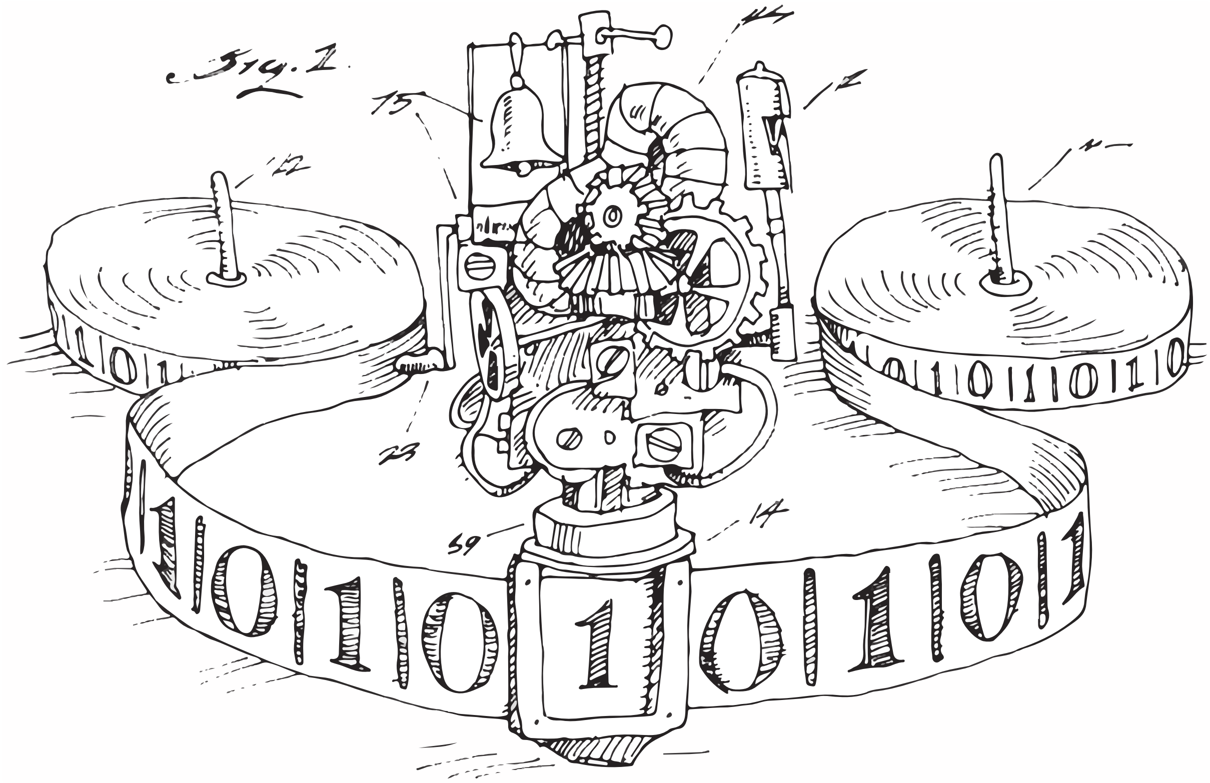
Notas para el curso de

Teoría de Lenguajes

Facultad de Ingeniería
Universidad de la República

Bruno Hernández

Versión 1.3



Índice

0. Definiciones Preliminares **3**

1. Lenguajes Regulares **5**
1.1. Expresiones Regulares . . . 5
1.2. Autómatas Finitos . . . 6
1.2.1. Autómata Finito Determinista . . . 6
1.2.2. Autómata Finito No–Determinista . . . 7
1.2.3. Autómata Finito No–Determinista con ε–transiciones . . . 11
1.2.4. Equivalencia entre Expresiones Regulares y Autómatas Finitos . . . 16
1.2.5. Relación R_M, Teorema de Myhill–Nerode y Minimización . . . 19
1.2.6. Autómatas con Salida . . . 21
1.2.7. Autómata Finito Determinista de Dos Cintas . . . 26
1.3. Propiedades de los Lenguajes Regulares . . . 31
1.3.1. Pumping Lemma para Lenguajes Regulares . . . 31
1.3.2. Contrarrecíproco del Pumping Lemma . . . 33
1.3.3. Propiedades de Clausura . . . 34

2. Lenguajes Libres de Contexto **37**
2.1. Gramáticas Libres de Contexto . . . 37
2.1.1. Introducción . . . 37
2.1.2. Gramáticas Regulares . . . 40
2.1.3. Simplificación . . . 42
2.1.4. Formas Normales . . . 43
2.2. Autómatas Push–Down . . . 44
2.2.1. Introducción . . . 44
2.2.2. Reconocimiento: Estado Final vs Pila Vacía . . . 45
2.2.3. Equivalencia entre Gramáticas Libres de Contexto y Autómatas Push–Down . . . 48
2.3. Propiedades de los Lenguajes Libres de Contexto . . . 49
2.3.1. Pumping Lemma para Lenguajes Libres de Contexto . . . 49
2.3.2. Contrarrecíproco del Pumping Lemma . . . 51
2.3.3. Propiedades de Clausura . . . 53
2.3.4. Lema de Ogden . . . 56

3. Lenguajes Recursivamente Enumerables **61**
3.1. Gramáticas Irrestrictas . . . 61
3.1.1. Caso General . . . 61
3.1.2. Gramáticas Sensibles al Contexto . . . 62
3.2. Máquinas de Turing . . . 63
3.2.1. Caso General . . . 63
3.2.2. Autómata Lineal Acotado . . . 65
3.3. Jerarquía de Chomsky . . . 66
3.4. Una introducción a la Teoría de la Computación . . . 67

4. Algoritmos **71**
4.1. Conversión AFND en AFD . . . 71
4.2. Conversión AFND–ε en AFND . . . 72
4.3. Conversión AFD en RegEx: Análisis de Kleene / Clases de Equivalencia . . . 73
4.4. Minimización de AFD . . . 74
4.5. Conversión Autómata Finito en Gramática Regular . . . 76
4.6. Simplificación de Gramáticas Libres de Contexto . . . 78

Prefacio

Estas notas fueron redactadas en base a mis apuntes del curso de *Teoría de Lenguajes* del año 2017 dictado en la Facultad de Ingeniería de la Universidad de la República (Montevideo, Uruguay) por el profesor Juan José Prada, habiendo elaborado la primera versión durante el año 2018. Procurando mejorar la redacción de algunos párrafos, he consultado el libro recomendado para esta asignatura: *Introduction to Automata Theory, Languages and Computation* de Hopcroft, Motwani y Ullman. Para obtener una visión integral de los temas aquí tratados, sugiero a los interesados también examinarlo. Salvo que se indique lo contrario, las ilustraciones presentes en esta obra son de mi autoría. La imagen de la portada fue tomada de la revista *American Scientist, Vol. 90, No. 2, p. 168.*

En esta versión se han incluido ejemplos básicos de cada concepto presentado, sin embargo – por razones de tiempo – no se han abarcado todos los ejemplos presentados en las clases desarrolladas durante el curso, los cuales contribuyen a la didáctica de los temas expuestos. Por lo tanto, **estas notas no pretenden sustituir la asistencia a clase**. Para tener un dominio efectivo de esta asignatura, aconsejo fuertemente involucrarse en todas las clases ya sean teóricas o prácticas, así como realizar los ejercicios propuestos por los docentes. Mi intención a futuro es incluir (junto con estas notas) ejemplos interactivos, los cuales considero que ayudarían a visualizar mejor ciertos conceptos y algoritmos.

Agradezco a aquellos que enriquecieron este trabajo con sus sugerencias durante la elaboración de la "Versión 1.0" ("de prueba"), así como a quienes deseen brindar cualquier comentario en pos de continuar mejorándolo, los cuales serán bienvenidos en `bhernandez@fing.edu.uy`. En `www.fing.edu.uy/~bhernandez` se publicará una lista con las erratas detectadas en la presente versión.

Espero que estas notas le sean útiles en su entendimiento de esta asignatura.

Bruno Hernández

0. Definiciones Preliminares

Definición 0.1 Usaremos la palabra ***lenguaje*** como sinónimo de la palabra **conjunto**. Llamaremos ***tiras*** (*strings, cadenas, palabras,* etc.) a los elementos que componen un lenguaje, las cuales consisten en secuencias de *caracteres* de determinado ***alfabeto*** Σ. A la ***tira vacía*** la representaremos con el símbolo ε.

Definición 0.2 Sean L_1 y L_2 dos lenguajes, con $v \in L_1$ y $w \in L_2$. La ***concatenación de v con w*** consiste en crear una nueva tira s, colocando w justo al final de v

$$s = vw$$

La **tira vacía** ε funciona como ***elemento neutro*** en la concatenación de palabras, de modo que $w\varepsilon = \varepsilon w = w$, para cualquier palabra w.

Ejemplo 0.1 *Si* $\boldsymbol{v = aba}$ *y* $\boldsymbol{w = 010}$*, la concatenación de v con w resulta en la tira* $\boldsymbol{s = aba010}$*.*

Ejemplo 0.2 $aba = \varepsilon aba = \varepsilon\varepsilon aba = \varepsilon aba\varepsilon = \varepsilon ab\varepsilon a = \varepsilon a\varepsilon\varepsilon b\varepsilon\varepsilon\varepsilon a\varepsilon$*. Etcétera.*

Definición 0.3 Dados dos lenguajes A y B, definimos un nuevo lenguaje llamado la ***concatenación de A con B*** como

$$A \cdot B = \{ab \mid a \in A \ \wedge \ b \in B\}$$

El **conjunto vacío** $\varnothing$ funciona como ***elemento absorbente*** en la concatenación de lenguajes, de modo que $A \cdot \varnothing = \varnothing \cdot A = \varnothing$, para cualquier lenguaje A. También es común omitir el $\cdot$ al referirse a esta operación.

Ejemplo 0.3 *Si* $A = \{x, y\}$ *y* $B = \{0, 1\}$*, entonces* $A \cdot B = \{x0, x1, y0, y1\}$*. Observar que la* ***concatenación no es conmutativa en general****, ya que* $B \cdot A = \{0x, 0y, 1x, 1y\}$*.*

Observación: Notar que $\{\varepsilon\} \neq \varnothing$, y también que $A \cdot \{\varepsilon\} = \{\varepsilon\} \cdot A = A$, para cualquier lenguaje A.

Ejemplo 0.4 *Si* $A = \{x, y\}$ *y* $B = \{\varepsilon, 0, 1\}$*, entonces* $A \cdot B = \{x, x0, x1, y, y0, y1\}$*.*

Definición 0.4 Sea A un lenguaje. Las ***potencias de A*** se definen de la siguiente manera recursiva:

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

Ejemplo 0.5 *Si* $A = \{ja\}$, *entonces*

$$\begin{aligned} A^3 &= A^2 \cdot A \\ &= A^1 \cdot A \cdot A \\ &= A^0 \cdot A \cdot A \cdot A \\ &= \{\varepsilon\} \cdot A \cdot A \cdot A \\ &= A \cdot A \cdot A \\ &= \{ja\} \cdot \{ja\} \cdot \{ja\} \\ &= \{jaja\} \cdot \{ja\} \\ &= \{jajaja\} \end{aligned}$$

Definición 0.5 Dado un lenguaje A cualquiera, la ***Clausura de Kleene*** de A es el conjunto

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

También podemos definir la Clausura de Kleene de manera recursiva:

$$\begin{cases} \varepsilon \in A^* \\ \textbf{Si} \quad a \in A \ \wedge \ x \in A^* \quad \textbf{entonces} \quad xa \in A^* \end{cases}$$

Ejemplo 0.6 *Si* $A = \{ja\}$, *entonces*

$$A^* = \{\varepsilon\} \cup \{ja\} \cup \{jaja\} \cup \{jajaja\} \cup \cdots = \{\varepsilon, ja, jaja, jajaja, \cdots\}$$

Definición 0.6 Dado un lenguaje A cualquiera, la ***Clausura Positiva*** de A es el conjunto

$$A^+ = \bigcup_{i=1}^{\infty} A^i$$

Ejemplo 0.7 *Si* $A = \{ja\}$, *entonces*

$$A^+ = \{ja\} \cup \{jaja\} \cup \{jajaja\} \cup \cdots = \{ja, jaja, jajaja, \cdots\}$$

Definición 0.7 Dado un lenguaje L definido sobre un alfabeto Σ, definimos la ***relación binaria*** $\boldsymbol{R_L}$ sobre Σ^*, de forma tal que – dados $x, y \in \Sigma^*$ – se tiene que $x \ R_L \ y$ si

$$\forall z \in \Sigma^* : xz \in L \iff yz \in L$$

Observación: R_L es una **relación de equivalencia**, es decir, cumple las propiedades reflexiva, simétrica y transitiva. Esto implica que R_L induce una **partición** de Σ^* en **clases de equivalencia**.

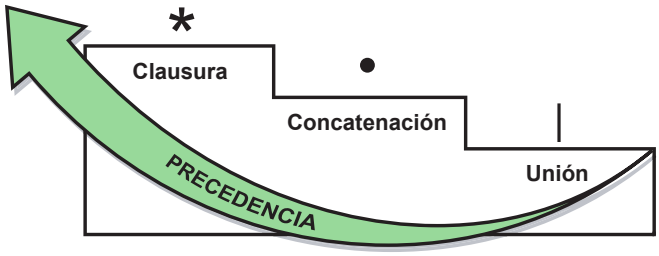
1. Lenguajes Regulares

1.1. Expresiones Regulares

Definición 1.1 Una ***expresión regular*** es una notación que sirve para representar a un lenguaje regular, y se define formalmente a partir de las siguientes cláusulas inductivas:

- **Casos base:**
  - $\varnothing$ es la expresión regular para $\varnothing$
  - ε es la expresión regular para $\{\varepsilon\}$
  - a es la expresión regular para $\{a\}$
- **Casos inductivos:** Si $L_1 = \mathscr{L}(r_1)$ y $L_2 = \mathscr{L}(r_2)$, entonces:
  - $r_1 | r_2$ es una expresión regular para $L_1 \cup L_2$
  - $r_1 \cdot r_2$ es una expresión regular para $L_1 \cdot L_2$
  - $r_1{}^*$ es una expresión regular para $L_1{}^*$

Observación: Los operadores tienen **precedencia**:



Por lo tanto, es muy común extender la notación de las expresiones regulares incorporándoles los paréntesis para poder interpretarlas correctamente.

Ejemplo 1.1 *Ejemplos varios para un alfabeto* $\Sigma = \{0,1\}$*:*

$$A = \{0,1\} \Rightarrow A = \{0\} \cup \{1\} \Rightarrow A = \mathscr{L}(0|1)$$

$$B = \{0,10\} \Rightarrow B = \{0\} \cup \{10\} \Rightarrow B = \{0\} \cup \{1\} \cdot \{0\} \Rightarrow B = \mathscr{L}(1|1\cdot 0)$$

$$A^* = \mathscr{L}\left((0|1)^*\right) \quad y \quad A^+ = \mathscr{L}\left((0|1)\cdot(0|1)^*\right)$$

$$C = \{w \in (0|1)^* \ / \ w \text{ tiene solo dos ceros}\} \Rightarrow C = \mathscr{L}(1^*\cdot 0\cdot 1^*\cdot 0\cdot 1^*)$$

$$D = \{w \in (0|1)^* \ / \ w \text{ termina con dos ceros}\} \Rightarrow D = \mathscr{L}\left((0|1)^*\cdot 0\cdot 0\right)$$

Definición 1.2 L es un ***lenguaje regular*** si existe alguna expresión regular que lo define.

1.2. Autómatas Finitos

Al igual que las expresiones regulares, los ***autómatas finitos*** son modelos que permiten describir a los lenguajes regulares. A continuación veremos tres variantes de este modelo, y más adelante veremos la equivalencia entre todos ellos.

1.2.1. Autómata Finito Determinista

Definición 1.3 Un ***autómata finito determinista (AFD)*** es una 5-tupla $M = (Q, \Sigma, \delta, q_0, F)$ donde:

- Q es el conjunto de estados
- Σ es el alfabeto de la entrada
- $\delta : Q \times \Sigma \to Q$ es la función de transición de estados
- $q_0 \in Q$ es el estado inicial
- $F \subseteq Q$ es el conjunto de estados de aceptación

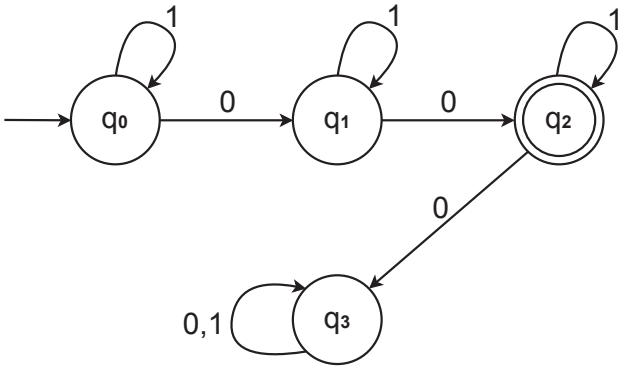
Como se observa, la función δ permite procesar de a un símbolo de Σ. Es por esto que necesitamos definir una función $\hat{\delta} : Q \times \Sigma^* \to Q$ que nos permita procesar toda una tira de Σ^* por completo:

$$\forall q \in Q \ : \ \hat{\delta}(q, \varepsilon) = q \tag{1.1}$$

$$\forall q \in Q \ , \ \forall a \in \Sigma \ , \ \forall x \in \Sigma^* \ : \ \hat{\delta}(q, xa) = \delta\left(\hat{\delta}(q, x), a\right) \tag{1.2}$$

Definición 1.4 Dado un AFD $M = (Q, \Sigma, \delta, q_0, F)$, el ***lenguaje aceptado por M*** es $\mathscr{L}(M) = \left\{x \in \Sigma^* \ / \ \hat{\delta}(q_0, x) \in F\right\}$. Es decir, $\mathscr{L}(M)$ es el conjunto de tiras de Σ^* tales que, partiendo desde q_0, M llega a un estado de aceptación tras procesarlas.

Ejemplo 1.2 *Un AFD puede ser representado mediante un* ***diagrama de estados*** *o a través de una* ***tabla de transiciones****. A continuación presentamos un AFD en ambas notaciones:*



(a) Diagrama de estados

δ	0	1
$\longrightarrow q_0$	q_1	q_0
q_1	q_2	q_1
$\textcircled{q_2}$	q_3	q_2
q_3	q_3	q_3

(b) Tabla de transiciones

Un AFD para el $\mathscr{L}(1^*\,0\,1^*\,0\,1^*)$

En la notación colocamos una flecha apuntando al estado inicial, y dibujamos un círculo interior para indicar que un estado es de aceptación. Representamos las transiciones con una flecha de un estado a otro, donde el rótulo indica con qué símbolo de Σ se produce la transición. Para evitar sobrecargar el diagrama con muchas flechas, se puede poner más de un rótulo sobre una misma flecha (como en q_3).

Observación: En el Ejemplo 1.2, un estado como q_3 se denomina ***estado pozo***. Un *estado pozo* es aquel que tiene una transición saliente hacia sí mismo para cada uno de los símbolos de Σ, y además no es un estado de aceptación. Visto de otra manera, es un estado de no-aceptación el cual una vez que se entra no se sale más, independientemente de lo que quede por consumir de la entrada.

Muchas veces **no** dibujaremos estados pozo así como tampoco sus transiciones entrantes, para evitar sobrecargar el diagrama. A causa de esto, tendremos dibujada una función δ parcialmente definida ("incompleta"). A efectos de tener una función δ totalmente definida, se dará por hecho que **cualquier transición no dibujada está presente y se dirige a un estado pozo**.

Definición 1.5 L es un ***lenguaje regular*** si es aceptado por algún AFD.

1.2.2. Autómata Finito No–Determinista

Como vimos en la Subsubsección 1.2.1, el conjunto de llegada (codominio) de la función δ de un AFD es su conjunto de estados. Es decir, que estando en un estado y leyendo un símbolo en la entrada, un AFD tiene **exactamente una** opción a donde dirigirse. El próximo estado queda *completamente determinado*.

En los autómatas finitos no–deterministas, estando en un estado y leyendo un símbolo en la entrada, se puede tener **más de una** (o incluso ninguna) opción a donde dirigirse. Es decir, el resultado de la función δ es un **conjunto de estados**, del cual el autómata "elegirá" uno para moverse. El próximo estado *no está determinado* a priori.

Definición 1.6 Un ***autómata finito no–determinista (AFND)*** es una 5-tupla $M = (Q, \Sigma, \delta, q_0, F)$ en la que:

- Q es el conjunto de estados
- Σ es el alfabeto de la entrada
- $\delta : Q \times \Sigma \to 2^Q$ es la función de transición de estados
- $q_0 \in Q$ es el estado inicial
- $F \subseteq Q$ es el conjunto de estados de aceptación

Nuevamente precisamos extender la capacidad de la función δ para poder procesar tiras de Σ^* por completo, con lo cual tendremos $\hat{\delta} : Q \times \Sigma^* \to 2^Q$ definida de la siguiente manera:

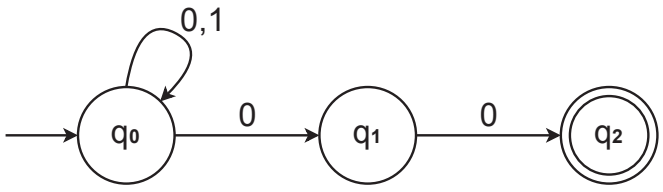
$$\forall q \in Q \ : \ \hat{\delta}(q, \varepsilon) = \{q\} \tag{1.3}$$

$$\forall q \in Q \ , \ \forall a \in \Sigma \ , \ \forall x \in \Sigma^* \ : \ \hat{\delta}(q, xa) = \tilde{\delta}\left(\hat{\delta}(q, x), a\right) \tag{1.4}$$

donde $\tilde{\delta} : 2^Q \times \Sigma \to 2^Q \ / \ \tilde{\delta}(P, a) = \bigcup_{q \in P} \delta(q, a)$.

Definición 1.7 Dado un AFND $M = (Q, \Sigma, \delta, q_0, F)$, el ***lenguaje aceptado por M*** es $\mathscr{L}(M) = \left\{x \in \Sigma^* \ / \ \hat{\delta}(q_0, x) \cap F \neq \varnothing\right\}$. Visto desde otro ángulo, $\mathscr{L}(M)$ es el conjunto de tiras de Σ^* tales que, partiendo desde q_0, existe una secuencia de decisiones durante su procesamiento que conducen a M hacia un estado de aceptación.

Ejemplo 1.3 *Como en los AFD, tenemos dos variantes para representar un AFND, el diagrama de estados o la tabla de transiciones:*



(a) Diagrama de estados

δ	0	1
$\longrightarrow q_0$	$\{ q_0 , q_1 \}$	$\{ q_0 \}$
q_1	$\{ q_2 \}$	$\varnothing$
$\textcircled{q_2}$	$\varnothing$	$\varnothing$

(b) Tabla de transiciones

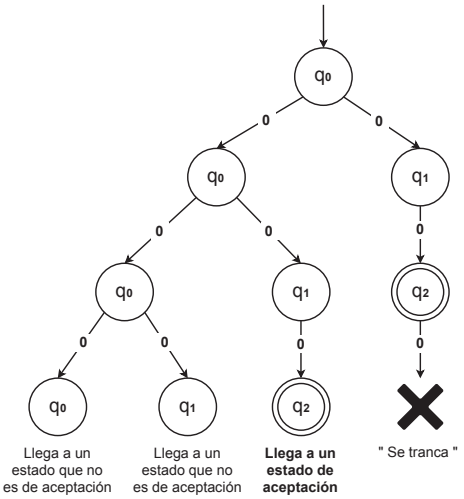
Un AFND para el $\mathscr{L}((0|1)^* \ 0 \ 0)$

Si quisiéramos saber si la tira $w = 000$ pertenece al lenguaje, debemos encontrar una secuencia de decisiones que conduzcan a un estado final:

I) Empezamos en q_0 y en la entrada tenemos el primer 0, entonces podemos elegir entre permanecer en q_0 o ir a q_1. Elegimos ir a q_1 y avanzamos en la tira. Ahora estamos en q_1 y en la entrada tenemos el segundo 0, con lo cual solo podemos ir a q_2. Nos movemos a q_2 y avanzamos en la tira. Ahora estamos en q_2 – que es un estado de aceptación – pero aún queda un 0 por consumir en la entrada. Con 0 (o bien con 1) en q_2 el autómata "se tranca" ya que no tiene a donde ir (lo mismo le sucede en q_1 con 1), con lo cual estas decisiones nos llevan a no aceptar la tira.

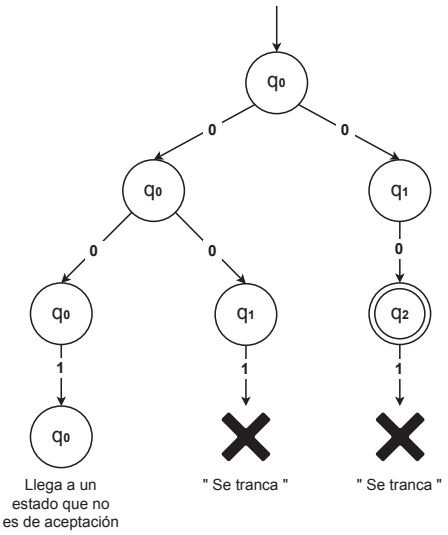
II) Empezamos en q_0 y en la entrada tenemos un 0, entonces podemos elegir entre permanecer en q_0 o ir a q_1. Decidimos quedarnos en q_0 y avanzamos en la tira. Ahora estamos en q_0 y en la entrada tenemos otro 0, con lo cual podemos elegir entre seguir en q_0 o ir a q_1. Esta vez elegimos ir a q_1 y avanzamos en la tira. Ahora estamos en q_1 y nos queda un 0 por consumir en la entrada. Aquí solo podemos movernos hacia q_2, entonces lo hacemos y avanzamos en la tira. Hemos llegado a un estado de aceptación y ya no quedan símbolos por consumir en la entrada, con lo cual estas decisiones nos llevan a aceptar la tira. ***Conclusión:*** *$w \in \mathscr{L}\left((0|1)^* \ 0 \ 0\right)$.*

Una buena opción para visualizar todas las posibles alternativas es dibujar un árbol de decisiones:



Árbol de decisiones para $w = 000$

Si quisiéramos saber si la tira $v = 001$ *pertenece al lenguaje, de la misma manera que con el caso anterior tenemos que ver todas las maneras posibles de ejecutar el autómata. Con el siguiente árbol de decisiones podemos probar que* $v \notin \mathscr{L}\left((0|1)^* \, 0 \, 0\right)$ *exhibiendo que todas las posibles alternativas no conducen a la aceptación:*



Árbol de decisiones para $v = 001$

Definición 1.8 L es un ***lenguaje regular*** si es aceptado por algún AFND.

Teorema 1.1 Conversión de AFND en AFD

Todos los AFND pueden ser convertidos a un AFD equivalente que compute exactamente el mismo lenguaje. A continuación mostramos los fundamentos del Algoritmo 4.1

que expresa el proceso de conversión de forma más "algorítmica".

Dado un AFND $M = (Q, \Sigma, \delta, q_0, F) \ / \ L = \mathscr{L}(M)$, crearemos un AFD $M' = (Q', \Sigma', \delta', q_0', F') \ / \ L = \mathscr{L}(M')$ según estas reglas:

1) $Q' \subseteq 2^Q$

2) $\Sigma' = \Sigma$

3) $q_0' = \{q_0\} = [q_0]$

4) $F' = \{[S] \in Q' \ / \ S \in Q \ \wedge \ S \cap F \neq \varnothing\}$

5) $\delta'\big([q_i, \cdots, q_j], a\big) = [p_r, \cdots, p_t] \iff \tilde{\delta}\big(\{q_i, \cdots, q_j\}, a\big) = \{p_r, \cdots, p_t\}$

Demostración de la correctitud: Haremos inducción sobre $|x|$, el largo de x, para probar que: $\hat{\delta}'\big([q_0], x\big) = [r_i, \cdots, r_k] \iff \hat{\delta}(q_0, x) = \{r_i, \cdots, r_k\}$

- **Paso base:** $|x| = 0$

$$\begin{aligned}
\hat{\delta}'\big([q_0], \varepsilon\big) &= [q_0] && \text{// Def. } \hat{\delta} \text{ para AFD en Ecuación 1.1} \\
&= \{q_0\} && \text{// Regla 3)} \\
&= \hat{\delta}(q_0, \varepsilon) && \text{// Def. } \hat{\delta} \text{ para AFND en Ecuación 1.3}
\end{aligned}$$

- **Paso inductivo:**

  H) La propiedad vale para $|x| \leq h$

  T) La propiedad vale para $|x| = h + 1$

  Dem.:

  Tenemos un x con $|x| = h+1$, entonces lo primero que haremos es renombrar $x = wa$ $(a \in \Sigma)$ para poner en juego a un w con $|w| = h$ que nos permita aplicar la hipótesis de inducción. Entonces nos concentraremos en probar

$$\hat{\delta}'\big([q_0], wa\big) = [r_i, \cdots, r_k] \iff \hat{\delta}(q_0, wa) = \{r_i, \cdots, r_k\}$$

  sabiendo que

$$\hat{\delta}'\big([q_0], w\big) = [p_j, \cdots, p_t] \iff \hat{\delta}(q_0, w) = \{p_j, \cdots, p_t\}$$

$$\begin{aligned}
\hat{\delta}'\big([q_0], wa\big) &= \delta'\Big(\hat{\delta}'\big([q_0], w\big), a\Big) && \text{// Def. } \hat{\delta} \text{ para AFD en Ecuación 1.2} \\
&= \delta'\big([p_j, \cdots, p_t], a\big) && \text{// H.I.: } \hat{\delta}'\big([q_0], w\big) = [p_j, \cdots, p_t] \\
&= [r_i, \cdots, r_k] && \text{// Def. } \delta \text{ para AFD en Definición 1.3} \\
&\iff \tilde{\delta}\big(\{p_j, \cdots, p_t\}, a\big) = \{r_i, \cdots, r_k\} && \text{// Regla 5)} \\
&\iff \tilde{\delta}\big(\hat{\delta}(q_0, w), a\big) = \{r_i, \cdots, r_k\} && \text{// H.I.: } \{p_j, \cdots, p_t\} = \hat{\delta}(q_0, w) \\
&\iff \hat{\delta}(q_0, wa) = \{r_i, \cdots, r_k\} && \text{// Def. } \hat{\delta} \text{ para AFND en Ecuación 1.4}
\end{aligned}$$

■

Por último observamos que M y M' aceptan si y sólo si $\hat{\delta}(q_0, x)$ o $\hat{\delta}'\big([q_0], x\big)$, respectivamente, contienen un estado de aceptación, completando así la prueba de que $\mathscr{L}(M) = \mathscr{L}(M')$.

