L}_{nr}$ a $\bar{\mathcal{L}}_U$, lo que es falso.

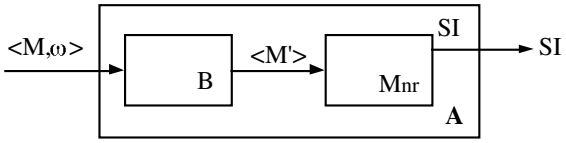
En la reducción se hace uso de la existencia de una Máquina de Turing, M′, cuyo funcionamiento es el siguiente: dada la cadena ⟨M,w⟩, la máquina simula el comportamiento de la máquina M con la cadena w. Si M acepta w, entonces M′ llega a un estado de aceptación. Además, M′ trabaja paralelamente sobre su propia entrada simulando el comportamiento de $\mathcal{M}_U$: en el caso de que M no acepte w, M′ también llega a un estado de aceptación si su entrada es una cadena de $\mathcal{L}_U$.



Con este comportamiento, el lenguaje reconocido por M′ es el siguiente[7]:

$$L(M') = \begin{cases} \Sigma^* \; si \; M \; acepta \; w \\ \mathcal{L}_U \; si \; M \; no \; acepta \; w \end{cases} .$$

Utilizando un algoritmo B tal que, teniendo como entrada la cadena ⟨M,w⟩, su salida es la cadena ⟨M′⟩, siendo M′ la Máquina de Turing descrita anteriormente, se puede construir la siguiente Máquina de Turing, $\mathcal{A}$:



$$Si \; M \; no \; acepta \; w \Rightarrow L(M')(= \mathcal{L}_U) \; no \; es \; recursivo \Rightarrow La \; MT \; \mathcal{A} \; dice \; SI.$$

Es decir, $\mathcal{A}$ acepta aquellas cadenas ⟨M,w⟩ tales que M no acepta $w \Rightarrow \bar{\mathcal{L}}_U$ es un lenguaje recursivamente enumerable.

Se ha llegado a una contradicción, por lo que la suposición de que existe el algoritmo $\mathcal{M}_{nr}$ (la suposición de que $\mathcal{L}_{nr}$ es recursivamente enumerable) tiene que ser falsa.

c.q.d.

[7] $\Sigma^* = \Sigma^* \cup \mathcal{L}u$.

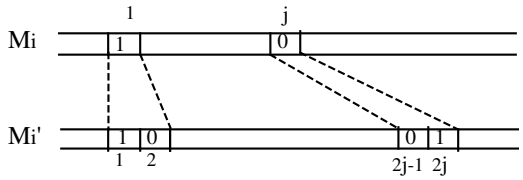
9.6. Apéndice 2: Ejemplo de cómo aplicar Rice fuera de su ámbito.

Es indecidible saber si una Máquina de Turing con alfabeto $\{0, 1, \mathtt{B}\}$*, imprimirá o no 111 si comienza a trabajar con la cinta en blanco.*

Para llegar a esta conclusión se relaciona esta cuestión con la propiedad

$$\mathcal{P} = \{L \mid \lambda \in L\},$$

que, según el teorema de Rice, al no ser trivial es indecidible. Para ello se considera lo siguiente: para cualquier Máquina de Turing, M_i*, es posible construir una Máquina de Turing equivalente,* M_i'*, que utilice 01 para codificar el 0 y 10 para codificar el 1. Es decir, cada símbolo de* M_i *se codifica por un par en* M_i' *y si* M_i *tiene un 0, por ejemplo, en la celda j de la cinta,* M_i' *tiene 01 en las celdas 2j-1 y 2j.*



Si M_i *cambia un símbolo,* M_i' *cambia 0 por 1 y 1 por 0,*

$$M_i\text{: } 0 \longrightarrow 1 \quad M_i'\text{: } 01 \longrightarrow 10$$
$$M_i\text{: } 1 \longrightarrow 0 \quad M_i'\text{: } 10 \longrightarrow 01$$

De esta forma es imposible que M_i' *escriba 111 en su cinta, salvo si modificamos su comportamiento para que escriba 111 cuando* M_i *acepte una cadena. Si utilizamos* M_i' *para simular el comportamiento de* M_i *con la cinta vacía, tenemos que* M_i' *escribe 111, sólo si* M_i *acepta* λ*. Por lo tanto, la cuestión de si una Máquina de Turing arbitraria imprime 111 cuando inicialmente la cinta está en blanco es indecidible.*

Capítulo 10

Introducción a la Complejidad Computacional

Índice General

10.1. Introducción. 177
10.1.1. El Dilema del Contrabandista. 178
10.2. Definiciones Básicas. 180
10.2.1. Clases de Complejidad. 183
10.3. Relaciones entre las Clases de Complejidad Computacional. 184
10.3.1. Relaciones entre Clases Deterministas y No Deterministas. . . . 185
10.3.2. Las Clases de Complejidad Polinómica. 187
10.4. Introducción a la Teoría de Complejidad Computacional. 190

10.1. Introducción.

Una vez establecidos los resultados que permiten saber qué problemas se pueden resolver por medio de un algoritmo, el objetivo es fijar el "precio" a pagar; es decir, cuál va a ser el coste asociado al algoritmo y si dicha solución algorítmica será factible, esto es, con un tiempo de ejecución "razonablemente" breve.

La *Teoría de la Complejidad* permite estimar la dificultad o, mejor dicho, la *tratabilidad* o *intratabilidad* de un problema y establecer si se dispone o no de una solución algorítmica *factible*.

Esta teoría dota a la informática de las herramientas necesarias para, por ejemplo, poder determinar a priori cuál es el comportamiento asintótico de un algoritmo a medida que crece el tamaño del problema (y si merece o no la pena intentar abordarlo) o cuando un determinado problema requiere de una solución tan compleja, que es preferible no intentar

resolver directamente su caso general, sino ir tratando subproblemas que representan casos particulares.

El objetivo de la teoría sería establecer una taxonomía, una clasificación de los problemas, atendiendo a cuál es la complejidad del mejor algoritmo conocido[1] para solucionar dicho problema. Desafortunadamente, esta teoría no suele ofrecer resultados absolutos: no se suelen realizar afirmaciones del tipo,

"... el problema X tiene un nivel de dificultad D ..."

sino como la siguiente:

"... el problema X es tan difícil de resolver como el problema Y; por lo tanto, encontrar una solución eficiente para X es tan difícil como encontrarla para Y..."

