Fundamentos de la Ciencia de la Computación

(Lenguajes Formales, Computabilidad y Complejidad)

Apuntes y Ejercicios

Gonzalo Navarro

Departamento de Ciencias de la Computación
Universidad de Chile
gnavarro@dcc.uchile.cl

9 de noviembre de 2007

Asimismo, agradeceré enviar un email al autor, *gnavarro@dcc.uchile.cl*, si utiliza esta obra fuera del Departamento de Ciencias de la Computación de la Universidad de Chile, para mis registros. Finalmente, toda sugerencia sobre el contenido, errores, omisiones, etc. es bienvenida al mismo email.

Índice General

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 demostraciones 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 *El* conjunto vacío*, denotado* $\emptyset$*, es un conjunto sin elementos.*

Definición 1.3 *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 1.4 *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* $x \in A \wedge y \in B$.

Definición 1.5 *Una* partición *de un conjunto* A *es un conjunto de conjuntos* $B_1, \ldots B_n$ *tal que* $A = \bigcup_{1 \leq i \leq n} B_i$ *y* $B_i \cap B_j = \emptyset$ *para todo* $i \neq j$*.*

Definición 1.6 *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 1.7 *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 1.8 *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 1.9 *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 cuociente.

Definición 1.10 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 1.11 *Una* función $f : A \longrightarrow B$ *es una relación en* $A \times B$ *que cumple que* $\forall a \in A, \exists!\, b \in B,\ afb$. *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 1.12 *Una función* $f : A \longrightarrow B$ *es:*

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

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 1.13 *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 1.14
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* $f : A \longrightarrow B$ *sobreyectiva.*
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 1.15 *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 sobreyección* $f : \mathbb{N} \longrightarrow A$, *o lo que es lo mismo, es posible listar los elementos de* A *en orden* $f(0)$, $f(1)$, $f(2)$, ... *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 sobreyección de $\{a_1,\ a_2, \ldots\}$ 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 1.16 *Si $|A| = \aleph_i$, se llama $\aleph_{i+1} = |\wp(A)|$.*

Definición 1.17 *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; $F(f) = f(0)f(1)f(2)\ldots$ 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 $0.01101\ldots$, para tener la biyección con las secuencias infinitas de bits. *Hay algunas sutilezas debido a que* $0.0011111\ldots = 0.0100000\ldots$*, 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, $\ldots$, 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 $X = \overline{B_1(1)}\ \overline{B_2(2)}\ \ldots$, 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. *$A^+ = A^1 \cup A^2 \cup A^3 \cup \ldots$*

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

1. $a_1,\ b_1,\ a_2,\ b_2, \ldots$ 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 $(a_1, b_1),\ (a_1, b_2),\ (a_1, b_3),\ \ldots$ 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^1 y el 1 de A^2; luego el 3 de A^1, 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 $f : \mathbb{N} \longrightarrow \{0, 1\}$. 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. Comezaremos por definir lo que es un alfabeto.

Definición 1.18 *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 $\{0,1\}$, $\{a,b\}$, $\{a \ldots z\}$, $\{0 \ldots 9\}$, etc., algunas veces usaremos conjuntos más sofisticados como alfabetos.

Definición 1.19 *Llamaremos* cadena *a una secuencia finita de símbolos de un alfabeto* Σ*, es decir, a un elemento de*

$$\Sigma^* \quad = \quad \Sigma^0 \ \cup \ \Sigma^1 \ \cup \ \Sigma^2 \ \cup \ \ldots$$

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* $|\varepsilon| = 0$*). Otro conjunto que usaremos es* $\Sigma^+ = \Sigma^* - \{\varepsilon\}$*.*

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

Definición 1.21 *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 1.22 *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 1.23 *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 $L_3 = \{a, aa\}$. 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. □

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,* $E_1 \star$ *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 "$|$" 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 "$|$". Esto se expresa diciendo que el orden de precedencia es $\star$, $\cdot$, $|$. 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 $a|b|c$, es decir ¿cuál de los dos "$|$" 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 yuxtaponer los símbolos uno después del otro. Notar también que, dado que "$|$" 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 $ba \star ba\star$. 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 empieza 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.

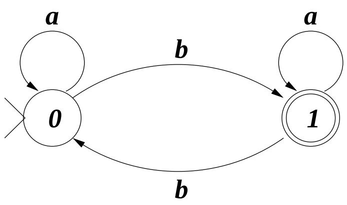
Ejemplo 2.6 Algunas aplicaciones prácticas donde se usan ERs es en la especificación de fechas, direcciones IP, tags XML, nombres de variables en Java, números en notación flotante, direcciones de email, etc. Son ejercicios interesantes, aunque algunos son algo tediosos.

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.

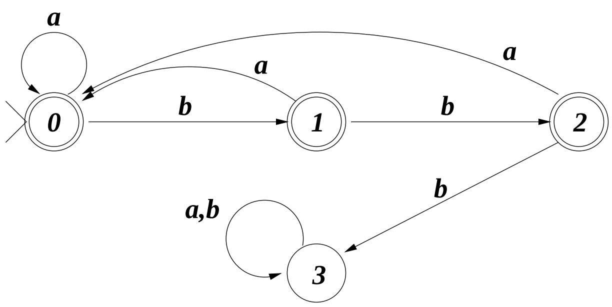


El AFD que hemos dibujado se interpreta de la siguiente manera. Los nodos del grafo son *estados*. El apuntado con un ángulo $\rangle$ es el *estado inicial*, en el que empieza la computación. Estando en un estado, el AFD lee una letra de la entrada y, según indique la flecha (llamada *transición*), 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*

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

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

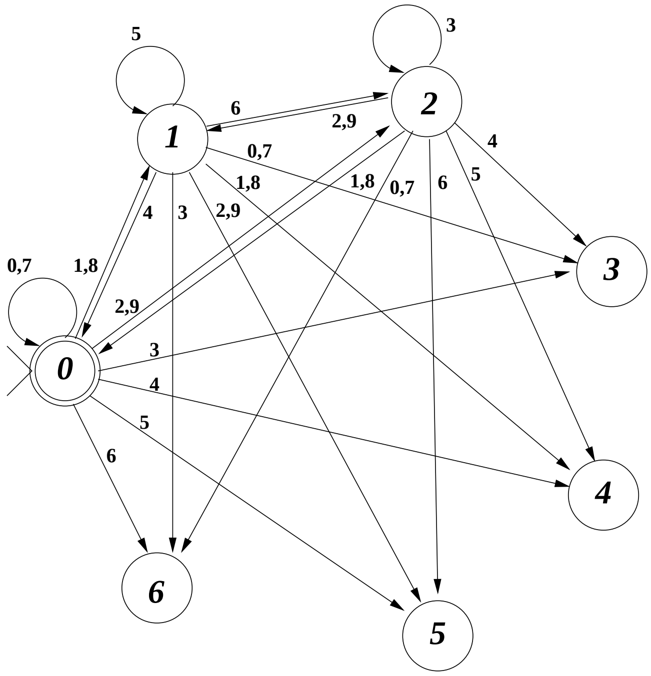
Ahora consideremos la cadena de entrada $x = abbababb$ y escribamos las configuraciones por las que pasa M al recibir x como entrada:

$$\begin{aligned} &(0, abbababb) \vdash (0, bbababb) \vdash (1, bababb) \vdash (2, ababb) \\ &\vdash (0, babb) \vdash (1, abb) \vdash (0, bb) \vdash (1, b) \vdash (2, \varepsilon). \end{aligned}$$

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.

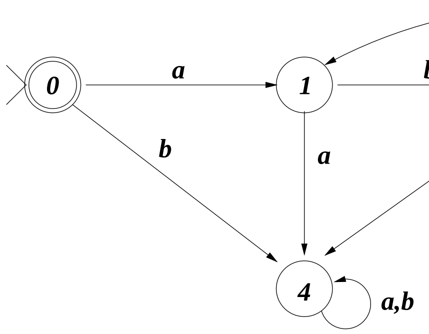
Ejemplo 2.11 El AFD que reconoce el mismo lenguaje del Ej. 2.4 se puede graficar de la siguiente forma. Para no enredar el gráfico de más, sólo se incluyen las flechas que salen de los estados 0, 1 y 2.



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)

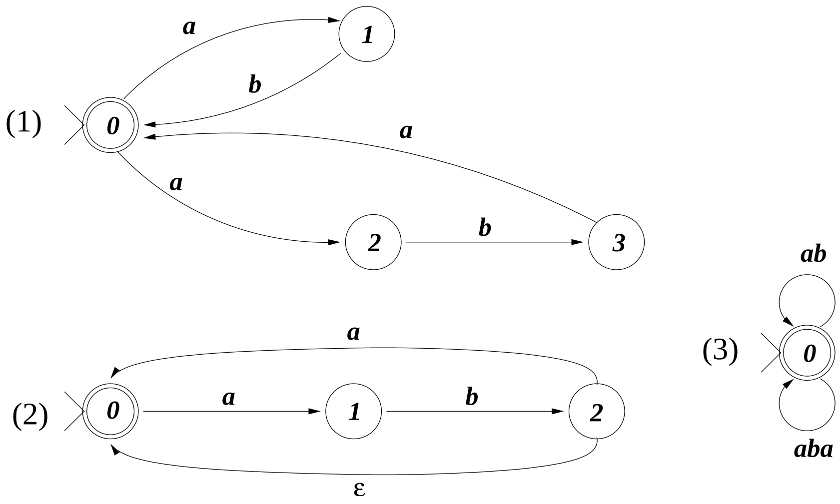
[LP81, sec 2.2]

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* $z \in \Sigma^*$*, sii* $(q, z, q') \in \Delta$.

Nótese que ahora, a partir de una cierta configuración, la relación $\vdash_M$ 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*

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

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).*

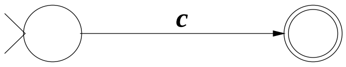
2.4 Conversión de ER a AFND *[LP81, sec 2.5]*

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.*

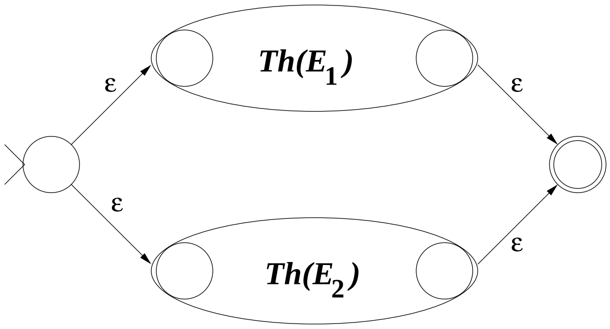
1. *Para $c \in \Sigma$, $Th(c) =$*



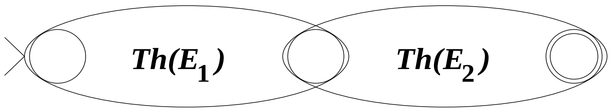
2. *$Th(\Phi) =$*

   ¡Sí, el grafo puede no ser conexo!

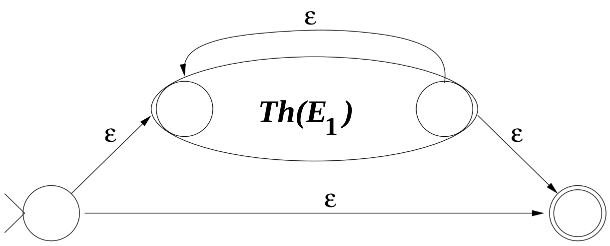
3. *$Th(E_1 \mid E_2) =$*



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.*

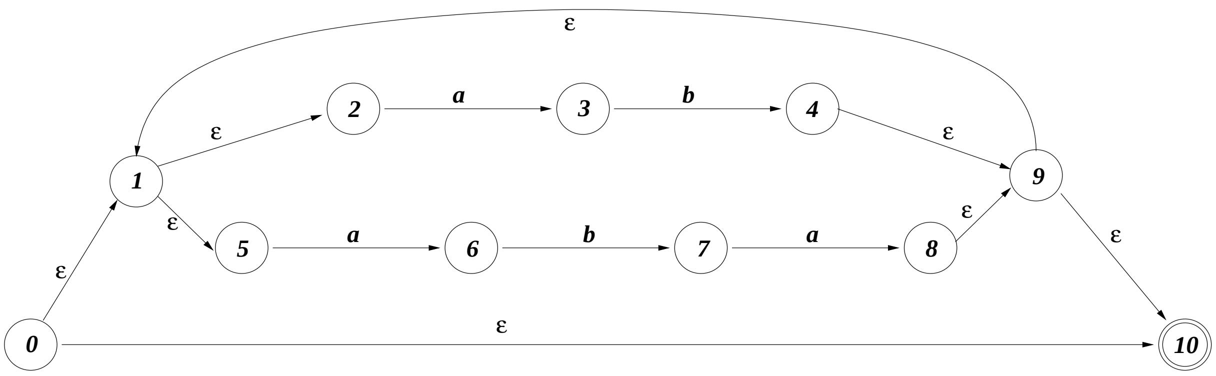
Por simplicidad nos hemos conformado con definir Th usando dibujos esquemáticos. Realmente Th debe definirse formalmente, lo cual el lector puede hacer como ejercicio. Por ejemplo, si $Th(E_1) = (K_1, \Sigma, \Delta_1, s_1, \{f_1\})$ y $Th(E_2) = (K_2, \Sigma, \Delta_2, s_2, \{f_2\})$, entonces $Th(E_1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

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-$\varepsilon$ *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.

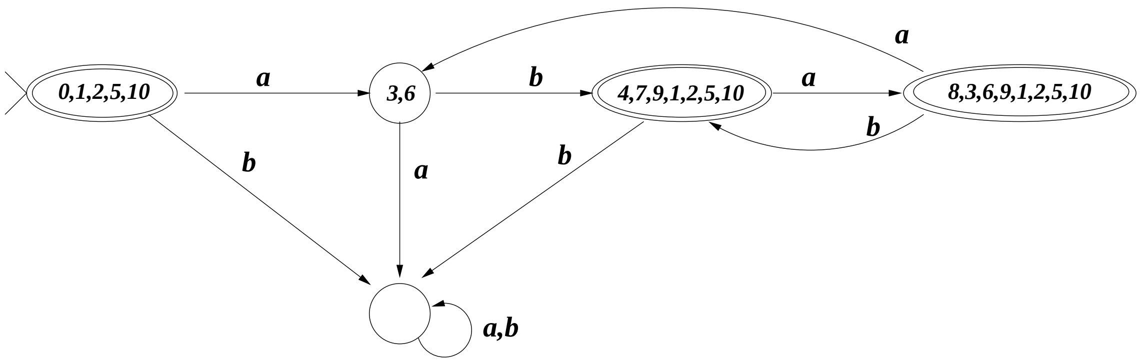
Definición 2.14 *Dado un AFND $M = (K, \Sigma, \Delta, s, F)$ que cumple $(q, x, q') \in \Delta \Rightarrow |x| \leq 1$, se define un AFD $det(M) = (K', \Sigma, \delta, s', F')$ de la siguiente manera:*

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 $c \in \Sigma$,*

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

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 determinizar 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 demostremos (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 $det(Th(\Sigma\star \cdot E))$, 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 $K = \{0, 1, \ldots, n-1\}$ 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. $R(i,j,0) = \begin{cases} \Phi \mid c_1 \mid c_2 \mid \ \ldots \ \mid c_l & si\ \{c_1, c_2, \ldots, c_l\} = \{c \in \Sigma,\ \delta(i,c) = j\}\ e\ i \neq j \\ \varepsilon \mid c_1 \mid c_2 \mid \ \ldots \ \mid c_l & si\ \{c_1, c_2, \ldots, c_l\} = \{c \in \Sigma,\ \delta(i,c) = j\}\ e\ i = j \end{cases}$

2. $R(i, j, k+1) = R(i, j, k) \mid R(i, k, k) \cdot R(k, k, k) \star \cdot R(k, j, k)$.

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 sólamente 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 ($R(k, j, k)$). □

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 $K = \{0, 1, \ldots, n-1\}$ un AFD, y $F = \{f_1, f_2, \ldots, f_m\}$. Entonces definimos la ER*

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

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. □

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

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

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 2.3. □

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 $L1^* = \mathcal{L}(E_1\star)$ 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 $D = 0 \mid 1 \mid \ldots \mid 9$, 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 $((\mathcal{L}(M_7) - \mathcal{L}(M_{11})) \cap \mathcal{L}(E_4)) - \mathcal{L}(0 \cdot D\star)$. ¿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 $\forall n \geq 0,\ xy^n z \in L$.

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 $w_1 w_2 \ldots w_i$:

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

Los estados $q_0,\ q_1, \ldots, q_N$ 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 $y = w_{i+1} w_{i+2} \ldots w_j$ de w, M llegará exactamente al mismo estado final al que llegaba antes:

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

y, similarmente, podemos duplicar y en w tantas veces como queramos y el resultado será el mismo. Llamando $x = w_1 \ldots w_i$, $y = w_{i+1} \ldots w_j$, y $z = w_{j+1} \ldots w_{|w|}$, 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, él 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 $L_2 \subseteq L_1$. $\square$

Observación 2.9 *En estas demostraciones hemos utilizado por primera vez el concepto de* reducción de problemas*: se reduce un problema que no se sabe resolver a uno que sí se sabe. Más adelante usaremos esta idea al revés: reduciremos un problema que sabemos que no se puede resolver a uno que queremos demostrar que no se puede resolver.*

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. Escríbalos formalmente y dibújelos.

   (a) $\{w \in \{a, b\}^*$, cada a en w está precedido y seguido por una $b\}$

   (b) $\{w \in \{a, b\}^*,\ w$ tiene $abab$ como subcadena$\}$

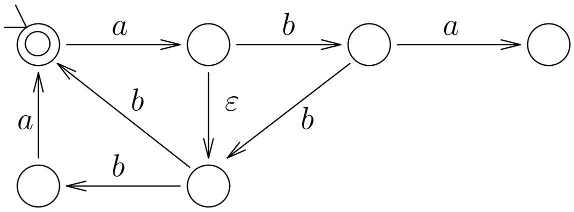
   (c) $\{w \in \{a, b\}^*,\ w$ no tiene aa ni bb como subcadena$\}$

   (d) $\{w \in \{a, b\}^*,\ w$ tiene una cantidad impar de a's y una cantidad par de b's$\}$.

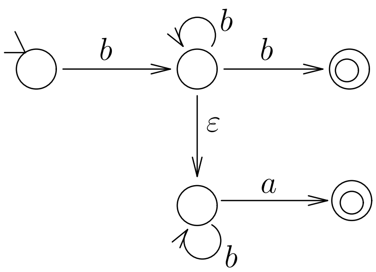
   (e) $\{w \in \{a, b\}^*,\ w$ tiene ab y ba como subcadenas$\}$.

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 $(aa \mid aab \mid aba)\star$.
   (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 $(a \,|\, b) \star aabab$.

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 sóla 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) $\{a^{10^n},\ n \geq 0\}$
   (b) $\{w \in \{0..9\}^*,\ w$ representa 10^n para algún $n \geq 0\}$
   (c) $\{w \in \{0..9\}^*,\ w$ 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) $\{w\overline{w},\ w \in \{a,b\}^*\}$. $\overline{w}$ 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 $\{w \in L,$ ningún prefijo propio de w 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 $\{L_1, L_2, ...\}$ 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 jusitificación 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 `"lolo"`. 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 $p =$`"lolo"`, 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:

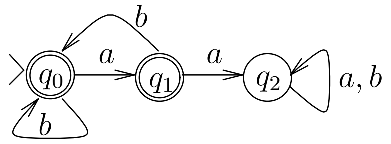
- Dé una expresión regular que lo genere.
- Conviértala a un autómata no determinístico con el método visto. Simplifique el autómata.
- Convierta este autómata simplificado a determinístico con el método visto. Simplifique el autómata obtenido.

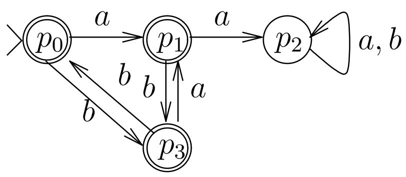
C1 1998, 1999

a) Utilice los métodos vistos para determinar si los siguientes autómatas son o no equivalentes.

b) Exprese el lenguaje aceptado por el autómata de la izquierda como una expresión regular.

c) 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 $a(a \mid ba)\star\ b\star$:

a) Indique todas las palabras de largo 4 que pertenecen al lenguaje representado por esta expresión regular.

b) 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 determínistico. 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) $\{a^n b^m,\ n > m\}$.

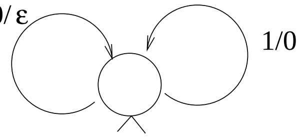
b) $\{a^n b^m a^r,\ r \geq n\}$.

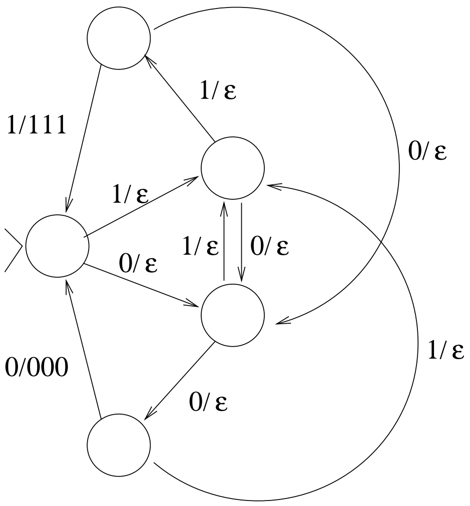
C1 2000 Un *transductor* es una tupla $M = (K, \Sigma, \delta, s)$, donde K es un conjunto finito de estados, Σ es un alfabeto finito, $s \in K$ es el estado inicial y

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

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. 0000111000001111 → 000111000111).





(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 con 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 sii 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.

C1 2006 Sean L_1 y L_2 lenguajes regulares. Se define

$$alt(x_1x_2\ldots x_n, y_1y_2\ldots y_n) = x_1y_1x_2y_2\ldots x_ny_n$$

y se define el lenguaje

$$L \;=\; \{alt(x,y),\ x \in L_1,\ y \in L_2,\ |x| = |y|\}$$

Demuestre que L es regular.

C1 2006 Sea $L \subseteq \{a,b\}^*$ el lenguaje de las cadenas donde todos los bloques de a's tienen el mismo largo (un bloque es una secuencia de a's consecutivas). Por ejemplo $bbbaabaabbaa \in L$, $abbababababba \in L$, $aaaabbaaaabaaaa \in L$, $baabbbaba \notin L$.

Demuestre que L no es regular.

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:

$$\begin{array}{lcl} S & \longrightarrow & aSb \\ S & \longrightarrow & \varepsilon \end{array}$$

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 $\Longrightarrow$ y $\Longrightarrow^*$ 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*

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

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:

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

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:

$$S \Longrightarrow (S)S \Longrightarrow ((S)S)S \Longrightarrow (()S)S \Longrightarrow (())S \Longrightarrow (())(S)S \Longrightarrow (())()S \Longrightarrow (())(),$$

y otra podría ser como sigue:

$$S \Longrightarrow (S)S \Longrightarrow (S)(S)S \Longrightarrow (S)()S \Longrightarrow (S)() \Longrightarrow ((S)S)() \Longrightarrow (()S)() \Longrightarrow (())().$$

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.*

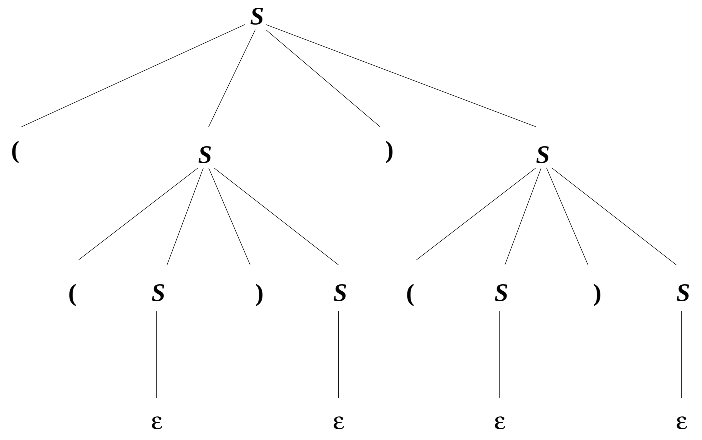
Definición 3.6 *Un* árbol de derivación *para una gramática $G = (V, \Sigma, R, S)$ es un árbol donde los hijos tienen orden y los nodos están rotulados con elementos de V ó Σ ó ε. La raíz está rotulada con S, y los nodos internos deben estar rotulados con elementos de V. Si los rótulos de los hijos de un nodo interno rotulado A son $a_1 \ldots a_k$, $k \geq 1$ y $a_i \in V \cup \Sigma$, debe existir una regla $A \longrightarrow a_1 \ldots a_k \in R$. 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 $S \Longrightarrow_G^* x$. Si $S \Longrightarrow_G^* x$, 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 $w \in \mathcal{L}(G)$.*

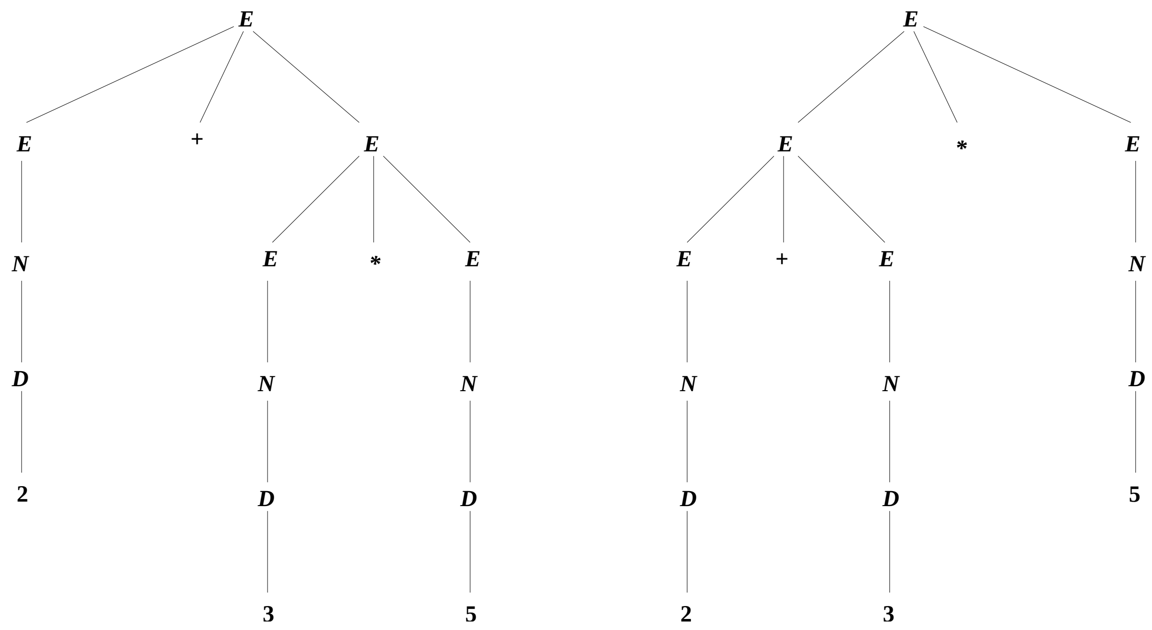
Generalmente ser ambigua es una propiedad indeseable para una GLC. Veamos un ejemplo de una GLC ambigua.

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

$$\begin{array}{lcl@{\qquad\qquad}lcl}
E & \longrightarrow & E+E & N & \longrightarrow & D \\
E & \longrightarrow & E*E & N & \longrightarrow & DN \\
E & \longrightarrow & (E) & D & \longrightarrow & 0 \\
E & \longrightarrow & N & D & \longrightarrow & \ldots \\
 & & & D & \longrightarrow & 9
\end{array}$$

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 arbol utilicemos para generarla.

Cuando se tiene una gramática ambigua, podemos intentar *desambiguarla*, mediante escribir otra que genere el mismo lenguaje pero que 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 $*$, $+$.

$$
\begin{array}{lcl@{\qquad\qquad}lcl}
E & \longrightarrow & E+T & N & \longrightarrow & D \\
E & \longrightarrow & T & N & \longrightarrow & DN \\
T & \longrightarrow & T*F & D & \longrightarrow & 0 \\
T & \longrightarrow & F & D & \longrightarrow & \ldots \\
F & \longrightarrow & (E) & D & \longrightarrow & 9 \\
F & \longrightarrow & N & & &
\end{array}
$$

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 $G = (\{S\}, \Sigma, \{S \longrightarrow a\}, S)$ 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 $G = (V_1 \cup V_2 \cup \{S\}, \Sigma, R_1 \cup R_2 \cup \{S \longrightarrow S_1, S \longrightarrow S_2\}, S)$ 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 $G = (V_1 \cup V_2 \cup \{S\}, \Sigma, R_1 \cup R_2 \cup \{S \longrightarrow S_1 S_2\}, S)$ 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 $G = (V_1 \cup \{S\}, \Sigma, R_1 \cup \{S \longrightarrow S_1 S, S \longrightarrow \varepsilon\}, S)$ genera L.

$\square$

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

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

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.

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

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

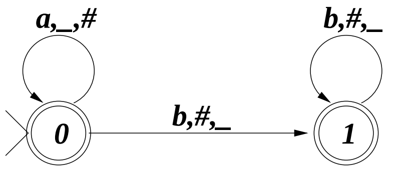
Ejemplo 3.8 Tal como con ERs, no siempre es fácil diseñar una GLC que genere cierto lenguaje. Un ejercicio interesante es $\{w \in \{a,b\}^*, \; w$ tiene la misma cantidad de a's y b's $\}$. Una respuesta sorprendentemente sencilla es $S \longrightarrow \varepsilon \mid aSbS \mid bSaS$. Es fácil ver 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.