Teorema 1.2 Equivalencia entre AFND y AFD
Con el Teorema 1.1 vimos como un AFND puede ser transformado en un AFD equivalente. Inversamente, podemos interpretar que un AFD es un AFND el cual tiene la particularidad de que $\forall q \in Q \ , \ \forall a \in \Sigma \ : \ |\delta(q, a)| = 1$.

1.2.3. Autómata Finito No–Determinista con ε–transiciones

Ahora veremos una extensión de los AFND que incorpora al modelo que ya disponemos la característica de las ε–*transiciones*.

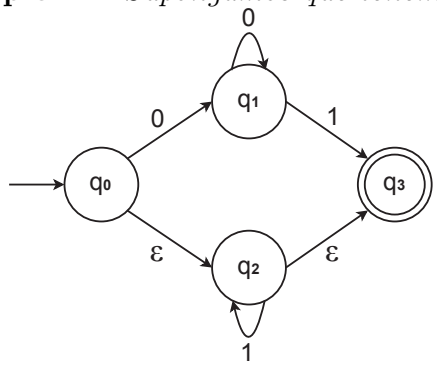
Definición 1.9 Una ***ε–transición*** entre dos estados es una transición que, cuando el autómata se encuentra en el estado de partida de la misma, esta puede ocurrir espontáneamente **sin consumir la entrada**.

Es de nuestro interés saber "qué tan lejos podemos llegar" a partir de ciertos estados, aplicando nada más que ε–transiciones. Por eso necesitamos la siguiente:

Definición 1.10 Dado $q \in Q$, la ***ε–clausura***(q) es el conjunto de estados a los que se puede llegar partiendo desde q, a través de cero o más arcos rotulados exclusivamente por ε. Más en general, si $P \subseteq Q$ tendremos que la ***ε–clausura***$(P) = \bigcup_{q \in P} \varepsilon\text{–}clausura(q)$. Para abreviar la denotaremos como ***ε-cℓ***$(\cdot)$.

Observación: Cualquier estado está contenido en su ε–clausura $\big(\forall q \in Q \ : \ q \in \varepsilon\text{-}c\ell(q)\big)$.

Ejemplo 1.4 *Supongamos que tenemos el siguiente AFND–ε:*



Ejemplo de un AFND–ε

Entonces:

$\varepsilon\text{-}c\ell(q_0) = \{q_0, q_2, q_3\}$ $\quad \varepsilon\text{-}c\ell(q_2) = \{q_2, q_3\}$
$\varepsilon\text{-}c\ell(q_1) = \{q_1\}$ $\quad \varepsilon\text{-}c\ell(q_3) = \{q_3\}$

$$\varepsilon\text{-}c\ell\big(\{q_1, q_3\}\big) = \{q_1, q_3\}$$
$$\varepsilon\text{-}c\ell\big(\{q_0, q_1\}\big) = \{q_0, q_1, q_2, q_3\}$$

Definición 1.11 Un ***autómata finito no–determinista con ε–transiciones (AFND–ε)*** es una 5-tupla $M = (Q, \Sigma, \delta, q_0, F)$ donde:

- Q es el conjunto de estados
- Σ es el alfabeto de la entrada
- $\delta : Q \times \big(\Sigma \cup \{\varepsilon\}\big) \to 2^Q$ es la función de transición de estados
- $q_0 \in Q$ es el estado inicial
- $F \subseteq Q$ es el conjunto de estados de aceptación

Al igual que en los otros casos, precisamos extender el poder de la función δ para que sea capaz de computar tiras completas de Σ^*, con lo cual tenemos $\hat{\delta} : Q \times \Sigma^* \to 2^Q$ definida de la siguiente manera:

$$\forall q \in Q \ : \ \hat{\delta}(q, \varepsilon) = \varepsilon\text{-}c\ell(q) \tag{1.5}$$

$$\forall q \in Q \ , \ \forall a \in \Sigma \ , \ \forall x \in \Sigma^* \ : \ \hat{\delta}(q, xa) = \varepsilon\text{-}c\ell\Big(\tilde{\delta}\big(\hat{\delta}(q, x), a\big)\Big) \tag{1.6}$$

donde $\tilde{\delta} : 2^Q \times \Sigma \to 2^Q \ / \ \tilde{\delta}(P, a) = \bigcup_{q \in P} \delta(q, a)$ (ídem Definición 1.6, pero con la δ aquí definida).

Definición 1.12 Dado un AFND–ε $M = (Q, \Sigma, \delta, q_0, F)$, el ***lenguaje aceptado por M*** es $\mathscr{L}(M) = \big\{x \in \Sigma^* \ / \ \hat{\delta}(q_0, x) \cap F \neq \varnothing\big\}$ (ídem Definición 1.6, pero con la $\hat{\delta}$ aquí definida).

Definición 1.13 L es un ***lenguaje regular*** si es aceptado por algún AFND–ε.

Teorema 1.3 Propiedades destacadas de la ε–clausura

1) **Unión:** Si S y T son dos conjuntos de estados de Q, se tiene que

$$\varepsilon\text{-}c\ell(S \cup T) = \varepsilon\text{-}c\ell(S) \cup \varepsilon\text{-}c\ell(T)$$

Luego, si tenemos una colección de conjuntos de estados S_i indexada por algún I finito:

$$\varepsilon\text{-}c\ell\left(\bigcup_{i \in I} S_i\right) = \bigcup_{i \in I} \varepsilon\text{-}c\ell(S_i)$$

2) **Idempotencia:** La aplicación reiterada de la ε–clausura no produce nada nuevo:

$$\forall q \in Q \ : \ \varepsilon\text{-}c\ell\big(\varepsilon\text{-}c\ell(q)\big) = \varepsilon\text{-}c\ell(q)$$

Más aún, si $S \subseteq Q$ se tiene que $\varepsilon\text{-}c\ell\big(\varepsilon\text{-}c\ell(S)\big) = \varepsilon\text{-}c\ell(S)$.

3) **Cerradura:** Si $S \subseteq Q$ y $r \in \varepsilon\text{-}c\ell(S) \ \Rightarrow \ \varepsilon\text{-}c\ell(r) \subseteq \varepsilon\text{-}c\ell(S)$.

Demostraciones:

1) $\varepsilon\text{-}cl(S \cup T) = \bigcup_{q \in (S \cup T)} \varepsilon\text{-}cl(q)$

$= \left(\bigcup_{q \in S} \varepsilon\text{-}cl(q)\right) \cup \left(\bigcup_{q \in T} \varepsilon\text{-}cl(q)\right)$

$= \varepsilon\text{-}cl(S) \cup \varepsilon\text{-}cl(T)$

$\varepsilon\text{-}cl\left(\bigcup_{i \in I} S_i\right) = \varepsilon\text{-}cl\big(S_{i_1} \cup (S_{i_2} \cup \cdots \cup S_{i_n})\big)$

$= \varepsilon\text{-}cl(S_{i_1}) \cup \varepsilon\text{-}cl(S_{i_2} \cup \cdots \cup S_{i_n})$

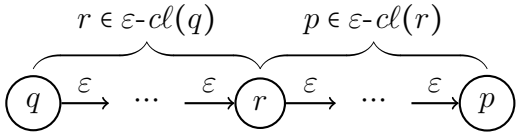
$\vdots$

$= \varepsilon\text{-}cl(S_{i_1}) \cup \cdots \cup \varepsilon\text{-}cl(S_{i_n})$

$= \bigcup_{i \in I} \varepsilon\text{-}cl(S_i)$

2) $\varepsilon\text{-}cl\big(\varepsilon\text{-}cl(q)\big) \subseteq \varepsilon\text{-}cl(q)$

Sea $p \in \varepsilon\text{-}cl\big(\varepsilon\text{-}cl(q)\big) = \bigcup_{r \in \varepsilon\text{-}cl(q)} \varepsilon\text{-}cl(r) \Rightarrow \exists\, r \in \varepsilon\text{-}cl(q) \;/\; p \in \varepsilon\text{-}cl(r)$. Entonces lo que tenemos es algo como esto:



Por lo tanto, se puede llegar desde q a p a través de ε–transiciones $\Rightarrow p \in \varepsilon\text{-}cl(q)$. Luego, como p es genérico, $\varepsilon\text{-}cl\big(\varepsilon\text{-}cl(q)\big) \subseteq \varepsilon\text{-}cl(q)$.

$\varepsilon\text{-}cl(q) \subseteq \varepsilon\text{-}cl\big(\varepsilon\text{-}cl(q)\big)$

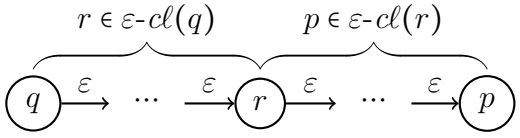
Visto que el propio $q \in \varepsilon\text{-}cl(q)$, sucede que $\varepsilon\text{-}cl(q) \subseteq \bigcup_{r \in \varepsilon\text{-}cl(q)} \varepsilon\text{-}cl(r) = \varepsilon\text{-}cl\big(\varepsilon\text{-}cl(q)\big)$.

Entonces, si $p \in \varepsilon\text{-}cl(q)$, la anterior inclusión implica que $p \in \varepsilon\text{-}cl\big(\varepsilon\text{-}cl(q)\big)$. Luego, como p es genérico, $\varepsilon\text{-}cl(q) \subseteq \varepsilon\text{-}cl\big(\varepsilon\text{-}cl(q)\big)$.

Por lo tanto, habiendo probado la inclusión en ambos sentidos, se tiene que ambos conjuntos son iguales. Luego, aplicando la propiedad de Unión y la que acabamos de demostrar, podemos establecer que:

$$\varepsilon\text{-}cl\big(\varepsilon\text{-}cl(S)\big) = \varepsilon\text{-}cl\left(\bigcup_{q \in S} \varepsilon\text{-}cl(q)\right) = \bigcup_{q \in S} \varepsilon\text{-}cl\big(\varepsilon\text{-}cl(q)\big) = \bigcup_{q \in S} \varepsilon\text{-}cl(q) = \varepsilon\text{-}cl(S)$$

3) Sea $p \in \varepsilon\text{-}cl(r)$. Por hipótesis $r \in \varepsilon\text{-}cl(S) = \bigcup_{q \in S} \varepsilon\text{-}cl(q) \Rightarrow \exists\, q \in S \;/\; r \in \varepsilon\text{-}cl(q)$. Entonces lo que tenemos es algo como esto:



Por lo tanto, se puede llegar desde q a p a través de ε–transiciones $\Rightarrow p \in \varepsilon\text{-}cl(q)$. Así que $\exists\, q \in S \;/\; p \in \varepsilon\text{-}cl(q) \Rightarrow p \in \bigcup_{q \in S} \varepsilon\text{-}cl(q) = \varepsilon\text{-}cl(S)$. Luego, como p es genérico, $\varepsilon\text{-}cl(r) \subseteq \varepsilon\text{-}cl(S)$.

■

Teorema 1.4 Conversión de AFND–ε en AFND

De manera similar como en el Teorema 1.1, los AFND–ε pueden ser convertidos a un AFND equivalente que compute exactamente el mismo lenguaje. A continuación mostramos los fundamentos del Algoritmo 4.2 que expresa el proceso de conversión de forma más "algorítmica".

Dado un AFND–ε $M = (Q, \Sigma, \delta, q_0, F) \;/\; L = \mathscr{L}(M)$, crearemos un AFND $M' = (Q', \Sigma', \delta', q_0', F') \;/\; L = \mathscr{L}(M')$ siguiendo estas reglas:

1) $Q' = Q$ †

2) $\Sigma' = \Sigma$

3) $q_0' = q_0$

4) $F' = \begin{cases} F \cup \{q_0\} & \text{si } \varepsilon\text{-}c\ell(q_0) \cap F \neq \varnothing \\ F & \text{si no} \end{cases}$

5) $\forall q \in Q \;,\; \forall a \in \Sigma \;:\; \delta'(q,a) = \varepsilon\text{-}c\ell\Big(\tilde{\delta}\big(\varepsilon\text{-}c\ell(q), a\big)\Big)$

† A priori se mantiene el conjunto de estados, pero luego de aplicar la Regla 5) podría pasar que queden estados sin transiciones entrantes (o sea, inaccesibles). En tal caso, con ánimos de "depurar" el autómata, uno iterativamente podría quitar esos estados así como las transiciones (salientes) asociadas a ellos.

Demostración de la correctitud: Haremos inducción sobre $|x|$, el largo de x, para probar que: $\hat{\delta}'(q_0', x) = \hat{\delta}(q_0, x)$.

- **Paso base:** $|x| = 1$

$$
\begin{aligned}
\hat{\delta}'(q_0', a) &= \hat{\delta}'(q_0, a) && \text{// Regla 3)} \\
&= \tilde{\delta}'\big(\hat{\delta}'(q_0, \varepsilon), a\big) && \text{// Def. } \hat{\delta} \text{ para AFND en Ecuación 1.4} \\
&= \tilde{\delta}'(\{q_0\}, a) && \text{// Def. } \hat{\delta} \text{ para AFND en Ecuación 1.3} \\
&= \bigcup_{q \in \{q_0\}} \delta'(q, a) && \text{// Def. } \tilde{\delta} \text{ en Definición 1.6} \\
&= \delta'(q_0, a) && \text{// Resolver unión} \\
&= \varepsilon\text{-}c\ell\Big(\tilde{\delta}\big(\varepsilon\text{-}c\ell(q_0), a\big)\Big) && \text{// Regla 5)} \\
&= \varepsilon\text{-}c\ell\Big(\tilde{\delta}\big(\hat{\delta}(q_0, \varepsilon), a\big)\Big) && \text{// Def. } \hat{\delta} \text{ para AFND–}\varepsilon \text{ en Ecuación 1.5} \\
&= \hat{\delta}(q_0, a) && \text{// Def. } \hat{\delta} \text{ para AFND–}\varepsilon \text{ en Ecuación 1.6}
\end{aligned}
$$

- **Paso inductivo:**

  H) La propiedad vale para $|x| \leq h$

  T) La propiedad vale para $|x| = h + 1$

  Dem.:

Tenemos un x con $|x| = h+1$, entonces lo primero que haremos es renombrar $x = wa$ ($a \in \Sigma$) para poner en juego a un w con $|w| = h$ que nos permita aplicar la hipótesis de inducción. Entonces nos concentraremos en probar

$$\hat{\delta}'(q_0{}', wa) = \hat{\delta}(q_0, wa)$$

asumiendo que se cumple $\hat{\delta}'(q_0{}', w) = \hat{\delta}(q_0, w)$.

$$\begin{aligned}
\hat{\delta}'(q_0{}', wa) &= \tilde{\delta}'\big(\hat{\delta}'(q_0{}', w), a\big) && \text{// Def. } \hat{\delta} \text{ para AFND en Ecuación 1.4}\\
&= \tilde{\delta}'\big(\hat{\delta}(q_0, w), a\big) && \text{// Hipótesis de inducción}\\
&= \tilde{\delta}'(P, a) && \text{// Renombrar } P = \hat{\delta}(q_0, w)\\
&= \bigcup_{q\in P} \delta'(q, a) && \text{// Def. } \tilde{\delta} \text{ en Definición 1.6}\\
&= \bigcup_{q\in P} \varepsilon\text{-}c\ell\Big(\tilde{\delta}\big(\varepsilon\text{-}c\ell(q), a\big)\Big) && \text{// Regla 5)}\\
&= \bigcup_{q\in P} \varepsilon\text{-}c\ell\Bigg(\bigcup_{p\in\varepsilon\text{-}c\ell(q)} \delta(p, a)\Bigg) && \text{// Def. } \tilde{\delta} \text{ en Definición 1.11}\\
&= \bigcup_{q\in P} \varepsilon\text{-}c\ell\big(\delta(q, a)\big) && \text{// Doble inclusión } {}^{\dagger}\\
&= \varepsilon\text{-}c\ell\Bigg(\bigcup_{q\in P} \delta(q, a)\Bigg) && \text{// Propiedad de Unión}\\
&= \varepsilon\text{-}c\ell\Big(\tilde{\delta}(P, a)\Big) && \text{// Def. } \tilde{\delta} \text{ en Definición 1.11}\\
&= \varepsilon\text{-}c\ell\Big(\tilde{\delta}\big(\hat{\delta}(q_0, w), a\big)\Big) && \text{// Deshacer renombre}\\
&= \hat{\delta}(q_0, wa) && \text{// Def. } \hat{\delta} \text{ para AFND}{-}\varepsilon \text{ en Ecuación 1.6}
\end{aligned}$$

† $\underline{\bigcup_{q\in P} \varepsilon\text{-}c\ell\big(\bigcup_{p\in\varepsilon\text{-}c\ell(q)} \delta(p,a)\big) \subseteq \bigcup_{q\in P} \varepsilon\text{-}c\ell\big(\delta(q,a)\big)}$

Sea $r \in \bigcup\limits_{q\in P} \varepsilon\text{-}c\ell\Bigg(\bigcup\limits_{p\in\varepsilon\text{-}c\ell(q)} \delta(p,a)\Bigg) \Rightarrow \exists q \in P \,/\, r \in \varepsilon\text{-}c\ell\Bigg(\bigcup\limits_{p\in\varepsilon\text{-}c\ell(q)} \delta(p,a)\Bigg)$, que por la propiedad de Unión, es igual a $\bigcup\limits_{p\in\varepsilon\text{-}c\ell(q)} \varepsilon\text{-}c\ell\big(\delta(p,a)\big) \Rightarrow \exists q \in P,\ \exists p \in \varepsilon\text{-}c\ell(q) \,/\, r \in \varepsilon\text{-}c\ell\big(\delta(p,a)\big)$. Ahora bien, recordar que $P = \hat{\delta}(q_0, w)$ y la función $\hat{\delta}$ está definida como una $\varepsilon-$clausura para los AFND$-\varepsilon$. Entonces, por la propiedad de Cerradura, $\varepsilon\text{-}c\ell(q) \subseteq P$ y por ende $p \in P$. Luego, $\exists p \in P \,/\, r \in \varepsilon\text{-}c\ell\big(\delta(p,a)\big) \Rightarrow r \in \bigcup\limits_{p\in P} \varepsilon\text{-}c\ell\big(\delta(p,a)\big)$. Y como r es genérico, se cumple la inclusión deseada.

$\underline{\bigcup_{q\in P} \varepsilon\text{-}c\ell\big(\delta(q,a)\big) \subseteq \bigcup_{q\in P} \varepsilon\text{-}c\ell\big(\bigcup_{p\in\varepsilon\text{-}c\ell(q)} \delta(p,a)\big)}$

Sea $r \in \bigcup\limits_{q\in P} \varepsilon\text{-}c\ell\big(\delta(q,a)\big) \Rightarrow \exists q \in P \,/\, r \in \varepsilon\text{-}c\ell\big(\delta(q,a)\big)$. Visto que el propio $q \in \varepsilon\text{-}c\ell(q)$, ocurre que $\varepsilon\text{-}c\ell\big(\delta(q,a)\big) \subseteq \bigcup\limits_{p\in\varepsilon\text{-}c\ell(q)} \varepsilon\text{-}c\ell\big(\delta(p,a)\big) \overset{\text{Unión}}{=\!=} \varepsilon\text{-}c\ell\Bigg(\bigcup\limits_{p\in\varepsilon\text{-}c\ell(q)} \delta(p,a)\Bigg)$. Luego, se tiene que $\exists q \in P \,/\, r \in \varepsilon\text{-}c\ell\Bigg(\bigcup\limits_{p\in\varepsilon\text{-}c\ell(q)} \delta(p,a)\Bigg) \Rightarrow r \in \bigcup\limits_{q\in P} \varepsilon\text{-}c\ell\Bigg(\bigcup\limits_{p\in\varepsilon\text{-}c\ell(q)} \delta(p,a)\Bigg)$.
Y como r es genérico, se cumple la inclusión deseada. ■

Por último observamos que M y M' aceptan si y sólo si $\hat{\delta}(q_0, x)$ o $\hat{\delta}'(q_0', x)$ contienen, respectivamente, un estado de aceptación, completando así la prueba de que $\mathscr{L}(M) = \mathscr{L}(M')$.

Teorema 1.5 Equivalencia entre AFND–ε y AFND

Con el Teorema 1.4 vimos cómo un AFND–ε puede ser transformado en un AFND equivalente. Inversamente, podemos interpretar que un AFND es un AFND–ε el cual tiene la particularidad de no tener ε–transiciones ($\forall q \in Q \ : \ \delta(q, \varepsilon) = \varnothing$).

1.2.4. Equivalencia entre Expresiones Regulares y Autómatas Finitos

Hasta el momento hemos visto por un lado a las *expresiones regulares* como una notación formal para describir lenguajes regulares, y por otro lado a los *autómatas finitos* como máquinas abstractas capaces de computar tiras de Σ^* para reconocer lenguajes regulares.

Es hora de mostrar como estos dos modelos son totalmente equivalentes a la hora de **caracterizar** a los lenguajes regulares:

Teorema de Kleene

Teorema 1.6 Conversión de Expresión Regular en AFND–ε

La idea es aprovechar la estructura recursiva de las expresiones regulares para formular el proceso de conversión:

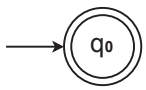
Casos base:

Sean r una expresión regular definida sobre un alfabeto Σ, y $M = (Q, \Sigma, \delta, q_0, F)$ el autómata a construir tal que $L = \mathscr{L}(r) = \mathscr{L}(M)$. Entonces:

- Si $r = \varnothing \ \Rightarrow \ L = \varnothing$, y creamos M de esta manera:

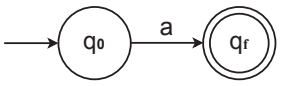
$$Q = \{q_0, q_f\} \qquad F = \{q_f\}$$
$$\forall q \in Q \ , \ \forall a \in \Sigma \ : \ \delta(q, a) = \varnothing$$

- Si $r = \varepsilon \ \Rightarrow \ L = \{\varepsilon\}$, y creamos M de esta manera:



$$Q = \{q_0\} \qquad F = \{q_0\}$$
$$\forall q \in Q \ , \ \forall a \in \Sigma \ : \ \delta(q, a) = \varnothing$$

- Si $r = a$ (con $a \in \Sigma$) $\Rightarrow$ $L = \{a\}$, y creamos M de esta manera:
  $Q = \{q_0, q_f\}$ $F = \{q_f\}$

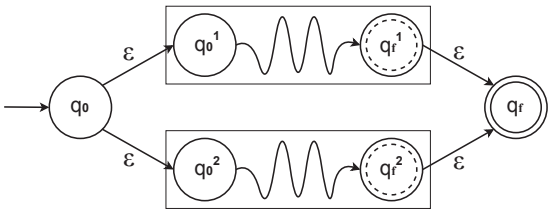


$$\forall q \in Q\ ,\ \forall \ell \in \Sigma\ :\ \delta(q,\ell) = \begin{cases} \{q_f\} & \text{si } q = q_0 \ \wedge\ \ell = a \\ \varnothing & \text{si } \ell \neq a \end{cases}$$

Casos inductivos:

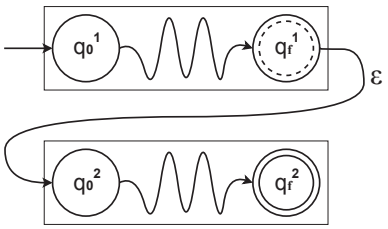
Sean r_1 y r_2 expresiones regulares definidas –respectivamente– sobre los alfabetos Σ_1 y Σ_2, y los autómatas $M_1 = \big(Q_1, \Sigma_1, \delta_1, {q_0}^1, \{{q_f}^1\}\big)$ y $M_2 = \big(Q_2, \Sigma_2, \delta_2, {q_0}^2, \{{q_f}^2\}\big)$ tales que $L_1 = \mathscr{L}(r_1) = \mathscr{L}(M_1)$ y $L_2 = \mathscr{L}(r_2) = \mathscr{L}(M_2)$. Entonces para la expresión regular r (que depende de las anteriores) definida sobre un alfabeto Σ, debemos construir un autómata $M = (Q, \Sigma, \delta, q_0, F)$ tal que $L = \mathscr{L}(r) = \mathscr{L}(M)$:

- Si $r = r_1 \mid r_2$ $\Rightarrow$ $L = L_1 \cup L_2$, y creamos M de esta manera:
  $\Sigma = \Sigma_1 \cup \Sigma_2$ $Q = Q_1 \cup Q_2 \cup \{q_0, q_f\}$ $F = \{q_f\}$



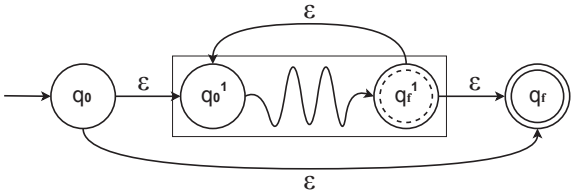
$$\delta(q,a) = \begin{cases} \delta_1(q,a) & \text{si } q \in Q_1 \ \wedge\ a \in \big(\Sigma_1 \cup \{\varepsilon\}\big) \\ \delta_2(q,a) & \text{si } q \in Q_2 \ \wedge\ a \in \big(\Sigma_2 \cup \{\varepsilon\}\big) \\ \{{q_0}^1, {q_0}^2\} & \text{si } q = q_0 \ \wedge\ a = \varepsilon \\ \{q_f\} & \text{si } q \in \{{q_f}^1, {q_f}^2\} \ \wedge\ a = \varepsilon \end{cases}$$

- Si $r = r_1 \cdot r_2$ $\Rightarrow$ $L = L_1 \cdot L_2$, y creamos M de esta manera:
  $\Sigma = \Sigma_1 \cup \Sigma_2$ $Q = Q_1 \cup Q_2$ $F = \{{q_f}^2\}$ $q_0 = {q_0}^1$



$$\delta(q,a) = \begin{cases} \delta_1(q,a) & \text{si } q \in Q_1 \ \wedge\ a \in \big(\Sigma_1 \cup \{\varepsilon\}\big) \\ \delta_2(q,a) & \text{si } q \in Q_2 \ \wedge\ a \in \big(\Sigma_2 \cup \{\varepsilon\}\big) \\ \{{q_0}^2\} & \text{si } q = {q_f}^1 \ \wedge\ a = \varepsilon \end{cases}$$

- Si $r = {r_1}^*$ $\Rightarrow$ $L = {L_1}^*$, y creamos M de esta manera:
  $\Sigma = \Sigma_1$ $Q = Q_1 \cup \{q_0, q_f\}$ $F = \{q_f\}$



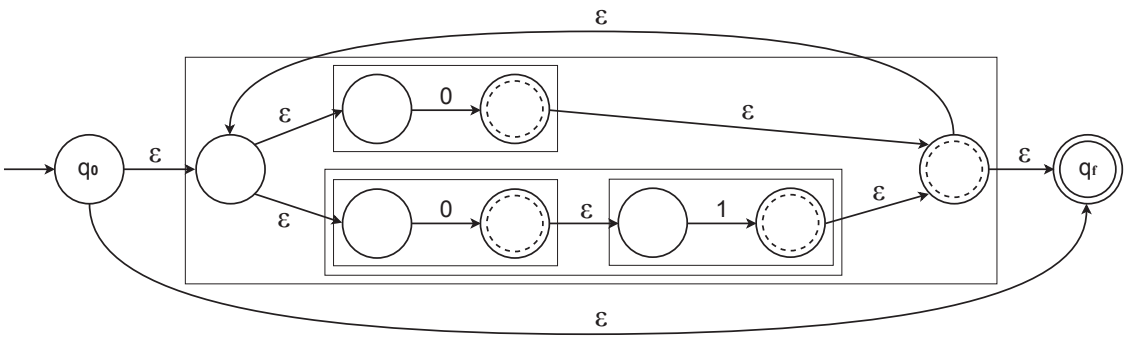
$$\delta(q,a) = \begin{cases} \delta_1(q,a) & \text{si } q \in Q_1 \ \wedge\ a \in \big(\Sigma_1 \cup \{\varepsilon\}\big) \\ \{{q_0}^1, q_f\} & \text{si } q \in \{q_0, {q_f}^1\} \ \wedge\ a = \varepsilon \end{cases}$$

■

Esta manera de convertir expresiones regulares en autómatas finitos que acabamos de ver, tiene la **desventaja** de que el resultado es una máquina con una gran cantidad de estados y ε–transiciones, por lo que en la práctica no es muy conveniente.

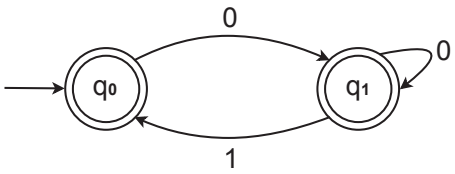
Ejemplo 1.5 *Tenemos la expresión regular* $r = (0|01)^*$ *y queremos un autómata finito que reconozca* $\mathscr{L}(r)$.