No es habitual que se enuncien categóricamente resultados sobre un problema dado (salvo la indicación de que todavía no se dispone de un algoritmo eficiente para ese problema), sino que lo habitual es expresar la dificultad de resolver un problema en términos de comparación con otro. Tal y como se verá, esto conduce a que la teoría clasifique los problemas en *clases* que representan el mismo nivel de dificultad[2]. Entre esas clases, destacan las que comprenden problemas tan difíciles (tan complejos) que aún no se han encontrado algoritmos eficientes para ellos y que llevan a buena parte de la comunidad científica a creer que esos problemas son realmente intratables. De hecho, el reto que se plantea la Teoría de la Complejidad puede ser tan simple como determinar si existen problemas intratables *"per se"* o si dicha comunidad científica es tan lerda como para no haber encontrado todavía la solución eficiente (¡y no resulta especialmente atrayente elegir ninguna de esas dos opciones!).

Los contenidos teóricos de este tema, no son más que la formalización en el marco de la Teoría de Complejidad, de resultados que se pueden haber visto, con mayor o menor detalle, en asignaturas de programación. Como una introducción informal de las principales ideas que se van a desarrollar y formalizar en el tema, se presenta el problema denominado *Dilema del contrabandista*.

10.1.1. El Dilema del Contrabandista.

> *Un aprendiz de contrabandista se está financiando el Máster en "Contrabando de Calidad–Superior" con los beneficios obtenidos al pasar monedas antiguas desde Asia. Para ello utiliza un compartimento secreto en su riñonera, que le permite almacenar, si no quiere que en la frontera descubran su "negocio", sólo 500 gramos de monedas.*

[1] Un algoritmo resuelve un problema, pero dado un problema hay más de un algoritmo que lo resuelve.

[2] A lo largo de este tema, se podría pensar en traducir la palabra "dificultad" por "complejidad"; pero, se pretende diferenciar entre la *dificultad* de resolver un *problema* y la *complejidad* del *algoritmo* que lo resuelva: es decir, cuando se dice que los problemas X e Y son igual de difíciles de resolver, es porque sus mejores algoritmos conocidos tienen una complejidad similar. Por purismo, o por mera deformación profesional (tanto leer en Inglés tiene consecuencias perniciosas sobre el vocabulario propio ;-), se mantiene "dificultad".

Para poder realizar comparaciones, se va a completar el enunciado con dos escenarios distintos:

Primer escenario:

> *Sus suministradores asiáticos le ofrecen 20 monedas distintas; cada una de ellas pesa 50 gramos y sus precios de venta en el mercado negro, sus valores, van desde los 100 euros hasta los 2000 euros.*

Segundo escenario:

> *Sus suministradores asiáticos le ofrecen 20 monedas distintas; cada moneda tiene distinto peso, entre 30 y 200 gramos, y sus valores van desde 100 euros hasta 2000 euros.*

Evidentemente, el dilema del contrabandista consiste en cómo realizar una elección óptima de monedas, de forma que le permita obtener el máximo beneficio en un viaje. Y cada escenario va a suponer una estrategia distinta:

Primer escenario:

> *Si todas las monedas pesan lo mismo, 50 gramos, puede llevar 10 monedas en la riñonera. Obviamente, le interesa elegir las 10 más valiosas, por lo que si ordena las 20 monedas por valor decreciente, le basta con elegir las 10 primeras.*

Segundo escenario:

> *Como cada moneda tiene distinto peso y valor, no es suficiente con realizar una ordenación por valor, ya que no garantiza el máximo beneficio. El contrabandista debería estudiar todas las combinaciones de monedas que pesan menos de 500 gramos y quedarse con la más ventajosa.*

El coste del algoritmo del primer escenario es el coste asociado a ordenar 20 monedas por valor decreciente: aún utilizando un algoritmo tan simple como el de ordenación por selección, la complejidad resulta ser de $O(\frac{N^2}{2})$, siendo N el número de monedas a ordenar. En este escenario, ello viene a suponer unas 200 comparaciones, para obtener un beneficio máximo.

Pero en el segundo escenario, el coste no es tan bajo: hay 2^N formas de seleccionar un conjunto de N monedas; en el ejemplo concreto, $N = 20$, se obtienen unas 2^{20} posibilidades a considerar (aproximadamente, un millón), ver cuáles son factibles y determinar cuál maximiza el beneficio.

De lo anterior, se puede sacar la impresión (correcta) de que el segundo escenario supone un problema "más difícil" de resolver que el problema asociado al primer escenario.

Pero aún falta introducir un nuevo aspecto: en su próximo viaje, el contrabandista se encuentra con una agradable sorpresa. Sus suministradores le informan de que se acaba de descubrir una tumba pre-babilónica con 100 monedas distintas. ¿Cuál es la influencia en los dos escenarios? En el primero, el contrabandista tendrá que realizar una ordenación que le supondrá unas 5000 comparaciones, para maximizar beneficios. En el segundo, sin embargo, ahora tendrá que considerar 2^{100} posibilidades distintas, ¿cuánto tiempo le llevará la elección?

En el primer escenario, el comportamiento es *polinómico*: a medida que crece el tamaño del problema (de 20 a 100, de 100 a 1000, ...) hay que realizar más operaciones, pero el crecimiento viene dado por un factor cuadrático: de 20 a 100, será 5^2, de 100 a 1000, será 10^2...

En el segundo escenario, el comportamiento es *exponencial*: con añadir un único elemento, con incrementar en una unidad el tamaño del problema, el número de operaciones se multiplica por 2; cuando se pasa de 20 a 100 monedas, se pasa de $1{,}000{,}000$ de posibilidades a $1{,}000{,}000{,}000{,}000{,}000{,}000{,}000{,}000$.