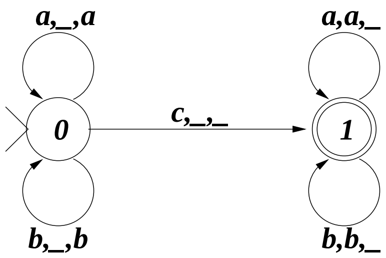
Ejemplo 3.9 Consideremos el siguiente AP:



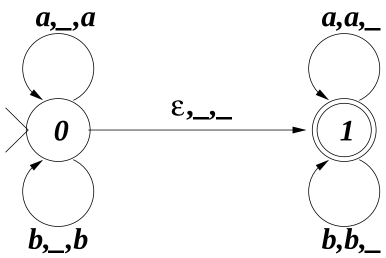
Las transiciones se leen de la siguiente manera: $(a, \_, \#)$ significa que, al leerse una a, independientemente de los caracteres que haya en el tope de la pila (_), se apilará un símbolo # y se seguirá la transición; $(b, \#, \_)$ significa que, al leerse una b, si hay un símbolo # en el tope de la 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 #:



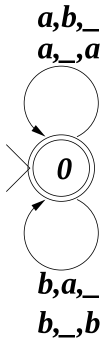
Una pregunta un poco más complicada es: ¿cómo aceptar $\{ww^R,\ w \in \{a,b\}^*\}$? ¡Esta vez no tenemos una marca (c) que nos indique cuándo empezar a desapilar! La solución se basa en explotar el no determinismo intrínseco de los APs:

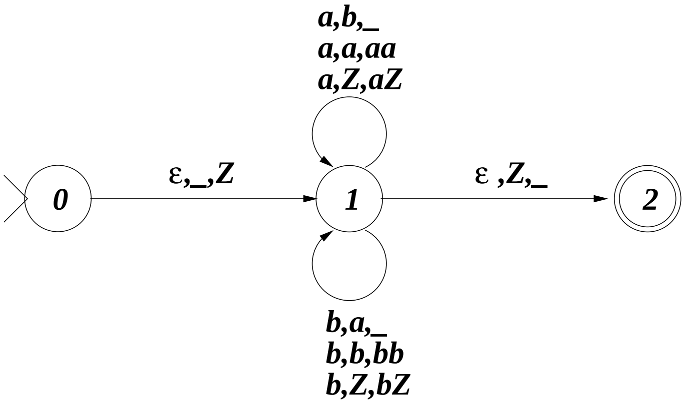


De este modo el AP *adivina* qué transición elegir (es decir cuándo empezar a desapilar), tal como lo hacían los AFNDs.

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*

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

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]*

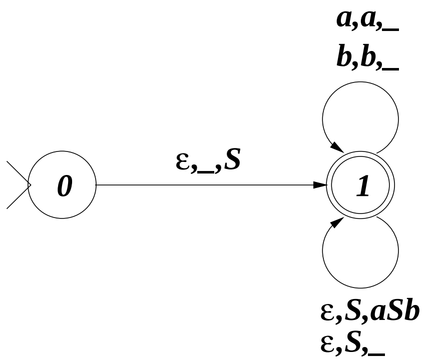
Vamos a demostrar ahora que ambos mecanismos, GLCs y APs, son equivalentes para denotar lenguajes libres del contexto. Comenzaremos con la conversión más sencilla.

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*

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

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 $S \Longrightarrow^*_G x\gamma$. 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 $\Longrightarrow_G$. 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 $\Longrightarrow_G^*$ 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 $((p, x, a_1a_2a_3), (q, \beta))$ 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.*

$$
\begin{aligned}
R &= \{S \longrightarrow \langle s,\varepsilon,f\rangle,\ f\in F\}\\
&\cup\ \{\langle p,\varepsilon,p\rangle \longrightarrow \varepsilon,\ p\in K\}\\
&\cup\ \{\langle p,A,r\rangle \longrightarrow x\langle q,\varepsilon,r\rangle,\ ((p,x,A),(q,\varepsilon))\in\Delta,\ r\in K\}\\
&\cup\ \{\langle p,\varepsilon,r\rangle \longrightarrow x\langle q,\varepsilon,r\rangle,\ ((p,x,\varepsilon),(q,\varepsilon))\in\Delta,\ r\in K\}\\
&\cup\ \{\langle p,A,r\rangle \longrightarrow x\langle q,A,r\rangle,\ ((p,x,\varepsilon),(q,\varepsilon))\in\Delta,\ r\in K,\ A\in\Gamma\}\\
&\cup\ \{\langle p,A,r\rangle \longrightarrow x\langle q,B_1,r_1\rangle\langle r_1,B_2,r_2\rangle\ldots\langle r_{k-1},B_k,r\rangle,\\
&\qquad\qquad ((p,x,A),(q,B_1B_2\ldots B_k))\in\Delta,\ r_1,r_2,\ldots,r_{k-1},r\in K\}\\
&\cup\ \{\langle p,\varepsilon,r\rangle \longrightarrow x\langle q,B_1,r_1\rangle\langle r_1,B_2,r_2\rangle\ldots\langle r_{k-1},B_k,r\rangle,\\
&\qquad\qquad ((p,x,\varepsilon),(q,B_1B_2\ldots B_k))\in\Delta,\ r_1,r_2,\ldots,r_{k-1},r\in K\}\\
&\cup\ \{\langle p,A,r\rangle \longrightarrow x\langle q,B_1,r_1\rangle\langle r_1,B_2,r_2\rangle\ldots\langle r_{k-1},B_k,r_k\rangle\langle r_k,A,r\rangle,\\
&\qquad\qquad ((p,x,\varepsilon),(q,B_1B_2\ldots B_k))\in\Delta,\ r_1,r_2,\ldots,r_k,r\in K\}
\end{aligned}
$$

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 lineas 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 $B_1B_2\ldots B_k$. 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]. □

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

$$
\begin{array}{llll}
S & \longrightarrow & \langle 0,\varepsilon,0\rangle & \text{primera línea Def. 3.15} \\
\langle 0,\varepsilon,0\rangle & \longrightarrow & \varepsilon & \text{segunda línea Def. 3.15} \\
\langle 0,b,0\rangle & \longrightarrow & a\langle 0,\varepsilon,0\rangle & \text{generada por } ((0,a,b),(0,\varepsilon)) \\
\langle 0,\varepsilon,0\rangle & \longrightarrow & a\langle 0,a,0\rangle & \text{generadas por } ((0,a,\varepsilon),(0,a)) \\
\langle 0,a,0\rangle & \longrightarrow & a\langle 0,a,0\rangle\langle 0,a,0\rangle & \\
\langle 0,b,0\rangle & \longrightarrow & a\langle 0,a,0\rangle\langle 0,b,0\rangle & \\
\langle 0,a,0\rangle & \longrightarrow & b\langle 0,\varepsilon,0\rangle & \text{generada por } ((0,b,a),(0,\varepsilon)) \\
\langle 0,\varepsilon,0\rangle & \longrightarrow & b\langle 0,b,0\rangle & \text{generadas por } ((0,b,\varepsilon),(0,b)) \\
\langle 0,b,0\rangle & \longrightarrow & b\langle 0,b,0\rangle\langle 0,b,0\rangle & \\
\langle 0,a,0\rangle & \longrightarrow & b\langle 0,b,0\rangle\langle 0,a,0\rangle &
\end{array}
$$

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.

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

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.

$$
\begin{array}{llll}
S & \longrightarrow & \langle 0,\varepsilon,0\rangle & \text{primera línea Def. 3.15} \\
S & \longrightarrow & \langle 0,\varepsilon,1\rangle & \\
\langle 0,\varepsilon,0\rangle & \longrightarrow & \varepsilon & \text{segunda línea Def. 3.15} \\
\langle 1,\varepsilon,1\rangle & \longrightarrow & \varepsilon & \\
\langle 0,\varepsilon,0\rangle & \longrightarrow & a\langle 0,\#,0\rangle & \text{generadas por } ((0,a,\varepsilon),(0,\#)) \\
\langle 0,\varepsilon,1\rangle & \longrightarrow & a\langle 0,\#,1\rangle & \\
\langle 0,\#,0\rangle & \longrightarrow & a\langle 0,\#,0\rangle\langle 0,\#,0\rangle & \\
\langle 0,\#,0\rangle & \longrightarrow & a\langle 0,\#,1\rangle\langle 1,\#,0\rangle & \\
\langle 0,\#,1\rangle & \longrightarrow & a\langle 0,\#,0\rangle\langle 0,\#,1\rangle & \\
\langle 0,\#,1\rangle & \longrightarrow & a\langle 0,\#,1\rangle\langle 1,\#,1\rangle & \\
\langle 0,\#,0\rangle & \longrightarrow & b\langle 1,\varepsilon,0\rangle & \text{generadas por } ((0,b,\#),(1,\varepsilon)) \\
\langle 0,\#,1\rangle & \longrightarrow & b\langle 1,\varepsilon,1\rangle & \\
\langle 1,\#,0\rangle & \longrightarrow & b\langle 1,\varepsilon,0\rangle & \text{generadas por } ((1,b,\#),(1,\varepsilon)) \\
\langle 1,\#,1\rangle & \longrightarrow & b\langle 1,\varepsilon,1\rangle &
\end{array}
$$

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. □

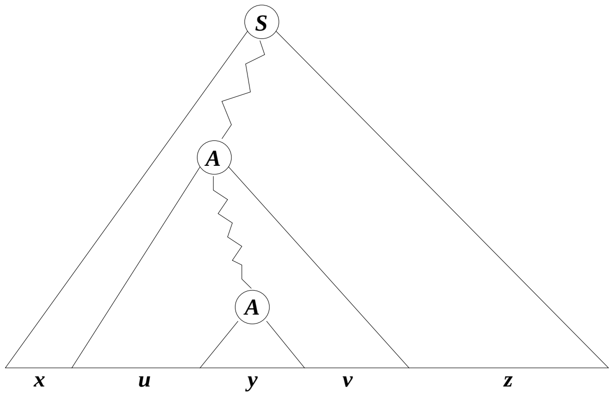
3.6 Teorema de Bombeo *[LP81, sec 3.5.2]*

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.

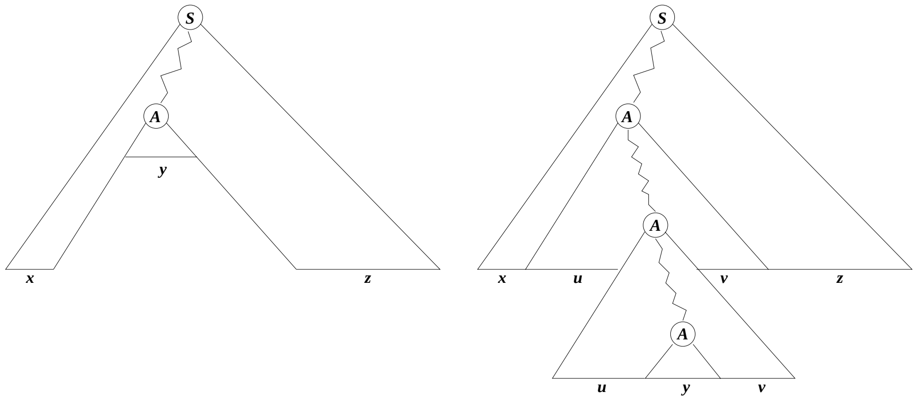
Teorema 3.5 (Teorema de Bombeo)

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 $\forall n \geq 0,\ xu^nyv^nz \in L$.

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 $p = \max\{|z|,\ A \longrightarrow z \in R\}$ 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 $A \Longrightarrow^* uyv$ y la segunda vez $A \Longrightarrow^* y$. Pero podríamos haber elegido $A \Longrightarrow^* y$ la primera vez. O podríamos haber elegido $A \Longrightarrow^* uyv$ la segunda vez. En general podríamos haber generado $S \Longrightarrow^* xu^nyv^nz$ 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. $\Box$

Ejemplo 3.17 Usemos el Teorema de Bombeo para demostrar que $L = \{a^nb^nc^n,\ n \geq 0\}$ no es LC. Dado el N, elegimos $w = a^Nb^Nc^N$. 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 $w = a^Nb^Na^Nb^N \in L$. 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. □

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. □

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. □

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*

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

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 $A \longrightarrow \varepsilon$, pues reducen el largo de la cadena derivada. Para poder eliminar esta regla, buscaremos todas las producciones de la forma $B \longrightarrow xAy$ y *agregaremos* otra regla $B \longrightarrow xy$ a G, adelantándonos al posible uso de $A \longrightarrow \varepsilon$ en una derivación. Cuando hayamos hecho esto con todas las reglas que mencionen A en la parte derecha, podemos descartar la regla $A \longrightarrow \varepsilon$. Nótese que no es necesario reprocesar las nuevas reglas introducidas según viejas reglas $A \longrightarrow \varepsilon$ ya descartadas, pues la regla paralela correspondiente ya existe. Lo que sí merece atención es que pueden aparecer nuevas reglas de la forma $B \longrightarrow \varepsilon$, 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 $S \longrightarrow \varepsilon$ 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 $S \longrightarrow \varepsilon$ ya que no es relevante para otras producciones.

2. Reglas de la forma $A \longrightarrow B$, 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 $A \Longrightarrow^* B$ sii existe un camino de A a B en ese grafo. Para cada uno de estos caminos, tomamos todas las producciones de tipo $C \longrightarrow xAy$ y agregamos $C \longrightarrow xBy$, adelantándonos al posible uso de esas flechas. Cuando hemos agregado todas las reglas nuevas, eliminamos en bloque todas las reglas de tipo $A \longrightarrow B$. Nuevamente, no podemos eliminar directamente las reglas de tipo $S \longrightarrow A$, 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: $A \longrightarrow BC$, $A \longrightarrow a$, y $S \longrightarrow \varepsilon$, 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, $w_i w_{i+1} \ldots w_j$, qué no terminales A derivan ese substring, $A \Longrightarrow^* w_i w_{i+1} \ldots w_j$. Para determinar esto se prueba, para cada regla $A \longrightarrow BC$, si existe un k tal que $B \Longrightarrow^* w_i \ldots w_k$ y $C \Longrightarrow^* w_{k+1} \ldots w_j$.*

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 $L = \{a^{m_1} b a^{m_2} b \ldots b a^{m_k},\ k \geq 2,\ m_1, m_1, \ldots, m_k \geq 0,\ m_i \neq m_j$ 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 $((0,b,\varepsilon),(0,b))$.

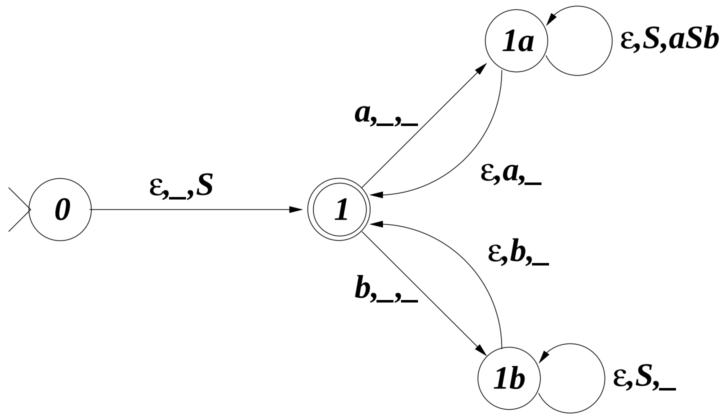
Lo que verdaderamente nos interesa es: ¿es posible diseñar un método para generar un AP determinístico a partir de una GLC? Ya sabemos que tal método no puede funcionar siempre, pero podemos aspirar a poderlo hacer en una buena cantidad de casos "interesantes".

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 $N \longrightarrow D$ y $N \longrightarrow DN$ 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 $N \longrightarrow DN'$, $N' \longrightarrow \varepsilon$, $N' \longrightarrow N$. 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 $E \longrightarrow E+T$ y $E \longrightarrow T$ (basta que siga una cadena de T de largo mayor que k). Este problema también puede resolverse, mediante reemplazar las reglas por $E \longrightarrow TE'$, $E' \longrightarrow \varepsilon$, $E' \longrightarrow +TE'$. En general, si tenemos reglas de la forma $A \longrightarrow A\alpha_i$ y otras $A \longrightarrow \beta_j$, las podemos reemplazar por $A \longrightarrow \beta_j A'$, $A' \longrightarrow \alpha_i A'$, $A' \longrightarrow \varepsilon$.

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.

$$
\begin{array}{lcl@{\qquad\qquad}lcl}
E & \longrightarrow & TE' & N & \longrightarrow & DN' \\
E' & \longrightarrow & +TE' & N' & \longrightarrow & N \\
E' & \longrightarrow & \varepsilon & N' & \longrightarrow & \varepsilon \\
T & \longrightarrow & FT' & D & \longrightarrow & 0 \\
T' & \longrightarrow & *FT' & D & \longrightarrow & \ldots \\
T' & \longrightarrow & \varepsilon & D & \longrightarrow & 9 \\
F & \longrightarrow & (E) & & & \\
F & \longrightarrow & N & & &
\end{array}
$$

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 $E' \longrightarrow \varepsilon$. 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 $E' \longrightarrow \varepsilon$ 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.

Parse$E$
if $\neg$ Parse$T$ **return** $false$
if $\neg$ ParseE' **return** $false$
return $true$

ParseE'
if $nextChar() = +$ **return** ParseE'_1
return ParseE'_2

ParseE'_1
$getChar()$;
if $\neg$ Parse$T$ **return** $false$
if $\neg$ ParseE' **return** $false$
return $true$

ParseE'_2
return $true$

... Parse$T$ y ParseT' muy similares

Parse$F$
if $nextChar() =$ (**return** ParseF_1
return ParseF_2

ParseF_1
$getChar()$;
if $\neg$ Parse$E$ **return** $false$
if $nextChar() \neq$) **return** $false$
$getChar()$;
return $true$

ParseF_2
return Parse$N$

Parse$N$
if $\neg$ Parse$D$ **return** $false$
if $\neg$ ParseN' **return** $false$
return $true$

ParseN'
if $nextChar() \in \{0 \ldots 9\}$ **return** ParseN'_1
return ParseN'_2

ParseN'_1
return Parse$N$

ParseN'_2
return $true$

Parse$D$
if $nextChar() = 0$ **return** ParseD_0
...
if $nextChar() = 9$ **return** ParseD_9

ParseD_0
$getChar()$
return $true$

... ParseD_1 a ParseD_9 similares

Los procedimientos pueden simplificarse a mano significativamente, pero la intención es enfatizar cómo salen prácticamente en forma automática de la GLC. Parse$E$ 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. ParseE', por ejemplo, supone que se le puede dar la prioridad a ParseE'_1, pues verifica directamente el primer carácter, y si no funciona sigue con ParseE'_2. Esto no funcionaría si el $+$ pudiera eventualmente aparecer siguiendo E' en la derivación.

Parsing Bottom-Up: Lenguajes 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 LR(k) es más flexible. La idea esta vez es construir el árbol de parsing de abajo hacia arriba.

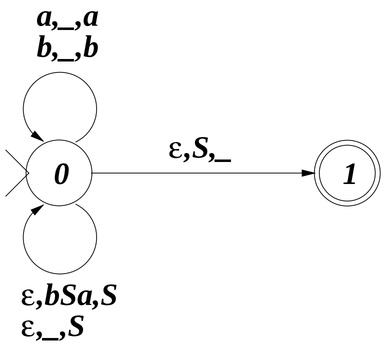
La idea de este parsing es que la pila contiene lo que el parser ha visto de la entrada, no lo que espera ver. Lo que se ha visto se expresa como una secuencia de terminales y no terminales. En cada momento se puede elegir entre apilar la primera letra de la entrada (con lo que ya la "hemos visto") o 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 LR(k). Los generadores profesionales de parsers usan el mecanismo LR(k), pues es más potente.

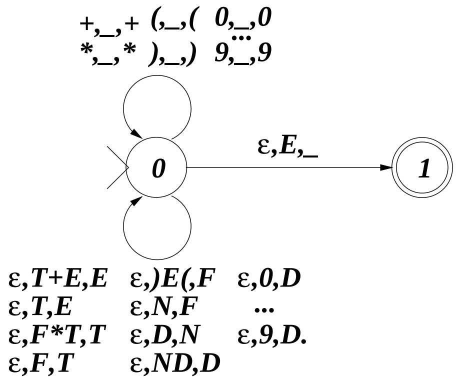
Ejemplo 3.23 Dibujemos el APLR para la GLC del Ej. 3.14. Notar que no es determinístico.



Nuevamente el APLR puede no ser determinístico, y se intenta determinizar considerando el siguiente carácter de la entrada. Surgen dos tipos de colisiones entre reglas (llamadas "conflictos"): *shift/reduce* cuando existe colisión entre una regla que indica apilar el siguiente símbolo de la entrada (*shift*) versus otra que indica transformar una parte derecha por una izquierda en la pila (*reduce*); y *reduce/reduce* cuando hay dos formas de reducir partes derechas a izquierdas en la pila.

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.*

Ejemplo 3.24 Tomemos la GLC del Ej. 3.5 y construyamos el APLR correspondiente:



Sigamos un parsing exitoso de $2 + 3 * 5$:

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

El ejemplo muestra la relevancia de los conflictos. Por ejemplo, si en el segundo paso, en vez de reducir $D \longrightarrow 2$ (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 $E \longrightarrow T$, habríamos fracasado. En cambio, en el paso 13, debemos apilar $*$ en vez de usar la regla $E \longrightarrow T$. 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 $R = \{S \longrightarrow AA, A \longrightarrow AAA, A \longrightarrow a, A \longrightarrow bA, A \longrightarrow Ab\}$.

(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 $R = \{S \longrightarrow aAa, S \longrightarrow bAb, S \longrightarrow \varepsilon, A \longrightarrow SS\}$.

   (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 $R = \{S \longrightarrow aSa, S \longrightarrow aSb, S \longrightarrow bSa, S \longrightarrow bSb, S \longrightarrow \varepsilon\}$ 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 $R = \{S \longrightarrow \varepsilon, S \longrightarrow SS, S \longrightarrow [S], S \longrightarrow (S)\}$.

   (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 $R = \{S \longrightarrow abA, S \longrightarrow B, S \longrightarrow baB, S \longrightarrow \varepsilon, A \longrightarrow bS, B \longrightarrow aS, A \longrightarrow b\}$. 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. Descríbalo 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

$$L \;=\; \{w_1w_2w_3w_4,\ (w_3 = w_1^R \ \vee\ w_2 = w_4^R) \ \wedge\ |w_1w_2w_3w_4| \text{ par}\}$$

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

a) $L \;=\; \{a^nb^ma^n,\ m < n\}$

b) $L \;=\; \{a^nb^mc^rd^s,\ 0 \leq n \leq m \leq r,\ 0 \leq s\}$

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 $(A\ or B)\ and\ C$ 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:

a) Demuestre que el lenguaje $L = \{\ a^ib^jc^id^j,\ i,j \geq 0\}$ no es LC.

b) 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 usan 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 $a^n b^n c^n d^n e^n$?

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

C1 2006 Demuestre que los siguientes lenguajes son libres del contexto.

1. $L = \{ww^R, \ w \in L'\}$, donde L' es un lenguaje regular.
2. $L = \{w \in \{a,b\}^*, \ w \neq w^R\}$ (w^R es w leído al revés).

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 solamente 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 que podamos hacer en nuestro PC. 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 supone que éste contiene un *blanco* (#). Como los autómatas, la MT está en

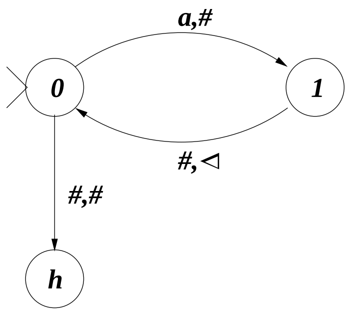
[1] Esta herramienta se desarrolló en el DCC, por mi alumno Marco Mora Godoy, en su Memoria de Ingeniería. Si se la usa debe tenerse en cuenta que no es del todo estable, y a veces tiende a borrar los archivos que ha grabado. Funciona mejor con `jdk 1.4`.

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 $\lhd$ y $\rhd$, 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 #. El alfabeto es $\{a, \#\}$.



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 no 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
- $\delta : K \times \Sigma \longrightarrow (K \cup \{h\}) \times (\Sigma \cup \{\lhd, \rhd\})$*,* $\lhd, \rhd \notin \Sigma$*, 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,$\lhd$

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* $b = \lhd$ *(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* $b = \rhd$ *(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 $C_0 \vdash_M C_1 \vdash_M \ldots \vdash_M C_n$.*

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

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

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 $\# \not\in \Sigma_0 \cup \Sigma_1$. 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, $f(w_1, w_2, \ldots, w_k)$, donde la MT debe operar de la siguiente forma:

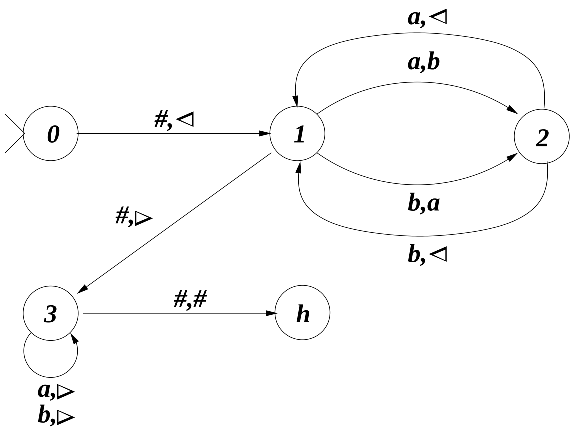
$$(s, \#w_1\#w_2\# \ldots \#w_k\underline{\#}) \vdash_M^* (h, \#f(w_1, w_2, \ldots, w_k)\underline{\#}).$$

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.

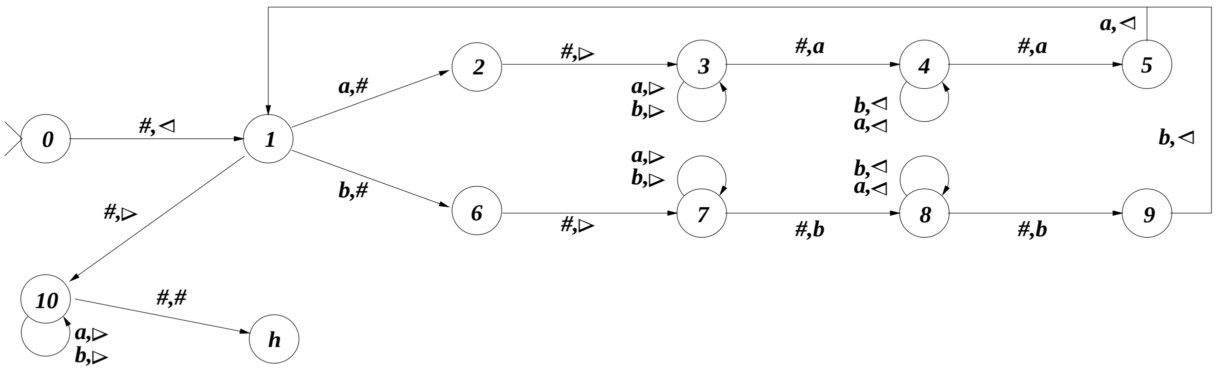
Observación 4.2 *Una observación muy importante sobre la Def. 4.5 es que M no se cuelga frente a ninguna entrada, sino que siempre llega a h. Esto significa que jamás intenta moverse hacia la izquierda del primer #. Por lo tanto, si sabemos que M calcula f, podemos con confianza aplicarla sobre una cinta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

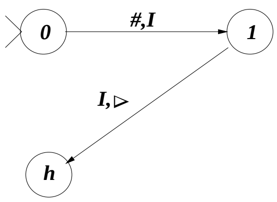


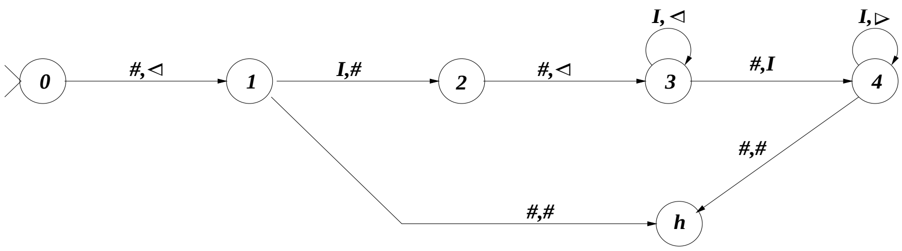
Es un ejercicio interesante derivar $f(w) = ww$ a partir de este ejemplo.

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* $f : \mathbb{N} \longrightarrow \mathbb{N}$. *Decimos que una MT* M computa f *si* M *computa (según la Def. 4.5)* $g : \{I\}^* \longrightarrow \{I\}^*$ *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 $\#\#I^m\underline{\#}$).

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 $f_L\ :\ \Sigma^* \longrightarrow \{\mathtt{S},\mathtt{N}\}$, definida como $f_L(w) = \mathtt{S} \Leftrightarrow w \in L$ (y si no, $f_L(w) = \mathtt{N}$). 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 $\Sigma_1 = \{\mathtt{S},\mathtt{N}\}$ (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. 1.17).*

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 $(s, \#w\underline{\#}) \vdash_M^* (h, u\underline{a}v) \ \Leftrightarrow\ w \in L$. 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 ($\lhd$): Esta es una MT que, pase lo que pase, se mueve hacia la izquierda una casilla y se detiene.* $\lhd = (\{s\}, \Sigma, \delta, s)$, *donde* $\forall a \in \Sigma,\ \delta(s, a) = (h, \lhd)$. *(Notar que estamos sobrecargando el símbolo $\lhd$, pero no debería haber confusión.)*

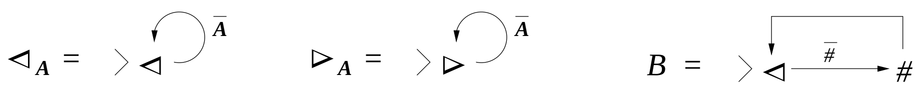
- *Moverse hacia la derecha ($\rhd$): Esta es una MT que, pase lo que pase, se mueve hacia la derecha una casilla y se detiene.* $\rhd = (\{s\}, \Sigma, \delta, s)$, *donde* $\forall a \in \Sigma,\ \delta(s, a) = (h, \rhd)$.

- *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 simplificarnos la vida. Se puede ver en el libro la demostración.*

Dos MTs sumamente útiles como acciones son $\lhd_A$ y $\rhd_A$, 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 $\lhd_A$, $\rhd_A$ y B se definen según el siguiente diagrama:*



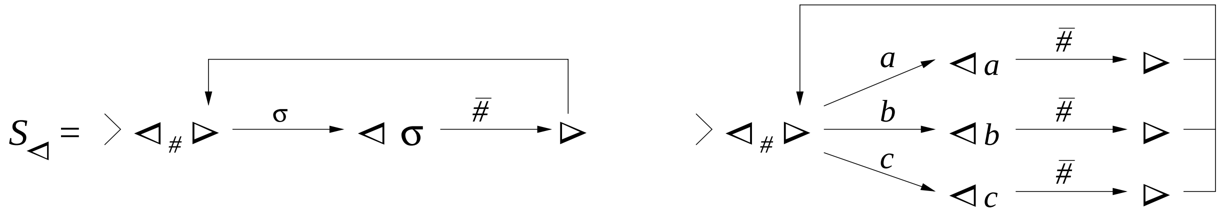
La máquina $\lhd_A$, por ejemplo, comienza moviéndose a la izquierda. Luego, si el carácter sobre el que está parada no está en A, vuelve a moverse, y así. Si, en cierto momento, queda sobre un carácter de A, entonces luego de ejecutar el nodo no tiene flecha aplicable que seguir, y se detiene.

Nótese que $\lhd_A$ y $\rhd_A$ *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 $\lhd_a$ y $\rhd_a$.

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 *demos 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 ($S_\lhd$) *se define de según el siguiente diagrama (parte izquierda).*



La máquina $S_\lhd$ 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 $A \to B \to C$. 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 $S_\lhd$ se expandiría en máquinas distintas según Σ. En realidad, lo mismo ocurre con las máquinas básicas.