Si aplicamos el Teorema 1.6, obtenemos el siguiente AFND–ε *que consta de 10 estados y 9* ε*–transiciones:*



Autómata finito para $\mathscr{L}(r)$ según el Teorema 1.6

Por otro lado, con un poco de ingenio, podemos crear este AFD para reconocer $\mathscr{L}(r)$ *con tan solo 2 estados:*



Autómata finito para $\mathscr{L}(r)$ equivalente al anterior

Teorema 1.7 Conversión de AFD en Expresión Regular

A continuación veremos el resultado que termina de probar el *Teorema de Kleene* (equivalencia entre expresiones regulares y autómatas finitos). Es importante aclarar que el método que aquí se presenta es un resultado teórico **inviable en la práctica**. En su lugar utilizaremos alguna de las estrategias descritas en la Subsección 4.3.

Dado un AFD $M = (Q, \Sigma, \delta, q_1, F)$ con $L = \mathscr{L}(M)$, queremos obtener una expresión regular r tal que $L = \mathscr{L}(r)$.

Supongamos que los estados de Q están numerados del 1 al n, con $n = |Q|$. Es decir, $Q = \{q_1, \cdots, q_n\}$. Lo primero que haremos es construir una colección de conjuntos que, de manera progresiva, describan caminos en el diagrama de estados de M cada vez más largos. Para eso definimos

$$R_{ij}{}^k = \{\textit{strings que van del estado } q_i \textit{ al } q_j\textit{, sin pasar por un estado mayor a } q_k\}$$

y así podemos establecer que si $F = \{q_{f_1}, \cdots, q_{f_r}\}$ $(1 \leq f_i \leq n)$, entonces

$$\mathscr{L}(M) = \bigcup_{i=1}^{r} {R_{1f_i}}^n$$

Evidentemente necesitamos una definición formal de los ${R_{ij}}^k$ para hacer rigurosa nuestra construcción, con lo cual la formularemos inductivamente según k:

- **Caso base:** ${R_{ij}}^0 = \begin{cases} \{a \in \Sigma \ / \ \delta(q_i, a) = q_j\} & \text{si} \quad i \neq j \\ \{a \in \Sigma \ / \ \delta(q_i, a) = q_j\} \cup \{\varepsilon\} & \text{si} \quad i = j \end{cases}$
- **Casos inductivos:** ${R_{ij}}^k = {R_{ij}}^{k-1} \cup \left({R_{ik}}^{k-1} \cdot \left({R_{kk}}^{k-1} \right)^* \cdot {R_{kj}}^{k-1} \right)$

Ahora lo único que resta es probar por inducción que podemos encontrar una expresión regular ${r_{ij}}^k$ para cada ${R_{ij}}^k$:

- **Paso base:** Como los elementos de ${R_{ij}}^0$ son ε o símbolos de Σ, tendremos una cantidad finita de ellos, con lo cual simplemente los juntamos todos con el operador de unión:

$${r_{ij}}^0 = \begin{cases} a_1 \mid \cdots \mid a_t & \text{si} \quad i \neq j \\ a_1 \mid \cdots \mid a_t \mid \varepsilon & \text{si} \quad i = j \end{cases} \qquad \text{donde} \quad a_i \in \Sigma \quad \text{y} \quad t = \left| {R_{ij}}^0 \right|$$

- **Paso inductivo:** Asumimos como hipótesis de inducción que tenemos las expresiones regulares para los conjuntos con exponente $k-1$, a saber: ${r_{ij}}^{k-1}$, ${r_{ik}}^{k-1}$, ${r_{kk}}^{k-1}$ y ${r_{kj}}^{k-1}$. Luego, nuestro objetivo es obtener una expresión regular para el conjunto con exponente k, i.e. ${r_{ij}}^k$:

$${r_{ij}}^k \ = \ {r_{ij}}^{k-1} \ \Big| \ {r_{ik}}^{k-1} \cdot \left({r_{kk}}^{k-1} \right)^* \cdot {r_{kj}}^{k-1}$$

■

1.2.5. Relación R_M, Teorema de Myhill–Nerode y Minimización

Hasta el momento hemos visto que para un mismo lenguaje regular, existen muchos autómatas finitos equivalentes. En esta sección veremos una serie de resultados que nos permitirán construir AFDs con una **cantidad mínima de estados**.

Definición 1.14 Dado un AFD $M = (Q, \Sigma, \delta, q_0, F)$, definimos la ***relación binaria*** $\boldsymbol{R_M}$ sobre Σ^*, de forma tal que – dados $x, y \in \Sigma^*$ – se tiene que $x \ R_M \ y$ si

$$\hat{\delta}(q_0, x) = \hat{\delta}(q_0, y)$$

(el estado al que llega M tras procesar x es el mismo que al que llega tras procesar y).

Observación: R_M es una **relación de equivalencia** en la cual $\#\texttt{clases}(R_M) = |Q|$. Además se cumple para cualquier lenguaje regular que $\#\texttt{clases}(R_L) \leq \#\texttt{clases}(R_M)$.

Teorema 1.8 Teorema de Myhill–Nerode

$$L \text{ es regular} \iff \#\texttt{clases}(R_L) < \infty$$

Demostración $\Rightarrow$

$$\begin{aligned} L \text{ es regular} &\Rightarrow \exists \text{ AFD } M = (Q, \Sigma, \delta, q_0, F) \;/\; L = \mathscr{L}(M) && // \text{ Def. leng. regular en Definición 1.5} \\ &\Rightarrow \#\texttt{clases}(R_L) \leq \#\texttt{clases}(R_M) = |Q| < \infty && // \text{ Observación previa} \\ &\Rightarrow \#\texttt{clases}(R_L) < \infty && // \text{ Transitividad} \end{aligned}$$

Demostración $\Leftarrow$

$\#\texttt{clases}(R_L) < \infty \Rightarrow$ Conocemos las clases de equivalencia de R_L, y a partir de ellas construimos un AFD $M = (Q, \Sigma, \delta, q_0, F) \;/\; L = \mathscr{L}(M)$ donde:

- $Q = \big\{[w_i] \;/\; w_i \text{ es el representante de la } i\text{–ésima clase de } R_L\big\}$
- Σ es el mismo alfabeto sobre el que se define L
- $\forall x \in \Sigma^*,\ \forall a \in \Sigma \;:\; \delta\big([x], a\big) = [xa]$
- $q_0 = [\varepsilon]$
- $F = \big\{[x] \;/\; x \in L\big\}$

$$\begin{aligned} \text{Luego } \; x \in \mathscr{L}(M) &\iff \hat{\delta}\big([\varepsilon], x\big) \in F && // \text{ Def. de aceptación en Definición 1.4} \\ &\iff [x] \in F && // \text{ Construcción del } \delta \text{ para este caso} \\ &\iff x \in L && // \text{ Construcción de } F \text{ para este caso} \end{aligned}$$

■

Definición 1.15 Dado un AFD $M = (Q, \Sigma, \delta, q_0, F)$, diremos que dos estados p y q son ***equivalentes*** si

$$\forall x \in \Sigma^* \;:\; \hat{\delta}(p, x) \in F \iff \hat{\delta}(q, x) \in F$$

Si $\exists\, x \in \Sigma^* \;/\; \hat{\delta}(p, x) \in F \;\wedge\; \hat{\delta}(q, x) \notin F$ o viceversa, decimos que p y q son ***distinguibles***.

Teorema 1.9 Algoritmo de Minimización de AFD

Dado un AFD $M = (Q, \Sigma, \delta, q_0, F)$, la idea subyacente a este algoritmo es particionar iterativamente el conjunto de estados Q de manera que, al finalizar cada iteración, los estados internos a cualquier subconjunto de la partición que se acaba de construir sean equivalentes entre sí (de acuerdo a la Definición 1.15).

Se comienza con una partición base

$$\pi_0 \;:\; Q \setminus F \qquad F$$

y se van construyendo sucesivas particiones π_i hasta que, para cierto i, $\pi_i = \pi_{i-1}$. Cuando llegamos a este punto hemos encontrado una partición que no se puede "refinar" más,

con lo cual paramos de iterar y en base a los subconjuntos que nos quedan construimos un nuevo autómata equivalente M' que será mínimo.

El Algoritmo 4.4 explica el procedimiento con más detalles.

1.2.6. Autómatas con Salida

En el correr de esta subsección hemos visto autómatas finitos que se limitan a reconocer lenguajes regulares, y todos ellos comparten la característica de tener una única cinta que sirve para leer la entrada.

La teoría de autómatas nos permite extender el modelo matemático de la 5-tupla para crear máquinas con dos cintas, capaces no sólo de leer una entrada desde una cinta, sino también de escribir una salida en otra cinta independiente de la primera. Formalmente a estas máquinas se les llama ***Transductores de Estados Finitos (FST)***, ya que *transducen* ("traducen") una tira leída en la entrada a otra tira que queda impresa en la salida.

En esta parte presentaremos dos variantes clásicas – ambas deterministas – de este tipo de máquinas, veremos la equivalencia entre ellas, y – al igual que los AFD – cómo se pueden minimizar.

Definición 1.16 Una ***máquina de Mealy*** es un FST cuya salida depende tanto del estado actual como de la entrada actual. Formalmente es una 6-tupla $M = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$ donde:

- Q es el conjunto de estados
- Σ es el alfabeto de la entrada
- Δ es el alfabeto de la salida
- $\delta : Q \times \Sigma \to Q$ es la función de transición de estados
- $\lambda : Q \times \Sigma \to \Delta$ es la función de salida
- $q_0 \in Q$ es el estado inicial

Al igual que antes, precisamos extender la función δ para poder procesar tiras completas de Σ^*, con lo cual tendremos $\hat{\delta} : Q \times \Sigma^* \to Q$ definida de la siguiente manera:

$$\forall q \in Q \ : \ \hat{\delta}(q, \varepsilon) = q$$
$$\forall q \in Q \ , \ \forall a \in \Sigma \ , \ \forall x \in \Sigma^* \ : \ \hat{\delta}(q, xa) = \delta\big(\hat{\delta}(q, x), a\big)$$

Observar que la función λ también es capaz de procesar de a un símbolo de Σ. Es por esto que para ella también necesitamos definir una función $\hat{\lambda} : Q \times \Sigma^* \to \Delta^*$ que nos

permita procesar toda una tira de Σ^* por completo:

$$\forall q \in Q \ : \ \hat{\lambda}(q, \varepsilon) = \varepsilon$$
$$\forall q \in Q \ , \ \forall a \in \Sigma \ , \ \forall x \in \Sigma^* \ : \ \hat{\lambda}(q, xa) = \hat{\lambda}(q, x) \cdot \lambda\big(\hat{\delta}(q, x), a\big)$$

Observación: Notar que un autómata con salida **no** tiene un conjunto de estados de aceptación (como sí tienen los autómatas finitos vistos anteriormente), ya que con este tipo de máquinas no buscamos decidir si una tira pertenece o no a cierto lenguaje.

Definición 1.17 Una ***máquina de Moore*** es un FST cuya salida depende únicamente del estado actual. Formalmente es una 6-tupla $M = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$ en la que:

- Q es el conjunto de estados
- Σ es el alfabeto de la entrada
- Δ es el alfabeto de la salida
- $\delta : Q \times \Sigma \to Q$ es la función de transición de estados
- $\lambda : Q \to \Delta$ es la función de salida
- $q_0 \in Q$ es el estado inicial

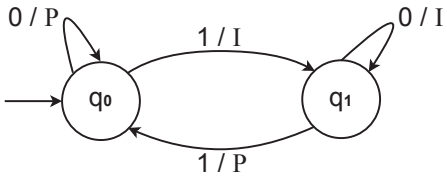
La extensión de la función δ para poder procesar tiras de Σ^* es igual que en el modelo de Mealy, es decir, $\hat{\delta} : Q \times \Sigma^* \to Q$ queda definida de la siguiente manera:

$$\forall q \in Q \ : \ \hat{\delta}(q, \varepsilon) = q$$
$$\forall q \in Q \ , \ \forall a \in \Sigma \ , \ \forall x \in \Sigma^* \ : \ \hat{\delta}(q, xa) = \delta\big(\hat{\delta}(q, x), a\big)$$

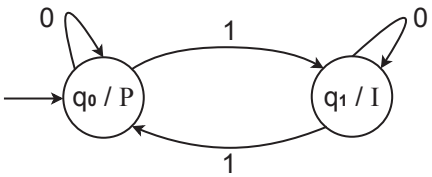
Pero a diferencia del modelo anterior, en este caso la función λ no depende de Σ, con lo cual no podemos extenderla a una $\hat{\lambda}$ que procese cadenas de Σ^*. En su lugar, tendremos otra función $\bar{\lambda} : Q \times \Sigma^* \to \Delta^*$ que sí pueda procesar tiras de Σ^* completamente:

$$\forall q \in Q \ : \ \bar{\lambda}(q, \varepsilon) = \varepsilon$$
$$\forall q \in Q \ , \ \forall a \in \Sigma \ , \ \forall x \in \Sigma^* \ : \ \bar{\lambda}(q, xa) = \bar{\lambda}(q, x) \cdot \lambda\big(\hat{\delta}(q, xa)\big)$$

Ejemplo 1.6 *Queremos una máquina con salida capaz de leer una señal binaria tal que, por cada bit leído, imprima P si la cantidad de unos leída hasta el momento es par, o imprima I en caso contrario.*



(a) Máquina de **Mealy**



(b) Máquina de **Moore**

Resolución del Ejemplo 1.6 según cada modelo

*En la notación gráfica para las máquinas de **Mealy** rotulamos las transiciones de la forma "`entrada/salida`", mientras que en las máquinas de **Moore** rotulamos los estados de la forma "`nombreEstado/salida`". Al igual que antes, en ambos ponemos una flecha apuntando al estado inicial.*

Como es de esperar, una máquina de Mealy tiene una máquina de Moore equivalente, y viceversa. Los Teoremas 1.10 y 1.11 prueban la equivalencia en ambas direcciones, exponiendo un algoritmo de conversión para cada caso.

Teorema 1.10 Conversión de Mealy a Moore

Dada una máquina de Mealy $M = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$, crearemos una máquina de Moore $M' = (Q', \Sigma', \Delta', \delta', \lambda', q_0')$ siguiendo estas reglas:

1) $Q' = \{q^t \ / \ q \in Q \ \wedge \ t \in \Delta\}$

2) $\Sigma' = \Sigma$

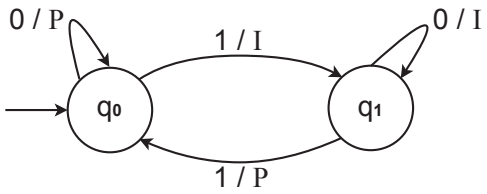
3) $\Delta' = \Delta$

4) $\forall q \in Q \ , \ \forall a \in \Sigma \ :$ **Si** $p = \delta(q, a)$ y $r = \lambda(q, a)$ **entonces** $\forall t \in \Delta \ : \ \delta'(q^t, a) = p^r$

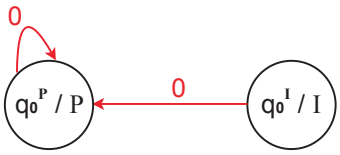
5) $\forall q^t \in Q' \ , \ \forall t \in \Delta \ : \ \lambda'(q^t) = t$

6) $q_0' = {q_0}^t \in Q'$ (para un $t \in \Delta$ arbitrario)

Ejemplo 1.7 *Retomemos la máquina de Mealy vista en el Ejemplo 1.6:*

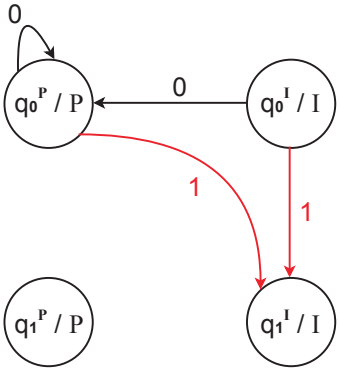


Vamos a aplicar el Teorema 1.10 para convertirla a una máquina de Moore:

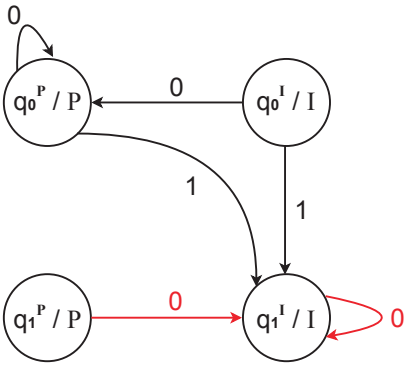


Primero creamos los estados q^t de Q', donde $q \in \{q_0, q_1\}$ y $t \in \{P, I\}$

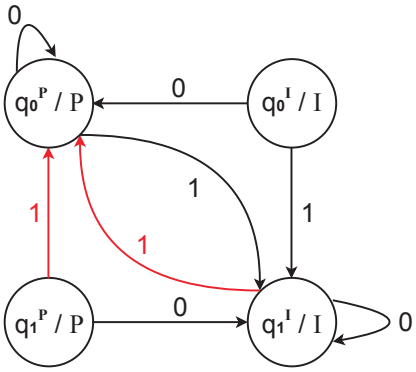
$\delta(q_0, 0) = q_0$ y $\lambda(q_0, 0) = P$
$\Rightarrow \ \delta'({q_0}^t, 0) = {q_0}^P$



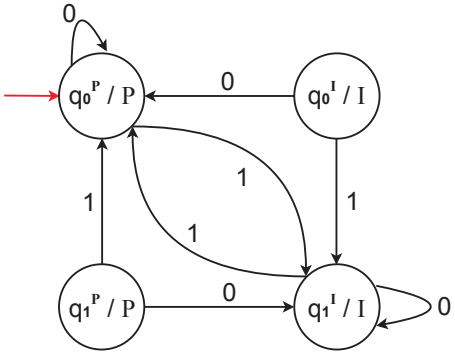
$\delta(q_0, 1) = q_1$ *y* $\lambda(q_0, 1) = I$
$\Rightarrow$ $\delta'({q_0}^t, 1) = {q_1}^I$



$\delta(q_1, 0) = q_1$ *y* $\lambda(q_1, 0) = I$
$\Rightarrow$ $\delta'({q_1}^t, 0) = {q_1}^I$

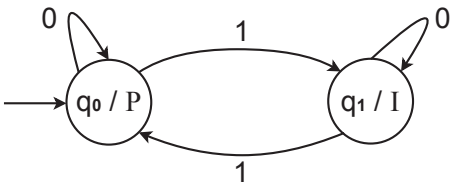


$\delta(q_1, 1) = q_0$ *y* $\lambda(q_1, 1) = P$
$\Rightarrow$ $\delta'({q_1}^t, 1) = {q_0}^P$



Finalmente elegimos uno de los ${q_0}^t$ para que sea el estado inicial

Notar que el estado ${q_1}^P$ *queda inalcanzable ("está de sobra"), y lo mismo sucede con* ${q_0}^I$ *dada nuestra elección del estado inicial. Si quitamos ambos nos queda la máquina que vimos en el Ejemplo 1.6:*



Teorema 1.11 Conversión de Moore a Mealy

Dada una máquina de Moore $M = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$, crearemos una máquina de Mealy $M' = (Q', \Sigma', \Delta', \delta', \lambda', {q_0}')$ siguiendo estas reglas:

1) $Q' = Q$

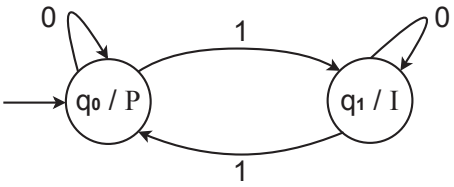
2) $\Sigma' = \Sigma$

3) $\Delta' = \Delta$

4) $\forall q \in Q\ ,\ \forall a \in \Sigma$: **Si** $p = \delta(q, a)$ y $r = \lambda(p)$ **entonces** $\delta'(q, a) = p$ y $\lambda'(q, a) = r$

5) ${q_0}' = q_0$

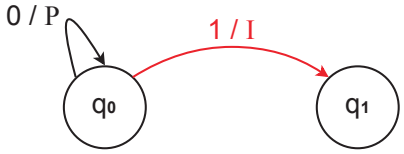
Ejemplo 1.8 *Recordemos la máquina de Moore vista en el Ejemplo 1.6:*



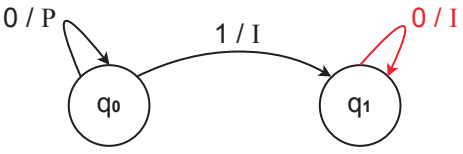
Vamos a aplicar el Teorema 1.11 para convertirla a una máquina de Mealy:

Primero ubicamos los estados de Q'

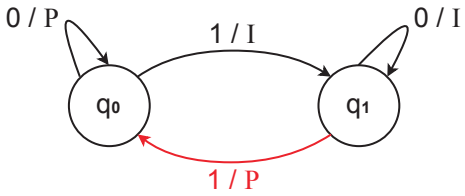
$\delta(q_0, 0) = q_0 \ y \ \lambda(q_0) = P$
$\Rightarrow \ \delta'(q_0, 0) = q_0 \ y \ \lambda'(q_0, 0) = P$



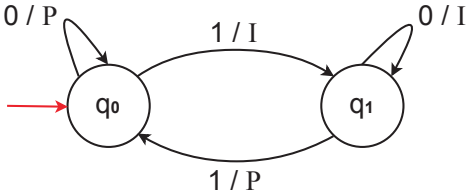
$\delta(q_0, 1) = q_1 \ y \ \lambda(q_1) = I$
$\Rightarrow \ \delta'(q_0, 1) = q_1 \ y \ \lambda'(q_0, 1) = I$



$\delta(q_1, 0) = q_1 \ y \ \lambda(q_1) = I$
$\Rightarrow \ \delta'(q_1, 0) = q_1 \ y \ \lambda'(q_1, 0) = I$



$\delta(q_1, 1) = q_0 \ y \ \lambda(q_0) = P$
$\Rightarrow \ \delta'(q_1, 1) = q_0 \ y \ \lambda'(q_1, 1) = P$



Marcamos el estado inicial

Tal como adelantábamos al principio de esta subsección, lo último que nos queda por estudiar es el proceso de minimización de los autómatas con salida.

Teorema 1.12 Minimización de Autómatas con Salida

Antes que nada hemos de aclarar que el siguiente algoritmo se aplica a máquinas de Mealy. En caso de tener una máquina de Moore simplemente aplicamos el Teorema 1.11 que acabamos de ver como paso previo. Entonces, a partir de una máquina de Mealy $M = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$, construiremos otra máquina de Mealy M' equivalente con una cantidad mínima de estados.

Comenzaremos redefiniendo la equivalencia entre estados que vimos en la Definición 1.15. En este contexto, diremos que dos estados p y q son equivalentes cuando

$$\forall x \in \Sigma^* \ : \ \hat{\lambda}(p, x) = \hat{\lambda}(q, x)$$

Luego el mecanismo de conversión no difiere sustancialmente de lo que explicamos en el Teorema 1.9 (Minimización de AFD). De hecho, la única diferencia radica en la construcción de la partición base, que en este caso lo hacemos según λ:

$$\pi_0 \ : \quad C_1 \quad \cdots \quad C_j \quad \cdots \quad C_t$$

donde $C_j = \{q_i, \cdots, q_k \in Q \ / \ \forall a \in \Sigma \ , \ \forall m, n \in \{i, \cdots, k\} \ : \ \lambda(q_m, a) = \lambda(q_n, a)\}$.

Es decir, armamos los subconjuntos C de π_0 agrupando aquellos estados que produzcan la misma salida para todos los símbolos de la entrada.

Un vez que tenemos establecida la partición base, construimos las sucesivas particiones π_i de la misma forma que en el otro algoritmo (i.e. según δ). Por lo tanto, el pseudocódigo de este procedimiento coincide con el Algoritmo 4.4, salvo en la línea 2 que es donde justamente se define el π_0.

1.2.7. Autómata Finito Determinista de Dos Cintas

En esta parte veremos una última variante de los AFD, que –como su nombre lo indica– tienen dos cintas independientes. La diferencia con los FST que acabamos de estudiar está en que aquí ambas cintas son de entrada, y entonces el propósito del autómata es decidir pares de strings.

Definición 1.18 Un ***autómata finito determinista de dos cintas*** es una 6-tupla $M = (Q_1, Q_2, \Sigma, \delta, q_0, F)$ donde:

- Q_1 es el conjunto de estados que avanzan sobre la cinta #1
- Q_2 es el conjunto de estados que avanzan sobre la cinta #2
- Σ es el alfabeto de los pares de entrada
- $\delta : (Q_1 \cup Q_2) \times \Sigma \to (Q_1 \cup Q_2)$ es la función de transición de estados
- $q_0 \in (Q_1 \cup Q_2)$ es el estado inicial
- $F \subseteq (Q_1 \cup Q_2)$ es el conjunto de estados de aceptación

Como todos los casos estudiados previamente, es necesario extender la capacidad de la función δ para que pueda procesar pares de tiras de $\Sigma^* \times \Sigma^*$ por completo, con lo cual tenemos $\hat{\delta} : (Q_1 \cup Q_2) \times \Sigma^* \times \Sigma^* \to (Q_1 \cup Q_2)$ definida como sigue:

$$\forall q \in (Q_1 \cup Q_2) \ : \ \hat{\delta}(q, \varepsilon, \varepsilon) = q$$

$$\forall q_i^1 \in Q_1 \ , \ \forall a \in \Sigma \ , \ \forall x, y \in \Sigma^* \ : \ \hat{\delta}\big(q_i^1, ax, y\big) = \hat{\delta}\Big(\delta\big(q_i^1, a\big), x, y\Big)$$

$$\forall q_j^2 \in Q_2 \ , \ \forall a \in \Sigma \ , \ \forall x, y \in \Sigma^* \ : \ \hat{\delta}\big(q_j^2, x, ay\big) = \hat{\delta}\Big(\delta\big(q_j^2, a\big), x, y\Big)$$

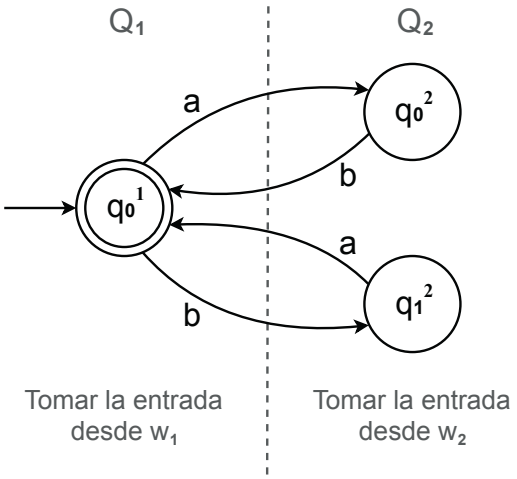
Definición 1.19 Dado un AFD de dos cintas $M = (Q_1, Q_2, \Sigma, \delta, q_0, F)$, el ***lenguaje aceptado por M*** es $\mathscr{L}(M) = \big\{(x, y) \in \Sigma^* \times \Sigma^* \ / \ \hat{\delta}(q_0, x, y) \in F\big\}$. Es decir, $\mathscr{L}(M)$ es el conjunto de pares de tiras de $\Sigma^* \times \Sigma^*$ tales que, partiendo desde q_0, M consume todos los símbolos de ambas tiras y se detiene en un estado de aceptación.

Observación: La función $\hat{\delta}$ expuesta en la Definición 1.18 no es total, ya que no tiene asociada una expresión para los casos tipo $\hat{\delta}\big(q_i^1, \varepsilon, ay\big)$. Estas son instancias en las que el autómata se encuentra en un estado $q_i^1 \in Q_1$, ya consumió toda la tira de la cinta #1, pero todavía le queda al menos un símbolo a por consumir en la cinta #2. Dado que el estado actual determina sobre qué cinta debe leer el autómata, es claro que en una situación como esta el autómata queda "trancado" y **rechaza** el par de strings dado como entrada. Una situación análoga sucede con los casos tipo $\hat{\delta}\big(q_j^2, ax, \varepsilon\big)$.

Como veremos en los ejemplos a continuación, para representar un AFD de dos cintas mediante un diagrama de estados, es de utilidad trazar una línea punteada que delimite los conjuntos de estados Q_1 y Q_2.

Ejemplo 1.9 *Construir un AFD de dos cintas que acepte el lenguaje:*

$$L_1 = \left\{(w_1, w_2) \in \{a,b\}^* \times \{a,b\}^* \ / \ |w_1| = |w_2| \ \wedge \ w_1(i) = a \leftrightarrow w_2(i) = b\right\}$$



Este autómata tiene como estado inicial $q_0 = q_0^1$, por lo tanto el primer símbolo que lea provendrá desde la tira w_1. Si el símbolo leído en w_1 es una a (análogamente para una b), el autómata pasa a q_0^2 (q_1^2) y allí queda habilitado solamente a leer una b (a) desde w_2. Si efectivamente se lee una b (a), el autómata vuelve a q_0^1 y repite este procedimiento con el próximo símbolo de w_1. Puesto que para cada símbolo leído desde w_1 inmediatamente se pasa a w_2 a buscar su complementario y – de encontrarlo – acto seguido se vuelve a w_1, sobre la marcha se va controlando que $|w_1| = |w_2|$. En caso de poder consumir todos los símbolos de ambas tiras como se espera, el autómata se detendrá en q_0^1 (el único estado de aceptación), y aceptará el par de strings dado como entrada.

Si en algún momento – ya sea en q_0^2 o en q_1^2 – el autómata no lee el símbolo esperado, al no tener una transición definida (dibujada), "se tranca" y no acepta la dupla de strings.

Si el autómata encuentra los símbolos esperados pero $|w_1| < |w_2|$, este se detendrá en q_0^1, que si bien es un estado de aceptación, como todavía le quedan símbolos por leer en w_2 no aceptará el par. De manera similar, si $|w_1| > |w_2|$, el autómata va a terminar en q_0^2 o q_1^2, pudiendo pasar que haya consumido por completo ambas tiras (precisamente cuando $|w_1| = 1 + |w_2|$), o bien que termine en uno de esos dos estados con símbolos pendientes para leer en w_1. En el primer caso el autómata rechaza el par por haberse detenido en un estado que no es de aceptación, mientras que en el segundo lo rechaza por no poder seguir leyendo los símbolos en w_1.