Estos dos escenarios vienen a mostrar cuál es la importancia de la escalabilidad de un problema y del comportamiento asintótico de los algoritmos resultantes. Usualmente, en Teoría de Complejidad se estima el comportamiento polinómico como deseable, y los problemas que pueden resolverse con un algoritmo que presenta una función de orden polinómica se consideran *tratables*. Cuando no se puede encontrar un algoritmo polinómico para resolver el problema, tal y como ocurre en el segundo escenario, el problema se considera *intratable*.

Y esto lleva al "dilema del informático": no hay resultados que permitan estimar cuándo un problema puede resolverse por medio de un algoritmo polinómico, a no ser, claro está, que tal algoritmo se haya encontrado. Por lo tanto, no hay ningún resultado teórico que permita establecer que un determinado problema es, por naturaleza, intratable: sólo puede afirmarse que el mejor algoritmo conocido hasta el momento tiene una complejidad superior a la polinómica. Una forma de atacar este "dilema" consiste en establecer *clases* de problemas, de forma que problemas con la misma dificultad estén en la misma clase. Esta clasificación, además, dota al informático de una herramienta interesante en su objetivo de encontrar algoritmos eficientes: si todos los problemas de una determinada clase son de la misma dificultad que un problema que es *representante* de esa clase[3], cualquier resultado o mejoría que se obtenga para dicho problema, podrá aplicarse a todos los demás de su misma clase.

10.2. Definiciones Básicas.

El análisis de algoritmos supone la obtención de su complejidad espacial y/o temporal como una función de la talla del problema. En este contexto, se entiende como *talla* el dato o conjunto de datos que cuando varía, hace que varíe el valor de la complejidad. Por su

[3]Es decir, encontrar una solución eficiente para cualquier problema de esa clase es igual de difícil que encontrar una solución eficiente para ese problema "especial".

parte, la *complejidad espacial* da una medida de la cantidad de objetos manejados en el algoritmo (entendida como una medida de la memoria que consumirá una computación) y la *complejidad temporal* da una medida del número de operaciones realizadas (entendido como una medida del tiempo necesario para realizar una computación).

De esta forma, para realizar el análisis primero hay que determinar cuál es la talla del problema. A continuación, se elige una unidad para poder determinar la complejidad espacial y una unidad para poder determinar cuál es la complejidad temporal. Sin embargo, en ocasiones no hay un criterio claro para escoger dichas unidades y poder establecer las correspondientes medidas. El modelo de Máquina de Turing permite definir fácilmente todos los conceptos anteriores:

- la talla del problema se identifica con la longitud de la cadena de entrada,
- la complejidad espacial se asocia al número de celdas de la Máquina de Turing visitadas, y
- la complejidad temporal viene dada por el número de movimientos del cabezal, asumiendo que cada movimiento se realizará en un tiempo fijo.

Las definiciones formales son:

Definición 10.1 (Complejidad Espacial) *Sea M una Máquina de Turing con una cinta sólo de entrada y k cintas de trabajo (Máquina de Turing offline). Supóngase que, sobre cualquier entrada de longitud* n*, las cabezas de lectura/escritura de las cintas de trabajo de M consultan como máximo* $\mathcal{S}(n)$ *celdas en cualquiera de las cintas, siendo* $\mathcal{S} : \mathcal{N} \longrightarrow \mathcal{N}$.
Entonces se dice que M tiene una complejidad espacial $\mathcal{S}(n)$ *o que es una Máquina de Turing acotada espacialmente por* $\mathcal{S}(n)$*. También se dice que L(M) es un lenguaje con complejidad espacial* $\mathcal{S}(n)$.

Esta definición, al estar basada en una máquina offline, no tiene en cuenta el número de celdas que ocupa la cadena de entrada en el cálculo del consumo espacial (de alguna forma, remarca la diferencia entre la *talla* del problema, *n* y la complejidad espacial, $\mathcal{S}(n)$). De esta forma, puede ocurrir que $\mathcal{S}(n) <$ n . Incluso puede ocurrir que una Máquina de Turing trabaje sobre una cadena vacía. Para evitar problemas en estos casos y dado que para trabajar con la cadena vacía se ha de consultar al menos una celda normalmente, cuando se está hablando de la cota espacial $\mathcal{S}(n)$, se asume el valor $\max(1, \lceil \mathcal{S}(n) \rceil)$.

Definición 10.2 (Complejidad Temporal) *Sea M una Máquina de Turing con k cintas. Supóngase que, sobre cualquier entrada de longitud* n*, M realiza como máximo* $\mathcal{T}(n)$ *movimientos, siendo* $\mathcal{T} : \mathcal{N} \longrightarrow \mathcal{N}$.
Entonces se dice que M tiene una complejidad temporal $\mathcal{T}(n)$ *o que es una Máquina de Turing acotada temporalmente por* $\mathcal{T}(n)$*. También se dice que L(M) es un lenguaje con complejidad temporal* $\mathcal{T}(n)$.

Nótese que, con respecto a la definición de complejidad espacial, en este caso la definición no se hace sobre el modelo offline, sino sobre una Máquina de Turing con *k* cintas de trabajo. Es más, de hecho, se suele hacer hincapié en que al ser necesario leer la cadena entera para poder decidir sobre ella, seguro que $\mathcal{T}(n) \geq n+1$. Esto no es cierto en todos los casos, pero suele asumirse así, ya que esta suposición de partida permite simplificar la discusión de algunos teoremas. Por lo tanto, es usual asumir que cuando se está hablando de la cota temporal $\mathcal{T}(n)$, realmente se trabaja con el valor $\max(n+1, \lceil \mathcal{T}(n) \rceil)$.

La definición de la complejidad espacial se basa en una Máquina de Turing con una cinta sólo de entrada y *k* cintas de trabajo. Para simular *k* cintas de trabajo se puede utilizar una cinta con *2k* sectores (veáse el teorema 7.2); tal y como se realiza la simulación, el número de celdas utilizado en la Máquina de Turing de una cinta de trabajo coincide con el número máximo de celdas utilizado en la Máquina de Turing con *k* cintas de trabajo. De esto se puede concluir que el número de cintas de trabajo no afecta a la definición de complejidad espacial.