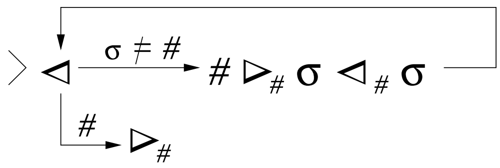
La máquina $S_\lhd$, entonces, está pensada para ser arrancada en una configuración tipo $(s, X\#w\underline{\#})$ (donde w no contiene blancos), y termina en la configuración $(h, Xw\underline{\#})$. Es decir, toma su argumento y lo mueve una casilla a la izquierda. Esto no cae dentro del formalismo de calcular funciones, pues $S_\lhd$ 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 $S_{\lhd} \ : \ \#w\underline{\#} \longrightarrow w\underline{\#}$. Nuevamente, como $S_{\lhd}$ 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.* $S_{\rhd} \ : \ \#w\underline{\#} \longrightarrow \#\#w\underline{\#}$.

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

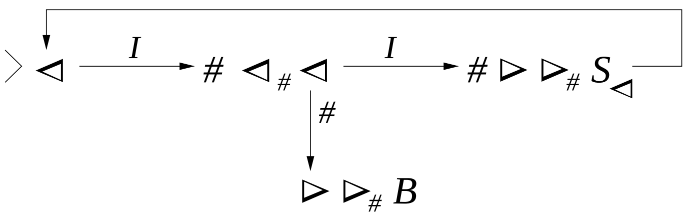
Ejemplo 4.7 La MT del Ej. 4.5 se puede dibujar en la notación modular de la siguiente forma. El dibujo no sólo es mucho más simple y fácil de entender, sino que es independiente de Σ.



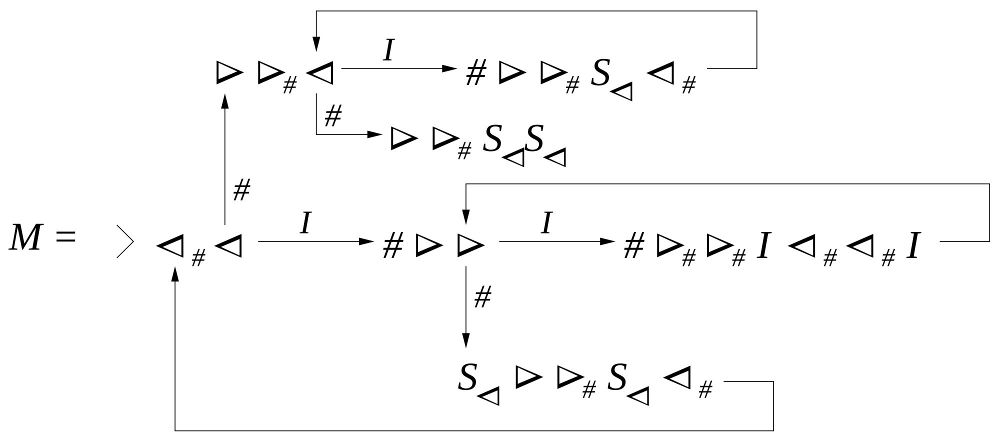
Ejemplo 4.8 Otra máquina interesante es la copiadora, $C \ : \ \#w\underline{\#} \longrightarrow \#w\#w\underline{\#}$. Se invita al lector a dibujarla, inspirándose en la que calcula $f(w) = ww^R$ del Ej. 4.7. Con esta máquina podemos componer fácilmente una que calcule $f(w) = ww$: $C\,S_{\lhd}$. Claro 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 Ej. 4.6, ahora puede escribirse simplemente como $S_{\lhd}$. La que incrementa un número es $I \rhd$. 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.

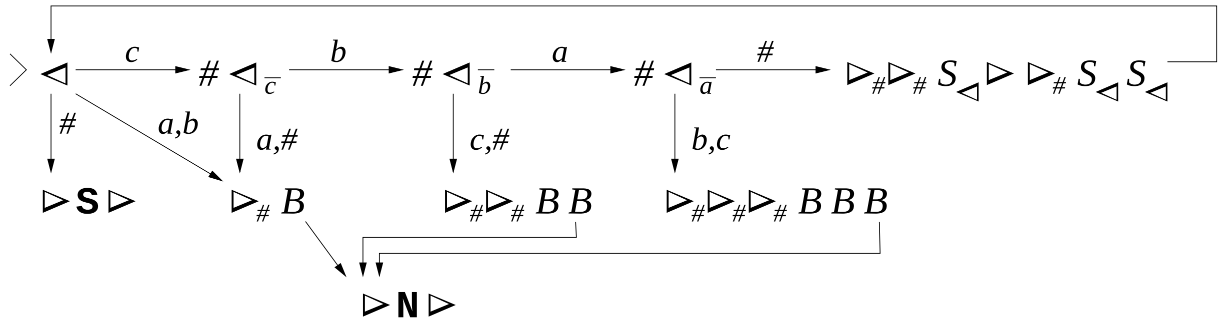


La MT M que implementa la multiplicación puede ser como sigue. En el ciclo principal vamos construyendo $\#I^n\#I^m\underline{\#}I^{n\cdot m}$ mediante ir reduciendo I^n e ir agregando una copia de I^m al final de la cinta. Cuando I^n desaparece, pasamos a otro ciclo que borra I^m para que quede sólamente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

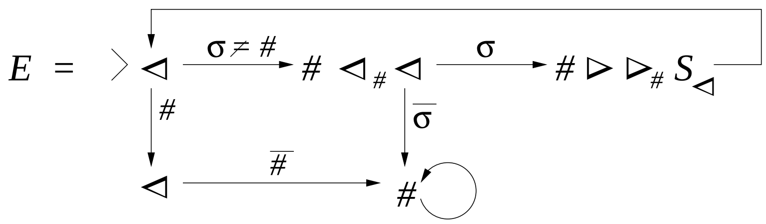


El siguiente ejemplo es importante porque demuestra que existen lenguajes decidibles que no son libres del contexto.

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 $\#u\#v\underline{\#}$ y se detiene sii $u = v$. La usaremos más adelante.



Observación 4.5 *El uso de variables para denotar nombres de letras no está exento de problemas. Imaginemos que haya dos formas de llegar a un nodo, una de ellas asignando σ y la otra no. En este caso, si intentáramos expandir la "macro", tendríamos problemas. Simplemente el dibujo no representaría una MT. Peor aún: una flecha rotulada σ que partiera de ese nodo se podría interpretar de distinta forma según vengamos de un nodo donde σ se asigne (en cuyo caso la flecha está diciendo que la letra debe ser igual al valor ya fijado de σ), o de otro donde no (en cuyo caso la flecha no pone una condición sobre la letra sino que la llama σ). Este tipo de ambigüedades debe evitarse, bajo la responsabilidad de quien dibuja*

la máquina. Nuevamente recordamos que este mecanismo no es parte de nuestro formalismo real sino que es una forma abreviada de dibujar las MTs. Es nuestra responsabilidad usarlo correctamente.

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

Es posible extender el mecanismo de la MT sin alterar lo que se puede hacer con la MT básica. Esto simplifica mucho dibujar MTs que realicen ciertas tareas, con la seguridad de que podríamos hacerlo con MTs básicas de ser necesario. Algunas de estas extensiones son: MTs que usan varias cintas, MTs con varios cabezales, MTs con cinta infinita hacia ambos lados, MTs con cinta k-dimensional, etc. De todos esos mecanismos, el más útil es la MT con k cintas, a la cual nos dedicaremos en esta sección. Otras extensiones se pueden ver en el libro.

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 4.13 *Una MT de k cintas es una tupla $M = (K, \Sigma, \delta, S)$ tal que K, Σ y s son como en la Def. 4.1 y $\delta : K \times \Sigma^k \longrightarrow (K \cup \{h\}) \times (\Sigma \cup \{\lhd, \rhd\})^k$.*

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 $(q, u_1\underline{a_1}v_1, u_2\underline{a_2}v_2, \ldots, u_k\underline{a_k}v_k)$.*

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: $(q, u_1\underline{a_1}v_1, u_2\underline{a_2}v_2, \ldots, u_k\underline{a_k}v_k) \vdash (q', u'_1\underline{a'_1}v'_1, u'_2\underline{a'_2}v'_2, \ldots, u'_k\underline{a'_k}v'_k)$ si $q \in K$, $\delta(q, a_1, a_2, \ldots, a_k) = (q', b_1, b_2, \ldots, b_k)$, y las reglas de la Def. 4.3 se cumplen para cada $u_i\underline{a_i}v_i$, $u'_i\underline{a'_i}v'_i$ y b_i.*

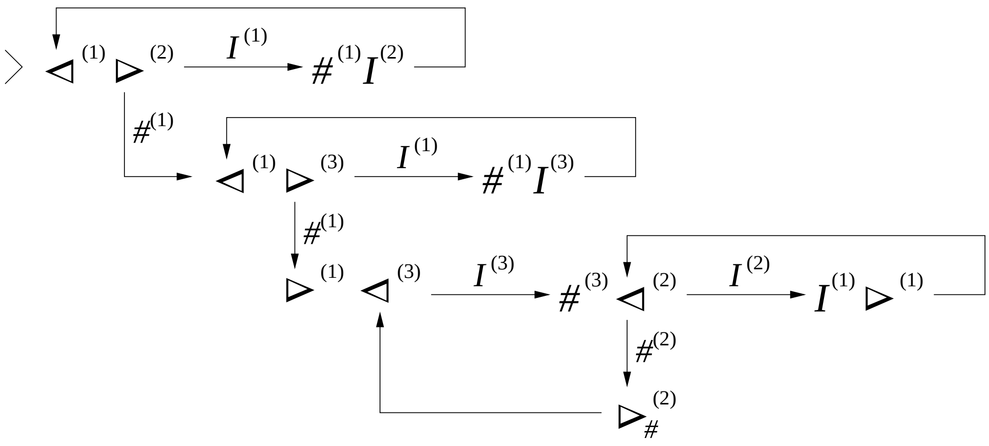
Definamos ahora cómo usaremos una MT de k cintas.

Definición 4.16 *Una MT de k cintas arranca con una configuración de la forma $\#w\underline{\#}$ en la cinta 1, y todas las otras cintas en la configuración $\underline{\#}$. 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.

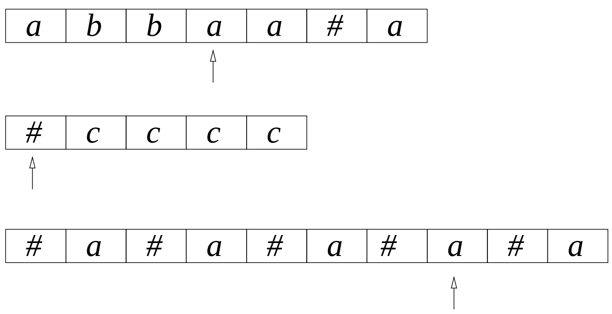
Ejemplo 4.12 Una MT M que implementa la multiplicación (como en el Ej. 4.9), ahora usando 3 cintas, puede ser como sigue. Ahora I^n e I^m se dejan en las cintas 2 y 3, respectivamente, y el resultado se construye en la cinta 1.



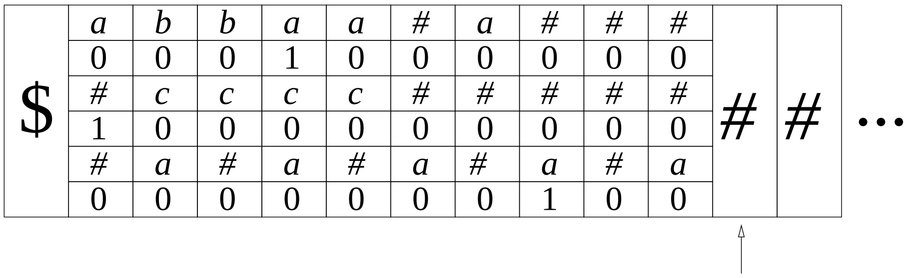
Veamos ahora cómo simular una MT de k cintas con una de 1 cinta, para convencernos de que podemos usar MTs de k cintas de ahora en adelante y saber que, de ser necesario, la podríamos convertir a una tradicional.

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



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 $\delta(q, \sigma_1, \ldots, \sigma_k) = (q', b_1, \ldots, b_k)$. ¿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 $\lhd$ ($\rhd$), 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 $(\Sigma \times \{0,1\})^k$.)

$$F_q = \; > \lhd_{(*,1,*,*,*,*)} \xrightarrow{(\sigma_1,1,*,*,*,*)} \rhd_{\#} \lhd_{(*,*,*,1,*,*)} \xrightarrow{(*,*,\sigma_2,1,*,*)} \rhd_{\#} \lhd_{(*,*,*,*,*,1)} \xrightarrow{(*,*,*,*,\sigma_3,1)} \rhd_{\#} D_{q,\sigma_1,\sigma_2,\sigma_3}$$

Estos módulos F_q terminan en módulos de la forma $D_{q,a_1,\ldots,a_k}$ (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 $\delta(q, a_1, \ldots, a_k)$. Por ejemplo, si $\delta(q, a_1, a_2, a_3) = (q', \lhd, \rhd, b)$, $b \in \Sigma$, 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).

$$\begin{tikzcd}
I_1 = \; > \lhd_{(*,1,*,*,*,*)} \arrow[r, "{(\sigma_1,1,\sigma_2,p_2,\sigma_3,p_3)}"] & (\sigma_1,0,\sigma_2,p_2,\sigma_3,p_3) \lhd \arrow[r, "{(\sigma_1',0,\sigma_2',p_2',\sigma_3',p_3')}"] \arrow[d, "\$"] & (\sigma_1',1,\sigma_2',p_2',\sigma_3',p_3') \rhd_{\#} \\
& \lhd &
\end{tikzcd}$$

$$\begin{tikzcd}
D_2 = \; > \lhd_{(*,*,*,1,*,*)} \arrow[r, "{(\sigma_1,p_1,\sigma_2,1,\sigma_3,p_3)}"] & (\sigma_1,p_1,\sigma_2,0,\sigma_3,p_3) \rhd \arrow[r, "{(\sigma_1',p_1',\sigma_2',0,\sigma_3',p_3')}"] \arrow[d, "\#"] & (\sigma_1',p_1',\sigma_2',1,\sigma_3',p_3') \rhd_{\#} \\
& (\#,0,\#,1,\#,0) \rhd &
\end{tikzcd}$$

$$W_{b,3} = \; > \lhd_{(*,*,*,*,*,1)} \xrightarrow{(\sigma_1,p_1,\sigma_2,p_2,\sigma_3,1)} (\sigma_1,p_1,\sigma_2,p_2,b,1) \rhd_{\#}$$

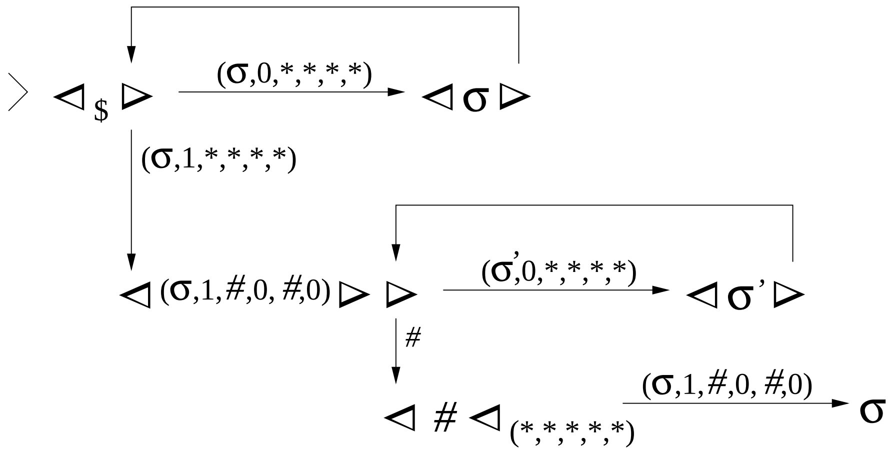
Observar, en particular, que si la MT simulada se cuelga (intenta moverse al \$), la simuladora se cuelga también (podríamos haber hecho otra cosa). Si, en cambio, la acción D_i se para sobre un #, significa que está tratando de acceder una parte de la cinta que aún no hemos particionado, por lo que 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

$$\begin{tikzcd}
> S_{\rhd} \lhd_{\#} \lhd \; \$ \; \rhd \; (\#,0,\#,1,\#,1) \rhd \arrow[r, "\sigma \neq \#"] \arrow[d, "\#"] & (\sigma,0,\#,0,\#,0) \arrow[l, bend right] \\
(\#,1,\#,0,\#,0) \rhd &
\end{tikzcd}$$

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

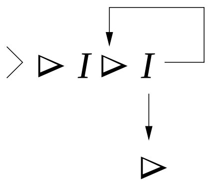
Definición 4.17 *Una Máquina de Turing no Determinística (MTND) es una tupla* $M = (K, \Sigma, \Delta, s)$*, donde* K*,* Σ *y* s *son como en la Def. 4.1 y* $\Delta \subseteq (K \times \Sigma) \times ((K \cup \{h\}) \times (\Sigma \cup \{\lhd, \rhd\}))$*.*

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 sólamente 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.

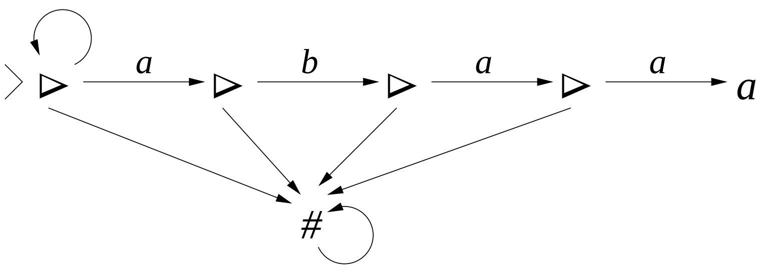
Ejemplo 4.13 Hagamos una MT que acepte el lenguaje de los números compuestos, $\{I^n,\ \exists p, q \geq 2, n = p \cdot q\}$. (Realmente este lenguaje es decidible, pero aceptarlo ilustra muy bien la idea.) Lo que haríamos con una MTD (y con nuestro lenguaje de programación favorito) sería generar, uno 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.

$G =$ 

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 sii recibe dos cadenas iguales (Ej. 4.11).

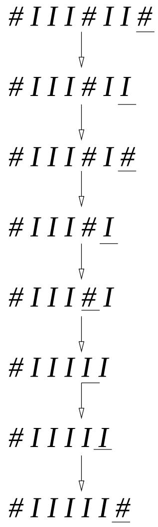
Es bueno detenerse a reflexionar sobre esta MTND. Primero aplica G dos veces, con lo que la cinta queda de la forma $\#I^n\#I^p\#I^q\underline{\#}$, para algún par $p, q \geq 2$. Al aplicar M, la cinta queda de la forma $\#I^n\#I^{pq}\underline{\#}$. Al aplicar E, ésta se detendrá sólo si $n = pq$. Esto 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.

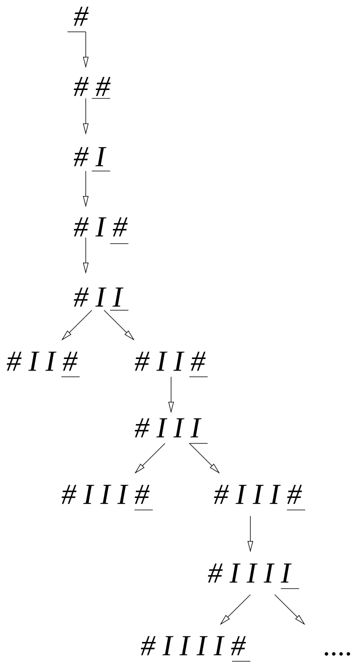
Ejemplo 4.14 Otro ejemplo útil es usar una MTND para buscar una secuencia dada en la cinta, como *abaa*. Una MTD debe considerar cada posición de comienzo posible, compararla con *abaa*, y volver a la siguiente posición de comienzo. Esto no es demasiado complicado, pero más simple es la siguiente MTND,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



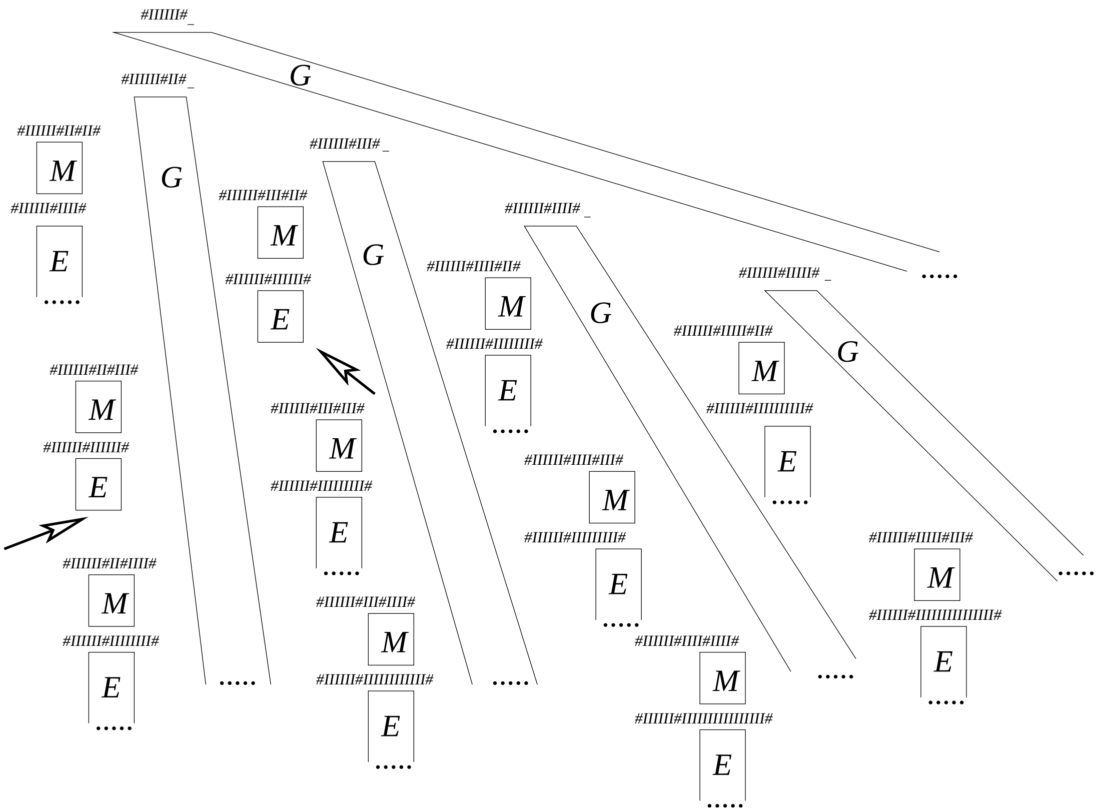
Los ejemplos anteriores nos llevan a preguntarnos cómo se pueden probar todas las alternativas de p y q, si son infinitas. Para comprender esto, y para la simulación de MTNDs con MTDs, es conveniente pensar de la siguiente forma. Una MTD produce una secuencia de configuraciones a lo largo del tiempo. 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 sii 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 $\#w\underline{\#}$ intacta.

3. En la tercera almacenaremos una secuencia de *dígitos* en base r (o sea símbolos sobre d_1, d_2, ..., 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:

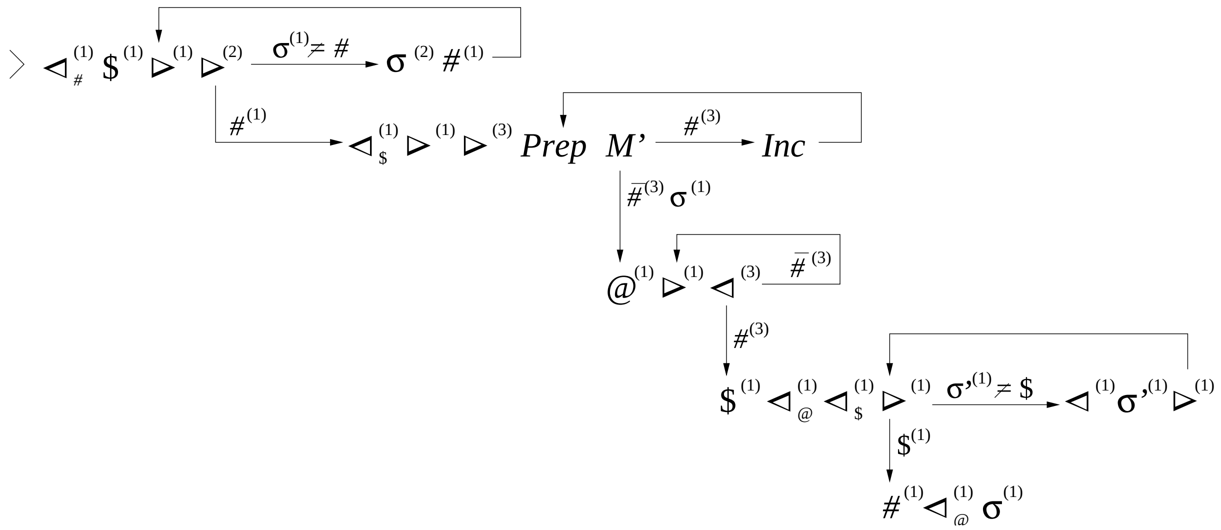
1. Limpiará la cinta 1 y copiará la cinta 2 en la cinta 1 (máquina *Prep*).

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).

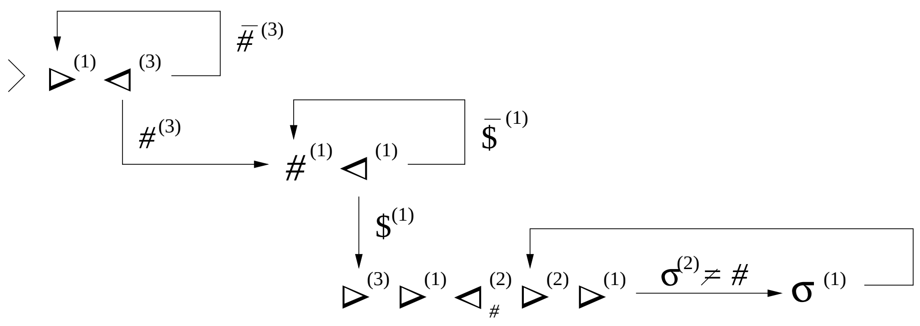


La máquina *Prep* realiza las siguientes acciones:

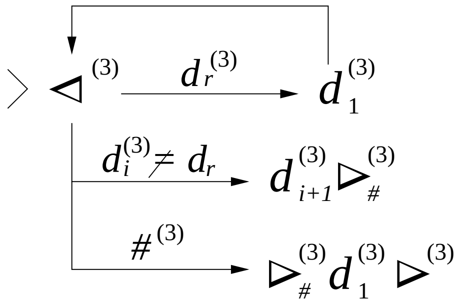
1. Se mueve hacia la derecha en la cinta 1 todo lo necesario para asegurarse de estar más a la derecha que cualquier cosa escrita. Como, en la cinta 3, hemos pasado por una directriz d_i por cada paso de M simulado, y M no puede moverse más de una casilla a la derecha por cada paso, y la simulación de 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.

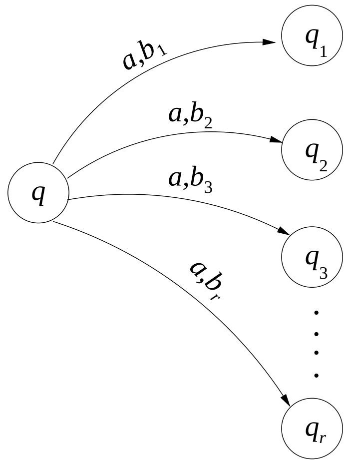
3. Copia la cinta 2 a la cinta 1, quedando en la configuración inicial $\#w\underline{\#}$.

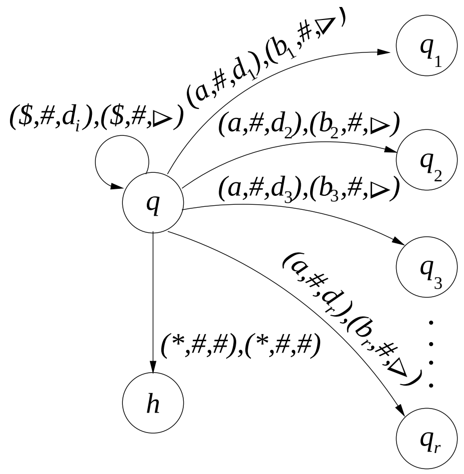


La máquina *Inc* comienza al final de las directivas en la cinta 3 y simula el cálculo del sucesor en un número en base r: Si $r = 3$, el sucesor de d_1d_1 es d_1d_2, luego d_1d_3, luego d_2d_1, y así hasta d_3d_3, cuyo sucesor es $d_1d_1d_1$. Por ello va hacia atrás convirtiendo d_r en d_1 hasta que 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.) □

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* $K = \{q_1,\ q_2, \ldots,\ q_{|K|}\}$ *y* $\Sigma = \{\#,\ a_2, \ldots,\ a_{|\Sigma|}\}$*. Definimos también* $K_\infty = \{q_1,\ q_2, \ldots\}$*,* $\Sigma_\infty\{\#,\ a_2, \ldots\}$*. 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*

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

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:*

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

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

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

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* $(s_{MUT}, \#\rho(M)\rho(w)\underline{\#})$*, 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* $(s, \#w\underline{\#})$*,* M *se detiene en una configuración* $(h, u\underline{a}v)$*, 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* $(s, \#w\underline{\#})$*,* M *no se detiene nunca, la MUT no se detiene nunca.*

3. *Si, arrancada en la configuración* $(s, \#w\underline{\#})$*,* 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 $\#\rho(M)\underline{\#}$ y no la modificaremos.

3. En la cinta 3 tendremos $\#\lambda(q)\underline{\#}$, donde q es el estado en que está la máquina simulada.

El primer paso de la simulación es, entonces, pasar de la configuración inicial $(\#\rho(M)\rho(w)\underline{\#}, \underline{\#}, \underline{\#})$, a la apropiada para comenzar la simulación: $(\rho(\#w)\lambda(\#)\underline{c}, \#\rho(M)\underline{\#}, \#\lambda(s)\underline{\#})$ ($\lambda(s)$ 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 $\#I\underline{\#}$, 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 $\#I^i\underline{\#}$ y en la cinta 1 alrededor del cabezal tenemos $\ldots cI^j\underline{c}\ldots$, entonces se busca en la cinta 2 el patrón ccI^icI^jc. 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^rcI^sc. Entonces reescribimos la cinta 3 para que diga $\#I^r\underline{\#}$, y:

(a) Si $s = 1$ debemos movernos a la izquierda en la cinta simulada. Ejecutamos $\lhd^{(1)}_{c,\#}$. 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 ($\lhd^{(1)}$), 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 $\ldots c\lambda(\#)\underline{c}\# \ldots = \ldots cIII\underline{c}\# \ldots$. 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 $\rhd^{(1)}_{c,\#}$. 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 $\ldots cI^j\underline{c} \ldots$ y debemos convertirlo en $\ldots cI^s\underline{c} \ldots$. 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 $\# \rhd_\# S_\rhd \lhd_\# c \lhd I \rhd$, o borrarlo con la secuencia $\lhd \# \rhd \# \rhd_\# S_\lhd \lhd_\# c$.