Por ejemplo:

- *El par $(ab, ba) \in \mathscr{L}(M)$, ya que $\hat{\delta}\left(q_0^1, ab, ba\right) = q_0^1 \in F = \{q_0^1\}$.*
- *El par $(aba, bbb) \notin \mathscr{L}(M)$, pues cuando lee la b de w_1 pasa a q_1^2, encontrándose en w_2 con una b donde se esperaba una a ("se tranca" debido a que $\hat{\delta}\left(q_0^1, aba, bbb\right) = \cdots = \hat{\delta}\left(q_1^2, a, bb\right) = \hat{\delta}\left(\delta\left(q_1^2, b\right), a, b\right)$, y $\delta\left(q_1^2, b\right)$ está indefinida).*
- *El par $(a, ba) \notin \mathscr{L}(M)$, dado que cuando llega al final de w_1 queda en un estado de Q_1 y todavía le resta una a por leer en w_2 ("se tranca" porque $\hat{\delta}\left(q_0^1, a, ba\right) = \cdots = \hat{\delta}\left(q_0^1, \varepsilon, a\right)$, y esta última no tiene una expresión definida).*

- *El par* $(ab, b) \notin \mathscr{L}(M)$*, ya que* $\hat{\delta}\big(q_0^1, ab, b\big) = q_1^2 \notin F$*.*
- *El par* $(bab, a) \notin \mathscr{L}(M)$*, pues cuando lee la a de* w_1 *pasa a* q_0^2*, quedando así en un estado de* Q_2 *y encontrándose ya en el final de* w_2*, pero teniendo una b por leer en* w_1 *("se tranca" porque* $\hat{\delta}\big(q_0^1, bab, a\big) = \cdots = \hat{\delta}\big(q_0^2, b, \varepsilon\big)$*, y esta última no tiene una expresión definida).*

Ejemplo 1.10 *Construir un AFD de dos cintas para reconocer el siguiente lenguaje:*

$$L_2 = \big\{(a^n\, c\, b^{n\,\%2}\,,\, a^m\, c) \;/\; m \geq n \geq 0\big\}$$

Antes que nada, observar que la condición "$m \geq n$*" implica que* $\exists k \geq 0 \;/\; m = n+k$*, entonces las duplas de este lenguaje se pueden escribir como* $(a^n\, c\, b^{n\,\%2}\,,\, a^{n+k}\, c) \;/\; n, k \geq 0$*, que es lo mismo que* $(a^n\, c\, b^{n\,\%2}\,,\, a^n\, a^k\, c) \;/\; n, k \geq 0$*.*

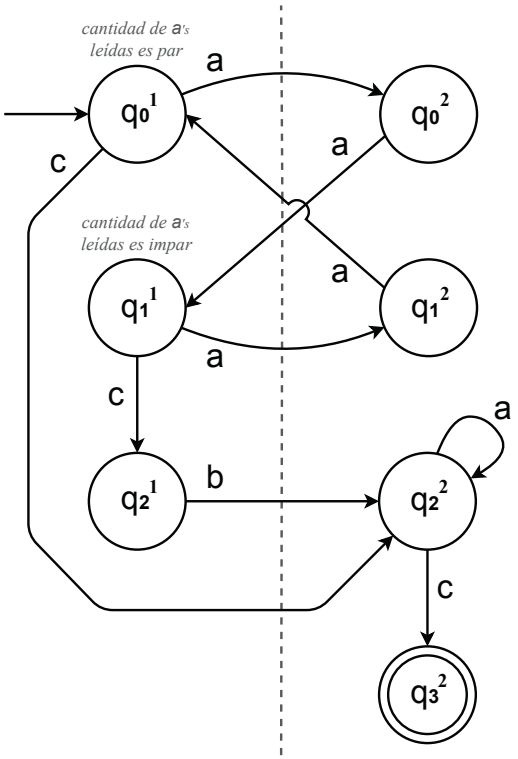
Más aún, el término "$n\,\%\,2$ *[n módulo 2]" nos permite segregar en dos tipos de duplas: cuando* $\boldsymbol{n}$ *es* ***par*** *las tiras de la cinta #1 no terminan en b, i.e. las duplas son de la forma*

$$(a^n\, c\;,\, a^n\, a^k\, c) \;/\; n, k \geq 0$$

y cuando $\boldsymbol{n}$ *es* ***impar*** *sí se tiene una b al final, siendo las duplas de la forma*

$$(a^n\, c\; b\;,\, a^n\, a^k\, c) \;/\; n, k \geq 0$$

Entonces, teniendo en cuenta lo anterior, creamos el siguiente autómata:



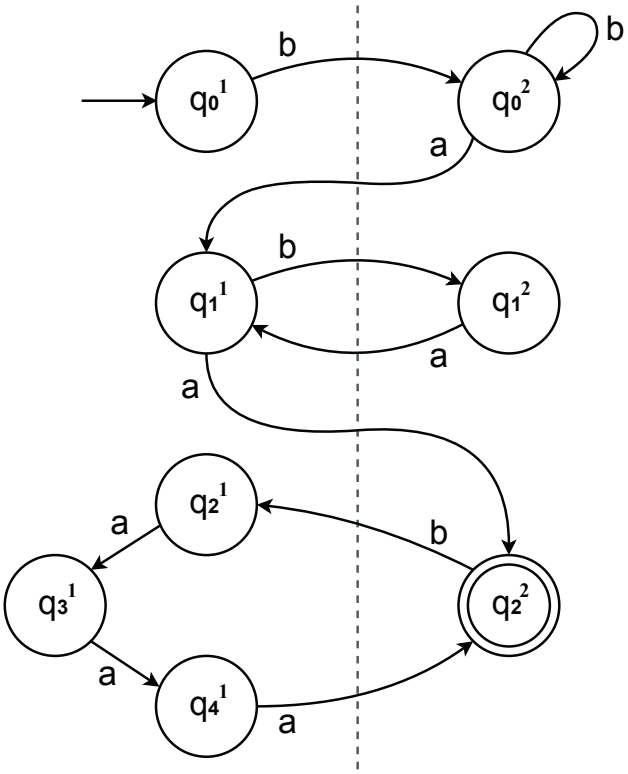
Ejemplo 1.11 *Construir un AFD de dos cintas que decida el siguiente lenguaje:*

$$L_3 = \left\{ (b^j\, a^k\, ,\, b^n\, a^j\, b^m) \;/\; k = 3m+1 \;\wedge\; m,n \geq 0 \;\wedge\; j > 0 \right\}$$

En primer lugar, notar que la condición "$j > 0$" implica $j \geq 1 \Rightarrow \exists \ell \geq 0 \;/\; j = 1+\ell$. Entonces los pares de tiras de este lenguaje se pueden escribir como $(b^{1+\ell}\, a^{1+3m}\, ,\, b^n\, a^{1+\ell}\, b^m)$ / $\ell, m, n \geq 0$, que es equivalente a

$$(b\; b^\ell\, a\; a^{3m}\, ,\, b^n\, a\; a^\ell\, b^m) \;/\; \ell, m, n \geq 0$$

Entonces, teniendo esto en cuenta, creamos el siguiente autómata:



1.3. Propiedades de los Lenguajes Regulares

1.3.1. Pumping Lemma para Lenguajes Regulares

El ***Pumping Lemma*** (*Lema de Bombeo* en español) es una propiedad fundamental que poseen todos los lenguajes regulares, pero **no** es el tipo de propiedad que podamos emplear para justificar que un lenguaje sea regular.

Esto último se debe a que el Pumping Lemma define una **condición necesaria pero no suficiente**, es decir: lo que dice esta propiedad es algo que cualquier lenguaje regular *necesariamente* **debe** cumplir para serlo, sin embargo, que cierto lenguaje cumpla lo que esta propiedad establece **no** es *suficiente* para afirmar que efectivamente se trate de un lenguaje regular.

Teorema 1.13 Pumping Lemma para Lenguajes Regulares

Si *L es un lenguaje regular* **entonces**

$$\exists n \in \mathbb{N} \;/\; \forall z \in L \text{ con } |z| \geq n \text{ se cumple que}$$

$$\exists \text{ una descomposición } z = uvw \;/\; |uv| \leq n \;,\; |v| \geq 1 \quad y \quad \forall i \geq 0 : uv^i w \in L$$

Demostración:

Por hipótesis sabemos que L es regular, por lo tanto –según la Definición 1.5– existe un AFD $M = (Q, \Sigma, \delta, q_0, F) \;/\; L = \mathscr{L}(M)$.

El Teorema lo primero que dice es que "$\exists n \in \mathbb{N} \;/ \cdots$", entonces nosotros proponemos que dicho n sea exactamente $|Q|$, i.e. la cantidad de estados de M.

Lo que sigue es un "$\forall z \in L$ *con* $|z| \geq n \cdots$", con lo cual para probar este para-todo consideramos un z genérico letra por letra: $z = a_1 \cdots a_m \;/\; z \in L \quad y \quad m \geq n \quad (a_i \in \Sigma)$.

Antes de continuar con el enunciado de la tesis, definamos la siguiente notación:

$$p_0 = \hat{\delta}(q_0, \varepsilon) \quad , \quad p_1 = \hat{\delta}(q_0, a_1) \quad , \quad \cdots \quad , \quad p_i = \hat{\delta}(q_0, a_1 \cdots a_i)$$

Observaciones:

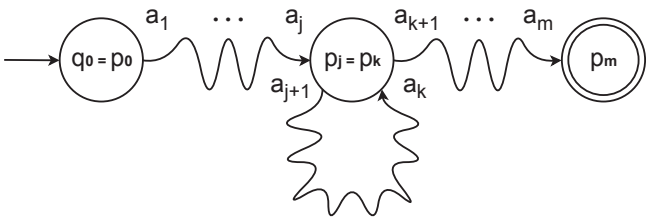
1) $p_0 = \hat{\delta}(q_0, \varepsilon)$ // Def. p_0
   $= q_0$ // Def. $\hat{\delta}$ en Ecuación 1.1

2) $z \in L \Rightarrow \hat{\delta}(q_0, z) \in F$ // Def. $\mathscr{L}(M)$ en Definición 1.4
   $\Rightarrow \hat{\delta}(q_0, a_1 \cdots a_m) \in F$ // $z = a_1 \cdots a_m$
   $\Rightarrow p_m \in F$ // Def. p_i para $i = m$

De la Observación 2) se deduce que, para reconocer z, M pasa por cada uno de los estados del conjunto $P = \{p_0, \cdots, p_m\}$. Es decir, M recorre $|P| = m + 1$ estados para reconocer z.

Ahora bien, $m + 1 > m$ y $m \geq n \;\Rightarrow\; m + 1 > n$. Recordando que $n = |Q|$, resulta evidente que si $m + 1 > n$, en el recorrido de M para reconocer z existe al menos un

estado $q_r \in Q$ **visitado más de una vez**. En otras palabras, $\exists\, p_j, p_k \in P \;/\; p_j = p_k$ para $0 \le j < k \le m$ (p_j y p_k están mapeados a un mismo q_r):



Dicho esto continuamos con la última parte del enunciado del Teorema:
"$\exists$ *una descomposición* $z = uvw \;/\; |uv| \le n \;,\; |v| \ge 1 \quad y \quad \forall i \ge 0 : uv^i w \in L$", a lo que nosotros proponemos la siguiente:

$$u = a_1 \cdots a_j \quad , \quad v = a_{j+1} \cdots a_k \quad y \quad w = a_{k+1} \cdots a_m$$

Ahora veamos que se cumplen las tres condiciones para esta descomposición:

1) $\underline{|uv| \le n}$

$|uv| = |a_1 \cdots a_j \; a_{j+1} \cdots a_k| \begin{cases} = n & \text{si } q_r = q_{n-1} \text{ (el estado repetido es el último)} \\ < n & \text{si no} \end{cases}$

$\Rightarrow |uv| \le n \quad \checkmark$

2) $\underline{|v| \ge 1}$

$|v| = |a_{j+1} \cdots a_k| = k - (j+1) + 1 = k - j$

Luego $|v| \ge 1 \iff k - j \ge 1 \iff k - j > 0 \iff k > j \quad \checkmark$

3) $\underline{\forall i \ge 0 \; : \; uv^i w \in L}$

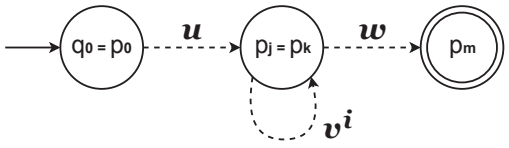
Por la Definición 1.4 de aceptación por AFD, esto es equivalente a comprobar que $\forall i \ge 0 \; : \; \hat{\delta}(q_0, uv^i w) \in F$.

$\hat{\delta}(p_0, u) \overset{\text{Def. } u}{=\!=} \hat{\delta}(p_0, a_1 \cdots a_j) \overset{\text{Obs. 1)}}{=\!=} \hat{\delta}(q_0, a_1 \cdots a_j) \overset{\text{Def. } p_i}{=\!=} p_j$

$\hat{\delta}(p_j, v) = \hat{\delta}(p_j, a_{j+1} \cdots a_k) \quad // \; v = a_{j+1} \cdots a_k$
$= p_k \quad // \; \hat{\delta}(p_0, uv) = \hat{\delta}(q_0, a_1 \cdots a_j \; a_{j+1} \cdots a_k) = p_k \;\Rightarrow\; \hat{\delta}(p_j, a_{j+1} \cdots a_k) = p_k$
$= p_j \quad // \text{ Dicho más arriba}$

Luego, $\hat{\delta}(p_j, v^i) = p_j$.

$\hat{\delta}(p_k, w) = \hat{\delta}(p_k, a_{k+1} \cdots a_m) \quad // \; w = a_{k+1} \cdots a_m$
$= p_m \quad // \; \hat{\delta}(p_0, uvw) = \hat{\delta}(q_0, a_1 \cdots a_j \; a_{j+1} \cdots a_k \; a_{k+1} \cdots a_m) = p_m$
$\Rightarrow \; \hat{\delta}(p_k, a_{k+1} \cdots a_m) = p_m$



$\Rightarrow \hat{\delta}(q_0, uv^i w) = p_m \overset{\text{Obs. 2)}}{\in} F \quad \checkmark$

■

1.3.2. Contrarrecíproco del Pumping Lemma

Tal como mencionamos en la subsección anterior, el Pumping Lemma no nos permite afirmar que un lenguaje dado sea regular. Sin embargo, al ser una condición necesaria que todos los lenguajes regulares han de cumplir, podemos utilizar su contrarrecíproco para probar que cierto lenguaje dado **no** es regular.

La formulación lógica del contrarrecíproco del Pumping Lemma para Lenguajes Regulares puede consultarse detalladamente en uno de los materiales del curso, y en síntesis el mismo establece lo siguiente:

Si L es tal que

$$\forall n \in \mathbb{N} \ : \ \exists z \in L \text{ con } |z| \geq n \text{ para el cual se cumple que}$$

$$\forall \text{ descomposición } z = uvw \text{ en la que } |uv| \leq n \ \text{ y } \ |v| \geq 1 \ : \ \exists i \geq 0 \ / \ uv^i w \notin L$$

entonces L no es un lenguaje regular.

¿Cómo lo usamos? Dado L, definimos $n \in \mathbb{N}$ la constante del Pumping y elegimos $z \in L \ / \ |z| \geq n$. Luego consideramos todas las descomposiciones $z = uvw$ que cumplan $|uv| \leq n$ y $|v| \geq 1$, y en cada una buscamos un $i \geq 0 \ / \ z_i = uv^i w \notin L$.

Ejemplo 1.12 *Decidir si* $L = \{0^k 1^k \ / \ k > 0\}$ *es un lenguaje regular.*

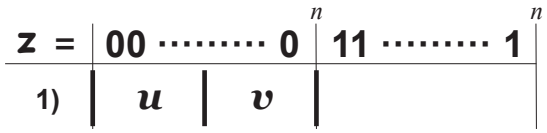
Pensando en un autómata, es evidente que si k *fuera un número fijo fácilmente podemos construir un autómata finito con* $2k+1$ *estados que reconozca el lenguaje. Pero este no es el caso, el* k *de nuestro lenguaje no está acotado superiormente, con lo cual parece imposible poder reconocer todos los casos para cada* $k > 0$ *con una cantidad finita de estados.* ***Cuidado****: que no se nos ocurra cómo construir un autómata finito (o una expresión regular)* ***no*** *es un argumento válido para sostener que el lenguaje no es regular.*

Tenemos la intuición de que este L no sería un lenguaje regular, con lo cual vamos a intentar aplicarle el Contrarrecíproco del Pumping Lemma para Lenguajes Regulares:

Sea $n \in \mathbb{N}$ *la constante del Pumping.*

Elegimos $z = 0^n 1^n \ \Rightarrow \ |z| = n + n = 2n \geq n \quad \checkmark$

Ahora consideramos todas las descomposiciones $z = uvw \ / \ |uv| \leq n \ y \ |v| \geq 1$*:*



Caso 1):

$u = 0^p \qquad p + q \leq n$

$v = 0^q \qquad q \geq 1$

$w = 0^{n-p-q}\ 1^n$

$$\Rightarrow z_i = uv^i w$$
$$= 0^p \ (0^q)^i \ 0^{n-p-q} \ 1^n$$
$$= 0^p \ 0^{qi} \ 0^{n-p-q} \ 1^n$$
$$= 0^{p+qi+n-p-q} \ 1^n$$
$$= 0^{n+q(i-1)} \ 1^n$$

Tomar $i = 2 \Rightarrow z_2 = 0^{n+q(i-1)}\ 1^n\ \big|_{i=2} = 0^{n+q}\ 1^n$.

Entonces $z_2 \in L \iff 0^{\overbrace{n+q}^{k}}\ 1^{\overbrace{n}^{k}} \in L \iff n+q = n \iff q = 0$ ↯ *Esto es absurdo ya que establecimos* $q \geq 1$*, por lo tanto* $z_2 \notin L$.

Estas son todas las descomposiciones $z = uvw$ *que cumplen* $|uv| \leq n$ *y* $|v| \geq 1$*, y en cada una encontramos* $i \geq 0$ / $z_i \notin L$*. Luego, por el Contrarrecíproco del Pumping Lemma para Lenguajes Regulares,* ***L no es regular****.*

Observación: La elección del z condiciona la cantidad de descomposiciones que se deban considerar, así como el éxito que se pueda tener en encontrar para cada una un $z_i \notin L$. Esto quiere decir que hay maneras de elegir el z que pueden hacer más trabajoso el estudio por casos, y hay otras maneras que directamente no sirven porque para algún caso no podemos encontrar un $z_i \notin L$.

La elección de un z adecuado y conveniente para cada lenguaje es una habilidad que se adquiere con la práctica.

1.3.3. Propiedades de Clausura

Teorema 1.14 Los lenguajes regulares son cerrados bajo estas operaciones:

1) **Unión**
$$\left.\begin{array}{l} A \text{ regular} \\ B \text{ regular} \end{array}\right\} \Rightarrow A \cup B \text{ regular}$$

2) **Concatenación**
$$\left.\begin{array}{l} A \text{ regular} \\ B \text{ regular} \end{array}\right\} \Rightarrow A \cdot B \text{ regular}$$

3) **Clausura de Kleene**
$$A \text{ regular} \Rightarrow A^* \text{ regular}$$

4) **Complemento**
$$A \text{ regular} \Rightarrow A^c \text{ regular}$$

5) **Intersección**
$$\left.\begin{array}{l} A \text{ regular} \\ B \text{ regular} \end{array}\right\} \Rightarrow A \cap B \text{ regular}$$

6) **Reverso**
$$A \text{ regular} \Rightarrow A^r \text{ regular}$$

7) **Cociente**
$$\left.\begin{array}{l} R \text{ regular} \\ L \text{ arbitrario} \end{array}\right\} \Rightarrow R \,/\, L \text{ regular}$$

8) **Sustitución**
$$\left.\begin{array}{l} A \text{ regular} \\ f \text{ sustitución} \end{array}\right\} \Rightarrow f(A) \text{ regular}$$

9) **Homomorfismo**
$$\left.\begin{array}{l} A \text{ regular} \\ h \text{ homomorfismo} \end{array}\right\} \Rightarrow h(A) \text{ regular}$$

10) **Homomorfismo Inverso**
$$\left.\begin{array}{l} A \text{ regular} \\ h \text{ homomorfismo} \end{array}\right\} \Rightarrow h^{-1}(A) \text{ regular}$$

Aclaraciones:

$A^r = \{x^r \in \Sigma^* \;/\; x \in L\}$

$x = a_1 \cdots a_{n-1} a_n$

$\Rightarrow\ x^r = a_n a_{n-1} \cdots a_1$

$A \,/\, B = \{x \in \Sigma^* \;/\; \exists y \in B : xy \in A\}$

$\overbrace{\underbrace{x_1 \cdots x_m}_{A \,/\, B}\, \underbrace{y_1 \cdots y_n}_{B}}^{A}$

Una *sustitución* f entre Σ y Δ^* es una función que mapea cada símbolo $a \in \Sigma$ con un lenguaje $L_a \subseteq \Delta^*$. Intuitivamente se extiende para cadenas $x \in \Sigma^*$ de la siguiente manera: $f(\varepsilon) = \varepsilon \wedge f(xa) = f(x)f(a)$. Para un lenguaje A se tiene que $f(A) = \{f(x) \in \Delta^* \,/\, x \in A\}$.

Un *homomorfismo* h es un caso particular de sustitución en el cual cada símbolo $a \in \Sigma$ se mapea con un string $s \in \Delta^*$. Luego $h(A) = \{h(x) \in \Delta^* \,/\, x \in A\}$.

A partir de un homomorfismo h, se define el *homomorfismo inverso* $h^{-1} : \Delta^* \to \Sigma^*$ / $h^{-1}(w) = \{x \in \Sigma^* \,/\, h(x) = w\}$. Luego $h^{-1}(A) = \{x \in \Sigma^* \,/\, h(x) \in A\}$.

Observación: Un homomorfismo h y su inverso asociado h^{-1} **no** son funciones recíprocas, es decir, $h^{-1}\big(h(L)\big) \neq L$ en general.

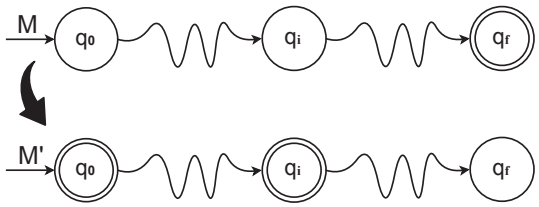
Algunas demostraciones:

1) A *y* B *regulares* $\Rightarrow$ $\exists\, r_a, r_b$ expresiones regulares / $A = \mathscr{L}(r_a) \;\wedge\; B = \mathscr{L}(r_b)$
$\Rightarrow$ $A \cup B = \mathscr{L}(r_a \,|\, r_b)$
$\Rightarrow$ $A \cup B$ *regular*

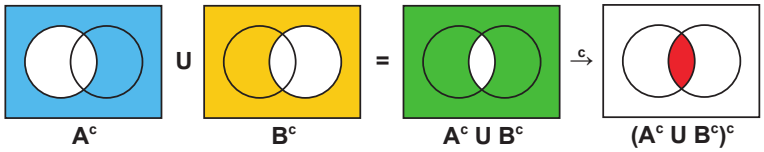
2) A *y* B *regulares* $\Rightarrow$ $\exists\, r_a, r_b$ expresiones regulares / $A = \mathscr{L}(r_a) \;\wedge\; B = \mathscr{L}(r_b)$
$\Rightarrow$ $A \cdot B = \mathscr{L}(r_a \cdot r_b)$
$\Rightarrow$ $A \cdot B$ *regular*

3) A *regular* $\Rightarrow$ $\exists\, r_a$ expresión regular / $A = \mathscr{L}(r_a)$
$\Rightarrow$ $A^* = \mathscr{L}({r_a}^*)$
$\Rightarrow$ A^* *regular*

4) A *regular* $\Rightarrow$ $\exists$ AFD M / $A = \mathscr{L}(M)$
$\Rightarrow$ { Construir M' haciendo que los estados de no aceptación de M sí sean de aceptación en M', y viceversa (se requiere que δ esté totalmente definida).
$\Rightarrow$ $\exists$ AFD M' / $A^c = \mathscr{L}(M')$
$\Rightarrow$ A^c *regular*



5) $A \cap B = (A^c \cup B^c)^c$ $\Rightarrow$ Partiendo de que A y B *son regulares*, aplicamos las Propiedades **1)** y **4)** y obtenemos que $(A^c \cup B^c)^c$ *es regular*.

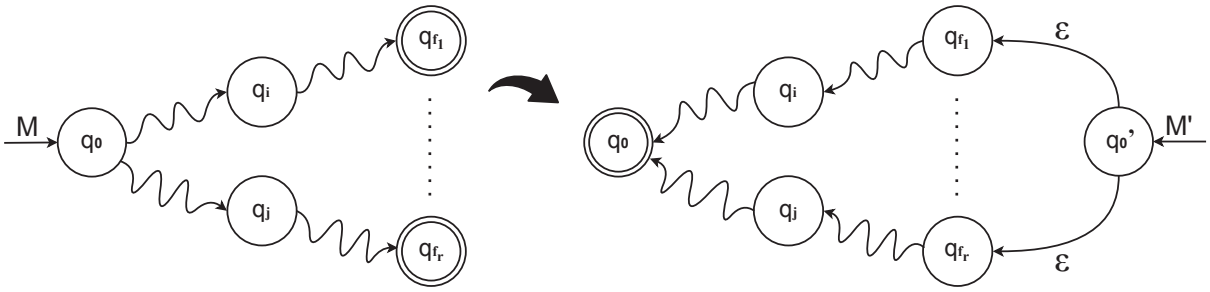


6) A *regular* $\Rightarrow$ $\exists$ AFD M / $A = \mathscr{L}(M)$

$\Rightarrow$ Construir M' revirtiendo la dirección de las transiciones de M, hacer que los estados de aceptación de M sean de no–aceptación en M' agregando $q_0{}'$ inicial con ε–transiciones hacia ellos, y hacer que q_0 sea el estado de aceptación.

$\Rightarrow$ $\exists$ AFND–ε M' / $A^r = \mathscr{L}(M')$

$\Rightarrow$ A^r *regular*



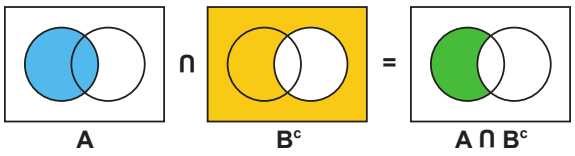
■

Observación: La **Diferencia** también es una propiedad de clausura:

$$\left.\begin{array}{l} A \text{ regular} \\ B \text{ regular} \end{array}\right\} \Rightarrow A \setminus B \text{ regular}$$

Recordar que $A \setminus B = \{x \in \Sigma^* \ / \ x \in A \ \wedge \ x \notin B\}$.

$A \setminus B = A \cap B^c \ \Rightarrow$ Partiendo de que A y B *son regulares*, aplicamos las Propiedades **4)** y **5)** y obtenemos que $A \cap B^c$ *es regular.*



Ejemplo 1.13 *Decidir si* $L = \{(ab)^k c^k \ / \ k > 0\}$ *es un lenguaje regular.*

A primera vista es muy similar al lenguaje del Ejemplo 1.12, por lo tanto sospechamos que no es regular. Una opción sería aplicar el contrarrecíproco del Pumping Lemma a este L, pero este camino tiene la "desventaja" de que requiere más de una descomposición para el $z = (ab)^n c^n$ *: si u es la cadena vacía, si u termina en a, si u termina en b ...*

Otro camino más rápido es definir un homomorfismo h que nos transforme el lenguaje que tenemos en otro más fácil de manejar, ya que como vimos en la Propiedad ***9)*** *del Teorema 1.14, si A es regular* $\Rightarrow h(A)$ *es regular, con lo cual se cumple su contrarrecíproco: si* $h(A)$ ***no*** *es regular* $\Rightarrow$ *A* ***no*** *es regular. Para eso considerar el homomorfismo* $h : \{a, b, c\} \to \{\varepsilon, 0, 1\}$ / $h(a) = \varepsilon \ \wedge \ h(b) = 0 \ \wedge \ h(c) = 1$.

Entonces $h\big((ab)^k c^k\big) = \big(h(ab)\big)^k \big(h(c)\big)^k = \big(h(a)h(b)\big)^k \big(h(c)\big)^k = 0^k 1^k$. *Por lo tanto* $h(L) = \{0^k 1^k \ / \ k > 0\}$, *que ya probamos que no es regular. Luego, L no es un lenguaje regular por el contrarrecíproco de la Propiedad* ***9)****.*

2. Lenguajes Libres de Contexto

2.1. Gramáticas Libres de Contexto

2.1.1. Introducción

Definición 2.1 Una ***gramática*** es una 4-tupla $G = (V, T, P, S)$ donde:

- V es el conjunto de símbolos variables
- T es el conjunto de símbolos terminales
- P es el conjunto de reglas de producción
- $S \in V$ es el símbolo inicial

Definición 2.2 Una ***gramática libre de contexto*** es una gramática en la que todas sus reglas de producción son de la forma

$$A \to \alpha$$

donde $A \in V$ y $\alpha \in (V \cup T)^*$.

Definición 2.3 Una ***derivación*** es una notación que sirve para representar la inferencia de una regla de producción. Dada una gramática libre de contexto $G = (V, T, P, S)$ escribimos

$$\alpha\, A\, \beta \;\Rightarrow\; \alpha\, \gamma\, \beta$$

cuando queremos mostrar que γ es el resultado de aplicar la regla «$A \to \gamma$» $\in P$, con $A \in V$ y $\alpha, \beta, \gamma \in (V \cup T)^*$.

