

CC30B - Fundamentos de la Ciencia de la Computación

(Lenguajes Formales, Computabilidad y Complejidad)

Apuntes y Ejercicios

Profesor: Gonzalo Navarro

Departamento de Ciencias de la Computación
Universidad de Chile

19 de octubre de 2006

Índice General

1	Conceptos Básicos	5
1.1	Inducción Estructural	5
1.2	Conjuntos, Relaciones y Funciones	6
1.3	Cardinalidad	7
1.4	Alfabetos, Cadenas y Lenguajes	10
1.5	Especificación Finita de Lenguajes	11
2	Lenguajes Regulares	13
2.1	Expresiones Regulares (ERs)	13
2.2	Autómatas Finitos Determinísticos (AFDs)	15
2.3	Autómatas Finitos No Determinísticos (AFNDs)	20
2.4	Conversión de ER a AFND	22
2.5	Conversión de AFND a AFD	24
2.6	Conversión de AFD a ER	26
2.7	Propiedades de Clausura	28
2.8	Lema de Bombeo	29
2.9	Propiedades Algorítmicas de Lenguajes Regulares	31
2.10	Ejercicios	31
2.11	Preguntas de Controles	35
2.12	Proyectos	39
3	Lenguajes Libres del Contexto	41
3.1	Gramáticas Libres del Contexto (GLCs)	41
3.2	Todo Lenguaje Regular es Libre del Contexto	46
3.3	Autómatas de Pila (AP)	47
3.4	Conversión de GLC a AP	51
3.5	Conversión a AP a GLC	52
3.6	Teorema de Bombeo	55
3.7	Propiedades de Clausura	57
3.8	Propiedades Algorítmicas	57
3.9	Determinismo y Parsing	59

3.10	Ejercicios	65
3.11	Preguntas de Controles	67
3.12	Proyectos	70
4	Máquinas de Turing y la Tesis de Church	73
4.1	La Máquina de Turing (MT)	73
4.2	Protocolos para Usar MTs	76
4.3	Notación Modular	79
4.4	MTs de k Cintas y Otras Extensiones	83
4.5	MTs no Determinísticas (MTNDs)	88
4.6	La Máquina Universal de Turing (MUT)	94
4.7	La Tesis de Church	98
4.8	Gramáticas Dependientes del Contexto (GDC)	101
4.9	Ejercicios	105
4.10	Preguntas de Controles	106
4.11	Proyectos	108
5	Computabilidad	111
6	Complejidad Computacional	113

Capítulo 1

Conceptos Básicos

[LP81, cap 1]

En este capítulo repasaremos brevemente conceptos elementales que el lector ya debiera conocer, y luego introduciremos elementos más relacionados con la materia. La mayoría de las definiciones, lemas, etc. de este capítulo no están indexados en el Índice de Materias al final del apunte, pues son demasiado básicos. Indexamos sólo lo que se refiere al tema específico de lenguajes formales, complejidad, y computabilidad.

No repasaremos el lenguaje de la lógica de predicados de primer orden, que usaremos directamente, ni nada sobre números.

1.1 Inducción Estructural

En muchas pruebas del curso haremos inducción sobre estructuras definidas recursivamente. La *inducción natural* que se supone que el lector ya conoce, $(P(0) \wedge (P(n) \Rightarrow P(n+1))) \Rightarrow \forall n \geq 0, P(n)$, puede extenderse a estas estructuras recursivas. Esencialmente lo que se hace es aplicar inducción natural sobre alguna propiedad de la estructura (como su tamaño), de modo que pueda suponerse que la propiedad vale para todas sus subestructuras.

Veamos un ejemplo. Un árbol binario es o bien un nodo hoja o bien un nodo interno del que cuelgan dos árboles binarios. Llamemos $i(A)$ y $h(A)$ a la cantidad de nodos internos y nodos hojas, respectivamente, de un árbol binario A . Demostremos por inducción estructural que, para todo árbol binario A , $i(A) = h(A) - 1$.

Caso base: Si el árbol A es un nodo hoja, entonces tiene cero nodos internos y una hoja, y la proposición vale pues $i(A) = 0$ y $h(A) = 1$.

Caso inductivo: Si el árbol A es un nodo interno del que cuelgan subárboles A_1 y A_2 , tenemos por hipótesis inductiva que $i(A_1) = h(A_1) - 1$ y $i(A_2) = h(A_2) - 1$. Ahora bien, los nodos de A son los de A_1 , los de A_2 , y un nuevo nodo interno. De modo que $i(A) = i(A_1) + i(A_2) + 1$ y $h(A) = h(A_1) + h(A_2)$. De aquí que $i(A) = h(A_1) - 1 + h(A_2) - 1 + 1 = h(A_1) + h(A_2) - 1 = h(A) - 1$ y hemos terminado.

1.2 Conjuntos, Relaciones y Funciones

Definición 1.1 Un conjunto A es una colección finita o infinita de objetos. Se dice que esos objetos pertenecen al conjunto, $x \in A$. Una condición lógica equivalente a $x \in A$ define el conjunto A .

Definición 1.2 Un conjunto B es subconjunto de un conjunto A , $B \subseteq A$, si $x \in B \Rightarrow x \in A$. Si además $B \neq A$, se puede decir $B \subset A$.

Definición 1.3 Algunas operaciones posibles sobre dos conjuntos A y B son:

1. Unión: $x \in A \cup B$ sii $x \in A \vee x \in B$.
2. Intersección: $x \in A \cap B$ sii $x \in A \wedge x \in B$.
3. Diferencia: $x \in A - B$ sii $x \in A \wedge x \notin B$.
4. Producto: $(x, y) \in A \times B$ sii $x \in A \wedge y \in B$.
5. Conjunto vacío: $\forall x, x \notin \emptyset$.

Definición 1.4 Una partición de un conjunto A es un conjunto de conjuntos B_1, \dots, B_n tal que $A = \bigcup_{0 \leq i \leq n} B_i$ y $B_i \cap B_j = \emptyset$ para todo $i \neq j$.

Definición 1.5 Una relación \mathcal{R} entre dos conjuntos A y B , es un subconjunto de $A \times B$. Si $(a, b) \in \mathcal{R}$ se dice también $a\mathcal{R}b$.

Definición 1.6 Algunas propiedades que puede tener una relación $\mathcal{R} \subseteq A \times A$ son:

- Reflexividad: $\forall a \in A, a\mathcal{R}a$.
- Simetría: $\forall a, b \in A, a\mathcal{R}b \Rightarrow b\mathcal{R}a$.
- Transitividad: $\forall a, b, c \in A, a\mathcal{R}b \wedge b\mathcal{R}c \Rightarrow a\mathcal{R}c$.
- Antisimetría: $\forall a \neq b \in A, a\mathcal{R}b \Rightarrow \neg b\mathcal{R}a$.

Definición 1.7 Algunos tipos de relaciones, según las propiedades que cumplen, son:

- de Equivalencia: Reflexiva, simétrica y transitiva.
- de Preorden: Reflexiva y transitiva.
- de Orden: Reflexiva, antisimétrica y transitiva.

Definición 1.8 Una relación de equivalencia \equiv en A (o sea $\equiv \subseteq A \times A$) particiona A en clases de equivalencia, de modo que $a, a' \in A$ están en la misma clase sii $a \equiv a'$. Al conjunto de las clases de equivalencia, A/\equiv , se lo llama conjunto cociente.

Definición 1.9 Clausurar una relación $\mathcal{R} \subseteq A \times A$ es agregarle la mínima cantidad de elementos necesaria para que cumpla una cierta propiedad.

- Clausura reflexiva: es la menor relación reflexiva que contiene \mathcal{R} (“menor” en sentido de que no contiene otra, vista como conjunto). Para obtenerla basta incluir todos los pares (a, a) , $a \in A$, en \mathcal{R} .
- Clausura transitiva: es la menor relación transitiva que contiene \mathcal{R} . Para obtenerla deben incluirse todos los pares (a, c) tales que $(a, b) \in \mathcal{R}$ y $(b, c) \in \mathcal{R}$. Deben considerarse también los nuevos pares que se van agregando!

Definición 1.10 Una función $f : A \longrightarrow B$ es una relación en $A \times B$ que cumple que $\forall a \in A, \exists! b \in B, a f b$. A ese único b se lo llama $f(a)$. A se llama el dominio y $\{f(a), a \in A\} \subseteq B$ la imagen de f .

Definición 1.11 Una función $f : A \longrightarrow B$ es:

- inyectiva si $a \neq a' \Rightarrow f(a) \neq f(a')$.
- suryectiva si $\forall b \in B, \exists a \in A, f(a) = b$.
- biyectiva si es inyectiva y suryectiva.

1.3 Cardinalidad

La cardinalidad de un conjunto finito es simplemente la cantidad de elementos que tiene. Esto es más complejo para conjuntos infinitos. Deben darse nombres especiales a estas cardinalidades, y no todas las cardinalidades infinitas son iguales.

Definición 1.12 La cardinalidad de un conjunto A se escribe $|A|$. Si A es finito, entonces $|A|$ es un número natural igual a la cantidad de elementos que pertenecen a A .

Definición 1.13

Se dice que $|A| \leq |B|$ si existe una función $f : A \longrightarrow B$ inyectiva.

Se dice que $|A| \geq |B|$ si existe una función $f : A \longrightarrow B$ suryectiva.

Se dice que $|A| = |B|$ si existe una función $f : A \longrightarrow B$ biyectiva.

Se dice $|A| < |B|$ si $|A| \leq |B|$ y no vale $|A| = |B|$; similarmente con $|A| > |B|$.

Definición 1.14 A la cardinalidad de \mathbb{N} se la llama $|\mathbb{N}| = \aleph_0$ (alef sub cero). A todo conjunto de cardinal $\leq \aleph_0$ se le dice numerable.

Observación 1.1 *Un conjunto numerable A , por definición, admite una suryección $f : \mathbb{N} \longrightarrow A$, o lo que es lo mismo, es posible listar los elementos de A en orden $f(0), f(1), f(2), \dots$ de manera que todo elemento de A se mencione alguna vez. De modo que para demostrar que A es numerable basta exhibir una forma de listar sus elementos y mostrar que todo elemento será listado en algún momento.*

Teorema 1.1 \aleph_0 es el menor cardinal infinito. Más precisamente, todo A tal que $|A| \leq \aleph_0$ cumple que $|A|$ es finito o $|A| = \aleph_0$.

Prueba: Si A es infinito, entonces $|A| > n$ para cualquier $n \geq 0$. Es decir, podemos definir subconjuntos $A_n \subset A$, $|A_n| = n$, para cada $n \geq 0$, de modo que $A_{n-1} \subseteq A_n$. Sea a_n el único elemento de $A_n - A_{n-1}$. Entonces todos los a_n son distintos y podemos hacer una suryección de $\{a_1, a_2, \dots\}$ en \mathbb{N} . \square

Observación 1.2 *El que $A \subset B$ no implica que $|A| < |B|$ en conjuntos infinitos. Por ejemplo el conjunto de los pares es del mismo cardinal que el de los naturales, mediante la biyección $f(n) = 2n$.*

Definición 1.15 Si $|A| = \aleph_i$, se llama $\aleph_{i+1} = |\wp(A)|$.

Definición 1.16 La hipótesis del continuo establece que no existe ningún conjunto cuyo cardinal esté entre \aleph_i y \aleph_{i+1} , es decir, si $\aleph_i \leq |A| \leq \aleph_{i+1}$, entonces $|A| = \aleph_i$ o $|A| = \aleph_{i+1}$. Se ha probado que esta hipótesis no se puede probar ni refutar con los axiomas usuales de la teoría de conjuntos, sino que debe introducirse (ella o su negación) como axioma.

Teorema 1.2 *El cardinal de $\wp(\mathbb{N})$ es estrictamente mayor que el de \mathbb{N} , o en otras palabras, $\aleph_0 < \aleph_1$. Esto puede probarse en general, $\aleph_i < \aleph_{i+1}$.*

Prueba: Es fácil ver, mediante biyecciones, que los siguientes conjuntos tienen el mismo cardinal que $\wp(\mathbb{N})$:

1. Las secuencias infinitas de bits, haciendo la biyección con $\wp(\mathbb{N})$ dada por: el i -ésimo bit es 1 sii $i - 1$ pertenece al subconjunto.
2. Las funciones $f : \mathbb{N} \longrightarrow \{0, 1\}$, haciendo la biyección con el punto 1; $F(f) = f(0)f(1)f(2)\dots$ es una secuencia infinita de bits que describe unívocamente a f .
3. Los números reales $0 \leq x < 1$: basta escribirlos en binario de la forma $0.01101\dots$, para tener la biyección con las secuencias infinitas de bits. Hay algunas sutilezas debido a que $0.001111\dots = 0.010000\dots$, pero pueden remediarse.
4. Los reales mismos, \mathbb{R} , mediante alguna función biyectiva con $[0, 1)$ (punto 3). Hay varias funciones trigonométricas, como la tangente, que sirven fácilmente a este propósito.

Utilizaremos el *método de diagonalización de Kantor* para demostrar que las secuencias infinitas de bits no son numerables. Supondremos, por contradicción, que podemos hacer una lista de todas las secuencias de bits, B_1, B_2, B_3, \dots , donde B_i es la i -ésima secuencia y $B_i(j)$ es el j -ésimo bit de B_i . Definamos ahora la secuencia de bits $X = \overline{B_1(1)} \overline{B_2(2)} \dots$, donde $\overline{0} = 1$ y $\overline{1} = 0$. Como $X(i) = \overline{B_i(i)} \neq B_i(i)$, se deduce que $X \neq B_i$ para todo B_i . Entonces X es una secuencia de bits que no aparece en la lista. Para cualquier listado, podemos generar un elemento que no aparece, por lo cual no puede existir un listado exhaustivo de todas las secuencias infinitas de bits. \square

Lema 1.1 Sean A y B numerables. Los siguientes conjuntos son numerables:

1. $A \cup B$.
2. $A \times B$.
3. A^k , donde $A^1 = A$ y $A^k = A \times A^{k-1}$.
4. $\bigcup A_i$, donde todos los A_i son numerables.
5. $A^+ = A^1 \cup A^2 \cup A^3 \cup \dots$

Prueba: Sean a_1, a_2, \dots y b_1, b_2, \dots listados que mencionan todos los elementos de A y B , respectivamente.

1. $a_1, b_1, a_2, b_2, \dots$ lista $A \cup B$ y todo elemento aparece en la lista alguna vez. Si $A \cap B \neq \emptyset$ esta lista puede tener repeticiones, pero eso está permitido.
2. No podemos listar $(a_1, b_1), (a_1, b_2), (a_1, b_3), \dots$ porque por ejemplo nunca llegaríamos a listar (a_2, b_1) . Debemos aplicar un recorrido sobre la matriz de índices de modo que a toda celda (a_i, b_j) le llegue su turno. Por ejemplo, por diagonales ($i + j$ creciente): (a_1, b_1) , luego $(a_2, b_1), (a_1, b_2)$, luego $(a_3, b_1), (a_2, b_2), (a_1, b_3)$, y así sucesivamente.
3. Por inducción sobre k y usando el punto 2.
4. Sea $a_i(j)$ el j -ésimo elemento de lista que numera A_i . Nuevamente se trata de recorrer una matriz para que le llegue el turno a todo $a_i(j)$, y se resuelve como el punto 2.
5. Es una unión de una cantidad numerable de conjuntos, donde cada uno de ellos es numerable por el punto 3, de modo que se puede aplicar el punto 4. Si esto parece demasiado esotérico, podemos expresar la solución concretamente: listemos el elemento 1 de A^1 ; luego el 2 de A y el 1 de A^2 ; luego el 3 de A , el 2 de A^2 y el 1 de A^3 ; etc. Está claro que a cada elemento de cada conjunto le llegará su turno.

\square

Observación 1.3 El último punto del Lema 1.1 se refiere al conjunto de todas las secuencias finitas donde los elementos pertenecen a un conjunto numerable. Si esto es numerable, está claro que las secuencias finitas de elementos de un conjunto finito también lo son. Curiosamente, las secuencias infinitas no son numerables, ni siquiera sobre conjuntos finitos, como se vió para el caso de bits en el Teo. 1.2.

Notablemente, aún sin haber visto casi nada de computabilidad, podemos establecer un resultado que nos plantea un desafío para el resto del curso:

Teorema 1.3 *Dado cualquier lenguaje de programación, existen funciones de los enteros que no se pueden calcular con ningún programa escrito en ese lenguaje.*

Prueba: Incluso restringiéndonos a las funciones que dado un entero deben responder “sí” o “no” (por ejemplo, ¿es n primo?), hay una cantidad no numerable de funciones $f : \mathbb{N} \rightarrow \{0, 1\}$. Todos los programas que se pueden escribir en su lenguaje de programación favorito, en cambio, son secuencias finitas de símbolos (ASCII, por ejemplo). Por lo tanto hay sólo una cantidad numerable de programas posibles. \square

Mucho más difícil será exhibir una función que no se pueda calcular, pero es interesante que la inmensa mayoría efectivamente no se puede calcular. *En realidad esto es un hecho más básico aún, por ejemplo la inmensa mayoría de los números reales no puede escribirse en ningún formalismo que consista de secuencias de símbolos sobre un alfabeto numerable.*

1.4 Alfabetos, Cadenas y Lenguajes

En esta sección introducimos notación más específica del curso. Comenzaremos por definir lo que es un alfabeto.

Definición 1.17 *Llamaremos alfabeto a cualquier conjunto finito no vacío. Usualmente lo denotaremos como Σ . Los elementos de Σ se llamarán símbolos o caracteres.*

Si bien normalmente usaremos alfabetos intuitivos como $\{0, 1\}$, $\{a, b\}$, $\{a \dots z\}$, $\{0 \dots 9\}$, etc., algunas veces usaremos conjuntos más sofisticados como alfabetos.

Definición 1.18 *Llamaremos cadena a una secuencia finita de símbolos de un alfabeto Σ , es decir, a un elemento de*

$$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \dots$$

donde $\Sigma^1 = \Sigma$ y $\Sigma^k = \Sigma \times \Sigma^{k-1}$. Σ^* denota, entonces, el conjunto de todas las secuencias finitas de símbolos de Σ . El conjunto Σ^0 es especial, tiene un sólo elemento llamado ε , que corresponde a la cadena vacía. Si una cadena $x \in \Sigma^k$ entonces decimos que su largo es $|x| = k$ (por ello $|\varepsilon| = 0$).

Observación 1.4 *Es fácil confundir entre una cadena de largo 1, $x = (a)$, y un carácter a . Normalmente nos permitiremos identificar ambas cosas.*

Definición 1.19 Una cadena x sobre Σ se escribirá juntaponiendo sus caracteres uno luego del otro, es decir $(a_1, a_2, \dots, a_{|x|})$, $a_i \in \Sigma$, se escribirá como $x = a_1 a_2 \dots a_{|x|}$. La concatenación de dos cadenas $x = a_1 a_2 \dots a_n$ e $y = b_1 b_2 \dots b_m$, se escribe $xy = a_1 a_2 \dots a_n b_1 b_2 \dots b_m$, $|xy| = |x| + |y|$. Finalmente usaremos x^k para denotar k concatenaciones sucesivas de x , es decir $x^0 = \varepsilon$ y $x^k = x x^{k-1}$.

Definición 1.20 Dadas cadenas x, y, z , diremos que x es un prefijo de xy , un sufijo de yx , y una subcadena o substring de yxz .

Definición 1.21 Un lenguaje sobre un alfabeto Σ es cualquier subconjunto de Σ^* .

Observación 1.5 El conjunto de todas las cadenas sobre cualquier alfabeto es numerable, $|\Sigma^*| = \aleph_0$, y por lo tanto todo lenguaje sobre un alfabeto finito (e incluso numerable) Σ es numerable. Sin embargo, la cantidad de lenguajes distintos es no numerable, pues es $|\wp(\Sigma^*)| = \aleph_1$.

Cualquier operación sobre conjuntos puede realizarse sobre lenguajes también. Definamos ahora algunas operaciones específicas sobre lenguajes.

Definición 1.22 Algunas operaciones aplicables a lenguajes sobre un alfabeto Σ son:

1. Concatenación: $L_1 \circ L_2 = \{xy, x \in L_1, y \in L_2\}$.
2. Potencia: $L^0 = \{\varepsilon\}$, $L^k = L \circ L^{k-1}$.
3. Clausura de Kleene: $L^* = \bigcup_{k \geq 0} L^k$.
4. Complemento: $L^c = \Sigma^* - L$.

1.5 Especificación Finita de Lenguajes

Si un lenguaje L es finito, se puede especificar por extensión, como $L_1 = \{aba, bbbbb, aa\}$. Si es infinito, se puede especificar mediante predicados, por ejemplo $L_2 = \{a^p, p \text{ es primo}\}$. Este mecanismo es poderoso, pero no permite tener una idea de la complejidad del lenguaje, en el sentido de cuán difícil es determinar si una cadena pertenece o no a L , o de enumerar las cadenas de L . Con L_1 esto es trivial, y con L_2 perfectamente factible. Pero ahora consideremos $L_3 = \{a^n, n \geq 0, \exists x, y, z \in \mathbb{N} - \{0\}, x^n + y^n = z^n\}$. L_3 está correctamente especificado, pero ¿ $aaa \in L_3$? Recién con la demostración del último Teorema de Fermat en 1995 (luego de más de 3 siglos de esfuerzos), se puede establecer que $L = \{a, aa\}$. Similarmente, se puede especificar $L_4 = \{w, w \text{ es un teorema de la teoría de números}\}$, y responder si $w \in L_4$ equivale a demostrar un teorema.

El tema central de este curso se puede ver como la búsqueda de descripciones finitas para lenguajes infinitos, de modo que sea posible determinar *mecánicamente* si una cadena

está en el lenguaje. ¿Qué interés tiene esto? No es difícil identificar *lenguajes* con *problemas de decisión*. Por ejemplo, la pregunta ¿el grafo G es bipartito? se puede traducir a una pregunta de tipo ¿ $w \in L$?, donde L es el conjunto de cadenas que representan los grafos bipartitos (*representados como una secuencia de alguna manera, finalmente todo son secuencias de bits en el computador!*), y w es la representación de G . Determinar que ciertos lenguajes no pueden decidirse mecánicamente equivale a determinar que ciertos problemas no pueden resolverse por computador.

El siguiente teorema, nuevamente, nos dice que la mayoría de los lenguajes no puede decidirse, en el sentido de poder decir si una cadena dada le pertenece o no. Nuevamente, es un desafío encontrar un ejemplo.

Teorema 1.4 *Dado cualquier lenguaje de programación, existen lenguajes que no pueden decidirse con ningún programa.*

Prueba: Nuevamente, la cantidad de lenguajes es no numerable y la de programas que se pueden escribir es numerable. \square

En el curso veremos mecanismos progresivamente más potentes para describir lenguajes cada vez más sofisticados y encontraremos los límites de lo que puede resolverse por computador. Varias de las cosas que veremos en el camino tienen además muchas aplicaciones prácticas.

Capítulo 2

Lenguajes Regulares

[LP81, sec 1.9 y cap 2]

En este capítulo estudiaremos una forma particularmente popular de representación finita de lenguajes. Los lenguajes regulares son interesantes por su simplicidad, la que permite manipularlos fácilmente, y a la vez porque incluyen muchos lenguajes relevantes en la práctica. Los mecanismos de búsqueda provistos por varios editores de texto (`vi`, `emacs`), así como por el shell de `Unix` y todas las herramientas asociadas para procesamiento de texto (`sed`, `awk`, `perl`), se basan en lenguajes regulares. Los lenguajes regulares también se usan en biología computacional para búsqueda en secuencias de ADN o proteínas (por ejemplo patrones PROSITE).

Los lenguajes regulares se pueden describir usando tres mecanismos distintos: expresiones regulares (ERs), autómatas finitos determinísticos (AFDs) y no determinísticos (AFNDs). Algunos de los mecanismos son buenos para describir lenguajes, y otros para implementar reconocedores eficientes.

2.1 Expresiones Regulares (ERs)

[LP81, sec 1.9]

Definición 2.1 Una expresión regular (ER) sobre un alfabeto finito Σ se define recursivamente como sigue:

1. Para todo $c \in \Sigma$, c es una ER.
2. Φ es una ER.
3. Si E_1 y E_2 son ERs, $E_1 \mid E_2$ es una ER.
4. Si E_1 y E_2 son ERs, $E_1 \cdot E_2$ es una ER.
5. Si E_1 es una ER, E_1^* es una ER.
6. Si E_1 es una ER, (E_1) es una ER.

Cuando se lee una expresión regular, hay que saber qué operador debe leerse primero. Esto se llama *precedencia*. Por ejemplo, la expresión $a \mid b \cdot c \star$, ¿debe entenderse como (1) la “ \star ” aplicada al resto? (2) ¿la “ \mid ” aplicada al resto? (3) ¿la “ \cdot ” aplicada al resto? La respuesta es que, primero que nada se aplican los “ \star ”, segundo los “ \cdot ”, y finalmente los “ \mid ”. Esto se expresa diciendo que el orden de precedencia es \star, \cdot, \mid . Los paréntesis sirven para alterar la precedencia. Por ejemplo, la expresión anterior, dado el orden de precedencia que establecimos, es equivalente a $a \mid (b \cdot (c \star))$. Se puede forzar otro orden de lectura de la ER cambiando los paréntesis, por ejemplo $(a \mid b) \cdot c \star$.

Asimismo, debe aclararse cómo se lee algo como $a \mid b \mid c$, es decir ¿cuál de los dos “ \mid ” se lee primero? Convengamos que en ambos operadores binarios se lee primero el de más a la izquierda (se dice que el operador “asocia a la izquierda”), pero realmente no es importante, por razones que veremos enseguida.

Observar que aún no hemos dicho *qué significa* una ER, sólo hemos dado su *sintaxis* pero no su *semántica*. De esto nos encargamos a continuación.

Definición 2.2 El lenguaje descrito por una ER E , $\mathcal{L}(E)$, se define recursivamente como sigue:

1. Si $c \in \Sigma$, $\mathcal{L}(c) = \{c\}$. Esto es un conjunto de una sola cadena de una sola letra.
2. $\mathcal{L}(\Phi) = \emptyset$.
3. $\mathcal{L}(E_1 \mid E_2) = \mathcal{L}(E_1) \cup \mathcal{L}(E_2)$.
4. $\mathcal{L}(E_1 \cdot E_2) = \mathcal{L}(E_1) \circ \mathcal{L}(E_2)$.
5. $\mathcal{L}(E_1 \star) = \mathcal{L}(E_1)^*$.

Notar que $\mathcal{L}(a \cdot b \cdot c \cdot d) = \{abcd\}$, por lo cual es común ignorar el símbolo “ \cdot ” y simplemente juxtaponer los símbolos uno después del otro. Notar también que, dado que “ \mid ” y “ \cdot ” se mapean a operadores asociativos, no es relevante si asocian a izquierda o a derecha.

Observación 2.1 Por definición de clausura de Kleene, $\mathcal{L}(\Phi \star) = \{\varepsilon\}$. Por ello, a pesar de no estar formalmente en la definición, permitiremos escribir ε como una expresión regular.

Definición 2.3 Un lenguaje L es regular si existe una ER E tal que $L = \mathcal{L}(E)$.

Ejemplo 2.1 ¿Cómo se podría escribir una ER para las cadenas de a ’s y b ’s que contuvieran una cantidad impar de b ’s? Una solución es $a \star (ba \star ba \star) \star ba \star$, donde lo más importante es la clausura de Kleene mayor, que encierra secuencias donde nos aseguramos que las b ’s vienen de a pares, separadas por cuantas a ’s se quieran. La primera clausura $(a \star)$ permite que la secuencia empiece con a ’s y la última agrega la b que hace que el total sea impar y además permite que haya a ’s al final. Es un buen ejercicio jugar con otras soluciones y comentarlas, por ejemplo $(a \star ba \star ba \star) \star ba \star$. Es fácil ver cómo generalizar este ejemplo para que la cantidad de b ’s módulo k sea r .

Algo importante en el Ej. 2.1 es cómo asegurarnos de que la ER realmente representa el lenguaje L que creemos. La técnica para esto tiene dos partes: (i) ver que toda cadena generada está en L ; (ii) ver que toda cadena de L se puede generar con la ER. En el Ej. 2.1 eso podría hacerse de la siguiente manera: Para (i) basta ver que la clausura de Kleene introduce las b 's de a dos, de modo que toda cadena generada por la ER tendrá una cantidad impar de b 's. Para (ii), se debe tomar una cadena cualquiera x con una cantidad impar de b 's y ver que la ER puede generarla. Esto no es difícil si consideramos las subcadenas de x que van desde una b impar (1era, 3era, ...) hasta la siguiente, y mostramos que cada una de esas subcadenas se pueden generar con $ba \star bba \star$. El resto es sencillo. Un ejemplo un poco más complicado es el siguiente.

Ejemplo 2.2 ¿Cómo se podría escribir una ER para las cadenas de a 's y b 's que nunca contuvieran tres b 's seguidas? Una solución parece ser $(a|ba|bba) \star$, pero ¿está correcta? Si se analiza rigurosamente, se notará que esta ER no permite que las cadenas terminen con b , por lo cual deberemos corregirla a $(a|ba|bba) \star (\varepsilon|b|bb)$.

Ejemplo 2.3 ¿Cómo se describiría el lenguaje denotado por la expresión regular $(ab|aba) \star$? Son las cadenas que se pueden descomponer en secuencias ab o aba . Describir con palabras el lenguaje denotado por una ER es un arte. En el Ej. 2.1, que empiezo con una bonita descripción concisa, uno podría caer en una descripción larga y mecánica de lo que significa la ER, como “primero viene una secuencia de a 's; después, varias veces, viene una b y una secuencia de a 's, dos veces; después...”. En general una descripción más concisa es mejor.

Ejemplo 2.4 ¿Se podría escribir una ER que denotara los números decimales que son múltiplos de 7? (es decir 7, 14, 21, ...) Sí, pero intentarlo directamente es una empresa temeraria. Veremos más adelante cómo lograrlo.

Observación 2.2 *Debería ser evidente que no todos los lenguajes que se me ocurran pueden ser descritos con ERs, pues la cantidad de lenguajes distintos sobre un alfabeto finito es no numerable, mientras que la cantidad de ERs es numerable. Otra cosa es encontrar lenguajes concretos no expresables con ERs y poder demostrar que no lo son.*

Ejemplo 2.5 ¿Se podría escribir una ER que denotara las cadenas de a 's cuyo largo es un número primo? No, no se puede. Veremos más adelante cómo demostrar que no se puede.

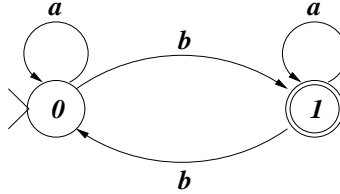
Ejemplo 2.6 Algunas aplicaciones prácticas donde se usan ERs es en la especificación de fechas, direcciones IP, tags XML, nombres de variables en Java, números en notación flotante, direcciones de email, etc. Son ejercicios interesantes, aunque algunos son algo tediosos.

2.2 Autómatas Finitos Determinísticos (AFDs)

[LP81, sec 2.1]

Un AFD es otro mecanismo para describir lenguajes. En vez de pensar en *generar* las cadenas (como las ERs), un AFD describe un lenguaje mediante *reconocer* las cadenas del lenguaje, y ninguna otra. El siguiente ejemplo ilustra un AFD.

Ejemplo 2.7 El AFD que reconoce el mismo lenguaje del Ej. 2.1 se puede graficar de la siguiente forma.

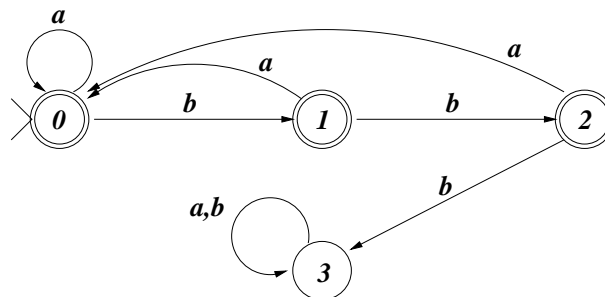


El AFD que hemos dibujado se interpreta de la siguiente manera. Los nodos del grafo son *estados*. El apuntado con un ángulo \rangle es el *estado inicial*, en el que empieza la computación. Estando en un estado, el AFD lee una letra de la entrada y, según indique la flecha (llamada *transición*), pasa a otro estado (siempre debe haber exactamente una flecha saliendo de cada estado por cada letra). Cuando se lee toda la cadena, el AFD la acepta o no según el estado al que haya llegado sea *final* o no. Los estados finales se dibujan con doble círculo.

En este AFD pasa algo que, más o menos explícitamente, siempre ocurre. Cada estado se puede asociar a un *invariante*, es decir, una afirmación sobre la cadena leída hasta ese momento. En nuestro caso el estado inicial corresponde al invariante “se ha visto una cantidad par de b 's hasta ahora”, mientras que el estado final corresponde a “se ha visto una cantidad impar de b 's hasta ahora”.

El siguiente ejemplo muestra la utilidad de esta visión. La correctitud de un AFD con respecto a un cierto lenguaje L que se pretende representar se puede demostrar a partir de establecer los invariantes, ver que los estados finales, unidos (pues puede haber más de uno), describen L , y que las flechas pasan correctamente de un invariante a otro.

Ejemplo 2.8 El AFD que reconoce el mismo lenguaje del Ej. 2.2 se puede graficar de la siguiente forma. Es un buen ejercicio describir el invariante que le corresponde a cada estado. Se ve además que puede haber varios estados finales. El estado 3 se llama *sumidero*, porque una vez caído en él, el AFD no puede salir y no puede aceptar la cadena.



Es hora de definir formalmente lo que es un AFD.

Definición 2.4 Un autómata finito determinístico (AFD) es una tupla $M = (K, \Sigma, \delta, s, F)$, tal que

- K es un conjunto finito de estados.
- Σ es un alfabeto finito.
- $s \in K$ es el estado inicial
- $F \subseteq K$ son los estados finales.
- $\delta : K \times \Sigma \longrightarrow K$ es la función de transición.

Ejemplo 2.9 El AFD del Ej. 2.7 se describe formalmente como $M = (K, \Sigma, \delta, s, F)$, donde $K = \{0, 1\}$, $\Sigma = \{a, b\}$, $s = 0$, $F = \{1\}$, y la función δ como sigue:

δ	0	1
a	0	1
b	1	0

No hemos descrito aún formalmente cómo funciona un AFD. Para ello necesitamos la noción de *configuración*, que contiene la información necesaria para completar el cómputo de un AFD.

Definición 2.5 Una configuración de un AFD $M = (K, \Sigma, \delta, s, F)$ es un elemento de $\mathcal{C}_M = K \times \Sigma^*$.

La idea es que la configuración (q, x) indica que M está en el estado q y le falta leer la cadena x de la entrada. Esta es información suficiente para predecir lo que ocurrirá en el futuro. Lo siguiente es describir cómo el AFD nos lleva de una configuración a la siguiente.

Definición 2.6 La relación lleva en un paso, $\vdash_M \subseteq \mathcal{C}_M \times \mathcal{C}_M$ se define de la siguiente manera: $(q, ax) \vdash_M (q', x)$, donde $a \in \Sigma$, sii $\delta(q, a) = q'$.

Escribiremos simplemente \vdash en vez de \vdash_M cuando quede claro de qué M estamos hablando.

Definición 2.7 La relación lleva en cero o más pasos \vdash_M^* es la clausura reflexiva y transitiva de \vdash_M .

Ya estamos en condiciones de definir el lenguaje aceptado por un AFD. La idea es que si el AFD es llevado del estado inicial a uno final por la cadena x , entonces la reconoce.

Definición 2.8 El lenguaje aceptado por un AFD $M = (K, \Sigma, \delta, s, F)$ se define como

$$\mathcal{L}(M) = \{x \in \Sigma^*, \exists f \in F, (s, x) \vdash_M^* (f, \varepsilon)\}.$$

Ejemplo 2.10 Tomemos el AFD del Ej. 2.8, el que se describe formalmente como $M = (K, \Sigma, \delta, s, F)$, donde $K = \{0, 1, 2, 3\}$, $\Sigma = \{a, b\}$, $s = 0$, $F = \{0, 1, 2\}$, y la función δ como sigue:

δ	0	1	2	3
a	0	0	0	3
b	1	2	3	3

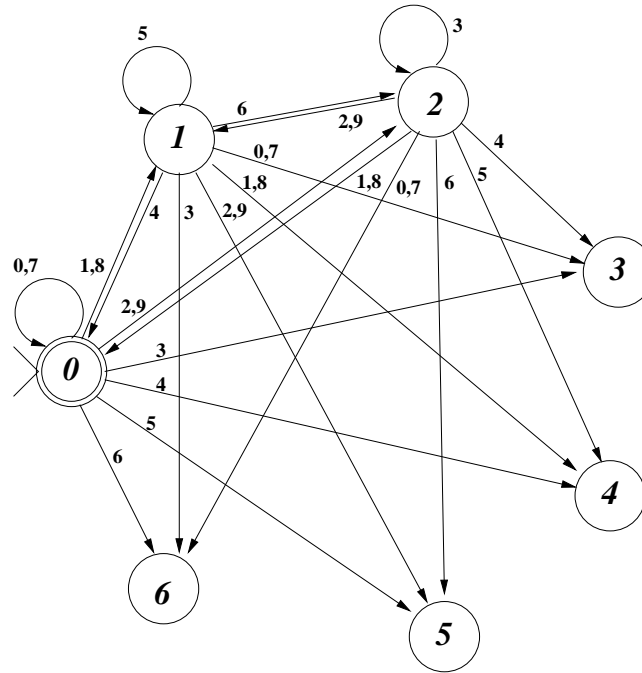
Ahora consideremos la cadena de entrada $x = abbababb$ y escribamos las configuraciones por las que pasa M al recibir x como entrada:

$$(0, abbababb) \vdash (0, bbababb) \vdash (1, bababb) \vdash (2, ababb) \\ \vdash (0, babb) \vdash (1, abb) \vdash (0, bb) \vdash (1, b) \vdash (2, \varepsilon).$$

Por lo tanto $(s, x) \vdash^* (2, \varepsilon)$, y como $2 \in F$, tenemos que $x \in \mathcal{L}(M)$.

Vamos al desafío del Ej. 2.4, el cual resolveremos con un AFD. La visión de invariantes es especialmente útil en este caso.

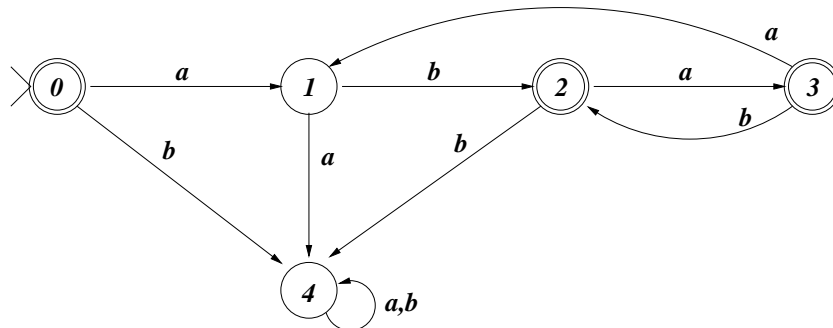
Ejemplo 2.11 El AFD que reconoce el mismo lenguaje del Ej. 2.4 se puede graficar de la siguiente forma. Para no enredar el gráfico de más, sólo se incluyen las flechas que salen de los estados 0, 1 y 2.