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 simularla en JTV puede ser entretenido.

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 significar 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` $w = bbabbabbbabbabbbb$ `then return S`
`if` $w = bb$ `then return N`
`if` $w = bbabbbaba$ `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 real. Esta es una pregunta difícil de responder sin sesgarnos a lo que conocemos. ¿Serán aceptables la computación cuántica (¿se podrá finalmente implementar de verdad?), 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 real 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* máqina 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 $l \geq 1$. 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 convenidas. 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 2 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. □

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*

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

Definición 4.28 *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. 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*

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

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 $\{w \in \{a, b, c\}^*,\ w$ tiene la misma cantidad de a's, b's, y c's $\}$ puede ser $V = \{S, A, B, C\}$ y R con las reglas:

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

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* $u \in \Sigma_0^*$*,* $xuy \Longrightarrow_G^* x'vy'$ *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* $[uqav] \Longrightarrow_G^* [u'q'a'v']$*.*

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

1. Si $\delta(q_1, a) = (q_2, \lhd)$ 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 $\delta(q_1, a) = (q_2, \rhd)$ agregamos a R las reglas $q_1ab \longrightarrow aq_2b$ para todo $b \in \Sigma$, 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. $\Box$

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, $(s, \#u\underline{\#}) \vdash_M^* (h, \#f(u)\underline{\#})$. 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. □

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 `S` o a `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 $w \in L(G)$, $(s, \underline{\#}) \vdash_M^* (h, \#w\underline{\#})$.*

Prueba: Similarmente al Teo. 4.1, ponemos todas las reglas en una cinta 2, y $\#S\underline{\#}$ 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. □

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 $\delta(q_0, a) = (q_1, b)$, $\delta(q_0, b) = (q_1, a)$, $\delta(q_0, \#) = (h, \#)$, $\delta(q_1, a) = (q_0, \rhd)$, $\delta = (q_1, b) = (q_0, \rhd)$, $\delta(q_1, \#) = (q_0, \rhd)$. Configuración inicial: $(q_0, \underline{a}abbba)$.

   (b) $M = (\{q_0, q_1, q_2\}, \{a, b, \#\}, \delta, q_0)$, con $\delta(q_0, a) = (q_1, \lhd)$, $\delta(q_0, b) = (q_0, \rhd)$, $\delta(q_0, \#) = (q_0, \rhd)$, $\delta(q_1, a) = (q_1, \lhd)$, $\delta(q_1, b) = (q_2, \rhd)$, $\delta(q_1, \#) = (q_1, \lhd)$, $\delta(q_2, a) = (q_2, \rhd)$, $\delta(q_2, b) = (q_2, \rhd)$, $\delta(q_2, \#) = (q_2, \#)$, a partir de $(q_0, a\underline{b}b\#bb\#\#aba)$.

   (c) $M = (\{q_0, q_1, q_2\}, \{a, \#\}, \delta, q_0)$, con $\delta(q_0, a) = (q_1, \lhd)$, $\delta(q_0, \#) = (q_0, \#)$, $\delta(q_1, a) = (q_2, \#)$, $\delta(q_1, \#) = (q_1, \#)$, $\delta(q_2, a) = (q_2, a)$, $\delta(q_2, \#) = (q_0, \lhd)$, 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 $\{w \in \{a,b\}^*,\ w$ contiene al menos una $a\}$.
   (c) Compute $f(w) = ww$.
   (d) Acepte el lenguaje a^*ba^*b.
   (e) Decida el lenguaje $\{w \in \{a,b\}^*,\ w$ contiene tantas a's como b's$\}$.
   (f) Compute $f(m,n) = m$ div n y m mod 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 $\{a^nb^nc^n,\ n \geq 0\}$.
   (c) Genere $\{w \in \{a,b,c\}^*,\ w$ tiene más a's que b's y más b's que c'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 $\{a^nb^nc^n,\ n \geq 0\}$.

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 sóla 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 $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$ y tal que $\sigma_1 < \sigma_2 < \ldots < \sigma_n$. 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 $\#u_1\#u_2\#...\#u_n\#$ y en la cinta 2 una cadena de la forma $\#v_1\#v_2\#...\#v_n\#$. 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 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 $\mathcal{L}(MGT) = \{\rho(G)\rho(w),\ w \in L(G)\}$.

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

$$(s, \#I^n\underline{\#}) \quad \vdash_M^* \quad (h, \#I^{n!}\underline{\#})$$

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

C2 2003 Dada una MT que computa una función $f : \Sigma^* \rightarrow \Sigma^*$, indique cómo construir una MT que compute la función inversa $x = f^{-1}(w)$ para algún x tal que $f(x) = w$ (puede hacer lo que sea si no existe tal x). Indique qué hace su MT cuando (i) existe más de un argumento x tal que $f(x) = w$; (ii) no existe ningún argumento x tal que $f(x) = w$.

Ex 2006 Considere un modelo de computación donde una MT (robotizada) puede, mediante una instrucción atómica, *fabricar* otra MT. La nueva MT es idéntica y arranca con la misma configuración de la MT fabricante. Para poder diferenciarlas, la MT fabricante queda con un **0** bajo el cabezal luego de fabricar la copia, mientras que la copia queda con un **1** bajo el cabezal cuando arranca su ejecución luego de ser copiada.

Las dos MTs siguen trabajando en paralelo y sincronizadamente, pero sin comunicarse. Tanto la copia como la original pueden fabricar nuevas copias. Cuando alguna copia se detiene, envía un mensaje a todas las demás para que se detengan también y se autodestruyan. La MT que terminó es la única que sobrevive.

Describa formalmente esta nueva MT: incluya la definición de la MT, lo que es una configuración, cómo se pasa de una configuración a la siguiente, la noción de decidir y aceptar, y el lenguaje decidido y aceptado por una MT.

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 computación. 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.

6. Investigue más sobre el modelo RAM de computación. Una fuente es [AHU74, cap 1]. Esto entra también en el tema del próximo capítulo.

Referencias

[AHU74] A. Aho, J. Hopcroft, J. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[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

[LP81, cap 5 y 6]

Una vez establecido nuestro modelo de computación y justificado con la Tesis de Church, en este capítulo vamos por fin a demostrar los resultados centrales de qué cosas se pueden computar y qué cosas no.

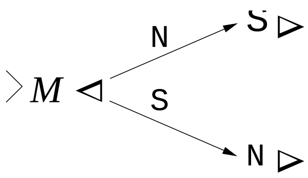
Comenzaremos con una famosa demostración de que el *problema de la detención* (o *halting problem*) no se puede resolver por computador, es decir, que es indecidible. Esta demostración arroja luz sobre la diferencia entre decidir y aceptar un lenguaje. Luego profundizaremos más en esa relación, y finalmente mostraremos cómo demostrar que otros problemas son indecidibles, dando ejemplos de problemas al parecer relativamente simples que no tienen solución.

5.1 El Problema de la Detención

Comencemos con algunas observaciones relativamente simples sobre la relación entre decidir y aceptar lenguajes.

Lema 5.1 *Si un lenguaje L es decidible, entonces L^c es decidible.*

Prueba: Si L es decidible entonces existe una MT M que, empezando en la configuración $(s, \#w\underline{\#}) \vdash_M^* (h, \#\mathtt{S}\underline{\#})$ si $w \in L$ y $(s, \#w\underline{\#}) \vdash_M^* (h, \#\mathtt{N}\underline{\#})$ si no. Entonces, basta con ejecutar M y luego invertir su respuesta para decidir L^c:



$\square$

Lema 5.2 *Si un lenguaje L es decidible, entonces es aceptable.*

Prueba: Si L es decidible entonces existe una MT M que, empezando en la configuración $(s, \#w\underline{\#}) \vdash_M^* (h, \#\mathtt{S}\underline{\#})$ si $w \in L$ y $(s, \#w\underline{\#}) \vdash_M^* (h, \#\mathtt{N}\underline{\#})$ si no. Entonces, la siguiente MT

$$> M \lhd \xrightarrow{\texttt{N}} \#\circlearrowleft$$

se detiene sii $w \in L$, es decir, acepta L y por lo tanto L es aceptable. □

El lema anterior es bastante evidente: si podemos responder sí o no frente a una cadena del lenguaje, claramente podemos detenernos si la cadena está en el lenguaje y no detenernos sino.

Tal como hemos visto que los complementos de lenguajes decidibles son decidibles, podríamos preguntarnos si los complementos de lenguajes aceptables son aceptables. El siguiente lema nos dice algo al respecto.

Lema 5.3 *Si L es aceptable y L^c es aceptable, entonces L es decidible.*

Prueba: Si tenemos una M_1 que acepta L y una M_2 que acepta L^c, entonces, dada una cadena w exactamente una entre M_1 y M_2 se detendrán frente a w. Podemos correr una MTND M que, no determinísticamente, elige correr M_1 ó M_2. Lo que corre en el caso de M_1 es realmente una variante que funciona en la cinta 2, y cambia el estado h por $\rhd^{(1)}\texttt{S}^{(1)}\rhd^{(1)}$. Lo que corre en el caso de M_2 es realmente una variante que funciona en la cinta 2, y cambia el estado h por $\rhd^{(1)}\texttt{N}^{(1)}\rhd^{(1)}$. Está claro que M siempre va a terminar y dejar en la cinta 1 la respuesta `S` o `N` según $w \in L$ o no. □

Por lo tanto, si los complementos de los lenguajes aceptables fueran aceptables, todos los lenguajes aceptables serían decidibles. La pregunta interesante es entonces: ¿será que todo lenguaje aceptable es decidible?

Observación 5.1 *¿Qué implicaciones tendría que esto fuera verdad? Esto significaría que, dada una MT que se detuviera sólo frente a cadenas de un cierto lenguaje L, podríamos construir otra que se detuviera siempre, y nos dijera si la cadena está en L o no, es decir, si la primera MT se detendría frente a L o no. Por ejemplo, no sería difícil hacer una MTND que intentara demostrar un cierto teorema (tal como hemos hecho una que intenta generar cierta palabra con una GDC, Teo. 4.2). Luego podríamos saber si el teorema es cierto o no mediante preguntarnos si la MTND se detendrá o no. ¡Tendríamos un mecanismo infalible para saber si cualquier teorema, en cualquier sistema formal, es demostrable o no! Podríamos haber resuelto el último teorema de Fermat o la hipótesis del continuo sin mayor esfuerzo, o la aún no demostrada conjetura de Goldbach (todo número par mayor que 2 es la suma de dos primos). Otra forma más pedestre de ver esto es como sigue. Bastaría hacer un programa que fuera generando cada contraejemplo posible a la conjetura de Goldbach (todos los números pares), probara si es la suma de dos primos, y se detuviera si no. Sabríamos si la conjetura es falsa o no mediante preguntarnos si el programa se detendrá o no.*

Esta observación, además de adelantar cierto pesimismo sobre la posibilidad de que los lenguajes aceptables sean decidibles, muestra que un cierto lenguaje en particular es esencial para responder esta pregunta.

Definición 5.1 *El* problema de la detención *(o* halting problem*) es el de, dada una MT M y una cadena w, determinar si M se detendrá frente a w, es decir, si M acepta w, o formalmente, si* $(s, \#w\underline{\#}) \vdash_M^* (h, u\underline{a}v)$ *para algún* uav.

Este problema se puede traducir al de decidir un lenguaje, gracias al formalismo introducido para la MUT (Sección 4.6).

Definición 5.2 *El lenguaje* $K_0 \subseteq \{I, c\}^*$ *se define como sigue:*

$$K_0 \;=\; \{\rho(M)\rho(w),\ M \textit{ acepta } w\}$$

Podemos pensar en K_0, informalmente, como el lenguaje de los pares (M, w) tal que M acepta w. Pero, por otro lado, K_0 es nada más que un conjunto de cadenas de c's e I's. Es inmediato que K_0 es aceptable.

Lema 5.4 K_0 *es aceptable.*

Prueba:
Basta ver que la MUT definida en la Def. 4.23 acepta precisamente K_0, mediante simular M frente a la entrada w. Entonces la MUT aceptará $\rho(M)\rho(w)$ sii M acepta w. El único detalle es que la MUT supone que la entrada es de la forma $\rho(M)\rho(w)$, mientras que ahora deberíamos primero verificar que la secuencia de c's y I's de la entrada tiene esta forma. Si no la tiene, no deberíamos aceptar la cadena (es decir, deberíamos entrar a un loop infinito). No es difícil decidir si la entrada tiene la forma correcta, queda como ejercicio para el lector. □

El siguiente lema establece la importancia de K_0 con respecto a nuestra pregunta.

Lema 5.5 K_0 *es decidible sii todo lenguaje aceptable es decidible.*

Prueba: La vuelta es evidente dado que K_0 es aceptable. Para la ida, supongamos que K_0 es decidible, es decir, existe una MT M_0 que decide K_0: dada $\rho(M)\rho(w)$, responde `S` si M se detendrá frente a w, y `N` sino (o si la entrada no es de la forma $\rho(M)\rho(w)$). Ahora supongamos que un cierto L es aceptable: tenemos una MT M que se detiene frente a w sii $w \in L$. Construimos una MT M' que escriba en la cinta $\rho(M)$ (una constante) y luego codifique w como $\rho(w)$. A la entrada resultante, $\rho(M)\rho(w)$, se le aplica la MT que decide K_0, con lo cual nos responderá, finalmente, si $w \in L$ o no. □

De este modo, nuestra pregunta se reduce a ¿es K_0 decidible? Para avanzar en este sentido, definiremos un lenguaje K_1 a partir de K_0.

Definición 5.3 *El lenguaje* K_1 *se define como*

$$K_1 \;=\; \{\rho(M),\ M \textit{ acepta } \rho(M)\}$$

y asimismo vale la pena escribir explícitamente

$$K_1^c \;=\; \{w \in \{c, I\}^*,\; w \textit{ es de la forma } \rho(M) \textit{ y } M \textit{ no acepta } \rho(M), \\ \textit{o } w \textit{ no es de la forma } \rho(M)\}$$

Para que esto tenga sentido, debemos aclarar que estamos hablando de MTs M codificables (Def. 4.18), y de que acepten la versión codificada de $\rho(M)$.

Observación 5.2 *Antes de continuar, asegurémosnos de entender el tecnicismo involucrado en la definición de K_1. No todas las MTs tienen un alfabeto formado por c's y I's, de modo que ¿qué sentido tiene decir que aceptan $\rho(M)$, que es una secuencia de c's y I's? Lo que aclara la definición es que no nos referimos a cualquier MT sino a MTs codificables, cuyo alfabeto es de la forma a_1, a_2, etc. (Def. 4.18). A su vez, hablamos de la versión codificada de $\rho(M)$, donde, por ejemplo, c se convierte en a_2 e I en a_3 (recordemos que a_1 está reservado para el #). Ahora sí funciona: toda MT que tenga al menos tres símbolos en su alfabeto (que renombramos a_1, a_2 y a_3) aceptan o no cadenas formadas por a_2 y a_3, y K_1 son las MTs tales que si las codificamos con c's e I's, traducimos esa codificacion a a_2's y a_3's, y se las damos como entrada a la misma M, terminarán. ¿Y las MTs que tienen alfabetos de menos de tres símbolos? Pues las dejamos fuera de K_1.*

Una vez comprendido el tecnicismo, es bueno no prestarle mucha antención y tratar de entender más intuitivamente qué es K_1. Esencialmente, K_1 es el conjunto de las MTs que se aceptan a sí mismas. Similarmente, K_1^c sería el conjunto de las MTs que no se aceptan a sí mismas (más las cadenas que no representan ninguna $\rho(M)$).

El siguiente lema debería ser evidente.

Lema 5.6 *Si K_0 es aceptable/decidible, entonces K_1 es aceptable/decidible. Por lo tanto K_1 es aceptable.*

Prueba: Sea M que acepta/decide K_0. Entonces una M' que acepta/decide K_1 simplemente recibe $\rho(M)$ en la cinta de entrada, le concatena $\rho(\rho(M))$, e invoca M. La doble ρ se debe a que estamos codificando la cadena $w = \rho(M)$ en el lenguaje de c's e I's *¡a pesar de que ya lo estuviera! Es decir, si $c = a_2$ y $I = a_3$, entonces c se (re)codificará como $IIII$ e I como $IIIII$.* □

Notar que bastaría que K_1^c fuera aceptable para que K_1 fuera decidible. En seguida demostraremos que no lo es.

Antes de ir a la demostración formal, es muy saludable recurrir a la *paradoja del barbero* de Russell (usada originalmente para demostrar que no toda propiedad define un conjunto).

Definición 5.4 *La* paradoja del barbero *de Russell es como sigue: Erase un pueblo con un cierto número de barberos. Estos afeitaban a los que no eran barberos. Pero ¿quién los afeitaría a ellos? Algunos querían afeitarse ellos mismos, otros preferían que los afeitara otro barbero. Después de discutir varios días, decidieron nombrar a uno sólo de ellos como* el barbero de todos los barberos*. Este barbero, entonces, estaría a cargo de afeitar exactamente a todos los barberos que no se afeitaran a sí mismos. El barbero designado quedó muy contento* *con su nombramiento, hasta que a la mañana siguiente se preguntó quién lo afeitaría a él. Si se afeitaba él mismo, entonces estaba afeitando a alguien que se afeitaba a sí mismo, incumpliendo su designación. Pero si no se afeitaba él mismo, entonces no estaría afeitando a alguien que no se afeitaba a sí mismo, también incumpliendo su designación. El barbero renunció y nunca lograron encontrar un reemplazante.*

La paradoja muestra que *no es posible* tener un barbero que afeite exactamente a los barberos que no se afeitan a sí mismos. Si cambiamos "barbero" por "MT" y "afeitar" por "aceptar" habremos comprendido la esencia del siguiente lema.

Lema 5.7 *K_1^c no es aceptable.*

Prueba: Supongamos que lo fuera. Entonces existirá una MT M que acepta K_1^c. Nos preguntamos entonces si la versión codificable de M acepta la cadena $w = \rho(M)$ (codificada).

1. Si la acepta, entonces $\rho(M) \in K_1$ por la Def. 5.3. Pero entonces M acepta una cadena $w = \rho(M)$ que no está en K_1^c sino en K_1, contradiciendo el hecho de que M acepta K_1^c.

2. Si no la acepta, entonces $\rho(M) \in K_1^c$ por la Def. 5.3. Pero entonces M no acepta una cadena $w = \rho(M)$ que está en K_1^c, nuevamente contradiciendo el hecho de que M acepta K_1^c.

En ambos casos llegamos a una contradicción, que partió del hecho de suponer que existía una M que aceptaba K_1^c. Entonces K_1^c no es aceptable. $\square$

Observación 5.3 *La demostración del Lema 5.7 es una forma de diagonalización (usado en el Teo. 1.2). En una matriz, enumeramos cada MT M en una fila, y cada cadena w en una columna. Cada celda vale 1 si la MT (fila) se detiene frente a la cadena (columna). En cada fila i de una MT M_i, una de las celdas (llamémosla c_i) corresponde a la cadena $\rho(M_i)$. Una MT que aceptara K_1^c debería tener un 0 en c_i si la MT M_i tiene un 1 (se acepta a sí misma) y un 1 en c_i si M_i tiene un 0 (no se acepta a sí misma). Pero entonces esa MT que acepta K_1^c no puede ser ninguna de las MT listadas, pues difiere de cada fila i de la matriz en la columna c_i. Por lo tanto no existe ninguna MT que acepte K_1^c.*

De aquí derivamos el teorema más importante del curso. Contiene varias afirmaciones relacionadas.

Teorema 5.1 *K_0 y K_1 son aceptables pero no decidibles. Ni K_0^c ni K_1^c son aceptables. Existen lenguajes aceptables y no decidibles. No todos los complementos de lenguajes aceptables son aceptables. El problema de la detención es indecidible.*

Prueba: K_1 no es decidible porque si lo fuera entonces K_1^c también sería decidible (Lema 5.1), y ya vimos que K_1^c no es siquiera aceptable (Lema 5.7). Como K_1 no es decidible, K_0 tampoco lo es (Lema 5.6), y por el mismo lema, como K_1^c no es aceptable, K_0^c tampoco lo es. Como K_0 y K_1 son aceptables y no decidibles, no todo lenguaje aceptable es decidible, y no todo complemento de lenguaje aceptable es aceptable (pues sino todos los lenguajes aceptables serían decidibles, Lema 5.3). En particular, el problema de la detención (decidir K_0) es indecidible. $\square$

Observación 5.4 *Esto significa, en particular, que es imposible determinar si un programa se detendrá frente a una entrada.* Es decir, en general. Para ciertos programas y/o ciertas entradas puede ser muy fácil, pero no se puede tener un método que siempre funcione. *Los métodos de verificación automática de programas (que intentan demostrar su correctitud) no tienen esperanza de algún día funcionar en forma totalmente automática, pues ni siquiera es posible saber si un programa se detendrá.*

El hecho de que las MTs puedan no detenerse parece ser un problema. ¿No sería mejor tener un formalismo de computación donde *todos* los programas terminaran? Por ejemplo un lenguaje de programación sin WHILE, sino sólo con un tipo de FOR que garantizara terminación. Si nos acostumbráramos a programar así, se evitarían muchos errores.

El siguiente teorema demuestra que no es así. *Todo mecanismo de computación, para ser tan completo como las MTs, requiere que existan programas que no terminan y más aún, que sea indecidible saber si terminarán o no.* (En todos los formalismos alternativos mencionados en la Sección 4.7 existen instrucciones que permiten la no terminación.)

Teorema 5.2 *Sea un mecanismo de computación de funciones de $\mathbb{N}$ en $\mathbb{N}$, tal que existe una forma finita de representar las funciones como secuencias de símbolos, de verificar la correctitud de una representación, y de simular las funciones con una MT a partir de la representación, de modo que la simulación siempre termina. Entonces existen funciones computables con MTs y que no se pueden calcular con este mecanismo.*

Prueba: Llamemos f_i a la i-ésima representación correcta de una función que aparece cuando se van generando todas las cadenas posibles en orden de longitud creciente y, dentro de la misma longitud, en orden lexicográfico. Definamos la matriz $F_{i,j} = f_i(j)$. Ahora, sea $f(n) = f_n(n) + 1$. Claramente esta función no está en la matriz, pues difiere de cada f_i en el valor $f_i(i)$, y por lo tanto no puede computarse con el nuevo mecanismo. Sin embargo, una MT puede calcularla: Basta enumerar cadenas e ir verificando si representan funciones correctas, deteniéndonos en la n-ésima, simulando su funcionamiento con el argumento n, y finalmente sumando 1 al resultado. $\Box$

El teorema está expresado en términos de funciones de naturales en naturales, pero no es difícil adaptarlo a lo que se desee. El enunciado se ve un poco técnico pero no tiene nada muy restrictivo. Es difícil imaginarse un sistema razonable de computación donde las funciones no puedan enumerarse (esto ya lo hemos discutido en la Sección 4.7), lo que es lo mismo que representarse como una secuencia de alguna forma; o que no se pueda saber si una secuencia representa una descripción sintácticamente correcta de una función; o que no pueda simularse con una MT a partir de la representación (Tesis de Church).

¿En qué falla el teorema si lo tratamos de aplicar a un formalismo donde algunas $f(i)$ pueden no terminar? En que, si resulta que $f_n(n)$ no termina, no podremos "sumarle 1". Nuestro mecanismo propuesto para calcular $f_n(n) + 1$ tampoco terminará. Ni siquiera podremos decidir darle un valor determinado a las que no terminan, porque no tenemos forma de saber si termina o no. Cualquier valor que le demos a $f(n)$, puede que $f_n(n)$ finalmente termine y entregue el mismo valor. Por lo tanto, es *esencial* que haya funciones que no terminan, y que el problema de la detención sea indecidible, para que un sistema de computación sea completo.

5.2 Decidir, Aceptar, Enumerar

En esta sección veremos algunas formas alternativas de pensar en lenguajes aceptables y decidibles, que serán útiles para demostrar la indecidibilidad de otros problemas.

Definición 5.5 *El* lenguaje de salida *de una MT M es el conjunto*

$$\{w \in (\Sigma - \{\#\})^*, \ \exists u \in (\Sigma - \{\#\})^*, \ (s, \#u\underline{\#}) \vdash_M^* (h, \#w\underline{\#}) \ \}$$

En el caso de que M compute una función f, esto es la imagen de f.

El siguiente lema dice que los lenguajes aceptables son los lenguajes de salida.

Lema 5.8 *Un lenguaje es aceptable sii es el lenguaje de salida de alguna MT.*

Prueba: Sea L un lenguaje aceptado por una MT M. Crearemos una MT M' que calcule la función $f(w) = w$ para las $w \in L$ y no se detenga si $w \notin L$. Claramente L será el lenguaje de salida de M'. M' simplemente copia la entrada w a la cinta 2 y corre M en la cinta 2. Si M termina ($w \in L$), la salida que quedará será la de la cinta 1. Si M no termina ($w \notin L$), M' tampoco termina.

Sea ahora L el lenguaje de salida de una MT M. Para aceptar L, una MTND puede generar no determinísticamente una u en una cinta 2, correr M sobre ella en la cinta 2, y detenerse sólo si el resultado de la cinta 2 es igual al de la cinta 1. □

Lo anterior también implica que no siempre será posible decidir si una cierta función computable puede o no entregar un cierto valor (aunque para algunas funciones puede ser posible).

Otra caracterización interesante de los lenguajes aceptables tiene que ver con el usar una MT para enumerar cadenas.

Definición 5.6 *Un lenguaje L es* Turing-enumerable *o simplemente* enumerable *si existe una MT $M = (K, \Sigma, \delta, s)$ y un estado $q \in K$ tal que*

$$L \ = \ \{w \in (\Sigma - \{\#\})^*, \ \exists u \in \Sigma^*, \ (s, \underline{\#}) \vdash_M^* (q, \#w\underline{\#}u)\}$$

Esto significa que M se arranca en la cinta vacía y va dejando las cadenas de L en la cinta, una por vez, pasando por q para indicar que ha generado una nueva cadena, y guardando luego del cabezal la información necesaria para continuar generando cadenas. Nótese que se permite generar una misma cadena varias veces.

Lema 5.9 *Un lenguaje L es enumerable sii es aceptable.*

Prueba: Si L es aceptado por una MT M, podemos crear una MTND que genere todas las cadenas posibles w. Una vez generada una cadena w no determinísticamente, la copia a la cinta 2 y corre M sobre ella. En caso de que M termine, la *simulación* de la MTND no se detiene como en el caso usual, sino que almacena las directivas (recordar Sección 4.5) luego de $\#w\#$ en la cinta 1, y recién retoma la simulación de la MTND.

Si L es enumerado por una MT M, es fácil aceptarlo. Dada una w en la cinta 1, corremos M en la cinta 2. Cada vez que M genera una nueva cadena w' pasando por su estado q, comparamos w con w' y nos detenemos si $w = w'$, sino retomamos la ejecución de M. □

Finalmente, estamos en condiciones de establecer la relación entre GDCs y MTs.

Teorema 5.3 *Un lenguaje L es generado por una GDC sii es aceptable.*

Prueba: Siguiendo el Teo. 4.2, si L es generado por una GDC, existe una MTND que produce cada cadena de L a partir de la cinta vacía. Dada una w en la cinta 1, se corre esta MTND en la cinta 2 y cada vez que genera una cadena, en vez de que se detenga, hacemos que la compare con la w de la cinta 1, deteniéndose sólo si w es igual a la cadena que generó. (Esto se parece a enumerar lenguajes.)

Inversamente, si L es aceptable, entonces es el lenguaje de salida de una MT M. Podemos construir una GDC G que genere las cadenas de ese lenguaje de salida de la siguiente forma. Primero generamos cualquier configuración posible de comienzo con $S \longrightarrow [\#C]$, $C \longrightarrow aC$ para cada $a \in \Sigma - \{\#\}$, y $C \longrightarrow s\#$. Luego, agregamos a la GDC las reglas que le permiten simular M (Lema 4.4), de modo que $(s, \#u\underline{\#}) \vdash_M^* (h, \#w\underline{\#})$ sii $[\#us\#] \Longrightarrow_G^* [\#wh\#]$. Con esto G será capaz de generar todas las cadenas $[\#wh\#]$ tal que $w \in L$. Finalmente, eliminamos los terminadores de la cadena con unas pocas reglas adicionales: $h\#] \longrightarrow X$, $aX \longrightarrow Xa$ para todo $a \in \Sigma - \{\#\}$, y $[\#X \longrightarrow \varepsilon$. $\square$

5.3 Demostrando Indecidibilidad por Reducción

En esta sección veremos cómo se puede usar la indecidibildad del problema de la detención para demostrar que otros problemas también son indecidibles.

La herramienta fundamental es la *reducción*. Si tengo un problema A que quiero probar indecidible, y otro B que sé que es indecidible, la idea es mostrar cómo, con una MT que decidiera A, podría construír una que decidiera B. Diremos que *reducimos* el problema B (que sabemos indecidible) al problema A (que queremos probar indecidible), estableciendo, intuitivamente, que A no es más fácil que B.

Probaremos primero varios resultados de indecidibildad sobre MTs.

Lema 5.10 *Los siguientes problemas sobre MTs son indecidibles:*

1. *Dadas M y w, ¿M se detiene frente a w?*

2. *Dada M, ¿se detiene arrancada con la cinta vacía?*

3. *Dada M, ¿se detiene frente a alguna/toda entrada posible?*

4. *Dadas M_1 y M_2, ¿se detienen frente a las mismas cadenas? Es decir, ¿aceptan el mismo lenguaje?*

5. *Dadas M_1 y M_2 que calculan funciones, ¿calculan la misma función?*

6. *Dada M, ¿el lenguaje que acepta M es finito? ¿regular? ¿libre del contexto? ¿decidible?*

Prueba:

1. Es exactamente lo que probamos en el Teo. 5.1.

2. Supongamos que pudiéramos decidir ese problema. Entonces, para decidir 1. con M y w, crearíamos una nueva $M' = \rhd w_1 \rhd w_2 \ldots \rhd w_{|w|} \rhd M$. Esta M', arrancada en la cinta vacía, primero escribe w y luego arranca M, de modo que M' termina frente a la cinta vacía sii M acepta w.

3. Si pudiéramos decidir ese problema, podríamos resolver 2. para una MT M creando $M' = BM$ (Def. 4.10), que borra la entrada y luego arranca M sobre la cinta vacía. Entonces M' se detiene frente a alguna/toda entrada sii M se detiene frente a la cinta vacía.