Si queremos indicar que la inferencia se produce en n pasos escribimos

$$\alpha\, A\, \beta \;\overset{n}{\Rightarrow}\; \alpha\, \gamma\, \beta$$

Y en caso de querer representar una cantidad indefinida de pasos cambiamos n por $*$.

Observación: La notación de *derivación* se extiende para gramáticas en general:
Si tenemos la regla «$\alpha \to \beta$» $\in P$ y la secuencia $x\,\alpha\,y$, podemos decir que

$$x\,\alpha\,y \;\Rightarrow\; x\,\beta\,y$$

para $x, y, \alpha, \beta \in (V \cup T)^* \;\wedge\; \alpha \neq \varepsilon$.

Definición 2.4 Dada una gramática $G = (V, T, P, S)$, el ***lenguaje generado por G*** es $\mathscr{L}(G) = \{x \in T^* \;/\; S \overset{*}{\Rightarrow} x\}$. Es decir, $\mathscr{L}(G)$ es el conjunto de secuencias de T^* que se derivan del símbolo inicial S.

Definición 2.5 L es un ***lenguaje libre de contexto*** si existe alguna gramática libre de contexto que lo genera.

Ejemplo 2.1 *En el Ejemplo 1.12 de la sección anterior vimos que el lenguaje* $L = \{0^k\,1^k \;/\; k > 0\}$ *no es regular. Vamos a probar que* L *es un lenguaje libre de contexto construyendo una gramática libre de contexto* $G = (V, T, P, S)$ *que lo genere:*

$$V = \{S\} \qquad T = \{0, 1\} \qquad P = \left\{ \begin{array}{l} S \to 0S1 \\ S \to 01 \end{array} \right\}$$

$$S \;\Rightarrow\; \underbrace{0S1 \;\Rightarrow\; 00S11 \;\Rightarrow\; \cdots \;\Rightarrow\; 0^{k-1}\,S\,1^{k-1}}_{\substack{\textit{Aplicar recursivamente } k-1 \textit{ veces la regla} \\ S \to 0S1}} \;\Rightarrow\; \underbrace{0^{k-1}\,01\,1^{k-1}}_{\substack{\textit{Finalizar con la regla} \\ \textit{para el caso del menor } k}} \;=\; 0^k\,1^k$$

Observación: Si tenemos n reglas de producción que comparten la misma variable A del lado izquierdo:

$$\begin{array}{c} A \to \alpha_1 \\ \vdots \\ A \to \alpha_n \end{array}$$

podemos "compactar" todas ellas en "una sola" regla de la forma: $A \to \alpha_1 \mid \cdots \mid \alpha_n$.

Definición 2.6 Una ***derivación de más a la izquierda*** es una secuencia de derivaciones en la que en cada paso se deriva la variable de más a la izquierda. Análogamente se define el concepto de ***derivación de más a la derecha***.

Ejemplo 2.2 *Construir una gramática libre de contexto para el lenguaje* $L = \{0^k\,1^k\,2^p \;/\; k, p \geq 0\}$.

$$V = \{S, K, P\} \qquad T = \{0, 1, 2\} \qquad P = \left\{ \begin{array}{l} S \to KP \\ K \to 0K1 \mid \varepsilon \\ P \to 2P \mid \varepsilon \end{array} \right\}$$

Si nos preguntamos si la tira 0122 *pertenece al lenguaje generado por la gramática, basta con encontrar alguna derivación que conduzca a ella:*

$S \Rightarrow KP \Rightarrow 0K1P \Rightarrow 0K12P \Rightarrow 0\varepsilon12P = 012P \Rightarrow 0122P \Rightarrow 0122\varepsilon = 0122$ *(Aleatoria)*

$S \Rightarrow KP \Rightarrow 0K1P \Rightarrow 0\varepsilon1P = 01P \Rightarrow 012P \Rightarrow 0122P \Rightarrow 0122\varepsilon = 0122$ *(Izquierda)*

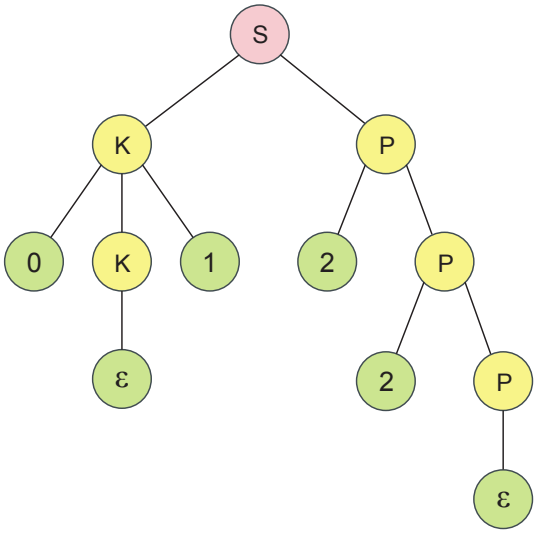
$S \Rightarrow KP \Rightarrow K2P \Rightarrow K22P \Rightarrow K22\varepsilon = K22 \Rightarrow 0K122 \Rightarrow 0\varepsilon122 = 0122$ *(Derecha)*

En general: $\left\{ \begin{array}{lll} S \;\Rightarrow\; KP \overset{k+1}{\Longrightarrow} 0^k\,1^k\,P \overset{p+1}{\Longrightarrow} 0^k\,1^k\,2^p & & \textit{(Izquierda)} \\ S \;\Rightarrow\; KP \overset{p+1}{\Longrightarrow} K\,2^p \overset{k+1}{\Longrightarrow} 0^k\,1^k\,2^p & & \textit{(Derecha)} \end{array} \right.$

Definición 2.7 Dada una gramática libre de contexto $G = (V, T, P, S)$ y una tira $w \in T^*$, un ***árbol de derivación para w*** es un grafo de tipo árbol con las siguientes características:

- El nodo raíz es la variable S
- Todos los nodos interiores son variables de V
- Todas las hojas son símbolos terminales de $T \cup \{\varepsilon\}$
- Para cada nodo interior A con k hijos $x_1, \cdots, x_k$ existe la regla $«A \to x_1 \cdots x_k» \in P$
- La cadena formada por la concatenación de izquierda a derecha de todas las hojas es igual a w

Ejemplo 2.3 *Un árbol de derivación para la tira* 0122 *del Ejemplo 2.2 es*



Como se observa, si concatenamos todas las hojas de izquierda a derecha obtenemos la tira $0\varepsilon122\varepsilon = 0122$ ✓

Claramente existe una fuerte vinculación entre las derivaciones y los árboles de derivación. De hecho, se ha demostrado el Teorema citado a continuación:

Teorema 2.1 Dada una gramática libre de contexto $G = (V, T, P, S)$ y una tira $w \in T^*$

$$S \overset{*}{\Rightarrow} w \iff \exists \text{ árbol de derivación para } w$$

Definición 2.8 Una gramática libre de contexto G se dice ***ambigua***, si existe alguna tira generada por G para la cual podemos obtener dos árboles de derivación distintos, dos derivaciones de más a la izquierda distintas, o bien dos derivaciones de más a la derecha distintas.

2.1.2. Gramáticas Regulares

Teorema 2.2 Todos los Lenguajes Regulares son Libres de Contexto

Sea L un lenguaje regular. Es sabido por la Definición 1.2 que existe una expresión regular $r \;/\; L = \mathscr{L}(r)$, entonces –como ya lo hemos hecho antes– usaremos la definición inductiva de las expresiones regulares para plantear un algoritmo que devuelva una gramática libre de contexto:

Casos base:

Sean r una expresión regular y $G = (V, T, P, S)$ una gramática libre de contexto, tales que $L = \mathscr{L}(r) = \mathscr{L}(G)$. Luego:

- Si $r = \varnothing \;\Rightarrow\; L = \varnothing$, y entonces:
  $V = \{S\}$
  $T = \varnothing$
  $P = \varnothing$
- Si $r = \varepsilon \;\Rightarrow\; L = \{\varepsilon\}$, y entonces:
  $V = \{S\}$
  $T = \varnothing$
  $P = \{\, S \to \varepsilon \,\}$
- Si $r = a \;\Rightarrow\; L = \{a\}$, y entonces:
  $V = \{S\}$
  $T = \{a\}$
  $P = \{\, S \to a \,\}$

Casos inductivos:

Sean r, r_1 y r_2 expresiones regulares, y $G = (V, T, P, S)$, $G_1 = (V_1, T_1, P_1, S_1)$ y $G_2 = (V_2, T_2, P_2, S_2)$ (con $V_1 \cap V_2 = \varnothing$) gramáticas libres de contexto tales que $L = \mathscr{L}(r) = \mathscr{L}(G)$, $L_1 = \mathscr{L}(r_1) = \mathscr{L}(G_1)$ y $L_2 = \mathscr{L}(r_2) = \mathscr{L}(G_2)$. Entonces:

- Si $r = r_1 \,|\, r_2 \;\Rightarrow\; L = L_1 \cup L_2$, y creamos G de esta manera:
  $V = V_1 \cup V_2 \cup \{S\} \qquad T = T_1 \cup T_2 \qquad P = P_1 \cup P_2 \cup \{\, S \to S_1 \mid S_2 \,\}$
- Si $r = r_1 \cdot r_2 \;\Rightarrow\; L = L_1 \cdot L_2$, y creamos G de esta manera:
  $V = V_1 \cup V_2 \cup \{S\} \qquad T = T_1 \cup T_2 \qquad P = P_1 \cup P_2 \cup \{\, S \to S_1 \cdot S_2 \,\}$
- Si $r = {r_1}^* \;\Rightarrow\; L = {L_1}^*$, y creamos G de esta manera:
  $V = V_1 \cup \{S\} \qquad T = T_1 \qquad P = P_1 \cup \{\, S \to S_1 \cdot S \mid \varepsilon \,\}$

■

Definición 2.9 Una ***gramática regular*** es una gramática libre de contexto en la que todas sus reglas de producción son de la forma

$A \to Bw$	$A \to wB$
$A \to w$	$A \to w$
(a) **Gramática Lineal Izquierda**	(b) **Gramática Lineal Derecha**

donde $A, B \in V$ y $w \in (T \cup \{\varepsilon\})$. Podemos decir que una gramática regular es ***extendida*** si permitimos que $w \in T^*$.

Observación: Mezclar reglas de ambos tipos de gramáticas puede generar lenguajes no regulares:

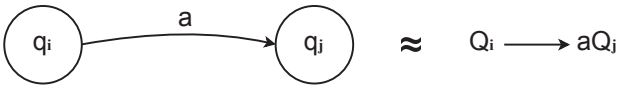
$$V = \{S, B\} \qquad T = \{a, b\} \qquad P = \left\{ \begin{array}{l} S \to aB \mid \varepsilon \\ B \to Sb \end{array} \right\}$$

$$S \Rightarrow aB \Rightarrow aSb \Rightarrow aaBb \Rightarrow aaSbb \Rightarrow \cdots \Rightarrow a^k Sb^k \Rightarrow a^k \varepsilon b^k = a^k b^k$$

Teorema 2.3 Equivalencia entre Autómata Finito y Gramática Regular

Conversión de Autómata Finito en Gramática Regular:

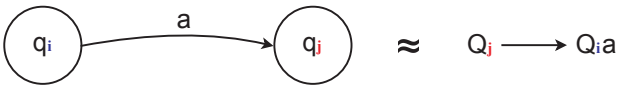
Dado un AFND–ε $M = (Q, \Sigma, \delta, q_0, F)$, la idea de este algoritmo para construir una gramática regular $G = (V, T, P, S)$ radica en asociar cada estado $q_i \in Q$ con una variable $Q_i \in V$, y crear una regla «$Q_i \to aQ_j$» $\in P$ si existe la transición con $a \in \big(\Sigma \cup \{\varepsilon\}\big)$ desde q_i a q_j:



La variable S coincide con la variable Q_0 asociada al estado inicial de M. Los estados de aceptación $q_{f_i} \in F$ determinan las reglas «$Q_{f_i} \to \varepsilon$» $\in P$.

El Algoritmo 4.5 explica este proceso de forma más "algorítmica".

Observar que lo anterior genera una Gramática Lineal *Derecha*. En caso de querer una Gramática Lineal *Izquierda*, la conversión requiere un paso previo: si el autómata tiene varios estados de aceptación, desde cada uno de ellos debemos añadir una ε–transición hacia un nuevo estado de aceptación "definitivo". Luego, la variable S coincide con la variable asociada al único estado de aceptación, el estado inicial determina la regla «$Q_0 \to \varepsilon$», y el resto de las reglas se crean de manera análoga:



Conversión de Gramática Regular en Autómata Finito:

Claramente los pasos a seguir dada una Gramática Lineal *Derecha* son bidireccionales, esto es: Cada variable de V determina un estado en Q, donde la variable S es la asociada a q_0. Una regla «$X \to aY$» $\in P$ para $a \in \big(T \cup \{\varepsilon\}\big)$ implica que $Y \in \delta(X, a)$. Las reglas «$A \to \varepsilon$» $\in P$ indican que el estado $A \in F$.

En caso de partir de una Gramática Lineal *Izquierda*, la variable S está ligada a q_f (el único estado de aceptación), una regla «$X \to Ya$» implica que $X \in \delta(Y, a)$, y en última instancia se crea q_0 (el estado inicial), donde cada regla del tipo «$Q_i \to \varepsilon$» determina una ε–transición desde q_0 hacia q_i.

■

Corolario: El lenguaje generado por cualquier gramática regular es un lenguaje regular, y cualquier lenguaje regular puede ser generado mediante alguna gramática regular.

2.1.3. Simplificación

Definición 2.10 Dada una gramática libre de contexto $G = (V, T, P, S)$, diremos que una variable $A \in V$ es:

$$\begin{aligned} \textbf{\textit{Anulable}}, &\text{ si } A \overset{*}{\Longrightarrow} \varepsilon \\ \textbf{\textit{Positiva}}, &\text{ si } A \overset{*}{\Longrightarrow} x \in T^* \\ \textbf{\textit{Alcanzable}}, &\text{ si } S \overset{*}{\Longrightarrow} \alpha A \beta \\ \textbf{\textit{Útil}}, &\text{ si } S \overset{*}{\Longrightarrow} \alpha A \beta \overset{*}{\Longrightarrow} x \in T^*, \quad \text{con } \alpha, \beta \in (V \cup T)^* \end{aligned}$$

Por otro lado, diremos que una producción de P es ***unitaria*** si es de la forma «$A \to B$» (con $A, B \in V$), y llamaremos ***ε–producción*** a una regla del tipo «$A \to \varepsilon$».

Definición 2.11 Una gramática libre de contexto se dice que está ***simplificada*** cuando no tiene ε–producciones ni producciones unitarias, y todas sus variables son útiles.

Observación: En el concepto de *simplificación* **no** interviene la *cantidad* de reglas de producción, y el mismo **tampoco** está relacionado con el concepto de *ambigüedad* (introducido en la Definición 2.8).

Teorema 2.4 Teorema de Simplificación

Todo lenguaje libre de contexto (no-vacío y que no contiene a ε) puede ser generado con una gramática libre de contexto simplificada de acuerdo a la Definición 2.11 †.

Dada una gramática libre de contexto $G = (V, T, P, S) \mid L = \mathscr{L}(G)$, construimos otra gramática libre de contexto $G' = (V', T', P', S') \mid L = \mathscr{L}(G')$ equivalente y simplificada.

El algoritmo de simplificación consta de cuatro etapas:

1) Eliminación de ε–producciones
2) Eliminación de producciones unitarias
3) Preservación de variables positivas
4) Preservación de variables alcanzables

y puede verse implementado en el Algoritmo 4.6.

† Si el lenguaje es el conjunto vacío, entonces la "mejor" gramática que podemos crear es $G = (\{S\}, \varnothing, \varnothing, S)$, y en este caso la variable S no sería útil. Si el lenguaje contiene a la tira vacía, entonces la "mejor" gramática que podemos crear contendría una única ε–producción que sería «$S \to \varepsilon$».

2.1.4. Formas Normales

Definición 2.12 Una gramática libre de contexto $G = (V, T, P, S)$ está en ***forma normal de Chomsky*** si todas sus reglas de producción son de la forma:

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

con $A, B, C \in V$ y $a \in T$.

Definición 2.13 Una gramática libre de contexto $G = (V, T, P, S)$ está en ***forma normal de Greibach*** si todas sus reglas de producción son de la forma:

$$A \to a\beta$$

con $a \in T$ y $\beta \in V^*$.

Teorema 2.5 Cualquier lenguaje libre de contexto (que no contiene a ε) puede ser generado con una gramática libre de contexto en forma normal de Chomsky y también con una en forma normal de Greibach.

La demostración de este teorema (que aquí no la veremos) consiste en un algoritmo que toma una gramática libre de contexto cualquiera y la transforma en otra equivalente en forma normal. Existe un algoritmo de este tipo tanto para la forma de Chomsky como para la de Greibach.

Como veremos más adelante, la propiedad establecida en el Teorema 2.5 nos permitirá probar resultados importantes acerca de las gramáticas libres de contexto.

2.2. Autómatas Push–Down

2.2.1. Introducción

A lo largo de la Sección 1 presentamos los *autómatas finitos* como máquinas capaces de computar los lenguajes regulares, y más adelante vimos que existen lenguajes que (al no ser regulares) no pueden ser reconocidos con este tipo de máquinas.

La teoría de autómatas nos provee otro tipo de máquinas llamadas *autómatas a pila* (*push–down* en inglés) para computar los lenguajes libres de contexto, que básicamente son AFND–ε que disponen de una pila. Pasamos de tener una memoria *limitada* a la cantidad de estados, a no solo disponer de esto sino que además una cantidad de memoria virtualmente *ilimitada* sujeta a las reglas de uso de una estructura de datos de tipo LIFO.

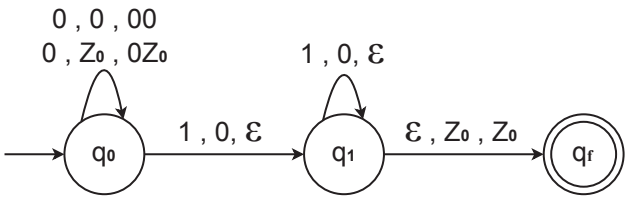
Definición 2.14 Un ***autómata push–down (APD)*** es una 7-tupla $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ donde:

- Q es el conjunto de estados
- Σ es el alfabeto de la entrada
- Γ es el alfabeto de la pila
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to 2^{Q \times \Gamma^*}$ es la función de transición de estados tal que

  $$\delta(q, a, X) = \{(p_1, \alpha_1), \cdots, (p_n, \alpha_n)\} \text{ con } \alpha_i = \begin{cases} RX & \text{para } \textbf{\textit{apilar}}\ R \text{ sobre } X \\ X & \text{para } \textbf{\textit{mantener}}\ X \text{ en el tope} \\ \varepsilon & \text{para } \textbf{\textit{desapilar}}\ X \end{cases}$$

  donde $p_i \in Q$, $a \in (\Sigma \cup \{\varepsilon\})$ y $R, X \in \Gamma$
- $q_0 \in Q$ es el estado inicial
- $Z_0 \in \Gamma$ es el símbolo inicial de la pila
- $F \subseteq Q$ es el conjunto de estados de aceptación

Ejemplo 2.4 *Vamos a construir un APD para el* $L = \{0^k 1^k \ / \ k > 0\}$.



Al igual que en los autómatas vistos anteriormente, colocamos una flecha apuntando al estado inicial y un circulo interior en los estados de aceptación. La notación que usamos para el rótulo de las transiciones es:

`entradaLeída , topeActual , acciónSobreLaPila`

La idea de este APD es la siguiente: El estado q_0 se encarga de apilar los $k \geq 1$ 0s que puedan venir en la entrada, y cuando llega el primer 1 desapila un 0 y pasa al estado q_1, que se encarga de desapilar los restantes $k-1$ 1s que vengan. Si lo leído hasta el momento corresponde a $0^k 1^k$, entonces –por como apilamos y desapilamos– en el tope debe estar Z_0, con lo cual pasamos al estado de aceptación con una ε–transición.

Observar que hay muchas transiciones no definidas, por nombrar algunas:

- $\delta(q_0, 1, Z_0)$ = *¿si lo primero que leo es un* 1*?*
- $\delta(q_1, 0, 0)$ = *¿si leyendo los* 1s *aparece un* 0*?*
- $\delta(q_f, 1, Z_0)$ = *¿si la tira era* $0^k 1^k 1$*?, es decir, llegué a* q_f *pero me falta leer un* 1

Recordar que el resultado de la función δ es un elemento de $2^{Q \times \Gamma^*}$, es decir, un conjunto. Por lo tanto, en todos estos casos que no escribimos, se dará por hecho que ***cualquier transición no dibujada significa que el resultado de δ es el conjunto vacío*** *(el autómata "se tranca" y no acepta la tira dada como entrada).*

2.2.2. Reconocimiento: Estado Final vs Pila Vacía

Definición 2.15 Sea $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ un APD. Una ***descripción instantánea*** es una terna (q, w, α) donde:

- $q \in Q$ es el estado actual
- $w \in \Sigma^*$ es lo que queda por leer de la entrada
- $\alpha \in \Gamma^*$ es el contenido actual de la pila

Convencionalmente escribimos en el extremo izquierdo de α el tope de la pila (y la parte inferior de esta hacia el extremo derecho).

Conjuntamente definimos la notación $\vdash$ para mostrar el cambio de una descripción instantánea en otra: Si $(p, \alpha) \in \delta(q, a, X)$ entonces

$$(q, aw, X\beta) \vdash (p, w, \alpha\beta)$$

representa la idea de que M se mueve del estado q al p con la entrada a cuando el tope es X, y reemplaza X en el tope de la pila por α.

También podemos usar la notación $\vdash^n$ o $\vdash^*$ para representar que el cambio ocurre en n o una cantidad indefinida de pasos, respectivamente.

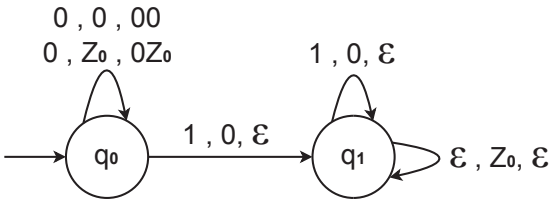
Definición 2.16 Dado un APD $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, el ***lenguaje aceptado por M por estado final*** es

$$\mathscr{L}(M) = \left\{ x \in \Sigma^* \;/\; (q_0, x, Z_0) \vdash^* (q_f, \varepsilon, \alpha) \text{ con } q_f \in F \;\wedge\; \alpha \in \Gamma^* \right\}$$

Definición 2.17 Dado un APD $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, \varnothing)$, el ***lenguaje aceptado por M por pila vacía*** es

$$\mathscr{L}(M) = \left\{ x \in \Sigma^* \ / \ (q_0, x, Z_0) \overset{*}{\vdash} (q, \varepsilon, \varepsilon) \text{ con } q \in Q \right\}$$

Ejemplo 2.5 *APD equivalente al del Ejemplo 2.4 pero que reconoce por pila vacía:*



Definición 2.18 Diremos que un APD $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ es ***determinista*** si cumple con las siguientes dos condiciones:

$$\forall q \in Q \ , \ \forall a \in (\Sigma \cup \{\varepsilon\}) \ , \ \forall X \in \Gamma \ : \ |\delta(q, a, X)| \leq 1$$
$$\forall q \in Q \ , \ \forall X \in \Gamma \ : \ \textbf{Si } |\delta(q, \varepsilon, X)| = 1 \textbf{ entonces } \forall b \in \Sigma \ : \ \delta(q, b, X) = \varnothing$$

Observación: La primera condición exige que al computar δ, el autómata solo tenga una opción como máximo hacia dónde dirigirse y qué hacer con la pila. La segunda condición establece que en caso de que un estado tenga una ε–transición para el tope X, entonces ese estado no puede tener otra transición para el mismo tope X, independientemente de la entrada.

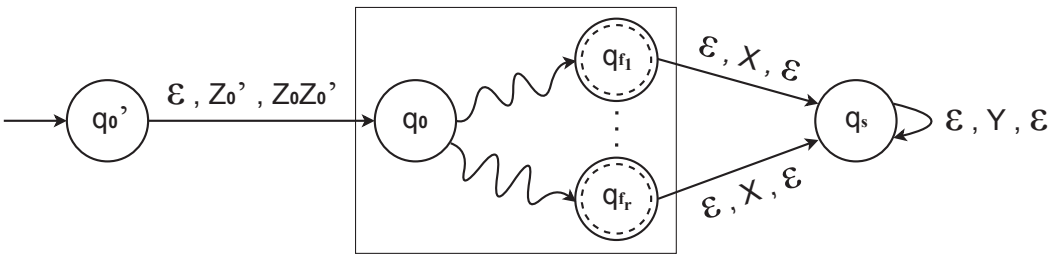
Por lo tanto, un APD con ε–transiciones puede ser determinista, y en tal caso el determinismo vendrá dado por el tope de la pila.

Teorema 2.6 Equivalencia entre aceptación por Estado Final y por Pila Vacía

Conversión de Estado Final a Pila Vacía:

Dado un APD de estado final $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F) \ / \ L = \mathscr{L}(M)$, creamos un APD de pila vacía $M' = (Q', \Sigma', \Gamma', \delta', q_0', Z_0', \varnothing) \ / \ L = \mathscr{L}(M')$ siguiendo estas reglas:

1) $Q' = Q \cup \{q_0', q_s\}$
2) $\Sigma' = \Sigma$
3) $\Gamma' = \Gamma \cup \{Z_0'\}$
4)
$$\begin{cases} \delta'(q_0', \varepsilon, Z_0') = \{(q_0, Z_0 Z_0')\} & \\ \delta'(q, a, X) = \delta(q, a, X) & \forall q \in Q \ , \ \forall a \in (\Sigma \cup \{\varepsilon\}) \ , \ \forall X \in \Gamma \\ \delta'(q_f, \varepsilon, X) = \{(q_s, \varepsilon)\} & \forall q_f \in F \ , \ \forall X \in \Gamma \\ \delta'(q_s, \varepsilon, Y) = \{(q_s, \varepsilon)\} & \forall Y \in \Gamma' \end{cases}$$



Correctitud: Sea $x \in \mathscr{L}(M)$ por estado final

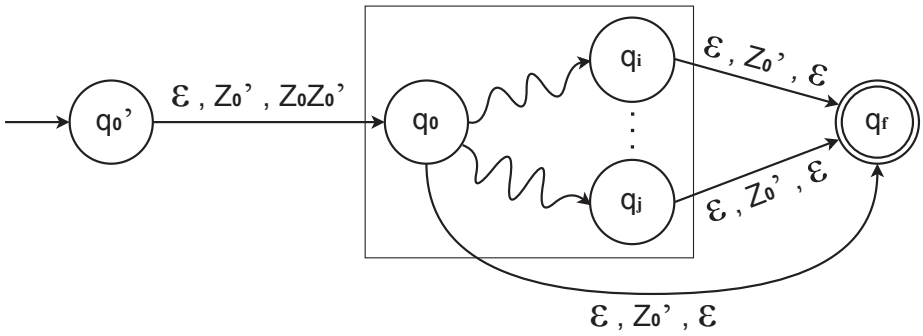
$$
\begin{aligned}
(q_0', x, Z_0') &\vdash (q_0, x, Z_0 Z_0') && // \ (q_0', Z_0 Z_0') \in \delta'(q_0', \varepsilon, Z_0') \text{ según Regla 4)} \\
&\vdash^* (q_f, \varepsilon, XY) && // \text{ Cuando } \delta'(q,a,X) = \delta(q,a,X), \text{ desde } q_0 \text{ se llega hasta algún } q_f \\
&\vdash (q_s, \varepsilon, Y) && // \ (q_s, \varepsilon) \in \delta'(q_f, \varepsilon, X) \text{ según Regla 4)} \\
&\vdash^* (q_s, \varepsilon, \varepsilon) && // \ (q_s, \varepsilon) \in \delta'(q_s, \varepsilon, Y) \text{ según Regla 4) (reiteradamente)} \\
&\Rightarrow x \in \mathscr{L}(M') && // \text{ Def. aceptación por pila vacía en Definición 2.17}
\end{aligned}
$$

Lo anterior demuestra que $\mathscr{L}(M) \subseteq \mathscr{L}(M')$. La inclusión $\mathscr{L}(M') \subseteq \mathscr{L}(M)$ se comprueba trivialmente ya que por nuestra construcción del δ los pasos anteriores son bidireccionales. Luego, $\mathscr{L}(M) = \mathscr{L}(M')$.

Conversión de Pila Vacía a Estado Final:

Dado un APD de pila vacía $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, \varnothing) \ / \ L = \mathscr{L}(M)$, se obtiene un APD de estado final $M' = (Q', \Sigma', \Gamma', \delta', q_0', Z_0', F') \ / \ L = \mathscr{L}(M')$ aplicando estas reglas:

1) $Q' = Q \cup \{q_0', q_f\}$
2) $\Sigma' = \Sigma$
3) $\Gamma' = \Gamma \cup \{Z_0'\}$
4) $F' = \{q_f\}$
5) $$\begin{cases} \delta'(q_0', \varepsilon, Z_0') &= \{(q_0, Z_0 Z_0')\} & \\ \delta'(q, a, X) &= \delta(q, a, X) & \forall q \in Q \ , \ \forall a \in (\Sigma \cup \{\varepsilon\}) \ , \ \forall X \in \Gamma \\ \delta'(q, \varepsilon, Z_0') &= \{(q_f, \varepsilon)\} & \forall q \in Q \end{cases}$$



Correctitud: Sea $x \in \mathscr{L}(M)$ por pila vacía

$$
\begin{aligned}
(q_0', x, Z_0') &\vdash (q_0, x, Z_0 Z_0') && // \ (q_0', Z_0 Z_0') \in \delta'(q_0', \varepsilon, Z_0') \text{ según Regla 5)} \\
&\vdash^* (q_i, \varepsilon, Z_0') && // \text{ Cuando } \delta'(q,a,X) = \delta(q,a,X), \text{ desde } q_0 \text{ se llega hasta algún } q_i \text{ que permite vaciar la pila} \\
&\vdash (q_f, \varepsilon, \varepsilon) && // \ (q_f, \varepsilon) \in \delta'(q_i, \varepsilon, Z_0') \text{ según Regla 5)} \\
&\Rightarrow x \in \mathscr{L}(M') && // \text{ Def. aceptación por estado final en Definición 2.16}
\end{aligned}
$$

Lo anterior demuestra que $\mathscr{L}(M) \subseteq \mathscr{L}(M')$. La inclusión $\mathscr{L}(M') \subseteq \mathscr{L}(M)$ se comprueba trivialmente ya que por nuestra construcción del δ los pasos anteriores son bidireccionales. Luego, $\mathscr{L}(M) = \mathscr{L}(M')$.