Sin embargo, sí que es significativo para la definición de la complejidad temporal, ya que ésta se ve afectada al pasar de una Máquina de Turing con *k* cintas a una Máquina de Turing con una cinta, según se desprende del siguiente resultado[4],

Teorema 10.1 *Supuesto que* $\inf_{n\to\infty} \frac{\mathcal{T}(n)}{n} = \infty$*, sea L un lenguaje aceptado por una Máquina de Turing M de k cintas, con cota temporal* $\mathcal{T}(n)$*. Hay una Máquina de Turing M' de una cinta que acepta L con complejidad temporal* $(\mathcal{T}(n))^2$.

Demostración: (Idea Intuitiva)

Cuando se pasa de una Máquina de Turing con *k* cintas a una Máquina de Turing con una cinta, cada movimiento en la multicinta equivale a un barrido en la de una cinta (teorema 7.2).

El peor caso para este barrido, en cuanto al número de movimientos, es que haya dos cabezales moviéndose siempre en sentido contrario. Si comienzan en la misma posición, después de un movimiento se habrán separado 2 celdas, después de 2 movimientos 4, después de 3 movimientos 6 y, en general, después de *t* movimientos, *2t* celdas.

Como el número de movimientos total es $\mathcal{T}(n)$, la expresión para definir el número de movimientos necesarios para completar los barridos sería,

$$\sum_{t=1}^{\mathcal{T}(n)} 2t \approx (\mathcal{T}(n))^2.$$

♣

[4] La notación $\inf_{n\to\infty} \frac{\mathcal{T}(n)}{n} = \infty$ representa la mayor cota inferior de la sucesión $\frac{\mathcal{T}(n)}{n}, \frac{\mathcal{T}(n+1)}{n+1}, \frac{\mathcal{T}(n+2)}{n+2}, \ldots$ Esta condición es necesaria para poder demostrar aquellos resultados en cuyo enunciado aparece; la idea es que $\mathcal{T}(n)$ y n no son del mismo orden de magnitud, con lo cual $\mathcal{T}(n)$ crece mucho más rápidamente que n.

10.2.1. Clases de Complejidad.

Cuando se realiza el análisis de los algoritmos en programación, nunca se busca una expresión exacta de la función de coste de un determinado algoritmo, sino que interesa más establecer cuál es su comportamiento asintótico. Así se obtiene una medida significativa, pero más simple, de cómo evolucionará dicho algoritmo a medida que la talla del problema crece. Y se introduce el concepto de *orden de coste* que permite, además, clasificar los algoritmos en "familias", según su comportamiento asintótico estimado.

Los teoremas siguientes permiten formalizar este concepto en el ámbito de la Teoría de Complejidad:

Teorema 10.2 (de Compresión) *Sea L un lenguaje aceptado por una Máquina de Turing M con k cintas de trabajo con cota espacial $\mathcal{S}(n)$. Para todo $c > 0$ hay una Máquina de Turing M' acotada espacialmente por $c\mathcal{S}(n)$ que acepta L.*

Demostración: (Idea Intuitiva)

Sea $r \in \mathcal{N} \mid rc \geq 1$. Se construye M' a partir de M, de forma que cada símbolo de su alfabeto codifica un bloque de *r* símbolos del alfabeto de M; por lo tanto, además, habrá que realizar las modificaciones correspondientes en las transiciones para que las máquinas sean equivalentes. Si la máquina M consulta $\mathcal{S}(n)$, M' sólo consultará $\mathcal{S}(n)/r$, es decir, $c\mathcal{S}(n)$.

♣

Teorema 10.3 (de Aceleración Lineal) *Supuesto que $\inf_{n\to\infty} \frac{\mathcal{T}(n)}{n} = \infty$, sea L un lenguaje aceptado por una Máquina de Turing M de k cintas, con cota temporal $\mathcal{T}(n)$. Para todo $c > 0$ hay una Máquina de Turing M' de k cintas, acotada temporalmente por $c\mathcal{T}(n)$ que acepta L.*

La idea de la demostración es similar a la del teorema de compresión (se codifica la cadena en bloques de *r* símbolos; con menos símbolos habrá menos movimientos), pero el aparato matemático es mayor.

Estos teoremas permiten realizar el análisis asintótico de la complejidad espacial y temporal, a fin de comparar el comportamiento de dos Máquinas de Turing: por ejemplo, dadas dos Máquina de Turing con complejidades espaciales $\mathcal{S}_1(n) = 3n^3 + 1$ y $\mathcal{S}_2(n) = n^3 + 5n^2 + 1$, se dice que ambas están acotadas espacialmente por $\mathcal{S}(n) = n^3$. Esto permite definir las *clases de complejidad*.

Definición 10.3 (Clases de Complejidad Espacial)
La familia de lenguajes aceptados por Máquinas de Turing deterministas con complejidad espacial $\mathcal{S}(n)$ es DSPACE($\mathcal{S}(n)$).
La familia de lenguajes aceptados por Máquinas de Turing no deterministas con complejidad espacial $\mathcal{S}(n)$ es NSPACE($\mathcal{S}(n)$).
Son conocidas como clases de complejidad espacial.

Definición 10.4 (Clases de Complejidad Temporal)
La familia de lenguajes aceptados por Máquinas de Turing deterministas con complejidad temporal $\mathcal{T}(n)$ es DTIME($\mathcal{T}(n)$).
La familia de lenguajes aceptados por Máquinas de Turing no deterministas con complejidad temporal $\mathcal{T}(n)$ es NTIME($\mathcal{T}(n)$).
Son conocidas como clases de complejidad temporal.