4. Si pudiéramos decidir ese problema, podríamos resolver 3. para una MT M, creando una M_2 que nunca se detuviera frente a nada ($\delta(s,a) = (s,a)$ para todo $a \in \Sigma$) y preguntando si $M_1 = M$ se detiene frente a las mismas entradas que M_2 (para el caso "se detiene frente a alguna entrada posible"; para el otro caso, la M_2 se debería detener siempre, $\delta(s,a) = (h,a)$).

5. Si pudiéramos decidir ese problema, podríamos resolver 1. para una MT M y una cadena w. Haríamos una M' a partir de (M,w) que calcule $f(n) = 1$ si M acepta w en menos de n pasos, y $f(n) = 0$ sino. Es fácil construir M' a partir de M, mediante simularla durante n pasos y responder. Ahora haremos una M_0 que calcule $f_0(n) = 0$ para todo n. Entonces M se detiene frente a w sii M' y M_0 no calculan la misma función.

6. Si pudiéramos decidir ese problema, podríamos resolver 2. para una MT M, de la siguiente forma. Crearíamos una MT M' que primero corriera M en la cinta 2, y en caso de que M terminara, corriera la MUT (Def. 4.23) sobre la entrada real de la cinta 1. Entonces, si M no se detiene frente a la cinta vacía, M' no acepta ninguna cadena (pues nunca llega a mirar la entrada, se queda pegada en correr M en la cinta 2) y por lo tanto el lenguaje que acepta M' es $\emptyset$. Si M se detiene, entonces M' se comporta exactamente como la MUT, por lo que acepta K_0. Podemos entonces saber si M se detiene o no frente a la cinta vacía porque $\emptyset$ es finito, regular, libre del contexto y decidible, mientras que K_0 no es ninguna de esas cosas.

$\square$

Veamos ahora algunas reducciones relacionadas con GDCs.

Lema 5.11 *Los siguientes problemas sobre GDCs son indecidibles:*

1. *Dadas G y w, ¿G genera w?*

2. *Dada G, ¿genera alguna cadena? ¿genera toda cadena posible?*

3. *Dadas G_1 y G_2, ¿generan el mismo lenguaje?*

4. *Dada G y $w, z \in (V \cup \Sigma)^*$, ¿$w \Longrightarrow_G^* z$?*

Prueba:

1. Si esto fuera posible, tomaríamos cualquier lenguaje aceptable, construiríamos la GDC G que lo genera como en el Teo. 5.3, y podríamos decidir si una w dada está en el lenguaje o no.

2. Si esto fuera posible, podríamos saber si una MT acepta alguna/toda cadena o no (punto 3. en el Lema 5.10), mediante obtener nuevamente la GDC que genera el lenguaje que la MT acepta y preguntarnos si esa GDC genera alguna/toda cadena.

3. Si esto fuera posible, podríamos saber si dos MTs aceptan el mismo lenguaje (punto 4. en el Lema 5.10), mediante obtener las GDCs que generan los lenguajes que las MT aceptan y preguntarnos si son iguales.

4. Si esto fuera posible, podríamos saber si una MT M se detiene frente a la cinta vacía (punto 2. en el Lema 5.10), mediante generar una M' que ejecute M en la cinta 2. Esta M' va de $(s, \underline{\#})$ a $(h, \underline{\#})$ sii M se detiene frente a la cinta vacía. Si ahora aplicamos la construcción del Lema 4.4, obtenemos una G que lleva de $[s\#]$ a $[h\#]$ sii M se detiene frente a la cinta vacía.

$\square$

5.4 Otros Problemas Indecidibles

Hasta ahora hemos obtenido problemas indecidibles relacionados con MTs y GDCs. Esto puede ser lo más natural, pero es esperable que podamos exhibir problemas indecidibles de tipo más general.

Comencemos con un problema que tiene semejanza con un juego de dominó.

Definición 5.7 *Un* sistema de correspondencia de Post *es un conjunto finito de pares* $P = \{(u_1, v_1),\ (u_2, v_2), \ldots, (u_n, v_n)\}$*, con* $u_i, v_i \in \Sigma^+$*. El* problema de correspondencia de Post *es, dado un sistema* P*, determinar si existe una cadena* $w \in \Sigma^+$ *(llamada* solución*) tal que* $w = u_{i_1} u_{i_2} \ldots u_{i_k} = v_{i_1} v_{i_2} \ldots v_{i_k}$ *para una secuencia* $i_1,\ i_2, \ldots, i_k$ *de pares de* P *a usar (los pares pueden repetirse).*

Ejemplo 5.1 El problema de correspondencia de Post para el sistema $P = \{(ca, a), (a, ab), (b, ca), (abc, c)\}$ tiene solución $abcaaabc$, que se obtiene utilizando los pares $(a, ab), (b, ca), (ca, a), (a, ab), (abc, c)$: concatenando las primeras o las segundas componentes se obtiene la misma cadena. En este caso $i_1 = 2$, $i_2 = 3$, $i_3 = 1$, $i_4 = 2$, $i_5 = 4$. Es útil ver los pares como fichas de dominó:

ca	a	b	abc
a	ab	ca	c

donde buscamos pegar una secuencia de fichas de modo de que se lea la misma cadena arriba y abajo:

a	b	ca	a	abc
ab	ca	a	ab	c

Una variante del problema, útil para demostrar varios resultados, es la siguiente, donde se fija cuál debe ser el primer par a usar.

Definición 5.8 *Un* sistema de Post modificado *es un par* $(P, (x, y))$*, donde* P *es un sistema de correspondencia de Post y* $(x, y) \in P$*. El* problema de Post modificado *es encontrar una cadena* $w \in \Sigma^+$ *tal que* $w = u_{i_1} u_{i_2} \ldots u_{i_k} = v_{i_1} v_{i_2} \ldots v_{i_k}$ *para una secuencia* $i_1,\ i_2, \ldots, i_k$*, donde* i_1 *corresponde a* (x, y)*.*

Lo primero a mostrar es que resolver un sistema de Post no es más fácil que resolver uno modificado.

Lema 5.12 *Dado un sistema de Post modificado* $(P, (x, y))$*, podemos crear un sistema de Post* P' *tal que* $(P, (x, y))$ *tiene solución sii* P' *la tiene.*

Prueba: Debemos crear P' de manera de obligar a que (x, y) sea el primer par utilizado. Definamos $L(w) = *w_1 * w_2 \ldots * w_{|w|}$ y $R(w) = w_1 * w_2 * \ldots w_{|w|}*$. Comenzaremos insertando $(L(x)*, L(y))$, que será necesariamente el primer par a usar pues en todos los demás una cadena comenzará con $*$ y la otra no. Para cada $(u, v) \in P$ (incluyendo (x, y), que puede usarse nuevamente dentro), insertamos $(R(u), L(v))$ en P'. Para poder terminar la cadena, insertamos $(\$, *\$)$ en P'. Los símbolos $*$ y $\$$ no están en Σ. Toda solución de P comenzando con (x, y) tendrá su solución equivalente en P', y viceversa. $\Box$

El siguiente lema puede parecer sorprendente, pues uno esperaría poder resolver este problema automáticamente.

Teorema 5.4 *El problema de correspondencia modificado de Post es indecidible, y por lo tanto el original también lo es.*

Prueba: Supongamos que queremos saber si $w \Longrightarrow_G^* z$ para una GDC $G = (V, \Sigma, R, S)$ y cadenas w, z cualquiera, con la restricción de que $u \longrightarrow v \in R$ implica $v \neq \varepsilon$. Crearemos un sistema de Post modificado que tendrá solución sii $w \Longrightarrow_G^* z$. Como esto último es indecidible por el Lema 5.11 (donde las G para las que lo demostramos efectivamente no tienen reglas del tipo $u \longrightarrow \varepsilon$), los sistemas de Post son indecidibles en general también.

En este sistema, de alfabeto $V \cup \Sigma \cup \{*\}$, tendremos un par inicial $(x, y) = (*, *w*)$, donde $* \notin V \cup \Sigma$. Tendremos asimismo todos los pares (a, a), $a \in V \cup \Sigma \cup \{*\}$, y también los (u, v), $u \longrightarrow v \in R$. Finalmente, tendremos el par $(z**, *)$.

Es fácil ver que existe una secuencia de derivaciones que lleva de w a z sii este sistema modificado tiene solución. La idea es que comenzamos con w abajo, y luego la "leemos" calzándola arriba y copiándola nuevamente abajo, hasta el punto en que decidimos aplicar una regla $u \longrightarrow v$. Al terminar de leer w, hemos creado una nueva w' abajo donde se ha aplicado una regla, $w \Longrightarrow w'$. Si terminamos produciendo z, podremos calzar la cadena completa del sistema de Post. $\Box$

Ejemplo 5.2 Supongamos que queremos saber si $S \Longrightarrow_G^* aaabbb$, donde las reglas son $S \longrightarrow aSb$ y $S \longrightarrow ab$. El sistema de Post modificado asociado es

$$P = \{(*, *S*),\ (a, a),\ (b, b),\ (*, *),\ (S, aSb),\ (S, ab),\ (aaabbb**, *)\}$$

y $(x, y) = (*, *S*)$. El calce que simula la derivación, escribiendo los pares con la primera componente arriba y la segunda abajo, es

$*$	S	$*$	a	S	b	$*$	a	a	S	b	b	$*$	$aaabbb**$
$*S*$	aSb	$*$	a	aSb	b	$*$	a	a	ab	b	b	$*$	$*$

donde, por ejemplo, para cuando logramos calzar el $*S*$ de abajo con las cadenas de arriba, ya hemos generado $aSb*$ abajo para calzar. Ilustra ver la misma concatenación de fichas, ahora alineando las secuencias.

$$
\begin{array}{l}
* \mid S \mid * \mid a \mid S \mid b \mid * \mid a \mid a \mid S \mid b \mid b \mid * \mid a\ a\ a\ b\ b\ b\ *\ * \mid \\
*\ S\ * \mid a\ S\ b \mid * \mid a \mid a\ S\ b \mid b \mid * \mid a \mid a \mid a\ b \mid b \mid b \mid * \mid * \mid
\end{array}
$$

El hecho de que los sistemas de Post sean indecidibles nos permite, mediante reducciones, demostrar que ciertos problemas sobre GLCs son indecidibles.

Teorema 5.5 *Dadas dos GLCs G_1 y G_2, es indecidible saber si $\mathcal{L}(G_1) \cap \mathcal{L}(G_2) = \emptyset$ o no.*

Prueba: Podemos reducir la solución de un sistema de Post a este problema. Sea $P = \{(u_1,v_1),\ (u_2,v_2),\dots,(u_n,v_n)\}$ nuestro sistema, sobre un alfabeto Σ. Definiremos $G_1 = (\{S_1\}, \Sigma \cup \{c\}, R_1, S_1)$, con $c \notin \Sigma$ y R_1 conteniendo las reglas $S_1 \longrightarrow u_i^R S_1 v_i \mid u_i^R c v_i$ para $1 \le i \le n$. Por otro lado, definiremos $G_2 = (\{S_2\}, \Sigma \cup \{c\}, R_2, S_2)$, con R_2 conteniendo $S_2 \longrightarrow aS_2a$ para todo $a \in \Sigma$ y $S_2 \longrightarrow c$. Entonces $\mathcal{L}(G_1) = \{u_{i_n}^R \dots u_{i_2}^R u_{i_1}^R c v_{i_1} v_{i_2} \dots v_{i_k},\ k > 0,\ 1 \le i_1,\ i_2,\dots,i_k \le n\}$ y $\mathcal{L}(G_2) = \{w^R c w,\ w \in \Sigma^*\}$. Está claro que P tiene solución sii $\mathcal{L}(G_1)$ y $\mathcal{L}(G_2)$ tienen intersección. $\square$

Otro problema importante en GLCs es determinar si una G es ambigua (Def. 3.7). Este problema tampoco es decidible.

Teorema 5.6 *Dada una GLC G, es indecidible saber si G es ambigua.*

Prueba: Podemos reducir la solución de un sistema de Post a este problema. Sea $P = \{(u_1,v_1),\ (u_2,v_2),\dots,(u_n,v_n)\}$ nuestro sistema, sobre un alfabeto Σ. Definiremos $G = (\{S,S_1,S_2\}, \Sigma \cup \{a_1,a_2,\dots,a_n\}, R, S)$, con $a_i \notin \Sigma$ y R conteniendo las reglas $S \longrightarrow S_1 \mid S_2$, $S_1 \longrightarrow a_i S_1 u_i \mid a_i u_i$ para cada i, y $S_2 \longrightarrow a_i S_2 v_i \mid a_i v_i$ para cada i. De S_1, entonces, se generan todas las cadenas de la forma $a_{i_k} \dots a_{i_2} a_{i_1} u_{i_1} u_{i_2} \dots u_{i_k}$, mientras que de S_2 se generan todas las cadenas de la forma $a_{i_k} \dots a_{i_2} a_{i_1} v_{i_1} v_{i_2} \dots v_{i_k}$. Está claro que G es ambigua sii hay una forma de generar una misma cadena, usando los mismos pares, mediante las u's y mediante las v's, es decir, si P tiene solución. $\square$

Finalmente, volveremos a utilizar MTs para demostrar que el siguiente problema, de encontrar una forma de embaldosar un piso infinito (o resolver un rompecabezas), no tiene solución algorítmica.

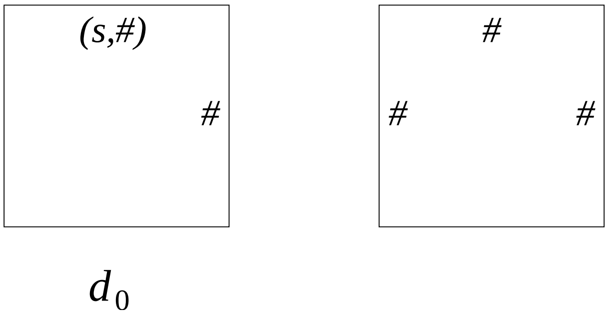
Definición 5.9 *Un* sistema de baldosas *es una tupla (D, d_0, H, V), donde D es un conjunto finito, $d_0 \in D$, y $H, V \subseteq D \times D$. Un* embaldosado *es una función $f : \mathbb{N} \times \mathbb{N} \longrightarrow D$, que asigna baldosas a cada celda de una matriz infinita (pero que comienza en una esquina), donde $f(0,0) = d_0$, y para todo $m, n \ge 0$, $(f(n,m), f(n+1,m)) \in V$, $(f(n,m), f(n,m+1)) \in H$.*

La idea es que H dice qué baldosas pueden ponerse a la derecha de qué otras, mientras que V dice qué baldosas pueden ponerse arriba de qué otras.

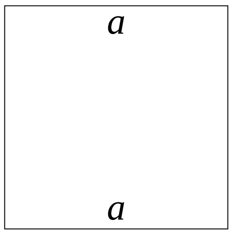
Teorema 5.7 *El problema de, dado un sistema de baldosas, determinar si existe un embaldosado, es indecidible.*

Prueba: La idea es definir un sistema de baldosas a partir de una MT M, de modo que M se detenga frente a la cinta vacía sii no existe un embaldosado. La fila i de la matriz corresponderá a la cinta de M en el paso i. Las baldosas tienen símbolos escritos en algunos de sus lados, y las reglas H y V indican que, para poner una baldosa pegada a la otra, lo que tienen escrito en los lados que se tocan debe ser igual.

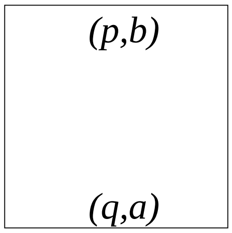
Las celdas de la primera fila son especiales: d_0 es la baldosa de la izquierda, y las demás baldosas de la primera fila sólo podrán ser escogidas iguales a la baldosa de la derecha.



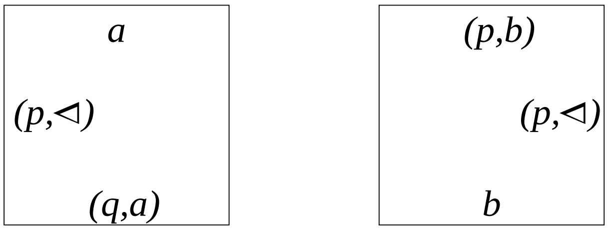
El lado superior $(s, \#)$ indica, además del carácter en la cinta, que el cabezal está en esa celda y el estado. Las celdas que están lejos del cabezal no cambian entre el paso i e $i + 1$. Para esto existen baldosas



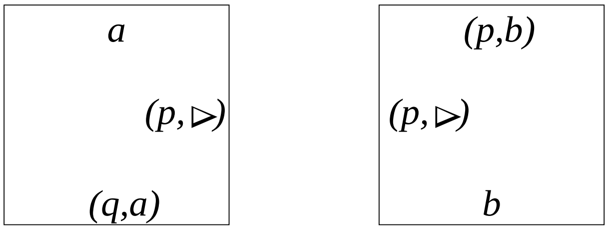
para cada $a \in \Sigma$. Para poner la baldosa que va arriba de la del cabezal, las reglas dependerán del δ de M. Si $\delta(q, a) = (p, b)$ con $b \in \Sigma$, tendremos una baldosa que nos permita continuar hacia arriba a partir de una baldosa rotulada (q, a) en el borde superior:



En cambio, si $\delta(q, a) = (p, \lhd)$, necesitamos baldosas que nos permitan indicar que el cabezal se ha movido hacia la izquierda:



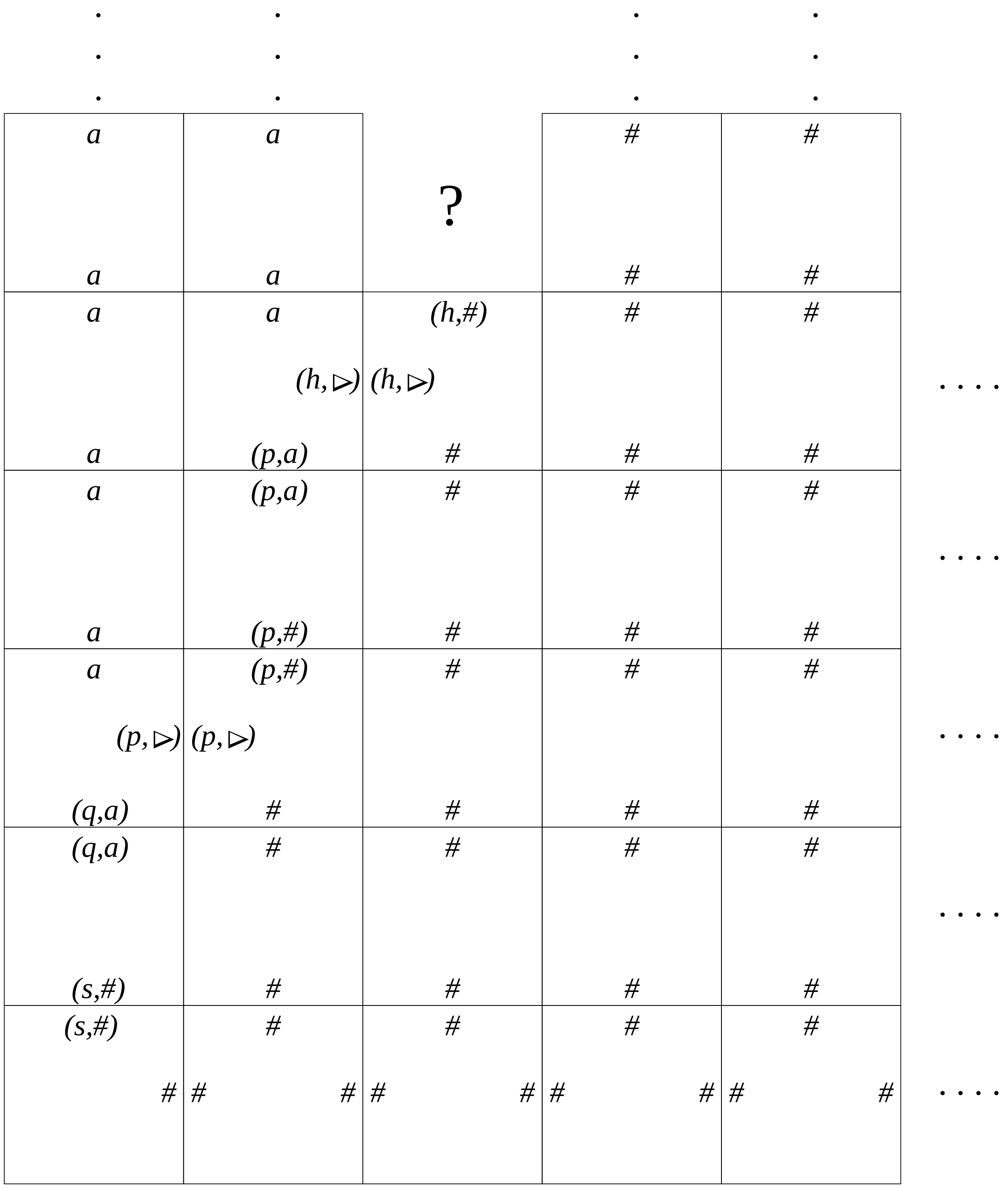
para cada $b \in \Sigma$, y similarmente para el caso $\delta(q, a) = (p, \rhd)$:



Es fácil ver que, si M se detiene frente a la cinta vacía, entonces llegaremos en algún paso i a una celda con el borde superior rotulado (h, b) para algún $b \in \Sigma$. Como no existe regla para poner ninguna baldosa sobre ésta, será imposible encontrar un embaldosado para este sistema. Si, en cambio, M corre para siempre, será posible embaldosar toda la matriz infinita.

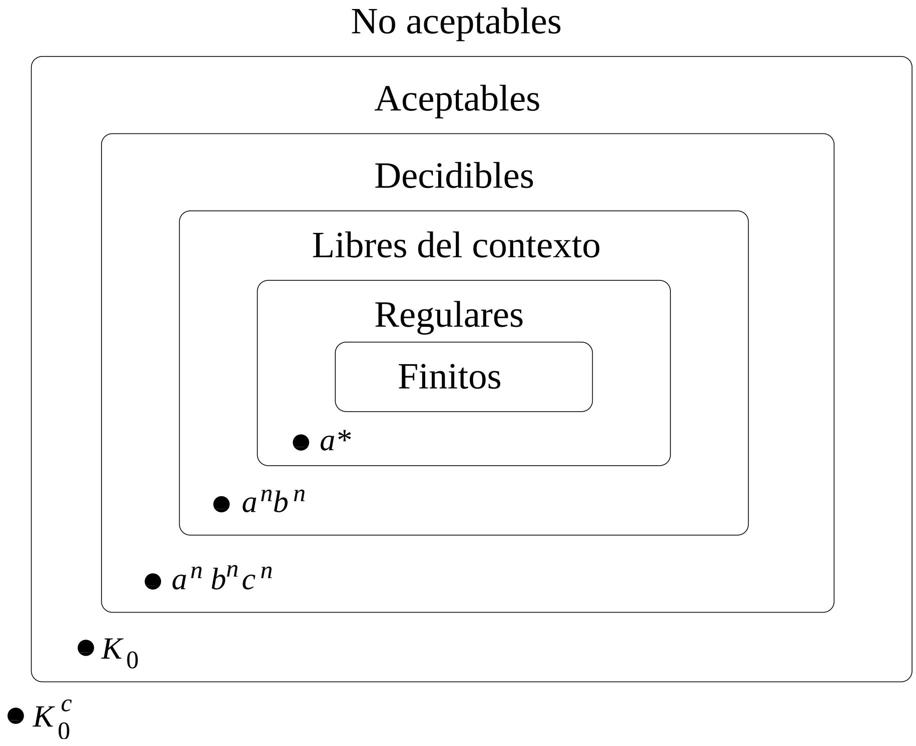
Existe un pequeño problema técnico si M se cuelga. Esto se puede evitar poniendo una marca \$ antes de la primera celda, y en caso de que M llegue a esa marca (indicando que se colgó), modificarla para que se quede infinitamente escribiendo \$ sobre la marca. De este modo, si M se cuelga, nuestra modificación no termina. $\square$

Ejemplo 5.3 Supongamos una MT M con las reglas $\delta(s, \#) = (q, a)$, $\delta(q, a) = (p, \rhd)$, $\delta(p, \#) = (p, a)$, $\delta(p, a) = (h, \rhd)$. El embaldosado correspondiente a esta MT es el siguiente, donde queda claro que no se puede embaldosar el plano infinito:



a a # #
?
a a # #
a a (h,#) # #
(h,▷) (h,▷)
a (p,a) # # #
a (p,a) # # #
a (p,#) # # #
a (p,#) # # #
(p,▷) (p,▷)
(q,a) # # # #
(q,a) # # # #
(s,#) # # # #
(s,#) # # # #
# # # # # # # # #

Terminaremos este capítulo ilustrando la jerarquía de lenguajes que hemos obtenido. Esta se refinará en el próximo capítulo.



5.5 Ejercicios

1. Un *autómata finito universal* es un autómata finito que recibe como entrada la codificación de un autómata finito y la de una cadena de entrada, es decir $\rho(M)\rho(w)$, y se comporta sobre $\rho(w)$ igual que como lo haría M. Explique por qué no puede existir un autómata finito universal.

2. Muestre que la unión y la intersección de lenguajes Turing-aceptables es Turing-aceptable (aunque no el complemento, como vimos).

3. Muestre que la unión, intersección, complemento, concatenación y clausura de Kleene de lenguajes Turing-decidibles es Turing-decidible.

4. Muestre que cualquier conjunto finito es Turing-decidible.

5. Muestre que L es Turing-aceptable si y sólo si, para alguna MTND M, $L = \{w \in \Sigma^*,\ (s, \underline{\#}) \vdash_M^* (h, \#w\underline{\#})\}$, donde s es el estado inicial de M.

6. Sea Σ un alfabeto que no contiene la letra c. Suponga que $L \subseteq \{w_1 c w_2,\ w_1, w_2 \in \Sigma^*\}$ es Turing-aceptable. Muestre que $L' = \{w_1,\ \exists w_2,\ w_1 c w_2 \in L\}$ es Turing-aceptable. Si L es decidible, ¿necesariamente L' es decidible?

7. Suponga que, para ahorrar espacio, quiere construir un algoritmo que minimice máquinas de Turing: dada una MT M, genera otra que acepta el mismo lenguaje pero tiene el menor número posible de estados. Dentro de las que tienen el menor número de estados, devuelve la que genera, lexicográficamente, la menor representación $\rho(M')$. Muestre que tal algoritmo no puede existir.

8. ¿Cuáles de los siguientes problemas problemas se pueden resolver por algoritmo y cuáles no? Explique.

   (a) Dadas M y w, determinar si M alguna vez alcanza el estado q al ser arrancada sobre w a partir del estado s.

   (b) Dadas M, w y un símbolo $a \in \Sigma$, determinar si M alguna vez escribe el símbolo a en la cinta, al ser arrancada sobre w a partir del estado s.

   (c) Dadas M_1 y M_2, determinar si hay una cadena w en la cual ambas se detienen.

   (d) Dada M, determinar si alguna vez mueve el cabezal hacia la izquierda, arrancada en el estado s sobre la cinta vacía.

   (e) Dada M, determinar si alguna vez llegará a una cierta posición n de la cinta.

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

a) Si se pudiera resolver el problema de la detención, todo lenguaje aceptable sería decidible.

b) Si se pudiera resolver el problema de la detención, todos los lenguajes serían decidibles.

c) Hay lenguajes que no son aceptables y su complemento tampoco lo es.

d) Dada una MT, hay un algoritmo para saber si es equivalente a un autómata finito (en términos del lenguaje que acepta).

e) Si una MT se pudiera mover sólamente hacia adelante y no escribir, sólo podría aceptar lenguajes regulares.

C2 1996 Se tiene una función *biyectiva* y Turing-computable $f : \Sigma^* \longrightarrow \Sigma^*$.

Demuestre que la inversa $f^{-1} : \Sigma^* \longrightarrow \Sigma^*$ también es Turing-computable. La demostración debe ser precisa y suficientemente detallada para que no queden dudas de su validez.

C2 1997 Sea M una máquina que recibe una cadena w y arranca la Máquina Universal de Turing (MUT) con la entrada $w\rho(w)$. Se pregunta si M acepta la cadena $\rho(M)$.

a) (1/5) Identifique, del material que hemos visto, qué lenguaje acepta M.

b) (2/5) A partir de la definición del lenguaje que acepta M, ¿puede deducir la respuesta a la pregunta? Comente.

c) (2/5) Considere la máquina operacionalmente y lo que hará la MUT cuando reciba su entrada y empiece a procesarla. ¿Puede responder ahora la pregunta? ¿La respuesta acarrea alguna contradicción con *b*)? Explique.

C2 1997 Sea f una función de Σ^* en Σ^*, tal que $\$ \notin \Sigma$.

- Demuestre que si el lenguaje $L = \{u \ \$ \ f(u),\ u \in \Sigma^*\}$ es decidible, entonces f es computable.
- Demuestre que si L es aceptable, f sigue siendo computable.

Ex 1997 Se propone el siguiente modelo de computación: se usan MTs como siempre pero con la restricción de que si la entrada tiene largo ℓ, entonces la máquina no podrá acceder las celdas más allá de la posición ℓ (si lo intenta hacer se cuelga igual que al moverse a la izquierda de la primera celda).

Mostrar que sobre estas máquinas el problema de la detención se vuelve decidible.

ExRec 1997 Un *oráculo* es un mecanismo (no necesariamente posible de construír) capaz de responder alguna pregunta indecidible.

Suponga que tiene un oráculo capaz de decidir el problema de la detención, y lo considera como un mecanismo utilizable (es decir, el problema se hace de repente decidible). Responda justificando brevemente (máximo 5 líneas).

a) ¿Todos los lenguajes aceptables pasarían ahora a ser decidibles? ¿Quedarían lenguajes no decidibles? ¿Quedarían lenguajes no aceptables?

b) Las mismas preguntas anteriores si dispusiera (a su elección) de una cantidad numerable de los oráculos que necesite.

c) Las mismas anteriores si la cantidad de oráculos pudiera ser no numerable.

Ex 1998 Considere una MT que calcula funciones a la cual, frente a la entrada w, se le permite realizar a lo sumo $f(|w|)$ pasos, donde f es una función fija para la MT. Si para ese momento no terminó se supone que su respuesta es, digamos, la cadena vacía.

Pruebe que con este tipo de máquinas no se pueden calcular todas las funciones Turing-computables.

Ex 1999 Dibuje un diagrama de conjuntos para indicar las relaciones de inclusión e intersección de las siguientes clases de lenguajes: regulares (R), libres del contexto (LC), todos los complementos de L (CLC), recursivos (C), recursivos enumerables (RE) y todos los complementos de RE (CRE). Justifique brevemente.

C1 2001 Un *autómata de dos pilas* permite operar sobre ambas pilas a la vez, y acepta una cadena cuando llega a un estado final, sin importar el 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) Demuestre que los lenguajes aceptados por un autómata de dos pilas son exactamente los lenguajes Turing-aceptables.