■

2.2.3. Equivalencia entre Gramáticas Libres de Contexto y Autómatas Push–Down

Como adelantábamos al comienzo de esta Subsección, los APD son las máquinas capaces de reconocer a todos los lenguajes libres de contexto. Con el Teorema presentado a continuación formalizaremos esta aseveración:

Teorema 2.7 Equivalencia entre Gramáticas Libres de Contexto y Autómatas Push–Down

Conversión Gramática Libre de Contexto en APD:

Dada una gramática libre de contexto $G = (V, T, P, S) \ / \ L = \mathscr{L}(G)$ en forma normal de Greibach, construimos un APD $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, \varnothing) \ / \ L = \mathscr{L}(M)$ de pila vacía de acuerdo a estas reglas:

1) $Q = \{q_0\}$
2) $\Sigma = T$
3) $\Gamma = V \cup \{Z_0\}$
4) $\begin{cases} \delta(q_0, \varepsilon, Z_0) = \{(q_0, SZ_0), (q_0, \varepsilon)\} \\ \textbf{Si } «A \to a\beta» \in P \Rightarrow (q_0, \beta) \in \delta(q_0, a, A) \end{cases}$

Conversión APD en Gramática Libre de Contexto:

Dado un un APD $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, \varnothing) \ / \ L = \mathscr{L}(M)$ de pila vacía, creamos una gramática libre de contexto $G = (V, T, P, S) \ / \ L = \mathscr{L}(G)$ de acuerdo a estas reglas:

1) $V = \{S\} \cup \{[pAq] \ / \ p, q \in Q \ \wedge \ A \in \Gamma\}$
2) $T = \Sigma$
3) $\begin{cases} «S \to [q_0 Z_0 p]» \in P & \forall p \in Q \\ \textbf{Si } (p, BA) \in \delta(q, a, A) & \Rightarrow «[qAq_j] \to a[pBq_k][q_k Aq_j]» \in P \quad \forall q_j, q_k \in Q \\ \textbf{Si } (p, a) \in \delta(q, a, A) & \Rightarrow «[qAq_j] \to a[pBq_j]» \in P \quad \forall q_j \in Q \\ \textbf{Si } (p, \varepsilon) \in \delta(q, a, A) & \Rightarrow «[qAp] \to a» \in P \end{cases}$

Observación: En la primera parte exigimos que la gramática que nos interesa convertir esté en forma normal de Greibach, por lo tanto en caso de que esta no cumpla con dicho formato simplemente aplicamos el algoritmo de conversión cuya existencia mencionamos en el Teorema 2.5. La razón por la cual pedimos esto es que con el autómata buscamos simular derivaciones (entonces nos interesaría poder optimizar las transiciones), y las gramáticas en forma normal de Greibach tienen la propiedad de que la cantidad de derivaciones necesarias para generar una palabra del lenguaje es exactamente el largo de la palabra.

En la segunda parte exigimos que el autómata reconozca por pila vacía, y en caso de que este no cumpla con tal condición solamente tenemos que convertirlo con el Teorema 2.6. Observar que el resultado es una gramática en forma normal de Greibach.

Definición 2.19 L es un ***lenguaje libre de contexto*** si es aceptado por algún APD, ya sea por estado final o por pila vacía.

2.3. Propiedades de los Lenguajes Libres de Contexto

2.3.1. Pumping Lemma para Lenguajes Libres de Contexto

El ***Pumping Lemma*** para lenguajes libres de contexto es una generalización del Teorema 1.13 que vimos para lenguajes regulares. Es decir, es una propiedad fundamental que cumplen todos los lenguajes libres de contexto, pero **no** es el tipo de propiedad que podamos emplear para justificar que un lenguaje efectivamente lo sea (recordar que el Pumping Lemma define una **condición necesaria pero no suficiente** para los lenguajes libres de contexto).

Teorema 2.8 Pumping Lemma para Lenguajes Libres de Contexto

Si L *es un lenguaje libre de contexto* **entonces**

$$\exists n \in \mathbb{N} \ / \ \forall z \in L \text{ con } |z| \geq n \text{ se cumple que}$$

$$\exists \text{ una descomposición } z = uvwxy \ / \ |vwx| \leq n \ , \ |vx| \geq 1 \quad y \quad \forall i \geq 0 : uv^i wx^i y \in L$$

Demostración:

Primero veamos unas cuestiones preliminares que usaremos más adelante:

Definición: La *altura* de un árbol se define como la cantidad de aristas del camino más largo que empieza en su raíz y termina en una hoja.

Propiedad: **Si** T es un árbol binario de altura h **entonces** $\#\texttt{hojas}(T) \leq 2^h$.

Observación: La cantidad de hojas en un árbol de derivación coincide con el largo de la tira derivada.

Ahora sí vamos de lleno a la demostración:

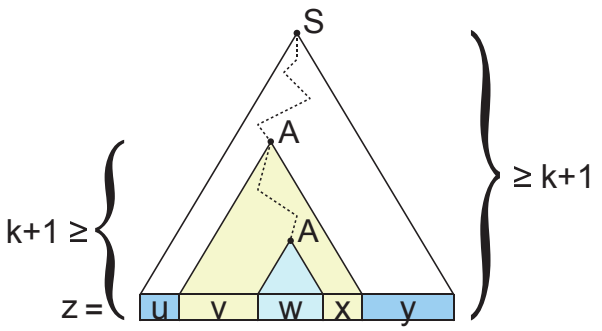
Por hipótesis sabemos que L es un lenguaje libre de contexto, por lo tanto –según el Teorema 2.5– existe una gramática libre de contexto $G = (V, T, P, S) \ / \ L = \mathscr{L}(G)$ que puede ser expresada en forma normal de Chomsky.

El Teorema lo primero que dice es que "$\exists n \in \mathbb{N} \ / \ \cdots$ ". Si nuestra gramática tiene k variables ($k = |V|$), entonces nosotros proponemos que dicho n sea exactamente 2^{k+1}.

Lo que sigue es un "$\forall z \in L \text{ con } |z| \geq n \cdots$", entonces antes de decir las descomposiciones examinemos qué sucede con un z en esas condiciones:

Si $z \in L$, entonces por la Definición 2.4 $S \overset{*}{\Longrightarrow} z$. Ahora bien, según el Teorema 2.1 existe un árbol de derivación para z, y como la gramática está en forma normal de Chomsky, este será un árbol binario de al menos 2^{k+1} hojas. Para lograr esto es suficiente que dicho árbol tenga una altura mayor o igual a $k+1$. Es decir, el camino más largo desde la raíz S hasta una hoja con un terminal tendrá $k+1$ aristas, y por lo tanto constará de al menos $k+2$ nodos. Como hemos dicho, al final del camino se encuentra un terminal, entonces tendremos al menos $k+1$ nodos variables. Recordar que $k = |V|$, con lo cual en este camino necesariamente debe existir al menos una variable $A \in V$ **repetida**.

Dicho esto continuamos con la última parte del enunciado del Teorema:
"$\exists$ *una descomposición* $z = uvwxy \ / \ |vwx| \leq n \ , \ |vx| \geq 1 \quad y \quad \forall i \geq 0 \ : \ uv^i wx^i y \in L$", a lo que nosotros proponemos la siguiente:



$S \overset{*}{\Rightarrow} uAy$

$A \overset{*}{\Rightarrow} vAx$

$A \overset{*}{\Rightarrow} w$

Ahora veamos que se cumplen las tres condiciones para esta descomposición:

1) $\underline{|vwx| \leq n}$

Considerar el subárbol con raíz en la primera A. La altura de este árbol es menor o igual a $k+1$, por lo tanto la cantidad de hojas que desprende está acotada por 2^{k+1}, y puesto que las hojas son lo que forman el substring vwx, tendremos que el largo de vwx estará acotado por $2^{k+1} = n$.

2) $\underline{|vx| \geq 1}$

$|vx| \geq 1 \iff v \neq \varepsilon \ \vee \ x \neq \varepsilon$

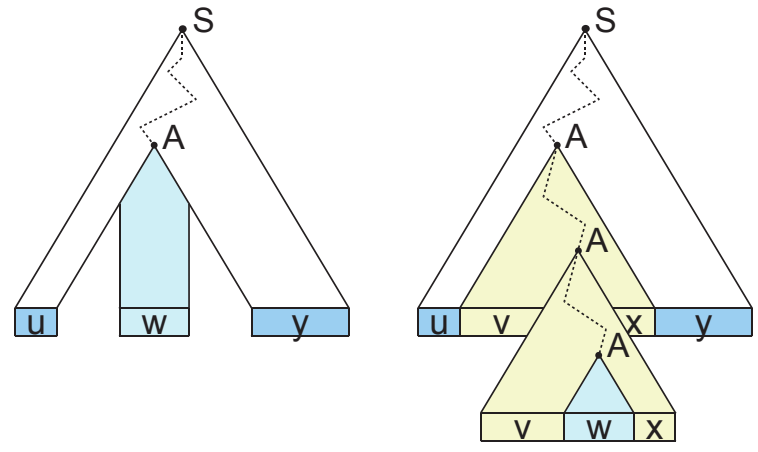
Suponer $v = \varepsilon \ \wedge \ x = \varepsilon$. Entonces como $A \overset{*}{\Rightarrow} vAx$ obtenemos que $A \overset{*}{\Rightarrow} A$, o sea que G no está en forma normal de Chomsky. ↯ Absurdo.
Luego, $v \neq \varepsilon \ \vee \ x \neq \varepsilon$.

3) $\underline{\forall i \geq 0 \ : \ uv^i wx^i y \in L}$

Según la Definición 2.4 esto es equivalente a comprobar que $S \overset{*}{\Rightarrow} uv^i wx^i y$:

$$S \overset{*}{\Rightarrow} uAy \overset{*}{\Rightarrow} uvAxy \overset{*}{\Rightarrow} uvvAxxy \overset{*}{\Rightarrow} \cdots \overset{*}{\Rightarrow} uv^i Ax^i y \overset{*}{\Rightarrow} uv^i wx^i y \quad \checkmark \ (i > 0)$$

$$S \overset{*}{\Rightarrow} uAy \overset{*}{\Rightarrow} uwy \quad \checkmark \ (i = 0)$$



Generación de uv^0wx^0y Generación de uv^2wx^2y

■

Las imágenes mostradas en esta página son una adaptación de las creadas por Jochen Burghardt para Wikimedia Commons bajo la licencia CC BY–SA 3.0.

2.3.2. Contrarrecíproco del Pumping Lemma

Como ya hemos mencionado antes, el Pumping Lemma no nos permite afirmar que un lenguaje dado sea libre de contexto. Sin embargo, al ser una condición necesaria que todos los lenguajes libres de contexto han de cumplir, podemos utilizar su contrarrecíproco para probar que cierto lenguaje dado **no** es libre de contexto.

La formulación del contrarrecíproco del Pumping Lemma para Lenguajes Libres de Contexto dice lo siguiente:

Si L es tal que

$$\forall n \in \mathbb{N} \ : \ \exists z \in L \text{ con } |z| \geq n \text{ para el cual se cumple que}$$

$$\forall \text{ descomposición } z = uvwxy \text{ en la que } |vwx| \leq n \ \text{ y } \ |vx| \geq 1 \ : \ \exists i \geq 0 \ / \ uv^i wx^i y \notin L$$

entonces L no es un lenguaje libre de contexto.

¿Cómo lo usamos? Dado L, definimos $n \in \mathbb{N}$ la constante del Pumping y elegimos $z \in L \ / \ |z| \geq n$. Luego consideramos todas las descomposiciones $z = uvwxy$ que cumplan $|vwx| \leq n$ y $|vx| \geq 1$, y en cada una buscamos un $i \geq 0 \ / \ z_i = uv^i wx^i y \notin L$.

Ejemplo 2.6 *Decidir si* $L = \{0^k 1^k 2^k \ / \ k > 0\}$ *es un lenguaje libre de contexto.*

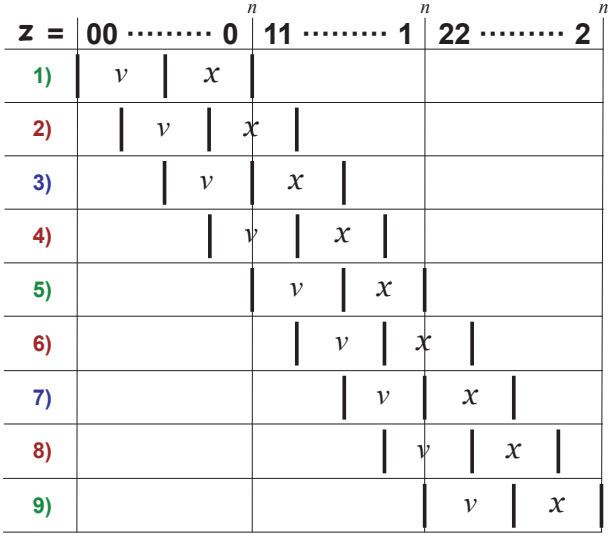
Pensando en un APD, podemos comprobar que la cantidad de 0*s sea igual a la de* 1*s llenando la pila con* 0*s y luego vaciándola a medida que leemos los* 1*s. Pero para cuando lleguen los* 2*s nos vamos a encontrar que la pila está vacía, con lo cual habremos perdido el rastro de cuantos* 0*s o* 1*s leímos.*

Entonces tenemos la intuición de que este L *no sería un lenguaje libre de contexto, con lo cual vamos a intentar aplicarle el Contrarrecíproco del Pumping Lemma para Lenguajes Libres de Contexto:*

Sea $n \in \mathbb{N}$ *la constante del Pumping.*

Elegimos $z = 0^n 1^n 2^n \ \Rightarrow \ |z| = n + n + n = 3n \geq n \quad \checkmark$

Ahora consideramos todas las descomposiciones $z = uvwxy \ / \ |vwx| \leq n \ y \ |vx| \geq 1$*:*



Caso 1):

$u = 0^p$ $\quad q + r + s \leq n$ $\quad \Rightarrow z_i = uv^iwx^iy$

$v = 0^q$ $\quad q + s \geq 1$ $\quad = 0^p\ (0^q)^i\ 0^r\ (0^s)^i\ 0^{n-p-q-r-s}\ 1^n\ 2^n$

$w = 0^r$ $\quad = 0^p\ 0^{qi}\ 0^r\ 0^{si}\ 0^{n-p-q-r-s}\ 1^n\ 2^n$

$x = 0^s$ $\quad = 0^{p+qi+r+si+n-p-q-r-s}\ 1^n\ 2^n$

$y = 0^{n-p-q-r-s}\ 1^n\ 2^n$ $\quad = 0^{n+q(i-1)+s(i-1)}\ 1^n\ 2^n$

Tomar $i = 2 \Rightarrow z_2 = 0^{n+q(i-1)+s(i-1)}\ 1^n\ 2^n\ |_{i=2} = 0^{n+q+s}\ 1^n\ 2^n$

Entonces $z_2 \in L \iff 0^{\overbrace{n+q+s}^{k}}\ 1^{\sqcap^k_n}\ 2^{\sqcap^k_n} \in L \iff n + q + s = n \iff q + s = 0$ ↯

Esto es absurdo ya que establecimos $q + s \geq 1$*, por lo tanto* $z_2 \notin L$*.*

Caso 5): *Es análogo al* ***Caso 1****, ya que* $z_i = 0^n\ 1^{n+q(i-1)+s(i-1)}\ 2^n$ *entonces tomando* $i = 2$ *se tiene que* $z_2 = 0^{\sqcap^k_n}\ 1^{\overbrace{n+q+s}^{k}}\ 2^{\sqcap^k_n} \notin L$ *dado que* $q + s \geq 1$*.*

Caso 9): *Es análogo al* ***Caso 1****, ya que* $z_i = 0^n\ 1^n\ 2^{n+q(i-1)+s(i-1)}$ *entonces tomando* $i = 2$ *se tiene que* $z_2 = 0^{\sqcap^k_n}\ 1^{\sqcap^k_n}\ 2^{\overbrace{n+q+s}^{k}} \notin L$ *dado que* $q + s \geq 1$*.*

Caso 2):

$u = 0^{n-p-q-r}$ $\quad p + q + r + s \leq n$ $\quad \Rightarrow z_i = uv^iwx^iy$

$v = 0^p$ $\quad p + r + s \geq 1$ $\quad = 0^{n-p-q-r}\ (0^p)^i\ 0^q\ (0^r\ 1^s)^i\ 1^{n-s}\ 2^n$

$w = 0^q$ $\quad = 0^{n-p-q-r}\ 0^{pi}\ 0^q\ (0^r\ 1^s)^i\ 1^{n-s}\ 2^n$

$x = 0^r\ 1^s$ $\quad r > 0$ *(*$r = 0$ *está en Caso 3)* $\quad = 0^{n-p-q-r+pi+q}\ (0^r\ 1^s)^i\ 1^{n-s}\ 2^n$

$y = 1^{n-s}\ 2^n$ $\quad s > 0$ *(*$s = 0$ *está en Caso 1)* $\quad = 0^{n+p(i-1)-r}\ (0^r\ 1^s)^i\ 1^{n-s}\ 2^n$

Tomar $i = 2 \Rightarrow z_2 = 0^{n+p(i-1)-r}\ (0^r\ 1^s)^i\ 1^{n-s}\ 2^n\ |_{i=2}$

$= 0^{n+p-r}\ (0^r\ 1^s)^2\ 1^{n-s}\ 2^n$

$= 0^{n+p-r}\ 0^r\ 1^s\ 0^r\ 1^s\ 1^{n-s}\ 2^n$

$= 0^{n+p-r+r}\ 1^s\ 0^r\ 1^{s+n-s}\ 2^n$

$= 0^{n+p}\ 1^s\ 0^r\ 1^n\ 2^n$

Dado que establecimos $r > 0$ *y* $s > 0$*, en* z_2 *se mezclan* 0*s con* 1*s (a causa de haber bombeado* x*), por lo tanto* z_2 *no es de la forma* $0^k\,1^k\,2^k$*, con lo cual* $z_2 \notin L$*.*

Caso 4): *Es análogo al* ***Caso 2****, ya que en* z_2 *se mezclan* 0*s con* 1*s (a causa de bombear* v*), con lo cual* $z_2 \notin L$*.*

Caso 6): *Es análogo al* ***Caso 2****, ya que en* z_2 *se mezclan* 1*s con* 2*s (a causa de bombear* x*), con lo cual* $z_2 \notin L$*.*

Caso 8): *Es análogo al* ***Caso 2****, ya que en* z_2 *se mezclan* 1*s con* 2*s (a causa de bombear* v*), con lo cual* $z_2 \notin L$*.*

Caso 3):

$u = 0^{n-p-q}$ $\quad p+q+r+s \leq n$ $\quad \Rightarrow z_i = uv^iwx^iy$

$v = 0^p$ $\quad p+s \geq 1$ $\quad = 0^{n-p-q}\ (0^p)^i\ 0^q\ 1^r\ (1^s)^i\ 1^{n-r-s}\ 2^n$

$w = 0^q\ 1^r$ $\quad = 0^{n-p-q}\ 0^{pi}\ 0^q\ 1^r\ 1^{si}\ 1^{n-r-s}\ 2^n$

$x = 1^s$ $\quad = 0^{n-p-q+pi+q}\ 1^{r+si+n-r-s}\ 2^n$

$y = 1^{n-r-s}\ 2^n$ $\quad = 0^{n+p(i-1)}\ 1^{n+s(i-1)}\ 2^n$

Tomar $i = 2 \Rightarrow z_2 = 0^{n+p(i-1)}\ 1^{n+s(i-1)}\ 2^n\ |_{i=2} = 0^{n+p}\ 1^{n+s}\ 2^n$

$$p+s \geq 1 \Rightarrow \textbf{\textit{Si}} \begin{cases} p = 0 \Rightarrow s \geq 1 \Rightarrow |z_2|_1 > |z_2|_2 \Rightarrow z_2 \notin L \\ p \neq 0 \Rightarrow p \geq 1 \Rightarrow |z_2|_0 > |z_2|_2 \Rightarrow z_2 \notin L \end{cases}$$

Caso 7): *Es análogo al* ***Caso 3****, ya que* $z_i = 0^n\ 1^{n+p(i-1)}\ 2^{n+s(i-1)}$ *entonces tomando* $i = 2$ *se tiene que* $z_2 = 0^n\ 1^{n+p}\ 2^{n+s}$*. Luego:*

$$p+s \geq 1 \Rightarrow \textbf{\textit{Si}} \begin{cases} p = 0 \Rightarrow s \geq 1 \Rightarrow |z_2|_2 > |z_2|_0 \Rightarrow z_2 \notin L \\ p \neq 0 \Rightarrow p \geq 1 \Rightarrow |z_2|_1 > |z_2|_0 \Rightarrow z_2 \notin L \end{cases}$$

Estas son todas las descomposiciones $z = uvwxy$ *que cumplen* $|vwx| \leq n$ *y* $|vx| \geq 1$*, y en cada una encontramos* $i \geq 0$ */* $z_i \notin L$*. Luego, por el Contrarrecíproco del Pumping Lemma para Lenguajes Libres de Contexto,* ***L no es libre de contexto****.*

2.3.3. Propiedades de Clausura

Teorema 2.9 Los lenguajes libres de contexto son cerrados bajo las siguientes operaciones:

1) Unión

$$\left.\begin{array}{l} A \text{ libre de contexto} \\ B \text{ libre de contexto} \end{array}\right\} \Rightarrow \underset{\text{libre de contexto}}{A \cup B}$$

2) Concatenación

$$\left.\begin{array}{l} A \text{ libre de contexto} \\ B \text{ libre de contexto} \end{array}\right\} \Rightarrow \underset{\text{libre de contexto}}{A \cdot B}$$

3) Clausura de Kleene

A *libre de contexto* $\Rightarrow A^*$ *libre de contexto*

4) Reverso

A *libre de contexto* $\Rightarrow A^r$ *libre de contexto*

5) Sustitución

$$\left.\begin{array}{l} A \text{ libre de contexto} \\ f \text{ sustitución} \end{array}\right\} \Rightarrow \underset{\text{libre de contexto}}{f(A)}$$

6) Homomorfismo

$$\left.\begin{array}{l} A \text{ libre de contexto} \\ h \text{ homomorfismo} \end{array}\right\} \Rightarrow \underset{\text{libre de contexto}}{h(A)}$$

7) Homomorfismo Inverso

$$\left.\begin{array}{l} A \text{ libre de contexto} \\ h \text{ homomorfismo} \end{array}\right\} \Rightarrow \underset{\text{libre de contexto}}{h^{-1}(A)}$$

Algunas demostraciones:

1) Sean A y B *libres de contexto* $\Rightarrow \exists G_A = (V_A, T_A, P_A, S_A)$ $\exists G_B = (V_B, T_B, P_B, S_B)$ (con $V_A \cap V_B = \varnothing$) gramáticas libres de contexto / $A = \mathscr{L}(G_A)$ y $B = \mathscr{L}(G_B)$. Luego, de acuerdo a las siguientes especificaciones, creamos una nueva gramática $G = (V, T, P, S)$ / $A \cup B = \mathscr{L}(G)$:

$$V = V_A \cup V_B \cup \{S\} \qquad T = T_A \cup T_B \qquad P = P_A \cup P_B \cup \{\, S \to S_A \mid S_B \,\}$$

2) Sean A y B *libres de contexto* $\Rightarrow \exists G_A = (V_A, T_A, P_A, S_A)$ $\exists G_B = (V_B, T_B, P_B, S_B)$ (con $V_A \cap V_B = \varnothing$) gramáticas libres de contexto / $A = \mathscr{L}(G_A)$ y $B = \mathscr{L}(G_B)$. Luego, de acuerdo a las siguientes especificaciones, creamos una nueva gramática $G = (V, T, P, S)$ / $A \cup B = \mathscr{L}(G)$:

$$V = V_A \cup V_B \cup \{S\} \qquad T = T_A \cup T_B \qquad P = P_A \cup P_B \cup \{\, S \to S_A \cdot S_B \,\}$$

3) Sea A *libre de contexto* $\Rightarrow \exists G_A = (V_A, T_A, P_A, S_A)$ gramática libre de contexto tal que $A = \mathscr{L}(G_A)$. Luego, de acuerdo a las siguientes especificaciones, creamos una nueva gramática $G = (V, T, P, S)$ / $A^* = \mathscr{L}(G)$:

$$V = V_A \cup \{S\} \qquad T = T_A \qquad P = P_A \cup \{\, S \to S_A \cdot S \mid \varepsilon \,\}$$

4) Sea A *libre de contexto* $\Rightarrow \exists G_A = (V_A, T_A, P_A, S_A)$ gramática libre de contexto tal que $A = \mathscr{L}(G_A)$. Luego, de acuerdo a las siguientes especificaciones, creamos una nueva gramática $G = (V, T, P, S)$ / $A^r = \mathscr{L}(G)$:

$$V = V_A \qquad T = T_A \qquad P = \{\, A \to \alpha^r \;/\; «A \to \alpha» \in P_A \} \qquad S = S_A$$

■

Observación: No son propiedades:

1) Intersección

$$\left.\begin{array}{l} A \text{ \textit{libre de contexto}} \\ B \text{ \textit{libre de contexto}} \end{array}\right\} \not\Rightarrow \quad A \cap B \text{ \textit{libre de contexto}}$$

Considerar el siguiente contraejemplo:

$A = \{0^k\, 1^k\, 2^p \;/\; k, p > 0\}$ y $B = \{0^p\, 1^k\, 2^k \;/\; k, p > 0\}$ son *libres de contexto*:

Gramática para A:

$S \to CD$
$C \to 0C1 \mid 01$
$D \to 2D \mid 2$

Gramática para B:

$S \to XY$
$X \to 0X \mid 0$
$Y \to 1Y2 \mid 12$

Pero $A \cap B = \{0^k\, 1^k\, 2^k \;/\; k > 0\}$, que como vimos en el Ejemplo 2.6 **no** es *libre de contexto*.

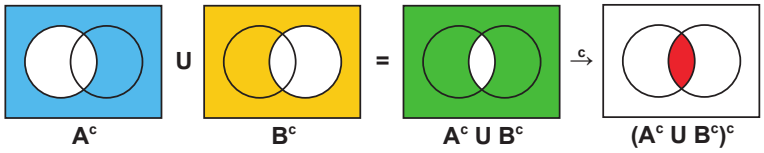
2) Complemento

A *libre de contexto* $\not\Rightarrow$ A^c *libre de contexto*

Suponer que el Complemento *sí* sea una propiedad de clausura:

Sean A y B dos lenguajes *libres de contexto*. Entonces, dado que el Complemento y la Unión son propiedades de clausura, tendríamos que $(A^c \cup B^c)^c$ también es libre de contexto.

Pero $(A^c \cup B^c)^c = A \cap B$, con lo cual la Intersección también sería una propiedad de clausura ↯ Absurdo.



Luego, el Complemento **no** es una propiedad de clausura.

Observación: La **intersección** de un lenguaje **libre de contexto** con un lenguaje **regular** es **libre de contexto**:

$$\left.\begin{array}{l} L \text{ libre de contexto} \\ R \text{ regular} \end{array}\right\} \Rightarrow \underset{\text{libre de contexto}}{L \cap R}$$

Demostración:

Si L es *libre de contexto*, entonces según la Definición 2.19 existe un APD de estado final $M_L = (Q_L, \Sigma, \Gamma, \delta_L, q_{0\,L}, Z_0, F_L) \;/\; L = \mathscr{L}(M_L)$. Si R es *regular*, entonces según la Definición 1.5 existe un AFD $M_R = (Q_R, \Sigma, \delta_R, q_{0\,R}, F_R) \;/\; R = \mathscr{L}(M_R)$.

Entonces la idea es construir un APD $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F) \;/\; L \cap R = \mathscr{L}(M)$ de estado final †, que ejecute "simultáneamente" M_L y M_R. Observar que M_L, M_R y M trabajan sobre el mismo Σ, y M reutiliza el Γ y el Z_0 de M_L.

La técnica se llama "*cross–product construction*" y consiste en construir el tercer autómata mediante el *producto cartesiano* de los otros dos: los estados son pares (q_ℓ, q_r) con $q_\ell \in Q_L$ y $q_r \in Q_R$, el estado inicial es el par $(q_{0\,L}, q_{0\,R})$ que contiene a los estados iniciales, los estados de aceptación son los pares (q_{f_ℓ}, q_{f_r}) que contienen estados de aceptación ($q_{f_\ell} \in F_L$ y $q_{f_r} \in F_R$), y la función de transición δ envía un par de estados a otro par según los resultados de las funciones δ_L y δ_R. En resumen:

1) $Q = Q_L \times Q_R$

2) $\forall a \in \Sigma \;,\; \forall X \in \Gamma \;,\; \forall q_\ell \in Q_L \;,\; \forall q_r \in Q_R \;:$
$\delta\big((q_\ell, q_r), a, X\big) = \Big\{\big((q_\ell', q_r'), \alpha\big) \;/\; (q_\ell', \alpha) \in \delta_L(q_\ell, a, X) \;\wedge\; q_r' = \delta_R(q_r, a)\Big\}$
$\delta\big((q_\ell, q_r), \varepsilon, X\big) = \Big\{\big((q_\ell', q_r), \alpha\big) \;/\; (q_\ell', \alpha) \in \delta_L(q_\ell, \varepsilon, X)\Big\}$ ††

3) $q_0 = (q_{0\,L}, q_{0\,R})$

4) $F = F_L \times F_R$ †††

El APD M construido de esta manera acepta $w \in \Sigma^*$ sí y solo sí M_L y M_R aceptan ambos w, lo cual demuestra que M reconoce el lenguaje de la intersección de L con R.