10.3. Relaciones entre las Clases de Complejidad Computacional.

Los siguientes teoremas establecen las relaciones básicas entre las clases de complejidad espacial:

Teorema 10.4 *Sean $\mathcal{S}$, $\mathcal{S}_1$ y $\mathcal{S}_2$ funciones de $\mathcal{N}$ en $\mathcal{N}$. Se asume que $\mathcal{S}_1(n) \leq \mathcal{S}_2(n)\ \forall\ n$ y que $c > 0$. Entonces*

1. $DSPACE(\mathcal{S}_1(n)) \subseteq DSPACE(\mathcal{S}_2(n))$,
2. $NSPACE(\mathcal{S}_1(n)) \subseteq NSPACE(\mathcal{S}_2(n))$,
3. $DSPACE(\mathcal{S}(n)) = DSPACE(c\mathcal{S}(n))$,

y entre las clases de complejidad temporal:

Teorema 10.5 *Sean $\mathcal{T}$, $\mathcal{T}_1$ y $\mathcal{T}_2$ funciones de $\mathcal{N}$ en $\mathcal{N}$. Se asume que $\mathcal{T}_1(n) \leq \mathcal{T}_2(n)\ \forall\ n$ y que $c > 0$. Entonces*

1. $DTIME(\mathcal{T}_1(n)) \subseteq DTIME(\mathcal{T}_2(n))$,
2. $NTIME(\mathcal{T}_1(n)) \subseteq NTIME(\mathcal{T}_2(n))$,
3. *Si* $\inf_{n\to\infty} \frac{\mathcal{T}(n)}{n} = \infty$, $DTIME(\mathcal{T}(n)) = DTIME(c\mathcal{T}(n))$.

En ambos teoremas, las relaciones *(1)* y *(2)* dan una medida que no es absoluta: se identifica un problema con la clase más restrictiva a la que pertenece (según el mejor algoritmo conocido para resolverlo), pero la relación entre clases se establece con un "$\subseteq$"; es decir, no se cierra la "frontera" entre clases, de forma que un problema puede pasar a una clase inferior (en cuanto se encuentre un algoritmo más eficiente, por ejemplo). Por su parte, la relación *(3)* supone una extensión de los teoremas de compresión y de aceleración lineal.

Además, puede establecerse una relación entre las clases de complejidad temporal y las clases de complejidad espacial:

Teorema 10.6 (Relación espacio/tiempo) *Si* $L \in DTIME(f(n))$ *entonces* $L \in DSPACE(f(n))$.

Demostración:

Si $L \in DTIME(f(n))$ la Máquina de Turing que lo acepta, en el peor de los casos habrá realizado como máximo *f(n)* movimientos, por lo que habrá visitado como mucho *f(n) +1* celdas $\Rightarrow L \in DSPACE(f(n)+1)$. Como $f(n)+1 \leq 2f(n)$, y con la suposición de que la complejidad temporal es al menos *n+1*, por el teorema de compresión $L \in DSPACE(f(n))$.

c.q.d

10.3.1. Relaciones entre Clases Deterministas y No Deterministas.

Un algoritmo determinista emplea procedimientos de decisión que siguen un curso único y predeterminado de forma que, con el mismo juego de entradas, siempre seguirá el mismo comportamiento. Pero es posible diseñar también algoritmos no deterministas, que ante una misma entrada pueden obtener salidas distintas, ya que al tener la posibilidad de elegir una de entre varias opciones, elegirá la más beneficiosa en cada situación.

Por ejemplo, ante el problema de salir de una habitación con cinco puertas, pero con una única puerta que conduzca a la salida, un algoritmo determinista puede consistir en ir probando una a una, hasta dar con la correcta; un algoritmo no determinista escogerá directamente cuál es la puerta correcta. Ambos algoritmos solucionan el mismo problema (se consigue encontrar la salida) e, intuitivamente, parece que el no determinista será más rápido en general que el determinista.

El modelo de Máquina de Turing contempla el determinismo y el no determinismo; y, de hecho, al presentar el modelo no determinista también se introducía, intutitamente, la idea de el no determinismo brinda un modelo de computación más eficiente que el determinista. Esa idea intuitiva se formaliza con el siguiente teorema:

Teorema 10.7 *Sean $\mathcal{S}$ y $\mathcal{T}$ funciones de $\mathcal{N}$ en $\mathcal{N}$. Entonces*

1. $DSPACE(\mathcal{S}(n)) \subseteq NSPACE(\mathcal{S}(n))$,
2. $DTIME(\mathcal{T}(n)) \subseteq NTIME(\mathcal{T}(n))$.

Pero, en el mundo real sólo se dispone de máquinas deterministas[5] y, en consecuencia, un algoritmo no determinista se debe transformar en determinista, si se pretende codificarlo en una máquina real. Por eso, interesa saber cuál es el precio que se debe pagar en esa transformación:

Teorema 10.8 (de Savitch) *Sea $\mathcal{S}$ una función de $\mathcal{N}$ en $\mathcal{N}$. Entonces* $NSPACE(\mathcal{S}(n)) \subseteq DSPACE((\mathcal{S}(n))^2)$.

Es decir, en el caso de la complejidad espacial, el paso de un algoritmo no determinista a determinista supone, como mucho, elevar la función de complejidad al cuadrado.

Teorema 10.9 *Si L es aceptado por una Máquina de Turing no determinista con complejidad temporal $\mathcal{T}(n)$, entonces L es aceptado por una Máquina de Turing determinista con complejidad temporal $d^{\mathcal{T}(n)}$, para alguna constante* d.
Es decir,

$$\textit{si } L \in NTIME(\mathcal{T}(n)) \textit{ entonces } L \in DTIME(d^{\mathcal{T}(n)}),$$

para alguna constante d.

Demostración: (Idea Intuitiva)

Para la demostración, hay que recordar (teorema 7.4) cómo se simula una Máquina de Turing no determinista mediante una Máquina de Turing determinista: la máquina determinista genera por orden canónico todas las cadenas sobre el alfabeto {1,2,...,r}, hasta que encuentra la secuencia de opciones correcta. La cadena que indica la secuencia de opciones correcta ha de tener longitud $\mathcal{T}(n)$; como la generación se hace por orden canónico, antes se habrán generado todas las cadenas de longitud 1, 2,..., $\mathcal{T}(n)$-1. Eso supone generar $r + r^2 + r^3 + .. + r^{\mathcal{T}(n)-1}$ cadenas, como mínimo, antes de finalizar la simulación. De ahí se deriva esa expresión exponencial.

♣

Nótese que una relación que para la complejidad espacial puede, como mucho, acabar resultando cuadrática, en el caso de la complejidad temporal puede llegar a suponer la transformación en una función exponencial. Este resultado tiene importantes consecuencias, tal y como se verá a continuación.

[5] ¿Por ahora? ;-).