C2 2002 Sea M_1 la MT que acepta el lenguaje K_1, mediante convertir la entrada $\rho(M)$ en $\rho(M)\rho(\rho(M))$ e invocar la MUT.

(a) Determine si $\rho(M_1) \in K_1$ o no. Explique su respuesta.

(b) Cualquiera haya sido su respuesta a la pregunta anterior, ¿es posible rediseñar M_1 para que la respuesta cambie, y siga siendo válido que M_1 acepta K_1?

Ex 2002 Considere la MGT del [Ex 2002] (página 105).

- ¿Puede hacer una MT que acepte $\mathcal{L}(MGT)^c$? Explique cómo o demuestre que no puede.
- ¿Puede hacer una MT que decida si $w \in L(G)$? Explique cómo o demuestre que no puede.
- Vuelva a responder la pregunta anterior suponiendo ahora que G es una gramática libre del contexto.

Ex 2003 Argumente de la forma más clara posible, sin exceder las 5 líneas, por qué los siguientes problemas son o no son computables.

(a) Dado un programa en C, determinar si es posible que una determinada variable se examine antes de que se le asigne un valor.

(b) Dado un programa en C, determinar si es posible llegar a una determinada línea del código sin antes haber ejecutado una determinada función.

(*c*) Dado un programa en C, determinar si es posible que termine antes de haber pasado más de una vez por alguna línea del programa (considere que se escribe a lo sumo un comando por línea).

C2 2004 Se define la *distancia de edición generalizada* de la siguiente forma. Se da un conjunto finito de sustituciones permitidas (α, β, c), con $\alpha \neq \beta \in \Sigma^*$ y $c \in \mathbb{R}^+$ el costo de la sustitución. Dada una cadena x, se le pueden aplicar repetidamente transformaciones de la forma: buscar un substring α y reemplazarlo por β. El costo de una serie de transformaciones es la suma de los costos individuales. Dadas dos cadenas x e y, la distancia de edición generalizada de x a y es el mínimo costo de transformar x en y, si esto es posible, e infinito sino.

Demuestre que la distancia de edición generalizada no es computable.

Ex 2004

(*a*) Demuestre que el problema de, dada una MT M, una entrada w, y un número k, saber si M llega a la k-ésima celda de la cinta frente a la entrada w, es decidible.

(*b*) Demuestre que el problema de, dada una MT M y una entrada w, saber si llega a la k-ésima celda para todo k (es decir, no usa una porción finita de la cinta), no es decidible.

(*c*) Un lenguaje es *sensitivo al contexto* si es generado por una gramática dependiente del contexto donde toda regla $u \rightarrow v$ cumple $|u| \leq |v|$ (por ejemplo, $\{a^n b^n c^n,\ n \geq 0\}$ es sensitivo al contexto). Demuestre que los lenguajes sensitivos al contexto son Turing-decidibles.

Ex 2005 De los siguientes productos ofrecidos por MediocreSoft, indique cuáles podrían, al menos en principio, funcionar de acuerdo a la descripción que se entrega, y cuáles no. Argumente brevemente.

***MediocreDebug!*:** Recibe como entrada el texto de un programa en Pascal e indica la presencia de loops o recursiones que, para alguna entrada al programa Pascal, no terminan.

***MediocreRegular!*:** Un sencillo lenguaje de programación pensado para producir parsers de lenguajes regulares. Permite variables globales escalares (no arreglos ni punteros), sin recursión. Hay un buffer de 1 carácter para leer la entrada, en la que no se puede retroceder. El software es un compilador que recibe como entrada un programa escrito en *MediocreRegular!*, lo compila y produce un ejecutable que parsea un determinado lenguaje regular. ¡Con *MediocreRegular!* usted podrá parsear cualquier lenguaje regular!

***MediocreContextFree!*:** Un maravilloso lenguaje de programación pensado para producir parsers de lenguajes libres del contexto. Permite variables globales escalares (no arreglos ni punteros), sin recursión. Se usa de la misma forma que *MediocreRegular!*, ¡y le permitirá parsear cualquier lenguaje libre del contexto!

***MediocreContextFree! (Professional)*:** Un completo lenguaje de programación para iniciados, con el mismo objetivo y forma de uso que *MediocreContextFree!*. El lenguaje ahora permite variables locales y recursión. Usted podrá parsear cualquier lenguaje libre del contexto, con más herramientas que *MediocreContextFree!*

***MediocreTuring!*:** Un editor gráfico de Máquinas de Turing, que incluye una herramienta simplificadora de máquinas. ¡Usted diseña su máquina y *MediocreTuring!* se la simplifica hasta obtener la menor máquina posible que acepta el mismo lenguaje que la que usted diseñó!

C2 2006 Demuestre que el problema de, dadas dos Máquinas de Turing M_1 y M_2 que computan funciones f_1 y f_2 de Σ_0^* en Σ_1^*, determinar si $f_1 = f_2$ es indecidible.

Ayuda: considere la función $f_M(w\$I^n) = \mathtt{S}$ si M se detiene frente a w en a lo sumo n pasos, y $\mathtt{N}$ en otro caso.

C2 2006 Demuestre que la unión e intersección de lenguajes Turing-aceptables es Turing-aceptable, pero no la diferencia.

Ex 2006 Harta de recibir críticas por que sus programas se cuelgan, la empresa MediocreSoft decide cambiar de lenguaje de programación para que sea imposible que tal cosa ocurra. En este nuevo lenguaje existen dos instrucciones de salto. Una es el **if-then-else** de siempre, y la otra es un **while** restringido: **while** ($expr_1$) **do** *sent* **max** ($expr_2$) ejecutará *sent* mientras $expr_1 \neq 0$, pero a lo sumo $expr_2$ veces. Si luego de $expr_2$ iteraciones el **while** no ha terminado, el programa aborta dando un error (pero no se queda pegado jamás). Esta $expr_2$ debe evaluar a un entero positivo, y no se vuelve a evaluar luego de cada iteración, sino sólo al comienzo del **while**.

Comente sobre el futuro de MediocreSoft cuando sus programadores sean obligados a usar este lenguaje.

5.7 Proyectos

1. Investigue sobre funciones recursivas primitivas y μ-recursivas como modelo alternativo de computación, esta vez con énfasis en el tema de la terminación. Una fuente es [LP81, sec. 5.3 a 5.6].

2. Investigue sobre otras preguntas indecidibles sobre GLCs. Una fuente es [Kel95, pág. 261 a 263], otra es [HMU01, sec. 9.5.2 a 9.5.4].

3. Investigue sobre oráculos y grados de indecidibilidad. Una fuente es [DW83, cap. 5].

4. Investigue sobre la complejidad de Kolmogorov y su relación con la computabilidad. Una fuente es [LV93].

5. Lea otros libros más de divulgación, pero muy entretenidos, sobre computabilidad, por ejemplo [Hof99] o [PG02]. ¿Es algorítmico nuestro pensamiento?

Referencias

[DW83] M. Davis, E. Weyuker. *Computability, Complexity, and Languages.* Academic Press, 1983.

[HMU01] J. Hopcroft, R. Motwani, J. Ullman. *Introduction to Automata Theory, Languages, and Computation.* 2nd Edition. Pearson Education, 2001.

[Hof99] D. Hofstadter. *Gödel, Escher, Bach: An Eternal Golden Braid.* Basic Books, 1999. (Hay ediciones anteriores, la primera de 1979.)

[Kel95] D. Kelley. *Teoría de Autómatas y Lenguajes Formales.* Prentice Hall, 1995.

[LP81] H. Lewis, C. Papadimitriou. *Elements of the Theory of Computation.* Prentice-Hall, 1981. Existe una segunda edición, bastante parecida, de 1998.

[LV93] M. Li, P. Vitányi. *An Introduction to Kolmogorov Complexity and Its Applications.* Springer-Verlag, 1993.

[PG02] R. Penrose, M. Gardner. *The Emperor's New Mind.* Oxford University Press, 2002.

Capítulo 6

Complejidad Computacional

[LP81, cap 7], [AHU74, cap 10 y 11]

En este capítulo nos preocuparemos por primera vez del *tiempo* que demora una MT en calcular una función o decidir un lenguaje. Todos los problemas que consideraremos serán decidibles, pero trataremos de distinguir, dentro de ellos, cuáles son más "fáciles" que otros.

Nuevamente identificaremos *decidir lenguajes* (saber si una cadena está en un conjunto) con *resolver problemas de decisión*, es decir responder sí o no frente a una entrada.

6.1 Tiempo de Computación *[LP81, sec 7.1]*

Comenzaremos definiendo el tiempo de una computación basándonos en el número de pasos que realiza una MT.

Definición 6.1 *Una* computación de n pasos *de una MT M es una secuencia de configuraciones $C_0 \vdash_M C_1 \vdash_M C_2 \ldots \vdash_M C_n$. Diremos que C_0* lleva en n pasos *a C_n y lo denotaremos $C_0 \vdash_M^n C_n$.*

Definición 6.2 *Diremos que $M = (K, \Sigma, \delta, s)$* calcula $f(w)$ en n pasos *si $(s, \#w\underline{\#}) \vdash_M^n (h, \#f(w)\underline{\#})$.*

Dada una MT M que calcula una cierta función f, queremos dar una noción de cuánto tiempo le toma a M calcular f. Decir cuánto demora para cada entrada w posible da más detalle del que nos interesa. Lo que quisiéramos es dar el tiempo en función del *largo* de la entrada. Un problema es que distintas entradas de distinto largo pueden requerir una cantidad de pasos distinta. Lo que usaremos será la noción de *peor caso* de algoritmos: nos interesará el *mayor* tiempo posible dentro de las entradas de un cierto largo.

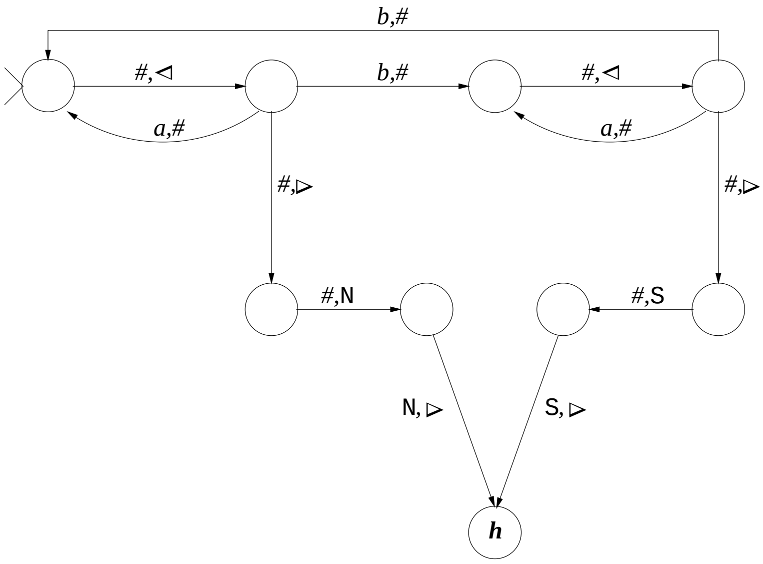
Definición 6.3 *Diremos que M* computa $f : \Sigma_0^* \rightarrow \Sigma_1^*$ en tiempo $T(n)$ *si, para cada $w \in \Sigma_0^*$, M calcula $f(w)$ en a lo sumo $T(|w|)$ pasos. Similarmente, diremos que M* decide L en tiempo $T(n)$ *si M calcula f_L en tiempo $T(n)$ (ver Def. 4.7).*

Es interesante que hay un mínimo de pasos en el que se puede decidir un lenguaje.

Lema 6.1 *No es posible decidir un lenguaje en tiempo menor a $T(n) = 2n + 4$.*

Prueba: Sólo para retroceder desde la configuración $\#w\underline{\#}$ borrando la entrada hasta detectar el comienzo de la cinta se necesitan $2n + 1$ pasos. Luego se necesitan a lo menos 3 pasos más para escribir `S` o `N` y posicionarse en la siguiente celda. □

Ejemplo 6.1 Una MT que decide $L = \{w \in \{a, b\}^*,\ w$ tiene una cantidad impar de b's $\}$ (recordar Ej. 2.7) puede ser como sigue (en notación no modular).



Es fácil ver que esta MT requiere siempre $2n + 4$ pasos para una entrada de largo n.

Observación 6.1 *Es fácil darse cuenta de que cualquier lenguaje regular se puede decidir en tiempo $2n + 4$, mediante modificar sistemáticamente el AFD que lo reconoce para que lea la cadena al revés, la vaya borrando, y luego escriba* ***S*** *o* ***N*** *según haya quedado en un estado final o no. ¿Qué pasa con los libres del contexto? Por otro lado, sí es posible calcular funciones más rápidamente: la función identidad se calcula en un sólo paso.*

Ejemplo 6.2 Bastante más complicado es calcular el tiempo que tarda la MT del Ej. 4.10 en decidir su lenguaje. Un análisis cuidadoso muestra que, si $w \in L$, el tiempo es $n^2/2 + 13n/2$, donde $n = |w|$, pero puede ser bastante menor (hasta $2n + 6$) si $w \notin L$.

En general, determinar la función exacta $T(n)$ para una MT puede ser bastante difícil *¡De hecho no es computable siquiera saber si terminará!*. Veremos a continuación que, afortunadamente, es irrelevante conocer tanto detalle.

6.2 Modelos de Computación y Tiempos *[LP81, sec 7.3]*

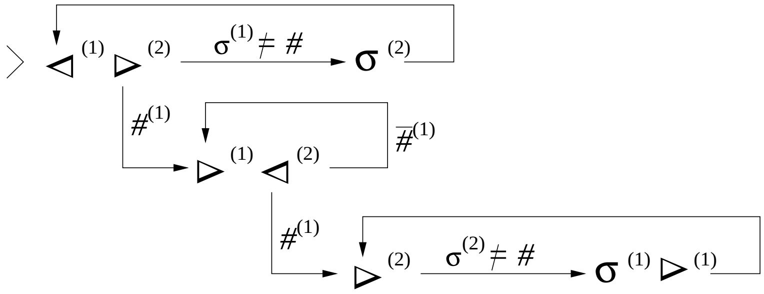
Hasta ahora nos hemos mantenido en el modelo de MTs determinísticas de una cinta. Veremos a continuación que los tiempos que obtenemos son bastante distintos si cambiamos levemente el modelo de computación. Esto nos hará cuestionarnos cuál es el modelo que realmente nos interesa, o si podemos obtener resultados suficientemente generales como para que estas diferencias no importen. De aquí en adelante necesitaremos usar la *notación O*.

Definición 6.4 *Se dice que $f(n)$ es $O(g(n))$, para f y g crecientes, si existen constantes $c, n_0 > 0$ tal que, para todo $n \geq n_0$, $f(n) \leq c \cdot g(n)$.*

Esto nos permite expresar cómodamente, por ejemplo, que el tiempo que toma una función es alguna constante multiplicada por n^2, diciendo que el tiempo es $O(n^2)$. Asimismo nos permite eliminar detalle innecesario, pues por ejemplo $n^2 + 2n - 3 = O(n^2)$.

Volvamos ahora al tema de la dependencia del modelo de computación.

Ejemplo 6.3 Recordemos la MT del Ej. 4.7, que calculaba $f(w) = ww^R$. Si se analiza cuidadosamente se verá que esa MT demora tiempo $T(n) = 2n^2 + 4n$. Consideremos ahora la siguiente MT de 2 cintas.



Analizándola, resulta que su $T(n) = 5n+3$. Por otro lado, se puede demostrar que es imposible obtener un $T(n)$ subcuadrático usando una sóla cinta.

Esto significa que, según el modelo de MT que usemos, el tiempo necesario para calcular una función puede variar mucho. Lo único que podemos garantizar acerca de la relación entre MTs de 1 y de k cintas es lo siguiente.

Lema 6.2 *Sea una MT M de k cintas que requiere tiempo $T(n) \geq n$ para calcular una función f. Entonces, existe una MT de una cinta que calcula f en tiempo $O(T(n)^2)$.*

Prueba: Nos basamos en una variante de la simulación de MTs de k cintas vista en la Sección 4.4. Esta variante es más compleja: en vez de hacer una pasada buscando cada cabezal, hace una única pasada recolectando los caracteres bajo los cabezales a medida que los va encontrando. Luego vuelve aplicando las transformaciones a la cinta. En tiempo $T(n)$ la MT de k cintas no puede alterar más de $n + T(n) \leq 2T(n)$ celdas de la cinta, de modo que la simulación de cada uno de los $T(n)$ pasos nos puede costar un recorrido sobre $2T(n)$ celdas. El costo total es de la forma $O(T(n)^2)$. En [LP81, sec 7.3] puede verse la fórmula exacta. □

Si, por ejemplo, usamos una MT de cinta doblemente infinita, la podemos simular en tiempo $6T(n) + 3n + 8$. Si usamos una MT de cinta bidimensional, la podemos simular en tiempo $O(T(n)^3)$. Una simulación particularmente importante es la siguiente.

Lema 6.3 *Sea una máquina RAM que requiere tiempo $T(n)$ para calcular una función f. Entonces, existe una MT de una cinta que calcula f en tiempo $O(T(n)^2)$.*

Prueba: Nos basamos en una variante de la simulación de máquinas RAM de la Sección 4.7. Esa simulación usaba 2 cintas. Hacerlo con una cinta es un poco más engorroso pero posible y no afecta el tiempo cuadrático que obtendremos. Nuevamente, la máquina RAM no puede escribir más de $T(n)$ celdas distintas en tiempo $T(n)$, y por ello la búsqueda de las celdas secuencialmente en la cinta 1 no puede demorar más de $O(T(n))$. Sumado sobre todas las $T(n)$ instrucciones a simular, tenemos $O(T(n)^2)$. *Es cierto que las celdas representadas en la MT pueden tener largo variable, pero si las representamos en binario en vez de unario el total de bits necesario será similar al de las máquinas RAM, a las que por otro lado es usual cobrarles tiempo proporcional a la cantidad de bits que manipulan.* $\Box$

Si queremos establecer resultados suficientemente generales como para que se apliquen a otros modelos de computación razonables (en particular las máquinas RAM), no deberá importarnos mucho la diferencia entre n^2 y n^4. Esto parece bastante decepcionante, pero no debería. Un resultado que obtengamos en un modelo tan permisivo será muy fuerte y muy general, y de hecho obtendremos resultados que se podrán trasladar directamente al modelo RAM.

Existe un modelo que hemos dejado de lado: las MTNDs. Hay dos buenas razones para ello. Una es que, a diferencia de los modelos anteriores, no sabemos cómo construir MTNDs reales. Otra es que es mucho más costoso simular una MTND con una MTD. Esta diferencia es central en la teoría de complejidad computacional.

Las MTNDs no calculan funciones, sólo aceptan lenguajes, por lo que requieren una definición adecuada.

Definición 6.5 *Una MTND $M = (K, \Sigma, \Delta, s)$* acepta un lenguaje L en tiempo $T(n)$ *si, para toda $w \in (\Sigma - \{\#\})^*$, M se detiene frente a w en a lo sumo $T(|w|)$ pasos sii $w \in L$.*

Observación 6.2 *Recordemos que seguimos dentro de los lenguajes decidibles, y hemos tenido que hablar de aceptar por un tecnicismo. Si una MTND M acepta L en tiempo $T(n)$, para cualquier $T(n)$, entonces L es decidible: Basta correr M durante $T(|w|)$ pasos. Si para entonces no se detuvo, $w \notin L$.*

Lema 6.4 *Sea M una MTND que acepta L en tiempo $T(n)$. Entonces existe una MTD que decide L en tiempo $c^{T(n)}$, para alguna constante c.*

Prueba: Consideremos una variante de la simulación vista en la Sección 4.5, que limpie la cinta y responda `S` si la MTND se detiene antes de la profundidad $T(n) + 1$ en el árbol, o limpie la cinta y responda `N` si llega a la profundidad $T(n) + 1$ sin detenerse. Esta simulación tiene un costo exponencial en $T(n)$, donde la base depende de la aridad del árbol. $\Box$

6.3 Las Clases $\mathcal{P}$ y $\mathcal{NP}$ *[LP81, sec 7.4 y 7.5]*

La sección anterior nos muestra que, dentro de los modelos de computación razonables (que excluyen las MTNDs), los tiempos de cómputo están relacionados polinomialmente. Para obtener resultados suficientemente generales, definiremos una clase de problemas "fáciles" y otro de "difíciles" que abstraiga del modelo de computación.

Definición 6.6 *La* clase $\mathcal{P}$ *es el conjunto de todos los lenguajes que pueden decidirse en tiempo polinomial con alguna MTD (es decir, $T(n)$ es algún polinomio en n). La* clase $\mathcal{NP}$ *es el conjunto de todos los lenguajes que pueden aceptarse en tiempo polinomial con alguna MTND.*

Un lenguaje en $\mathcal{P}$ se puede resolver en tiempo polinomial usando MTs de una cinta, k cintas, cintas bidimensionales, máquinas RAM, etc. Un lenguaje en $\mathcal{NP}$ puede resolverse en tiempo polinomial usando MTNDs. En cierto sentido, $\mathcal{P}$ representa la clase de problemas que se pueden resolver en tiempo razonable con las tecnologías conocidas.

Es evidente que $\mathcal{P} \subseteq \mathcal{NP}$. La vuelta, es decir la pregunta ¿$\mathcal{P} = \mathcal{NP}$? es el problema abierto más importante en computación teórica en la actualidad, y ha resistido décadas de esfuerzos. Resolver si $\mathcal{P} = \mathcal{NP}$ equivale a determinar que, dada una MTND, hay siempre una forma de simularla en tiempo polinomial, o que hay MTNDs para las cuales eso no es posible.

Observación 6.3 *Es interesante especular con las consecuencias de que $\mathcal{P}$ fuera igual a $\mathcal{NP}$. Hay muchos problemas difíciles que se resuelven fácilmente en una MTND mediante "adivinar" una solución y luego verificarla. Por ejemplo, se podrían romper los sistemas criptográficos mediante generar una clave no determinísticamente y luego corriendo el algoritmo (eficiente) que verifica si la clave es correcta. Hoy en día pocos creen que $\mathcal{P}$ pueda ser igual a $\mathcal{NP}$, pero esto no se ha podido demostrar. Pronto veremos por qué todos son tan escépticos.*

Un paso fundamental hacia la solución del problema es la definición de la clase de problemas NP-completos. Para explicar lo que son estos problemas debemos comenzar con el concepto de reducción polinomial.

Definición 6.7 *Un lenguaje $L' \subseteq \Sigma_0^*$* reduce polinomialmente *a otro lenguaje $L \subseteq \Sigma_1^*$, denotado $L' \leq L$, si existe una función $f : \Sigma_0^* \rightarrow \Sigma_1^*$ computable en tiempo polinomial en una MTD, tal que $w \in L' \Leftrightarrow f(w) \in L$.*

Esto indica que, en cierto sentido, L' no es más difícil que L (si todo lo polinomial nos da lo mismo), pues para resolver L' basta aplicar f a la entrada y resolver L.

Lema 6.5 *Si $L' \leq L$ y $L \in \mathcal{P}$, entonces $L' \in \mathcal{P}$.*

Prueba: Para determinar si $w \in L'$, aplico la MTD que computa $f(w)$ (en tiempo polinomial $T_f(|w|)$), y a eso le aplico la MTD que decide L en tiempo polinomial $T(n)$. El tiempo que tomará el proceso completo es $T_f(|w|)+T(|f(w)|)$. Como f se calcula en tiempo $T_f(|w|)$, en ese tiempo es imposible escribir una salida de largo mayor a $T_f(|w|)$, por lo tanto $|f(w)| \le |w| + T_f(|w|)$, lo cual es un polinomio en $|w|$, y también lo será al componerlo con otro polinomio, $T(\cdot)$. $\Box$

Como es de esperarse, esta relación $\le$ es transitiva.

Lema 6.6 *Si $L'' \le L'$ y $L' \le L$, entonces $L'' \le L$.*

Prueba: Sea f la reducción polinomial de L'' a L' y g la de L' a L. Sean $T_f(n)$ y $T_g(n)$ sus tiempos de cómputo. Entonces $h(w) = g(f(w))$ reduce polinomialmente de L'' a L. Por un lado, $w \in L'' \Leftrightarrow f(w) \in L' \Leftrightarrow g(f(w)) \in L$. Por otro, el tiempo de aplicar $g(f(w))$ es $T_f(|w|)+T_g(|f(w)|)$. Ya hemos visto en el Lema 6.5 que esto es necesariamente polinomial en $|w|$. $\Box$

Los problemas NP-completos son, en cierto sentido, los más difíciles dentro de la clase $\mathcal{NP}$.

Definición 6.8 *Un lenguaje L es* NP-completo *si (a) $L \in \mathcal{NP}$, (b) $\forall L' \in \mathcal{NP},\ L' \le L$.*

El siguiente lema muestra en qué sentido los problemas NP-completos son los más difíciles de $\mathcal{NP}$.

Lema 6.7 *Si L es NP-completo y $L \in \mathcal{P}$, entonces $\mathcal{P} = \mathcal{NP}$.*

Prueba: Sea un $L' \in \mathcal{NP}$. Como L es NP-completo, entonces $L' \le L$, y si $L \in \mathcal{P}$, por el Lema 6.5, $L' \in \mathcal{P}$. $\Box$

Observación 6.4 *Esto significa que si se pudiera resolver* cualquier *problema NP-completo en tiempo polinomial, entonces inmediatamente todos los problemas $\mathcal{NP}$ se resolverían en tiempo polinomial. Se conocen cientos de problemas NP-completos, y tras décadas de esfuerzo nadie ha logrado resolver uno de ellos en tiempo polinomial. De aquí la creencia generalizada de que $\mathcal{P} \ne \mathcal{NP}$.*

Nótese que todos los problemas NP-completos son equivalentes, en el sentido de que cualquiera de ellos reduce a cualquier otro. Pero, ¿cómo se puede establecer que un problema es NP-completo? La forma estándar es demostrar que algún problema NP-completo es "más fácil" que el nuestro (el problema de cómo se estableció el primer problema NP-completo se verá en la próxima sección).

Lema 6.8 *Si L' es NP-completo, $L \in \mathcal{NP}$, y $L' \le L$, entonces L es NP-completo.*

Prueba: Como L' es NP-completo, $L'' \le L'$ para todo $L'' \in \mathcal{NP}$. Pero $L' \le L$, entonces por transitividad (Lema 6.6) $L'' \le L$. $\Box$

Observación 6.5 *En la práctica, es bueno conocer un conjunto variado de problemas NP-completos. Eso ayudará a intuir que un problema dado es NP-completo, y también qué problema NP-completo conocido se puede reducir a él, para probar la NP-completitud. Si un problema es NP-completo, en la práctica no es esperable resolverlo eficientemente, por lo que se deberá recurrir a algoritmos aproximados, probabilísticos, o meras heurísticas para tratarlo.*

6.4 SAT es NP-completo *[AHU74, sec 10.4]*

Lo que hemos visto nos entrega herramientas para mostrar que un problema es NP-completo mediante reducir a él otro problema que ya sabemos que es NP-completo. Pero, ¿cómo obtenemos el primer problema NP-completo? No es fácil, pues debemos demostrar que *cualquier* problema NP reduce a él.

El lenguaje que elegiremos como nuestro primer problema NP-completo se llama SAT.

Definición 6.9 *El lenguaje* SAT *es el de las fórmulas proposicionales satisfactibles, es decir, aquellas que es posible hacer verdaderas con alguna asignación de valor de verdad a sus variables. Permitiremos los paréntesis, la disyunción $\vee$, la conjunción $\wedge$, y la negación $\sim$. Los nombres de las* variables proposicionales *serán cadenas sobre algún alfabeto de letras, aunque por simplicidad pensaremos en letras individuales (no hará diferencia). Llamaremos* literales *a variables o variables negadas,* conjunciones *a fórmulas de la forma $P_1 \wedge \ldots \wedge P_q$ y* disyunciones *a fórmulas de la forma $P_1 \vee \ldots \vee P_q$.*

Observación 6.6 *Si una fórmula P tiene una cantidad pequeña k de variables proposicionales distintas, se puede probar una a una las 2^k combinaciones de valores de verdad y ver si alguna combinación hace P verdadera. El problema es que k puede ser cercano al largo de P, con lo cual este método toma tiempo exponencial en el largo de la entrada.*

Ejemplo 6.4 Considere la fórmula

$$\begin{aligned} & (p \vee \sim q \ \vee \ r) \ \wedge \ (\sim p \ \vee \ q \vee \sim r) \ \wedge \ (\sim p \vee \sim q \ \vee \ r) \ \wedge \ (p \vee \sim q \vee \sim r) \\ \wedge \ & (\sim p \vee \sim q \vee \sim r) \ \wedge \ (p \ \vee \ q \ \vee \ r) \end{aligned}$$

¿Es satisfactible? Toma algo de trabajo, pero sí lo es, con las asignaciones $p = 0$, $q = 0$ y $r = 1$, o $p = 1$, $q = 0$ y $r = 0$ (estamos escribiendo 1 para verdadero y 0 para falso).

Para mostrar que SAT es NP-completo, debemos comenzar mostrando que SAT $\in \mathcal{NP}$. Esto es evidente: Una MTND puede adivinar los valores a asignar a las variables y luego evaluar la fórmula en tiempo polinomial.

La parte compleja es mostrar que todo $L \in \mathcal{NP}$ reduce a SAT. La idea esencial es que, si $L \in \mathcal{NP}$, entonces existe una MTND $M = (K, \Sigma, \Delta, s)$ que se detiene en tiempo $p(|w|)$ o menos sii $w \in L$, donde $p(n)$ es un polinomio. A partir de M, w y p, construiremos una fórmula proposicional P que será satisfactible sii M se detiene frente a w en a lo sumo $p(n)$ pasos. Esta construcción $f_{M,p}(w) = P$ será nuestra función f, por lo que debemos cuidar que P tenga largo polinomial en $|w|$ y que se pueda escribir en tiempo polinomial.

La fórmula P debe expresar todo el funcionamiento de una MTND, incluyendo afirmaciones que nosotros mismos hemos obviado por ser intuitivas. Comenzaremos con un par de simplificaciones y observaciones para lo que sigue.

- Como la MTND arranca en la configuración $(s, \#w\underline{\#})$, en $p(|w|)$ pasos sólo puede llegar a la celda número $p'(w) = |w| + 2 + p(|w|)$, el cual también es un polinomio en $|w|$.

- Modificaremos la MTND M para que, si llega al estado h en un paso anterior a $p(|w|)$, se mantenga en ese estado de ahí en adelante. De este modo $w \in L$ sii M está en el estado h en el paso $p(|w|)$. No es difícil de hacer esta modificación: basta agregar las reglas (h, a, h, a) a Δ, para todo $a \in \Sigma$ (notar que esto no está realmente permitido en el formalismo, pero lo podemos hacer).