† Este requerimiento no le quita generalidad a la prueba ya que el Teorema 2.6 demuestra la equivalencia entre el reconocimiento por estado final y por pila vacía.

†† Según la Definición 2.14 un APD puede trabajar con ε, sin embargo, de acuerdo a la Definición 1.3 un AFD no. Entonces en este caso M debe simular la ejecución de M_L pero sin cambiar de estado en M_R.

††† Si utilizáramos un modelo de aceptación por pila vacía, entonces $F_L = \varnothing$ y este producto cartesiano sería vacío.

■

2.3.4. Lema de Ogden

Ejemplo 2.7 *Motivación: Decidir si* $L = \{a^i\, b^j\, c^k\, d^\ell \;/\; i = 0 \;\; ó \;\; j = k = \ell > 0\}$ *es un lenguaje libre de contexto.*

En primer lugar observar que $L = \{b^j\, c^k\, d^\ell \;/\; j,k,\ell > 0\} \uplus \{a^i\, b^k\, c^k\, d^k \;/\; i,k > 0\}$*, con lo cual si prestamos atención al segundo conjunto podemos ver que tiene la misma cualidad que el paradigmático* $\{0^k\, 1^k\, 2^k \;/\; k > 0\}$ *(que como demostramos en el Ejemplo 2.6, no es libre de contexto). Es decir, nuestro lenguaje tiene algunas tiras que aparentemente no podríamos reconocer con un APD, por lo tanto sospechamos que se trata de un lenguaje que* ***no*** *es libre de contexto.*

Entonces para confirmar nuestra intuición vamos a intentar aplicarle el Contrarrecíproco del Pumping Lemma para Lenguajes Libres de Contexto:

Sea $n \in \mathbb{N}$ *la constante del Pumping. Y luego, ¿qué* $z \in L$ *elijo?*

Si elijo $z = a^n\, b^n\, c^n\, d^n$*, es fácil ver que en el caso donde* v *y* x *están en totalmente dentro de la zona de las a's,* $z_i = a^{n+p(i-1)+q(i-1)}\; b^n\; c^n\; d^n$*. Luego como* $|z_i|_a \neq 0$ *vamos a tener que* $\forall i \geq 0 : z_i \in L$ *debido a que* $|z_i|_b = |z_i|_c = |z_i|_d$*. Entonces este z directamente no sirve.*

Entonces vamos a sacar las a's y elegir $z = b^n\; c^n\; d^n$*. Pero aquí también sucede que cuando* v *y* x *están en totalmente dentro de la zona de las b's,* $z_i = b^{n+p(i-1)+q(i-1)}\; c^n\; d^n$*, y entonces* $\forall i \geq 0 : z_i \in L$ *debido a que* $|z_i|_a = 0$*. Este z tampoco sirve.*

¿Y si ponemos una sola a, i.e. $z = a\; b^n\; c^n\; d^n$*? En el caso donde*

$$
\begin{aligned}
u &= \varepsilon \\
v &= a \\
w &= \varepsilon \\
x &= b^p \\
y &= b^{n-p}\; c^n\; d^n
\end{aligned}
\qquad\qquad
\begin{aligned}
1 + 0 + p \leq n \;&\Rightarrow\; 1 + p \leq n \\
1 + p \geq 1 \;&\Rightarrow\; p \geq 0
\end{aligned}
$$

tenemos que $z_i = a^i\; b^{n+p(i-1)}\; c^n\; d^n$*. Luego si* $i = 0 \;\Rightarrow\; z_i = b^{n-p}\; c^n\; d^n \;\in L$ *ya que* $|z_i|_a = 0$*, y si* $i = m > 0 \;\Rightarrow\; z_i = a^m\; b^{n+p(m-1)}\; c^n\; d^n$*, pero aquí podemos tomar* $p = 0$ *y entonces* $z_i = a^m\; b^n\; c^n\; d^n \;\in L$ *debido a que* $|z_i|_b = |z_i|_c = |z_i|_d$*. Otra vez elegimos un z incorrecto.*

*Como ya habíamos mencionado, elegir un z adecuado es una tarea que requiere ingenio y que no siempre puede resultar trivial. Pero en este caso nunca lo vamos a encontrar, este es un caso en el que **no** se puede aplicar el Contrarrecíproco del Pumping Lemma para Lenguajes Libres de Contexto.*

Para este tipo de lenguajes en los que no se puede aplicar el Contrarrecíproco del Pumping Lemma para Lenguajes Libres de Contexto, existe un teorema más fuerte llamado *Lema de Ogden*, que no es más que una generalización del Pumping Lemma para Lenguajes Libres de Contexto. El contrarrecíproco de este resultado permite concentrarnos en ciertas posiciones del z que llamaremos *"distinguidas"*, para así no tener que considerar casos que nos derribarían la demostración. El Pumping Lemma para Lenguajes Libres de Contexto que hemos venido estudiando hasta ahora sería un caso particular donde todas las posiciones de la tira son distinguidas.

Teorema 2.10 Lema de Ogden

Si L *es un lenguaje libre de contexto* **entonces**

$$\exists n \in \mathbb{N} \ / \ \forall z \in L \text{ con } \texttt{dist}(z) \geq n \text{ se cumple que}$$

$$\exists \text{ una descomposición } z = uvwxy \ / \ \texttt{dist}(vwx) \leq n \ , \ \texttt{dist}(vx) \geq 1 \ \ y \ \ \forall i \geq 0 : uv^iwx^iy \in L$$

Y la formulación del **Contrarrecíproco del Lema de Ogden** dice lo siguiente:

Si L es tal que

$$\forall n \in \mathbb{N} \ : \ \exists z \in L \text{ con } \texttt{dist}(z) \geq n \text{ para el cual se cumple que}$$

$$\forall \text{ descomp. } z = uvwxy \text{ en la que } \texttt{dist}(vwx) \leq n \ \ y \ \ \texttt{dist}(vx) \geq 1 \ : \ \exists i \geq 0 \ / \ uv^iwx^iy \notin L$$

entonces L no es un lenguaje libre de contexto.

¿Cómo lo usamos? Dado L, fijamos $n \in \mathbb{N}$ la constante del Pumping y elegimos $z \in L$ así como sus posiciones distinguidas / $\texttt{dist}(z) \geq n$. Luego consideramos todas las descomposiciones $z = uvwxy$ que cumplan $\texttt{dist}(vwx) \leq n$ y $\texttt{dist}(vx) \geq 1$, y en cada una buscamos un $i \geq 0$ / $z_i = uv^iwx^iy \notin L$.

Observación: Para poder demostrar que un lenguaje no es libre de contexto mediante el Contrarrecíproco del Lema de Ogden, no solo hay que elegir "bien" el z sino que además estará presente la dificultad de decidir qué posiciones distinguir. Por ejemplo, si en el caso que acabamos de ver elegimos $z = a^j b^n c^n d^n$ con cualquier $j > 0$ deseado, definitivamente **no** nos sirve distinguir a's, pero sí nos sirve distinguir las b's, las c's o las d's, aunque sería más conveniente distinguir todas ellas y de manera contigua para así tener menos descomposiciones.

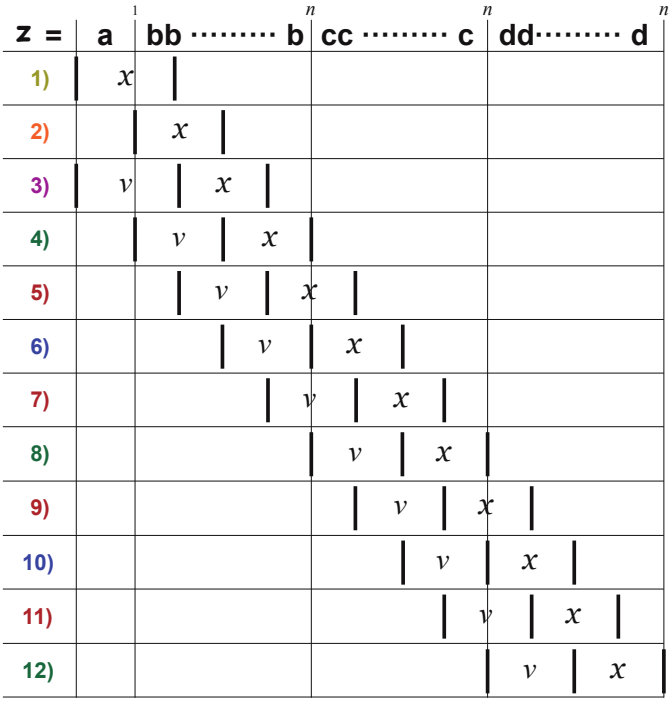
Ejemplo 2.8 *Continuación del Ejemplo 2.7, pero esta vez intentando con el Contrarrecíproco del Lema de Ogden. Recordar que:*

$$L = \{a^i\, b^j\, c^k\, d^\ell \;/\; i = 0 \;\; ó \;\; j = k = \ell > 0\}$$
$$= \{b^j\, c^k\, d^\ell \;/\; j, k, \ell > 0\} \biguplus \{a^i\, b^k\, c^k\, d^k \;/\; i, k > 0\}$$

Sea $n \in \mathbb{N}$ la constante del Pumping.

Elegimos $z = a\; b^n\, c^n\, d^n$ y como posiciones distinguidas las de la parte $b^n\, c^n\, d^n$, es decir, todas las posiciones de la tira a excepción de la primera (que contiene la a) $\Rightarrow$ $\texttt{dist}(z) = n + n + n = 3n \geq n$ ✓

Ahora consideramos todas las descomposiciones $z = uvwxy$ / $\texttt{dist}(vwx) \leq n$ y $\texttt{dist}(vx) \geq 1$:



Dado que la tira comienza con una a, podemos discernir dos grandes "familias": cuando u contiene la a, y cuando v, w, o x contiene la a. Observar que al haber solo una a, la sección que la contenga implicará que las secciones anteriores sean iguales a ε. En ese sentido, notar que y no puede contener la a ya que, al ser la sección final, implicaría que $u = v = w = x = \varepsilon$ $\Rightarrow$ no se cumple la condición $\texttt{dist}(vx) \geq 1$.

*Los Casos **1)**, **2)** y **3)** corresponden, respectivamente, a cuando x, w y v contienen la a. Observar que para asegurar la condición $\texttt{dist}(vx) \geq 1$, es importante que cuando sea v o x la sección que contiene la a, esta también alcance al menos una b.*

*Los Casos **4)** al **12)** corresponden a cuando u contiene la a. Más aún, como el resto de la tira está toda distinguida, las condiciones $\texttt{dist}(vwx) \leq n$ y $\texttt{dist}(vx) \geq 1$ son equivalentes a $|vwx| \leq n$ y $|vx| \geq 1$. Por lo tanto, los casos **4)** al **12)** son análogos, respectivamente, a los **1)** al **9)** estudiados en la prueba por Contrarrecíproco del Pumping Lemma para Lenguajes Libres de Contexto del Ejemplo 2.6.*

Caso 1):

$u = \varepsilon$ $\quad p \leq n$ $\quad \Rightarrow z_i = uv^i wx^i y$

$v = \varepsilon$ $\quad p \geq 1$ $\quad = (a\,b^p)^i\ b^{n-p}\ c^n\ d^n$

$w = \varepsilon$

$x = a\,b^p$

$y = b^{n-p}\ c^n\ d^n$

$$\text{Tomar } i = 2 \Rightarrow z_2 = (a\,b^p)^i\ b^{n-p}\ c^n\ d^n\ |_{i=2}$$
$$= (a\,b^p)^2\ b^{n-p}\ c^n\ d^n$$
$$= a\ b^p\ a\ b^p\ b^{n-p}\ c^n\ d^n$$
$$= a\ b^p\ a\ b^n\ c^n\ d^n$$

Dado que establecimos $p \geq 1$, en z_2 se mezclan a's con b's (a causa de haber bombeado x). Luego z_2 tiene a's pero no es de la forma $a^i b^k c^k d^k$, con lo cual $z_2 \notin L$.

Caso 2):

$u = \varepsilon$ $\quad p + q \leq n$ $\quad \Rightarrow z_i = uv^i wx^i y$

$v = \varepsilon$ $\quad q \geq 1$ $\quad = a\ b^p\ (b^q)^i\ b^{n-p-q}\ c^n\ d^n$

$w = a\,b^p$ $\quad = a\ b^p\ b^{qi}\ b^{n-p-q}\ c^n\ d^n$

$x = b^q$ $\quad = a\ b^{n+q(i-1)}\ c^n\ d^n$

$y = b^{n-p-q}\ c^n\ d^n$

$$\text{Tomar } i = 2 \Rightarrow z_2 = a\ b^{n+q(i-1)}\ c^n\ d^n\ |_{i=2} = a\ b^{n+q}\ c^n\ d^n$$

Dado que z_2 tiene a's, para pertenecer a L debe ser de la forma $a^i b^k c^k d^k$. Luego $z_2 \in L \iff \overset{i}{\overbrace{a}}^{1}\ \overset{k}{\overbrace{b^{n+q}}}\ \overset{k}{\overbrace{c^n}}\ \overset{k}{\overbrace{d^n}} \in L \iff n + q = n \iff q = 0$ ↯ *Esto es absurdo ya que establecimos $q \geq 1$, luego $z_2 \notin L$.*

Caso 3):

$u = \varepsilon$ $\quad p + q + r \leq n$ $\quad \Rightarrow z_i = uv^i wx^i y$

$v = a\,b^p$ $\quad p + r \geq 1$ $\quad = (a\ b^p)^i\ b^q\ (b^r)^i\ b^{n-p-q-r}\ c^n\ d^n$

$w = b^q$ $\quad = (a\ b^p)^i\ b^q\ b^{ri}\ b^{n-p-q-r}\ c^n\ d^n$

$x = b^r$ $\quad = (a\ b^p)^i\ b^{n+r(i-1)-p}\ c^n\ d^n$

$y = b^{n-p-q-r}\ c^n\ d^n$

$$\text{Tomar } i = 2 \Rightarrow z_2 = (a\ b^p)^i\ b^{n+r(i-1)-p}\ c^n\ d^n\ |_{i=2}$$
$$= (a\ b^p)^2\ b^{n+r-p}\ c^n\ d^n$$
$$= a\ b^p\ a\ b^p\ b^{n+r-p}\ c^n\ d^n$$
$$= a\ b^p\ a\ b^{n+r}\ c^n\ d^n$$

$$p + r \geq 1 \Rightarrow \textbf{\textit{Si}} \begin{cases} p = 0 \Rightarrow r \geq 1 \Rightarrow |z_2|_b > |z_2|_c\text{. } Y \text{ como } z_2 \text{ tiene a's} \Rightarrow z_2 \notin L \\ p \neq 0 \Rightarrow p \geq 1 \Rightarrow \begin{array}{c} \text{En } z_2 \text{ se mezclan a's con b's} \\ \text{(a causa de haber bombeado v)} \end{array} \Rightarrow z_2 \notin L \end{cases}$$

Como ya habíamos adelantado, los siguientes casos son análogos a los del Ejemplo 2.6. Simplemente hay que agregar una a al comienzo de la u en cada descomposición, y reemplazar el 0 por la b, el 1 por la c, y el 2 por la d. En todas las descomposiciones es suficiente considerar z_2, y como en todos estos casos z_2 contiene a's, debe respetar la forma $a^i\, b^k\, c^k\, d^k$ para pertenecer a L. A continuación se muestra un resumen de la demostración para los casos restantes:

Caso 4): *$z_2 = a\ b^{n+q+s}\ c^n\ d^n$. Entonces $z_2 \notin L$ porque tiene un exceso de b's.*

Caso 8): *$z_2 \notin L$ porque tiene un exceso de c's.*

Caso 12): *$z_2 \notin L$ porque tiene un exceso de d's.*

Caso 5): *$z_2 = a\ b^{n+p}\ c^s\ b^r\ c^n\ d^n$. Entonces $z_2 \notin L$ porque mezcla b's con c's (a causa de haber bombeado x).*

Caso 7): *$z_2 \notin L$ porque mezcla b's con c's (a causa de haber bombeado v).*

Caso 9): *$z_2 \notin L$ porque mezcla c's con d's (a causa de haber bombeado x).*

Caso 11): *$z_2 \notin L$ porque mezcla c's con d's (a causa de haber bombeado v).*

Caso 6): *$z_2 = a\ b^{n+p}\ c^{n+s}\ d^n$. Entonces, según $p = 0$ o $p \neq 0$, $z_2 \notin L$ porque hay más c's que d's o más b's que d's, respectivamente.*

Caso 10): *$z_2 \notin L$ porque hay más d's que b's o más c's que b's.*

Estas son todas las descomposiciones $z = uvwxy$ que cumplen $\texttt{dist}(vwx) \leq n$ y $\texttt{dist}(vx) \geq 1$, y en cada una encontramos $i \geq 0$ / $z_i \notin L$. Luego, por el Contrarrecíproco del Lema de Ogden, ***L no es libre de contexto****.*

3. Lenguajes Recursivamente Enumerables

3.1. Gramáticas Irrestrictas

3.1.1. Caso General

Definición 3.1 Una ***gramática irrestricta*** es una gramática en la que todas sus reglas de producción son de la forma

$$\alpha \to \beta$$

donde $\alpha, \beta \in (V \cup T)^*$ y $\alpha \neq \varepsilon$.

Definición 3.2 L es un ***lenguaje recursivamente enumerable*** si existe una gramática irrestricta que lo genera.

Ejemplo 3.1 *En el Ejemplo 2.6 de la Sección anterior vimos que el lenguaje* $L = \{0^k\,1^k\,2^k \;/\; k > 0\}$ *no es libre de contexto. Vamos a probar que* L *es un lenguaje recursivamente enumerable construyendo una gramática irrestricta* $G = (V, T, P, S)$ *que lo genere:*

$$V = \{S, U\} \qquad T = \{0, 1, 2\} \qquad P = \left\{ \begin{array}{l} S \to 0SU2 \mid 012 \\ 2U \to U2 \\ 1U \to 11 \end{array} \right\}$$

Por ejemplo, si queremos generar la tira 000111222 $(k = 3)$*:*

$$\begin{array}{rll} S & \overset{2}{\Longrightarrow} 00SU2U2 & // \textit{ Genera recursivamente } 0^{k-1}\,S\,(U2)^{k-1} \\ & \Longrightarrow 00012U2U2 & // \textit{ Termina la recursión} \\ & \overset{3}{\Longrightarrow} 0001UU22 & // \textit{ Acomoda las } U\textit{'s en el medio} \\ & \overset{2}{\Longrightarrow} 000111222 & // \textit{ Reemplaza las } U\textit{'s por } 1s \end{array}$$

Ejemplo 3.2 *Construir una gramática irrestricta para* $L = \{a^n \,\#\, b^m \;/\; m = \dot{n}\}$

$$V = \{S, A, X\} \qquad T = \{a, b, \#\} \qquad P = \left\{ \begin{array}{ll} S \to XS \mid A & X\# \to \# \\ A \to aA \mid \# & b\# \to \#b \\ Xa \to abX & ba \to ab \end{array} \right\}$$

Por ejemplo, si queremos generar la tira $aa\#bbbbb$ $(n = 2,\ m = 3 \cdot 2)$*, procederíamos de la siguiente manera (tener en mente que* $m = \dot{n} \iff \underset{k \in \mathbb{N}}{m = kn} \iff m = \underbrace{n + \cdots + n}_{k \text{ veces } n}$*):*

$S \overset{3}{\Longrightarrow} XXXS$ // *Termina de generar el factor de multiplicidad (la cantidad de X es el k)*

$\Longrightarrow XXXA$ // *Da inicio a la generación de las a's*

$\overset{3}{\Longrightarrow} XXXaa\#$ // *Termina de generar las n a's y el #*

$\Longrightarrow XXabXa\#$ // *Comienza el pasaje de la X que está más a la derecha a través de las a's*

$\Longrightarrow XXababX\#$ // *Cuando una X llega al # significa que cumplió su función y puede descartarse*

$\Longrightarrow XXabab\#$ // *El paso de una X a través de las n a's deja la misma cantidad de b's*

$\Longrightarrow XXaabb\#$ // *Estas n b's deben pasar a la derecha del # para no intervenir más*

$\overset{2}{\Longrightarrow} XXaa\#bb$ // *El camino hasta el # queda "libre" para la siguiente X*

$\overset{6}{\Longrightarrow} Xaa\#bb\,bb$ // *Se repiten los 6 pasos anteriores con la siguiente X, y se obtienen otras n b's*

$\overset{6}{\Longrightarrow} \underbrace{aa}_{n}\#\ \underbrace{bb\,bb\,bb}_{k \text{ veces } n}$ // *El proceso de atravesar las a's y generar n b's se repite hasta la k–ésima X*

3.1.2. Gramáticas Sensibles al Contexto

Un caso particular de gramáticas irrestrictas son las denominadas *gramáticas sensibles al contexto*. La característica fundamental que poseen estas gramáticas es que sus reglas son *no–contractivas*. Esto significa que por norma general, en todas las reglas lo que está del lado izquierdo debe ser a lo sumo tan largo como lo que está del lado derecho.

Definición 3.3 Una ***gramática sensible al contexto*** es una gramática en la que, en principio, todas sus reglas de producción son de la forma

$$\alpha \to \beta$$

donde $\alpha, \beta \in (V \cup T)^*$, $\alpha \neq \varepsilon$ y $|\alpha| \leq |\beta|$. La excepción a esta última condición solo puede ocurrir por única vez con la producción «$S \to \varepsilon$», en caso de que el lenguaje contenga la tira vacía.

Definición 3.4 L es un ***lenguaje sensible al contexto*** si existe una gramática sensible al contexto que lo genera.

Observación: La gramática dada en el Ejemplo 3.1 es no–contractiva, por ende, se trata de una gramática sensible al contexto. En consecuencia, el lenguaje $L = \{0^k\,1^k\,2^k \;/\; k > 0\}$ para el cual hemos demostrado en el Ejemplo 2.6 que no es libre de contexto, es –de hecho– un lenguaje sensible al contexto.

3.2. Máquinas de Turing

3.2.1. Caso General

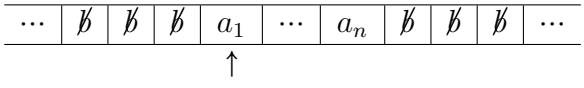
A lo largo de la Sección 2 presentamos los *autómatas push–down* como máquinas capaces de computar los lenguajes libres de contexto, y más adelante vimos que existen lenguajes que (al no ser de este tipo) no pueden ser reconocidos con esta clase de máquinas.

El último modelo de computación que veremos es el de las llamadas *máquinas de Turing* que nos permitirán reconocer los lenguajes recursivamente enumerables. En este modelo dispondremos de una cantidad *finita* de estados junto con una cinta virtualmente *infinita* sobre la cual podremos movernos a la izquierda o hacia la derecha tantas veces como sea necesario, así como leer y escribirla donde queramos. Nos restringiremos a las máquinas de Turing deterministas, de una cinta y con un solo estado de aceptación del cual no salen transiciones (de modo que, al llegar a este, la máquina se detenga), pero la teoría se puede extender a máquinas no–deterministas e incluso con múltiples cintas, y se puede demostrar que todas las variantes son equivalentes al modelo aquí tratado.

Definición 3.5 Una ***máquina de Turing (MT)*** es una 7-tupla $M = (Q, \Sigma, \Gamma, \delta, q_0, \not b, F)$ donde:

- Q es el conjunto de estados
- $\Sigma \subset \Gamma$ es el alfabeto de la entrada
- Γ es el alfabeto de la cinta
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{I, D\}$ es la función de transición de estados
- $q_0 \in Q$ es el estado inicial
- $\not b \in (\Gamma \setminus \Sigma)$ es el caracter `BLANK`, que representa que una celda de la cinta está vacía
- $F \subseteq Q$ es el conjunto unitario del estado de aceptación, para el cual δ está indefinida

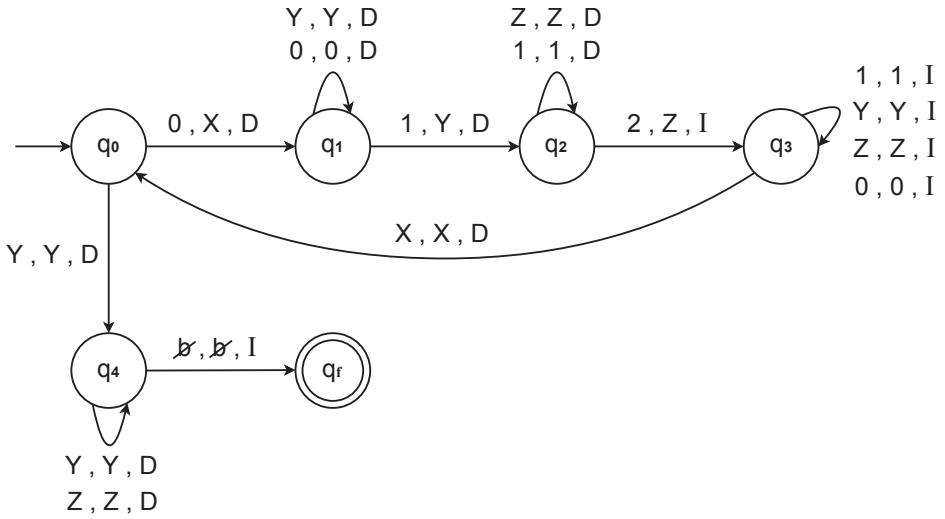
Asumimos que la máquina se inicia con el cabezal de lectura–escritura ubicado sobre el primer símbolo de la tira $a_1 \cdots a_n$ que queremos decidir si pertenece al lenguaje.



Definición 3.6 Dada una MT $M = (Q, \Sigma, \Gamma, \delta, q_0, \not b, F)$, el ***lenguaje aceptado por M*** es $\mathscr{L}(M) = \{x \in \Sigma^* \mid M \text{ llega al } q_f \in F \text{ con } x \text{ como entrada}\}$. Es decir, $\mathscr{L}(M)$ es el conjunto de tiras de Σ^* tales que, partiendo desde q_0, M se detiene en su estado de aceptación tras procesarlas.

Definición 3.7 L es un ***lenguaje recursivamente enumerable*** si es aceptado por alguna MT.

Ejemplo 3.3 *Vamos a construir una MT para el lenguaje* $L = \{0^k\,1^k\,2^k \;/\; k > 0\}$*:*

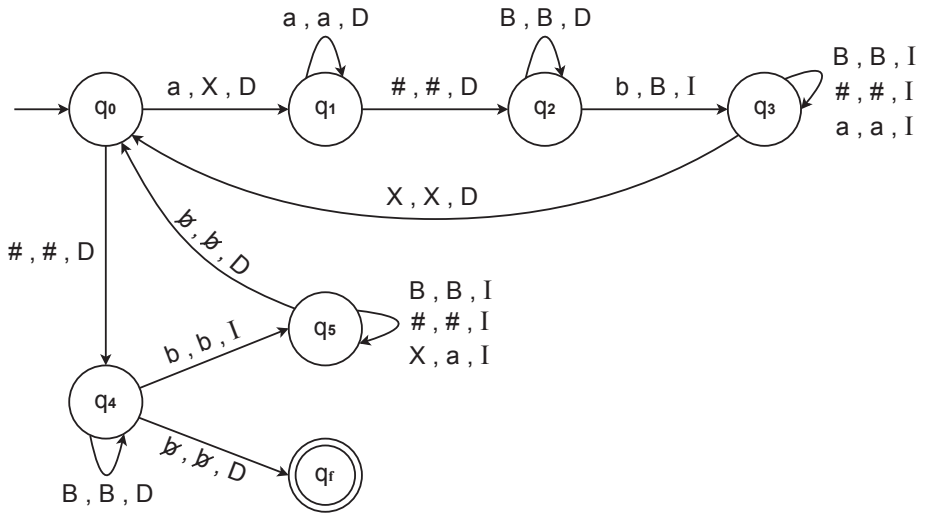


Al igual que en los autómatas vistos anteriormente, colocamos una flecha apuntando al estado inicial y un circulo interior en el estado de aceptación. La notación que usamos para el rótulo de las transiciones es:

símboloLeído , símboloImpreso , movimientoSobreLaCinta

La idea de cómo funciona esta MT es la siguiente: Para cada 0 *leído estando en* q_0*, marcarlo con* X *y avanzar hacia la derecha ignorando el resto de los* 0*s hasta encontrar un* 1*. Tras encontrarlo, marcarlo con* Y *y avanzar hacia la derecha ignorando el resto de los* 1*s hasta encontrar un* 2*. Luego de encontrarlo, marcarlo con* Z *y retroceder hacia la izquierda ignorando* 0*s,* 1*s,* Y*'s y* Z*'s hasta volver a la última* X *escrita. Estando allí, moverse a la derecha y ver qué hay. Si hay otro* 0*, volver a repetir todo lo anterior (ahora al avanzar hacia la derecha, también habrá que ignorar las* Y*'s y* Z*'s escritas previamente), y si lo que se encuentra es una* Y*, es porque ya no quedan* 0*'s por contabilizar. En este último caso, si la entrada era de la forma* $0^k\,1^k\,2^k$*, lo único que debería haber quedado escrito en la cinta (de acuerdo al procedimiento seguido) es* $X^k\,Y^k\,Z^k$*. Por lo tanto, avanzamos hacia la derecha salteando* Y*'s y* Z*'s hasta alcanzar el final de la tira (lo cual se reconoce con una celda vacía). En esta situación exitosa, la máquina habrá llegado al estado de aceptación y entonces reconocerá que la tira dada como entrada forma parte de* L*. Si en algún momento no se encontrara el símbolo esperado, es porque la cantidad de ese símbolo es menor a la de otro (entonces la tira no era de la forma* $0^k\,1^k\,2^k$*), y en tal caso la máquina –al no tener una transición especificada– se va a detener en un estado que no es el de aceptación, y por ende,* ***no*** *reconocerá a la entrada como parte de* L*. Si en la recorrida final en lugar de leer* Y*'s y* Z*'s apareciera otro símbolo, es porque hay un exceso de ese (entonces la entrada tampoco era de la forma* $0^k\,1^k\,2^k$*), y de vuelta –al no tener una transición definida– nuestra máquina se detiene en un estado que no es el de aceptación y rechaza la tira.*

Ejemplo 3.4 *Una MT para el lenguaje* $L = \{a^n \# b^m \mid m = \dot{n}\}$*:*



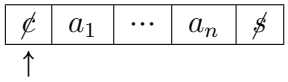
3.2.2. Autómata Lineal Acotado

Un "caso especial" de MT son los *autómatas lineales acotados*, lo cuales permiten computar todos los lenguajes sensibles al contexto. La diferencia fundamental con una MT "genérica" es que la palabra a ser analizada se encuentra encerrada entre dos símbolos especiales, $¢$ y $£$. En consecuencia, las operaciones que se pueden aplicar sobre esta cinta *acotada* están restringidas al espacio delimitado por dichos caracteres, donde estos últimos no pueden ser sobrescritos por ningún símbolo distinto.