El razonamiento es el siguiente. Cada estado representa el resto del número leído hasta ahora, módulo 7. El estado inicial (y final) representa el cero. Si estoy en el estado 2 y viene un 4, significa que el número que leí hasta ahora era $n \equiv 2 \pmod{7}$ y ahora el nuevo número leído es $10 \cdot n + 4 \equiv 10 \cdot 2 + 4 \equiv 24 \equiv 3 \pmod{7}$. Por ello se pasa al estado 3. El lector puede completar las flechas que faltan en el diagrama.

Hemos resuelto usando AFDs un problema que es bastante más complicado usando ERs. El siguiente ejemplo ilustra el caso contrario: el Ej. 2.3, sumamente fácil con ERs, es relativamente complejo con AFDs, y de hecho no es fácil convencerse de su correctitud. El principal problema es, cuando se ha leído ab , determinar si una a que sigue inicia una nueva cadena (pues hemos leído la cadena ab) o es el último carácter de aba .

Ejemplo 2.12 El lenguaje descrito en el Ej. 2.3 se puede reconocer con el siguiente AFD.



2.3 Autómatas Finitos No Determinísticos (AFNDs)

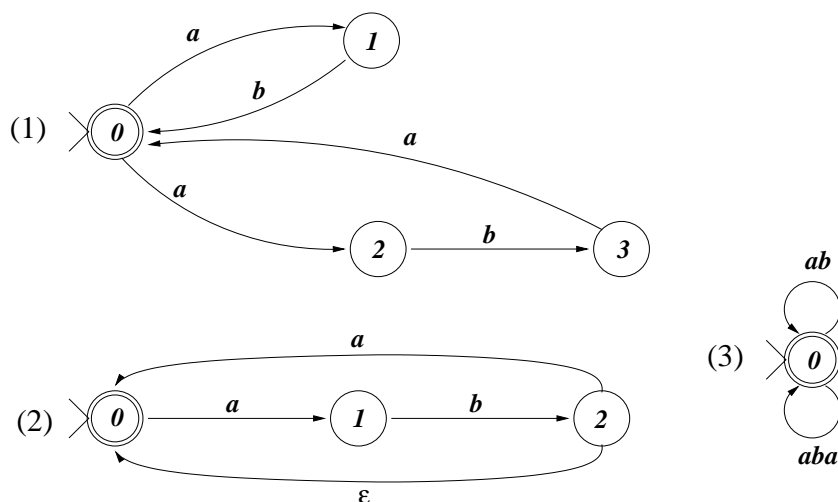
[LP81, sec 2.2]

Dado el estado actual y el siguiente carácter, el AFD pasa exactamente a un siguiente estado. Por eso se lo llama determinístico. Una versión en principio más potente es un AFND, donde frente a un estado actual y un siguiente carácter, es posible tener cero, uno o más estados siguientes.

Hay dos formas posibles de entender cómo funciona un AFND. La primera es pensar que, cuando hay varias alternativas, el AFND elige alguna de ellas. Si *existe una forma* de elegir el siguiente estado que me lleve finalmente a aceptar la cadena, entonces el AFND la aceptará. La segunda forma es imaginarse que el AFND está *en varios estados a la vez* (en todos en los que “puede estar” de acuerdo a la primera visión). Si luego de leer la cadena puede estar en un estado final, acepta la cadena. En cualquier caso, es bueno por un rato no pensar en cómo implementar un AFND.

Una libertad adicional que permitiremos en los AFNDs es la de rotular las transiciones con cadenas, no sólo con caracteres. Tal transición se puede seguir cuando los caracteres de la entrada calzan con la cadena que rotula la transición, consumiendo los caracteres de la entrada. Un caso particularmente relevante es el de las llamadas *transiciones- ϵ* , rotuladas por la cadena vacía. Una transición- ϵ de un estado p a uno q permite activar q siempre que se active p , sin necesidad de leer ningún carácter de la entrada.

Ejemplo 2.13 Según la descripción, es muy fácil definir un AFND que acepte el lenguaje del Ej. 2.3. Se presentan varias alternativas, donde en la (2) y la (3) se hace uso de cadenas rotulando transiciones.



El Ej. 2.13 ilustra en el AFND (3) un punto interesante. Este AFND tiene sólo un estado y éste es final. ¿Cómo puede no aceptar una cadena? Supongamos que recibe como entrada

bb. Parte del estado inicial (y final), y no tiene transiciones para moverse. Queda, pues, en ese estado. ¿Acepta la cadena? No, pues no ha logrado consumirla. Un AFND acepta una cadena cuando *tiene una forma de consumirla y llegar a un estado final*. Es hora de formalizar.

Definición 2.9 Un autómata finito no determinístico (AFND) es una tupla $M = (K, \Sigma, \Delta, s, F)$, tal que

- K es un conjunto finito de estados.
- Σ es un alfabeto finito.
- $s \in K$ es el estado inicial
- $F \subseteq K$ son los estados finales.
- $\Delta \subset_F K \times \Sigma^* \times K$ es la relación de transición, finita.

Ejemplo 2.14 El AFND (2) del Ej. 2.13 se describe formalmente como $M = (K, \Sigma, \Delta, s, F)$, donde $K = \{0, 1, 2\}$, $\Sigma = \{a, b\}$, $s = 0$, $F = \{0\}$, y la relación $\Delta = \{(0, a, 1), (1, b, 2), (2, a, 0), (2, \varepsilon, 0)\}$.

Para describir la semántica de un AFND reutilizaremos la noción de configuración (Def. 2.5). Redefiniremos la relación \vdash_M para el caso de AFNDs.

Definición 2.10 La relación lleva en un paso, $\vdash_M \subseteq \mathcal{C}_M \times \mathcal{C}_M$, donde $M = (K, \Sigma, \Delta, s, F)$ es un AFND, se define de la siguiente manera: $(q, zx) \vdash_M (q', x)$, donde $z \in \Sigma^*$, si $(q, z, q') \in \Delta$.

Nótese que ahora, a partir de una cierta configuración, la relación \vdash nos puede llevar a varias configuraciones distintas, o incluso a ninguna. La clausura reflexiva y transitiva de \vdash_M se llama, nuevamente, *lleva en cero o más pasos*, \vdash_M^* . Finalmente, definimos casi idénticamente al caso de AFDs el lenguaje aceptado por un AFND.

Definición 2.11 El lenguaje aceptado por un AFND $M = (K, \Sigma, \Delta, s, F)$ se define como

$$\mathcal{L}(M) = \{x \in \Sigma^*, \exists f \in F, (s, x) \vdash_M^* (f, \varepsilon)\}.$$

A diferencia del caso de AFDs, dada una cadena x , es posible llegar a varios estados distintos (o a ninguno) luego de haberla consumido. La cadena se declara aceptada si alguno de los estados a los que se llega es final.

Ejemplo 2.15 Consideremos la cadena de entrada $x = ababaababa$ y escribamos las configuraciones por las que pasa el AFND (3) del Ej. 2.13 al recibir x como entrada. En un primer intento:

$$(0, ababaababa) \vdash (0, abaababa) \vdash (0, aababa)$$

no logramos consumir la cadena (por haber “tomado las transiciones incorrectas”). Pero si elegimos otras transiciones:

$$(0, ababaababa) \vdash (0, abaababa) \vdash (0, ababa) \vdash (0, ba) \vdash (0, \varepsilon).$$

Por lo tanto $(s, x) \vdash^* (0, \varepsilon)$, y como $0 \in F$, tenemos que $x \in \mathcal{L}(M)$. Esto es válido a pesar de que existe otro camino por el que $(s, x) \vdash^* (0, aababa)$, de donde no es posible seguir avanzando.

Terminaremos con una nota acerca de cómo simular un AFND. En las siguientes secciones veremos que de hecho los AFNDs pueden convertirse a AFDs, donde es evidente cómo simularlos eficientemente.

Observación 2.3 *Un AFND con n estados y m transiciones puede simularse en un computador en tiempo $O(n + m)$ por cada símbolo de la cadena de entrada. Es un buen ejercicio pensar cómo (tiene que ver con recorrido de grafos, especialmente por las transiciones- ε).*

2.4 Conversión de ER a AFND

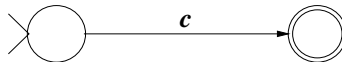
[LP81, sec 2.5]

Como adelantamos, ERs, AFDs y AFNDs son mecanismos equivalentes para denotar los lenguajes regulares. En estas tres secciones demostraremos esto mediante convertir $ER \rightarrow AFND \rightarrow AFD \rightarrow ER$. Las dos primeras conversiones son muy relevantes en la práctica, pues permiten construir verificadores o buscadores eficientes a partir de ERs.

Hay distintas formas de convertir una ER E a un AFND M , de modo que $\mathcal{L}(E) = \mathcal{L}(M)$. Veremos el método de Thompson, que es de los más sencillos.

Definición 2.12 *La función Th convierte ERs en AFNDs según las siguientes reglas.*

1. Para $c \in \Sigma$, $Th(c) =$

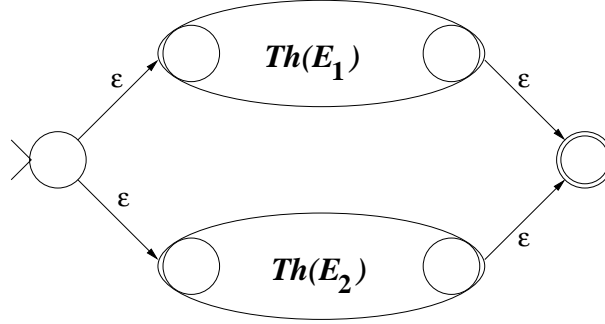


2. $Th(\Phi) =$

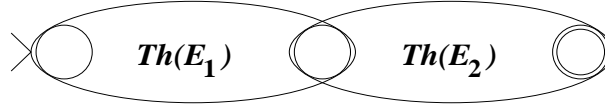


¡Sí, el grafo puede no ser conexo!

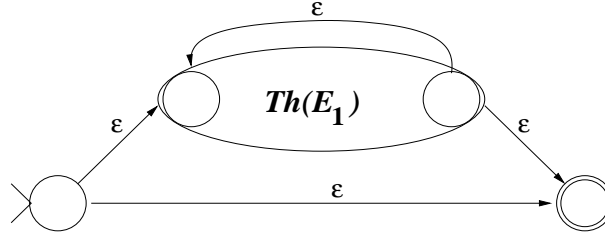
3. $Th(E_1 \mid E_2) =$



4. $Th(E_1 \cdot E_2) =$



5. $Th(E_1 \star) =$



6. $Th((E_1)) = Th(E_1)$.

Observación 2.4 Es fácil, y un buen ejercicio, demostrar varias propiedades de $Th(E)$ por inducción estructural: (i) $Th(E)$ tiene un sólo estado final, distinto del inicial; (ii) $Th(E)$ tiene a lo sumo $2e$ estados y $4e$ aristas, donde e es el número de caracteres en E ; (iii) La cantidad de transiciones que llegan y salen de cualquier nodo en $Th(E)$ no supera 2 en cada caso; (iv) al estado inicial de $Th(E)$ no llegan transiciones, y del final no salen transiciones.

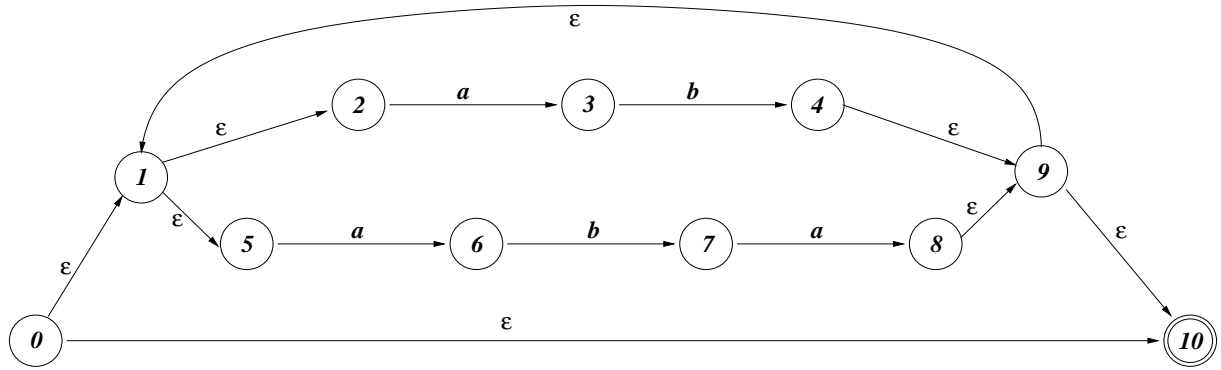
Por simplicidad nos hemos conformado con definir Th usando dibujos esquemáticos. Realmente Th debe definirse formalmente, lo cual el lector puede hacer como ejercicio. Por ejemplo, si $Th(E_1) = (K_1, \Sigma, \Delta_1, s_1, \{f_1\})$ y $Th(E_2) = (K_2, \Sigma, \Delta_2, s_2, \{f_2\})$, entonces $Th(E_1 \mid E_2) = (K_1 \cup K_2 \cup \{s, f\}, \Sigma, \Delta_1 \cup \Delta_2 \cup \{(s, \varepsilon, s_1), (s, \varepsilon, s_2), (f_1, \varepsilon, f), (f_2, \varepsilon, f)\}, s, \{f\})$.

El siguiente teorema indica que Th convierte correctamente ERs en AFNDs, de modo que el AFND reconoce las mismas cadenas que la ER genera.

Teorema 2.1 Sea E una ER, entonces $\mathcal{L}(Th(E)) = \mathcal{L}(E)$.

Prueba: Es fácil verificarlo por inspección y aplicando inducción estructural. La única parte que puede causar problemas es la clausura de Kleene, donde otros esquemas alternativos que podrían sugerirse (por ejemplo $M = (K_1, \Sigma, \Delta_1 \cup \{(f_1, \varepsilon, s_1), (s_1, \varepsilon, f_1)\}, s_1, \{f_1\})$) tienen el problema de permitir terminar un recorrido de $Th(E_1)$ antes de tiempo. Por ejemplo el ejemplo que acabamos de dar, aplicado sobre $E_1 = a \star b$, reconocería la cadena $x = aa$. \square

Ejemplo 2.16 Si aplicamos el método de Thompson para convertir la ER del Ej. 2.3 a AFND, el resultado es distinto de las tres variantes dadas en el Ej. 2.13.



2.5 Conversión de AFND a AFD

[LP81, sec 2.3]

Si bien los AFNDs tienen en principio más flexibilidad que los AFDs, es posible construir siempre un AFD equivalente a un AFND dado. La razón fundamental, y la idea de la conversión, es que el *conjunto* de estados del AFND que *pueden* estar activos después de haber leído una cadena x es una función únicamente de x . Por ello, puede diseñarse un AFD basado en los conjuntos de estados del AFND.

Lo primero que necesitamos es describir, a partir de un estado q del AFND, a qué estados q' podemos llegar sin consumir caracteres de la entrada.

Definición 2.13 Dado un AFND $M = (K, \Sigma, \Delta, s, F)$, la clausura- ε de un estado $q \in K$ se define como

$$E(q) = \{q' \in K, (q, \varepsilon) \vdash_M^* (q', \varepsilon)\}.$$

Ya estamos en condiciones de definir la conversión de un AFND a un AFD. Para ello supondremos que las transiciones del AFND están rotuladas o bien por ε o bien por una sóla letra. Es muy fácil adaptar cualquier AFND para que cumpla esto.

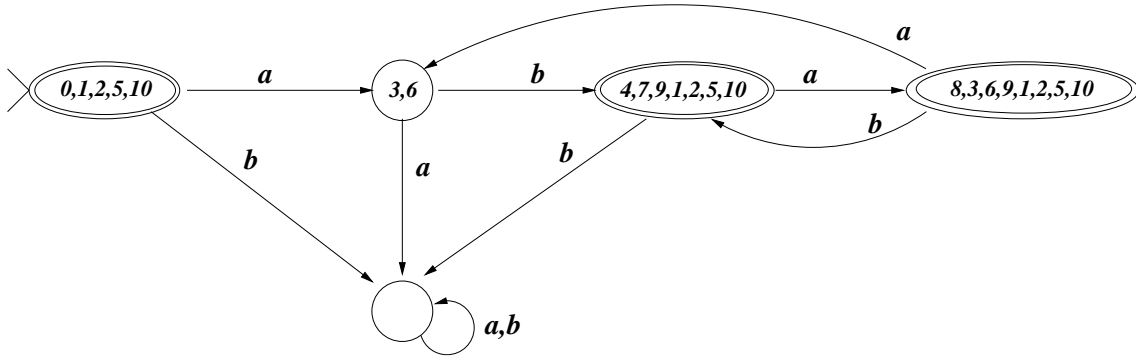
Definición 2.14 Dado un AFND $M = (K, \Sigma, \Delta, s, F)$ que cumple $(q, x, q') \in \Delta \Rightarrow |x| \leq 1$, se define un AFD $det(M) = (K', \Sigma, \delta, s', F')$ de la siguiente manera:

1. $K' = \wp(K)$. Es decir los subconjuntos de K , o conjuntos de estados de M .
2. $s' = E(s)$. Es decir la clausura- ε del estado inicial de M .
3. $F' = K' - \wp(K - F)$. Es decir todos los conjuntos de estados de M que contengan algún estado de F .
4. Para todo $Q \in K'$ (o sea $Q \subseteq K$) y $c \in \Sigma$,

$$\delta(Q, c) = \bigcup_{q \in Q, (q, c, q') \in \Delta} E(q').$$

Esta última ecuación es la que preserva la semántica que buscamos para el AFD.

Ejemplo 2.17 Si calculamos *det* sobre el AFND del Ej. 2.16 obtenemos el siguiente resultado. Observar que se trata del mismo AFD que presentamos en el Ej. 2.12. Lo que era un desafío hacer directamente, ahora lo podemos hacer mecánicamente mediante convertir $ER \rightarrow AFND \rightarrow AFD$.



En el Ej. 2.17 sólo hemos graficado algunos de los estados de K' , más precisamente aquellos alcanzables desde el estado inicial. Los demás son irrelevantes. La forma de determinar un AFND en la práctica es calcular $s' = E(s)$, luego calcular $\delta(s', c)$ para cada $c \in \Sigma$, y recursivamente calcular las transiciones que salen de estos nuevos estados, hasta que todos los estados nuevos producidos sean ya conocidos. De este modo se calculan solamente los estados necesarios de K' .

Observación 2.5 No hay garantía de que el método visto genere el menor AFD que reconoce el mismo lenguaje que el AFND. Existen, sin embargo, técnicas para minimizar AFDs, que no veremos aquí.

El siguiente teorema establece la equivalencia entre un AFND y el AFD que se obtiene con la técnica expuesta.

Teorema 2.2 *Sea M un AFND, entonces $\mathcal{L}(\det(M)) = \mathcal{L}(M)$.*

Prueba: Demostraremos que toda cadena reconocida por el AFD $M' = \det(M)$ también es reconocida por el AFND M , y viceversa. En cada caso, se procede por inducción sobre la longitud de la cadena. Lo que se demuestra es algo un poco más fuerte, para que la inducción funcione: (i) si x lleva de s a q en el AFND, entonces lleva de $s' = E(s)$ a algún Q tal que $q \in Q$ en el AFD; (ii) si x lleva de $E(s)$ a Q en el AFD, entonces lleva de s a cualquier $q \in Q$ en el AFND. De esto se deduce inmediatamente que $x \in \mathcal{L}(M) \Leftrightarrow x \in \mathcal{L}(M')$.

Primero demostramos (i) y (ii) para el caso base $x = \varepsilon$. Es fácil ver que $(\varepsilon, s) \vdash_M^* (\varepsilon, q)$ sii $q \in E(s)$. Por otro lado $(\varepsilon, E(s)) \vdash_{M'}^* (\varepsilon, Q)$ sii $Q = E(s)$ pues M' es determinístico. Se deducen (i) y (ii) inmediatamente.

Veamos ahora el caso inductivo $x = ya$, $a \in \Sigma$, para (i). Si $(s, ya) \vdash_M^* (q, \varepsilon)$, como M consume las letras de a una, existe un camino de la forma $(s, ya) \vdash_M^* (q', a) \vdash_M (q'', \varepsilon) \vdash_M^* (q, \varepsilon)$. Notar que esto implica que $(q', a, q'') \in \Delta$ y $q \in E(q'')$. Por hipótesis inductiva, además, tenemos $(E(s), ya) \vdash_{M'}^* (Q', a)$ para algún Q' que contiene q' . Ahora bien, $(Q', a) \vdash_{M'} (Q, \varepsilon)$, donde $Q = \delta(Q', a)$ incluye, por la Def. 2.14, a $E(q'')$, pues $q' \in Q'$ y $(q', a, q'') \in \Delta$. Finalmente, como $q \in E(q'')$, tenemos $q \in Q$ y terminamos.

Veamos ahora el caso inductivo $x = ya$, $a \in \Sigma$, para (ii). Si $(E(s), ya) \vdash_{M'}^* (Q, \varepsilon)$ debemos tener un camino de la forma $(E(s), ya) \vdash_{M'}^* (Q', a) \vdash_{M'} (Q, \varepsilon)$, donde $Q = \delta(Q', a)$. Por hipótesis inductiva, esto implica $(s, ya) \vdash_M^* (q', a)$ para todo $q' \in Q'$. Asimismo, $(q', a) \vdash_M (q'', \varepsilon) \vdash_M^* (q, \varepsilon)$, para todo $(q', a, q'') \in \Delta$, y $q \in E(q'')$. De la Def. 2.14 se deduce que cualquier $q \in Q$ pertenece a algún $E(q'')$ donde $(q', a, q'') \in \Delta$ y $q' \in Q'$. Hemos visto que M' puede llevar a cualquiera de esos estados. \square

La siguiente observación indica cómo buscar las ocurrencias de una ER en un texto.

Observación 2.6 *Supongamos que queremos buscar las ocurrencias en un texto T de una ER E . Si calculamos $\det(\text{Th}(\Sigma \star \cdot E))$, obtenemos un AFD que reconoce cadenas terminadas en E . Si alimentamos este AFD con el texto T , llegará al estado final en todas las posiciones de T que terminan una ocurrencia de una cadena de E . El algoritmo resultante es muy eficiente en términos del largo de T , si bien la conversión de AFND a AFD puede tomar tiempo exponencial en el largo de E .*

2.6 Conversión de AFD a ER

[LP81, sec 2.5]

Finalmente, cerraremos el triángulo mostrando que los AFDs se pueden convertir a ERs que generen el mismo lenguaje. Esta conversión tiene poco interés práctico, pero es esencial para mostrar que los tres mecanismos de especificar lenguajes son equivalentes.

La idea es numerar los estados de K de cero en adelante, y definir ERs de la forma $R(i, j, k)$, que denotarán las cadenas que llevan al AFD del estado i al estado j utilizando en el camino solamente estados numerados $< k$. Notar que los caminos pueden ser arbitrariamente largos, pues la limitación está dada por los estados intermedios que se pueden usar. Asimismo la limitación no vale (obviamente) para los extremos i y j .

Definición 2.15 Dado un AFD $M = (K, \Sigma, \delta, s, F)$ con $K = \{0, 1, \dots, n-1\}$ definimos expresiones regulares $R(i, j, k)$ para todo $0 \leq i, j < n$, $0 \leq k \leq n$, inductivamente sobre k como sigue.

1. $R(i, j, 0) = \begin{cases} \Phi & \text{si } \{c_1, c_2, \dots, c_l\} = \{c \in \Sigma, \delta(i, c) = j\} \text{ e } i \neq j \\ \varepsilon & \text{si } \{c_1, c_2, \dots, c_l\} = \{c \in \Sigma, \delta(i, c) = j\} \text{ e } i = j \end{cases}$
2. $R(i, j, k+1) = R(i, j, k) \mid R(i, j, k) \cdot R(k, k, k) \star \cdot R(k, j, k).$

Notar que el Φ se usa para el caso en que $l = 0$.

En el siguiente lema establecemos que la definición de las R hace lo que esperamos de ellas.

Lema 2.1 $R(i, j, k)$ es el conjunto de cadenas que reconoce M al pasar del estado i al estado j usando como nodos intermedios solamente nodos numerados $< k$.

Prueba: Para el caso base, la única forma de ir de i a j es mediante transiciones directas entre los nodos, pues no está permitido usar ningún nodo intermedio. Por lo tanto solamente podemos reconocer cadenas de un carácter. Si $i = j$ entonces también la cadena vacía nos lleva de i a i . Para el caso inductivo, tenemos que ir de i a j pasando por nodos numerados hasta k . Una posibilidad es sólo usar nodos $< k$ en el camino, y las cadenas resultantes son $R(i, j, k)$. La otra es usar el nodo k una ó más veces. Entre dos pasadas consecutivas por el nodo k , no se pasa por el nodo k . De modo que partimos el camino entre: lo que se reconoce antes de llegar a k por primera vez ($R(i, k, k)$), lo que se reconoce al ir (dando cero ó más vueltas) de k a k ($R(k, k, k) \star$), y lo que se reconoce al partir de k por última vez y llegar a j ($R(k, j, k)$). \square

Del Lema 2.1 es bastante evidente lo apropiado de la siguiente definición. Indica que el lenguaje reconocido por el AFD es la unión de las R desde el estado inicial hasta los distintos estados finales, usando cualquier nodo posible en el camino intermedio.

Definición 2.16 Sea $M = (K, \Sigma, \delta, s, F)$ con $K = \{0, 1, \dots, n-1\}$ un AFD, y $F = \{f_1, f_2, \dots, f_m\}$. Entonces definimos la ER

$$er(M) = R(s, f_1, n) \mid R(s, f_2, n) \mid \dots \mid R(s, f_m, n).$$

De lo anterior se deduce que es posible generar una ER para cualquier AFD, manteniendo el mismo lenguaje.

Teorema 2.3 Sea M un AFD, entonces $\mathcal{L}(er(M)) = \mathcal{L}(M)$.

Prueba: Es evidente a partir del Lema 2.1 y del hecho de que las cadenas que acepta un AFD son aquellas que lo llevan del estado inicial a algún estado final, pasando por cualquier estado intermedio. \square

Ejemplo 2.18 Consideremos el AFD del Ej. 2.7 y generemos $er(M)$.

$$\begin{aligned}
er(M) &= R(0, 1, 2) \\
R(0, 1, 2) &= R(0, 1, 1) \mid R(0, 1, 1) \cdot R(1, 1, 1) \star R(1, 1, 1) \\
R(0, 1, 1) &= R(0, 1, 0) \mid R(0, 0, 0) \cdot R(0, 0, 0) \star R(0, 1, 0) \\
R(1, 1, 1) &= R(1, 1, 0) \mid R(1, 0, 0) \cdot R(0, 0, 0) \star R(0, 1, 0) \\
R(0, 1, 0) &= b \\
R(0, 0, 0) &= a \mid \varepsilon \\
R(1, 1, 0) &= a \mid \varepsilon \\
R(1, 0, 0) &= b \\
R(1, 1, 1) &= a \mid \varepsilon \mid b \cdot (a \mid \varepsilon) \star b \\
&= a \mid \varepsilon \mid ba \star b \\
R(0, 1, 1) &= b \mid (a \mid \varepsilon) \cdot (a \mid \varepsilon) \star b \\
&= a \star b \\
R(0, 1, 2) &= a \star b \mid a \star b \cdot (a \mid \varepsilon \mid ba \star b) \star (a \mid \varepsilon \mid ba \star b) \\
er(M) &= a \star b (a \mid ba \star b) \star
\end{aligned}$$

Notar que nos hemos permitido algunas simplificaciones en las ERs antes de utilizarlas para R 's superiores. El resultado no es el mismo que el que obtuvimos a mano en el Ej. 2.1, y de hecho toma algo de tiempo convencerse de que es correcto.

Como puede verse, no es necesario en la práctica calcular todas las $R(i, j, k)$, sino que basta partir de las que solicita $er(M)$ e ir produciendo recursivamente las que se van necesitando.

Por último, habiendo cerrado el triángulo, podemos establecer el siguiente teorema fundamental de los lenguajes regulares.

Teorema 2.4 *Todo lenguaje regular puede ser especificado con una ER, o bien con un AFND, o bien con un AFD.*

Prueba: Inmediato a partir de los Teos. 2.1, 2.2 y . □

De ahora en adelante, cada vez que se hable de un lenguaje regular, se puede suponer que se lo tiene descrito con una ER, AFND o AFD, según resulte más conveniente.

Ejemplo 2.19 El desafío del Ej. 2.4 ahora es viable en forma mecánica, aplicando er al AFD del Ej. 2.11. Toma trabajo pero puede hacerse automáticamente.

2.7 Propiedades de Clausura

[LP81, sec 2.4 y 2.6]

Las propiedades de clausura se refieren a qué operaciones podemos hacer sobre lenguajes regulares de modo que el resultado siga siendo un lenguaje regular. Primero demostraremos algunas propiedades sencillas de clausura.

Lema 2.2 *La unión, concatenación y clausura de lenguajes regulares es regular.*

Prueba: Basta considerar ERs E_1 y E_2 , de modo que los lenguajes $L_1 = \mathcal{L}(E_1)$ y $L_2 = \mathcal{L}(E_2)$. Entonces $L_1 \cup L_2 = \mathcal{L}(E_1 \mid E_2)$, $L_1 \circ L_2 = \mathcal{L}(E_1 \cdot E_2)$ y $L_1^* = \mathcal{L}(E_1^*)$ son regulares. \square

Una pregunta un poco menos evidente se refiere a la complementación e intersección de lenguajes regulares.

Lema 2.3 *El complemento de un lenguaje regular es regular, y la intersección y diferencia de dos lenguajes regulares es regular.*

Prueba: Para el complemento basta considerar el AFD $M = (K, \Sigma, \delta, s, F)$ que reconoce L , y ver que $M' = (K, \Sigma, \delta, s, K - F)$ reconoce $L^c = \Sigma^* - L$. La intersección es inmediata a partir de la unión y el complemento, $L_1 \cap L_2 = (L_1^c \cup L_2^c)^c$. La diferencia es $L_1 - L_2 = L_1 \cap L_2^c$. \square

Observación 2.7 *Es posible obtener la intersección en forma más directa, considerando un AFD con estados $K = K_1 \times K_2$. Es un ejercicio interesante imaginar cómo opera este AFD y definirlo formalmente.*

Ejemplo 2.20 Es un desafío obtener directamente la ER de la diferencia de dos ERs. Ahora tenemos que esto puede hacerse mecánicamente. Es un ejercicio interesante, para apreciar la sofisticación obtenida, indicar paso a paso cómo se haría para obtener la ER de $L_1 - L_2$ a partir de las ERs de L_1 y L_2 .

Ejemplo 2.21 Las operaciones sobre lenguajes regulares permiten demostrar que ciertos lenguajes son regulares con más herramientas que las provistas por ERs o autómatas. Por ejemplo, se puede demostrar que los números decimales correctamente escritos (sin ceros delante) que son múltiplos de 7 pero no múltiplos de 11, y que además tienen una cantidad impar de dígitos '4', forman un lenguaje regular. Llamando $D = 0 \mid 1 \mid \dots \mid 9$, M_7 al AFD del Ej. 2.11, M_{11} a uno similar para los múltiplos de 11, y E_4 a una ER similar a la del Ej. 2.1 pero que cuenta 4's, el lenguaje que queremos es $((\mathcal{L}(M_7) - \mathcal{L}(M_{11})) \cap \mathcal{L}(E_4)) - \mathcal{L}(0 \cdot D^*)$. ¿Se atreve a dar una ER o AFND para el resultado? (no es en serio, puede llevarle mucho tiempo).

2.8 Lema de Bombeo

[LP81, sec 2.6]

Hasta ahora hemos visto diversas formas de mostrar que un lenguaje es regular, pero ninguna (aparte de que no nos funcione nada de lo que sabemos hacer) para mostrar que no lo es. Veremos ahora una herramienta para demostrar que un cierto L no es regular.

Observación 2.8 *Pregunta capciosa: Esta herramienta que veremos, ¿funciona para todos los lenguajes no regulares? ¡Imposible, pues hay más lenguajes no regulares que demostraciones!*

La idea esencial es que un lenguaje regular debe tener cierta repetitividad, producto de la capacidad limitada del AFD que lo reconoce. Más precisamente, todo lo que el AFD recuerda sobre la cadena ya leída se condensa en su estado actual, para el cual hay sólo una cantidad finita de posibilidades. El siguiente teorema (que por alguna razón se llama Lema de Bombeo) explota precisamente este hecho, aunque a primera vista no se note, ni tampoco se vea cómo usarlo para probar que un lenguaje no es regular.

Teorema 2.5 (Lema de Bombeo)

Sea L un lenguaje regular. Entonces existe un número $N > 0$ tal que toda cadena $w \in L$ de largo $|w| > N$ se puede escribir como $w = xyz$ de modo que $y \neq \varepsilon$, $|xy| \leq N$, $y \neq \varepsilon$, $\forall n \geq 0, xy^n z \in L$.

Prueba: Sea $M = (K, \Sigma, \delta, s, F)$ un AFD que reconoce L . Definiremos $N = |K|$. Al leer w , M pasa por distintas configuraciones hasta llegar a un estado final. Consideremos los primeros N caracteres de w en este camino, llamándole q_i al estado al que se llega luego de consumir $w_1 w_2 \dots w_i$:

$$(q_0, w_1 w_2 \dots) \vdash (q_1, w_2 \dots) \vdash \dots \vdash (q_i, w_{i+1} \dots) \vdash \dots \vdash (q_j, w_{j+1} \dots) \vdash \dots \vdash (q_N, w_{N+1} \dots) \vdash \dots$$

Los estados q_0, q_1, \dots, q_N no pueden ser todos distintos, pues M tiene sólo N estados. De modo que en algún punto del camino se repite algún estado. Digamos $q_i = q_j$, $i < j$. Eso significa que, si eliminamos $y = w_{i+1} w_{i+2} \dots w_j$ de w , M llegará exactamente al mismo estado final al que llegaba antes:

$$(q_0, w_1 w_2 \dots) \vdash (q_1, w_2 \dots) \vdash \dots \vdash (q_{i-1}, w_i \dots) \vdash (q_i = q_j, w_{j+1} \dots) \vdash \dots \vdash (q_N, w_{N+1} \dots) \vdash \dots$$

y, similarmente, podemos duplicar y en w tantas veces como queramos y el resultado será el mismo. Llamando $x = w_1 \dots w_i$, $y = w_{i+1} \dots w_j$, y $z = w_{j+1} \dots w_{|w|}$, tenemos entonces el teorema. Es fácil verificar que todas las condiciones valen. \square

¿Cómo utilizar el Lema de Bombeo para demostrar que un lenguaje no es regular? La idea es negar las condiciones del Teo. 2.5.

1. Para *cualquier* longitud N ,
2. debemos ser capaces de elegir *alguna* $w \in L$, $|w| > N$,
3. de modo que para *cualquier* forma de partir $w = xyz$, $y \neq \varepsilon$, $|xy| \leq N$,
4. podamos encontrar *alguna* $n \geq 0$ tal que $xy^n z \notin L$.

Una buena forma de pensar en este proceso es en que se juega contra un adversario. El elige N , nosotros w , el la particiona en xyz , nosotros elegimos n . Si somos capaces de ganarle haga lo que haga, hemos demostrado que L no es regular.

Ejemplo 2.22 Demostremos que $L = \{a^n b^n, n \geq 0\}$ no es regular. Dado N , elegimos $w = a^N b^N$. Ahora, se elija como se elija y dentro de w , ésta constará de puras a 's, es decir, $x = a^r$, $y = a^s$, $z = a^{N-r-s} b^N$, $r+s \leq N$, $s > 0$. Ahora basta mostrar que $xy^0 z = xz = a^r a^{N-r-s} b^N = a^{N-s} b^N \notin L$ pues $s > 0$.

Un ejemplo que requiere algo más de inspiración es el siguiente.

Ejemplo 2.23 Demostremos que $L = \{a^p, p \text{ es primo}\}$ no es regular. Dado N , elegimos un primo $p > N + 1$, $w = a^p$. Ahora, para toda elección $x = a^r$, $y = a^s$, $z = a^t$, $r + s + t = p$, debemos encontrar algún $n \geq 0$ tal que $a^{r+ns+t} \notin L$, es decir, $r + ns + t$ no es primo. Pero esto siempre es posible, basta con elegir $n = r + t$ para tener $r + ns + t = (r + t)(s + 1)$ compuesto. Ambos factores son mayores que 1 porque $s > 0$ y $r + t = p - s > (N + 1) - N$.

2.9 Propiedades Algorítmicas de Lenguajes Regulares

[LP81, sec 2.4]

Antes de terminar con lenguajes regulares, examinemos algunas propiedades llamadas “algorítmicas”, que tienen relación con qué tipo de preguntas pueden hacerse sobre lenguajes regulares y responderse en forma mecánica. Si bien a esta altura pueden parecer algo esotéricas, estas preguntas adquieren sentido más adelante.

Lema 2.4 *Dados lenguajes regulares L, L_1, L_2 (descritos mediante ERs o autómatas), las siguientes preguntas tienen respuesta algorítmica:*

1. Dada $w \in \Sigma^*$, ¿es $w \in L$?
2. ¿Es $L = \emptyset$?
3. ¿Es $L = \Sigma^*$?
4. ¿Es $L_1 \subseteq L_2$?
5. ¿Es $L_1 = L_2$?