10.3.2. Las Clases de Complejidad Polinómica.

Como se comentó en la introducción, se consideran poblemas tratables aquellos problemas que se pueden resolver mediante algoritmos con comportamiento descrito por funciones polinómicas. Por lo tanto, puede resultar interesante agrupar en una misma "superclase" a todos los problemas tratables. Lo que sigue son la definición de esas clases, *PSPACE*, *NPSPACE*, *P* y *NP*, según que se trate de complejidad espacial o temporal, sobre Máquinas de Turing deterministas o no deterministas.

Definición 10.5 *Si L es aceptado por una Máquina de Turing determinista con complejidad espacial polinómica, $\mathcal{S}(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_0$, se dice que L está en la clase de lenguajes PSPACE.*
Si L es aceptado por una Máquina de Turing no determinista con cota espacial polinómica, se dice que L está en la clase NPSPACE.

Definición 10.6 *Si L es aceptado por una Máquina de Turing determinista con complejidad temporal polinómica, $\mathcal{T}(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_0$, se dice que L está en la clase de lenguajes P.*
Si L es aceptado por una Máquina de Turing no determinista con cota temporal polinómica, se dice que L está en la clase NP.

Es posible caracterizar estas clases, así como establecer relaciones entre ellas. En el caso de la complejidad espacial, por ejemplo, y puesto que se cumple la relación,

$$a_k n^k + a_{k-1} n^{k-1} + \ldots + a_0 \leq (|a_k| + |a_{k-1}| + \ldots + |a_0|) n^k,$$

aplicando el teorema 10.4(1) también se cumplirá que

$$DSPACE(a_k n^k + a_{k-1} n^{k-1} + \ldots + a_0) \subseteq DSPACE(n^k).$$

Por lo tanto, se puede caracterizar la clase de lenguajes *PSPACE* como

$$PSPACE = \bigcup_{k=0}^{\infty} DSPACE(n^k)$$

es decir, la unión de todas las clases de complejidad espacial polinómicas sobre Máquinas de Turing deterministas. De forma análoga, aplicando el teorema 10.4(2), para las no deterministas se tiene que

$$NPSPACE = \bigcup_{k=0}^{\infty} NSPACE(n^k).$$

Si se aplica el apartado 1 del teorema 10.7, se sigue que

$$DSPACE(n^k) \subseteq NSPACE(n^k).$$

Por lo tanto, como cada elemento de *PSPACE* está contenido en un elemento de *NPSPACE*, se tiene que

$$PSPACE \subseteq NPSPACE.$$

Pero la relación en sentido contrario también es cierta. Si se aplica el Teorema de Savitch, se obtiene

$$NSPACE(n^k) \subseteq DSPACE(n^{2k}).$$

Es decir, cada elemento de *NPSPACE* también está contenido en un elemento de *PSPACE*, y como las clases *PSPACE* y *NPSPACE* son la unión de cualquier clase de complejidad polinómica, se tiene que

$$NPSPACE \subseteq PSPACE.$$

Si se cumple la doble inclusión se está en condiciones de afirmar lo siguiente:

PSPACE = NPSPACE.

Si un problema se resuelve mediante un algoritmo no determinista de complejidad espacial polinómica, también se puede resolver mediante un algoritmo determinista de complejidad espacial polinómica.

¿Qué ocurre si se intenta llegar al mismo resultado en el ámbito de la complejidad temporal? De nuevo se utiliza la relación,

$$a_k n^k + a_{k-1} n^{k-1} + \ldots + a_0 \leq (|a_k| + |a_{k-1}| + \ldots + |a_0|) n^k,$$

para poder afirmar

$$DTIME(a_k n^k + a_{k-1} n^{k-1} + \ldots + a_0) \subseteq DTIME(n^k),$$

y, por lo tanto, se puede caracterizar la clase de lenguajes *P* como

$$P = \bigcup_{k=1}^{\infty} DTIME(n^k),$$

la unión de todos los problemas que se resuelven con cota temporal polinómica usando Máquinas de Turing deterministas y, de forma análoga,

$$NP = \bigcup_{k=1}^{\infty} NTIME(n^k),$$

como la unión de todos los problemas con cota temporal polinómica sobre máquinas no deterministas.

Para intentar establecer una relación entre ambas clases, similar a la relación entre *PSPACE* y *NPSPACE*, se puede utilizar el apartado 2 del teorema 10.7 y, ya que,

$$DTIME(n^k) \subseteq NTIME(n^k)$$

se puede afirmar que,

$$P \subseteq NP.$$

Pero ahora no es posible establecer la inclusión inversa para intentar establecer la igualdad: el único resultado del que se dispone es el teorema 10.9, que establece que, en general, el paso de trabajar con una máquina no determinista a una máquina determinista puede llegar a suponer un aumento exponencial de la complejidad temporal[6].

Es decir: hay una serie de problemas en la clase *NP* que podrían solucionarse con algoritmos de cota temporal polinómica si se dispusiera de la máquinas no deterministas, gobernadas por algoritmos no deterministas. Mientras tanto, se deben transformar en sus equivalentes deterministas. Y en esta transformación su coste computacional puede llegar a transformarse en exponencial. Y, de hecho, los denominados en programación *Problemas NP*, son problemas que plantean este reto a la comunidad científica: sus algoritmos no deterministas son polinómicos, sus algoritmos deterministas son exponenciales... ¿son problemas *"intratables"* ... o son problemas que no se sabe cómo tratar? ¿existe un algoritmo determinista polinómico para ellos y no se conoce todavía?

Por lo tanto, queda abierta la cuestión

¿P = NP?

seguramente, el problema más importante en Teoría de Computación, hoy en día. La comunida científica se divide entre los que pretenden demostrar la igualdad – si esto se demostrara, los problemas que están en la clase NP también podrían resolverse en tiempo polinómico sobre máquinas deterministas – y los que pretenden demostrar la desigualdad, que P $\subset$ NP – en cuyo caso se sabría que hay problemas que nunca se podrán resolver sobre máquinas deterministas con cota polinómica. Es decir, se trabaja sobre las cuestiones:

¿P = NP? para lo que habría que encontrar una forma de transformar toda Máquina de Turing no determinista en una Máquina de Turing determinista con cota temporal polinómica (sean o no los polinomios del mismo grado).