Utilizaremos las siguientes variables proposicionales en P. De ahora en adelante llamaremos $n = |w|$, y renombraremos $\Sigma = \{1,\ 2, \ldots,\ |\Sigma|\}$ y $K = \{1,\ 2, \ldots,\ |K|\}$.

- $C(i, j, t)$, para cada $1 \leq i \leq p'(n)$, $1 \leq j \leq |\Sigma|$, $0 \leq t \leq p(n)$, se interpretará como que en la celda i, en el paso t, está el carácter j.
- $H(i, t)$, para cada $1 \leq i \leq p'(n)$, $0 \leq t \leq p(n)$, se interpretará como que el cabezal está en la celda i en el paso t.
- $S(k, t)$, para cada $1 \leq k \leq |K|$, $0 \leq t \leq p(n)$, se interpretará como que la MTND está en el estado k en el paso t.

La cantidad de variables proposicionales es $O(p'(n)^2)$ (pues K y Σ son constantes, siempre nos referimos al largo de w). Como tenemos que usar un alfabeto fijo para los nombres de variables, realmente los largos que reportamos a continuación deben multiplicarse por algo del tipo $\log p'(n) = O(\log n)$, lo cual no afecta su polinomialidad.

La fórmula P tiene siete partes:

$$P \quad = \quad A \wedge B \wedge C \wedge D \wedge E \wedge F \wedge G$$

cada una de las cuales fija un aspecto de la computación:

A: El cabezal está exactamente en un lugar en cada paso de una computación.

$$A \quad = \quad \bigwedge_{0 \leq t \leq p(n)} U(H(1, t),\ H(2, t), \ldots,\ H(p'(n), t)).$$

Hemos usado la notación U para indicar que uno y sólo uno de los argumentos debe ser verdadero, es decir,

$$U(x_1, x_2, \ldots, x_r) \quad = \quad \left(\bigvee_{1 \leq m \leq r} x_m \right) \quad \wedge \quad \left(\bigwedge_{1 \leq m < m' \leq r} \sim (x_m \wedge x_{m'}) \right).$$

Como el largo de U es $O(r^2)$, el largo de A es $O(p'(n)^3)$.

B: Cada celda contiene exactamente un símbolo en cada paso de una computación.

$$B \quad = \quad \bigwedge_{\substack{1 \leq i \leq p'(n) \\ 0 \leq t \leq p(n)}} U(C(i, 1, t),\ C(i, 2, t), \ldots,\ C(i, |\Sigma|, t)),$$

cuyo largo es $O(p'(n)^2)$.

C: La MTND está exactamente en un estado en cada paso de una computación.

$$C \quad = \quad \bigwedge_{0\le t\le p(n)} U(S(1,t),\ S(2,t),\ldots,\ S(|K|,t)),$$

cuyo largo es $O(p'(n))$.

D: La única celda que puede cambiar entre un paso y el siguiente es aquella donde está el cabezal.

$$D \quad = \quad \bigwedge_{\substack{1\le i\le p'(n)\\ 1\le j\le|\Sigma|\\ 0\le t<p(n)}} (H(i,t)\ \vee\ (C(i,j,t)\equiv C(i,j,t+1)),$$

donde hemos abreviado $x\equiv y$ para decir $(x\ \wedge\ y)\ \vee\ (\sim x\wedge\sim y)$. El largo de D es $O(p'(n)^2)$.

E: Recién aquí empezamos a considerar la M específica, pues lo anterior es general para toda MT. En E se especifica lo que ocurre con la posición del cabezal y el contenido de la cinta en esa posición en el siguiente instante de tiempo, según las opciones que dé el Δ de M.

$$E \quad = \quad \bigwedge_{\substack{1\le i\le p'(n)\\ 1\le j\le|\Sigma|\\ 1\le k\le|K|\\ 0\le t<p(n)}} \begin{array}{l}(C(i,j,t)\ \wedge\ H(i,t)\ \wedge\ S(k,t)\ \Longrightarrow\\ \qquad \bigvee_{(k,j,k_\ell,b_\ell)\in\Delta}(C(i,j_\ell,t+1)\ \wedge\ H(i_\ell,t+1)\ \wedge\ S(k_\ell,t+1)),\end{array}$$

donde $x\Rightarrow y$ es una abreviatura para $\sim x\ \vee\ y$. Los valores j_ℓ e i_ℓ son función de b_ℓ:

- Si $b_\ell\in\Sigma$, $i_\ell=i$ y $j_\ell=b_\ell$.
- Si $b_\ell=\lhd$, $i_\ell=i-1$ y $j_\ell=j$ *Esta regla se agrega sólo si* $i>1$.
- Si $b_\ell=\rhd$, $i_\ell=i+1$ y $j_\ell=j$.

El tamaño de E es $O(p'(n)^2)$. Nótese cómo aparece el no determinismo aquí. Dada una P que afirma algo para cierto t, para el tiempo $t+1$ vale la disyunción de varias posibilidades, según la MTND.

F: Establece las condiciones en que comienza la computación.

$$F \quad = \quad S(s,0)\wedge H(n{+}2,0)\wedge C(1,\#,0)\wedge \bigwedge_{2\le i\le n+1} C(i,w_{i-1},0)\wedge \bigwedge_{n+1<i\le p'(n)} C(i,\#,0),$$

donde recordemos que s, $\#$ y los w_i son todos números.

G: Establece, finalmente, que M está en el estado h en el instante $p(n)$.

$$G \quad = \quad S(h,p(n)).$$

Teorema 6.1 SAT *es NP-completo.*

Prueba: Vimos que SAT $\in \mathcal{NP}$. Luego, mostramos cómo reducir polinomialmente cualquier $L \in \mathcal{NP}$ a SAT: Describimos la construcción de una fórmula proposicional $P = f_{M,p}(w)$ que dice que existe una computación de M que empieza en la configuración $(s, \#w\underline{\#})$ y llega al estado h en el paso $p(n)$. Es posible hacer verdadera P (es decir, $P \in$ SAT) sii M acepta w (es decir, $w \in L$). Asignar los valores de las variables proposicionales $C(i,j,t)$, $H(i,t)$ y $S(k,t)$ equivale a decir qué computación elegiremos que sea válida y termine en el estado h. Si $|w| = n$, $|P| = O(p'(n)^3 \log n)$, es decir polinomial en $|w|$, y puede construirse fácilmente en tiempo polinomial. $\square$

Observación 6.7 *Esto significa que, si halláramos un método determinístico de tiempo polinomial para resolver* SAT*, inmediatamente podríamos resolver en tiempo polinomial cualquier problema en $\mathcal{NP}$. Basta construir una MTND M que lo resuelva, hallar el polinomio $p(n)$ que acota el tiempo en que operará, construir la P correspondiente a partir de M, p y la cadena w que nos interesa saber si está en L, y finalmente determinar en tiempo polinomial si P es satisfactible.*

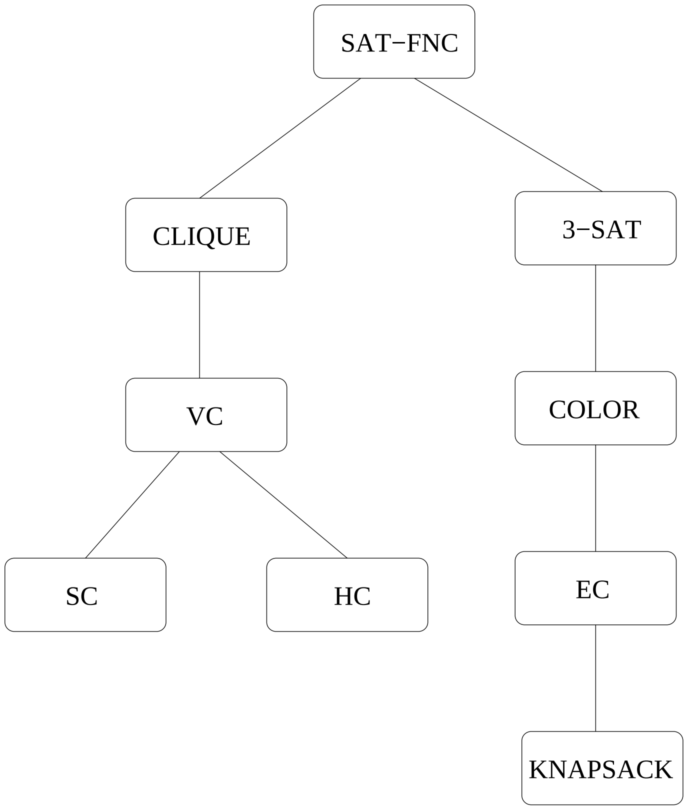
Definición 6.10 *Una fórmula proposicional está en* forma normal conjuntiva (FNC) *si es una conjunción de disyunciones de literales.* SAT-FNC *es el lenguaje de las fórmulas satisfactibles que están en forma normal conjuntiva.*

Lema 6.9 SAT-FNC *es NP-completo.*

Prueba: Determinar si una fórmula está en FNC es simple, de modo que el verdadero problema es saber si es satisfactible. Ya sabemos que SAT es NP-completo, pero podría ser que este caso particular fuera más fácil. No lo es. La fórmula P que se construye para SAT está prácticamente en FNC, sólo hay que redistribuir algunas fórmulas de tamaño constante en D y E. Por ejemplo, en D, debemos convertir $x \vee (y \equiv z)$, es decir, $x \vee (y \wedge z) \vee (\sim y \wedge \sim z)$, en $(x \vee y \vee \sim z) \wedge (x \vee \sim y \vee z)$. En el caso de E la fórmula también es de largo constante (independiente de n). $\square$

6.5 Otros Problemas NP-Completos *[AHU74, sec 10.5]*

Una vez que tenemos el primer problema NP-completo, es mucho más fácil obtener otros mediante reducciones. Veremos sólo unos pocos de los muchos problemas NP-completos conocidos. Los problemas NP-completos se suelen dibujar en un árbol, donde los nodos son problemas NP-completos y un nodo u hijo de v indica que una forma fácil o conocida de probar que u es NP-completo es reducir v a u. La raíz de este árbol será SAT-FNC.



Repasemos la metodología general para establecer que L es NP-completo:

1. Mostrar que $L \in \mathcal{NP}$. Esto suele ser muy fácil (¡cuando es verdad!).
2. Elegir un L' NP-completo para mostrar que $L' \leq L$. Esto puede requerir intuición y experiencia, pero a veces es muy evidente también.
   (a) Diseñar la transformación polinomial f. Esto suele hacerse junto con la elección de L', y es realmente la parte creativa del ejercicio.
   (b) Mostrar que f se puede calcular en tiempo determinístico polinomial. Esto suele ser muy fácil (¡cuando es verdad!).
   (c) Mostrar que $w \in L' \Leftrightarrow f(w) \in L$. Esto suele ser difícil, y normalmente va junto con el diseño de la f. Notar la implicación doble a demostrar.

Comenzaremos con una restricción aún mayor a SAT-FNC.

Definición 6.11 3-SAT *es el conjunto de fórmulas proposicionales satisfactibles en FNC, donde ninguna disyunción tiene más de 3 literales.*

Teorema 6.2 3-SAT *es NP-completo.*

Prueba: Primero, 3-SAT $\in \mathcal{NP}$ ya que 3-SAT es un caso particular de SAT. Para mostrar que es NP-completo, veremos que SAT-FNC $\leq$ 3-SAT. La reducción es como sigue. Sea $F = F_1 \ \wedge \ldots \wedge \ F_q$

la fórmula original en FNC. Transformaremos cada F_i con más de tres literales en una conjunción donde cada elemento tenga la disyunción de tres literales. Sea $F_i = x_1 \lor x_2 \lor \ldots \lor x_k$, con x_j literal (variable o variable negada). Introduciremos variables nuevas y_1 a y_{k-3}, y reemplazaremos F_i por

$$\begin{aligned} F_i' &= (x_1 \lor x_2 \lor y_1) \land (\sim y_1 \lor x_3 \lor y_2) \land (\sim y_2 \lor x_4 \lor y_3) \land \ldots \\ &\quad \land (\sim y_{k-4} \lor x_{k-2} \lor y_{k-3}) \land (\sim y_{k-3} \lor x_{k-1} \lor x_k). \end{aligned}$$

Está claro que F_i' se puede construir en tiempo polinomial a partir de F_i. Veamos ahora que la transformación preserva satisfactibilidad:

- Si F es satisfactible, existe una asignación de valores a las variables que hace verdadero cada F_i. Dentro de cada F_i, al menos uno de los literales x_j se hace verdadero con esta asignación. Conservemos esa asignación para F_i', y veamos que se puede asignar valores a los y_l que hagan verdadera a F_i'. Digamos que $3 \leq j \leq k-2$. Podemos asignar $y_{j-2} = 1$ y $y_{j-1} = 0$, pues $(\sim y_{j-2} \lor x_j \lor y_{j-1})$ se mantiene verdadero. El valor asignado a y_{j-2} hace verdadera la disyunción anterior, $(\sim y_{j-3} \lor x_{j-1} \lor y_{j-2})$, lo cual nos permite asignar $y_{j-3} = 1$ y seguir la cadena hacia atrás. Similarmente, el valor asignado a y_{j-1} hace verdadera la disyunción siguiente, $(\sim y_{j-1} \lor x_{j+1} \lor y_j)$, lo que nos permite asignar $y_j = 0$ y continuar la cadena hacia adelante. De modo que las F_i' son satisfactibles también.

- Si $F' = F_1' \land \ldots \land F_q'$ es satisfactible, hay una asignación de variables de F' que hace verdadera cada F_i'. Veremos que no es posible lograr eso si ninguna de las x_j se ha hecho verdadera. Si todas las x_j se han hecho falsas, entonces y_1 debe ser verdadera. Pero entonces, para hacer verdadera a $(\sim y_1 \lor x_3 \lor y_2)$ necesitamos que y_2 sea verdadera, y así siguiendo, necesitaremos finalmente que y_{k-3} sea verdadera, con lo que la última disyunción, $(\sim y_{k-3} \lor x_{k-1} \lor x_k)$ es falsa.

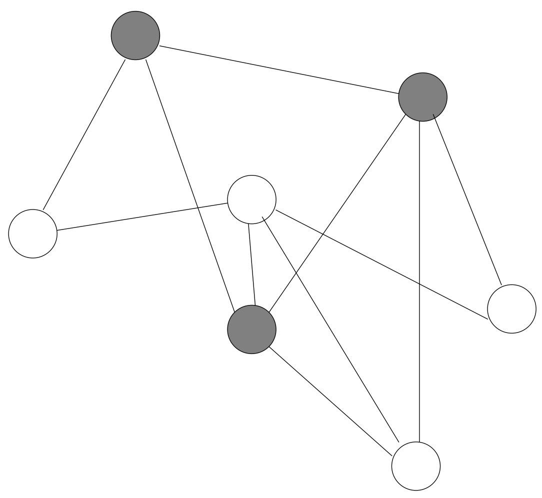
$\square$

Veamos ahora un problema NP-completo que no tiene nada que ver con fórmulas proposicionales, sino con grafos.

Definición 6.12 *Un* k-clique *en un grafo no dirigido* $G = (V, E)$ *es un subconjunto de* V *de tamaño* k *donde todos los vértices están conectados con todos. El lenguaje* CLIQUE *es el conjunto de pares* (G, k) *tal que* G *tiene un* k*-clique. Corresponde al problema de, dado un grafo, determinar si contiene un* k*-clique.*

Encontrar cliques en grafos es útil, por ejemplo, para identificar clusters o comunidades en redes sociales, entre muchas otras aplicaciones. Lamentablemente, no es fácil hallar cliques.

Ejemplo 6.5 El siguiente grafo tiene un 3-clique, que hemos marcado. ¿Puede encontrar otro? ¿Y un 4-clique?



Teorema 6.3 CLIQUE *es NP-completo.*

Prueba: Está claro que CLIQUE $\in \mathcal{NP}$: Una MTND puede adivinar los k vértices y luego verificar en tiempo polinomial que forman un k-clique. Para ver que es NP-completo, mostraremos que SAT-FNC $\leq$ CLIQUE. Sea $F = F_1 \ \wedge \ldots \wedge \ F_q$ una fórmula en FNC y sea $F_i = x_{i,1} \ \vee \ x_{i,2} \ \vee \ldots \vee \ x_{i,m_i}$. Construiremos un grafo $G = (V, E)$ que tendrá un q-clique sii F es satisfactible.

G tendrá un vértice por cada literal de F, formalmente $V = \{(i, j), \ 1 \leq i \leq q, \ 1 \leq j \leq m_i\}$. Y tendrá aristas entre literales de distintas componentes F_i y $F_{i'}$ que no sean uno la negación del otro, formalmente $E = \{((i, j), (i', j')), \ i \neq i', \ x_{i,j} \ \not\equiv \ \sim x_{i',j'}\}$.

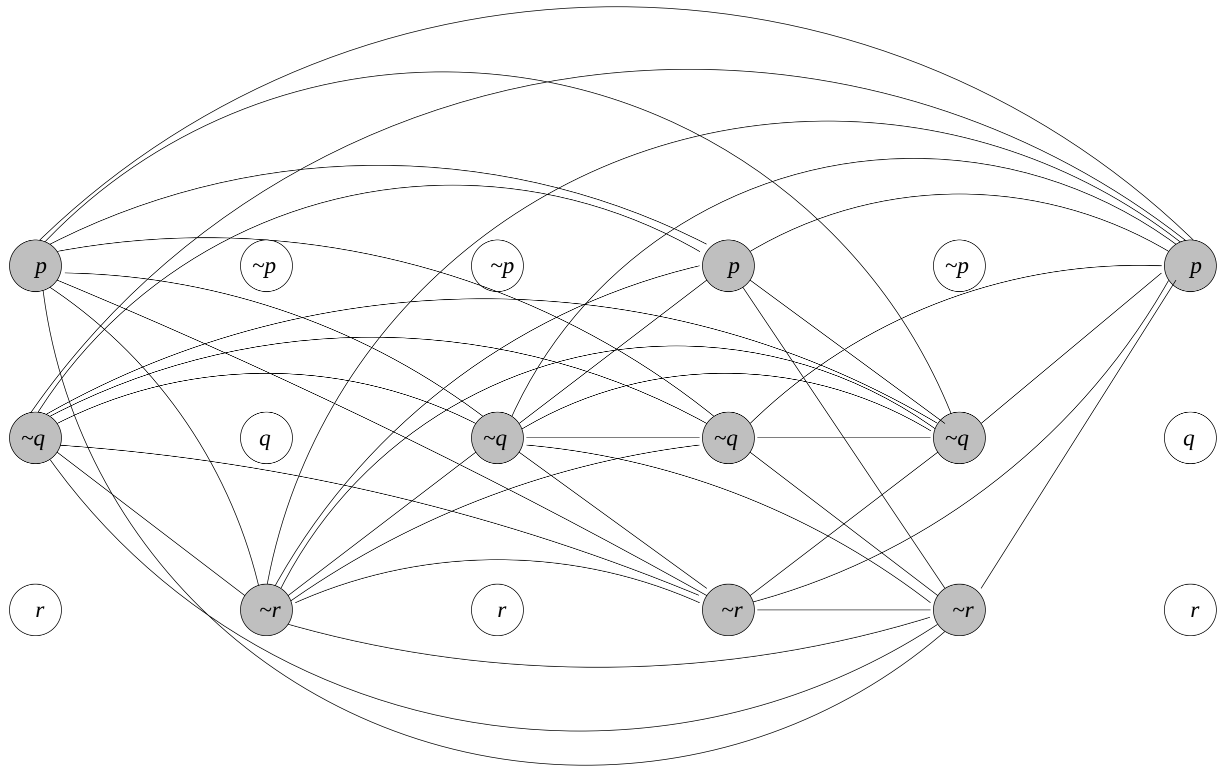
Está claro que G se puede construir en tiempo polinomial a partir de F. Veamos ahora que la transformación es correcta.

- Si F es satisfactible, podemos asignar valores a las variables de modo que tengamos al menos un literal verdadero $x_{i,v(i)}$ en cada F_i. Esos q literales verdaderos no pueden ser ninguno la negación del otro, pues se han hecho verdaderos todos a la vez. Como todos esos literales están en distintas componentes y no son ninguno la negación del otro, los nodos correspondientes están todos conectados en G, formando un q-clique.

- Si G tiene un q-clique, los literales asociados a los q nodos participantes deben estar todos en distintas componentes y no ser ninguno la negación del otro. Eso significa que se pueden hacer verdaderos todos a la vez, y tendremos un literal verdadero en cada F_i, con lo que F puede hacerse verdadera.

$\square$

Ejemplo 6.6 Tomemos la fórmula del Ej. 6.4 y construyamos el grafo G asociado a ella (con una columna por disyunción). Como aquella fórmula de 6 disyunciones es satisfactible, este G tiene al menos un 6-clique. Para evitar una maraña ilegible, hemos considerado la asignación de variables $p = 1$, $q = 0$, $r = 0$, marcando los nodos que corresponden a literales verdaderos (se hace verdadero al menos un nodo en cada columna), y hemos dibujado sólamente las aristas entre esos nodos que hemos seleccionado (que están todas conectadas con todas, de modo que se forma necesariamente al menos un 6-clique).



Observación 6.8 *Notar que, para cualquier k* fijo, *se puede determinar si $G = (V, E)$ tiene un k-clique en tiempo $O(n^k)$, lo cual es un polinomio en $|G|$. Esta solución, sin embargo, es exponencial en el largo de la entrada cuando k es un parámetro que puede ser tan grande como n.*

El siguiente problema tiene que ver con optimización de recursos. Por ejemplo, ¿cómo distribuir faroles en las esquinas de un barrio de modo que todas las cuadras estén iluminadas y minimicemos el número de faroles? Supongamos que cada farol alumbra una cuadra, hasta la próxima esquina, en todas las direcciones. El problema es fácil si el barrio tiene un trazado regular, pero si es arbitrario, es sorprendentemente difícil.

Definición 6.13 *Un* recubrimiento de vértices (vertex cover) *de tamaño k de un grafo no dirigido $G = (V, E)$ es un subconjunto de k nodos de V tal que para toda arista $(u, v) \in E$, al menos uno entre u y v están en el subconjunto elegido. El lenguaje* VC *se define como los pares (G, k) tal que G tiene un recubrimiento de vértices de tamaño k.*

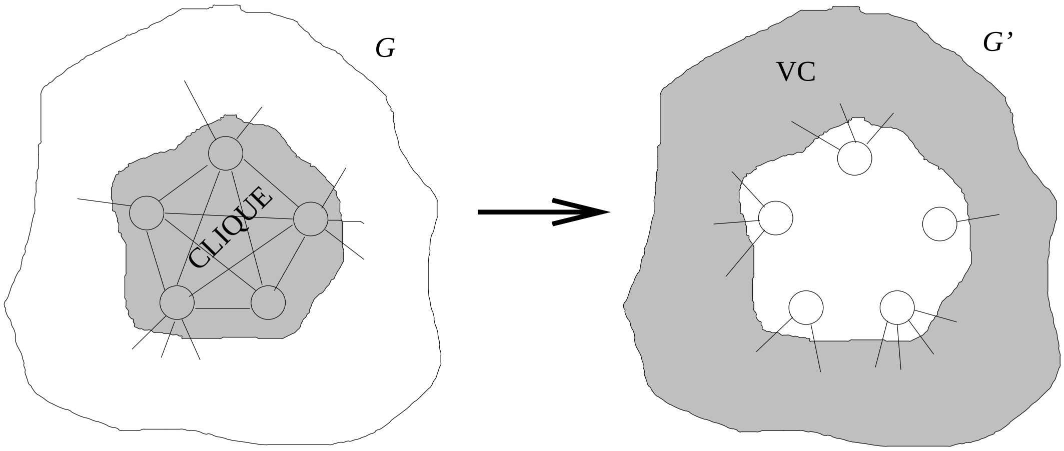
Teorema 6.4 VC *es NP-completo.*

Prueba: Primero, VC $\in \mathcal{NP}$, pues una MTND puede adivinar los k vértices y luego verificar en tiempo polinomial que toda arista incide en al menos un vértice elegido. Para ver que es NP-completo, probaremos que CLIQUE $\leq$ VC.

Esto es muy sencillo. Sea E' el complemento de las aristas de E. Entonces V' es un clique en $G = (V, E)$ sii $V - V'$ es un recubrimiento de vértices en $G' = (V, E')$. Una vez que nos convenzamos de esto, es inmediato cómo reducir: Un $G = (V, E)$ dado tendrá un k-clique sii $G' = (V, E')$ tiene un recubrimiento de vértices de tamaño $|V| - k$.

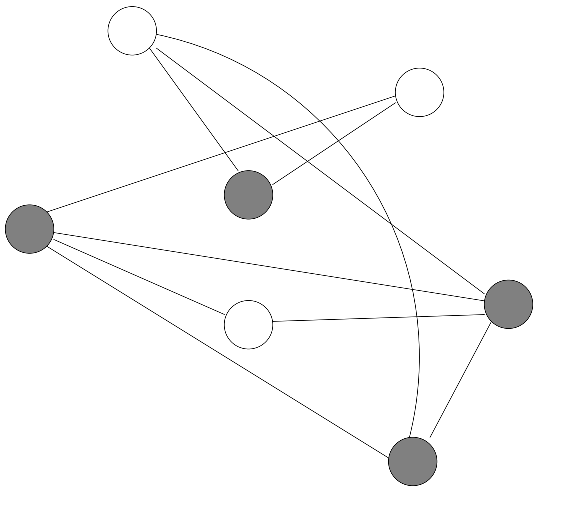
- Sea V' un clique en G. Sus nodos están todos conectados con todos. Si complementamos las aristas de G para formar G', ahora esos nodos no están conectados ninguno con ninguno. Eso significa que los demás vértices, $V - V'$, cubren todas las aristas de G', pues toda arista tiene al menos uno de sus extremos en $V - V'$ (es decir fuera de V').

- Sea V' un recubrimiento de vértices en G. Si consideramos $V - V'$, ninguna arista puede tener ambos extremos en $V - V'$, es decir, conectar dos nodos de $V - V'$. Al complementar las aristas para formar G', ahora todos los nodos de $V - V'$ están conectados entre sí, formando un clique.



$\square$

Ejemplo 6.7 Tomemos el grafo del Ej. 6.5 y complementemos las aristas. Como aquél grafo tenía un 3-clique y el grafo tiene 7 nodos, este grafo complementado tiene un recubrimiento de vértices de tamaño 4 (el complemento de los nodos de aquél clique). ¿Se puede cubrir todas las aristas con 3 vértices?



Veremos ahora un problema que no tiene que ver con grafos. Nuevamente tiene aplicaciones en optimización. Supongamos que queremos tener todas las obras de Mozart en

CDs, pero las obras vienen repetidas en los distintos CDs que están a la venta. Claramente no es necesario comprarlos todos para tener todas las obras. ¿Cuál es la mínima cantidad de CDs que necesito comprar? Un problema ligeramente más complejo es el de comprar un set de productos a mínimo costo dado un conjunto de ofertas de paquetes de productos.

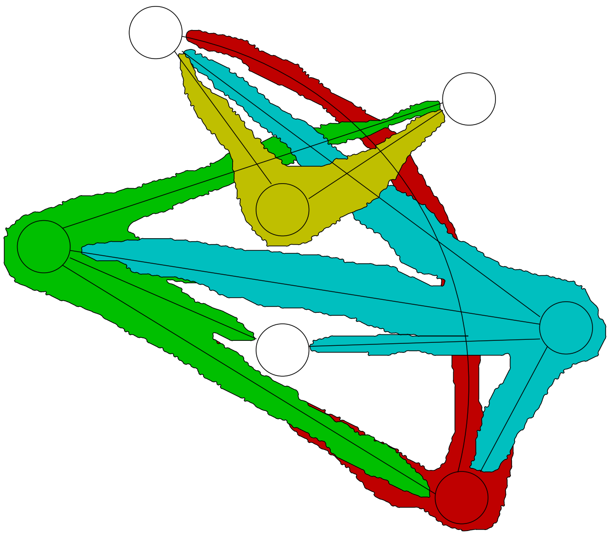
Definición 6.14 *Dados conjuntos $S_1,\ S_2,\ldots,\ S_n$, un* recubrimiento de conjuntos (set cover) *de tamaño k es un grupo de k conjuntos $S_{i_1},\ S_{i_2},\ldots,\ S_{i_k}$, tal que $\bigcup_{1\leq j\leq k} S_{i_j} = \bigcup_{1\leq i\leq n} S_i$. El lenguaje* SC *es el de los pares $(\mathcal{S} = \{S_1,\ S_2,\ldots,\ S_n\}, k)$ tal que $\mathcal{S}$ tiene un recubrimiento de conjuntos de tamaño k.*

Teorema 6.5 SC *es NP-completo.*

Prueba: Es fácil ver que SC $\in \mathcal{NP}$. Una MTND puede adivinar los k conjuntos a unir, unirlos y verificar que se obtiene la unión de todos los conjuntos. Para ver que es NP-completo, reduciremos VC $\leq$ SC.

La reducción es muy simple. Sea un grafo $G = (V, E)$. Asociaremos a cada vértice $v \in V$ un conjunto S_v conteniendo las aristas que tocan v, formalmente $S_v = \{(u, v) \in E\}$ (recordar que G no es dirigido, por lo que $(u, v) = (v, u)$. Evidentemente esto puede hacerse en tiempo polinomial. Además no es difícil ver que si $\{v_1,\ v_2,\ldots,\ v_k\}$ es un recubrimiento de vértices de G, $S_{v_1},\ S_{v_2},\ldots,\ S_{v_k}$ es un recubrimiento de conjuntos de $\mathcal{S}$, y viceversa. Lo primero dice que toda arista tiene al menos un extremo en algún v_i y lo segundo que toda arista está contenida en algún S_{v_i}, y ambas cosas son lo mismo porque precisamente S_{v_i} contiene las aristas que inciden en v_i. $\square$

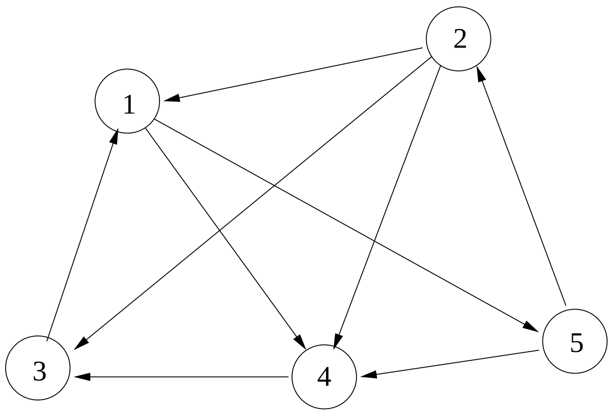
Ejemplo 6.8 Tomemos el grafo del Ej. 6.7 y dibujemos los conjuntos de aristas del problema correspondiente de SC directamente sobre el grafo. Hemos dibujado sólamente los que corresponden a la solución del VC en aquél ejemplo. Toda arista está en algún conjunto.