Prueba: Para (1) tomamos el AFD que reconoce L y lo alimentamos con w , viendo si llega a un estado final o no. ¡Para eso son los AFDs!. Para (2), vemos si en el AFD existe un camino del estado inicial a un estado final. *Esto se resuelve fácilmente con algoritmos de grafos.* Para (3), complementamos el AFD de L y reducimos la pregunta a (2). Para (4), calculamos $L = L_1 - L_2$ y reducimos la pregunta a (2) con respecto a L . Para (5), reducimos la pregunta a (4), $L_1 \subseteq L_2$ y $L_2 \subseteq L_1$. \square

2.10 Ejercicios

Expresiones Regulares

1. ¿Qué lenguaje representa la expresión $((a \star a) b) \mid b$?
2. Reescriba las siguientes expresiones regulares de una forma más simple

- (a) $\Phi \star \mid a \star \mid b \star \mid (a \mid b) \star$
 - (b) $((a \star b \star) \star (b \star a \star) \star) \star$
 - (c) $(a \star b) \star \mid (b \star a) \star$
 - (d) $(a \mid b) \star a (a \mid b) \star$
3. Sea $\Sigma = \{a, b\}$. Escriba expresiones regulares para los siguientes conjuntos
- (a) Las cadenas en Σ^* con no más de 3 a 's.
 - (b) Las cadenas en Σ^* con una cantidad de a 's divisible por 3.
 - (c) Las cadenas en Σ^* con exactamente una ocurrencia de la subcadena aaa .
4. Pruebe que si L es regular, también lo es $L' = \{uw, u \in \Sigma^*, w \in L\}$, mediante hallar una expresión regular para L' .
5. ¿Cuáles de las siguientes afirmaciones son verdaderas? Explique. (Abusaremos de la notación escribiendo c^* para $\{c\}^*$).
- (a) $baa \in a^*b^*a^*b^*$
 - (b) $b^*a^* \cap a^*b^* = a^* \cup b^*$
 - (c) $a^*b^* \cap c^*d^* = \emptyset$
 - (d) $abcd \in (a(cd)^*b)^*$

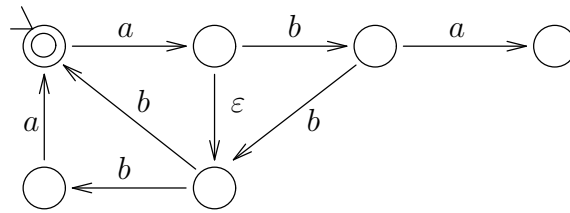
AFDs, AFNDs y Conversiones

1. Dibuje los siguientes AFDs y describa informalmente el lenguaje que aceptan. Hemos escrito la función δ como un conjunto.
- (a) $K = \{q_0, q_1, q_2, q_3\}$, $\Sigma = \{a, b\}$, $s = q_0$, $F = \{q_1\}$,
 $\delta = \{(q_0, a, q_1), (q_0, b, q_2), (q_1, a, q_3), (q_1, b, q_0), (q_2, a, q_2), (q_2, b, q_2), (q_3, a, q_2), (q_3, b, q_2)\}$.
 - (b) $K = \{q_0, q_1, q_2, q_3, q_4\}$, $\Sigma = \{a, b\}$, $s = q_0$, $F = \{q_2, q_3\}$,
 $\delta = \{(q_0, a, q_1), (q_0, b, q_3), (q_1, a, q_1), (q_1, b, q_2), (q_2, a, q_4), (q_2, b, q_4), (q_3, a, q_4), (q_3, b, q_4), (q_4, a, q_4), (q_4, b, q_4)\}$.
 - (c) $K = \{q_0, q_1, q_2, q_3\}$, $\Sigma = \{a, b\}$, $s = q_0$, $F = \{q_0\}$,
 $\delta = \{(q_0, a, q_1), (q_0, b, q_3), (q_1, a, q_2), (q_1, b, q_0), (q_2, a, q_3), (q_2, b, q_1), (q_3, a, q_3), (q_3, b, q_3)\}$.
 - (d) Idem al anterior pero $s = q_1$, $F = \{q_1\}$.
2. Construya AFDs que acepten cada uno de los siguientes lenguajes. Escribálos formalmente y dibújelos.
- (a) $\{w \in \{a, b\}^*, \text{ cada } a \text{ en } w \text{ está precedido y seguido por una } b\}$

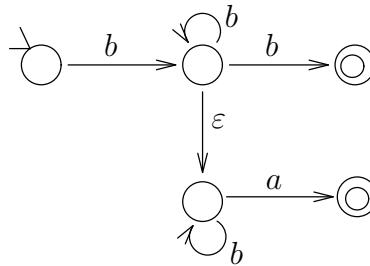
- (b) $\{w \in \{a, b\}^*, w \text{ tiene } abab \text{ como subcadena}\}$
- (c) $\{w \in \{a, b\}^*, w \text{ no tiene } aa \text{ ni } bb \text{ como subcadena}\}$
- (d) $\{w \in \{a, b\}^*, w \text{ tiene una cantidad impar de } a\text{'s y una cantidad par de } b\text{'s}\}$.
- (e) $\{w \in \{a, b\}^*, w \text{ tiene } ab \text{ y } ba \text{ como subcadenas}\}$.

3. ¿Cuáles de las siguientes cadenas son aceptadas por los siguientes autómatas?

- (a) $aa, aba, abb, ab, abab$.



- (b) ba, ab, bb, b, bba .



4. Dibuje AFNDs que acepten los siguientes lenguajes. Luego conviértalos a AFDs.

- (a) $(ab)^*(ba)^* \cup aa^*$
- (b) $((ab \cup aab)^*a^*)^*$
- (c) $((a^*b^*a^*)^*b)^*$
- (d) $(ba \cup b)^* \cup (bb \cup a)^*$

5. Escriba expresiones regulares para los lenguajes aceptados por los siguientes AFNDs.

- (a) $K = \{q_0, q_1\}, \Sigma = \{a, b\}, s = q_0, F = \{q_0\},$
 $\Delta = \{(q_0, ab, q_0), (q_0, a, q_1), (q_1, bb, q_1)\}$
- (b) $K = \{q_0, q_1, q_2, q_3\}, \Sigma = \{a, b\}, s = q_0, F = \{q_0, q_2\},$
 $\Delta = \{(q_0, a, q_1), (q_1, b, q_2), (q_2, a, q_1), (q_1, b, q_3), (q_3, a, q_2)\}$
- (c) $K = \{q_0, q_1, q_2, q_3, q_4, q_5\}, \Sigma = \{a, b\}, s = q_0, F = \{q_1, q_5\},$
 $\Delta = \{(q_0, \varepsilon, q_1), (q_0, a, q_4), (q_1, a, q_2), (q_2, a, q_3), (q_3, a, q_1), (q_4, a, q_5), (q_5, a, q_4)\}$

6. (a) Encuentre un AFND simple para $(aa \mid aab \mid aba)^*$.
- (b) Conviértalo en un autómata determinístico usando el algoritmo visto.
- (c) Trate de entender el funcionamiento del autómata. ¿Puede hallar uno con menos estados que reconozca el mismo lenguaje?
- (d) Repita los mismos pasos para $(a \mid b)^* aabab$.

Propiedades de Clausura y Algorítmicas

1. Pruebe que si L es regular, entonces los siguientes conjuntos también lo son
 - (a) $Pref(L) = \{x, \exists y, xy \in L\}$
 - (b) $Suf(L) = \{y, \exists x, xy \in L\}$
 - (c) $Subs(L) = \{y, \exists x, z, xyz \in L\}$
 - (d) $Max(L) = \{w \in L, x \neq \varepsilon \Rightarrow wx \notin L\}$
 - (e) $L^R = \{w^R, w \in L\}$ (w^R es w leído al revés).
2. Muestre que hay algoritmos para responder las siguientes preguntas, donde L_1 y L_2 son lenguajes regulares
 - (a) No hay una sola cadena w en común entre L_1 y L_2 .
 - (b) L_1 y L_2 son uno el complemento del otro
 - (c) $L_1^* = L_2$
 - (d) $L_1 = Pref(L_2)$

Lenguajes Regulares y No Regulares

1. Demuestre que cada uno de los siguientes conjuntos es o no es regular.
 - (a) $\{a^{10^n}, n \in \mathbb{Z}_{\geq 0}\}$
 - (b) $\{w \in \{0..9\}^*, w \text{ representa } 10^n \text{ para algún } n \geq 0\}$
 - (c) $\{w \in \{0..9\}^*, w \text{ es una secuencia de dígitos que aparece en la expansión decimal de } 1/7 = 0.142857\ 142857\ 142857...\}$
2. Demuestre que el conjunto $\{a^n b a^m b a^{n+m}, n, m \geq 0\}$ no es regular. Visto operacionalmente, esto implica que los autómatas finitos no saben “sumar”.
3. Pruebe que los siguientes conjuntos no son regulares.
 - (a) $\{ww^R, w \in \{a, b\}^*\}$

- (b) $\{ww, w \in \{a, b\}^*\}$
 - (c) $\{w\bar{w}, w \in \{a, b\}^*\}$. \bar{w} es w donde cada a se cambia por una b y viceversa.
4. ¿Cierto o falso? Demuestre o dé contraejemplos.
- (a) Todo subconjunto de un lenguaje regular es regular
 - (b) Todo lenguaje regular tiene un subconjunto propio regular
 - (c) Si L es regular también lo es $\{xy, x \in L, y \notin L\}$
 - (d) Si L es regular, también lo es $\{w \in L, \text{ ningún prefijo propio de } w \text{ pertenece a } L\}$.
 - (e) $\{w, w = w^R\}$ es regular
 - (f) Si L es regular, también lo es $\{w, w \in L, w^R \in L\}$
 - (g) Si $\{L_1, L_2, \dots\}$ es un conjunto *infinito* de lenguajes regulares, también lo es $\bigcup L_i$, o sea la unión de todos ellos. ¿Y si el conjunto es finito?
 - (h) $\{xyx^R, x, y \in \Sigma^*\}$ es regular.

2.11 Preguntas de Controles

A continuación se muestran algunos ejercicios de controles de años pasados, para dar una idea de lo que se puede esperar en los próximos. Hemos omitido (i) (casi) repeticiones, (ii) cosas que ahora no se ven, (iii) cosas que ahora se dan como parte de la materia y/o están en los ejercicios anteriores. Por lo mismo a veces los ejercicios se han alterado un poco o se presenta sólo parte de ellos, o se mezclan versiones de ejercicios de distintos años para que no sea repetitivo.

C1 1996, 1997 Responda verdadero o falso y justifique brevemente (máximo 5 líneas). Una respuesta sin justificación no vale *nada* aunque esté correcta, una respuesta incorrecta puede tener algún valor por la justificación.

- a) Si un autómata “finito” pudiera tener infinitos estados, podría reconocer *cualquier* lenguaje.
- b) No hay algoritmo para saber si un autómata finito reconoce un lenguaje finito o infinito.
- c) La unión o intersección de dos lenguajes no regulares no puede ser regular.
- d) Si la aplicación del Lema del Bombeo para lenguajes regulares falla, entonces el lenguaje es regular.
- e) Dados dos lenguajes regulares L_1 y L_2 , existe algoritmo para determinar si el conjunto de prefijos de L_1 es igual al conjunto de sufijos de L_2 .

Hemos unido ejercicios similares de esos años.

C1 1996 Suponga que tiene que buscar un patrón p en un texto, ejemplo "lolol". Queremos construir un autómata finito que acepte un texto si y sólo si éste contiene el patrón p .

- a) Escriba la expresión regular que corresponde a los textos que desea aceptar.
- b) Dibuje un autómata finito equivalente a la expresión regular.
- c) Para el ejemplo de $p = \text{"lolol"}$, convierta el autómata a determinístico. Observe que no vale la pena generar más de un estado final, son todos equivalentes.

ER 1996 Un *autómata de múltiple entrada* es igual a los normales, excepto porque puede tener varios estados iniciales. El autómata acepta x si comenzando de *algún* estado inicial se acepta x .

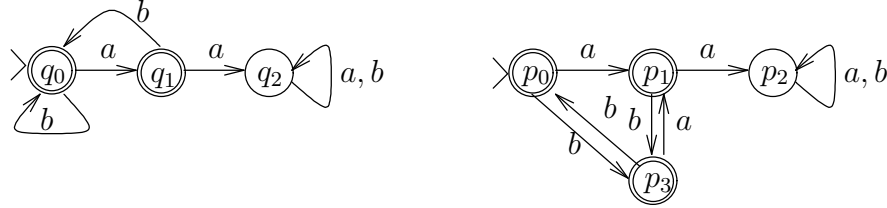
- a) Use un autómata de dos entradas para reconocer el lenguaje de todas las cadenas de ceros y unos sin dos símbolos iguales consecutivos.
- b) Describa formalmente un autómata de múltiple entrada, en su versión determinística y no determinística. Describa formalmente el conjunto de cadenas aceptadas por tales autómatas.
- c) ¿Los autómatas de múltiple entrada son más potentes que los normales o no? Demuéstrelo.

C1 1997 Dado el lenguaje de las cadenas binarias donde nunca aparece un cero aislado:

- Dé una expresión regular que lo genere.
- Conviértala a un autómata no determinístico con el método visto. Simplifique el autómata.
- Convierta este autómata simplificado a determinístico con el método visto. Simplifique el autómata obtenido.

C1 1998, 1999

- a) Utilice los métodos vistos para determinar si los siguientes autómatas son o no equivalentes.
- a) Expresé el lenguaje aceptado por el autómata de la izquierda como una expresión regular.
- b) Convierta el autómata de la derecha en un autómata finito determinístico.



Se unieron distintas preguntas sobre los mismos autómatas en esos años.

C1 1998 Dada la expresión regular $a(a \mid ba)^* b^*$:

- Indique todas las palabras de largo 4 que pertenecen al lenguaje representado por esta expresión regular.
- Construya un autómata finito no determinístico que reconozca este lenguaje.

C1 1998 Es fácil determinar si una palabra pertenece o no a un lenguaje usando un autómata finito determinístico. Sin embargo, al pasar de un AFND a un AFD el número de estados puede aumentar exponencialmente, lo que aumenta el espacio necesario. Una solución alternativa es simular un autómata finito no determinístico.

Escriba en pseudo-lenguaje una función $Acepta(M, w)$ que dado un AFND $M = (K, \Sigma, \Delta, s, F)$ y una palabra w , retorne V o F , si pertenece o no al lenguaje que acepta M . Puede usar la notación $M.K$, $M.s$, etc., para obtener cada elemento del autómata y suponer que todas las operaciones básicas que necesite ya existen (por ejemplo, operaciones de conjuntos).

Ex 1999, C1 2002 Demuestre que los siguientes lenguajes *no* son regulares.

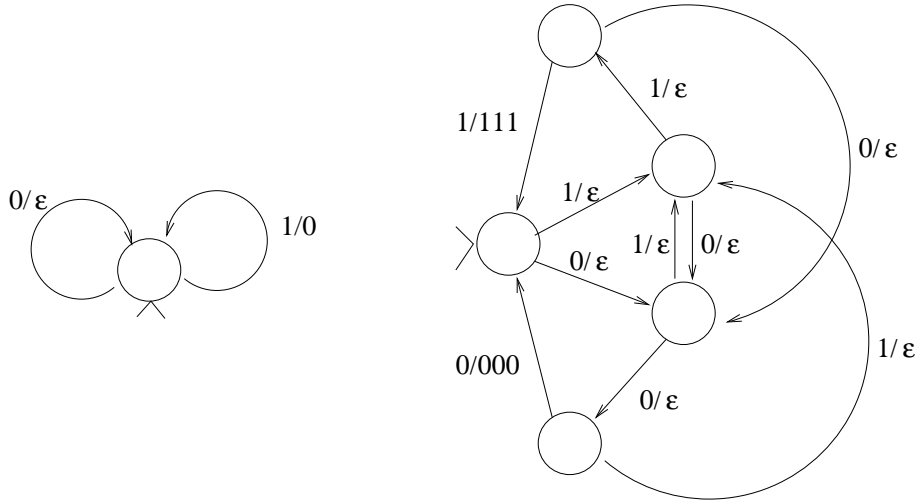
- $\{a^n b^m, n > m\}$.
- $\{a^n b^m a^r, r \geq n\}$.

C1 2000 Un *transductor* es una tupla $M = (K, \Sigma, \delta, s)$, donde K es un conjunto finito de estados, Σ es un alfabeto finito, $s \in K$ es el estado inicial y

$$\delta : K \times \Sigma \longrightarrow K \times \Sigma^*$$

La idea es que un transductor lee una cadena de entrada y *produce* una cadena de salida. Si estamos en el estado q y leemos el carácter a , y $\delta(q, a) = (q', x)$ entonces el transductor pasa al estado q' y produce la cadena x . En la representación gráfica se pone a/x sobre la flecha que va de q a q' .

Por ejemplo, el transductor izquierdo de la figura elimina todos los ceros de la entrada y a los unos restantes los convierte en ceros. El de la derecha considera las secuencias seguidas de ceros o unos y las trunca para que sus longitudes sean múltiplos de 3 (ej. $00001111000001111 \rightarrow 0001111000111$).



- (a) Dibuje un transductor que tome las secuencias seguidas de ceros o unos de la entrada y sólo deje un representante de cada secuencia, por ejemplo $001110011001000111 \rightarrow 01010101$.
- (b) Defina formalmente la función salida (S) de un transductor, que recibe la cadena de entrada y entrega la salida que producirá el transductor. Ayuda: defina $S(w) = T(s, w)$, donde $T(q, w)$ depende del estado actual del transductor, y luego considere los casos $w = \varepsilon$ y $w = a \cdot w'$, es decir la letra a concatenada con el resto de la cadena, w' .
- (c) El *lenguaje de salida* de un transductor es el conjunto de cadenas que puede producir (es decir $\{S(w), w \in \Sigma^*\}$). Demuestre que el lenguaje de salida de un transductor es regular. Ayuda: dado un transductor, muestre cómo obtener el AFND que reconozca lo que el transductor podría generar.
- C1 2001** Demuestre que si L es un lenguaje regular entonces el lenguaje de los prefijos reversos de cadenas de L también es regular. Formalmente, demuestre que $L' = \{x^R, \exists y, xy \in L\}$ es regular.
- C1 2001** Intente usar el Lema de Bombeo *sin la restricción* $|xy| \leq N$ para probar que $L = \{w \in \{a, b\}^*, w = w^R\}$ no es regular. ¿Por qué falla? Hágalo ahora con el Lema sin la restricción.
- C1 2002** Dada la expresión regular $(AA|AT)((AG|AAA)^*)$, realice lo siguiente usando los métodos vistos:
- (a) Construya el autómata finito no determinístico que la reconoce.
- (b) Convierta el autómata obtenido a determinístico.

C1 2004 Considere la expresión regular $(AB|CD)^*AFF^*$.

- (a) Construya el AFND correspondiente.
- (b) Convierta el AFND en AFD. Omita el estado sumidero y las aristas que llevan a él. El resultado tiene 7 estados.
- (c) Minimice el AFD, usando la regla siguiente: si dos estados q y q' son ambos finales o ambos no finales, de ellos salen aristas por las mismas letras, y las aristas que salen de ellos por cada letra llevan a los mismos estados, entonces q y q' se pueden unir en un mismo estado. Reduzca el AFD a 5 estados usando esta regla.
- (d) Convierta el AFD nuevamente en expresión regular.

C1 2004 Demuestre que:

- 1. Si L es regular, $nosubstr(L)$ es regular ($nosubstr(L)$ es el conjunto de cadenas que no son substrings de alguna cadena en L).
- 2. Si L es regular, $Ext(L)$ es regular ($Ext(L)$ es el conjunto de cadenas con algún prefijo en L , $Ext(L) = \{xy, x \in L\}$).

Ex 2005 Un Autómata Finito de Doble Dirección (AFDD) se comporta similarmente a un Autómata Finito Determinístico (AFD), excepto porque tiene la posibilidad de volver hacia atrás en la lectura de la cadena. Es decir, junto con indicar a qué estado pasa al leer un carácter, indica si se mueve hacia atrás o hacia adelante. El autómata termina su procesamiento cuando se mueve hacia adelante del último carácter. En este momento, acepta la cadena si a la vez pasa a un estado final. Si nunca pasa del último carácter de la cadena, el AFDD no la acepta. Si el AFDD trata de moverse hacia atrás del primer carácter, este comando se ignora y permanece en el primer carácter.

Defina formalmente los AFDDs como una tupla de componentes; luego defina lo que es una configuración; cómo es una transición entre configuraciones; el concepto de aceptar o no una cadena; y finalmente defina el lenguaje aceptado por un AFDD.

2.12 Proyectos

- 1. Investigue sobre minimización de AFDs. Una posible fuente es [ASU86], desde página 135. Otra es [HMU01], sección 4.4, desde página 154.
- 2. Investigue sobre métodos alternativos a Thompson para convertir una ER en AFND. Por ejemplo, el método de Glushkov se describe en [NR02], sección 5.2.2, desde página 105.

3. Investigue una forma alternativa de convertir AFDs en ERs, donde los estados se van eliminando y se van poniendo ERs cada vez más complejas en las aristas del autómata. Se describe por ejemplo en [HMU01], sección 3.2.2, desde página 96.
4. Investigue sobre implementación eficiente de autómatas. Algunas fuentes son [NR02], sección 5.4, desde página 117; y [ASU86], sección 3.9, desde página 134. También puede investigar la implementación de las herramientas `grep` de Gnu y `lex` de Unix.
5. Programe el ciclo completo de conversión $ER \rightarrow AFND \rightarrow AFD \rightarrow ER$.
6. Programe un buscador eficiente de ERs en texto, de modo que reciba una ER y lea la entrada estándar, enviando a la salida estándar las líneas que contienen una ocurrencia de la ER.

Referencias

- [ASU86] A. Aho, R. Sethi, J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
- [HMU01] J. Hopcroft, R. Motwani, J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. 2nd Edition. Pearson Education, 2001.
- [LP81] H. Lewis, C. Papadimitriou. *Elements of the Theory of Computation*. Prentice-Hall, 1981. Existe una segunda edición, bastante parecida, de 1998.
- [NR02] G. Navarro, M. Raffinot. *Flexible Pattern Matching in Strings*. Cambridge University Press, 2002.

Capítulo 3

Lenguajes Libres del Contexto

[LP81, cap 3]

En este capítulo estudiaremos una forma de representación de lenguajes más potentes que los regulares. Los lenguajes libres del contexto (LC) son importantes porque sirven como mecanismo formal para expresar la gramática de lenguajes de programación o los semiestructurados. Por ejemplo la popular “Backus-Naur form” es esencialmente una gramática libre del contexto. Similarmente, los DTDs usados para indicar el formato permitido en documentos XML son esencialmente gramáticas que describen lenguajes LC. Los lenguajes LC también se usan en biología computacional para modelar las propiedades que se buscan en secuencias de ADN o proteínas. El estudio de este tipo de lenguajes deriva en la construcción semiautomática de *parsers* (reconocedores) eficientes, los cuales son esenciales en la construcción de compiladores e intérpretes, así como para procesar textos semiestructurados. Una herramienta conocida para esta construcción semiautomática es *lex/yacc* en C/Unix, y sus distintas versiones para otros ambientes. Estas herramientas reciben esencialmente una especificación de un lenguaje LC y producen un programa que parsea tal lenguaje.

En términos teóricos, los lenguajes LC son interesantes porque van más allá de la memoria finita sobre el pasado permitida a los regulares, pudiendo almacenar una cantidad arbitraria de información sobre el pasado, siempre que esta información se acceda en forma de pila. Es interesante ver los lenguajes que resultan de esta restricción.

3.1 Gramáticas Libres del Contexto (GLCs) [LP81, sec 3.1]

Una *gramática libre del contexto (GLC)* es una serie de *reglas de derivación, producción o reescritura* que indican que un cierto símbolo puede convertirse en (o reescribirse como) una secuencia de otros símbolos, los cuales a su vez pueden convertirse en otros, hasta obtener una cadena del lenguaje. Es una forma particular de *sistema de reescritura*, restringida a que las reglas aplicables para reescribir un símbolo son independientes de lo que tiene alrededor

en la cadena que se está generando (de allí el nombre “libre del contexto”).

Distinguiremos entre los *símbolos terminales* (los del alfabeto Σ que formarán la cadena final) y los *símbolos no terminales* (los que deben reescribirse como otros y no pueden aparecer en la cadena final). Una GLC tiene un *símbolo inicial* del que parten todas las derivaciones, y se dice que *genera* cualquier secuencia de símbolos terminales que se puedan obtener desde el inicial mediante reescrituras.

Ejemplo 3.1 Consideremos las siguientes reglas de reescritura:

$$\begin{aligned} S &\longrightarrow aSb \\ S &\longrightarrow \varepsilon \end{aligned}$$

donde S es el símbolo (no terminal) inicial, y $\{a, b\}$ son los símbolos terminales. Las cadenas que se pueden generar con esta GLC forman precisamente el conjunto $\{a^n b^n, n \geq 0\}$, que en el Ej. 2.22 vimos que no era regular. De modo que este mecanismo permite expresar lenguajes no regulares.

Formalicemos ahora lo que es una GLC y el lenguaje que describe.

Definición 3.1 Una gramática libre del contexto (GLC) es una tupla $G = (V, \Sigma, R, S)$, donde

1. V es un conjunto finito de símbolos no terminales.
2. Σ es un conjunto finito de símbolos terminales, $V \cap \Sigma = \emptyset$.
3. $S \in V$ es el símbolo inicial.
4. $R \subset_F V \times (V \cup \Sigma)^*$ son las reglas de derivación (conjunto finito).

Escribiremos las reglas de R como $A \longrightarrow_G z$ o simplemente $A \longrightarrow z$ en vez de (A, z) .

Ahora definiremos formalmente el lenguaje descrito por una GLC.

Definición 3.2 Dada una GLC $G = (V, \Sigma, R, S)$, la relación lleva en un paso $\Longrightarrow_G \subseteq (V \cup \Sigma)^* \times (V \cup \Sigma)^*$ se define como

$$\forall x, y, \forall A \longrightarrow z \in R, xAy \Longrightarrow_G xzy.$$

Definición 3.3 Definimos la relación lleva en cero o más pasos, \Longrightarrow_G^* , como la clausura reflexiva y transitiva de \Longrightarrow_G .

Escribiremos simplemente \Longrightarrow e \Longrightarrow^* cuando G sea evidente.

Notamos que se puede llevar en cero o más pasos a una secuencia que aún contiene no terminales. Las derivaciones que nos interesan finalmente son las que llevan del símbolo inicial a secuencias de terminales.

Definición 3.4 Dada una GLC $G = (V, \Sigma, R, S)$, definimos el lenguaje generado por G , $\mathcal{L}(G)$, como

$$\mathcal{L}(G) = \{w \in \Sigma^*, S \Rightarrow_G^* w\}.$$

Finalmente definimos los lenguajes libres del contexto como los expresables con una GLC.

Definición 3.5 Un lenguaje L es libre del contexto (LC) si existe una GLC G tal que $L = \mathcal{L}(G)$.

Ejemplo 3.2 ¿Cómo podrían describirse las secuencias de paréntesis bien balanceados? (donde nunca se han cerrado más paréntesis de los que se han abierto, y al final los números coinciden). Una GLC que lo describa es sumamente simple:

$$\begin{aligned} S &\longrightarrow (S)S \\ S &\longrightarrow \varepsilon \end{aligned}$$

la que formalmente se escribe como $V = \{S\}$, $\Sigma = \{(\,,\,)\}$, $R = \{(S, (S)S), (S, \varepsilon)\}$. Una derivación de la cadena $((\,))(\,)$ a partir de S podría ser como sigue:

$$S \Rightarrow (S)S \Rightarrow ((S)S)S \Rightarrow ((S)S) \Rightarrow ((\,))S \Rightarrow ((\,))(S)S \Rightarrow ((\,))(\,))S \Rightarrow ((\,))(\,)),$$

y otra podría ser como sigue:

$$S \Rightarrow (S)S \Rightarrow (S)(S)S \Rightarrow (S)(\,))S \Rightarrow (S)(\,) \Rightarrow ((S)S)(\,) \Rightarrow ((S)S)(\,) \Rightarrow ((\,))(\,)).$$

Esto ilustra un hecho interesante: existen distintas derivaciones para una misma cadena, producto de aplicar las reglas en distinto orden.

Observación 3.1 ¿Puede el lenguaje del Ej. 3.2 ser regular? No, pues entonces su intersección con $(^*)^*$ también lo sería, pero esa intersección es $\{(^n)^n, n \geq 0\}$, que ya sabemos que no es regular.

Una herramienta muy útil para visualizar derivaciones, y que se independiza del orden en que se aplican las reglas, es el *árbol de derivación*.

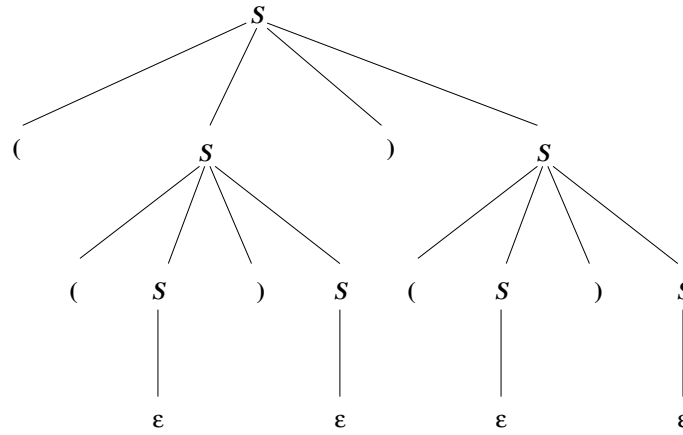
Definición 3.6 Un árbol de derivación para una gramática $G = (V, \Sigma, R, S)$ es un árbol donde los hijos tienen orden y los nodos están rotulados con elementos de V ó Σ ó ε . La raíz está rotulada con S , y los nodos internos deben estar rotulados con elementos de V . Si los rótulos de los hijos de un nodo interno rotulado A son $a_1 \dots a_k$, $k \geq 1$ y $a_i \in V \cup \Sigma$, debe existir una regla $A \longrightarrow a_1 \dots a_k \in R$. Si un nodo interno rotulado A tiene un único hijo rotulado ε , debe haber una regla $A \longrightarrow \varepsilon \in R$. Diremos que el árbol genera la cadena que resulta de concatenar todos los símbolos de sus hojas, de izquierda a derecha, vistos como cadenas de largo 1 (o cero para ε).

Observar que la definición permite que un árbol de derivación tenga símbolos no terminales en sus hojas, es decir, puede representar una derivación parcial. El siguiente lema es inmediato.

Lema 3.1 *Si un árbol de derivación para G genera $x \in (V \cup \Sigma)^*$, entonces $S \Rightarrow_G^* x$. Si $S \Rightarrow_G^* x$, existe un árbol de derivación que genera x .*

Prueba: Muy fácil por inducción estructural sobre el árbol o por inducción sobre la longitud de la derivación, según el caso. \square

Ejemplo 3.3 El árbol de derivación para la cadena del Ej. 3.2 es como sigue:



y abstrae de ambos órdenes de derivación.

Sin embargo, los distintos órdenes de derivación no son los únicos responsables de que existan distintas formas de derivar una misma cadena. Es posible que una misma cadena tenga dos árboles de derivación distintos.

Definición 3.7 *Una GLC G es ambigua si existen dos árboles de derivación distintos para G que generan una misma cadena $w \in \mathcal{L}(G)$.*

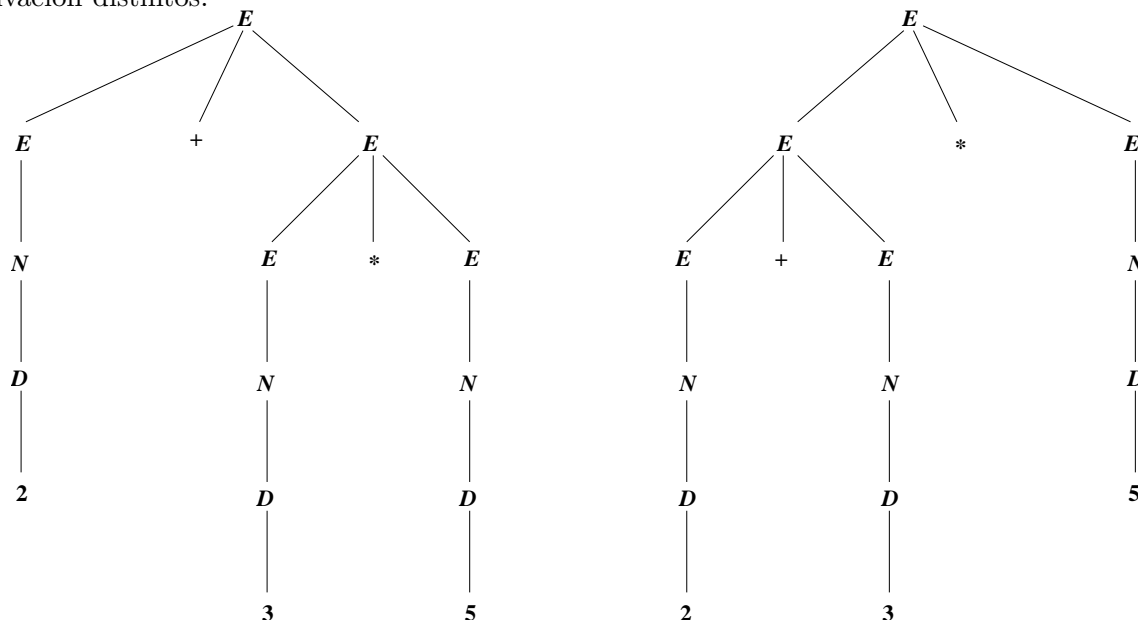
Generalmente ser ambigua es una propiedad indeseable para una GLC. Veamos un ejemplo de una GLC ambigua.

Ejemplo 3.4 La siguiente GLC describe un subconjunto de expresiones aritméticas correctas.

$$\begin{array}{ll}
 E \longrightarrow E + E & N \longrightarrow D \\
 E \longrightarrow E * E & N \longrightarrow DN \\
 E \longrightarrow (E) & D \longrightarrow 0 \\
 E \longrightarrow N & D \longrightarrow \dots \\
 & D \longrightarrow 9
 \end{array}$$

donde $V = \{E, N, D\}$, E es el símbolo inicial, y todos los demás son terminales.

Por ejemplo, $2 + 3 * 5$ pertenece al lenguaje generado por esta GLC, pero tiene dos árboles de derivación distintos:



Dado que lo normal es asignar semántica a una expresión a partir de su árbol de derivación, el valor en este ejemplo puede ser 25 ó 17 según qué árbol utilicemos para generarla.

Cuando se tiene una gramática ambigua, es posible *desambiguarla*, mediante escribir otra que genere el mismo lenguaje pero no sea ambigua.

Ejemplo 3.5 La siguiente GLC genera el mismo lenguaje que la del Ej. 3.4, pero no es ambigua. La técnica usada ha sido distinguir lo que son sumandos (o términos T) de factores (F), de modo de forzar la precedencia $*$, $+$.

$$\begin{array}{ll}
 E \longrightarrow E + T & N \longrightarrow D \\
 E \longrightarrow T & N \longrightarrow DN \\
 T \longrightarrow T * F & D \longrightarrow 0 \\
 T \longrightarrow F & D \longrightarrow \dots \\
 F \longrightarrow (E) & D \longrightarrow 9 \\
 F \longrightarrow N &
 \end{array}$$

Ahora el lector puede verificar que $2 + 3 * 5$ sólo permite la derivación que queremos, pues hemos obligado a que primero se consideren los sumandos y luego los factores.

3.2 Todo Lenguaje Regular es Libre del Contexto

[LP81, sec 3.2]

Hemos ya mostrado (Ej. 3.1) que existen lenguajes LC que no son regulares. Vamos ahora a completar esta observación con algo más profundo: el conjunto de los lenguajes LC incluye al de los regulares.

Teorema 3.1 *Si $L \subseteq \Sigma^*$ es un lenguaje regular, entonces L es LC.*

Prueba: Lo demostramos por inducción estructural sobre la ER que genera L . Sería más fácil usando autómatas finitos y de pila (que veremos enseguida), pero esta demostración ilustra otros hechos útiles para más adelante.

1. Si $L = \emptyset$, la GLC $G = (\{S\}, \Sigma, \emptyset, S)$ genera L . ¡Esta es una GLC sin reglas!
2. Si $L = \{a\}$, la GLC $G = (\{S\}, \Sigma, \{S \rightarrow a\}, S)$ genera L .
3. Si $L = L_1 \cup L_2$ y tenemos (por hipótesis inductiva) GLCs $G_1 = (V_1, \Sigma, R_1, S_1)$ y $G_2 = (V_2, \Sigma, R_2, S_2)$ que generan L_1 y L_2 respectivamente, entonces la GLC $G = (V_1 \cup V_2 \cup \{S\}, \Sigma, R_1 \cup R_2 \cup \{S \rightarrow S_1, S \rightarrow S_2\}, S)$ genera L .
4. Si $L = L_1 \circ L_2$ y tenemos GLCs $G_1 = (V_1, \Sigma, R_1, S_1)$ y $G_2 = (V_2, \Sigma, R_2, S_2)$ que generan L_1 y L_2 respectivamente, entonces la GLC $G = (V_1 \cup V_2 \cup \{S\}, \Sigma, R_1 \cup R_2 \cup \{S \rightarrow S_1 S_2\}, S)$ genera L .
5. Si $L = L_1^*$ y tenemos una GLC $G_1 = (V_1, \Sigma, R_1, S_1)$ que genera L_1 , entonces la GLC $G = (V_1 \cup \{S\}, \Sigma, R_1 \cup \{S \rightarrow S_1 S, S \rightarrow \varepsilon\}, S)$ genera L .

□

Ejemplo 3.6 Si derivamos una GLC para $(a \star | b) \star a$ obtendremos

$$\begin{array}{ll}
 S & \longrightarrow S_1 S_2 & S_4 & \longrightarrow S_6 S_4 \\
 S_1 & \longrightarrow S_3 S_1 & S_4 & \longrightarrow \varepsilon \\
 S_1 & \longrightarrow \varepsilon & S_6 & \longrightarrow a \\
 S_3 & \longrightarrow S_4 & S_5 & \longrightarrow b \\
 S_3 & \longrightarrow S_5 & S_2 & \longrightarrow a
 \end{array}$$

El Teo. 3.1 nos muestra cómo convertir cualquier ER en una GLC. Con esto a mano, nos permitiremos escribir ERs en los lados derechos de las reglas de una GLC.

Ejemplo 3.7 La GLC del Ej. 3.5 se puede escribir de la siguiente forma.

$$\begin{array}{ll}
 E & \longrightarrow E + T \mid T \\
 T & \longrightarrow T * F \mid F \\
 F & \longrightarrow (E) \mid DD \star \\
 D & \longrightarrow 0 \mid \dots \mid 9
 \end{array}$$

si bien, para cualquier propósito formal, deberemos antes convertirla a la forma básica.

Ejemplo 3.8 Tal como con ERs, no siempre es fácil diseñar una GLC que genere cierto lenguaje. Un ejercicio interesante es $\{w \in \{a, b\}^*, w \text{ tiene la misma cantidad de } a\text{'s y } b\text{'s}\}$. Una respuesta sorprendentemente sencilla es $S \rightarrow \varepsilon \mid aSbS \mid bSaS$. Es fácil ver por inducción que genera solamente cadenas correctas, pero es un buen ejercicio convencerse de que genera todas las cadenas correctas.

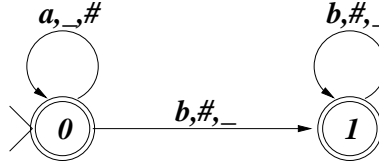
3.3 Autómatas de Pila (AP)

[LP81, sec 3.3]

Tal como en el Capítulo 2, donde teníamos un mecanismo para generar lenguajes regulares (las ERs) y otro para reconocerlos (AFDs y AFNDs), tendremos mecanismos para generar lenguajes LC (las GLCs) y para reconocerlos. Estos últimos son una extensión de los AFNDs, donde además de utilizar el estado del autómata como memoria del pasado, es posible almacenar una cantidad arbitraria de símbolos en una *pila*.

Un *autómata de pila (AP)* se diferencia de un AFND en que las transiciones involucran condiciones no sólo sobre la cadena de entrada sino también sobre los símbolos que hay en el tope de la pila. Asimismo la transición puede involucrar realizar cambios en el tope de la pila.

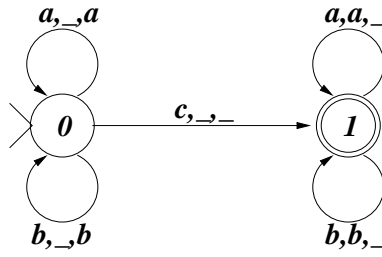
Ejemplo 3.9 Consideremos el siguiente AP:



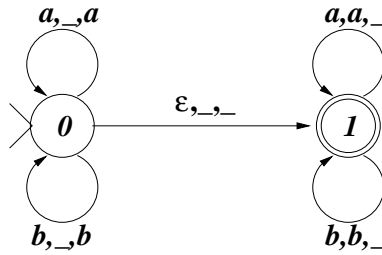
Las transiciones se leen de la siguiente manera: $(a, -, \#)$ significa que, al leerse una a , independientemente de los caracteres que haya en el tope de la pila ($-$), se apilará un símbolo $\#$ y se seguirá la transición; $(b, \#, -)$ significa que, al leerse una b , si hay un símbolo $\#$ en el tope de la pila, se desapilará (es decir, se reemplazará por $-$, que denota la cadena vacía en los dibujos). El AP acepta la cadena sólo si es posible llegar a un estado final habiendo consumido toda la entrada y quedando con la pila vacía. Es fácil ver que el AP del ejemplo acepta las cadenas del lenguaje $\{a^n b^n, n \geq 0\}$. Notar que el estado 0 es final para poder aceptar la cadena vacía.