Definición 3.8 Un ***autómata lineal acotado (ALA)*** es una 8-tupla $M = (Q, \Sigma, \Gamma, \delta, q_0, ¢, £, F)$ donde:

- Q es el conjunto de estados
- $\Sigma \subseteq \Gamma$ es el alfabeto de la entrada
- Γ es el alfabeto de la cinta
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{I, D\}$ es la función de transición de estados
- $q_0 \in Q$ es el estado inicial
- $¢ \in \Sigma$ es el caracter delimitador del **origen** de la entrada, donde cualquier transición que lo involucre debe ser de la forma $\delta(q, ¢) = (p, ¢, D)$, con $p, q \in Q$
- $£ \in \Sigma$ es el caracter delimitador del **final** de la entrada, donde cualquier transición que lo involucre debe ser de la forma $\delta(q, £) = (p, £, I)$, con $p, q \in Q$
- $F \subseteq Q$ es el conjunto unitario del estado de aceptación, para el cual δ está indefinida

Asumimos que el autómata se inicia con el cabezal de lectura–escritura ubicado sobre el símbolo especial que indica el comienzo (a su derecha) de la tira $a_1 \cdots a_n$ que queremos decidir si pertenece al lenguaje.



Definición 3.9 *L* es un ***lenguaje sensible al contexto*** si es aceptado por algún ALA.

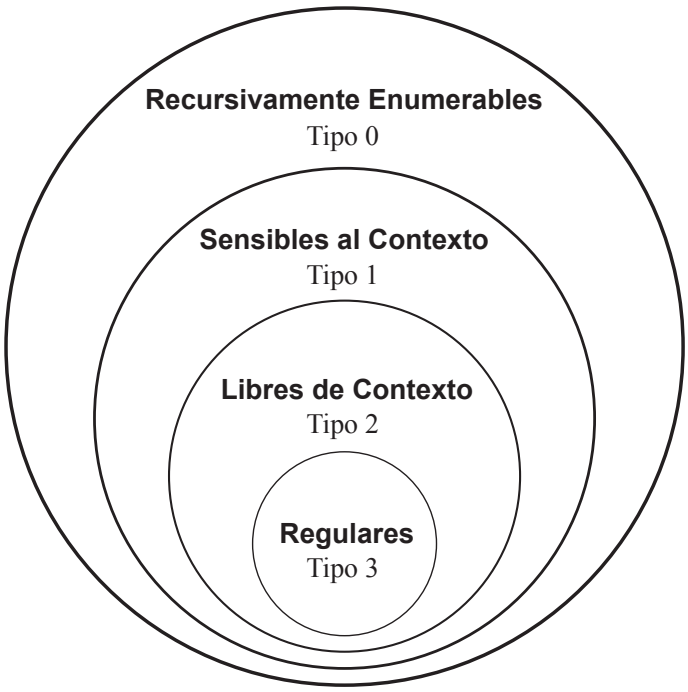
Observación: Según *Hopcroft, Motwani y Ullman*, "actualmente las gramáticas sensibles al contexto no juegan un rol importante en la práctica", y en consecuencia tampoco los ALAs. Debido a eso, cuando en este curso corresponda construir una gramática irrestricta (o bien una MT), no haremos hincapié en que la misma sea sensible al contexto (o bien se trate de un ALA), aun cuando teóricamente sea posible hacerlo para el lenguaje en cuestión. No obstante, para lo que sigue es importante destacar que se ha demostrado la existencia de lenguajes recursivamente enumerables que **no** son sensibles al contexto.

3.3. Jerarquía de Chomsky

Todos los tipos de lenguajes que hemos visto a lo largo de este curso corresponden a una clasificación denominada *Jerarquía de Chomsky*, la cual fue presentada en 1959 por el lingüista estadounidense **Noam Chomsky**.

Teorema 3.1 Teorema de la Jerarquía de Chomsky

Lenguajes de **Tipo $i+1$** $\subset$ *Lenguajes de* **Tipo i**

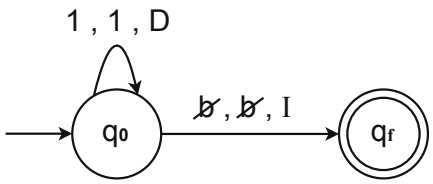


3.4. Una introducción a la Teoría de la Computación

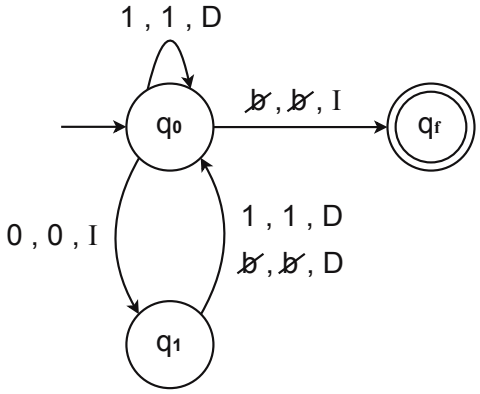
Naturalmente surge la pregunta de si existen lenguajes más allá de los recursivamente enumerables, y la respuesta es que sí los hay.

Comencemos recordando que, dados una MT M definida sobre un alfabeto Σ y una tira $x \in \Sigma^*$, para que $x \notin \mathscr{L}(M)$ debe suceder –de acuerdo a la Definición 3.6– que M no se detenga en su estado de aceptación con x como entrada. El hecho de "no detenerse en su estado de aceptación" puede darse por dos situaciones: o bien porque M se detiene en un estado que *no es el de aceptación*, o bien porque M *no se detiene*.

Por lo tanto, nuestra definición de máquina de Turing no excluye la posibilidad de crear máquinas que entren en **bucles infinitos**. Para ver un ejemplo bien simple, considerar las siguientes MT para reconocer el lenguaje $\mathscr{L}(1^*)$ definido sobre $\Sigma = \{0, 1\}$:



Máquina que siempre se detiene



Máquina que entra en *loop* si lee un 0

Se dice que un lenguaje es ***decidible*** cuando existe una MT M que determina si $x \notin \mathscr{L}(M)$ por el primer caso para todo $x \in \Sigma^*$. Es decir, un lenguaje es decidible si es reconocido por alguna MT que **siempre se detiene**.

En 1928, el matemático alemán David Hilbert planteó un problema denominado ***Entscheidungsproblem*** ("*problema de decisión*"), que consistía en determinar si existe un *procedimiento para decidir* si una fórmula lógica de primer orden arbitraria es o no *universalmente válida* (es decir, si es o no un teorema).

Tenemos la noción intuitiva de que un *procedimiento* es "una secuencia de pasos bien definidos", y que cuando el procedimiento es *para decidir*, entonces la secuencia debe ser *finita* (con lo cual, al terminar de ejecutarla, *siempre* se podrá devolver una respuesta por «sí» o por «no»). Pero ¿cómo podríamos formalizar estos conceptos, de tal modo que fuera posible *demostrar* afirmaciones como la del *Entscheidungsproblem*?

En 1936, el matemático británico Alan Turing quien se había interesado por el *Entscheidungsproblem*, publicó un artículo en el que definía unas "máquinas de computar" que le ayudarían a definir *procedimiento* con rigor. Y en particular, un *procedimiento para decidir*, sería aquel que puede ser formalizado con una máquina que siempre se detiene.

Por lo tanto, el problema planteado por Hilbert resulta equivalente a determinar si es decidible el lenguaje

$$ENTS = \{\varphi \mid \varphi \text{ es un teorema}\}$$

En el mismo trabajo, Turing mostró cómo cualquiera de sus máquinas M se puede codificar e identificar mediante un *número descriptor* $\mathcal{D}(M)$. Y más sorprendentemente, explicó cómo crear una máquina "universal", en el sentido de que recibe como entradas el número descriptor $\mathcal{D}(M)$ de alguna máquina M definida sobre un alfabeto Σ y una tira $x \in \Sigma^*$, y es capaz de decodificar $\mathcal{D}(M)$ y simular el comportamiento de M con x como entrada.

Luego, a partir de ese marco teórico, consideró otro problema denominado ***Halting Problem*** ("*problema de la parada*"), el cual consistía en determinar si existe un *procedimiento para decidir* si una máquina arbitraria dotada con alguna entrada se detiene. O equivalentemente, si es decidible el lenguaje

$$HALT = \{ < \mathcal{D}(M), x > \mid M \text{ se detiene con } x \text{ como entrada} \}$$

Y logró reducir el *Halting Problem* al *Entscheidungsproblem*, es decir, suponiendo que el lenguaje $ENTS$ fuera decidible, mostró cómo utilizar la hipotética máquina que lo decide para construir otra que decida el lenguaje $HALT$. Pero por otro lado, Turing demostró que ***HALT* es indecidible**. Por lo tanto, si $ENTS$ fuera decidible se generaría un absurdo, concluyendo así que el problema planteado por Hilbert de verificar teoremas "mecánicamente", es irresoluble.

Notar que empleando una MT universal $\mathcal{U}$ es posible computar $HALT$:

- Si $x \in \mathscr{L}(M) \Rightarrow M$ se detiene $\Rightarrow < \mathcal{D}(M), x > \in HALT$. Por otro lado, como $\mathcal{U}$ simula a M, cuando esta se haya detenido $\mathcal{U}$ puede moverse a su propio estado de aceptación y detenerse allí $\Rightarrow \mathcal{U}$ reconoce el par ✓
- Si $x \notin \mathscr{L}(M) \Rightarrow M$ podría detenerse o no.
  - Si M se detiene $\Rightarrow < \mathcal{D}(M), x > \in HALT$. Entonces, al igual que el caso anterior, $\mathcal{U}$ puede desplazarse a su propio estado de aceptación $\Rightarrow \mathcal{U}$ reconoce el par ✓
  - Si M no se detiene $\Rightarrow < \mathcal{D}(M), x > \notin HALT$, y por otro lado, como $\mathcal{U}$ va a estar imitando a M, tampoco se va a detener $\Rightarrow \mathcal{U}$ no acepta el par ✓

Por lo tanto, ***HALT* es recursivamente enumerable**.

Una propiedad destacada que vincula los lenguajes recursivamente enumerables con los decidibles dice lo siguiente:

Si L y L^c son ambos recursivamente enumerables $\Rightarrow$ L y L^c son ambos decidibles

Entonces, si consideramos el complemento de $HALT$, i.e. los pares $< \mathcal{D}(M), x >$ tales que M **no** se detiene con x como entrada, o sea, aquellos pares en los que x hace que M entre en un bucle infinito:

$$HALT^c = \{ < \mathcal{D}(M), x > \mid M \text{ cae en un bucle infinito con } x \text{ como entrada} \}$$

y el contrarrecíproco de la propiedad anterior:

$$\textit{Si } L \textit{ o } L^c \textit{ es indecidible } \Rightarrow \textit{ } L \textit{ o } L^c \textit{ no es recursivamente enumerable}$$

estamos en condiciones de responder la pregunta con la que iniciamos esta Subsección.

Gracias a Turing sabemos que $HALT$ es indecidible, entonces $HALT$ o $HALT^c$ no ha de ser recursivamente enumerable. Pero nosotros hemos mostrado que $HALT$ sí lo es, por lo tanto, debe suceder que **$HALT^c$ no es recursivamente enumerable**.

Corolario: Entendiendo a las computadoras como máquinas de Turing, **existen problemas informáticos que *ninguna* computadora puede resolver**. Ninguna computadora que se haya construido, ninguna computadora que actualmente exista, y ninguna computadora que alguna vez se vaya a construir. Particularmente, si vemos a un programa como una computadora dedicada a resolver cierta tarea, el hecho de que el lenguaje $HALT^c$ no pueda ser reconocido con una máquina de Turing, demuestra que ***nunca* existirá un programa que reconozca si *cualquier* programa con *cualquier* entrada entrará en un bucle infinito**.

Entonces, ¿por qué usamos como modelo de computación una teoría que sabemos que tiene, por lo pronto, la anterior limitación? Más aun sabiendo que en la década del 30', cuando Turing publicó su extraordinario artículo, ya existían otros modelos de computación como las ***funciones recursivas*** o el ***cálculo lambda*** (obras de los matemáticos Kurt Gödel y Alonzo Church, respectivamente).

Si bien las "máquinas de computar" definidas por Turing no son el único modelo de computación que se ha inventado, resultó ser que ¡todos son equivalentes entre sí! Esto es, para cada caso se ha probado que todo lo que uno puede abarcar también lo cubre el otro, y viceversa. La suposición (indemostrable, pero ampliamente aceptada al punto de considerarse un *axioma*) de que "ser modelado por una máquina de Turing" es equivalente a "ser computable", se conoce como la *tesis de Turing–Church*. Luego, asumiendo que todos los modelos que sean capaces de abarcar "lo computable" tienen las mismas limitaciones teóricas, surge otra pregunta:

¿Qué hizo que el modelo de máquinas de Turing prevaleciera sobre los demás, y fuera tomado por otros investigadores posteriores (como John von Neumann) para definir las bases de las computadoras como las conocemos hoy en día? Probablemente haya sido a causa de la naturalidad que nos transmiten este tipo de máquinas, gracias a la simple pero aun creativa idea de Turing de preguntarse cómo computaría un "humano ideal".

Para responder esto, imaginó que este dispondría de todo el papel que fuera necesario, mantendría en su mente una cantidad finita de estados en los que pudiera ubicarse durante el cómputo, no se involucraría en tareas complejas (solo realizaría operaciones elementales tales como escribir en un sector del papel o pasar a observar un sector contiguo), y no se preocuparía por ser consciente más allá de la observación actual.

1. *Computing machines.*

We have said that the computable numbers are those whose decimals are calculable by finite means. This requires rather more explicit definition. (···) For the present I shall only say that the justification lies in the fact that the human memory is necessarily limited.

We may compare a man in the process of computing a real number to a machine which is only capable of a finite number of conditions $q_1, q_2, \ldots, q_R$ which will be called "m-configurations". The machine is supplied with a "tape" (the analogue of paper) running through it, and divided into sections (called "squares") each capable of bearing a "symbol".

The "scanned symbol" is the only one of which the machine is, so to speak, "directly aware". However, by altering its m-configuration the machine can effectively remember some of the symbols which it has "seen" (scanned) previously. The possible behaviour of the machine at any moment is determined by the m-configuration q_n and the scanned symbol $\mathfrak{S}(r)$.

In some of the configurations in which the scanned square is blank (*i.e.* bears no symbol) the machine writes down a new symbol on the scanned square: in other configurations it erases the scanned symbol. The machine may also change the square which is being scanned, but only by shifting it one place to right or left. In addition to any of these operations the m-configuration may be changed.

Extractos del Capítulo 1 de "*On Computable Numbers, with Application to the Entscheidungsproblem*" por Alan Turing

4. Algoritmos

4.1. Conversión AFND en AFD

ENTRADA: $AFND\ M_N = (Q_N, \Sigma_N, \delta_N, q_{0N}, F_N)$
SALIDA : $AFD\ M = (Q, \Sigma, \delta, q_0, F)$ *equivalente*

$\Sigma \leftarrow \Sigma_N$	// Ambos autómatas leen los mismos símbolos.
$q_0 \leftarrow [q_{0N}]$	// El nuevo q_0 es el conjunto formado por el anterior estado inicial.
$Q \leftarrow \varnothing$	// El nuevo Q se inicia vacío.
AÑADIR(q_0, Q)	// Se empieza a construir desde q_0.
FOR-EACH q IN Q DO	// Para cada estado-conjunto q en Q ...
IF (NOT MARCADO(q)) THEN	// Si aún no fue procesado ...
FOR-EACH a IN Σ DO	// Para cada posible entrada ...
$R_a \leftarrow \bigcup_{p \in q} \delta_N(p, a)$	// Juntar todos los estados alcanzables con ella desde q
AÑADIR(R_a, Q)	// Agregar eso como un nuevo estado-conj.
$\delta(q, a) \leftarrow R_a$	// Definir la transición con a desde el q actual al estado-conjunto recién creado.
ENDFOR	
MARCAR(q)	// Marcar q como ya procesado.
ENDIF	
ENDFOR	
$F \leftarrow \{q \in Q \ / \ q \cap F_N \neq \varnothing\}$	// Un estado-conjunto q será de aceptación sii contiene alguno de los anteriores estados de aceptación.

4.2. Conversión AFND–ε en AFND

ENTRADA: $AFND-\varepsilon\ \ M_E = (Q_E, \Sigma_E, \delta_E, q_{0E}, F_E)$
SALIDA : $AFND\ \ M_N = (Q_N, \Sigma_N, \delta_N, q_{0N}, F_N)\ \ equivalente$

$\Sigma_N \leftarrow \Sigma_E$	// El alfabeto se mantiene.
$q_{0N} \leftarrow q_{0E}$	// El estado inicial se mantiene.
$Q_N \leftarrow Q_E$	// El conjunto de estados no cambia.

FOR-EACH q IN Q_N DO
$E \leftarrow \varepsilon\text{-}c\ell(q)$
FOR-EACH a IN Σ_N DO
$\delta_N(q,a) \leftarrow \varepsilon\text{-}c\ell\left(\bigcup_{p \in E} \delta_E(p,a)\right)$
ENDFOR
ENDFOR
	// Para cada estado q en Q_N ...
	// Calcular su ε–clausura, y
	// Para cada posible entrada ...
	// Re-definir la transición desde q con la entrada a.

$F_N \leftarrow \begin{cases} F_E \cup \{q_{0E}\} & si\ \ \varepsilon\text{-}c\ell(q_{0E}) \cap F_E \neq \varnothing \\ F_E & si\ \ no \end{cases}$	// El conjunto de estados de aceptación se mantiene, y se le agrega q_{0E} sii $\varepsilon\text{-}c\ell(q_{0E})$ contiene algún estado de aceptación.

4.3. Conversión AFD en RegEx: Análisis de Kleene / Clases de Equivalencia

	Análisis de Kleene	**Clases de Equivalencia**
Cómo **formar** el sistema de ecuaciones:	Transiciones **salientes**: `símbolo·estadoDestino`, cada uno separado por | . Agregar ε a los estados de **aceptación**.	Transiciones **entrantes**: `estadoPartida·símbolo`, cada uno separado por | . Agregar ε al estado **inicial**.
Cómo **resolver** el sistema de ecuaciones:	Lema de Arden "Versión 1" $X = rX \mid s \xrightarrow{\varepsilon \notin \mathscr{L}(r)} X = r^*s$	Lema de Arden "Versión 2" $X = Xr \mid s \xrightarrow{\varepsilon \notin \mathscr{L}(r)} X = sr^*$
Objetivo de resolver el sistema de ecuaciones:	Hallar X_0, pues $\mathscr{L}(M) = \mathscr{L}(X_0)$. Cada X_k representa como llegar a un estado de aceptación desde q_k.	Hallar cada X_k, luego $\mathscr{L}(M) = \mathscr{L}(X_{f_1} \mid \cdots \mid X_{f_r})$, donde $q_{f_i} \in F$. Cada X_k representa una clase de R_M.
Qué hacer con los **estados pozo**:	Eliminarlos, así como las transiciones que llegan hacia ellos.	No hay problema †.

† Si lo que buscamos es simplemente conseguir una expresión regular para $\mathscr{L}(M)$, de acuerdo a este método los estados pozo – al no ser estados de aceptación – no van a ser empleados a la hora de construir la expresión. Por lo tanto, en la práctica es indiferente tenerlos en cuenta.

Sin embargo, si es de nuestro interés obtener expresiones para todas las clases de la relación R_L teniendo como base un autómata mínimo (donde, justamente por ser mínimo, estas clases coinciden con las de la relación R_M vinculada a los estados), es importante **conservar el estado pozo**. Esto se debe a que si no consideráramos al estado pozo, entonces ¡estaríamos ignorando toda la clase de las tiras que este representa! Observar que en un AFD mínimo, si tiene un estado pozo, este es único (como las clases de equivalencia inducen una *partición* de Σ^*, una tira no puede estar simultáneamente en dos clases).

4.4. Minimización de AFD

ENTRADA: $AFD\ \ M = (Q, \Sigma, \delta, q_0, F)$
SALIDA : $AFD\ \ M' = (Q', \Sigma', \delta', q_0', F')$ *equivalente y* ***mínimo***

```
# PASO 1: Crear la partición base #
π_0 :   Q \ F    F

i ← 1

# PASO 2: Lo siguiente es para crear los subconjuntos C del nivel π_i #
FOR-EACH C_j IN π_{i-1} DO
   FOR-EACH {p,q}_{p≠q} IN C_j × C_j DO
      IF (∀a ∈ Σ : ∃C_k ⊂ π_{i-1} / δ(p,a) ∈ C_k ∧ δ(q,a) ∈ C_k) THEN
         p,q ∈ C_r ⊂ π_i
      ELSE
         p ∈ C_s ⊂ π_i
         q ∈ C_t ⊂ π_i , t ≠ s
      ENDIF
   ENDFOR
ENDFOR

# PASO 3: ¿Terminé? #
IF (π_i = π_{i-1}) THEN
   π_f ← π_i
   GOTO PASO 4
ELSE
   i ← i + 1
   GOTO PASO 2
ENDIF

# PASO 4: Construcción del autómata mínimo #
Q' ← {C_α / C_α ⊂ π_f}

Σ' ← Σ

δ'(C_x, a) = C_y  ⟺  ∃x ∈ C_x ∧ ∃y ∈ C_y / δ(x,a) = y   (C_x, C_y ⊂ π_f , a ∈ Σ')

q_0' ← C_β ⊂ π_f / q_0 ∈ C_β

F' ← {C_φ ⊂ π_f / C_φ ∩ F ≠ ∅}
```

Explicaciones:

En el `PASO 1` (líneas 1 a 4) se define la partición base, y se inicializa el contador de niveles posteriores al base.

En el `PASO 2` (líneas 7 a 16):

7 Para cada subconjunto C_j en la partición π_{i-1} anterior ...
8 Para cada par de estados p y q (distintos) del C_j considerado ...
9 Si para cada símbolo de Σ, p y q van a un mismo C_k del nivel π_{i-1} anterior ...
p **y** q **son equivalentes**, entonces deben ser agregados a un mismo subconjunto del nivel π_i que se está creando.

11 Si no (†) ...
p **y** q **son distinguibles**, y por lo tanto deben ser agregados a dos subconjuntos distintos del nivel π_i que se está creando.

14 FinSi
15 FinPara
16 FinPara

(†) Es decir, existe al menos un símbolo de Σ para el cual p y q van a dos subconjuntos distintos del nivel π_{i-1} anterior

En el `PASO 3` (líneas 19 a 25) se evalúa si se ha llegado a un punto donde las particiones no se pueden "refinar" más ($\pi_i = \pi_{i-1}$). En caso afirmativo se deja de particionar y se pasa a construir el autómata equivalente, y en caso contrario simplemente se vuelve al punto anterior para construir el siguiente nivel π_{i+1}.

En el `PASO 4` (líneas 28 a 36):

28 Los estados de Q' serán cada uno de los subconjuntos C de la partición π_f final.

30 El alfabeto se mantiene.

32 La función de transición δ' se define según las transiciones de los estados internos de cada subconjunto C de π_f.

34 El estado inicial $q_0{}'$ será aquel subconjunto C de π_f que contenga al q_0 original.

36 Serán estados de aceptación todos los subconjuntos C de π_f que contengan alguno de los estados de aceptación originales.

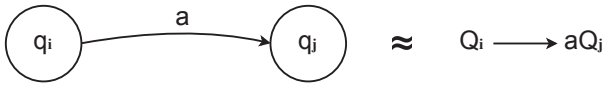
4.5. Conversión Autómata Finito en Gramática Regular

ENTRADA: $AFND-\varepsilon\ M = (Q, \Sigma, \delta, q_0, F)$
SALIDA : $Gramática\ Lineal\ Derecha\ G = (V, T, P, S)\ equivalente$

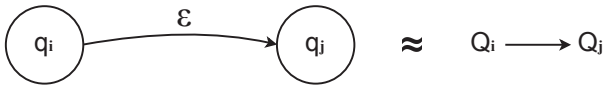
```
# PASO 1: Asociar cada estado q_i ∈ Q con una variable Q_i ∈ V #
V ← ∅
FOR-EACH q_i IN Q DO
   AÑADIR(Q_i, V)
ENDFOR

# PASO 2: Mismos símbolos #
T ← Σ

# PASO 3: Definir las reglas de producción de P en función de las transiciones y
          los estados de aceptación #
P ← ∅
FOR-EACH q_i IN Q DO
   FOR-EACH a IN (Σ ∪ {ε}) DO
      R_a ← δ(q_i, a)
      FOR-EACH q_j IN R_a DO
         AÑADIR(«Q_i → aQ_j», P)
      ENDFOR
   ENDFOR
   IF (q_i ∈ F) THEN
      AÑADIR(«Q_i → ε», P)
   ENDIF
ENDFOR

# PASO 4: S = Q_0 #
RENOMBRAR(Q_0, S)
```

Observación: Podemos aplicar el algoritmo más rápidamente si disponemos del diagrama de estados del autómata:

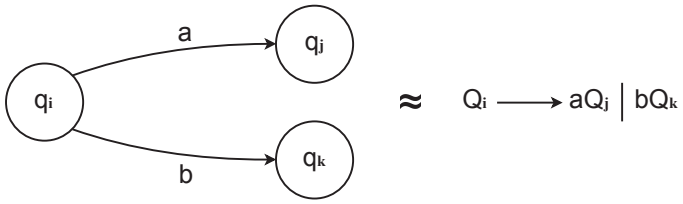
- Para una transición de q_i a q_j con $a \in \Sigma$, creamos la regla «$Q_i \to aQ_j$» $\in P$



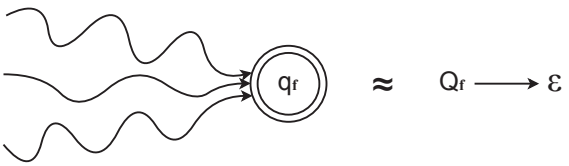
- Para una ε–transición entre q_i y q_j, creamos la regla «$Q_i \to Q_j$» $\in P$



- Para transiciones saliendo desde un mismo estado podemos "juntar" las reglas



- Los estados de aceptación producen ε



- Conviene llamar S a la variable del estado inicial (en vez de Q_0) desde un principio, para así no tener que renombrar todas las apariciones de Q_0 a lo largo de todas las reglas de producción creadas.

4.6. Simplificación de Gramáticas Libres de Contexto

ENTRADA: G^a *Libre de Contexto* $G = (V, T, P, S)$
SALIDA : G^a *Libre de Contexto* $G' = (V', T', P', S')$ *equivalente y* ***simplificada***

```
# PASO 0: Establecer configuraciones iniciales #
V' ← V
T' ← T
P' ← P
S' ← S

# PASO 1: Eliminar ε-producciones #
FOR-EACH «X_i → ε» IN P' DO
   FOR-EACH «A → X_1 ··· X_{i-1} X_i X_{i+1} ··· X_t» IN P' DO
      AÑADIR(«A → X_1 ··· X_{i-1} X_{i+1} ··· X_t», P')
   ENDFOR
   REMOVER(«X_i → ε», P')
ENDFOR

# PASO 2: Eliminación de producciones unitarias #
WHILE (∃ «A → B» ∈ P') DO
   FOR-EACH «B → α» IN P' DO
      AÑADIR(«A → α», P')
   ENDFOR
   REMOVER(«A → B», P')
ENDWHILE

# PASO 3: Preservación de variables positivas #
POS ← {A ∈ V' / «A → w» ∈ P' con w ∈ T*}
REPEAT
   IF (∃ «A → α» ∈ P' / A ∉ POS ∧ α ∈ (POS ∪ T)*) THEN
      AÑADIR(A, POS)
   ENDIF
UNTIL ((POS no cambia) AND (Todas las reglas chequeadas))
Remover de P' aquellas reglas que involucren variables fuera de POS
V' ← POS

# PASO 4: Preservación de variables alcanzables #
ALC ← {S}
REPEAT
   IF (∃ «A → αBβ» ∈ P' / A ∈ ALC ∧ B ∉ ALC) THEN
      AÑADIR(B, ALC)
   ENDIF
UNTIL ((ALC no cambia) AND (Todas las reglas chequeadas))
Remover de P' aquellas reglas que involucren variables fuera de ALC
V' ← ALC
```