¿P $\neq$NP? para lo que habría que encontrar un lenguaje que esté en NP y no esté en P. Es decir, demostrar para un determinado lenguaje reconocido con cota polinómica por una Máquina de Turing no determinista que es imposible que lo reconozca una Máquina de Turing determinista con cota polinómica.

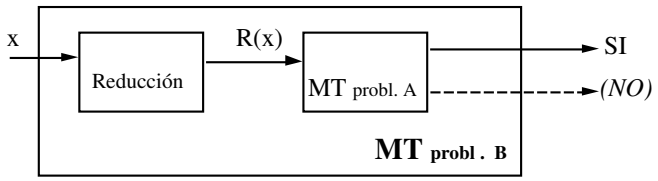
Hay una tercera línea de trabajo cuyo objetivo sería probar que estas cuestiones son indemostrables.

[6]Un polinomio al cuadrado sigue siendo un polinomio, pero si interviene una función exponencial en la transformación...

10.4. Introducción a la Teoría de Complejidad Computacional.

Buena parte de los esfuerzos realizados en la Teoría de la Complejidad Computacional se centra en las clases P y NP y en el estudio de las relaciones entre ambas clases, para intentar establecer resultados que permitan resolver la cuestión *¿P=NP?*. Para ello, además del conocimiento sobre qué son las clases de complejidad y cuáles son las relaciones entre ellas, se utiliza la reducibilidad de lenguajes como herramienta básica.

En el tema 9, se definió, en general, el mecanismo de reducción del lenguaje A (o problema A) al lenguaje B (o problema B), como



En el tema 9 la reducción se utilizaba para poder deducir si el lenguaje B (el problema B) era o no era recursivo (era o no era decidible) a partir del conocimiento que se tenía del lenguaje A (del problema A).

Si lo que se pretende es estudiar si A y B pertenecen a la misma clase de complejidad además es importante asegurar que esa transformación no afecta a la complejidad temporal: debe tenerse en cuenta que para resolver B, se debe resolver A y además realizar la reducción (transformar una cadena de B en una cadena de A). El tiempo de cómputo de la función R, afecta al tiempo de cómputo total. Por eso, si se puede calcular en tiempo polinómico, se dice que B se puede reducir en tiempo polinómico a A, B $<_p$ A.

Definición 10.7 (Reducibilidad Polinómica) *Un lenguaje L_1 es reducible en tiempo polinómico a otro lenguaje L_2, si hay una función de cadena computable en tiempo polinómico, f, para la cual $f(u) \in L_2 \Leftrightarrow u \in L_1$. Es decir, es posible calcular $f(u)$ en tiempo polinómico en una Máquina de Turing. Se denota como $L_1 <_p L_2$.*

La importancia de este concepto se pone de relieve con el siguiente teorema, que garantiza que la reducibilidad polinómica mantiene al lenguaje reducido en la misma clase que el lenguaje al que se reduce:

Teorema 10.10 *Si L_1 es reducible en tiempo polinómico a L_2, entonces:*

1. *si $L_2 \in P \Rightarrow L_1 \in P$,*
2. *si $L_2 \in NP \Rightarrow L_1 \in NP$.*

Demostración:

Las demostraciones son similares en ambos casos: del caso determinista al no determinista sólo varían en que la correspondiente Máquina de Turing sea o no determinista.

Se supone que $L_2 \in \{\text{P}|\text{NP}\}$ y que f reduce L_1 a L_2 en tiempo polinómico.

Sea $w \in L_1 \Rightarrow$ hay una Máquina de Turing, M_1, que acepta *w* con cota polinómica $f(w)$.

Sea $w \in L_1 \Rightarrow f(w) \in L_2 \Rightarrow$ hay una Máquina de Turing, M_2, que acepta $f(w)$ con cota polinómica $g(f(w))$.

Por lo tanto, M_2 acepta w con cota polinómica puesto que la composición de polinomios es un polinomio.

c.q.d

El concepto de reducibilidad polinómica es una herramienta básica para determinar a cuál de las clases, *P* o *NP*, pertenece un lenguaje. Puede servir como punto de partida para el siguiente razonamiento: si cualquier lenguaje de una determinada clase de complejidad se pudiera reducir a un determinado lenguaje, basta estudiar cómo reconocer ese lenguaje y cuál es la cota temporal de la Máquina de Turing que lo reconoce. Es decir, basta con estudiar cuál es el algoritmo con mejor cota temporal para resolver el problema que representa ese lenguaje y *como todos los problemas de su clase* no superan la cota polinómica en su reducción a él, serán igual de *difíciles* de resolver.

Definición 10.8 *Para cualquier clase $\mathcal{C}$ de lenguajes, un lenguaje L se dice que es $\mathcal{C} - \mathbf{hard}$ ($\mathcal{C}$-díficil ó $\mathcal{C}$-duro) si, $\forall\ L' \in \mathcal{C},\ L' <_p L$. Es decir, todos los lenguajes de $\mathcal{C}$ se reducen a L en tiempo polinómico.*
En particular, L es **NP-hard** *si para cualquier lenguaje $L' \in NP,\ L' <_p L$.*

Definición 10.9 *Si L es $\mathcal{C}$-hard y $L \in \mathcal{C}$, entonces L es $\mathcal{C} - \mathbf{completo}$.*
En particular, si L es **NP-hard** *y L ∈ NP, entonces L es* **NP-completo**.

Si $L_1 <_p L_2$ entonces determinar si $w \in L_1$ no es más díficil que determinar si $f(w) \in L_2$, siendo *f* la función que reduce L_1 a L_2 en tiempo polinómico. Y, en concreto, si se habla de lenguajes $\mathcal{C} - completos$, la reducción polinómica puede aplicarse a todos los lenguajes de la clase $\mathcal{C}$. Por supuesto, si se habla de las clases *NP* o *P*, el resultado es trascendental: sólo con encontrar un lenguaje *NP-completo* reducible en tiempo polinómico a un lenguaje de la clase *P*, se sabría que cualquier otro lenguaje de *NP* también admitiría una reducción polinómica.

Teorema 10.11 (de Lambsmother)
Si L es un lenguaje NP-completo y $L \in P \Rightarrow$ **P = NP**.

Demostración:

Sea cualquier lenguaje $L_1 \in NP \Rightarrow L_1 <_p L$, que es NP-completo.