Observación 3.2 *Notar que una condición $-$ sobre la pila no quiere decir que la pila debe estar vacía. Eso no se puede expresar directamente. Lo que se puede expresar es “en el tope de la pila deben estar estos caracteres”. Cuando esa condición es $-$ lo que se está diciendo es que no hay ninguna condición sobre el tope de la pila, es decir, que los cero caracteres del tope de la pila deben formar ε , lo que siempre es cierto.*

Ejemplo 3.10 ¿Cómo sería un AP que reconociera las cadenas de $\{wcw^R, w \in \{a, b\}^*\}$? Esta vez debemos recordar qué carácter vimos antes, no basta con apilar contadores $\#$:



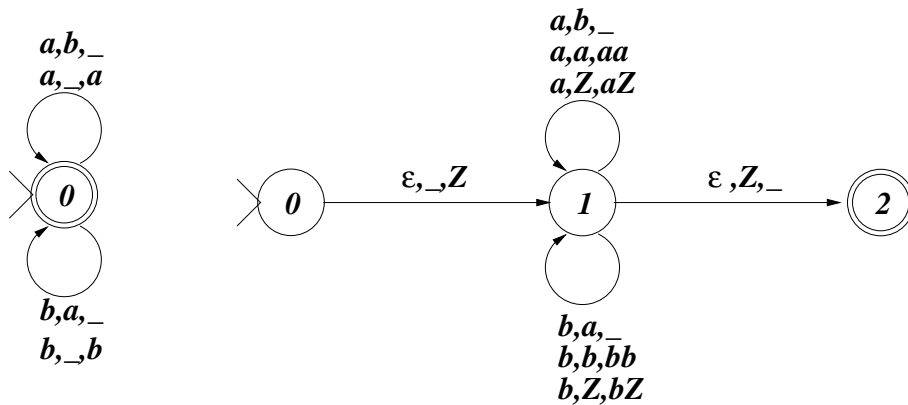
Una pregunta un poco más complicada es: ¿cómo aceptar $\{ww^R, w \in \{a,b\}^*\}$? ¡Esta vez no tenemos una marca (c) que nos indique cuándo empezar a desapilar! La solución se basa en explotar el no determinismo intrínseco de los APs:



De este modo el AP *adivina* qué transición elegir (es decir cuándo empezar a desapilar), tal como lo hacían los AFNDs.

Algunas veces es más fácil diseñar un AP que una GLC para describir un cierto lenguaje. Por ejemplo, sin entrenamiento previo no es sencillo dibujar un AP que reconozca el lenguaje del Ej. 3.5, mientras que el siguiente ejemplo muestra la situación opuesta.

Ejemplo 3.11 Generar un AP que reconozca el lenguaje del Ej. 3.8 es bastante más intuitivo, y es más fácil ver que funciona correctamente. Esencialmente el AP almacena en la pila el exceso de a 's sobre b 's o viceversa. Cuando la pila está vacía las cantidades son iguales. Sino, la pila debería contener sólo a 's o sólo b 's. La idea es que, cuando se recibe una a y en el tope de la pila hay una b , se “cancelan” consumiendo la a y desapilando la b , y viceversa. Cuando se reciba una a y en la pila haya exceso de a 's, se apila la nueva a , y lo mismo con b . El problema es que debe ser posible apilar la nueva letra cuando la pila está vacía. Como no puede expresarse el hecho de que la pila debe estar vacía, presentamos dos soluciones al problema.



Comencemos por el AP de la izquierda. Este AP puede siempre apilar la letra que viene (en particular, si la pila está vacía). Puede apilar incorrectamente una a cuando la pila contiene b 's, en cuyo caso puede no aceptar una cadena que pertenece al lenguaje (es decir, puede quedar con la pila no vacía, aunque ésta contenga igual cantidad de a 's y b 's). Sin embargo, debe notarse que el AP es no determinístico, y basta con que exista una secuencia de transiciones que termine en estado final con la pila vacía para que el AP acepte la cadena. Es decir, si bien hay caminos que en cadenas correctas no vacían la pila, siempre hay caminos que lo hacen, y por ello el AP funciona correctamente.

El AP de la derecha es menos sutil pero es más fácil ver que es correcto. Además ilustra una técnica bastante común para poder detectar la pila vacía. Primero se apila un símbolo especial Z , de modo que luego se sabe que la pila está realmente vacía cuando se tiene Z en el tope. Ahora las transiciones pueden indicar precisamente qué hacer cuando viene una a según lo que haya en el tope de la pila: b (cancelar), a (apilar) y Z (apilar); similarmente con b . Obsérvese cómo se indica el apilar otra a cuando viene una a : la a del tope de la pila se reemplaza por aa . Finalmente, con la pila vacía se puede desapilar la Z y pasar al estado final.

Es hora de definir formalmente un AP y su funcionamiento.

Definición 3.8 Un autómata de pila (AP) es una tupla $M = (K, \Sigma, \Gamma, \Delta, s, F)$, tal que

- K es un conjunto finito de estados.
- Σ es un alfabeto finito.
- Γ es el alfabeto de la pila, finito.
- $s \in K$ es el estado inicial.
- $F \subseteq K$ son los estados finales.
- $\Delta \subseteq_F (K \times \Sigma^* \times \Gamma^*) \times (K \times \Gamma^*)$, conjunto finito de transiciones.

Las transiciones $((q, x, \alpha), (q', \beta))$ son las que hemos graficado como flechas rotuladas " x, α, β " que va de q a q' , y se pueden recorrer cuando viene x en la entrada y se lee α (de arriba hacia abajo) en el tope de la pila, de modo que al pasar a q' ese α se reemplazará por β .

Ejemplo 3.12 El segundo AP del Ej. 3.11 se describe formalmente como $M = (K, \Sigma, \Gamma, \Delta, s, F)$, donde $K = \{0, 1, 2\}$, $\Sigma = \{a, b\}$, $\Gamma = \{a, b, Z\}$, $s = 0$, $F = \{2\}$, y

$$\begin{aligned} \Delta = \{ & ((0, \varepsilon, \varepsilon), (1, Z)), ((1, \varepsilon, Z), (2, \varepsilon)), \\ & ((1, a, b), (1, \varepsilon)), ((1, a, a), (1, aa)), ((1, a, Z), (1, aZ)), \\ & ((1, b, a), (1, \varepsilon)), ((1, b, b), (1, bb)), ((1, b, Z), (1, bZ)) \} \end{aligned}$$

Nuevamente definiremos la noción de configuración, que contiene la información necesaria para completar el cómputo de un AP.

Definición 3.9 Una configuración de un AP $M = (K, \Sigma, \Gamma, \Delta, s, F)$ es un elemento de $\mathcal{C}_M = K \times \Sigma^* \times \Gamma^*$.

La idea es que la configuración (q, x, γ) indica que M está en el estado q , le falta leer la cadena x de la entrada, y el contenido completo de la pila (de arriba hacia abajo) es γ . Esta es información suficiente para predecir lo que ocurrirá en el futuro.

Lo siguiente es describir cómo el AP nos lleva de una configuración a la siguiente.

Definición 3.10 La relación lleva en un paso, $\vdash_M \subseteq \mathcal{C}_M \times \mathcal{C}_M$, para un AP $M = (K, \Sigma, \Gamma, \Delta, s, F)$, se define de la siguiente manera: $(q, xy, \alpha\gamma) \vdash_M (q', y, \beta\gamma)$ sii $((q, x, \alpha), (q', \beta)) \in \Delta$.

Definición 3.11 La relación lleva en cero o más pasos \vdash_M^* es la clausura reflexiva y transitiva de \vdash_M .

Escribiremos simplemente \vdash y \vdash^* en vez de \vdash_M y \vdash_M^* cuando quede claro de qué M estamos hablando.

Definición 3.12 El lenguaje aceptado por un AP $M = (K, \Sigma, \Gamma, \Delta, s, F)$ se define como

$$\mathcal{L}(M) = \{x \in \Sigma^*, \exists f \in F, (s, x, \varepsilon) \vdash_M^* (f, \varepsilon, \varepsilon)\}.$$

Ejemplo 3.13 Tomemos el segundo AP del Ej. 3.10, el que se describe formalmente como $M = (K, \Sigma, \Gamma, \Delta, s, F)$, donde $K = \{0, 1\}$, $\Sigma = \Gamma = \{a, b\}$, $s = 0$, $F = \{1\}$, y $\Delta = \{((0, a, \varepsilon), (0, a)), ((0, b, \varepsilon), (0, b)), ((0, \varepsilon, \varepsilon), (1, \varepsilon)), ((1, a, a), (1, \varepsilon)), ((1, b, b), (1, \varepsilon))\}$.

Consideremos la cadena de entrada $x = abbbba$ y escribamos las configuraciones por las que pasa M al recibir x como entrada:

$$\begin{aligned} (0, abbbba, \varepsilon) &\vdash (0, bbbba, a) \vdash (0, bbba, ba) \vdash (0, bba, bba) \\ &\vdash (1, bba, bba) \vdash (1, ba, ba) \vdash (1, a, a) \vdash (1, \varepsilon, \varepsilon). \end{aligned}$$

Por lo tanto $(0, x, \varepsilon) \vdash^* (1, \varepsilon, \varepsilon)$, y como $1 \in F$, tenemos que $x \in \mathcal{L}(M)$. Notar que existen otros caminos que no llevan a la configuración en que se acepta la cadena, pero tal como los AFNDs, la definición indica que basta que exista un camino que acepta x para que el AP acepte x .

3.4 Conversión de GLC a AP

[LP81, sec 3.4]

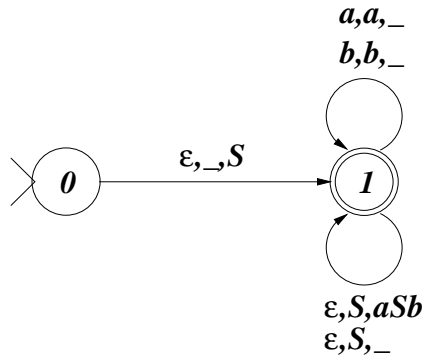
Vamos a demostrar ahora que ambos mecanismos, GLCs y APs, son equivalentes para denotar lenguajes libres del contexto. Comenzaremos con la conversión más sencilla.

Definición 3.13 Sea $G = (V, \Sigma, R, S)$ una GLC. Entonces definiremos $ap(G) = (K, \Sigma, \Gamma, \Delta, s, F)$ de la siguiente manera: $K = \{s, f\}$ (es decir el AP tiene sólo dos estados), $\Gamma = V \cup \Sigma$ (podemos apilar símbolos terminales y no terminales), $F = \{f\}$, y

$$\begin{aligned} \Delta = & \{((s, \varepsilon, \varepsilon), (f, S))\} \\ & \cup \{((f, a, a), (f, \varepsilon)), a \in \Sigma\} \\ & \cup \{((f, \varepsilon, A), (f, z)), A \rightarrow z \in R\} \end{aligned}$$

La idea de $ap(G)$ es que mantiene en la pila lo que aún espera leer. Esto está expresado como una secuencia de terminales y no terminales. Los terminales deben verse tal como aparecen, y “ver” un no terminal significa ver cualquier cadena de terminales que se pueda derivar de él. Por ello comienza indicando que espera ver S , cancela un terminal de la entrada con el que espera ver según la pila, y puede cambiar un no terminal del tope de la pila usando cualquier regla de derivación. Nótese que basta un AP de dos estados para simular cualquier GLC. Esto indica que los estados no son la parte importante de la memoria de un AP, sino su pila.

Ejemplo 3.14 Dibujemos $ap(G)$ para la GLC del Ej. 3.1. El AP resultante es bastante distinto de los que mostramos en el Ej. 3.9.



También es ahora muy fácil generar el AP para la GLC del Ej. 3.5, lo cual originalmente no era nada sencillo.

Obtener el AP de una GLC es el primer paso para poder hacer el parsing de un lenguaje LC. Demostraremos ahora que esta construcción es correcta.

Teorema 3.2 Sea G una GLC, entonces $\mathcal{L}(G) = \mathcal{L}(ap(G))$.

Prueba: El invariante es que en todo momento, si x es la cadena ya leída y γ es el contenido de la pila, entonces $S \Rightarrow_G^* x\gamma$. Este invariante se establece con la primera transición de s a f . Si en algún momento se ha leído toda la entrada x y la pila está vacía, es inmediato que $x \in \mathcal{L}(G)$. Es muy fácil ver que los dos tipos de transición mantienen el invariante, pues las que desapilan terminales de la entrada simplemente mueven el primer carácter de γ al final de x , y las que reemplazan un no terminal por la parte derecha de una regla ejecutan un paso de \Rightarrow_G . Con esto demostramos que $x \in \mathcal{L}(ap(G)) \Rightarrow x \in \mathcal{L}(G)$, es decir toda cadena aceptada por el AP es generada por la GLC. Para ver la vuelta, basta considerar una secuencia de pasos \Rightarrow_G^* que genere x desde S , donde el no terminal que se expanda siempre sea el de más a la izquierda. Se puede ver entonces que el AP puede realizar las transiciones consumiendo los terminales que aparecen al comienzo de la cadena que se va derivando y reemplazando los no terminales del comienzo por la misma regla que se usa en la derivación de x . \square

3.5 Conversión a AP a GLC

[LP81, sec 3.4]

Esta conversión es un poco más complicada, pero imprescindible para demostrar la equivalencia entre GLCs y APs. La idea es generar una gramática donde los no terminales son de la forma $\langle p, A, q \rangle$ y expresen el *objetivo* de llegar del estado p al estado q , partiendo con una pila que contiene el símbolo A y terminando con la pila vacía. Permitiremos también objetivos de la forma $\langle p, \varepsilon, q \rangle$, que indican llegar de p a q partiendo y terminando con la pila vacía. Usando las transiciones del AP, reescribiremos algunos objetivos en términos de otros de todas las formas posibles.

Para simplificar este proceso, supondremos que el AP no pone condiciones sobre más de un símbolo de la pila a la vez.

Definición 3.14 Un AP simplificado $M = (K, \Sigma, \Gamma, \Delta, s, F)$ cumple que $((p, x, \alpha), (q, \beta)) \in \Delta \Rightarrow |\alpha| \leq 1$.

Es fácil “simplificar” un AP. Basta reemplazar las transiciones de tipo $((p, x, a_1 a_2 a_3), (q, \beta))$ por una secuencia de estados nuevos: $((p, x, a_1), (p_1, \varepsilon)), ((p_1, \varepsilon, a_2), (p_2, \varepsilon)), ((p_2, \varepsilon, a_3), (q, \beta))$. Ahora definiremos la GLC que se asocia a un AP simplificado.

Definición 3.15 Sea $M = (K, \Sigma, \Gamma, \Delta, s, F)$ un AP simplificado. Entonces la GLC $glc(M) = (V, \Sigma, R, S)$ se define como $V = \{S\} \cup (K \times (\Gamma \cup \{\varepsilon\}) \times K)$, y las siguientes

reglas.

$$\begin{aligned}
R = & \{S \longrightarrow \langle s, \varepsilon, f \rangle, f \in F\} \\
& \cup \{ \langle p, \varepsilon, p \rangle \longrightarrow \varepsilon, p \in K \} \\
& \cup \{ \langle p, A, r \rangle \longrightarrow x \langle q, \varepsilon, r \rangle, ((p, x, A), (q, \varepsilon)) \in \Delta, r \in K \} \\
& \cup \{ \langle p, \varepsilon, r \rangle \longrightarrow x \langle q, \varepsilon, r \rangle, ((p, x, \varepsilon), (q, \varepsilon)) \in \Delta, r \in K \} \\
& \cup \{ \langle p, A, r \rangle \longrightarrow x \langle q, A, r \rangle, ((p, x, \varepsilon), (q, \varepsilon)) \in \Delta, r \in K, A \in \Gamma \} \\
& \cup \{ \langle p, A, r \rangle \longrightarrow x \langle q, B_1, r_1 \rangle \langle r_1, B_2, r_2 \rangle \dots \langle r_{k-1}, B_k, r \rangle, \\
& \quad ((p, x, A), (q, B_1 B_2 \dots B_k)) \in \Delta, r_1, r_2, \dots, r_{k-1}, r \in K \} \\
& \cup \{ \langle p, \varepsilon, r \rangle \longrightarrow x \langle q, B_1, r_1 \rangle \langle r_1, B_2, r_2 \rangle \dots \langle r_{k-1}, B_k, r \rangle, \\
& \quad ((p, x, \varepsilon), (q, B_1 B_2 \dots B_k)) \in \Delta, r_1, r_2, \dots, r_{k-1}, r \in K \} \\
& \cup \{ \langle p, A, r \rangle \longrightarrow x \langle q, B_1, r_1 \rangle \langle r_1, B_2, r_2 \rangle \dots \langle r_{k-1}, B_k, r_k \rangle \langle r_k, A, r \rangle, \\
& \quad ((p, x, \varepsilon), (q, B_1 B_2 \dots B_k)) \in \Delta, r_1, r_2, \dots, r_k, r \in K \}
\end{aligned}$$

Vamos a explicar ahora el funcionamiento de $glc(M)$.

Teorema 3.3 *Sea M un AP simplificado, entonces $\mathcal{L}(M) = \mathcal{L}(glc(M))$.*

Prueba: Como se explicó antes, las tuplas $\langle p, A, q \rangle$ o $\langle p, \varepsilon, q \rangle$ representan objetivos a cumplir, o también el lenguaje de las cadenas que llevan de p a q eliminando A de la pila o yendo de pila vacía a pila vacía, respectivamente. La primera línea de R establece que las cadenas que acepta el AP son las que lo llevan del estado inicial s hasta algún estado final $f \in F$, partiendo y terminando con la pila vacía. La segunda línea establece que la cadena vacía me lleva de p a p sin alterar la pila (es la única forma de eliminar no terminales en la GLC). La tercera línea dice que, si queremos pasar de p a algún r consumiendo A de la pila en el camino, y tenemos una transición del AP que me lleva de p a q consumiendo x de la entrada y A de la pila, entonces una forma de cumplir el objetivo es generar x y luego ir de q a r partiendo y terminando con la pila vacía. La cuarta línea es similar, pero la transición no altera la pila, por lo que me sirve para objetivos del tipo $\langle p, \varepsilon, r \rangle$. Sin embargo, podría usar esa transición también para objetivos tipo $\langle p, A, q \rangle$, si paso a q y dejo como siguiente objetivo $\langle q, A, r \rangle$ (quinta línea). Las últimas tres líneas son análogas a las líneas 3–5, esta vez considerando el caso más complejo en que la transición no elimina A de la pila sino que la reemplaza por $B_1 B_2 \dots B_k$. En ese caso, el objetivo de ir de p a r se reemplaza por una secuencia de objetivos, cada uno de los cuales se encarga de eliminar una de las B_i de la pila por vez, pasando por estados intermedios desconocidos (y por eso se agregan reglas para usar todos los estados intermedios posibles). Esto no constituye una demostración sino más bien una explicación intuitiva de por qué $glc(M)$ debería funcionar como esperamos. Una demostración basada en inducción en el largo de las derivaciones se puede encontrar en el libro [LP81]. \square

Ejemplo 3.15 Calculemos $glc(M)$ para el AP de la izquierda del Ej. 3.11, que tiene un sólo estado. La GLC resultante es

S	\longrightarrow	$\langle 0, \varepsilon, 0 \rangle$	primera línea Def. 3.15
$\langle 0, \varepsilon, 0 \rangle$	\longrightarrow	ε	segunda línea Def. 3.15
$\langle 0, b, 0 \rangle$	\longrightarrow	$a\langle 0, \varepsilon, 0 \rangle$	generada por $((0, a, b), (0, \varepsilon))$
$\langle 0, \varepsilon, 0 \rangle$	\longrightarrow	$a\langle 0, a, 0 \rangle$	generadas por $((0, a, \varepsilon), (0, a))$
$\langle 0, a, 0 \rangle$	\longrightarrow	$a\langle 0, a, 0 \rangle\langle 0, a, 0 \rangle$	
$\langle 0, b, 0 \rangle$	\longrightarrow	$a\langle 0, a, 0 \rangle\langle 0, b, 0 \rangle$	
$\langle 0, a, 0 \rangle$	\longrightarrow	$b\langle 0, \varepsilon, 0 \rangle$	generada por $((0, b, a), (0, \varepsilon))$
$\langle 0, \varepsilon, 0 \rangle$	\longrightarrow	$b\langle 0, b, 0 \rangle$	generadas por $((0, b, \varepsilon), (0, b))$
$\langle 0, b, 0 \rangle$	\longrightarrow	$b\langle 0, b, 0 \rangle\langle 0, b, 0 \rangle$	
$\langle 0, a, 0 \rangle$	\longrightarrow	$b\langle 0, b, 0 \rangle\langle 0, a, 0 \rangle$	

Para hacer algo de luz sobre esta GLC, podemos identificar $\langle 0, \varepsilon, 0 \rangle$ con S , y llamar B a $\langle 0, a, 0 \rangle$ y A a $\langle 0, b, 0 \rangle$. En ese caso obtendremos una solución novedosa al problema original del Ej. 3.8.

$$\begin{aligned}
S &\longrightarrow \varepsilon \mid aB \mid bA \\
A &\longrightarrow aS \mid bAA \mid aBA \\
B &\longrightarrow bS \mid aBB \mid bAB
\end{aligned}$$

Es fácil comprender cómo funciona esta gramática que hemos obtenido automáticamente. A representa la tarea de generar una a de más, B la de generar una b de más, y S la de producir una secuencia balanceada. Entonces S se reescribe como: la cadena vacía, o bien generar una a y luego compensarla con una B , o bien generar una b y luego compensarla con una A . A se reescribe como: compensar la a y luego generar una secuencia balanceada S , o bien generar una b (aumentando el desbalance) y luego compensar con dos A 's. Similarmente con B . Las terceras alternativas $A \longrightarrow aBA$ y $B \longrightarrow bAB$ son redundantes. Observar que provienen de apilar una letra incorrectamente en vez de cancelarla con la letra de la pila, como se discutió en el Ej. 3.8.

Ejemplo 3.16 Repitamos el procedimiento para el AP del Ej. 3.9. Este tiene dos estados, por lo que la cantidad de reglas que se generarán es mayor.

S	\longrightarrow	$\langle 0, \varepsilon, 0 \rangle$	primera línea Def. 3.15
S	\longrightarrow	$\langle 0, \varepsilon, 1 \rangle$	
$\langle 0, \varepsilon, 0 \rangle$	\longrightarrow	ε	segunda línea Def. 3.15
$\langle 1, \varepsilon, 1 \rangle$	\longrightarrow	ε	
$\langle 0, \varepsilon, 0 \rangle$	\longrightarrow	$a\langle 0, \#, 0 \rangle$	generadas por $((0, a, \varepsilon), (0, \#))$
$\langle 0, \varepsilon, 1 \rangle$	\longrightarrow	$a\langle 0, \#, 1 \rangle$	
$\langle 0, \#, 0 \rangle$	\longrightarrow	$a\langle 0, \#, 0 \rangle\langle 0, \#, 0 \rangle$	
$\langle 0, \#, 0 \rangle$	\longrightarrow	$a\langle 0, \#, 1 \rangle\langle 1, \#, 0 \rangle$	
$\langle 0, \#, 1 \rangle$	\longrightarrow	$a\langle 0, \#, 0 \rangle\langle 0, \#, 1 \rangle$	
$\langle 0, \#, 1 \rangle$	\longrightarrow	$a\langle 0, \#, 1 \rangle\langle 1, \#, 1 \rangle$	
$\langle 0, \#, 0 \rangle$	\longrightarrow	$b\langle 1, \varepsilon, 0 \rangle$	generadas por $((0, b, \#), (1, \varepsilon))$
$\langle 0, \#, 1 \rangle$	\longrightarrow	$b\langle 1, \varepsilon, 1 \rangle$	
$\langle 1, \#, 0 \rangle$	\longrightarrow	$b\langle 1, \varepsilon, 0 \rangle$	generadas por $((1, b, \#), (1, \varepsilon))$
$\langle 1, \#, 1 \rangle$	\longrightarrow	$b\langle 1, \varepsilon, 1 \rangle$	

Nuevamente, simplifiquemos la GLC para comprenderla. Primero, se puede ver que los no terminales de la forma $\langle 1, *, 0 \rangle$ son inútiles pues reducen siempre a otros del mismo tipo, de modo que nunca generarán una secuencia de terminales. Esto demuestra que las reglas en las líneas 8, 11 y 13 pueden eliminarse. Similarmente, no hay forma de generar cadenas de terminales a partir de $\langle 0, \#, 0 \rangle$, lo que permite eliminar las reglas 5, 7 y 9. Podemos deshacernos de no terminales que reescriben de una única manera, reemplazándolos por su parte derecha. Llamando $X = \langle 0, \#, 1 \rangle$ al único no terminal sobreviviente aparte de S tenemos la curiosa gramática:

$$\begin{array}{lcl} S & \longrightarrow & \varepsilon \mid aX \\ X & \longrightarrow & b \mid aXb \end{array}$$

la cual no es difícil identificar con la mucho más intuitiva $S \longrightarrow \varepsilon \mid aSb$. La asimetría que ha aparecido es consecuencia del tratamiento especial que recibe la b que comienza el desapilado en el AP del Ej. 3.9.

El siguiente teorema fundamental de los lenguajes LC completa el círculo.

Teorema 3.4 *Todo lenguaje libre del contexto puede ser especificado por una GLC, o alternativamente, por un AP.*

Prueba: Inmediato a partir de los Teos. 3.2 y 3.3. □

3.6 Teorema de Bombeo

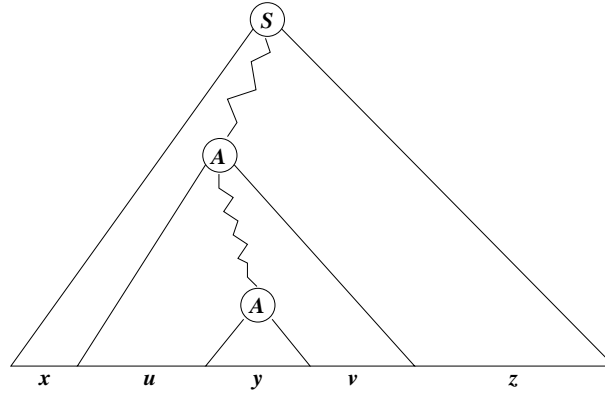
[LP81, sec 3.5.2]

Nuevamente, presentaremos una forma de determinar que ciertos lenguajes no son LC. El mecanismo es similar al visto para lenguajes regulares (Sección 2.8), si bien el tipo de repetitividad que se produce en los lenguajes LC es ligeramente más complejo.

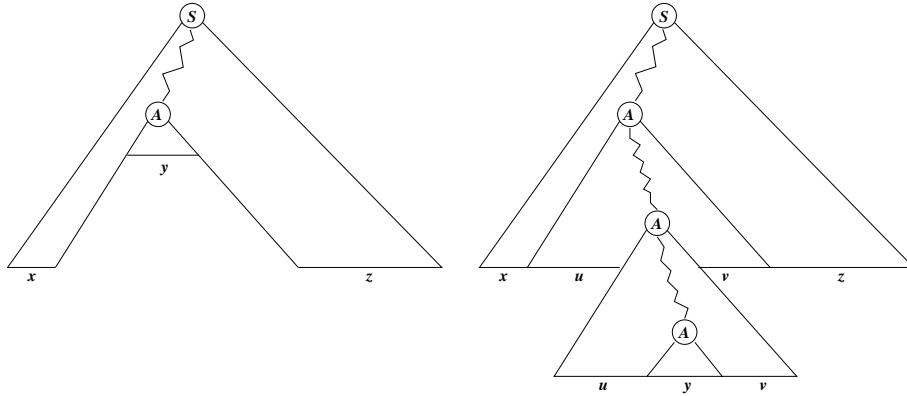
Teorema 3.5 (Teorema de Bombeo)

Sea L un lenguaje LC. Entonces existe un número $N > 0$ tal que toda cadena $w \in L$ de largo $|w| > N$ se puede escribir como $w = xuyvz$ de modo que $uv \neq \varepsilon$, $|uyv| \leq N$, y $\forall n \geq 0, xu^n yv^n z \in L$.

Prueba: Sea $G = (V, \Sigma, R, S)$ una GLC que genera L . Comenzaremos por mostrar que una cadena suficientemente larga necesita de un árbol de derivación suficientemente alto. Sea $p = \max_{A \rightarrow z \in R} |z|$ el largo máximo de la parte derecha de una regla. Entonces es fácil ver que un árbol de derivación de altura h (midiendo altura como la máxima cantidad de aristas desde la raíz hasta una hoja) no puede producir cadenas de largo superior a p^h . Tomaremos entonces $N = p^{|V|-1}$, de modo que toda cadena de largo $> N$ tiene un árbol de derivación de altura al menos $|V|$. Lo fundamental es que esto significa que, en algún camino desde la raíz hasta una hoja, debe haber un no terminal repetido. Esto se debe a que, en cualquier camino del árbol de derivación, hay tantos nodos internos (rotulados por no terminales) como el largo del camino.



Hemos asignado nombres a las partes de la cadena que deriva el árbol de acuerdo a la partición $w = xuyvz$ que realizaremos. Obsérvese que, por ser el mecanismo de expansión de no terminales libre del contexto, cada vez que aparece A podríamos elegir expandirlo como queramos. En el ejemplo, la primera vez elegimos reglas que llevaron $A \Rightarrow^* uyv$ y la segunda vez $A \Rightarrow^* y$. Pero podríamos haber elegido $A \Rightarrow^* y$ la primera vez. O podríamos haber elegido $A \Rightarrow^* uyv$ la segunda vez. En general podríamos haber generado $S \Rightarrow^* xu^n y v^n z$ para cualquier $n \geq 0$.



Sólo queda aclarar que $|uyv| \leq N$ porque de otro modo se podría aplicar el argumento al subárbol cuya raíz es la A superior, para obtener un uyv más corto; y que $uv \neq \varepsilon$ porque de otro modo podríamos repetir el argumento sobre xyz , la cual aún es suficientemente larga y por lo tanto debe tener un árbol suficientemente alto. Es decir, no puede ser posible eliminar todos los caminos suficientemente largos con este argumento y terminar teniendo un árbol muy pequeño para la cadena que deriva. \square

Ejemplo 3.17 Usemos el Teorema de Bombeo para demostrar que $L = \{a^n b^n c^n, n \geq 0\}$ no es LC. Dado el N , elegimos $w = a^N b^N c^N$. Dentro de w el adversario puede elegir u y v como quiera, y en cualquiera de los casos la cadena deja de pertenecer a L si eliminamos u y v de w .

Ejemplo 3.18 Otro ejemplo importante es mostrar que $L = \{ww, w \in \{a, b\}^*\}$ no es LC. Para ello, tomemos $w = a^N b^N a^N b^N \in L$. Debido a que $|uyv| \leq N$, el adversario no puede elegir u dentro de la primera zona de a 's (o b 's) y v dentro de la segunda. Cualquier otra elección hará que $xyz \notin L$.

3.7 Propiedades de Clausura

[LP81, sec 3.5.1 y 3.5.2]

Hemos visto que los lenguajes regulares se pueden operar de diversas formas y el resultado sigue siendo regular (Sección 2.7). Algunas de esas propiedades se mantienen en los lenguajes LC, pero otras no.

Lema 3.2 *La unión, concatenación y clausura de Kleene de lenguajes LC es LC.*

Prueba: Basta recordar las construcciones hechas para demostrar el Teo. 3.1. \square

Lema 3.3 *La intersección de dos lenguajes LC no necesariamente es LC.*

Prueba: Considérense los lenguajes $L_{ab} = \{a^n b^n c^m, n, m \geq 0\}$ y $L_{bc} = \{a^m b^n c^n, n, m \geq 0\}$. Está claro que ambos son LC, pues $L_{ab} = \{a^n b^n, n \geq 0\} \circ c^*$, y $L_{bc} = a^* \circ \{b^n c^n, n \geq 0\}$. Sin embargo, $L_{ab} \cap L_{bc} = \{a^n b^n c^n, n \geq 0\}$, el cual hemos visto en el Ej. 3.17 que no es LC. \square

Lema 3.4 *El complemento de un lenguaje LC no necesariamente es LC.*

Prueba: $L_1 \cap L_2 = (L_1^c \cup L_2^c)^c$, de manera que si el complemento de un lenguaje LC fuera siempre LC, entonces el Lema 3.3 sería falso. \square

Observación 3.3 *El que no siempre se puedan complementar los lenguajes LC nos dice que el hecho de que los APs sean no determinísticos no es superficial (como lo era el no determinismo de los AFNDs), sino un hecho que difícilmente se podrá eliminar. Esto tiene consecuencias importantes para el uso práctico de los APs, lo que se discutirá en las siguientes secciones.*

Observación 3.4 *Es interesante que sí se puede intersectar un lenguaje LC con uno regular para obtener un lenguaje LC. Sea $M_1 = (K_1, \Sigma, \Gamma, \Delta_1, s_1, F_1)$ el AP y $M_2 = (K_2, \Sigma, \delta, s_2, F_2)$ el AFD correspondientes. Entonces el AP $M = (K_1 \times K_2, \Sigma, \Gamma, \Delta, (s_1, s_2), F_1 \times F_2)$, con*

$$\Delta = \{((p_1, p_2), x, \alpha), ((q_1, q_2), \beta)), ((p_1, x, \alpha), (q_1, \beta)) \in \Delta_1, (p_2, x) \vdash_{M_2}^* (q_2, \varepsilon)\},$$

reconoce la intersección de los dos lenguajes. La idea es recordar en los estados de M en qué estado están ambos autómatas simultáneamente. El problema para intersectar dos APs es que hacen cosas distintas con una misma pila, pero ese problema no se presenta aquí.

El método descrito nos da también una forma de intersectar dos lenguajes regulares más directa que la vista en la Sección 2.7.

3.8 Propiedades Algorítmicas

[LP81, sec 3.5.3]

Veremos un par de preguntas sobre lenguajes LC que pueden responderse algorítmicamente. Las que faltan con respecto a los regulares no pueden responderse, pero esto se verá mucho más adelante.

Lema 3.5 *Dado un lenguaje LC L y una cadena w , existe un algoritmo para determinar si $w \in L$.*

Prueba: Lo natural parecería ser usar un AP, pero esto no es tan sencillo como parece: el AP no es determinístico y la cantidad de estados potenciales es infinita (considerando la pila). Aún peor, puede pasar mucho tiempo operando sólo en la pila sin consumir caracteres de la entrada. No es fácil determinar si alguna vez consumirá la cadena o no. Utilizaremos, en cambio, una GLC $G = (V, \Sigma, R, S)$ que genera L . La idea esencial es escribir todas las derivaciones posibles, hasta o bien generar w o bien determinar que w nunca será generada. Esto último es el problema. Para poder determinar cuándo detenernos, modificaremos G de modo que todas las producciones, o bien hagan crecer el largo de la cadena, o bien la dejen igual pero conviertan un no terminal en terminal. Si logramos esto, basta con probar todas las derivaciones de largo $2|w|$.

Debemos entonces eliminar dos tipos de reglas de G .

1. Reglas de la forma $A \rightarrow \varepsilon$, pues reducen el largo de la cadena derivada. Para poder eliminar esta regla, buscaremos todas las producciones de la forma $B \rightarrow xAy$ y *agregaremos* otra regla $B \rightarrow xy$ a G , adelantándonos al posible uso de $A \rightarrow \varepsilon$ en una derivación. Cuando hayamos hecho esto con todas las reglas que mencionen A en la parte derecha, podemos descartar la regla $A \rightarrow \varepsilon$. Nótese que no es necesario reprocesar las nuevas reglas introducidas según viejas reglas $A \rightarrow \varepsilon$ ya descartadas, pues la regla paralela correspondiente ya existe. Lo que sí merece atención es que pueden aparecer nuevas reglas de la forma $B \rightarrow \varepsilon$, las cuales deben introducirse al conjunto de reglas a eliminar. Este proceso no puede continuar indefinidamente porque existen a lo más $|V|$ reglas de este tipo. Incluso el conjunto de reglas nuevas que se introducirán está limitado por el hecho de que cada regla nueva tiene en su parte derecha una subsecuencia de alguna parte derecha original. Finalmente, nótese que si descartamos la regla $S \rightarrow \varepsilon$ cambiaremos el lenguaje, pues esta regla permitirá generar la cadena vacía y no habrá otra forma de generarla. Esto no es problema: si aparece esta regla, entonces si $w = \varepsilon$ se responde $w \in L$ y se termina, de otro modo se puede descartar la regla $S \rightarrow \varepsilon$ ya que no es relevante para otras producciones.
2. Reglas de la forma $A \rightarrow B$, pues no aumentan el largo de la cadena derivada y no convierten el no terminal en terminal. En este caso dibujamos el grafo dirigido de qué no terminales pueden convertirse en otros, de modo que $A \Rightarrow^* B$ si existe un camino de A a B en ese grafo. Para cada uno de estos caminos, tomamos todas las producciones de tipo $C \rightarrow xAy$ y agregamos $C \rightarrow xBy$, adelantándonos al posible uso de esas flechas. Cuando hemos agregado todas las reglas nuevas, eliminamos en bloque todas las reglas de tipo $A \rightarrow B$. Nuevamente, no podemos eliminar directamente las reglas de tipo $S \rightarrow A$, pero éstas se aplicarán a lo sumo una vez, como primera regla, y para ello basta permitir derivaciones de un paso más.

□

Lema 3.6 *Dado un lenguaje LC L , existe un algoritmo para determinar si $L = \emptyset$.*

Prueba: El punto tiene mucho que ver con la demostración del Teo. 3.5. Si la gramática $G = (V, \Sigma, R, S)$ asociada a L genera alguna cadena, entonces puede generar una cadena sin repetir

símbolos no terminales en el camino de la raíz a una hoja del árbol de derivación (pues basta reemplazar el subárbol que cuelga de la primera ocurrencia por el que cuelga de la última, tantas veces como sea necesario, para quedarnos con un árbol que genera otra cadena y no repite símbolos no terminales). Por ello, basta con escribir todos los árboles de derivación de altura $|V|$. Si para entonces no se ha generado una cadena, no se generará ninguna. \square

3.9 Determinismo y Parsing

[LP81, sec 3.6]

Las secciones anteriores levantan una pregunta práctica evidente: ¿cómo se puede parsear eficientemente un lenguaje? El hecho de que el no determinismo de los APs no sea superficial, y de que hayamos utilizado un método tan tortuoso e ineficiente para responder si $w \in L$ en el Lema 3.5, indican que parsear eficientemente un lenguaje LC no es algo tan inmediato como para un lenguaje regular.

Lo primero es un resultado que indica que es posible parsear cualquier lenguaje LC en tiempo polinomial (a diferencia del método del Lema 3.5).

Definición 3.16 *La forma normal de Chomsky para GLCs establece que se permiten tres tipos de reglas: $A \rightarrow BC$, $A \rightarrow a$, y $S \rightarrow \varepsilon$, donde A, B, C son no terminales, S es el símbolo inicial, y a es terminal. Para toda GLC que genere L , existe otra GLC en forma normal de Chomsky que genera L .*

Observación 3.5 *Si tenemos una GLC en forma normal de Chomsky, es posible determinar en tiempo $O(|R|n^3)$ si una cadena $w \in L$, donde $n = |w|$ y R son las reglas de la gramática. Esto se logra mediante programación dinámica, determinando para todo substring de w , $w_i w_{i+1} \dots w_j$, qué no terminales A derivan ese substring, $A \Rightarrow^* w_i w_{i+1} \dots w_j$. Para determinar esto se prueba, para cada regla $A \rightarrow BC$, si existe un k tal que $B \Rightarrow^* w_i \dots w_k$ y $C \Rightarrow^* w_{k+1} \dots w_j$.*

Este resultado es interesante, pero aún no lo suficientemente práctico. Realmente necesitamos algoritmos de tiempo lineal para determinar si $w \in L$. Esto sería factible si el AP que usamos fuera *determinístico*: en cualquier situación posible, este AP debe tener a lo sumo una transición a seguir.

Definición 3.17 *Dos reglas $((q, x, \alpha), (p, \beta)) \neq ((q, x', \alpha'), (p', \beta'))$ de un AP colisionan si x es prefijo de x' (o viceversa) y α es prefijo de α' (o viceversa).*

Definición 3.18 *Un AP $M = (K, \Sigma, \Gamma, \Delta, s, F)$ es determinístico si no existen dos reglas $((q, x, \alpha), (p, \beta)) \neq ((q, x', \alpha'), (p', \beta')) \in \Delta$ que colisionan.*

Un AP no determinístico tiene un estado q del que parten dos transiciones, de modo que puede elegir cualquiera de las dos si en la entrada viene el prefijo común de x y x' , y en el tope de la pila se puede leer el prefijo común de α y α' . No todo AP puede convertirse a determinístico.

Ejemplo 3.19 El lenguaje $L = \{a^{m_1}ba^{m_2}b \dots ba^{m_k}, k \geq 2, m_1, m_1, \dots, m_k \geq 0, m_i \neq m_j \text{ para algún } i, j\}$ es LC pero no puede reconocerse por ningún AP determinístico.

Un AP determinístico es el del Ej. 3.9, así como el de wcw^R en el Ej. 3.10. El AP del ww^R del mismo Ej. 3.9 no es determinístico, pues la transición $((0, \varepsilon, \varepsilon), (1, \varepsilon))$ colisiona con $((0, a, \varepsilon), (0, a))$ y $((0, b, \varepsilon), (0, b))$.

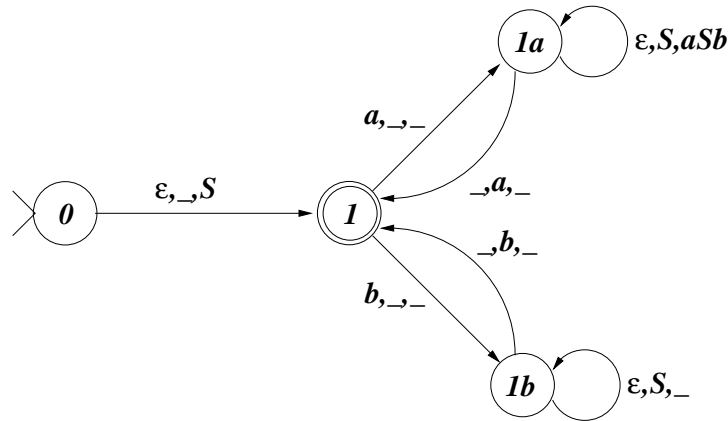
Lo que verdaderamente nos interesa es: ¿es posible diseñar un método para generar un AP determinístico a partir de una GLC? Ya sabemos que tal método no puede funcionar siempre, pero podemos aspirar a poderlo hacer en una buena cantidad de casos “interesantes”.

Parsing Top-Down: Lenguajes $LL(k)$

Volvamos a la conversión de GLC a AP del Teo. 3.2. La idea esencial es, dado que esperamos ver un cierto no terminal en la entrada, decidir de antemano qué regla aplicaremos para convertirlo en otra secuencia. El que debamos aplicar una regla entre varias puede introducir no determinismo.

Tomemos el Ej. 3.14. El AP resultante no es determinístico, pues las transiciones $((1, \varepsilon, S), (1, \varepsilon))$ y $((1, \varepsilon, S), (1, aSb))$ colisionan. Sin embargo, no es difícil determinar cuál es la que deberíamos seguir en cada caso: si el próximo carácter de la entrada es una a , debemos reemplazar la S que esperamos ver por aSb , mientras que si es una b debemos reemplazarla por ε . Esto sugiere la siguiente modificación del AP.

Ejemplo 3.20 La siguiente es una versión determinística del AP del Ej. 3.14.



El resultado es bastante distinto del que derivamos manualmente en el Ej. 3.9.

El mecanismo general es agregar al estado final f del AP generado por el Teo. 3.2 los estados f_c , $c \in \Sigma$, junto con transiciones de ida $((f, c, \varepsilon), (f_c, \varepsilon))$, y de vuelta $((f_c, \varepsilon, c), (f, \varepsilon))$. Si dentro de cada estado f_c podemos determinar la parte derecha que corresponde aplicar a cada no terminal que esté en el tope de la pila, habremos obtenido un AP determinístico.

Definición 3.19 Los lenguajes LC que se pueden reconocer con la construcción descrita arriba se llaman $LL(1)$. Si se pueden reconocer mediante mirar de antemano los k caracteres de la entrada se llaman $LL(k)$.

Este tipo de parsing se llama “top-down” porque se puede visualizar como generando el árbol de derivación desde arriba hacia abajo, pues decidimos qué producción utilizar (es decir la raíz del árbol) antes de ver la cadena que se derivará.

Esta técnica puede fracasar por razones superficiales, que se pueden corregir en la GLC de la que parte generándose el AP original.

1. Factorización a la izquierda. Consideremos las reglas $N \rightarrow D$ y $N \rightarrow DN$ en la GLC del Ej. 3.5. Por más que veamos que lo que sigue en la entrada es un dígito, no tenemos forma de saber qué regla aplicar en un $LL(1)$. Sin embargo, es muy fácil reemplazar estas reglas por $N \rightarrow DN'$, $N' \rightarrow \varepsilon$, $N' \rightarrow N$. En general si dos o más reglas comparten un prefijo común en su parte derecha, éste se puede factorizar.
2. Recursión a la izquierda. Para cualquier k fijo, puede ser imposible saber qué regla aplicar entre $E \rightarrow E + T$ y $E \rightarrow T$ (basta que siga una cadena de T de largo mayor que k). Este problema también puede resolverse, mediante reemplazar las reglas por $E \rightarrow TE'$, $E' \rightarrow \varepsilon$, $E' \rightarrow +TE'$. En general, si tenemos reglas de la forma $A \rightarrow A\alpha_i$ y otras $A \rightarrow \beta_j$, las podemos reemplazar por $A \rightarrow \beta_j A'$, $A' \rightarrow \alpha_i A'$, $A' \rightarrow \varepsilon$.

Ejemplo 3.21 Aplicando las técnicas descritas, la GLC del Ej. 3.5 se convierte en la siguiente, que puede parsearse con un AP determinístico que mira el siguiente carácter.

$$\begin{array}{ll}
 E & \rightarrow TE' \\
 E' & \rightarrow +TE' \\
 E' & \rightarrow \varepsilon \\
 T & \rightarrow FT' \\
 T' & \rightarrow *FT' \\
 T' & \rightarrow \varepsilon \\
 F & \rightarrow (E) \\
 F & \rightarrow N
 \end{array}
 \qquad
 \begin{array}{ll}
 N & \rightarrow DN' \\
 N' & \rightarrow N \\
 N' & \rightarrow \varepsilon \\
 D & \rightarrow 0 \\
 D & \rightarrow \dots \\
 D & \rightarrow 9
 \end{array}$$

Obsérvese que no es inmediato que el AP tipo $LL(1)$ que obtengamos de esta GLC será determinístico. Lo que complica las cosas son las reglas como $E' \rightarrow \varepsilon$. Tal como está, la idea es que, si tenemos E' en el tope de la pila y viene un $+$ en la entrada, debemos reemplazar E' por $+TE'$, mientras que si viene cualquier otra cosa, debemos eliminarla de la pila (o sea reemplazarla por ε). En esta gramática esto funciona. Existe un método general para verificar que la GLC obtenida funciona, y se ve en cursos de compiladores: la idea es calcular qué caracteres pueden seguir a E' en una cadena del lenguaje; si $+$ no puede seguirla, es seguro aplicar $E' \rightarrow \varepsilon$ cuando el siguiente carácter no sea $+$.

Algo interesante de estos parsers top-down es que permiten una implementación manual muy sencilla a partir de la GLC. Esto es lo que un programador hace intuitivamente cuando se enfrenta a un problema de parsing. A continuación haremos un ejemplo que sólo indica si la cadena pertenece al lenguaje o no, pero este parsing recursivo permite también realizar acciones de modo de por ejemplo evaluar la expresión o construir su árbol sintáctico. En un compilador, lo que hace el parsing es generar una representación intermedia del código para que después sea traducido al lenguaje de máquina.

Ejemplo 3.22 Se puede obtener casi automáticamente un parser recursivo asociado a la GLC del Ej. 3.21. *nextChar()* devuelve el siguiente carácter de la entrada, y *getChar()* lo consume.

```
ParseE
if  $\neg$  ParseT return false
if  $\neg$  ParseE' return false
return true
```

```
ParseE'
if nextChar() = + return ParseE'1
return ParseE'2
```

```
ParseE'1
getChar();
if  $\neg$  ParseT return false
if  $\neg$  ParseE' return false
return true
```

```
ParseE'2
return true
```

... **ParseT** y **ParseT'** muy similares

```
ParseF
if nextChar() = ( return ParseF1
return ParseF2
```

```
ParseF1
getChar();
if  $\neg$  ParseE return false
if nextChar()  $\neq$  ) return false
getChar();
return true
```

```
ParseF2
return ParseN
```

```
ParseN
if  $\neg$  ParseD return false
if  $\neg$  ParseN' return false
return true
```

```
ParseN'
if nextChar()  $\in$  {0...9} return ParseN'1
return ParseN'2
```

```
ParseN'1
return ParseN
```

```
ParseN'2
return true
```

```
ParseD
if nextChar() = 0 return ParseD0
...
if nextChar() = 9 return ParseD9
```

```
ParseD0
getChar()
return true
```

... **ParseD₁** a **ParseD₉** similares

Los procedimientos pueden simplificarse a mano significativamente, pero la intención es enfatizar cómo salen prácticamente en forma automática de la GLC. **ParseE** devolverá si pudo parsear la entrada o no, y consumirá lo que pudo parsear. Si devuelve *true* y consume la entrada correctamente, ésta es válida. Cada regla tiene su procedimiento asociado, y cada no terminal también. Por ejemplo **ParseE'** parsea un *E'*, y recurre a dos reglas posibles, **ParseE'₁** y **ParseE'₂**. Para elegir, se considera el siguiente carácter.

Las suposiciones sobre la correctitud de la GLC para un parsing determinístico usando el siguiente carácter se han trasladado al código. $\text{Parse}E'$, por ejemplo, supone que se le puede dar la prioridad a $\text{Parse}E'_1$, pues verifica directamente el primer carácter, y si no funciona sigue con $\text{Parse}E'_2$. Esto no funcionaría si el $+$ pudiera eventualmente aparecer siguiendo E' en la derivación.

Parsing Bottom-Up: Lenguajes $\text{LR}(k)$

El parsing top-down exige que, finalmente, seamos capaces de determinar qué regla aplicar a partir de ver el siguiente carácter (o los siguientes k caracteres). El parsing $\text{LR}(k)$ es más flexible. La idea esta vez es construir el árbol de parsing de abajo hacia arriba.

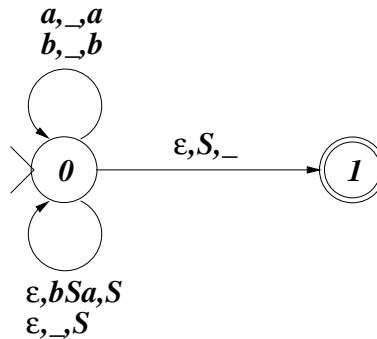
La idea de este parsing es que la pila contiene lo que el parser ha visto de la entrada, no lo que espera ver. Lo que se ha visto se expresa como una secuencia de terminales y no terminales. En cada momento se puede elegir entre apilar la primera letra de la entrada (con lo que ya la “hemos visto”) o identificar la parte derecha de una regla en la pila y reemplazarla por la parte izquierda. Al final, si hemos visto toda la entrada y en la pila está el símbolo inicial, la cadena pertenece al lenguaje.

Definición 3.20 El autómata de pila LR (APLR) de una GLC $G = (V, \Sigma, R, S)$ se define como $M = (K, \Sigma, \Gamma, \Delta, s, F)$ donde $K = \{s, f\}$, $\Gamma = V \cup \Sigma$, $F = \{f\}$ y

$$\begin{aligned} \Delta &= \{((s, \varepsilon, S), (f, \varepsilon))\} \\ &\cup \{((s, a, \varepsilon), (s, a), a \in \Sigma)\} \\ &\cup \{((s, \varepsilon, z^R), (s, A), A \longrightarrow z \in R)\} \end{aligned}$$

Notar que el APLR es una alternativa a la construcción que hicimos en la Def. 3.13, más adecuada al parsing $\text{LR}(k)$. Los generadores profesionales de parsers usan el mecanismo $\text{LR}(k)$, pues es más potente.

Ejemplo 3.23 Dibujemos el APLR para la GLC del Ej. 3.14. Notar que no es determinístico.



Definición 3.21 *Los lenguajes LC que se pueden reconocer con la construcción descrita arriba se llaman LR(k), donde k es la cantidad de caracteres que deben mirarse en la entrada para decidir los conflictos.*

E, E	$\varepsilon,)E, (F$	$\varepsilon, 0, D$
ε, N, F	\dots	
T, T	ε, D, N	$\varepsilon, 9, D.$
	ε, ND, D	

$$\begin{array}{l}
(0, 2 + 3 * 5, \varepsilon) \vdash (0, +3 * 5, 2) \vdash (0, +3 * 5, D) \vdash (0, +3 * 5, N) \vdash (0, +3 * 5, F) \\
\vdash (0, +3 * 5, T) \vdash (0, +3 * 5, E) \vdash (0, 3 * 5, +E) \vdash (0, *5, 3 + E) \\
\vdash (0, *5, D + E) \vdash (0, *5, N + E) \vdash (0, *5, F + E) \vdash (0, *5, T + E) \\
\vdash (0, 5, *T + E) \vdash (0, \varepsilon, 5 * T + E) \vdash (0, \varepsilon, D * T + E) \vdash (0, \varepsilon, N * T + E) \\
\vdash (0, \varepsilon, F * T + E) \vdash (0, \varepsilon, T + E) \vdash (0, \varepsilon, E) \vdash (1, \varepsilon, \varepsilon)
\end{array}$$

El ejemplo muestra la relevancia de los conflictos. Por ejemplo, si en el segundo paso, en vez de reducir $D \rightarrow 2$ (aplicando la regla $((0, \varepsilon, 2), (0, D))$) hubiéramos apilado el $+$, nunca habríamos logrado reducir toda la cadena a E . Similarmente, en el paso 6, si hubiéramos apilado el $+$ en vez de favorecer la regla $E \rightarrow T$, habríamos fracasado. En cambio, en el paso 13, debemos apilar $*$ en vez de usar la regla $E \rightarrow T$. Esto indica que, por ejemplo, en el caso del conflicto *shift/reduce* de la regla $((0, +, \varepsilon), (0, +))$ con $((0, \varepsilon, T), (0, E))$ debe favorecerse el reduce, mientras que en el conflicto de $((0, *, \varepsilon), (0, *))$ con $((0, \varepsilon, T), (0, E))$ debe favorecerse el shift. Esto está relacionado con la precedencia de los operadores. El caso

de conflictos *reduce/reduce* suele resolverse priorizando la parte derecha más larga, lo cual casi siempre es lo correcto. Los generadores de parsers permiten indicar cómo resolver cada conflicto posible en la gramática, los cuales pueden ser precalculados.

3.10 Ejercicios

Gramáticas Libres del Contexto

1. Considere la gramática $G = (\{S, A\}, \{a, b\}, R, S)$, con $R = \{S \rightarrow AA, A \rightarrow AAA, A \rightarrow a, A \rightarrow bA, A \rightarrow Ab\}$.
 - (a) ¿Qué cadenas de $\mathcal{L}(G)$ se pueden generar con derivaciones de cuatro pasos o menos?
 - (b) Dé al menos cuatro derivaciones distintas para *babbab* y dibuje los árboles de derivación correspondientes (los cuales podrían no ser distintos).
2. Sea la gramática $G = (\{S, A\}, \{a, b\}, R, S)$, con $R = \{S \rightarrow aAa, S \rightarrow bAb, S \rightarrow \varepsilon, A \rightarrow SS\}$.
 - (a) Dé una derivación para *baabbb* y dibuje el árbol de derivación.
 - (b) Describa $\mathcal{L}(G)$ con palabras.
3. Considere el alfabeto $\Sigma = \{a, b, (,), |, \star, \Phi\}$. Construya una GLC que genere todas las expresiones regulares válidas sobre $\{a, b\}$.
4. Sea $G = (\{S\}, \{a, b\}, R, S)$, con $R = \{S \rightarrow aSa, S \rightarrow aSb, S \rightarrow bSa, S \rightarrow bSb, S \rightarrow \varepsilon\}$. Muestre que $\mathcal{L}(G)$ es regular.
5. Construya GLCs para los siguientes lenguajes
 - (a) $\{a^m b^n, m \geq n\}$.
 - (b) $\{a^m b^n c^p d^q, m + n = p + q\}$.
 - (c) $\{uawb, u, w \in \{a, b\}^*, |u| = |w|\}$

Autómatas de Pila

1. Construya autómatas que reconozcan los lenguajes del ejercicio 5 de la sección anterior. Hágalo directamente, no transformando la gramática.
2. Dado el autómata $M = (\{s, f\}, \{a, b\}, \{a\}, \Delta, s, \{f\})$, con $\Delta = \{((s, a, \varepsilon), (s, a)), ((s, b, \varepsilon), (s, a)), ((s, a, \varepsilon), (f, \varepsilon)), ((f, a, a), (f, \varepsilon)), ((f, b, a), (f, \varepsilon))\}$.

- (a) Dé todas las posibles secuencias de transiciones para aba .
 - (b) Muestre que $aba, aa, abb \notin \mathcal{L}(M)$, pero $baa, bab, baaaa \in \mathcal{L}(M)$.
 - (c) Describa $\mathcal{L}(M)$ en palabras.
3. Construya autómatas que reconozcan los siguientes lenguajes
- (a) El lenguaje generado por $G = (\{S\}, \{[,], (,)\}, R, S)$, con $R = \{S \rightarrow \varepsilon, S \rightarrow SS, S \rightarrow [S], S \rightarrow (S)\}$.
 - (b) $\{a^m b^n, m \leq n \leq 2m\}$
 - (c) $\{w \in \{a, b\}^*, w = w^R\}$.

Gramáticas y Autómatas

1. Considere la GLC $G = (\{S, A, B\}, \{a, b\}, R, S)$, con $R = \{S \rightarrow abA, S \rightarrow B, S \rightarrow baB, S \rightarrow \varepsilon, A \rightarrow bS, B \rightarrow aS, A \rightarrow b\}$. Construya el autómata asociado.
2. Repita el ejercicio 1 de la parte anterior, esta vez obteniendo los autómatas directamente de la gramática. Compárelos.
3. Considere nuevamente el ejercicio 1 de la parte anterior. Obtenga, usando el algoritmo visto, la gramática que corresponde a cada autómata que usted generó manualmente. Compárelas con las gramáticas originales de las que partió cuando hizo ese ejercicio.

Lenguajes Libres de Contexto

1. Use las propiedades de clausura (y otros ejercicios ya hechos) para probar que los siguientes lenguajes son LC.
 - (a) $\{a^m b^n, m \neq n\}$
 - (b) $\{a^m b^n c^p d^q, n = q \vee m \leq p \vee m + n = p + q\}$
 - (c) $\{a^m b^n c^p, m = n \vee n = p \vee m = p\}$
 - (d) $\{a^m b^n c^p, m \neq n \vee n \neq p \vee m \neq p\}$
2. Use el Teorema de Bombeo para probar que los siguientes lenguajes no son LC.
 - (a) $\{a^p, p \text{ es primo}\}$.
 - (b) $\{a^{n^2}, n \geq 0\}$.
 - (c) $\{www, w \in \{a, b\}^*\}$.
 - (d) $\{a^m b^n c^p, m = n \wedge n = p \wedge m = p\}$ (¿lo reconoce?)
3. Sean M_1, M_2 autómatas de pila. Construya directamente autómatas para $\mathcal{L}(M_1) \cup \mathcal{L}(M_2)$, $\mathcal{L}(M_1)\mathcal{L}(M_2)$ y $\mathcal{L}(M_1)^*$.

3.11 Preguntas de Controles

A continuación se muestran algunos ejercicios de controles de años pasados, para dar una idea de lo que se puede esperar en los próximos. Hemos omitido (i) (casi) repeticiones, (ii) cosas que ahora no se ven, (iii) cosas que ahora se dan como parte de la materia y/o están en los ejercicios anteriores. Por lo mismo a veces los ejercicios se han alterado un poco o se presenta sólo parte de ellos, o se mezclan versiones de ejercicios de distintos años para que no sea repetitivo.

C1 1996, 1997, 2005 Responda verdadero o falso y justifique brevemente (máximo 5 líneas). Una respuesta sin justificación no vale *nada* aunque esté correcta, una respuesta incorrecta puede tener algún valor por la justificación.

- a) Un lenguaje regular también es LC, y además determinístico.
- b) Los APs *determinísticos* no son más potentes que los autómatas finitos. Los que son más potentes son los no determinísticos.
- c) Si restringimos el tamaño máximo de la pila de los autómatas de pila a 100, éstos aun pueden reconocer ciertos lenguajes no regulares, como $\{a^n b^n, n \leq 100\}$.
- d) Si un autómata de pila pudiera tener dos pilas en vez de una sería más poderoso.
- e) El complemento de un lenguaje LC no regular tampoco es regular.
- f) Si L es LC, L^R también lo es.
- g) Un autómata de pila puede determinar si un programa escrito en C será aceptado por el compilador (si no sabe C reemplácelo por Pascal, Turing o Java).
- h) Todo subconjunto de un lenguaje LC es LC.
- i) Los prefijos de un lenguaje LC forman un lenguaje LC.

Hemos unido ejercicios similares de esos años.

C1 1996 En la siguiente secuencia, si no logra hacer un ítem puede suponer que lo ha resuelto y usar el resultado para los siguientes.

- a) Intente aplicar el Teorema de Bombeo *sin la restricción* $|uyv| \leq N$ para demostrar que el lenguaje $\{ww, w \in \{a, b\}^*\}$ no es LC. ¿Por qué no funciona?
- d) ¿En qué falla el Teorema de Bombeo si quiere aplicarlo a $\{ww^R, w \in \{a, b\}^*\}$? ¿Es posible reforzar el Teorema para probar que ese conjunto no es LC?

C1 1997 La historia de los movimientos de una cuenta corriente se puede ver como una secuencia de depósitos y extracciones, donde nunca hay saldo negativo. Considere que cada depósito es un entero entre 1 y k (fijo) y cada extracción un entero entre $-k$ y -1 . Se supone que la cuenta empieza con saldo cero. El lenguaje de los movimientos aceptables es entonces un subconjunto de $\{-k..k\}^*$, donde nunca el saldo es negativo.

- a) Dibuje un autómata de pila para este lenguaje. En beneficio suyo y pensando en la parte b), hágalo de un sólo estado. Describalo formalmente como una tupla.
- b) Transforme el autómata anterior a una GLC con el método visto, para el caso $k = 1$. ¿Qué problema se le presentaría para $k > 1$?
- c) Modifique el autómata de modo que permita un sobregiro de n unidades pero al final el saldo no pueda ser negativo.

Ex 1997 Sea $M = (K, \Sigma, \Gamma, \Delta, s, F)$ un autómata de pila. Se definió que el lenguaje aceptado por M es $\mathcal{L}(M) = \{w \in \Sigma^*, (s, w, \varepsilon) \vdash_M^* (f, \varepsilon, \varepsilon)\}$ (donde $f \in F$). Definimos ahora a partir de M otros dos lenguajes: $\mathcal{L}_1(M) = \{w \in \Sigma^*, (s, w, \varepsilon) \vdash_M^* (f, \varepsilon, \alpha)\}$ (donde $f \in F$ y $\alpha \in \Gamma^*$), y $\mathcal{L}_2(M) = \{w \in \Sigma^*, (s, w, \varepsilon) \vdash_M^* (q, \varepsilon, \varepsilon)\}$ (donde $q \in K$).

- a) Describa en palabras lo que significan \mathcal{L}_1 y \mathcal{L}_2 .
- b) Demuestre que todo autómata de pila M se puede modificar (obteniendo un M') de modo que $\mathcal{L}(M') = \mathcal{L}_1(M)$, y viceversa.
- c) Lo mismo que b) para \mathcal{L}_2 .

C1 1998 Sea P un pasillo estrecho sin salida diseñado para ser raptados por extraterrestres. La gente entra al pasillo, permanece un cierto tiempo, y finalmente o bien es raptada por los extraterrestres o bien sale (desilusionada) por donde entró (note que el último que entra es el primero que sale, si es que sale). En cualquier momento pueden entrar nuevas personas o salir otras, pero se debe respetar el orden impuesto por el pasillo. Dada una cadena formada por una secuencia de entradas y salidas de personas, se desea determinar si es correcta o no. Las entradas se indican como $E i$, es decir el carácter E que indica la entrada y la i que identifica a la persona. Las salidas se indican como $S j$. En principio i y j deberían ser cadenas de caracteres, pero simplificaremos y diremos que son caracteres. Por ejemplo, $E1E2E3S2E4S1$ es correcta (3 y 4 fueron raptados), pero $E1E2E3S2E4S3$ es incorrecta (pues 2 entró antes que 3 y salió antes).

- a) Dibuje un autómata de pila que reconozca este lenguaje.
- b) Dé una GLC que genere este lenguaje.

C2 1998 Use propiedades de clausura para demostrar que el siguiente lenguaje es LC

$$L = \{w_1 w_2 w_3 w_4, (w_3 = w_1^R \vee w_2 = w_4^R) \wedge |w_1 w_2 w_3 w_4| \text{ par}\}$$

Ex 1998, C1 2003 Use el Teorema del Bombeo para probar que los siguientes lenguajes no son LC:

- a) $L = \{a^n b^m a^n, m < n\}$

$$b) L = \{a^n b^m c^r d^s, 0 \leq n \leq m \leq r, 0 \leq s\}$$

C2 1999 Suponga que tiene una calculadora lógica que usa notación polaca (tipo HP). Es decir, primero se ingresan los dos operandos y luego la operación. Considerando los valores V y F y las operaciones $+$ (or), $*$ (and) y $-$ (complemento), especifique un autómata de pila que reconoce una secuencia válida de operaciones y queda en un estado final si el último valor es V .

Por ejemplo, para calcular $(A \text{ or } B) \text{ and } C$ y los valores de A , B y C son V , F y V , respectivamente; usamos la secuencia $VF + V*$.

C2 1999 Escoja una de las dos siguientes preguntas:

- Demuestre que el lenguaje $L = \{a^i b^j c^i d^j, i, j \geq 0\}$ no es LC.
- Si L es LC y R es regular entonces ¿ $L - R$ es LC? ¿Qué pasa con $R - L$? Justifique su respuesta.

Ex 1999 Escriba una GLC G que genere todas las posibles GLC sobre el alfabeto $\{a, b\}$ y que usen los no terminales S , A , y B . Cada posible GLC es una secuencia de producciones separadas por comas, entre paréntesis y usando el símbolo igual para indicar una producción. Por ejemplo, una palabra generada por G es $(S = ASB, A = a, B = b, S = \varepsilon)$. Indique claramente cuales son los no terminales y terminales de G .

C1 2000 (a) Un *autómata de 2 pilas* es similar a un autómata de pila, pero puede manejar dos pilas a la vez (es decir poner condiciones sobre ambas pilas y modificarlas simultáneamente). Muestre que con un autómata de 2 pilas puede reconocer el lenguaje $a^n b^n c^n$. ¿Y $a^n b^n c^n d^n$? ¿Y $a^n b^n c^n d^n e^n$?

- Se tiene una GLC G_1 que define un lenguaje L_1 , y una expresión regular E_2 que define un lenguaje L_2 . ¿Qué pasos seguiría para generar un autómata que reconozca las cadenas de L_1 que no se puedan expresar como una cadena $w_1 w_2^R$, donde w_1 y w_2 pertenecen a L_2 ?

C1 2001 Considere un proceso donde Pikachu intenta subir los pisos de un edificio, para lo cual recibe la energía necesaria en una cadena de entrada. La cadena contiene al comienzo una cierta cantidad de letras “ I ”, tantas como pisos tiene el edificio. El resto de la cadena está formada por signos “ E ” y “ $-$ ”. Cada símbolo E en la entrada le da a Pikachu una unidad de energía, y cada tres unidades de energía Pikachu puede subir un piso más. Por otro lado, cada “ $-$ ” es un intervalo de tiempo sin recibir energía. Si pasan cuatro intervalos de tiempo sin recibir energía, Pikachu pierde una unidad E de energía. Si se recibe una E antes de pasar cuatro intervalos de tiempo de “ $-$ ”s, no se pierde nada (es decir, los últimos “ $-$ ”s se ignoran). Si recibe cuatro “ $-$ ”s cuando no tiene nada de energía almacenada, Pikachu muere.

Diseñe un autómata de pila que acepte las cadenas de la forma xy donde $x = I^n$ e $y \in \{E, -\}^*$, tal que y le da a Pikachu energía suficiente para subir un edificio de n pisos (puede sobrar energía).

Ex 2002 Demuestre que para toda GLC G , existe una GLC G' en forma normal de Chomsky que genera el mismo lenguaje.

C1 2004 Demuestre que si L es LC, entonces las cadenas de L cuyo largo no es múltiplo de 5 pero sí de 3, es LC.

C1 2004 Se tiene la siguiente GLC: $E \longrightarrow E \wedge E \mid E \vee E \mid (E) \mid 0 \mid 1$.

1. Utilice el método básico para obtener un autómata de pila que reconozca $\mathcal{L}(E)$.
2. Repita el procedimiento, esta vez utilizando el método visto para parsing bottom-up.
3. Modifique la GLC y/o alguno de los APs para obtener un autómata de pila determinístico. El “ \wedge ” tiene mayor precedencia que el “ \vee ”.

C1 2005 Para cada uno de los siguientes lenguajes, demuestre que o bien es regular, o bien LC y no regular, o bien no es LC.

1. $\{a^n b^m, n \leq m \leq 2n\}$
2. $\{w \in \{a, b\}^*, w \text{ tiene el doble de } a\text{'s que } b\text{'s}\}$
3. $\{a^{pn+q}, n \geq 0\}$, para cualquier $p, q \geq 0$.
4. $\{a^n b^m c^{2(n+m)}, m, n \geq 0\}$.

3.12 Proyectos

1. Investigue sobre la relación entre GLCs y las DTDs utilizadas en XML.
2. Investigue más sobre determinismo y parsing, lenguajes $LL(k)$ y $LR(k)$. Hay algo de material en el mismo capítulo indicado del libro, pero mucho más en un libro de compiladores, como [ASU86, cap 4] o [AU72, cap 5]. En estos libros puede encontrar material sobre otras formas de parsing.
3. Investigue sobre herramientas para generar parsers automáticamente. En **C/Unix** se llaman **lex** y **yacc**, pero existen para otros sistemas operativos y lenguajes. Construya un parser de algún lenguaje de programación pequeño y luego conviértalo en intérprete.
4. Programe el ciclo de conversión $GLC \rightarrow AP \rightarrow GLC$.
5. Programe la conversión a Forma Normal de Chomsky y el parser de tiempo $O(n^3)$ asociado. Esto se ve, por ejemplo, en [HMU01 sec 7.4].

Referencias

- [ASU86] A. Aho, R. Sethi, J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
- [AU72] A. Aho, J. Ullman. *The Theory of Parsing, Translations, and Compiling*. Volume I: Parsing. Prentice-Hall, 1972.
- [HMU01] J. Hopcroft, R. Motwani, J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. 2nd Edition. Pearson Education, 2001.
- [LP81] H. Lewis, C. Papadimitriou. *Elements of the Theory of Computation*. Prentice-Hall, 1981. Existe una segunda edición, bastante parecida, de 1998.

Capítulo 4

Máquinas de Turing y la Tesis de Church

[LP81, cap 4 y 5]

La computabilidad se puede estudiar usando diversos formalismos, todos ellos equivalentes. En este curso nos hemos decidido por las Máquinas de Turing por dos razones: (i) se parecen a los autómatas de distinto tipo que venimos viendo de antes, (ii) son el modelo canónico para estudiar NP-completitud, que es el último capítulo del curso.

En este capítulo nos centraremos sólo en el formalismo, y cómo utilizarlo y extenderlo para distintos propósitos, y en el siguiente lo utilizaremos para obtener los resultados de computabilidad. Recomendamos al lector el uso de un simulador de MTs (que usa la notación modular descrita en la Sección 4.3) que permite dibujarlas y hacerlas funcionar. Se llama *Java Turing Visual (JTV)* y está disponible en <http://www.dcc.uchile.cl/jtv>.¹

Al final del capítulo intentaremos convencer al lector de la Tesis de Church, que dice que las Máquinas de Turing son equivalentes a cualquier modelo de computación factible de construir. Asimismo, veremos las gramáticas dependientes del contexto, que extienden las GLCs, para completar nuestro esquema de reconocedores/generadores de lenguajes.

4.1 La Máquina de Turing (MT)

[LP81, sec 4.1]

La Máquina de Turing es un mecanismo de computación notoriamente primitivo, y sin embargo (como se verá más adelante) permite llevar a cabo cualquier cómputo. Informalmente, una MT opera con un *cabezal* dispuesto sobre una *cinta* que tiene comienzo pero no tiene fin, extendiéndose hacia la derecha tanto como se quiera. Cada celda de la cinta almacena un carácter, y cuando se examina un carácter de la cinta nunca visto, se

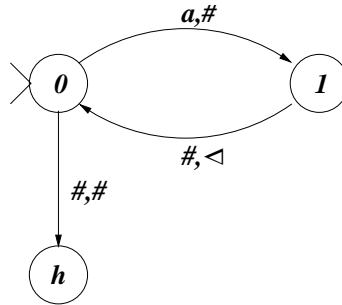
¹Esta herramienta se desarrolló en el DCC, por mi alumno Marco Mora Godoy, en su Memoria de Ingeniería.

supone que éste contiene un *blanco* ($\#$). Como los autómatas, la MT está en un *estado*, de un conjunto finito de posibilidades. En cada paso, la MT lee el carácter que tiene bajo el cabezal y, según ese carácter y el estado en que está, pasa a un nuevo estado y lleva a cabo una *acción* sobre la cinta: cambiar el carácter que leyó por uno nuevo (en la misma celda donde tiene el cabezal), o mover el cabezal hacia la izquierda o hacia la derecha.

Como puede *escribir* la cinta, la MT no tendrá estados finales o no finales, pues puede dejar escrita la respuesta (sí o no) al detenerse. Existe simplemente un estado especial, denominado h , al llegar al cual la computación de la MT se detiene. La computación también se interrumpe (sin llegar al estado h) si la MT trata de moverse hacia la izquierda de la primera celda de la cinta. En tal caso decimos que la MT se “cuelga”, pero *no* que se detiene o que la computación termina.

Tal como con autómatas, dibujaremos las MTs como grafos donde los nodos son los estados y las transiciones están rotuladas. Una transición de p a q rotulada a, b significa que si la MT está en el estado p y hay una letra a bajo el cabezal, entonces pasa al estado q y realiza la acción b . La acción de escribir una letra $a \in \Sigma$ se denota simplemente a . La de moverse a la izquierda o derecha se denota \triangleleft y \triangleright , respectivamente. La acción de escribir un blanco ($\#$) se llama también *borrar* la celda.

Ejemplo 4.1 Dibujemos una MT que, una vez arrancada, borre todas las a 's que hay desde el cabezal hacia atrás, hasta encontrar otro $\#$.



Notar que si se arranca esta MT con una cinta que tenga puras a 's desde el comienzo de la cinta hasta el cabezal, la máquina se colgará. Notar también que no decimos qué hacer si estamos en el estado 1 y leemos $\#$. Formalmente siempre debemos decir qué hacer en cada estado ante cada carácter (como con un AFD), pero nos permitiremos no dibujar los casos imposibles, como el que hemos omitido. Finalmente, véase que la transición $\#, \#$ es una forma de no hacer nada al pasar a h .

Definamos formalmente una MT y su operación.

Definición 4.1 Una Máquina de Turing (MT) es una tupla $M = (K, \Sigma, \delta, s)$, donde

- K es un conjunto finito de estados, $h \notin K$.
- Σ es un alfabeto finito, $\# \in \Sigma$.

- $s \in K$ es el estado inicial
- $\delta : K \times \Sigma \longrightarrow (K \cup \{h\}) \times (\Sigma \cup \{\triangleleft, \triangleright\})$, $\triangleleft, \triangleright \notin \Sigma$, es la función de transición.

Ejemplo 4.2 La MT del Ej. 4.1 se escribe formalmente como $M = (K, \Sigma, \delta, s)$ con $K = \{0, 1\}$, $\Sigma = \{a, \#\}$, $s = 0$, y

δ	0	1
a	1, #	1, a
$\#$	$h, \#$	0, \triangleleft

Notar que hemos debido completar de alguna forma la celda $\delta(1, a) = (1, a)$, que nunca puede aplicarse dada la forma de la MT. Véase que, en un caso así, la MT funcionaría eternamente sin detenerse (ni colgarse).

Definamos ahora lo que es una configuración. La información que necesitamos para poder completar una computación de una MT es: su estado actual, el contenido de la cinta, y la posición del cabezal. Los dos últimos elementos se expresan particionando la cinta en tres partes: la cadena que precede al cabezal (ε si estamos al inicio de la cinta), el carácter sobre el que está el cabezal, y la cadena a la derecha del cabezal. Como ésta es infinita, esta cadena se indica sólo hasta la última posición distinta de $\#$. Se fuerza en la definición, por tanto, a que esta cadena no pueda terminar en $\#$.

Definición 4.2 Una configuración de una MT $M = (K, \Sigma, \delta, s)$ es un elemento de $\mathcal{C}_M = (K \cup \{h\}) \times \Sigma^* \times \Sigma \times (\Sigma^* - (\Sigma^* \circ \{\#\}))$. Una configuración (q, u, a, v) se escribirá también $(q, u\underline{a}v)$, e incluso simplemente $u\underline{a}v$ cuando el estado es irrelevante. Una configuración de la forma (h, u, a, v) se llama configuración detenida.

El funcionamiento de la MT se describe mediante cómo nos lleva de una configuración a otra.

Definición 4.3 La relación lleva en un paso para una MT $M = (K, \Sigma, \delta, s)$, $\vdash_M \subseteq \mathcal{C}_M \times \mathcal{C}_M$, se define como: $(q, u, a, v) \vdash_M (q', u', a', v')$ si $q \in K$, $\delta(q, a) = (q', b)$, y

1. Si $b \in \Sigma$ (la acción es escribir el carácter b en la cinta), entonces $u' = u$, $v' = v$, $a' = b$.
2. Si $b = \triangleleft$ (la acción es moverse a la izquierda), entonces $u'a' = u$ y (i) si $av \neq \#$, entonces $v' = av$, de otro modo $v' = \varepsilon$.
3. Si $b = \triangleright$ (la acción es moverse a la derecha), entonces $u' = ua$ y (i) si $v \neq \varepsilon$ entonces $a'v' = v$, de otro modo $a'v' = \#$.

Observación 4.1 Nótese lo que ocurre si la MT quiere moverse hacia la izquierda estando en la primera celda de la cinta: la ecuación $u'a' = u = \varepsilon$ no puede satisfacerse y la configuración no lleva a ninguna otra, pese a no ser una configuración detenida. Entonces la computación no avanza, pero no ha terminado (está “colgada”).