Como $L \in P \Rightarrow L_1 \in P$.

c.q.d

Este teorema permitiría demostrar *P=NP*, en el caso de encontrar un lenguaje que cumpliera las condiciones. Por lo tanto, el primer paso en este sentido fue buscar un lenguaje NP completo. El primer lenguaje NP completo encontrado fue $\mathcal{L}_{sat}$.

Este lenguaje representa el *Problema de la Satisfactibilidad*:

> "Dado un conjunto de cláusulas booleanas (expresiones booleanas formadas con negadores y disyunciones de constantes y/o variables booleanas), ¿existe un conjunto de valores para las variables que intervienen en las cláusulas que las satisfagan todas?".

Este problema de decisión se asocia al lenguaje

$$\mathcal{L}_{sat} = \{w \in \Sigma^* \mid w \text{ representa un conjunto de cláusulas satifactibles}\},$$

siendo Σ = {'0', '1', ',', '&', '¬', '∨'} asumiendo que la variable x_i se codifica como la cadena '&i', en la que *i* se expresa en binario.

Ejemplo:

$$C_1 = \{x_1 \vee \neg x_2, \neg x_1 \vee \neg x_2 \vee x_3, \neg x_1 \vee x_2 \vee x_3\}$$

	$x_1 \vee \neg x_2$	$\neg x_1 \vee \neg x_2 \vee x_3$	$\neg x_1 \vee x_2 \vee x_3$
000	**1**	**1**	**1**
001	**1**	**1**	**1**
010	*0*		
011	*0*		
100	*1*	*1*	*0*
101	**1**	**1**	**1**
110	*1*	*0*	
111	**1**	**1**	**1**

es un conjunto de cláusulas satifactibles, ya que $x_1 = 0$, $x_2 = 0$, $x_3 = 0$ *ó* $x_1 = 1$, $x_2 = 0$, $x_3 = 1$, *son valores que satisfacen todas las cláusulas. Sin embargo, C2 no lo es:*

$$C_2 = \{\neg x_1, x_1 \vee x_2, x_1 \vee \neg x_2\}$$

	$\neg x_1$	$x_1 \vee x_2$	$x_1 \vee \neg x_2$
00	*1*	*0*	
01	*1*	*1*	*0*
10	*0*		
11	*0*		

Teorema 10.12 (de Cook) *$\mathcal{L}_{sat}$ es un lenguaje NP-completo.*

La demostración se puede encontrar en la bibliografía (recomiendo la demostración que se hace en el libro de Dean Kelley, *"Teoría de Autómatas y Lenguajes Formales"*).

La importancia de esta demostración es que, una vez que se ha demostrado que un lenguaje es NP-completo, se simplifica la búsqueda de otros ya que, para ello, se pueden aplicar los siguientes resultados:

Lema 10.1 *Si L_1 es NP-completo y $L_1 <_p L_2$, entonces L_2 es NP-hard.*

Corolario 10.1 *Si L_1 es NP-completo y $L_1 <_p L_2$ y $L_2 \in NP$, entonces L_2 es NP-completo.*

La idea es que cuántos más problemas NP-completos se conozcan, más posibilidades existen de que haya alguno que cumpla las condiciones del teorema 10.11. En la actualidad se ha demostrado que más de 400 lenguajes son NP-completos, pero aún no se ha podido encontrar un lenguaje que satisfaga las condiciones de dicho teorema.

Bibliografía

[Cook71] Stephen Cook. "The Complexity of Theorem Proving Procedures". *Conference Record of Third Annual ACM Symposium on Theory of Computing*, Shaker Heighs, Ohio, pp. 151–158. 1971.

Si alguien está interesado tengo un revisión de este artículo del propio Stephen Cook, titulado *"The P versus NP Problem"*. Creo que es de 2000.

[Dowe01] Gilles Dowek. "El Infinito y el Universo de los Algoritmos". *Investigación y Ciencia*, Temas 23, pp. 74–76. 2001.

[Gare79] Michael R. Garey , David S. Johnson. "Computers and Intractability. A Guide to the Theory of NP-Completeness". *W.H. Freeman and Company, New York*. 1979.

[Hofs79] Douglas Hofstadter. "Gödel, Escher, Bach: Un Eterno y Grácil Bucle". *Colección Metatemas 14, Tusquets Editores*. 1979.

[Hopc79] John E. Hopcroft, Jeffrey D. Ullman. "Introduction to Automata Theory, Languages and Computation". *Addison-Wesley Publishing Company*. 1979.

[Hopc84] John E. Hopcroft. "Máquinas de Turing". *Investigación y Ciencia*, pp. 8–19. Julio 1984.

[Hopc02] John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman. "Introducción a la Teoría de Autómatas, Lenguajes y Computación". *Addison Wesley*. 2002.

[Johns00] David Johnson. "Challenges for Theorical Computer Science (Draft)". Last Updated, June 2000.

<http://www.research.att.com/~dsj/nsflist.html> [Última visita: 4 de Mayo de 2004]

[Kelley95] Dean Kelley. "Teoría de Autómatas y Lenguajes Formales". *Prentice-Hall Inc.* 1995.

[Lewis81] Harry R. Lewis, Christos Papadimitriou. "Elements of the Theory of Computation". *Prentice-Hall, Inc.* 1981.

[Mahon98] Michael S. Mahoney. "The Structures of Computation". *Proc. of the International Conference on the History of Computing, Heinz Nixdorf Forum*. Paderborn, Germany, 14–16. August 1998.

<http://www.princeton.edu/~mike> sec: Articles on the History of Computing [Última visita: 4 de Mayo de 2004]

[Shall95] Jeffrey Shallit. “A Very Brief History of Computer Science”. *Handout for CS134 course*. University of Waterloo. 1995.

<http://www.math.uwaterloo.ca/~shallit/Courses/134/history.html>

[Última visita: 4 de Mayo de 2004]

[Shall98] Jeffrey Shallit. “The Busy Beaver Problem”. *Handout for CS360 course*. University of Waterloo. 1998.

[Sipser97] Michael Sipser. “Introduction to the Theory of Computation”. *PWS Publishing Company*. 1997.