Como siempre, diremos \vdash en vez de \vdash_M cuando M sea evidente, y \vdash^* (\vdash_M^*) será la clausura reflexiva y transitiva de \vdash (\vdash_M), “lleva en cero o más pasos”.

Definición 4.4 Una computación de M de n pasos es una secuencia de configuraciones $C_0 \vdash_M C_1 \vdash_M \dots \vdash_M C_n$.

Ejemplo 4.3 Mostremos las configuraciones por las que pasa la MT del Ej. 4.1 al ser arrancada desde la configuración $\#aaa\underline{a}$:

$$\begin{aligned} (0, \#aaa\underline{a}) &\vdash (1, \#aaa\underline{\#}) \vdash (0, \#aa\underline{a}) \vdash (1, \#aa\underline{\#}) \vdash (0, \#a\underline{a}) \\ &\vdash (1, \#a\underline{\#}) \vdash (0, \#\underline{a}) \vdash (1, \#\underline{\#}) \vdash (0, \underline{\#}) \vdash (h, \underline{\#}) \end{aligned}$$

a partir de la cual ya no habrá más pasos porque es una configuración detenida. En cambio, si la arrancamos sobre $aa\underline{a}$ la MT se colgará:

$$(0, aa\underline{a}) \vdash (1, aa\underline{\#}) \vdash (0, a\underline{a}) \vdash (1, a\underline{\#}) \vdash (0, \underline{a}) \vdash (1, \underline{\#})$$

y de aquí ya no se moverá a otra configuración.

4.2 Protocolos para Usar MTs

[LP81, sec 4.2]

Puede verse que las MTs son mecanismos mucho más versátiles que los autómatas que hemos visto antes, que no tenían otro propósito posible que leer una cadena de principio a fin y terminar o no en un estado final (con pila vacía o no). Una MT *transforma* el contenido de la cinta, por lo que puede utilizarse, por ejemplo, para calcular funciones de cadenas en cadenas.

Definición 4.5 Sea $f : \Sigma_0^* \longrightarrow \Sigma_1^*$, donde $\# \notin \Sigma_0 \cup \Sigma_1$. Decimos que una MT $M = (K, \Sigma, \delta, s)$ computa f si

$$\forall w \in \Sigma_0^*, (s, \#w\underline{\#}) \vdash_M^* (h, \#f(w)\underline{\#}).$$

La definición se extiende al caso de funciones de múltiples argumentos, $f(w_1, w_2, \dots, w_k)$, donde la MT debe operar de la siguiente forma:

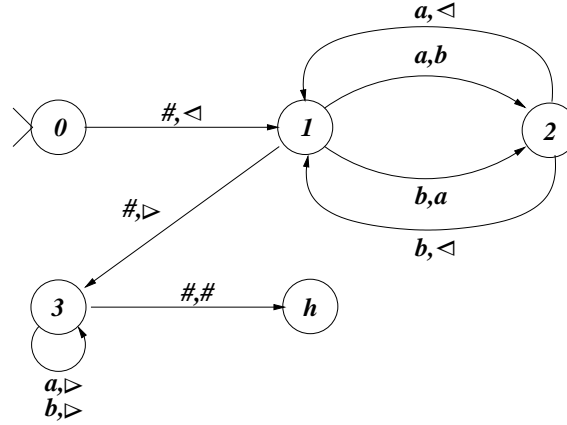
$$(s, \#w_1\#w_2\# \dots \#w_k\underline{\#}) \vdash_M^* (h, \#f(w_1, w_2, \dots, w_k)\underline{\#}).$$

Una función para la cual existe una MT que la computa se dice Turing-computable o simplemente computable.

Esto nos indica el *protocolo* con el cual esperamos usar una MT para calcular funciones: el dominio e imagen de la función no permiten el carácter #, ya que éste se usa para delimitar el argumento y la respuesta. El cabezal empieza y termina al final de la cadena, dejando limpio el resto de la cinta.

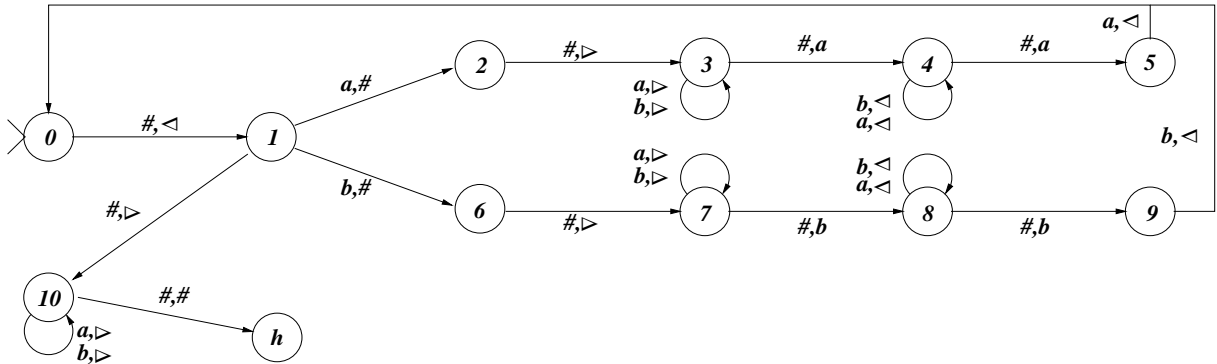
Observación 4.2 Una observación muy importante sobre la Def. 4.5 es que M no se cuelga frente a ninguna entrada, sino que siempre llega a h . Esto significa que jamás intenta moverse hacia la izquierda del primer #. Por lo tanto, si sabemos que M calcula f , podemos con confianza aplicarla sobre una cinta de la forma $\#x\#y\#w\#$ y saber que terminará y dejará la cinta en la forma $\#x\#y\#f(w)\#$ sin alterar x o y .

Ejemplo 4.4 Una MT que calcula $f(w) = \overline{w}$ (es decir, cambiar las a 's por b 's y viceversa en $w \in \{a, b\}^*$), es la que sigue:



Nótese que no se especifica qué hacer si, al ser arrancada, el cabezal está sobre un carácter distinto de #, ni en general qué ocurre si la cinta no sigue el protocolo establecido, pues ello no afecta el hecho de que esta MT calcule f según la definición.

Ejemplo 4.5 Una MT que calcula $f(w) = ww^R$ (donde w^R es w escrita al revés), es la que sigue.

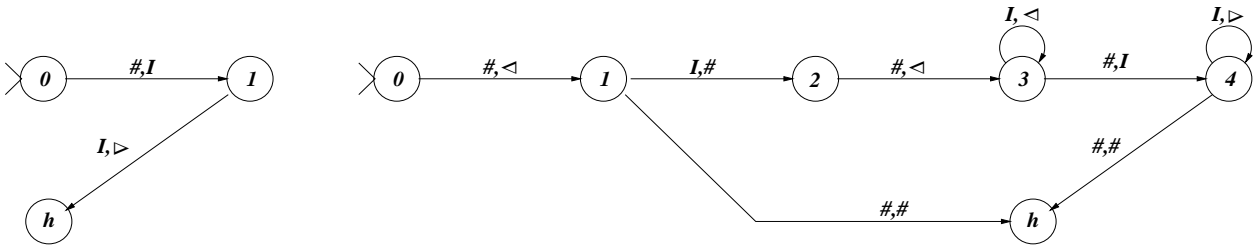


Es un ejercicio interesante derivar $f(w) = ww$ a partir de este ejemplo.

Es posible también usar este formalismo para computar funciones de números naturales, mediante representar n con la cadena I^n (lo que es casi notación unaria).

Definición 4.6 Sea $f : \mathbb{N} \rightarrow \mathbb{N}$. Decimos que una MT M computa f si M computa (según la Def. 4.5) $g : \{I\}^* \rightarrow \{I\}^*$ dada por $g(I^n) = I^{f(n)}$. La definición se extiende similarmente a funciones de varias variables y se puede hablar de funciones Turing-computables entre los números naturales.

Ejemplo 4.6 Las MTs que siguen calculan $f(n) = n + 1$ (izquierda) y $f(n, m) = n + m$ (derecha).



Verifíquese que la MT de la suma funciona también para los casos $f(n, 0)$ (cinta $\#I^n\#\underline{\#}$) y $f(0, m)$ (cinta $\#\underline{\#}I^m\#\underline{\#}$).

Hemos visto cómo calcular funciones usando MTs. Volvamos ahora al plan original de utilizarlas para reconocer lenguajes. Reconocer un lenguaje es, esencialmente, responder “sí” o “no” frente a una cadena, dependiendo que esté o no en el lenguaje.

Definición 4.7 Una MT decide un lenguaje L si calcula la función $f_L : \Sigma^* \rightarrow \{\mathbf{S}, \mathbf{N}\}$, definida como $f_L(w) = \mathbf{S} \Leftrightarrow w \in L$ (y si no, $f_L(w) = \mathbf{N}$). Si existe tal MT, decimos que L es Turing-decidible o simplemente decidible.

Notar que la definición anterior es un caso particular de calcular funciones donde $\Sigma_0 = \Sigma$ y $\Sigma_1 = \{\mathbf{S}, \mathbf{N}\}$ (pero f_L sólo retornará cadenas de largo 1 de ese alfabeto).

Existe una noción más débil que la de decidir un lenguaje, que será esencial en el próximo capítulo. Imaginemos que nos piden que determinemos si una cierta proposición es demostrable a partir de un cierto conjunto de axiomas. Podemos probar, disciplinadamente, todas las demostraciones de largo creciente. Si la proposición es demostrable, algún día daremos con su demostración, pero si no... nunca lo podremos saber. *Sí, podríamos tratar de demostrar su negación en paralelo, pero en todo sistema de axiomas suficientemente potente existen proposiciones indemostrables tanto como su negación, recordar por ejemplo la hipótesis del continuo (Def. 1.16).*

Definición 4.8 Una MT $M = (K, \Sigma, \delta, s)$ acepta un lenguaje L si se detiene exactamente frente a las cadenas de L , es decir $(s, \#w\#) \vdash_M^* (h, u\underline{v}) \Leftrightarrow w \in L$. Si existe tal MT, decimos que L es Turing-aceptable o simplemente aceptable.

Observación 4.3 Es fácil ver que todo lenguaje decidible es aceptable, pero la inversa no se ve tan simple. Esto es el tema central del próximo capítulo.

4.3 Notación Modular

[LP81, sec 4.3 y 4.4]

No llegaremos muy lejos si insistimos en usar la notación aparatosa de MTs vista hasta ahora. Necesitaremos MTs mucho más potentes para enfrentar el próximo capítulo (y para convencernos de que una MT es equivalente a un computador!). En esta sección definiremos una notación para MTs que permite expresarlas en forma mucho más sucinta y, lo que es muy importante, poder componer MTs para formar otras.

En la *notación modular de MTs* una MT se verá como un grafo, donde los nodos serán *acciones* y las aristas *condiciones*. En cada nodo se podrá escribir una secuencia de acciones, que se ejecutan al llegar al nodo. Luego de ejecutarlas, se consideran las aristas que salen del nodo. Estas son, en principio, flechas rotuladas con símbolos de Σ . Si la flecha que sale del nodo está rotulada con la letra que coincide con la que tenemos bajo el cabezal luego de ejecutar el nodo, entonces seguimos la flecha y llegamos a otro nodo. Nunca debe haber más de una flecha aplicable a cada nodo (hasta que lleguemos a la Sección 4.5). Permitiremos rotular las flechas con conjuntos de caracteres. Habrá un nodo inicial, donde la MT comienza a operar, y cuando de un nodo no haya otro nodo adonde ir, la MT se detendrá.

Las acciones son realmente MTs. Comenzaremos con $2 + |\Sigma|$ *acciones básicas*, que corresponden a las acciones que pueden escribirse en δ , y luego podremos usar cualquier MT que definamos como acción para componer otras.

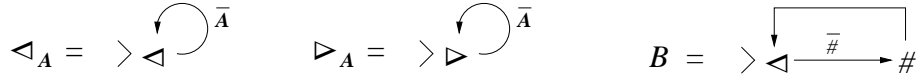
Definición 4.9 *Las acciones básicas de la notación modular de MTs son:*

- *Moverse hacia la izquierda (\triangleleft): Esta es una MT que, pase lo que pase, se mueve hacia la izquierda una casilla y se detiene. $\triangleleft = (\{s\}, \Sigma, \delta, s)$, donde $\forall a \in \Sigma, \delta(s, a) = (h, \triangleleft)$. (Notar que estamos sobrecargando el símbolo \triangleleft , pero no debería haber confusión.)*
- *Moverse hacia la derecha (\triangleright): Esta es una MT que, pase lo que pase, se mueve hacia la derecha una casilla y se detiene. $\triangleright = (\{s\}, \Sigma, \delta, s)$, donde $\forall a \in \Sigma, \delta(s, a) = (h, \triangleright)$.*
- *Escribir el símbolo $b \in \Sigma$ (b): Esta es una MT que, pase lo que pase, escribe b en la cinta y se detiene. $b = (\{s\}, \Sigma, \delta, s)$, donde $\forall a \in \Sigma, \delta(s, a) = (h, b)$. Nuevamente, estamos sobrecargando el símbolo $b \in \Sigma$ para denotar una MT.*

Observación 4.4 *Deberíamos definir formalmente este mecanismo y demostrar que es equivalente a las MTs. No lo haremos porque es bastante evidente que lo es, su definición formal es aparatosa, y finalmente podríamos vivir sin este formalismo, cuyo único objetivo es simplificar la vida. Se puede ver en el libro la demostración.*

Dos MTs sumamente útiles como acciones son \triangleleft_A y \triangleright_A , que se mueven hacia la izquierda o derecha, respectivamente, hasta encontrar en la cinta un símbolo de $A \subseteq \Sigma$. Otra es B , que borra la cadena que tiene hacia la izquierda (hasta el blanco).

Definición 4.10 Las máquinas \triangleleft_A , \triangleright_A y B se definen según el siguiente diagrama:



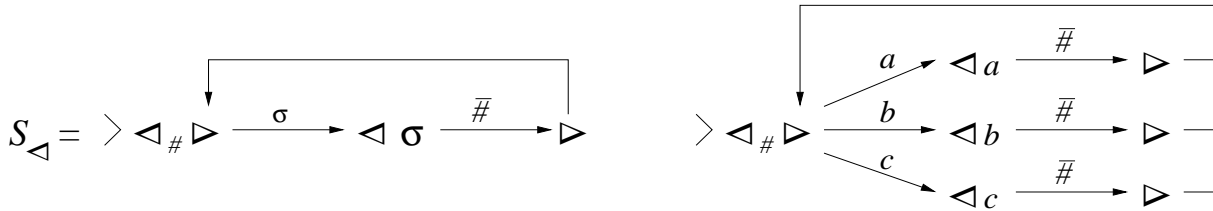
La máquina \triangleleft_A , por ejemplo, comienza moviéndose a la izquierda. Luego, si el carácter sobre el que está parada no está en A , vuelve a moverse, y así. Si, en cierto momento, queda sobre un carácter de A , entonces luego de ejecutar el nodo no tiene flecha aplicable que seguir, y se detiene.

Nótese que \triangleleft_A y \triangleright_A primero se mueven, y luego empiezan a verificar la condición. Es decir, si se arrancan paradas sobre una letra de A , no la verán. Cuando $A = \{a\}$ escribiremos \triangleleft_a y \triangleright_a .

Nótese también la forma de escribir cualquier conjunto en las flechas. También pudimos escribir $\sigma \notin A$, por ejemplo. Si $A = \{a\}$ podríamos haber escrito $\sigma \neq a$. Cuando les damos nombre a los caracteres en las transiciones utilizaremos letras griegas para no confundirnos con las letras de la cinta.

El dar nombre a la letra que está bajo el cabezal se usa para algo mucho más poderoso que expresar condiciones. Nos permitiremos usar ese nombre en el nodo destino de la flecha. Ejemplificaremos esto en la siguiente máquina.

Definición 4.11 La máquina shift left (S_{\triangleleft}) se define de según el siguiente diagrama (parte izquierda).



La máquina S_{\triangleleft} comienza buscando el $\#$ hacia la izquierda (nótese que hemos ya utilizado una máquina no básica como acción). Luego se mueve hacia la derecha una celda. La flecha que sale de este nodo se puede seguir siempre, pues no estamos realmente poniendo una condición sino llamando σ a la letra sobre la que estamos parados. En el nodo destino, la MT se mueve a la izquierda, escribe la σ y, si no está parada sobre el $\#$, se mueve a la derecha y vuelve al nodo original. Más precisamente, vuelve a la acción de moverse a la derecha de ese nodo. Nótese que nos permitimos flechas que llegan a la mitad de un nodo, y ejecutan las acciones del nodo de ahí hacia adelante. Esto no es raro ya que un nodo de tres acciones ABC se puede descomponer en tres nodos $A \rightarrow B \rightarrow C$. Vale la pena recalcar que las dos ocurrencias de σ en el dibujo indican cosas diferentes. La primera denota una letra de la cinta, la segunda una acción.

No existe magia en utilizar variables en esta forma. A la derecha de la Def. 4.11 mostramos una versión alternativa sobre el alfabeto $\Sigma = \{a, b, c, \#\}$. La variable realmente actúa como

una *macro*, y no afecta el modelo de MTs porque el conjunto de valores que puede tomar siempre es finito. Es interesante que este mecanismo, además, *nos independiza del alfabeto*, pues la definición de S_{\triangleleft} se expandiría en máquinas distintas según Σ . En realidad, lo mismo ocurre con las máquinas básicas.

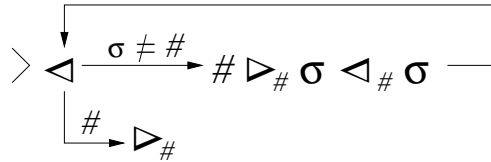
La máquina S_{\triangleleft} , entonces, está pensada para ser arrancada en una configuración tipo $(s, X\#w\#)$ (donde w no contiene blancos), y termina en la configuración $(h, Xw\#)$. Es decir, toma su argumento y lo mueve una casilla a la izquierda. Esto no cae dentro del formalismo de calcular funciones, pues S_{\triangleleft} puede no retornar un $\#$ al comienzo de la cinta. Más bien es una máquina auxiliar para ser usada como acción. La forma de especificar lo que hacen estas máquinas será indicar de qué configuración de la cinta llevan a qué otra configuración, sin indicar el estado. Es decir, diremos $S_{\triangleleft} : \#w\# \rightarrow w\#$. Nuevamente, como S_{\triangleleft} no se cuelga haciendo esto, está claro que si hubiera una X antes del primer blanco, ésta quedaría inalterada.

Existe una máquina similar que mueve w hacia la derecha. Se invita al lector a dibujarla. No olvide dejar el cabezal al final al terminar.

Definición 4.12 La máquina “shift right” opera de la siguiente forma. $S_{\triangleright} : \#w\# \rightarrow \#\#w\#$.

Repetiremos ahora el Ejemplo 4.5 para mostrar cómo se simplifica dibujar MTs con la notación modular.

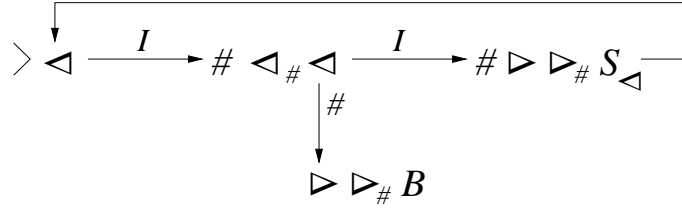
Ejemplo 4.7 La MT del Ejemplo 4.5 se puede dibujar en la notación modular de la siguiente forma. El dibujo no sólo es mucho más simple y fácil de entender, sino que es independiente de Σ .



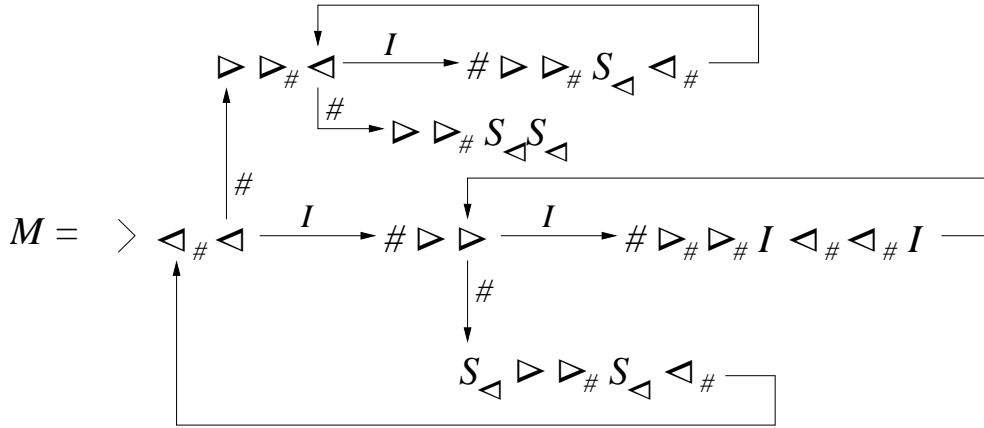
Ejemplo 4.8 Otra máquina interesante es la copiadora, $C : \#w\# \rightarrow \#w\#w\#$. Se invita al lector a dibujarla, inspirándose en la que calcula $f(w) = ww^R$ del Ejemplo 4.7. Con esta máquina podemos componer fácilmente una que calcule $f(w) = ww$: $C S_{\triangleleft}$. Claro que esta no es la máquina más *eficiente* para calcular f (o sea, que lo logre en menos pasos), pero hasta el Capítulo 6 la eficiencia no será un tema del que preocuparnos. *Ya tendremos bastantes problemas con lo que se puede calcular y lo que no.*

Es interesante que la MT que suma dos números, en el Ejemplo 4.6, ahora puede escribirse simplemente como S_{\triangleleft} . La que incrementa un número es $I \triangleright$. Con la notación modular nos podemos atrever a implementar operaciones aritméticas más complejas.

Ejemplo 4.9 Una MT que implementa la función *diferencia* entre números naturales (dando cero cuando la diferencia es negativa) puede ser como sigue.

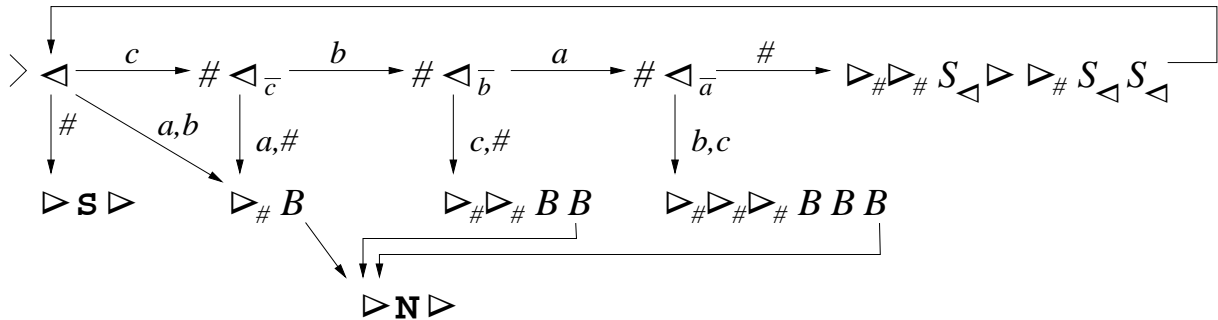


La MT M que implementa la multiplicación puede ser como sigue. En el ciclo principal vamos construyendo $\#I^n\#I^m\#I^{n\cdot m}$ mediante ir reduciendo I^n e ir agregando una copia de I^m al final de la cinta. Cuando I^n desaparece, pasamos a otro ciclo que borra I^m para que quede solamente $\#I^{n\cdot m}\#$.

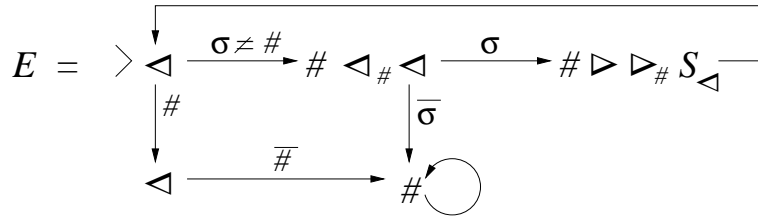


El siguiente ejemplo es importante porque demuestra que existen lenguajes decidibles que no son libres del contexto.

Ejemplo 4.10 La siguiente MT decide el lenguaje $\{a^n b^n c^n, n \geq 0\}$.



Ejemplo 4.11 La siguiente máquina, que llamaremos E , recibe entradas de la forma $\#u\#v\#$ y se detiene sii $u = v$. La usaremos más adelante.



4.4 MTs de k Cintas y Otras Extensiones [LP81, sec 4.5]

Una MT de k cintas tiene un cabezal en cada cinta. En cada paso, lee simultáneamente los k caracteres bajo los cabezales, y toma una decisión basada en la k -upla. Esta consiste de pasar a un nuevo estado y realizar una acción en cada cinta.

Definición 4.14 Una configuración de una MT de k cintas $M = (K, \Sigma, \delta, s)$ es un elemento de $\mathcal{C}_M = (K \cup \{h\}) \times (\Sigma^* \times \Sigma \times (\Sigma^* - (\Sigma^* \circ \{\#\})))^k$. Una configuración se escribirá $(q, u_1 a_1 v_1, u_2 a_2 v_2, \dots, u_k a_k v_k)$.

Definición 4.15 La relación lleva en un paso para una MT de k cintas $M = (K, \Sigma, \delta, s)$, $\vdash_M \subseteq \mathcal{C}_M \times \mathcal{C}_M$, se define como: $(q, u_1 \underline{a_1} v_1, u_2 \underline{a_2} v_2, \dots, u_k \underline{a_k} v_k) \vdash (q', u'_1 \underline{a'_1} v'_1, u'_2 \underline{a'_2} v'_2, \dots, u'_k \underline{a'_k} v'_k)$ si $q \in K$, $\delta(q, a_1, a_2, \dots, a_k) = (q', b_1, b_2, \dots, b_k)$, y las reglas de la Def. 4.3 se cumplen para cada $u_i a_i v_i$, $u'_i a'_i v'_i$ y b_i .

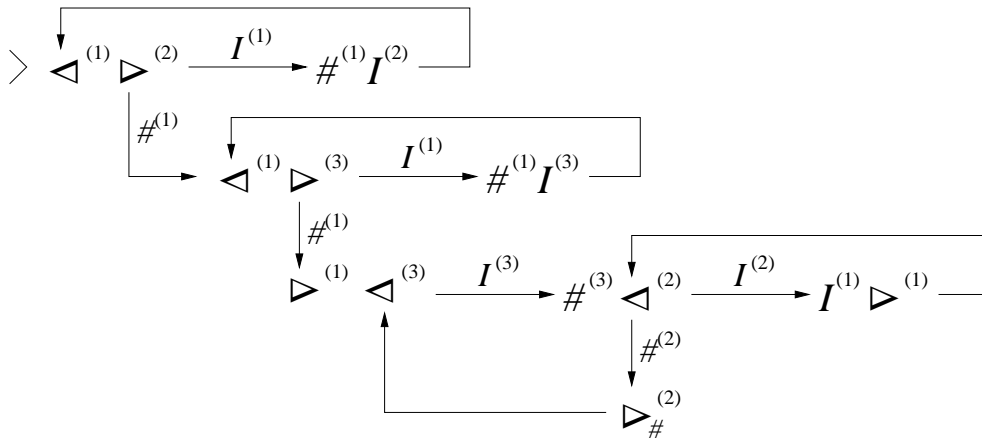
Definamos ahora cómo usaremos una MT de k cintas.

Definición 4.16 Una MT de k cintas arranca con una configuración de la forma $\#w\#$ en la cinta 1, y todas las otras cintas en la configuración $\#$. Cuando se detiene, la configuración de la cinta 1 es la salida, mientras que los contenidos de las otras cintas se ignoran.

La forma en que usaremos una MT de k cintas en la notación modular es algo distinta. En principio parece ser menos potente pero es fácil ver que no es así. En cada acción y cada condición, pondremos un supraíndice de la forma $^{(i)}$ indicando en qué cinta se realiza cada acción o al carácter de qué cinta nos referimos en cada condición. Se puede poner condiciones sobre distintas cintas a la vez, como $a^{(1)}\#^{(2)}$.

El siguiente ejemplo muestra cómo se simplifican algunos problemas si se pueden utilizar varias cintas. La MT misma no es mucho más chica, pero su operatoria es mucho más sencilla de entender.

Ejemplo 4.12 Una MT M que implementa la multiplicación (como en el Ej. 4.9), ahora usando 3 cintas, puede ser como sigue. Ahora I^n e I^m se dejan en las cintas 2 y 3, respectivamente, y el resultado se construye en la cinta 1.

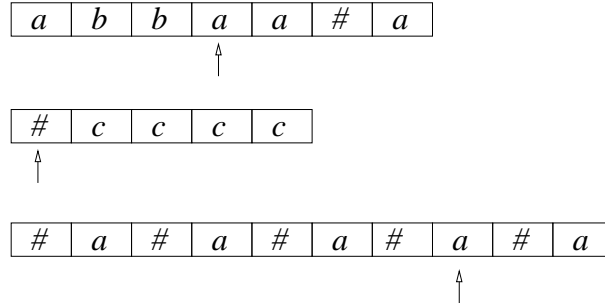


Veamos ahora cómo simular una MT de k cintas con una de 1 cinta, para convencernos de que podemos usar MTs de k cintas de ahora en adelante y saber que, de ser necesario, la podríamos convertir a una tradicional.

La forma en la que simularemos una MT de k cintas consiste en “particionar” la (única) cinta de la MT simuladora (que es una MT normal de una cinta) en $2k$ “pistas”. Cada cinta de la MT simulada se representará con 2 pistas en la cinta simuladora. En la primera pista pondremos el contenido de la cinta, y en la segunda marcaremos la posición del cabezal: esa pista tendrá todos 0’s, excepto un 1 donde está el cabezal. Formalmente, esta cinta particionada se expresa teniendo símbolos de $(\Sigma \times \{0, 1\})^k$ en el alfabeto. También debemos tener los símbolos originales (Σ) porque la simulación empieza y termina con el contenido

de la cinta 1. Y tendremos un símbolo adicional $\$ \notin \Sigma$ para marcar el comienzo de la cinta simuladora.

Por ejemplo, si la MT simulada tiene las cintas en la configuración



entonces la MT simuladora estará en la siguiente configuración

\$	a	b	b	a	a	$\#$	a	$\#$	$\#$	$\#$	#	#	...
	0	0	0	1	0	0	0	0	0	0			
	$\#$	c	c	c	c	$\#$	$\#$	$\#$	$\#$	$\#$			
	1	0	0	0	0	0	0	0	0	0			
	$\#$	a	$\#$	a	$\#$	a	$\#$	a	$\#$	a			
	0	0	0	0	0	0	0	1	0	0			

↑

Notemos que los símbolos grandes son el \$ y el # *verdadero*. Cada *columna* de símbolos pequeños es en realidad *un único carácter* de $(\Sigma \times \{0, 1\})^k$. El cabezal de la simuladora estará siempre al final de la cinta particionada. Para simular un sólo paso de la MT simulada en su estado q , la MT simuladora realizará el siguiente procedimiento:

1. Buscará el 1 en la pista 2, para saber dónde está el cabezal de la cinta 1 simulada.
2. Leerá el símbolo en la pista 1 y lo recordará. ¿Cómo lo recordará? Continuando por una MT distinta para cada símbolo $a \in \Sigma$ que pueda leer. Llamemos σ_1 a este símbolo.
3. Buscará el 1 en la pista 4, para saber dónde está el cabezal de la cinta 2 simulada.
4. Leerá el símbolo en la pista 3 y lo recordará en σ_2 .
- ...
5. Observará el valor de $\delta(q, \sigma_1, \dots, \sigma_k) = (q', b_1, \dots, b_k)$. ¿Cómo? Realmente hay una rutina de éstas para cada q de la MT simulada. La parte de lectura de los caracteres es igual en todas, pero ahora difieren en qué hacen frente a cada tupla de caracteres. Asimismo, para cada una de las $|\Sigma|^k$ posibles tuplas leídas, las acciones a ejecutar serán distintas.
6. Buscará nuevamente el 1 en la pista 2, para saber dónde está el cabezal de la cinta 1 simulada.

7. Ejecutará la acción correspondiente a la cinta 1. Si es \triangleleft (\triangleright), moverá el 1 de la pista 2 hacia la izquierda (derecha). Si es escribir $b \in \Sigma$, reemplazará la letra de la pista 1 por b .
8. Buscará nuevamente el 1 en la pista 4, para saber dónde está el cabezal de la cinta 2 simulada.
9. Ejecutará la acción correspondiente a la cinta 2.
- ...
10. Transferirá el control al módulo que simula un paso cuando la MT simulada está en el estado q' .

Más formalmente, sea $M = (K, \Sigma, \delta, s)$ la MT de k cintas simulada. Entonces tendremos un módulo F_q para cada $q \in K$. (Los $*$ significan cualquier carácter, es una forma de describir un conjunto finito de caracteres de $(\Sigma \times \{0, 1\})^k$.)

$$F_q = \triangleright \triangleleft_{(*,1,*,*,*,*)} \xrightarrow{(\sigma_1,1,*,*,*,*)} \triangleright_{\#} \triangleleft_{(*,*,*,1,*,*)} \xrightarrow{(*,*,\sigma_2,1,*,*)} \triangleright_{\#} \triangleleft_{(*,*,*,*,*,1)} \xrightarrow{(*,*,*,*,\sigma_3,1)} \triangleright_{\#} D_{q,\sigma_1,\sigma_2,\sigma_3}$$

Estos módulos F_q terminan en módulos de la forma D_{q,a_1,\dots,a_k} (notar que, una vez expandidas las macros en el dibujo anterior, hay $|\Sigma|^k$ de éstos módulos D al final, uno para cada posible tupla leída). Lo que hace exactamente cada módulo D depende precisamente de $\delta(q, a_1, \dots, a_k)$. Por ejemplo, si $\delta(q, a_1, a_2, a_3) = (q', \triangleleft, \triangleright, b)$, $b \in \Sigma$, entonces $D_{q,a_1,a_2,a_3} = L_1 R_2 W_{b,3} \longrightarrow F_{q'}$. Estas acciones individuales mueven el cabezal hacia la izquierda (L) o derecha (R) en la pista indicada, o escriben b (W).

$$I_1 = \triangleright \triangleleft_{(*,1,*,*,*,*)} \xrightarrow{(\sigma_1,1,\sigma_2,p_2,\sigma_3,p_3)} (\sigma_1,0,\sigma_2,p_2,\sigma_3,p_3) \triangleleft \xrightarrow{(\sigma'_1,0,\sigma'_2,p'_2,\sigma'_3,p'_3)} (\sigma'_1,1,\sigma'_2,p'_2,\sigma'_3,p'_3) \triangleright_{\#}$$

$$\downarrow \$$$

$$\triangleleft$$

$$D_2 = \triangleright \triangleleft_{(*,*,*,1,*,*)} \xrightarrow{(\sigma_1,p_1,\sigma_2,1,\sigma_3,p_3)} (\sigma_1,p_1,\sigma_2,0,\sigma_3,p_3) \triangleright \xrightarrow{(\sigma'_1,p'_1,\sigma'_2,0,\sigma'_3,p'_3)} (\sigma'_1,p'_1,\sigma'_2,1,\sigma'_3,p'_3) \triangleright_{\#}$$

$$\downarrow \#$$

$$(\#,0,\#,1,\#,0) \triangleright$$

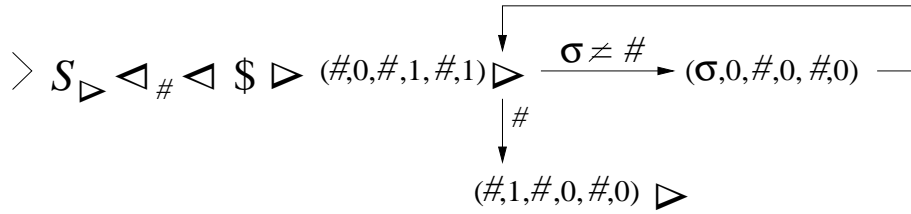
$$W_{b,3} = \triangleright \triangleleft_{(*,*,*,*,*,1)} \xrightarrow{(\sigma_1,p_1,\sigma_2,p_2,\sigma_3,1)} (\sigma_1,p_1,\sigma_2,p_2,b,1) \triangleright_{\#}$$

Observar, en particular, que si la MT simulada se cuelga (intenta moverse al $\$$), la simuladora se cuelga también (podríamos haber hecho otra cosa). Si, en cambio, la acción

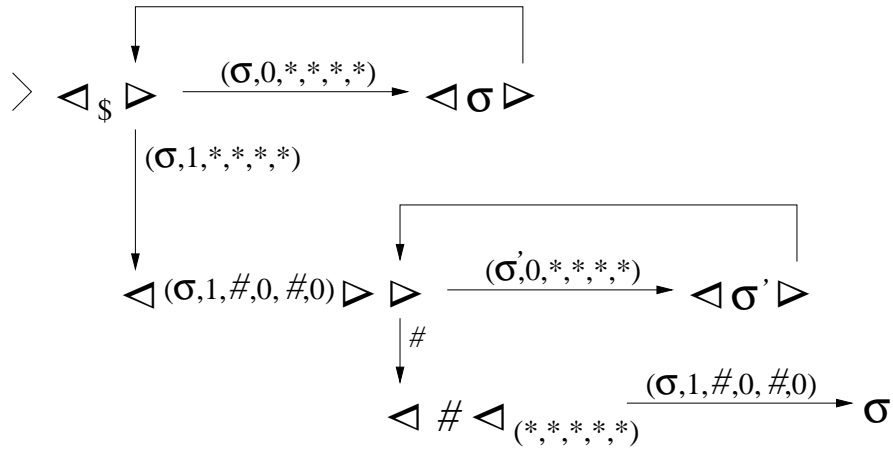
D_i se para sobre un $\#$, significa que está tratando de acceder una parte de la cinta que aún no hemos particionado, por lo que es el momento de hacerlo.

Hemos ya descrito la parte más importante de la simulación. Lo que nos queda es más sencillo: debemos particionar la cinta antes de comenzar, y debemos “des-particionar” la cinta luego de terminar. Lo primero se hace al comenzar y antes de transferir el control a F_s , mientras que lo último se hace en la máquina correspondiente a F_h (que es especial, distinta de todas las otras F_q , $q \in K$).

Para particionar la cinta, simplemente ejecutamos



Finalmente, sigue la máquina des-particionadora F_h . No es complicada, aunque es un poco más enredada de lo que podría ser pues, además del contenido, queremos asegurar de dejar el cabezal de la MT real en la posición en que la MT simulada lo tenía en la cinta 1.



Lema 4.1 Sea una MT de k cintas tal que, arrancada en la configuración $(\#w\#, \#, \dots, \#)$, $w \in (\Sigma - \{\#\})^*$, (1) se detiene en la configuración $(q, u_1 \underline{a_1} v_1, u_2 \underline{a_2} v_2, \dots, u_k \underline{a_k} v_k)$, (2) se cuelga, (3) nunca termina. Entonces se puede construir una MT de una cinta que, arrancada en la configuración $\#w\#$, (1) se detiene en la configuración $u_1 \underline{a_1} v_1$, (2) se cuelga, (3) nunca termina.

Prueba: Basta aplicar la simulación que acabamos de ver sobre la MT de k cintas. \square

4.5 MTs no Determinísticas (MTNDs)

[LP81, sec 4.6]

Una extensión de las MTs que resultará particularmente importante en el Capítulo 6, y que además nos simplificará la vida en varias ocasiones, son las MT no determinísticas (MTNDs). Estas resultan ser equivalentes a las MT tradicionales (determinísticas, que ahora también llamaremos MTD). Una MTND puede, estando en un cierto estado y viendo un cierto carácter bajo el cabezal, tener cero, una, o más transiciones aplicables, y puede elegir cualquiera de ellas. Si no tiene transiciones aplicables, se cuelga (en el sentido de que no pasa a otra configuración, a pesar de no estar en la configuración detenida).

En la notación modular, tendremos cero o más flechas aplicables a partir de un cierto nodo. Si no hay flechas aplicables, la MTND se detiene (notar la diferencia con la notación tradicional de estados y transiciones). Si hay al menos una flecha aplicable, la MTND *debe* elegir alguna, no puede elegir detenerse si puede no hacerlo. Si se desea explicitar que una alternativa válida es detenerse, debe agregarse una flecha hacia un nodo que no haga nada y no tenga salidas.

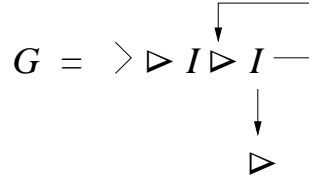
Definición 4.17 Una Máquina de Turing no Determinística (MTND) es una tupla $M = (K, \Sigma, \Delta, s)$, donde K , Σ y s son como en la Def. 4.1 y $\Delta \subseteq (K \times \Sigma) \times ((K \cup \{h\}) \times (\Sigma \cup \{\triangleleft, \triangleright\}))$.

Las configuraciones de una MTND y la relación \vdash son idénticas a las de las MTDs. Ahora, a partir de una cierta configuración, la MTND puede llevar a más de una configuración. Según esta definición, la MTND *se detiene* frente a una cierta entrada sii existe una secuencia de elecciones que la llevan a la configuración detenida, es decir, si tiene forma de detenerse.

Observación 4.6 No es conveniente usar MTNDs para calcular funciones, pues pueden entregar varias respuestas a una misma entrada. En principio las utilizaremos sólo para aceptar lenguajes, es decir, para ver si se detienen o no frente a una cierta entrada. En varios casos, sin embargo, las usaremos como submáquinas y nos interesará lo que puedan dejar en la cinta.

Las MTNDs son sumamente útiles cuando hay que resolver un problema mediante probar todas las alternativas de solución. Permiten reemplazar el mecanismo tedioso de ir generando las opciones una por una, sin que se nos escape ninguna, por un mecanismo mucho más simple de “adivinar” (generar no determinísticamente) una única opción y probarla.

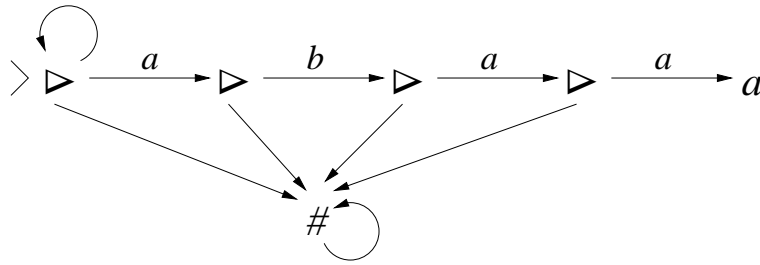
Ejemplo 4.13 Hagamos una MT que acepte el lenguaje de los números compuestos, $\{I^n, \exists p, q \geq 2, n = p \cdot q\}$. (Realmente este lenguaje es decidible, pero aceptarlo ilustra muy bien la idea.) Lo que haríamos con una MTD (y con nuestro lenguaje de programación favorito) sería generar, uno a uno, todos los posibles divisores de n , desde 2 hasta \sqrt{n} , y probarlos. Si encontramos un divisor, n es compuesto, sino es primo. Pero con una MTND es mucho más sencillo. La siguiente MTND genera, no determinísticamente, una cantidad de I 's mayor o igual a 2.



Ahora, una MTND que acepta los números compuestos es $GGME$, donde G es la MTND de arriba, M es la multiplicadora (Ej. 4.9) y E es la MT que se detiene si recibe dos cadenas iguales (Ej. 4.11).

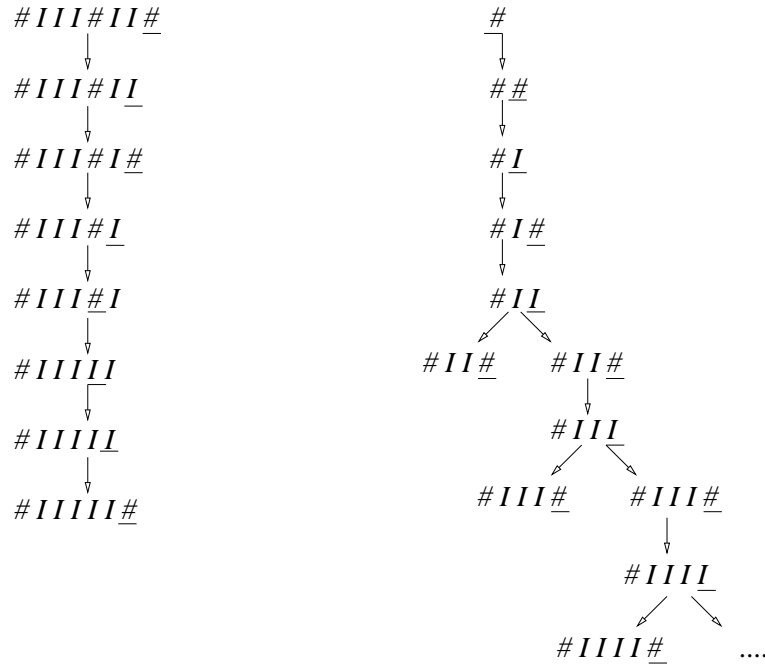
Es bueno detenerse a reflexionar sobre esta MTND. Primero aplica G dos veces, con lo que la cinta queda de la forma $\#I^n\#I^p\#I^q\#$, para algún par $p, q \geq 2$. Al aplicar M , la cinta queda de la forma $\#I^n\#I^{pq}\#$. Al aplicar E , ésta se detendrá sólo si $n = pq$. Esto significa que la inmensa mayoría (infinitas!) de las alternativas que produce GG llevan a correr E para siempre sin detenerse. Sin embargo, si n es compuesto, existe al menos una elección que llevará E a detenerse y la MTND aceptará n .

Ejemplo 4.14 Otro ejemplo útil es usar una MTND para buscar una secuencia dada en la cinta, como $abaa$. Una MTD debe considerar cada posición de comienzo posible, compararla con $abaa$, y volver a la siguiente posición de comienzo. Esto no es demasiado complicado, pero más simple es la siguiente MTND, que naturalmente se detiene en cada ocurrencia posible de $abaa$ en la w , si se la arranca en $\#w$.

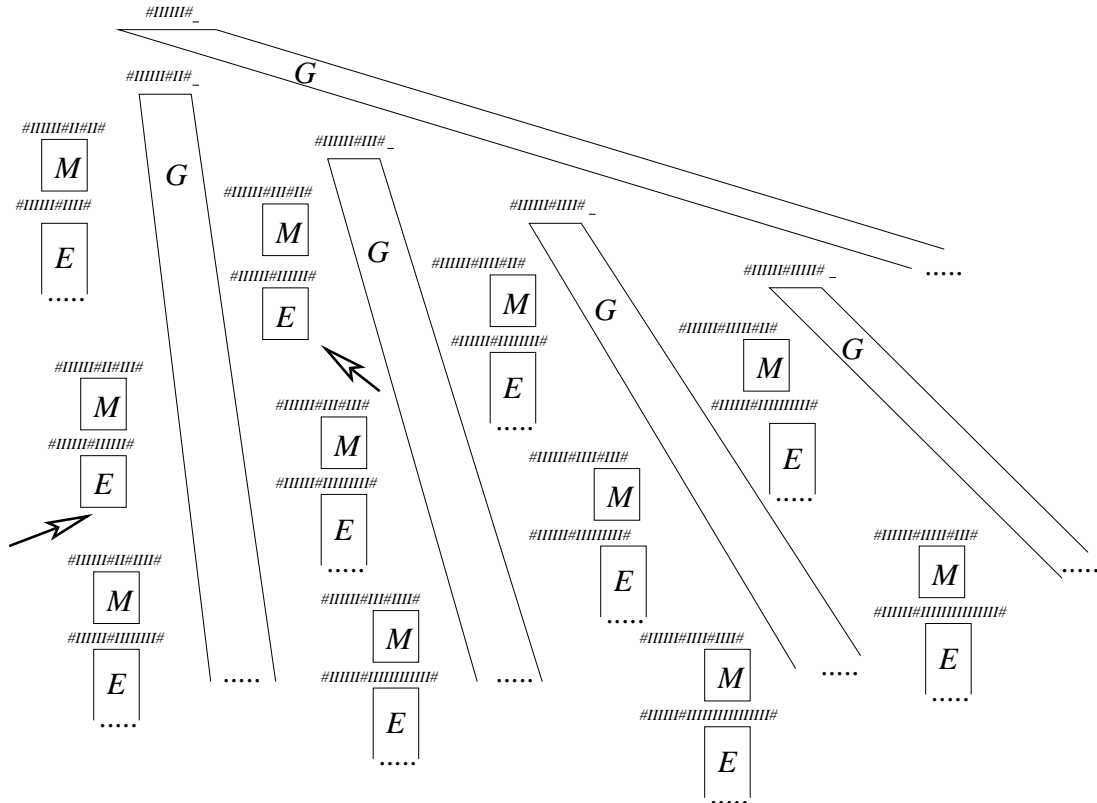


Los ejemplos anteriores nos llevan a preguntarnos cómo se pueden probar todas las alternativas de p y q , si son infinitas. Para comprender esto, y para la simulación de MTNDs con MTDs, es conveniente pensar de la siguiente forma. Una MTD produce una secuencia de configuraciones a lo largo del tiempo. Si las dibujamos verticalmente, tendremos una línea. Una MTND puede, en cada paso, generar más de una configuración. Si las dibujamos verticalmente, con el tiempo fluyendo hacia abajo, tenemos un árbol. En el instante t , el conjunto de configuraciones posibles está indicado por todos los nodos de profundidad t en ese árbol.

Por ejemplo, aquí vemos una ejecución (determinística) de la MT sumadora (Ej. 4.6) a la izquierda, y una ejecución (no determinística) de G (Ej. 4.13) a la derecha. Se ve cómo G puede generar cualquier número: demora más tiempo en generar números más largos, pero todo número puede ser generado si se espera lo suficiente.



De hecho una visualización esquemática de la ejecución de $GGME$ (Ej. 4.13) con la entrada I^6 es como sigue.



Hemos señalado con flechas gruesas los únicos casos (2×3 y 3×2) donde la ejecución termina. Nuevamente, a medida que va pasando más tiempo se van produciendo combinaciones mayores de p, q .

La simulación de una MTND M con una MTD se basa en la idea del árbol. Recorreremos todos los nodos del árbol hasta encontrar uno donde la MTND se detenga (llegue al estado h), o lo recorreremos para siempre si no existe tal nodo. De este modo la MTD simuladora se detendrá si la MTND simulada se detiene. Como se trata de un árbol infinito, hay que recorrerlo por niveles para asegurarse de que, si existe un nodo con configuración detenida, lo encontraremos.

Nótese que la aridad de este árbol es a lo sumo $r = (|K| + 1) \cdot (|\Sigma| + 2)$, pues ése es el total de estados y acciones distintos que pueden derivarse de una misma configuración. Por un rato supondremos que *todos* los nodos del árbol tienen aridad r , y luego resolveremos el caso general.

La MTD simuladora usará tres cintas:

1. En la primera cinta mantendremos la configuración actual del nodo que estamos simulando. La tendremos precedida por una marca \$, necesaria para poder limpiar la cinta al probar un nuevo nodo del árbol.
2. En la segunda guardaremos una copia de la entrada $\#w\#$ intacta.
3. En la tercera almacenaremos una secuencia de *dígitos* en base r (o sea símbolos sobre d_1, d_2, \dots, d_r), llamados *directivas*. Esta secuencia indica el camino desde la raíz hasta el nodo actual. Por ejemplo si se llega al nodo bajando por el tercer hijo de la raíz, luego por el primer hijo del hijo, y luego por el segundo hijo del nieto de la raíz, entonces el contenido de la cinta será $\#d_3d_1d_2\#$. El nodo raíz corresponde a la cadena vacía. Cuando estemos simulando el k -ésimo paso para llegar al nodo actual, estaremos sobre el k -ésimo dígito en la secuencia.

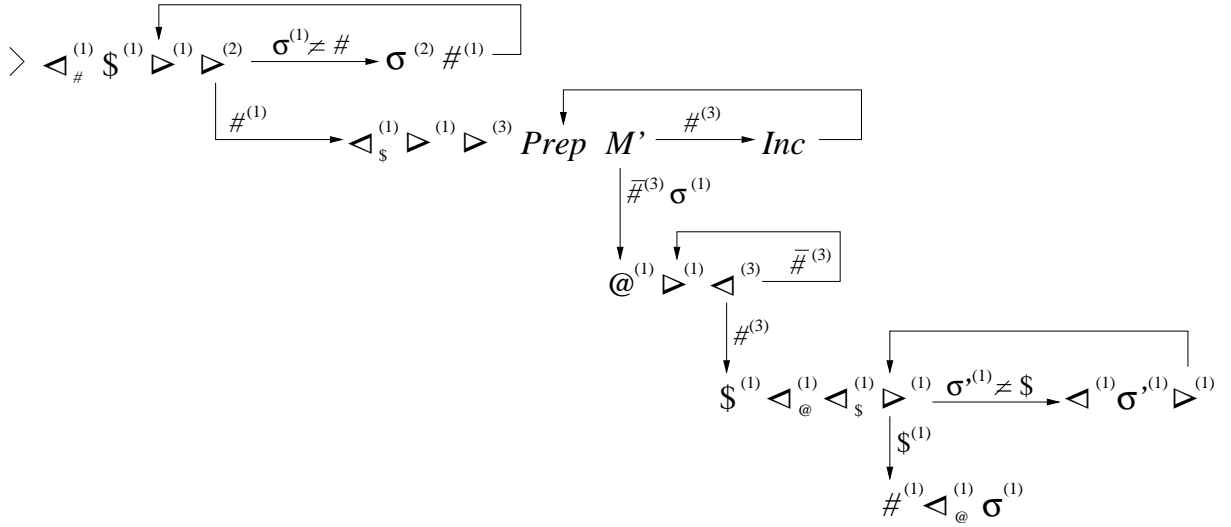
Lo primero que hace la MTD simuladora es copiar la cinta 1 en la 2, poner la marca inicial \$ en la cinta 1, y borrarla. Luego entra en el siguiente ciclo general:

1. Limpiará la cinta 1 y copiará la cinta 2 en la cinta 1 (máquina *Prep*).
2. Ejecutará la MTND en la cinta 1, siguiendo los pasos indicados en las directivas de la cinta 3 (máquina M').
3. Si la MTND no se ha detenido, pasará al siguiente nodo de la cinta 3 (máquina *Inc*) y volverá al paso 1.

Finalmente, eliminará el \$ de la cinta 1 y se asegurará de dejar el cabezal donde la MTND simulada lo tenía al detenerse.

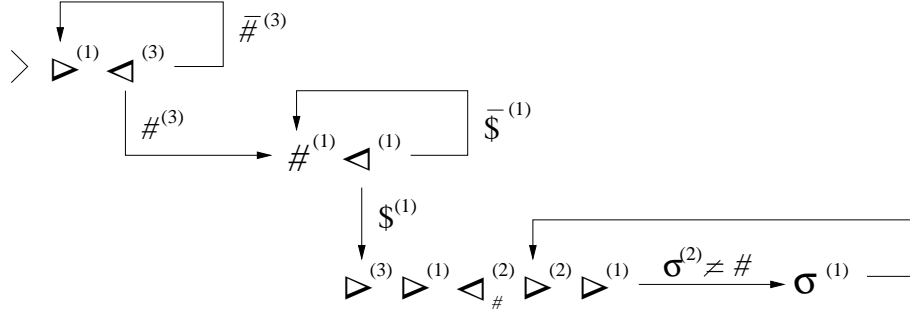
La MTD simuladora es entonces como sigue. El ciclo se detiene si, luego de ejecutar M' , en la cinta de las directivas *no* estamos parados sobre el $\#$ final que sigue a las directivas.

Esto significa que M' se detuvo antes de leerlas todas. O sea, M' se detuvo porque llegó a h y no porque se le acabaron las directivas. En realidad pueden haber pasado ambas cosas a la vez, en cuyo caso no nos daremos cuenta de que M' terminó justo a tiempo. Pero no es problema, nos daremos cuenta cuando tratemos de ejecutar un nodo que descienda del actual, en el siguiente nivel. Para la burocracia final necesitamos otra marca @. Para comprender del todo cómo funciona esta burocracia final es bueno leer primero cómo funciona *Prep* (especialmente el primer punto, pues aquí se hace lo mismo).

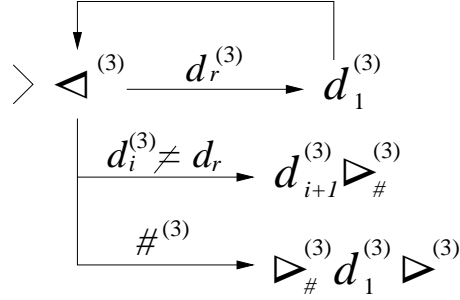


La máquina *Prep* realiza las siguientes acciones:

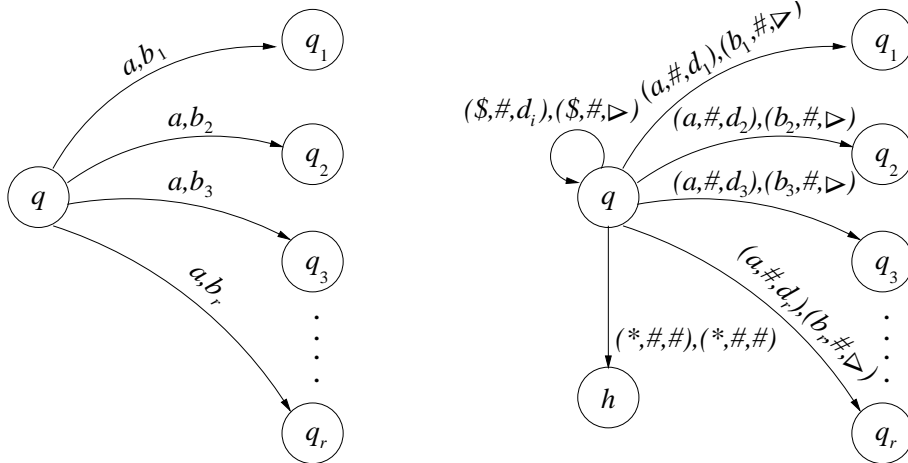
1. Se mueve hacia la derecha en la cinta 1 todo lo necesario para asegurarse de estar más a la derecha que cualquier cosa escrita. Como, en la cinta 3, hemos pasado por una directriz d_i por cada paso de M simulado, y M no puede moverse más de una casilla a la derecha por cada paso, y la simulación de M comenzó con el cabezal al final de su configuración, basta con moverse en la cinta 1 hacia la derecha mientras se mueve a la izquierda en la cinta 3, hasta llegar al primer $\#$ de la cinta 3. Luego vuelve borrando en la cinta 1 hasta el $\$$.
2. Se mueve una casilla hacia adelante en la cinta 3, quedando sobre la primera directiva a seguir en la ejecución del nodo que viene.
3. Copia la cinta 2 a la cinta 1, quedando en la configuración inicial $\#w\#$.



La máquina *Inc* comienza al final de las directivas en la cinta 3 y simula el cálculo del sucesor en un número en base r : Si $r = 3$, el sucesor de d_1d_1 es d_1d_2 , luego d_1d_3 , luego d_2d_1 , y así hasta d_3d_3 , cuyo sucesor es $d_1d_1d_1$. Por ello va hacia atrás convirtiendo d_r en d_1 hasta que encuentra un $d_i \neq d_r$, el cual cambia por d_{i+1} (no hay suma real aquí, hemos abreviado para no poner cada i separadamente). Si llega al comienzo, es porque eran todos d_r , y es hora de pasar al siguiente nivel del árbol.



Finalmente, la máquina M' es la MTD que realmente simula la MTND M , evitando el no determinismo mediante las directivas de la cinta 3. M' tiene los mismos estados y casi las mismas transiciones que M . Cada vez que M tiene r salidas desde un estado por un cierto carácter (izquierda), M' le cambia los rótulos a esas transiciones, considerando las 3 cintas en las que actúa (derecha):



Nótese que M' es determinística, pues no existen dos transiciones que salen de ningún q por las mismas letras (ahora tripletas). Se sabe que siempre está sobre $\#$ en la cinta 2. Ahora elige pasar al estado q_i ejecutando la acción b_i (en la cinta 1) siempre que la directiva actual indique d_i . En la cinta 3, pasa a la siguiente directiva. Además, si se acabaron las directivas, se detiene para dar paso a la simulación del siguiente nodo del árbol. Otro detalle es que, si la MTND se cuelga, lo detectamos porque la simuladora queda sobre el $\$$ en la cinta 1. En ese caso hacemos como si la MTND hubiera usado todas las instrucciones sin terminar, de modo de dar lugar a otros nodos del árbol.

Si en cualquier estado q frente a cualquier carácter a hubiera menos de r alternativas distintas, lo más simple es crear las alternativas restantes, que hagan lo mismo que algunas que ya existen. Si no hubiera ninguna alternativa la MTND se colgaría, por lo que deberíamos agregar una rutina similar a la que hicimos para el caso en que toque $\$$ en la cinta 1.

Lema 4.2 *Todo lenguaje aceptado por una MTND es aceptado por una MTD.*

Prueba: Basta producir la MTD que simula la MTND según lo visto recién y correrla sobre la misma entrada. (Esta MTD es a su vez una MT de 3 cintas que debe ser simulada por una MT de una cinta, según lo visto en el Lema 4.1.) \square

4.6 La Máquina Universal de Turing (MUT) [LP81, sec 5.7]

El principio fundamental de los computadores de propósito general es que no se cablea un computador para cada problema que se desea resolver, sino que se cablea un único computador capaz de *interpretar* programas escritos en algún lenguaje. Ese lenguaje tiene su propio modelo de funcionamiento y el computador *simula* lo que haría ese programa en una cierta entrada. Tanto el programa como la entrada conviven en la memoria. El programa tiene su propio alfabeto (caracteres ASCII, por ejemplo) y manipula elementos de un cierto tipo de datos (incluyendo por ejemplo números enteros), los que el computador *codifica* en su propio lenguaje (bits), en el cual también queda expresada la salida que después el usuario interpretará en términos de los tipos de datos de su lenguaje de programación. El computador debe tener, en su propio cableado, suficiente poder para simular cualquier programa escrito en ese lenguaje de programación, por ejemplo no podría simular un programa en Java si no tuviera una instrucción GOTO o similar.

Resultará sumamente útil para el Capítulo 5 tener un modelo similar para MTs. En particular, elegimos las MTs como nuestro modelo de máquina “cableada” y a la vez como nuestro modelo de lenguaje de programación. La *Máquina Universal de Turing (MUT)* recibirá dos entradas: una MT M y una entrada w , *codificadas de alguna forma*, y *simulará* el funcionamiento de M sobre w . La simulación se detendrá, se colgará, o correrá para siempre según M lo haga con w . En caso de terminar, dejará en la cinta la codificación de lo que M dejaría en la cinta frente a w .

¿Por qué necesitamos codificar? Si vamos a representar toda MT posible, existen MTs con alfabeto Σ (finito) de tamaño n para todo n , por lo cual el alfabeto de nuestra MT

debería ser infinito. El mismo problema se presenta en un computador para representar cualquier número natural, por ejemplo. La solución es similar: codificar cada símbolo del alfabeto como una *secuencia* de símbolos sobre un alfabeto finito. Lo mismo pasa con la codificación de los estados de M .

Para poder hacer esta codificación impondremos una condición a la MT M , la cual obviamente no es restrictiva.

Definición 4.18 Una MT $M = (K, \Sigma, \delta, s)$ es codificable si $K = \{q_1, q_2, \dots, q_{|K|}\}$ y $\Sigma = \{\#, a_2, \dots, a_{|\Sigma|}\}$. Definimos también $K_\infty = \{q_1, q_2, \dots\}$, $\Sigma_\infty = \{\#, a_2, \dots\}$. Consideraremos $a_1 = \#$.

Es obvio que para toda MT M' existe una MT M codificable similar, en el sentido de que lleva de la misma configuración a la misma configuración una vez que mapeamos los estados y el alfabeto.

Definiremos ahora la codificación que usaremos para MTs codificables. Usaremos una función auxiliar λ para denotar estados, símbolos y acciones.

Definición 4.19 La función $\lambda : K_\infty \cup \{h\} \cup \Sigma_\infty \cup \{\triangleleft, \triangleright\} \longrightarrow I^*$ se define como sigue:

x	$\lambda(x)$
h	I
q_i	I^{i+1}
\triangleleft	I
\triangleright	II
$\#$	III
a_i	I^{i+2}

Nótese que λ puede asignar el mismo símbolo a un estado y a un carácter, pero no daremos lugar a confusión.

Para codificar una MT esencialmente codificaremos su estado inicial y todas las celdas de δ . Una celda se codificará de la siguiente forma.

Definición 4.20 Sea $\delta(q_i, a_j) = (q', b)$ una entrada de una MT codificable. Entonces

$$S_{i,j} = c \lambda(q_i) c \lambda(a_j) c \lambda(q') c \lambda(b) c$$

Con esto ya podemos definir cómo se codifican MTs y cintas. Notar que el nombre de función ρ está sobrecargado.

Definición 4.21 La función ρ convierte una MT codificable $M = (K, \Sigma, \delta, s)$ en una secuencia sobre $\{c, I\}^*$, de la siguiente forma:

$$\rho(M) = c \lambda(s) c S_{1,1} S_{1,2} \dots S_{1,|\Sigma|} S_{2,1} S_{2,2} \dots S_{2,|\Sigma|} \dots S_{|K|,1} S_{|K|,2} \dots S_{|K|,|\Sigma|} c$$

Definición 4.22 La función ρ convierte una $w \in \Sigma_\infty^*$ en una secuencia sobre $\{c, I\}^*$, de la siguiente forma:

$$\rho(w) = c \lambda(w_1) c \lambda(w_2) c \dots c \lambda(w_{|w|}) c$$

Notar que $\rho(\varepsilon) = c$.

Finalmente, estamos en condiciones de definir la MUT.

Definición 4.23 La Máquina Universal de Turing (MUT), arrancada en una configuración $(s_{MUT}, \# \rho(M) \rho(w) \#)$, donde s_{MUT} es su estado inicial, $M = (K, \Sigma, \delta, s)$ es una MT codificable, y todo $w_i \in \Sigma - \{\#\}$, hace lo siguiente:

1. Si, arrancada en la configuración $(s, \# w \#)$, M se detiene en una configuración $(h, u \underline{v})$, entonces la MUT se detiene en la configuración $(h, \# \rho(u) \lambda(a) \rho(v))$, con el cabezal en la primera c de $\rho(v)$.
2. Si, arrancada en la configuración $(s, \# w \#)$, M no se detiene nunca, la MUT no se detiene nunca.
3. Si, arrancada en la configuración $(s, \# w \#)$, M se cuelga, la MUT se cuelga.

Veamos ahora cómo construir la MUT. Haremos una construcción de 3 cintas, ya que sabemos que ésta se puede traducir a una cinta:

1. En la cinta 1 tendremos la representación de la cinta simulada (inicialmente $\rho(\# w \#)$) y el cabezal estará en la c que sigue a la representación del carácter donde está el cabezal representado. Inicialmente, la configuración es $\# \rho(\# w) \lambda(\#) \underline{c}$.
2. En la cinta 2 tendremos siempre $\# \rho(M) \#$ y no la modificaremos.
3. En la cinta 3 tendremos $\# \lambda(q) \#$, donde q es el estado en que está la máquina simulada.

El primer paso de la simulación es, entonces, pasar de la configuración inicial $(\# \rho(M) \rho(w) \#, \#, \#)$, a la apropiada para comenzar la simulación: $(\rho(\# w) \lambda(\#) \underline{c}, \# \rho(M) \#, \# \lambda(s) \#)$ ($\lambda(s)$ se obtiene del comienzo de $\rho(M)$ y en realidad se puede eliminar de $\rho(M)$ al moverlo a la cinta 2). Esto no conlleva ninguna dificultad. (Nótese que se puede saber dónde empieza $\rho(w)$ porque es el único lugar de la cinta con tres c 's seguidas.)

Luego de esto, la MUT entra en un ciclo de la siguiente forma:

1. Verificamos si la cinta 3 es igual a $\#I\#$, en cuyo caso se detiene (pues $I = \lambda(h)$ indica que la MT simulada se ha detenido). Recordemos que en una simulación de k cintas, la cinta 1 es la que se entrega al terminar. Esta es justamente la cinta donde tenemos codificada la cinta que dejó la MT simulada.
2. Si la cinta 3 es igual a $\#I^i\#$ y en la cinta 1 alrededor del cabezal tenemos $\dots cI^j\underline{c}\dots$, entonces se busca en la cinta 2 el patrón $ccI^i cI^j c$. Notar que esta entrada *debe* estar si nos dieron una representación correcta de una MT y una w con el alfabeto adecuado.
3. Una vez que encontramos ese patrón, examinamos lo que le sigue. Digamos que es de la forma $I^r cI^s c$. Entonces reescribimos la cinta 3 para que diga $\#I^r\#$, y:
 - (a) Si $s = 1$ debemos movernos a la izquierda en la cinta simulada. Ejecutamos $\triangleleft_{c,\#}^{(1)}$. Si quedamos sobre una c , terminamos de simular este paso. Si quedamos sobre un blanco $\#$, la MT simulada se ha colgado y debemos colgarnos también ($\triangleleft^{(1)}$), aunque podríamos hacer otra cosa. En cualquier caso, al movernos debemos asegurarnos de no dejar $\lambda(\#)c$ al final de la cinta, por la regla de que las configuraciones no deberían terminar en $\#$. Así, antes de movernos a la izquierda debemos verificar que la cinta que nos rodea no es de la forma $\dots c\lambda(\#)\underline{c}\# \dots = \dots cIII\underline{c}\# \dots$. Si lo es, debemos borrar el $\lambda(\#)c$ final antes que nada.
 - (b) Si $s = 2$, debemos movernos a la derecha en la cinta simulada. Ejecutamos $\triangleright_{c,\#}^{(1)}$. Si quedamos sobre una c , terminamos de simular este paso. Si quedamos sobre un blanco $\#$, la MT simulada se ha movido a la derecha a una celda nunca explorada. En este caso, escribimos $\lambda(\#)c = IIIc$ a partir del $\#$ y quedamos parados sobre la c final.
 - (c) Si $s > 2$, debemos modificar el símbolo bajo el cabezal de la cinta simulada. Es decir, el entorno alrededor del cabezal en la cinta 1 es $\dots cI^j\underline{c}\dots$ y debemos convertirlo en $\dots cI^s\underline{c}\dots$. Esto no es difícil pero es un poco trabajoso, ya que involucra hacer o eliminar espacio para el nuevo símbolo, que tiene otro largo. No es difícil crear un espacio con la secuencia de acciones $\# \triangleright_{\#} S_{\triangleright} \triangleleft_{\#} c \triangleleft I \triangleright$, o borrarlo con la secuencia $\triangleleft_{\#} \triangleright_{\#} S_{\triangleleft} \triangleleft_{\#} c$.
4. Volvemos al paso 1.

La descripción demuestra que la MUT realmente no es excesivamente compleja. De hecho, escribirla explícitamente es un buen ejercicio para demostrar maestría en el manejo de MTs, y no debería tomar más de un par de horas, debugging con *JTV* incluido.

4.7 La Tesis de Church

[LP81, sec 5.1]

Al principio de este capítulo se explicaron las razones para preferir las MTs como mecanismo para estudiar computabilidad. Es hora de dar soporte a la correctitud de esta decisión.

¿Qué debería que significara que algo es o no “computable”, para que lo que podamos demostrar sobre computabilidad sea relevante para nosotros? Quisiéramos que la definición capture los procedimientos que pueden ser realizados en forma mecánica y sistemática, con una cantidad finita (pero ilimitada) de recursos (tiempo, memoria).

¿Qué tipo de objetos quisiéramos manejar? Está claro que cadenas sobre alfabetos finitos o numerables, o números enteros o racionales son realistas, porque existe una *representación finita* para ellos. No estaría tan bien el permitirnos representar cualquier número real, pues no tienen una representación finita (no alcanzan las secuencias finitas de símbolos en ningún alfabeto para representarlos, recordar el Teo. 1.2). Si los conjuntos de cardinal \aleph_1 se permitieran, podríamos también permitir programas infinitos, que podrían reconocer cualquier lenguaje o resolver cualquier problema mediante un código que considerara las infinitas entradas posibles una a una:

```
if w = abbab then return S
if w = bbabbabbabbabbbb then return S
if w = bb then return N
if w = bbabbaba then return S
...
```

lo cual no es ni interesante ni realista, al menos con la tecnología conocida.

¿Qué tipo de acciones quisiéramos permitir sobre los datos? Está claro que los autómatas finitos o de pila son mecanismos insatisfactorios, pues no pueden reconocer lenguajes que se pueden reconocer fácilmente en nuestro PC. Las MTs nos han permitido resolver todo lo que se nos ha ocurrido hasta ahora, pero pronto veremos cosas que no se pueden hacer. Por lo tanto, es válido preguntarse si un límite de las MTs debe tomarse en serio, o más generalmente, cuál es un modelo válido de computación en el mundo físico. Esta es una pregunta difícil de responder sin sesgarnos a lo que conocemos. ¿Serán aceptables la computación cuántica (¿se podrá finalmente implementar de verdad en el mundo físico?), la computación biológica (al menos ocurre en la realidad), la computación con cristales (se ha dicho que la forma de cristalizarse de algunas estructuras al pasar al estado sólido resuelve problemas considerados no computables)? ¿No se descubrirá mañana un mecanismo hoy impensable de computación?

La discusión debería convencer al lector de que el tema es debatible y además que no se puede *demostrar* algo, pues estamos hablando del mundo físico y no de objetos abstractos. Nos deberemos contentar con un modelo que nos parezca *razonable y convincente* de qué es lo computable. En este sentido, es muy afortunado que los distintos modelos de computación

que se han usado para expresar lo que todos entienden por computable, se han demostrado equivalentes entre sí. Algunos son:

1. Máquinas de Turing.
2. Máquinas de Acceso Aleatorio (RAM).
3. Funciones recursivas.
4. Lenguajes de programación (teóricos y reales).
5. Cálculo λ .
6. Gramáticas y sistemas de reescritura.

Esta saludable coincidencia es la que le da fuerza a la llamada *Tesis de Church*.

Definición 4.24 *La Tesis de Church establece que las funciones y problemas computables son precisamente los que pueden resolverse con una Máquina de Turing.*

Una buena forma de convencer a alguien con formación en computación es mostrar que las MTs son equivalentes a las máquinas RAM, pues estas últimas son una abstracción de los computadores que usamos todos los días. Existen muchos modelos de máquinas RAM. Describimos uno simple a continuación.

Definición 4.25 *Un modelo de máquina RAM es como sigue: existe una memoria formada por celdas, cada una almacenando un número natural m_i e indexada por un número natural $i \geq 0$. Un programa es una secuencia de instrucciones L_l , numeradas en líneas $l \geq 0$. La instrucción en cada línea puede ser:*

1. SET i, a , que asigna $m_i \leftarrow a$, donde a es constante.
2. MOV i, j , que asigna $m_i \leftarrow m_j$.
3. SUM i, j , que asigna $m_i \leftarrow m_i + m_j$.
4. SUB i, j , que asigna $m_i \leftarrow \max(0, m_i - m_j)$.
5. IFZ i, l , que si $m_i = 0$ transfiere el control a la línea L_l , donde l es una constante.

*En todas las instrucciones, i (lo mismo j) puede ser un simple número (representando una celda fija m_i), o también de la forma $*i$, para una constante i , donde i es ahora la dirección de la celda que nos interesa (m_{m_i}).*

El control comienza en la línea L_1 , y luego de ejecutar la L_l pasa a la línea L_{l+1} , salvo posiblemente en el caso de IFZ. La entrada y la salida quedan en la memoria en posiciones convenientes. Una celda no accesada contiene el valor cero. La ejecución termina luego de ejecutar la última línea.

No es difícil convencerse de que el modelo de máquina RAM que hemos definido es tan potente como cualquier lenguaje Ensamblador (Assembler) de una arquitectura (al cual a su vez se traducen los programas escritos en cualquier lenguaje de programación). *De hecho podríamos haber usado un lenguaje aún más primitivo, sin SET, SUM y SUB sino sólo INC m_i , que incrementa m_i .* Tampoco es difícil ver que una máquina RAM puede simular una MT (¡el JTV es un buen ejemplo!). Veamos que también puede hacerse al revés.

Una MT de 3 cintas que simula nuestra máquina RAM almacenará las celdas que han sido inicializadas en la cinta 1 de la siguiente forma: si $m_i = a$ almacenará en la cinta 1 una cadena de la forma $cI^{i+1}cI^{a+1}c$. La cinta estará compuesta de todas las celdas asignadas, con esta representación concatenada, y todo precedido por una c (para que toda celda comience con cc). Cada línea L_l tendrá una pequeña MT M_l que la simula. Luego de todas las líneas, hay una M_l extra que simplemente se detiene.

1. Si L_l dice SET i, a , la M_l buscará $ccI^{i+1}c$ en la cinta 1. Si no la encuentra agregará $ccI^{i+1}cIc$ al final de la cinta (inicializando así $m_i \leftarrow 0$). Ahora, modificará lo que sigue a $ccI^{i+1}c$ para que sea $I^{a+1}c$ (haciendo espacio de ser necesario) y pasará a M_{l+1} . Si la instrucción dijera SET $*i, a$, entonces se averigua (e inicializa de ser necesario) el valor de m_i sólo para copiar I^{m_i} a una cinta 2. Luego debe buscarse la celda que empieza con $ccI^{m_i}c$ en la cinta 1, y recién reemplazar lo que sigue por $I^{a+1}c$. En los siguientes ítems las inicializaciones serán implícitas para toda celda que no se encuentre, y no se volverán a mencionar.
2. Si L_l dice MOV i, j , la M_l buscará $ccI^{j+1}c$ en la cinta 1 y copiará los I 's que le siguen en la cinta 2. Luego, buscará $ccI^{i+1}c$ en la cinta 1 y modificará los I 's que siguen para que sean iguales al contenido de la cinta 2. Luego pasará a M_{l+1} . Las adaptaciones para los casos $*i$ y/o $*j$ son similares a los de SET y no se volverán a mencionar (se puede llegar a usar la tercera cinta en este caso, por comodidad).
3. Si L_l dice SUM i, j , la M_l buscará $ccI^{j+1}c$ en la cinta 1 y copiará los I 's que le siguen en la cinta 2. Luego, buscará $ccI^{i+1}c$ en la cinta 1 y, a los I 's que le siguen, les agregará los de la cinta 2 menos uno.
4. Si L_l dice SUB i, j , la M_l buscará $ccI^{j+1}c$ en la cinta 1 y copiará los I 's que le siguen en la cinta 2. Luego, buscará $ccI^{i+1}c$ en la cinta 1 y, a los I 's que le siguen, les quitará la cantidad que haya en la cinta 2 (dejando sólo un I si son la misma cantidad o menos).
5. Si L_l dice IFZ i, l' , la M_l buscará $ccI^{i+1}c$ en la cinta 1. Luego verá qué sigue a $ccI^{i+1}c$. Si es Ic , pasará a la $M_{l'}$, sino a la M_{l+1} .

No es difícil ver que la simulación es correcta y que no hay nada del modelo RAM que una MT no pueda hacer. Asimismo es fácil ver que se puede calcular lo que uno quiera calcular en un PC usando este modelo RAM (restringido a los naturales, pero éstos bastan para representar otras cosas como enteros, racionales, e incluso strings si se numeran

adecuadamente). Si el lector está pensando en los reales, debe recordar que en un PC no se puede almacenar cualquier real, sino sólo algunos racionales.

Lema 4.3 *Los modelos de la MT y la máquina RAM son computacionalmente equivalentes.*

Prueba: La simulación y discusión anterior lo prueban. \square

En lo que resta, en virtud de la Tesis de Church, raramente volveremos a prefijar las palabras “decidible” y “aceptable” con “Turing-”, aunque algunas veces valdrá la pena enfatizar el modelo de MT.

4.8 Gramáticas Dependientes del Contexto (GDC)

[LP81, sec 5.2]

Otro modelo de computación equivalente a MTs es el de las gramáticas dependientes del contexto, también llamadas “sistemas de reescritura”. Las estudiaremos en esta sección porque completan de modo natural la dicotomía que venimos haciendo entre mecanismos para generar versus reconocer lenguajes.

Definición 4.26 *Una gramática dependiente del contexto (GDC) es una tupla $G = (V, \Sigma, R, S)$, donde*

1. V es un conjunto finito de símbolos no terminales.
2. Σ es un conjunto finito de símbolos terminales, $V \cap \Sigma = \emptyset$.
3. $S \in V$ es el símbolo inicial.
4. $R \subset_F ((V \cup \Sigma)^* - \Sigma^*) \times (V \cup \Sigma)^*$ son las reglas de derivación (conjunto finito).

Escribiremos las reglas de R como $x \longrightarrow_G z$ o simplemente $x \longrightarrow z$ en vez de (x, z) .

Se ve que las GDCs se parecen bastante, en principio, a las GLCs del Capítulo 3, con la diferencia de que se transforman subcadenas completas (dentro de una mayor) en otras, no sólo un único símbolo no terminal. Lo único que se pide es que haya algún no terminal en la cadena a reemplazar. Ahora definiremos formalmente el lenguaje descrito por una GDC.

Definición 4.27 *Dada una GDC $G = (V, \Sigma, R, S)$, la relación lleva en un paso $\Longrightarrow_G \subseteq (V \cup \Sigma)^* \times (V \cup \Sigma)^*$ se define como*

$$\forall u, v, \forall x \longrightarrow z \in R, \quad uxv \Longrightarrow_G uzv.$$

Definición 4.28 Definimos la relación lleva en cero o más pasos, \Rightarrow_G^* , como la clausura reflexiva y transitiva de \Rightarrow_G .

Escribiremos simplemente \Rightarrow e \Rightarrow^* cuando G sea evidente.

Notamos que se puede llevar en cero o más pasos a una secuencia que aún contiene no terminales. Cuando la secuencia tiene sólo terminales, ya no se puede transformar más.

Definición 4.29 Dada una GDC $G = (V, \Sigma, R, S)$, definimos el lenguaje generado por G , $\mathcal{L}(G)$, como

$$\mathcal{L}(G) = \{w \in \Sigma^*, S \Rightarrow_G^* w\}.$$

Finalmente definimos los lenguajes dependientes del contexto como los expresables con una GDC.

Definición 4.30 Un lenguaje L es dependiente del contexto (DC) si existe una GDC G tal que $L = \mathcal{L}(G)$.

Un par de ejemplos ilustrarán el tipo de cosas que se pueden hacer con GDCs. El primero genera un lenguaje que no es LC.

Ejemplo 4.15 Una GDC que genere el lenguaje $\{w \in \{a, b, c\}^*, w \text{ tiene la misma cantidad de } a\text{'s, } b\text{'s, y } c\text{'s}\}$ puede ser $V = \{S, A, B, C\}$ y R con las reglas:

$$\begin{array}{llllll} S \longrightarrow ABCS & AB \longrightarrow BA & AC \longrightarrow CA & BC \longrightarrow CB & A \longrightarrow a \\ S \longrightarrow \varepsilon & BA \longrightarrow AB & CA \longrightarrow AC & CB \longrightarrow BC & B \longrightarrow b \\ & & & & C \longrightarrow c \end{array}$$

A partir de S se genera una secuencia $(ABC)^n$, y las demás reglas permiten alterar el orden de esta secuencia de cualquier manera. Finalmente, los no terminales se convierten a terminales.

El segundo ejemplo genera otro lenguaje que no es LC, e ilustra cómo una GDC permite funcionar como si tuviéramos un *cursor* sobre la cadena. Esta idea es esencial para probar la equivalencia con MTs.

Ejemplo 4.16 Una GDC que genera $\{a^{2^n}, n \geq 0\}$, puede ser como sigue: $V = \{S, [,], A, D\}$ y R conteniendo las reglas:

$$\begin{array}{ll} S \longrightarrow [A] & [\longrightarrow [D \\ [\longrightarrow \varepsilon & D] \longrightarrow] \\] \longrightarrow \varepsilon & DA \longrightarrow AAD \\ A \longrightarrow a & \end{array}$$

La operatoria es como sigue. Primero se genera $[A]$. Luego, tantas veces como se quiera, aparece el “duplicador” D por la izquierda, y pasa por sobre la secuencia duplicando la cantidad de A 's, para desaparecer por la derecha. Finalmente, se eliminan los corchetes y las A 's se convierten en a 's. Si bien la GDC puede intentar operar en otro orden, es fácil ver que no puede generar otras cosas (por ejemplo, si se le ocurre hacer desaparecer un corchete cuando tiene una D por la mitad de la secuencia, nunca logrará generar nada; también puede verse que aunque se tengan varias D 's simultáneas en la cadena no se pueden producir resultados incorrectos).

Tal como las MTs, las GDCs son tan poderosas que pueden utilizarse para otras cosas además de generar lenguajes. Por ejemplo, pueden usarse para calcular funciones:

Definición 4.31 Una GDC $G = (V, \Sigma, R, S)$ computa una función $f : \Sigma_0^* \longrightarrow \Sigma_1^*$ ($\Sigma_0 \cup \Sigma_1 \subseteq \Sigma - \{\#\}$) si existen cadenas $x, y, x', y' \in (V \cup \Sigma)^*$ tal que, para toda $u \in \Sigma_0^*$, $xuy \xRightarrow{*}_G x'vy'$ sii $v = f(u)$. Si existe tal G decimos que f es gramaticalmente computable. Esta definición incluye las funciones de \mathbb{N} en \mathbb{N} mediante convertir I^n en $I^{f(n)}$.

Ejemplo 4.17 Una GDC que calcule $f(n) = 2n$ es parecida a la del Ej. (4.16), $V = \{D\}$, R conteniendo la regla $Da \longrightarrow aaD$, y con $x = D$, $y = \varepsilon$, $x' = \varepsilon$, $y' = D$. Notar que es irrelevante cuál es el símbolo inicial de G .

Otro ejemplo interesante, que ilustra nuevamente el uso de cursores, es el siguiente.

Ejemplo 4.18 Una GDC que calcule $f(w) = w^R$ con $\Sigma = \{a, b\}$ puede ser como sigue: $x = [, y = *], x' = [, y' = *]$, y las reglas

$$\begin{array}{ll} [a \longrightarrow [A & [b \longrightarrow B \\ Aa \longrightarrow aA & Ba \longrightarrow aB \\ Ab \longrightarrow bA & Bb \longrightarrow bB \\ A* \longrightarrow *a & B* \longrightarrow *b \end{array}$$

Demostremos ahora que las GDCs son equivalentes a las MTs. Cómo hacer esto no es tan claro como en los capítulos anteriores, porque podemos usar tanto las GDCs como las MTs para diversos propósitos. Pero vamos a demostrar algo suficientemente fundamental como para derivar fácilmente lo que queramos.

Lema 4.4 Sea $M = (K, \Sigma, \delta, s)$ una MTD. Entonces existe una GDC $G = (V, \Sigma, R, S)$ donde $V = K \cup \{h, [,]\}$ tal que $(q, u\underline{a}v) \vdash_M^* (q', u'\underline{a}'v')$ sii $[uqav] \xRightarrow{*}_G [u'q'a'v']$.

Prueba: Las reglas necesarias se construyen en función de δ .

1. Si $\delta(q_1, a) = (q_2, \triangleleft)$ agregamos a R las reglas $bq_1ac \longrightarrow q_2bac$ para todo $b \in \Sigma$, $c \in \Sigma \cup \{\}$, excepto el caso $ac = \#]$, donde $bq_1\#] \longrightarrow q_2b]$ evita que queden $\#$'s espurios al final de la configuración.

2. Si $\delta(q_1, a) = (q_2, \triangleright)$ agregamos a R las reglas $q_1ab \longrightarrow aq_2c$ para todo $c \in \Sigma$, y $q_1a] \longrightarrow aq_2\#]$ para extender la cinta cuando sea necesario.
3. Si $\delta(q_1, a) = (q_2, b)$ agregamos a R la regla $q_1a \longrightarrow q_2b$.

Es fácil ver por inspección que estas reglas simulan exactamente el comportamiento de M . \square

Con el Lema 4.4 es fácil establecer la equivalencia de MTs y GDCs para calcular funciones.

Teorema 4.1 *Toda función Turing-computable es gramaticalmente computable, y viceversa.*

Prueba: Sea f Turing-computable. Entonces existe una MTD $M = (K, \Sigma, \delta, s)$ tal que para todo u , $(s, \#u\#) \vdash_M^* (h, \#f(u)\#)$. Por el Lema 4.4, existe una GDC G tal que $[\#us\#] \Longrightarrow_G^* [\#f(u)h\#]$ (y a ninguna otra cadena terminada con $h\#$), pues M es determinística). Entonces, según la Def. 4.31, f es gramaticalmente computable, si elegimos $x = [\#, y = s\#]$, $x' = [\#, y' = h\#]$.

La vuelta no la probaremos tan detalladamente. Luego de haber construido la MUT (Sección 4.6), no es difícil ver que uno puede poner la cadena inicial (rematada por x e y en las dos puntas) en la cinta 1, todas las reglas en una cinta 2, y usar una MTND que elija la regla a aplicar, el lugar donde aplicarla, y si tal cosa es posible, cambie en la cinta 1 la parte izquierda de la regla por la parte derecha. Luego se verifica si los topes de la cinta son x' e y' . Si lo son, la MTND elimina x' e y' y se detiene, sino vuelve a elegir otra regla e itera. En este caso sabemos que la MTND siempre se terminará deteniendo y que dejará el resultado correcto en la cinta 1. \square

Nótese que, usando esto, podemos hacer una GDC que “decida” cualquier lenguaje Turing-decidible L . Lo curioso es que, en vez de *generar* las cadenas de L , esta GLC las *convierte* a \mathbf{S} o a \mathbf{N} según estén o no en L . ¿Y qué pasa con los lenguajes Turing-aceptables? La relación exacta entre los lenguajes generados por una GDC y los lenguajes decidibles o aceptables se verá en el próximo capítulo, pero aquí demostraremos algo relativamente sencillo de ver.

Teorema 4.2 *Sea $G = (V, \Sigma, R, S)$ una GDC. Entonces existe una MTND M tal que, para toda cadena $w \in L$, $(s, \#) \vdash_M^* (h, \#w\#)$.*

Prueba: Similarmente al Teo. 4.1, ponemos todas las reglas en una cinta 2, y $\#S\#$ en la cinta 1. Iterativamente, elegimos una parte izquierda a aplicar de la cinta 2, un lugar donde aplicarla en la cinta 1, y si tal cosa es posible (si no lo es la MTND cicla para siempre), reemplazamos la parte izquierda hallada por la parte derecha. Verificamos que la cinta 1 tenga puros terminales, y si es así nos detenemos. Sino volvemos a buscar una regla a aplicar. \square

El teorema nos dice que una MTND puede, en cierto sentido, “generar” en la cinta cualquier cadena de un lenguaje DC. Profundizaremos lo que ésto significa en el siguiente capítulo.

4.9 Ejercicios

1. Para las siguientes MTs, trace la secuencia de configuraciones a partir de la que se indica, y describa informalmente lo que hacen.
 - (a) $M = (\{q_0, q_1\}, \{a, b, \#\}, \delta, q_0)$, con $\delta(q_0, a) = (q_1, b)$, $\delta(q_0, b) = (q_1, a)$, $\delta(q_0, \#) = (h, \#)$, $\delta(q_1, a) = (q_0, \triangleright)$, $\delta(q_1, b) = (q_0, \triangleright)$, $\delta(q_1, \#) = (q_0, \triangleright)$. Configuración inicial: $(q_0, \underline{a}abbba)$.
 - (b) $M = (\{q_0, q_1, q_2\}, \{a, b, \#\}, \delta, q_0)$, con $\delta(q_0, a) = (q_1, \triangleleft)$, $\delta(q_0, b) = (q_0, \triangleright)$, $\delta(q_0, \#) = (q_0, \triangleright)$, $\delta(q_1, a) = (q_1, \triangleleft)$, $\delta(q_1, b) = (q_2, \triangleright)$, $\delta(q_1, \#) = (q_1, \triangleleft)$, $\delta(q_2, a) = (q_2, \triangleright)$, $\delta(q_2, b) = (q_2, \triangleright)$, $\delta(q_2, \#) = (q_2, \#)$, a partir de $(q_0, \underline{a}bb\#bb\#aba)$.
 - (c) $M = (\{q_0, q_1, q_2\}, \{a, \#\}, \delta, q_0)$, con $\delta(q_0, a) = (q_1, \triangleleft)$, $\delta(q_0, \#) = (q_0, \#)$, $\delta(q_1, a) = (q_2, \#)$, $\delta(q_1, \#) = (q_1, \#)$, $\delta(q_2, a) = (q_2, a)$, $\delta(q_2, \#) = (q_0, \triangleleft)$, a partir de $(q_0, \#a^n\underline{a})$ ($n \geq 0$).
2. Construya una MT que:
 - (a) Busque hacia la izquierda hasta encontrar aa (dos a seguidas) y pare.
 - (b) Decida el lenguaje $\{w \in \{a, b\}^*, w \text{ contiene al menos una } a\}$.
 - (c) Compute $f(w) = ww$.
 - (d) Acepte el lenguaje a^*ba^*b .
 - (e) Decida el lenguaje $\{w \in \{a, b\}^*, w \text{ contiene tantas } a\text{'s como } b\text{'s}\}$.
 - (f) Compute $f(m, n) = m \text{ div } n \text{ y } m \bmod n$.
 - (g) Compute $f(m, n) = m^n$.
 - (h) Compute $f(m, n) = \lfloor \log_m n \rfloor$.
3. Considere una MT donde la cinta es doblemente infinita (en ambos sentidos). Defínala formalmente junto con su operatoria. Luego muestre que se puede simular con una MT normal.
4. Imagine una MT que opere sobre una cinta 2-dimensional, infinita hacia la derecha y hacia arriba. Se decide que la entrada y el resultado quedarán escritos en la primera fila. La máquina puede moverse en las cuatro direcciones. Simule esta máquina para mostrar que no es más potente que una tradicional.
5. Imagine una MT que posea k cabezales pero sobre una misma cinta. En un paso lee los k caracteres, y para cada uno decide qué escribir y adónde moverse. Descríbala formalmente y muestre cómo simularla con un sólo cabezal.

6. Construya MTNDs que realicen las siguientes funciones.

- (a) Acepte $a^*abb^*baa^*$.
- (b) Acepte $\{ww^R, w \in \{a, b\}^*\}$.
- (c) Acepte $\{a^n, \exists p, q \geq 0, n = p^2 + q^2\}$.
- (d) Termine si y sólo si el Teorema de Fermat es verdadero ($\exists x, y, z, n \in \mathbb{N}, n > 2, x, y, z > 0, x^n + y^n = z^n$).

7. Codifique las MTs del Ejercicio 1 usando ρ . Siga las computaciones sugeridas en la versión representada, tal como las haría la MUT.

8. Construya una GDC que:

- (a) Calcule $f(n) = 2^n$.
- (b) Genere $\{a^n b^n c^n, n \geq 0\}$.
- (c) Genere $\{w \in \{a, b, c\}^*, w \text{ tiene más } a\text{'s que } b\text{'s y más } b\text{'s que } c\text{'s}\}$.
- (d) Genere $\{ww, w \in \{a, b\}^*\}$.

4.10 Preguntas de Controles

A continuación se muestran algunos ejercicios de controles de años pasados, para dar una idea de lo que se puede esperar en los próximos. Hemos omitido (i) (casi) repeticiones, (ii) cosas que ahora no se ven, (iii) cosas que ahora se dan como parte de la materia y/o están en los ejercicios anteriores. Por lo mismo a veces los ejercicios se han alterado un poco o se presenta sólo parte de ellos, o se mezclan versiones de ejercicios de distintos años para que no sea repetitivo.

C2 1996 Cuando usamos MT para simular funciones entre naturales, representamos al entero n como I^n . Muestre que también se puede trabajar con los números representados en binario. Suponga que tiene una MT con un alfabeto $\Sigma = \{0, 1\}$ (puede agregar símbolos si lo desea). Siga los siguientes pasos (si no puede hacer alguno suponga que lo hizo y siga con los demás):

- a) Dibuje una MT que sume 1 a su entrada, suponiendo que se siguen las convenciones usuales (el cabezal empieza y termina al final del número), por ejemplo $(s, \#011\underline{\#}) \rightarrow^* (h, \#100\underline{\#})$.
- b) Similarmente dibuje una MT que reste 1 a su entrada si es que ésta no es cero.
- c) Explique cómo utilizaría las dos máquinas anteriores para implementar la suma y diferencia (dando cero cuando el resultado es negativo), por ejemplo en el caso de la suma: $(s, \#011\#101\underline{\#}) \rightarrow^* (h, \#1000\underline{\#})$.

Ex 1996, 2001 Considere los *autómatas de 2 pilas*. Estos son similares a los de una pila, pero pueden actuar sobre las dos pilas a la vez, independientemente. Aceptan una cadena cuando llegan a un estado final, independientemente del contenido de las pilas.

- a)* Defina formalmente este autómata, la noción de configuración, la forma en que se pasa de una configuración a la siguiente, y el lenguaje que acepta.
- b)* Use un autómata de 2 pilas para reconocer el lenguaje $\{a^n b^n c^n, n \geq 0\}$.
- c)* Muestre cómo puede simular el funcionamiento de una MT cualquiera usando un autómata de 2 pilas. Qué demuestra esto acerca del poder de estos autómatas?
- d)* Se incrementa el poder si agregamos más pilas? Por qué?

C2 1997 Diseñe una MT que maneje un conjunto indexado por claves. En la cinta viene una secuencia de operaciones, terminada por #. Cada operación tiene un código (un carácter) y luego vienen los datos. Las operaciones son

- Insertar: El código es *I*, luego viene una clave (secuencia de dígitos) y luego el dato (secuencia de letras entre 'a' y 'z'). Si la clave ya está en el conjunto, reemplazar el dato anterior.
- Borrar: El código es *B*, luego viene una clave. Si la clave no está en el conjunto, ignorar el comando, sino eliminarla.
- Buscar: Viene exactamente una operación de buscar al final de la secuencia, el código es *S* y luego viene una clave. Se debe buscar la clave y si se la encuentra dejar escrito en la cinta el dato correspondiente. Si no se la encuentra se deja vacía la cinta.

Para más comodidad suponga que las claves tienen un largo fijo, y lo mismo con los datos. Mantenga su conjunto en una cinta auxiliar de la forma que le resulte más cómoda. Puede usar extensiones de la MT, como otras cintas, no determinismo, etc., pero el diseño de la MT debe ser detallado.

Ex 1997, 2001 Se propone la siguiente extensión a la MT tradicional: en vez de una sola MT tenemos varias (una cantidad fija) que operan sobre una cinta *compartida*, cada una con su propio cabezal. A cada instrucción, cada máquina lee el carácter que corresponde a su cabezal. Una vez que todas leyeron, cada máquina toma una acción según el carácter leído y su estado actual. La acción puede ser mover su cabezal o escribir en la cinta. Si dos máquinas escriben a la vez en una misma posición puede prevalecer cualquiera de las dos. La máquina para cuando todas paran.

Explique en palabras (pero en detalle) cómo puede simular esta MT extendida usando una MT tradicional (o con alguna extensión vista).

Indique como utilizaría una de estas máquinas para resolver el problema de ordenar una entrada de K caracteres (no necesariamente todos distintos) del alfabeto $\Sigma = \{\sigma_1, \sigma_2, \dots, \sigma_n\}$ y tal que $\sigma_1 < \sigma_2 < \dots < \sigma_n$. Indique el número de MT's originales que la componen y cuál sería su funcionamiento.

C2 1998 Construya (puede usar extensiones) una MT que reciba en la cinta 1 una cadena de la forma $\#u_1\#u_2\#\dots\#u_n\#$ y en la cinta 2 una cadena de la forma $\#v_1\#v_2\#\dots\#v_n\#$. Se supone que todas las cadenas u_i y v_j pertenecen a $\{a, b\}^*$, y que tenemos el mismo número de cadenas en ambas cintas.

La máquina debe determinar si las u_i son un reordenamiento de las v_j o no y detenerse sólo si *no* lo son.

C2 2002 Diseñe una gramática dependiente del contexto que genere el lenguaje $L = \{a^n, n \text{ es un número compuesto}\}$, donde un número compuesto es el producto de dos números mayores que uno.

Ex 2002 La *Máquina Gramatical de Turing* (MGT) se define de la siguiente forma. Recibe $\rho(G)\rho(w)$, donde G es una gramática *dependiente* del contexto y $\rho(G)$ alguna representación razonable de G ; y una cadena w representada mediante $\rho(w)$. La MGT se detiene si y sólo si $w \in L(G)$, es decir acepta el lenguaje $\mathcal{L} = \{\rho(G)\rho(w), w \in L(G)\}$.

Describa la forma de operar de la MGT (detalladamente, pero no necesita dibujar MT's, y si lo hace de todos modos debe explicar qué se supone que está haciendo).

C2 2003 Diseñe una MT que calcule el factorial. Formalmente, la máquina M debe cumplir

$$(s, \#I^n\#) \vdash_M^* (h, \#I^n\#)$$

Puede usar varias cintas y una máquina multiplicadora si lo desea.

4.11 Proyectos

1. Familiarícese con *JTV* (<http://www.dcc.uchile.cl/jtv>) y traduzca algunas de las MTs vistas, tanto MTDs como MTNDs, para simularlas.
2. Dibuje la MUT usando *JTV* y simule algunas ejecuciones.
3. Tome algún lenguaje ensamblador y muestre que las funciones que puede calcular son las mismas que en nuestro modelo de máquina RAM.
4. Investigue sobre funciones recursivas como modelo alternativo de computabilidad. Una fuente es [LP81, sec. 5.3 a 5.6].

5. Investigue sobre lenguajes simples de programación como modelo alternativo de computabilidad. Una fuente es [DW83, cap. 2 a 5]. Esto entra también en el tema del próximo capítulo.

Referencias

- [DW83] M. Davis, E. Weyuker. *Computability, Complexity, and Languages*. Academic Press, 1983.
- [LP81] H. Lewis, C. Papadimitriou. *Elements of the Theory of Computation*. Prentice-Hall, 1981. Existe una segunda edición, bastante parecida, de 1998.

Capítulo 5

Computabilidad

Capítulo 6

Complejidad Computacional

Índice de Materias

Símbolos

- # (en una MT), 74
- $|x|$, para cadenas, 10
- $|$, 13
- \cdot , 13
- Δ (en un AFND), 21
- Δ (en un AP), 49
- Δ (en una MTND), 88
- $\delta(q, a)$ (en un AFD), 17
- $\delta(q, a)$ (en una MT), 74
- ε , 10
- ε (como ER), 14
- $\longrightarrow_G, \longrightarrow$, 42, 101
- Γ (en un AP), 49
- $\Longrightarrow_G, \Longrightarrow$, 42, 101
- $\Longrightarrow_G^*, \Longrightarrow^*$, 42, 102
- $\triangleleft, \triangleright$ (acciones MT), 74, 88
- $\triangleleft, \triangleright$ (MTs modulares), 79
- $\triangleleft_A, \triangleright_A$ (MTs), 80
- λ (para codificar MTs), 95
- $\mathcal{L}(E)$, para ER E , 14
- $\mathcal{L}(G)$, para GDC G , 102
- $\mathcal{L}(G)$, para GLC G , 43
- $\mathcal{L}(M)$, para AFD M , 18
- $\mathcal{L}(M)$, para AP M , 50
- \vdash_M, \vdash
 - para AFDs, 17
 - para AFNDs, 21
 - para APs, 50
 - para MTNDs, 88
 - para MTs, 75
 - para MTs de k cintas, 83
- \vdash_M^*, \vdash^*
 - para AFDs, 17
 - para AFNDs, 21
 - para APs, 50
 - para MTs, 76
- Φ , 13
- $\rho(M)$, 96
- $\rho(w)$, 96
- Σ , 10
 - en un autómata, 17, 21, 49
 - en una gramática, 42, 101
 - en una MT, 74, 88
- Σ^* , 10
- Σ_∞ (para codificar MTs), 95
- \star , 13
- $ap(G)$, para GLC G , 51
- B (MT), 80
- C (MT), 81
- c (para codificar MTs), 95
- $det(M)$, 24
- E (MT), 82
- $E(q)$, 24
- $er(M)$, 27
- F (en un autómata), 17, 21, 49
- G (MT), 89
- $glc(M)$, para AP M , 52
- h (en una MT), 74, 88
- I (para codificar MTs), 95
- I (para representar números), 78
- K (en un autómata), 17, 21, 49
- K (en una MT), 74, 88
- K_∞ (para codificar MTs), 95
- M (MT), 82, 84
- R (en una GDC), 101

- R (en una GLC), 42
- $R(i, j, k)$, 27
- s (en un autómata), 17, 21, 49
- S (en una gramática), 42, 101
- s (en una MT), 74, 88
- $S_{\triangleleft}, S_{\triangleright}$ (MTs), 80, 81
- $S_{i,j}$ (para codificar MTs), 95
- $Th(E)$, 22
- V (en una gramática), 42, 101
- Z (en un AP), 49
- \mathcal{C}_M
 - en un AFD, 17
 - en un AP, 50
 - en una MT, 75
 - en una MT de k cintas, 83
- S,N (para decidir lenguajes), 78
- Definiciones
 - Def. 1.17 (alfabeto Σ), 10
 - Def. 1.18 (cadena, largo, Σ^* , ε), 10
 - Def. 1.20 (prefijo, etc.), 11
 - Def. 1.21 (lenguaje), 11
 - Def. 1.22 (\circ , L^k , L^* , L^c), 11
 - Def. 2.1 (ER), 13
 - Def. 2.2 ($\mathcal{L}(E)$ en ER), 14
 - Def. 2.3 (lenguaje regular), 14
 - Def. 2.4 (AFD), 17
 - Def. 2.5 (configuración AFD), 17
 - Def. 2.6 (\vdash en AFD), 17
 - Def. 2.7 (\vdash^* en AFD), 17
 - Def. 2.8 ($\mathcal{L}(M)$ en AFD), 18
 - Def. 2.9 (AFND), 21
 - Def. 2.10 (\vdash en AFND), 21
 - Def. 2.11 ($\mathcal{L}(M)$ en AFND), 21
 - Def. 2.12 ($Th(E)$), 22
 - Def. 2.13 (clausura- ε), 24
 - Def. 2.14 ($det(M)$), 24
 - Def. 2.15 ($R(i, j, k)$), 27
 - Def. 2.16 ($er(M)$), 27
 - Def. 3.1 (GLC), 42
 - Def. 3.2 (\implies en GLC), 42
 - Def. 3.3 (\implies^* en GLC), 42
 - Def. 3.4 ($\mathcal{L}(G)$ en GLC), 43
 - Def. 3.5 (lenguaje LC), 43
 - Def. 3.6 (árbol derivación), 43
 - Def. 3.7 (GLC ambigua), 44
 - Def. 3.8 (AP), 49
 - Def. 3.9 (configuración AP), 50
 - Def. 3.10 (\vdash_M en AP), 50
 - Def. 3.11 (\vdash_M^* en AP), 50
 - Def. 3.12 ($\mathcal{L}(M)$ en AP), 50
 - Def. 3.13 ($ap(G)$), 51
 - Def. 3.14 (AP simplificado), 52
 - Def. 3.15 ($glc(M)$), 52
 - Def. 3.16 (GLC Chomsky), 59
 - Def. 3.17 (colisión), 59
 - Def. 3.18 (AP determinístico), 59
 - Def. 3.19 ($LL(k)$), 61
 - Def. 3.20 (APLR), 63
 - Def. 3.21 ($LR(k)$), 64
 - Def. 4.1 (MT), 74
 - Def. 4.2 (configuración MT), 75
 - Def. 4.3 (\vdash_M en MT), 75
 - Def. 4.4 (computación), 76
 - Def. 4.5 (función computable), 76
 - Def. 4.6 (función computable en \mathbb{N}), 78
 - Def. 4.7 (lenguaje decidable), 78
 - Def. 4.8 (lenguaje aceptable), 78
 - Def. 4.9 (MTs modulares básicas), 79
 - Def. 4.10 ($\triangleleft_A, \triangleright_A$), 80
 - Def. 4.11 (S_{\triangleleft}), 80
 - Def. 4.12 (S_{\triangleright}), 81
 - Def. 4.13 (MT de k cintas), 83
 - Def. 4.14 (configuración MT k cintas), 83
 - Def. 4.15 (\vdash_M en MT de k cintas), 83
 - Def. 4.16 (uso MT de k cintas), 84
 - Def. 4.17 (MTND), 88
 - Def. 4.18 (MT codificable), 95
 - Def. 4.19 (λ), 95
 - Def. 4.20 ($S_{i,j}$), 95
 - Def. 4.21 ($\rho(M)$), 96
 - Def. 4.22 ($\rho(w)$), 96

Def. 4.23 (MUT), 96
 Def. 4.24 (Tesis de Church), 99
 Def. 4.25 (máquina RAM), 99
 Def. 4.26 (GDC), 101
 Def. 4.27 (\implies en GDC), 101
 Def. 4.28 (\implies^* en GDC), 102
 Def. 4.29 ($\mathcal{L}(G)$ en GDC), 102
 Def. 4.30 (lenguaje DC), 102

Ejemplos

Ej. 2.1, 14
 Ej. 2.2, 15
 Ej. 2.3, 15
 Ej. 2.4, 15
 Ej. 2.5, 15
 Ej. 2.6, 15
 Ej. 2.7, 16
 Ej. 2.8, 16
 Ej. 2.9, 17
 Ej. 2.10, 18
 Ej. 2.11, 18
 Ej. 2.12, 19
 Ej. 2.13, 20
 Ej. 2.14, 21
 Ej. 2.15, 22
 Ej. 2.16, 24
 Ej. 2.17, 25
 Ej. 2.18, 28
 Ej. 2.19, 28
 Ej. 2.20, 29
 Ej. 2.21, 29
 Ej. 2.22, 30
 Ej. 2.23, 31
 Ej. 3.1, 42
 Ej. 3.2, 43
 Ej. 3.3, 44
 Ej. 3.4, 44
 Ej. 3.5, 45
 Ej. 3.6, 46
 Ej. 3.7, 46
 Ej. 3.8, 47
 Ej. 3.9, 47

Ej. 3.10, 47
 Ej. 3.11, 48
 Ej. 3.12, 49
 Ej. 3.13, 50
 Ej. 3.14, 51
 Ej. 3.15, 53
 Ej. 3.16, 54
 Ej. 3.17, 56
 Ej. 3.18, 56
 Ej. 3.19, 60
 Ej. 3.20, 60
 Ej. 3.21, 61
 Ej. 3.22, 62
 Ej. 3.23, 63
 Ej. 3.24, 64
 Ej. 4.1, 74
 Ej. 4.2, 75
 Ej. 4.3, 76
 Ej. 4.4, 77
 Ej. 4.5, 77
 Ej. 4.6, 78
 Ej. 4.7, 81
 Ej. 4.8, 81
 Ej. 4.9, 82
 Ej. 4.10, 82
 Ej. 4.11, 82
 Ej. 4.12, 84
 Ej. 4.13, 88
 Ej. 4.14, 89
 Ej. 4.15, 102
 Ej. 4.16, 102
 Ej. 4.17, 103
 Ej. 4.18, 103

Teoremas

Teo. 2.1 ($ER \rightarrow AFND$), 23
 Teo. 2.2 ($AFND \rightarrow AFD$), 26
 Teo. 2.3 ($AFD \rightarrow ER$), 27
 Teo. 2.4 ($ER \equiv AFD \equiv AFND$), 28
 Teo. 2.5 (Lema de Bombeo), 30
 Teo. 3.1 ($ER \rightarrow GLC$), 46
 Teo. 3.2 ($GLC \rightarrow AP$), 52

- Teo. 3.3 ($AP \rightarrow GLC$), 53
- Teo. 3.4 ($AP \equiv GLC$), 55
- Teo. 3.5 (Teorema de Bombeo), 55
- Teo. 4.1 ($MT \equiv GDC$ funciones), 104
- Teo. 4.1 ($MT \equiv GDC$ lenguajes), 104
- Lemas
 - Lema 2.1 ($R(i, j, k)$), 27
 - Lema 2.2 ($\cup, \circ, *$ regulares), 29
 - Lema 2.3 (\cap y c regulares), 29
 - Lema 2.4 (algoritmos para regulares), 31
 - Lema 3.1 (árbol de derivación), 44
 - Lema 3.2 ($\cup, \circ, *$ LC), 57
 - Lema 3.3 (intersección LCs), 57
 - Lema 3.4 (complemento LC), 57
 - Lema 3.5 ($w \in L$ para LC), 58
 - Lema 3.6 ($L = \emptyset$ para LC), 58
 - Lema 4.1 (MT de k cintas), 87
 - Lema 4.2 (MTND \rightarrow MTD), 94
 - Lema 4.3 ($MT \equiv RAM$), 101
 - Lema 4.4 ($MT \rightarrow GDC$), 103
- acción (de una MT), 74
- aceptable, lenguaje, 78
- AFD, *véase* Autómata finito determinístico
- AFND, *véase* Autómata finito no determinístico
- alfabeto, 10
- ambigüedad (de GLCs), 44, 45
- AP, *véase* Autómata de pila
- arbol de configuraciones, 89
- arbol de derivación, 43
- Autómata
 - conversión de AFD a ER, 27
 - conversión de AFND a AFD, 24
 - conversión de AP a GLC, 52
 - de dos pilas, 69
 - de múltiple entrada, 36
 - de pila, 47, 49
 - de pila determinístico, 57, 59
 - de pila LR (APLR), 63
 - de pila simplificado, 52
 - finito de doble dirección, 39
 - finito determinístico (AFD), 17
 - finito no determinístico (AFND), 21
 - intersección de un AP con un AFD, 57
 - minimización de AFDs, 25, 39
 - para buscar en texto, 22, 26
- Bombeo, lema/teorema de, 30, 55
- borrar una celda (MTs), 74
- cabezal (de una MT), 73
- cadena, 10
- cardinalidad, 7–12
- Chomsky, forma normal de, 59
- Church, tesis de, 98, 99
- cinta (de una MT), 73
- clausura de Kleene, 11
- clausura- ε , 24
- colgarse (una MT), 74, 76
- colisión de reglas, 59
- complemento de un lenguaje, 11
- computable, función, 76, 78
- Computación (de una MT), 76
- computar una función (una MT), 76, 78
- concatenación de lenguajes, 11
- Configuración
 - de un AFD, 17
 - de un AP, 50
 - de una MT, 75, 81
 - de una MT de k cintas, 83
 - de una MTND, 88, 89
 - detenida, 75, 88
- decidible, lenguaje, 78
- detenerse (una MT), 74, 88
- diagonalización, 9
- ER, *véase* Expresión regular
- estados (de un autómata) , 16, 17, 21, 49
 - finales, 16, 17, 21, 49

- inicial, 16, 17, 21, 49
 - sumidero, 16
- estados (de una MT) , 74, 88
 - estado h , 74, 88
 - inicial, 74, 88
- Expresión regular (ER) , 13
 - búsqueda en texto, 26
 - conversión a AFND, 22, 23, 39
- función (Turing-)computable, 76, 78
- GLC, *véase* Gramática libre del contexto
- Glushkov, método de, 39
- Gramática
 - ambigua, 44
 - conversión a $LL(k)$, 61
 - conversión de GLC a AP, 51
 - conversión de GLC a APLR, 63
 - dependiente del contexto (GDC), 101
 - forma normal de Chomsky, 59
 - libre del contexto (GLC), 41, 42
- invariante (de un autómata), 16
- Java Turing Visual (JTV), 73, 100
- Kleene, clausura de, 11
- LC, *véase* Lenguaje libre del contexto
- lenguaje , 11
 - (Turing-)aceptable, 78
 - (Turing-)decidible, 78
 - aceptado por un AFD, 18
 - aceptado por un AFND, 21
 - aceptado por un AP, 50
 - de salida (transductores), 38
 - dependiente del contexto (DC), 102
 - descrito por una ER, 14
 - generado por una GDC, 102
 - generado por una GLC, 43
 - libre del contexto (LC), 41, 43
 - $LL(k)$, 61
 - $LR(k)$, 64
 - regular, 13, 14
- Lleva en cero o más pasos
 - para AFDs, 17
 - para AFNDs, 21
 - para APs, 50
 - para GDCs, 102
 - para GLCs, 42
 - para MTs, 76
- Lleva en un paso
 - para AFDs, 17
 - para AFNDs, 21
 - para APs, 50
 - para GDCs, 101
 - para GLCs, 42
 - para MTNDs, 88
 - para MTs, 75
 - para MTs de k cintas, 83
- Máquina RAM, 99
- Máquina Universal de Turing (MUT), 94, 96
- Máquinas de Turing (MT) , 73, 74
 - buscadoras, 80
 - codificable, 95
 - colgada, 74
 - de k cintas, 83, 84
 - determinísticas (MTD), 88
 - modulares básicas, 79
 - no determinísticas (MTND), 88
 - para operaciones aritméticas, 78, 81, 82, 84
 - shift left, right, 80, 81
 - universal (MUT), 96
- MT, *véase* Máquinas de Turing
- MTD, *véase* Máquinas de Turing determinísticas
- MTND, *véase* Máquinas de Turing no determinísticas
- MUT, *véase* Máquina Universal de Turing
- notación modular (de MTs), 79

- parsing , 41, 59
 - conflictos con shift y reduce, 64
 - top-down, 61
- pila (de un AP) , 47, 49
 - detectar pila vacía, 47, 49
- precedencia, 14, 64
- prefijo, 11
- programa (de máquina RAM), 99
- protocolo (para usar una MT), 76, 84

- RAM, máquina, 99
- reglas de derivación, 41, 42, 101

- símbolo
 - inicial, 42, 101
 - no terminal, 42, 101
 - terminal, 42, 101
- sistema de reescritura, 41
- substring o subcadena, 11
- sufijo, 11

- Tesis de Church, 98, 99
- Thompson, método de, 22
- transductor, 37
- transiciones
 - función de (AFDs), 17
 - función de (MTs), 74
 - para AFDs, 16, 17
 - para AFNDs, 21
 - para APs, 49
 - para MTs, 74, 88
 - transiciones- ε (AFNDs), 20
- Turing-aceptable, lenguaje, 78
- Turing-computable, función, 76, 78
- Turing-decidible, lenguaje, 78