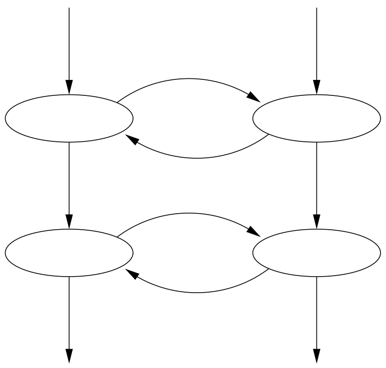
Volvamos a problemas en grafos. Un problema de optimización muy común en aplicaciones de transporte es el de recorrer un conjunto de sitios a costo mínimo, cuando existe un costo arbitrario para ir de cada sitio a otro. El siguiente problema es una simplificación de este escenario, la cual ya es NP-completa.

Definición 6.15 *Un grafo dirigido $G = (V, E)$ tiene un* circuito hamiltoniano (Hamiltonian circuit) *si es posible partir de uno de sus nodos y, moviéndose por aristas, ir tocando cada nodo de V exactamente una vez, volviendo al nodo original. El lenguaje* HC *es el de los grafos dirigidos G que tienen un circuito hamiltoniano.*

Ejemplo 6.9 ¿Tiene este grafo un circuito hamiltoniano? Cuesta un poco encontrarlo, pero lo tiene: 5,2,4,3,1,5.



Antes de demostrar que HC es NP-completo, estudiemos el siguiente subgrafo:



Nótese que, si se entra por la derecha y se sale por la izquierda, o viceversa, siempre quedará un nodo excluido del potencial circuito hamiltoniano. De modo que en cualquier circuito hamiltoniano que involucre este subgrafo, si el circuito entra por la izquierda debe salir por la izquierda, y si entra por la derecha debe salir por la derecha. Si entra por la izquierda, puede o no tocar los nodos de la derecha (y viceversa). Con esto estamos listos para mostrar que HC es NP-completo.

Teorema 6.6 HC *es NP-completo.*

Prueba: Es fácil ver que HC $\in \mathcal{NP}$: Una MTND puede adivinar la permutación que forma el circuito y luego verificar que existe una arista entre cada nodo y el siguiente de la permutación. Para ver que es NP-completo, mostraremos que VC $\leq$ HC. Dado un par (G, k), construiremos un grafo dirigido G_D tal que G tendrá un recubrimiento de vértices de tamaño k sii G_D tiene un circuito hamiltoniano.

La construcción es como sigue. Sea $G = (V, E)$. Por cada nodo $v \in V$ tendremos una *lista* de nodos en G_D. Por cada arista $(u, v) \in E$ pondremos un par de nodos en la lista de u y otro par en la lista de v, formando el subgrafo mostrado recién entre esos cuatro nodos. Finalmente, agregaremos k nodos $a_1,\ a_2, \ldots,\ a_k$, de los que saldrán aristas al comienzo de cada una de las $|V|$ listas y a las que llegarán aristas desde el final de cada una de las $|V|$ listas.

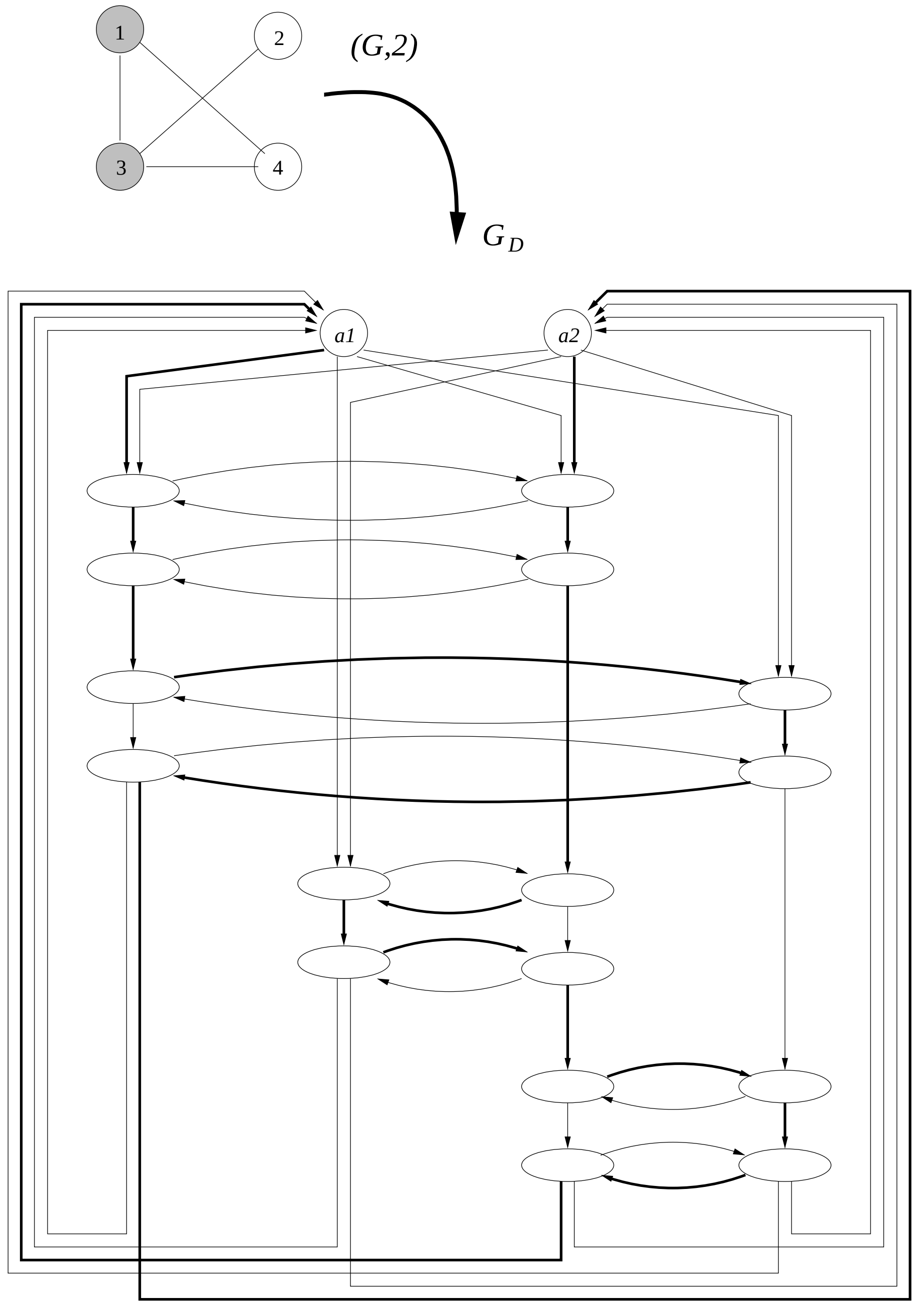
Está claro que este grafo G_D se puede construir en tiempo polinomial en $|G|$. Veamos ahora que $(G, k) \in$ VC sii $G_D \in$ HC.

- Si hay k nodos $\{v_1,\ v_2, \ldots,\ v_k\} \subseteq V$ que cubren todas las aristas, el circuito en G_D pasará por las listas que corresponden a los nodos elegidos en V. Comenzaremos por a_1, luego pasaremos por la lista de v_1, al salir iremos a a_2, luego a la lista de v_2, y así hasta recorrer la lista de v_k y volver a a_1. Este circuito recorre todos los nodos de G_D que están en las listas elegidas. ¿Qué pasa con los nodos en las listas no elegidas? Estos aparecen de a pares, y corresponden a aristas que conectan los nodos no elegidos con nodos, necesariamente, elegidos (pues los nodos elegidos cubren todas las aristas). Entonces, cada uno de estos pares se puede recorrer en el momento en que se pase por el par de nodos correspondiente del vértice elegido.

- Si G_D tiene un circuito hamiltoniano, cada una de las a_i debe aparecer exactamente una vez. Debe recorrerse exactamente una lista luego de cada a_i. Esas k listas que se recorren son las de vértices que necesariamente cubren todas las aristas de G, pues los pares de nodos de las listas no elegidas han sido incluidas en el circuito y eso implica que los otros dos nodos que les corresponden están en listas elegidas.

$\square$

Ejemplo 6.10 Tomemos un pequeño ejemplo de recubrimiento de vértices de tamaño 2, donde $\{1, 3\}$ es una solución. Hemos dibujado el problema HC asociado, y el circuito que corresponde a seleccionar esos dos vértices. Obsérvese cómo se pasa por los vértices de las listas no elegidas cuando es necesario para incluirlos en el circuito.

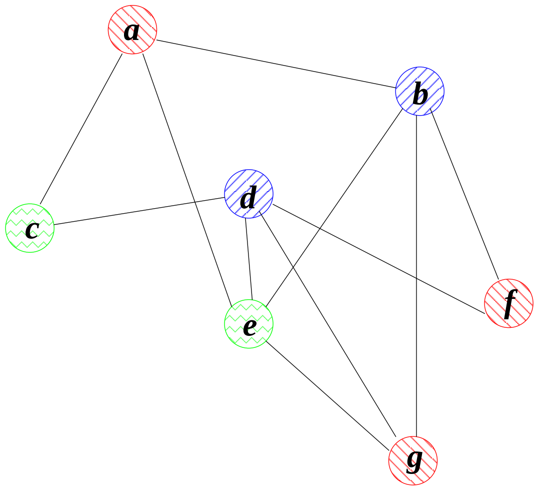


Otro problema importante en grafos es el de colorearlos, es decir asignarle una clase (de un conjunto finito) a cada nodo, de modo que nodos adyacentes sean de distinta clase. Una aplicación elemental es colorear un mapa, pero se puede usar para modelar cosas como distribuir tareas en servidores de modo que algunas tareas (por consumir el mismo tipo de recursos) no deberían estar juntas.

Definición 6.16 *Un k-*coloreo *de un grafo no dirigido $G = (V, E)$ es una función $c : V \longrightarrow [1, k]$, tal que si $(u, v) \in E$, entonces $c(u) \neq c(v)$. El lenguaje* COLOR *es el conjunto de los pares (G, k) tal que G tiene un k-coloreo.*

El problema es trivial para $k = 1$ y fácil de resolver polinomialmente para $k = 2$ *¡inténtelo!*, pero NP-completo a partir de $k = 3$.

Ejemplo 6.11 Considere el grafo del Ej. 6.5. Al tener un 3-clique, está claro que se necesitan al menos 3 colores para colorearlo. ¿Basta con tres? Toma algo de trabajo convencerse, pero se puede. Una solución es la que sigue.



Teorema 6.7 COLOR *es NP-completo.*

Prueba: Una MTND puede fácilmente adivinar el color a asignar a cada vértice y luego verificar en tiempo polinomial que no hay pares conectados del mismo color. Para ver que COLOR es NP-completo, mostraremos que 3-SAT $\leq$ COLOR.

Supongamos que tenemos una fórmula $F = F_1 \wedge F_2 \wedge \ldots \wedge F_q$, formada con variables proposicionales $v_1, v_2, \ldots, v_n$. Supondremos $n \geq 4$, lo cual no es problema porque con cualquier n constante 3-SAT se puede resolver probando todas las combinaciones en tiempo polinomial. *O sea, si a un conjunto difícil le restamos un subconjunto fácil, lo que queda aún es difícil.* Construiremos un grafo $G = (V, E)$ que será coloreable con $n + 1$ colores sii F es satisfactible.

G tendrá los siguientes vértices: (i) v_i, $\sim v_i$ y x_i, $1 \leq i \leq n$, donde las v_i son las variables proposicionales y x_i son símbolos nuevos, (ii) F_j, $1 \leq j \leq q$. Las aristas conectarán los siguientes pares: (a) todos los (x_i, x_j), $i \neq j$; (b) todos los (x_i, v_j) y $(x_i, \sim v_j)$, $i \neq j$; (c) todos los $(v_i, \sim v_i)$; (d) todos los (v_i, F_j) donde v_i *no* es un literal en F_j; (e) todos los $(\sim v_i, F_j)$ donde $\sim v_i$ *no* es un literal en F_j.

La construcción no es muy intuitiva, pero obviamente puede hacerse en tiempo polinomial. Debemos ver ahora que funciona. Comencemos por algunas observaciones. El n-clique formado por los x_i's obliga a usar al menos n colores, uno para cada x_i. A su vez, los v_i y $\sim v_i$ no pueden tomar el color de un x_j con $j \neq i$. Sí pueden tomar el color de su x_i, pero no pueden hacerlo tanto v_i como $\sim v_i$ porque ellos también están conectados entre sí. De modo que uno de los dos debe recibir un color más, que llamaremos *gris* (la intuición es que un literal gris corresponde a hacerlo falso). Hasta ahora podemos colorear G con $n+1$ colores, pero faltan aún los nodos F_j. Si queremos un $n + 1$ coloreo debemos colorear estas F_j sin usar nuevos colores.

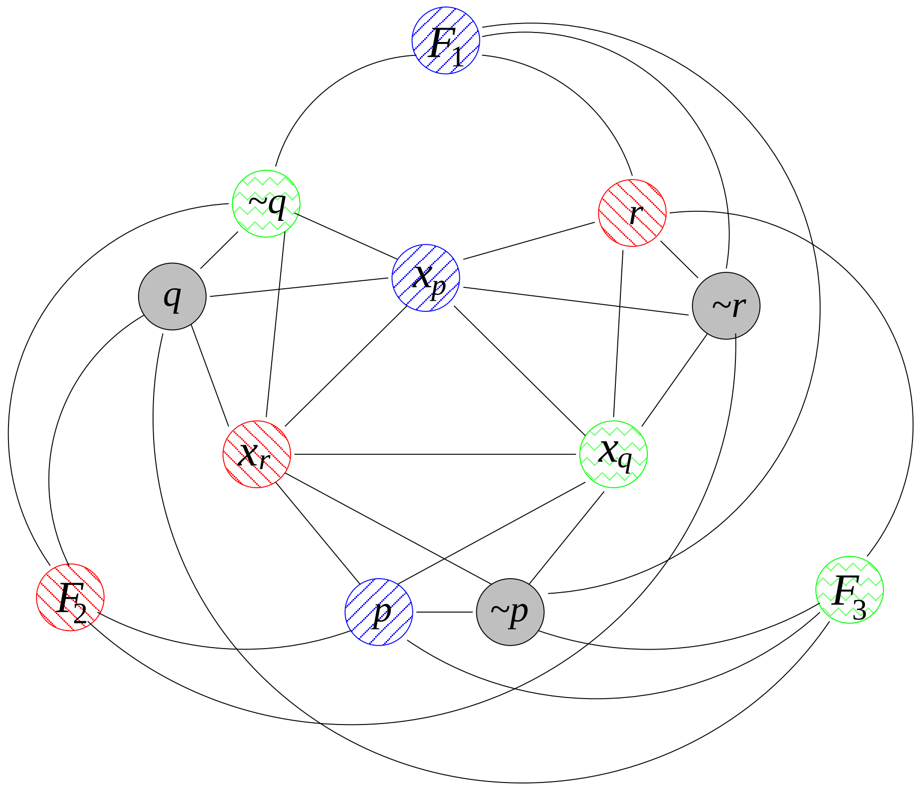
Como hay al menos 4 variables distintas y las F_j mencionan 3 literales, existe por lo menos una variable v_i que no se menciona en F_j. Esto hace que F_j esté conectado con v_i y con $\sim v_i$, y por ende no pueda ser coloreada de gris. Los únicos colores que puede tomar F_j corresponden a los literales (variables afirmadas o negadas) que aparecen en F_j, pues está conectada con todos los demás. Si estos literales son todos grises, como F_j no puede ser gris, se necesitará un color más. Si, en cambio, alguno de ellos es de otro color, F_j podrá tomar ese color y no requerir uno nuevo.

- Supongamos que F es satisfactible. Entonces existe una asignación de valores de verdad a las v_i tal que cada F_j contiene un literal que se hizo verdadero. Si coloreamos de gris a las v_i y $\sim v_i$ que se hacen falsas, y coloreamos igual que x_i a las v_i y $\sim v_i$ que se hacen verdaderas, entonces cada F_j podrá tomar el color no-gris de alguna de las v_i o $\sim v_i$ que aparecen en ella. Entonces G será coloreable con $n + 1$ colores.

- Supongamos que G se puede colorear con $n+1$ colores. Eso significa que podemos elegir cuál entre v_i y $\sim v_i$ será del color de su x_i (y la otra será gris), de modo que cada F_j contendrá al menos un literal coloreado no-gris. Entonces se puede hacer falsos a los literales coloreados de gris, y cada F_j tendrá un literal que se haga verdadero.

□

Ejemplo 6.12 No nos sirve el Ej. 6.4 porque sólo tiene 3 variables. Para que nos quede algo legible, usaremos $F = (p \ \vee \ q) \ \wedge \ (\sim p \ \vee \ r) \ \wedge \ (\sim q \vee \sim r)$. Este también tiene 3 variables, pero sólo 2 literales por disyunción, de modo que sigue valiendo que hay alguna variable que no aparece en cada F_j. El grafo que se genera para esta F es el siguiente. El grafo es 4-coloreable porque F es satisfactible, por ejemplo con $p = 1$, $q = 0$, $r = 1$. Hemos coloreado el grafo de acuerdo a esa asignación de variables.



Un problema relevante de conjuntos, parecido al recubrimiento de vértices, es el de cubrir el conjunto con subconjuntos sin traslapes.

Definición 6.17 *Dados conjuntos $S_1,\ S_2, \ldots,\ S_n$, un* recubrimiento exacto (exact cover) *es un grupo de conjuntos $S_{i_1},\ S_{i_2}, \ldots,\ S_{i_k}$, tal que $\bigcup_{1 \le j \le k} S_{i_j} = \bigcup_{1 \le i \le n} S_i$ y $S_{i_j} \cap S_{i_{j'}} = \emptyset$ para todo $j \neq j'$. El lenguaje* EC *es el de los conjuntos $\mathcal{S} = \{S_1,\ S_2, \ldots,\ S_n\}$ tal que $\mathcal{S}$ tiene un recubrimiento exacto.*

A pesar de lo que podría esperarse, la reducción más fácil hacia EC no es desde SC sino desde COLOR.

Teorema 6.8 EC *es NP-completo.*

Prueba: Es fácil ver que EC $\in \mathcal{NP}$, pues una MTND puede adivinar los conjuntos y luego verificar que forman un recubrimiento exacto. Para mostrar que es NP-completo, mostraremos que COLOR $\le$ EC.

Dada la entrada $(G = (V,E),k)$ a COLOR, generaremos una entrada para EC de la siguiente forma. Los conjuntos de $\mathcal{S}$ serán como sigue: (i) para cada $v \in V$ y cada $1 \le i \le k$, el conjunto $S_{v,i} = \{v\} \cup \{(u,v,i),\ u \in V,\ (u,v) \in E\}$, (ii) para cada $(u,v) \in E$ y $1 \le i \le k$, el conjunto $S_{u,v,i} = \{(u,v,i)\}$. La unión de todos los conjuntos es $V \cup (E \times [1,k])$. Claramente estos conjuntos se pueden construir en tiempo polinomial. Veamos ahora que la transformación es correcta.

- Si es posible asignar colores $c(v) \in [1,k]$ a cada $v \in V$ de modo que no haya nodos del mismo color conectados, entonces es posible cubrir $\mathcal{S}$ exactamente: elegiremos los conjuntos $S_{v,c(v)}$ para cada v, y después agregaremos los $S_{u,v,i}$ que falten para completar el conjunto. Está claro que tenemos todos los elementos de V en la unión de estos conjuntos, y también los de $E \times [1,k]$ pues agregamos todos los elementos (u,v,i) que sean necesarios. Por otro lado, no hay intersección en los conjuntos elegidos: claramente no la hay en los $S_{u,v,i}$, y no la hay en ningún par $S_{v,c(v)}$ y $S_{u,c(u)}$, pues el único elemento que podrían compartir es $(u,v,c(u)) = (u,v,c(v))$, para lo cual $c(v)$ y $c(u)$ deberían ser iguales. Esto es imposible porque u y v son adyacentes.

- Si existe un recubrimiento exacto, debemos haber elegido exactamente un $S_{v,i}$ por cada $v \in V$ para poder cubrir V. Si no hay traslapes entre estos conjuntos es porque para todo par de nodos adyacentes u y v, se han elegido distintos valores de i (pues sino habría un elemento repetido (u,v,i) entre los conjuntos). Podemos colorear v del color $c(v) = i$ si elegimos $S_{v,i}$, y tendremos un k-coloreo (sin nodos adyacentes con el mismo color i).

$\square$

Ejemplo 6.13 Reduzcamos el problema de 3-colorear el grafo del Ej. 6.11 a EC. Los conjuntos son los siguientes. Hemos ordenado los pares en forma consistente, pues las aristas no tienen dirección.

$$\begin{aligned}
S_{a,1} &= \{a,\ (a,b,1),\ (a,c,1),\ (a,e,1)\} \\
S_{a,2} &= \{a,\ (a,b,2),\ (a,c,2),\ (a,e,2)\} \\
S_{a,3} &= \{a,\ (a,b,3),\ (a,c,3),\ (a,e,3)\} \\
S_{b,1} &= \{b,\ (a,b,1),\ (b,e,1),\ (b,f,1)\} \\
S_{b,2} &= \{b,\ (a,b,2),\ (b,e,2),\ (b,f,2)\} \\
S_{b,3} &= \{b,\ (a,b,3),\ (b,e,3),\ (b,f,3)\} \\
S_{c,1} &= \{c,\ (a,c,1),\ (c,d,1)\} \\
S_{c,2} &= \{c,\ (a,c,2),\ (c,d,2)\} \\
S_{c,3} &= \{c,\ (a,c,3),\ (c,d,3)\} \\
S_{d,1} &= \{d,\ (c,d,1),\ (d,e,1),\ (d,g,1),\ (d,f,1)\} \\
S_{d,2} &= \{d,\ (c,d,2),\ (d,e,2),\ (d,g,2),\ (d,f,2)\} \\
S_{d,3} &= \{d,\ (c,d,3),\ (d,e,3),\ (d,g,3),\ (d,f,3)\} \\
S_{e,1} &= \{e,\ (a,e,1),\ (b,e,1),\ (d,e,1),\ (e,g,1)\} \\
S_{e,2} &= \{e,\ (a,e,2),\ (b,e,2),\ (d,e,2),\ (e,g,2)\} \\
S_{e,3} &= \{e,\ (a,e,3),\ (b,e,3),\ (d,e,3),\ (e,g,3)\} \\
S_{f,1} &= \{f,\ (b,f,1),\ (d,f,1)\} \\
S_{f,2} &= \{f,\ (b,f,2),\ (d,f,2)\} \\
S_{f,3} &= \{f,\ (b,f,3),\ (d,f,3)\} \\
S_{g,1} &= \{g,\ (b,g,1),\ (d,g,1),\ (e,g,1)\} \\
S_{g,2} &= \{g,\ (b,g,2),\ (d,g,2),\ (e,g,2)\} \\
S_{g,3} &= \{g,\ (b,g,3),\ (d,g,3),\ (e,g,3)\}
\end{aligned}$$

además de todos los conjuntos $\{(x,y,i)\}$ para cada arista (x,y) de E y color $1 \leq i \leq k$. La solución correspondiente al coloreo que hemos dado, identificando el color de a con 1, el de b con 2, y el de c con 3, correspondería a elegir $S_{a,1}$, $S_{b,2}$, $S_{c,3}$, $S_{d,2}$, $S_{e,3}$, $S_{f,1}$, $S_{g,1}$, más todos los $\{(x,y,i)\}$ que falten para completar $E \times \{1,2,3\}$.

Otro problema de optimización importante, con aplicaciones obvias a, por ejemplo, transporte o almacenamiento de mercancías, es el de la mochila (también llamado suma de subconjuntos). Se trata de ver si es posible llenar exactamente una mochila (unidimensional) eligiendo objetos de distinto tamaño de un conjunto. La versión bidimensional (mucho más difícil intuitivamente) tiene aplicaciones a corte de piezas en láminas de madera, por ejemplo.

Definición 6.18 *El* problema de la mochila (knapsack) *es el de, dado un multiconjunto finito de números naturales y un natural K, determinar si es posible elegir un subconjunto de esos números que sume exactamente K. El lenguaje* KNAPSACK *es el de los pares $(\mathcal{S}, K)$ tal que un subconjunto de $\mathcal{S}$ suma K.*

Ejemplo 6.14 ¿Es posible llenar exactamente una mochila de tamaño 50 con objetos del conjunto (de tamaños) $\{17, 22, 14, 6, 18, 25, 11, 17, 35, 45\}$? No es fácil encontrar la respuesta, pero existe una solución: $22 + 11 + 17$.

Teorema 6.9 KNAPSACK *es NP-completo.*

Prueba: Una MTND puede adivinar el subconjunto correcto y sumarlo, por lo que el problema está en $\mathcal{NP}$. Para ver que es NP-completo, mostraremos que EC $\leq$ KNAPSACK.

La idea es partir de una entrada a EC $\mathcal{S} = \{S_1,\ S_2, \ldots,\ S_n\}$, con $S = \cup_{1 \leq i \leq n} S_i = \{x_0,\ x_1, \ldots,\ x_m\}$. Identificaremos cada elemento x_j con el número 2^{tj}, para $t = \lceil \log_2(n+1) \rceil$. A cada conjunto $S_i = \{x_{i_1},\ x_{i_2}, \ldots,\ x_{i_{m_i}}\}$ le haremos corresponder el número $N_i = 2^{ti_1} + 2^{ti_2} + \ldots + 2^{ti_{m_i}}$. Nuestro problema de KNAPSACK es entonces $(\{N_1,\ N_2, \ldots,\ N_n\}, K)$, con $K = 2^0 + 2^t + 2^{2t} + \ldots + 2^{mt}$. El largo de la entrada a KNAPSACK es $O(nmt)$ que es polinomial en $|\mathcal{S}|$, y no es difícil construir esta entrada en tiempo polinomial. Veremos ahora que la transformación es correcta.

- Supongamos que existen conjuntos $S_{i_1},\ S_{i_2}, \ldots,\ S_{i_k}$ que cubren S exactamente. Entonces, cada x_j aparece exactamente en un S_{i_r}, por lo que al sumar $N_{i_1} + N_{i_2} + \ldots + N_{i_k}$ el sumando 2^{tj} aparece exactamente una vez, para cada j. No hay otros sumandos, de modo que la suma de los N_{i_r} es precisamente K.

- Supongamos que podemos sumar números $N_{i_1} + N_{i_2} + \ldots + N_{i_k} = K$. Cada uno de los términos 2^{tj} de K se pueden obtener únicamente mediante incluir un N_{i_r} que contenga 2^{tj}, pues sumando hasta n términos $2^{t(j-1)}$ se obtiene a lo sumo $n2^{t(j-1)} = 2^{t(j-1)+\log_2 n} < 2^{tj}$. Similarmente, si incuyéramos sumandos N_{i_r} que contuvieran un 2^{tj} repetido, sería imposible deshacernos de ese término 2^{tj+1} que no debe aparecer en K, pues ni sumándolo n veces llegaríamos al $2^{t(j+1)}$. Por lo tanto, cada término 2^{tj} debe aparecer exactamente en un N_{i_r}, y entonces los S_{i_r} forman un recubrimiento exacto.

$\square$

Ejemplo 6.15 Tomemos el siguiente problema de EC: $\{a,b,c\},\ \{a,b,e\},\ \{b,d\},\ \{c,e\},\ \{c,d\}$, que tiene solución $\{a,b,e\},\ \{c,d\}$. En este caso $n = 5$ y por lo tanto $t = 3$. Asociaremos 2^0 a a, 2^t a b, y así hasta 2^{4t} a e. Es ilustrativo escribir los números N_i en binario:

$$
\begin{array}{rcccl}
N_1 & = & 000\ 000\ 001\ 001\ 001 & = & 73 \\
N_2 & = & 001\ 000\ 000\ 001\ 001 & = & 4105 \\
N_3 & = & 000\ 001\ 000\ 001\ 000 & = & 520 \\
N_4 & = & 001\ 000\ 001\ 000\ 000 & = & 4160 \\
N_5 & = & 000\ 001\ 001\ 000\ 000 & = & 576 \\
K & = & 001\ 001\ 001\ 001\ 001 & = & 4681
\end{array}
$$

y efectivamente obtenemos $K = 4681$ sumando $N_2 + N_5 = 4105 + 576$.

6.6 La Jerarquía de Complejidad *[AHU74, sec 10.6 y cap 11]*

Terminaremos el capítulo (y el apunte) dando una visión superficial de lo que hay más allá en el área de complejidad computacional.

Notemos que, en todos los problemas NP-completos vistos en la sección anterior, siempre era fácil saber que el problema estaba en $\mathcal{NP}$ porque una MTND podía *adivinar* una solución que luego se verificaba en tiempo polinomial. Esto que se adivina se llama *certificado*: es una secuencia de símbolos que permite determinar en tiempo polinomial que $w \in L$ (por ejemplo la permutación de nodos para CH, el conjunto de números a sumar para KNAPSACK, etc.). Todos los problemas de $\mathcal{NP}$ tienen esta estructura: se adivina un certificado que después se puede verificar en tiempo polinomial (esto es general: un certificado válido para cualquier MTND es el camino en el árbol de configuraciones que me lleva al estado en que se detiene frente a w). El problema en una MTD es que no es fácil encontrar un certificado válido.

Pero ¿qué pasa con los complementos de los problemas que están en $\mathcal{NP}$? Por ejemplo, si quiero los grafos dirigidos que tienen un circuito hamiltoniano, me puede costar encontrar el circuito, pero si me dicen cuál es, me convencen fácilmente de que $G \in$ CH. Pero si quiero los grafos dirigidos que *no* tienen un circuito hamiltoniano, ¿qué certificado me pueden mostrar para convencerme de que no existe tal circuito? ¿Puede una MTND aceptar los grafos que no tienen un circuito hamiltoniano? ¿Los complementos de lenguajes en $\mathcal{NP}$ están en $\mathcal{NP}$? Es intrigante que esta pregunta no tiene una respuesta fácil, hasta el punto de que se define la clase co-$\mathcal{NP}$ para representar estos problemas.

Definición 6.19 *La clase* co-$\mathcal{NP}$ *es la de los lenguajes cuyo complemento está en* $\mathcal{NP}$*.*

Nótese que esto tiene que ver con la estructura asimétrica de la aceptación por una MTND: acepta si tiene forma de aceptar. Esto hace que sea difícil convertir una MTND que acepte un lenguaje L en otra que acepte L^c, incluso si estamos hablando de aceptar en tiempo polinomial. Esto no ocurre en $\mathcal{P}$, el cual es obviamente cerrado por complemento.

Como los complementos de los lenguajes en $\mathcal{P}$ sí están en $\mathcal{P}$, tenemos que $\mathcal{P} \subseteq \mathcal{NP} \cap$ co-$\mathcal{NP}$. Se cree que $\mathcal{NP} \neq$ co-$\mathcal{NP}$ y que ambos incluyen estrictamente a $\mathcal{P}$, aunque esto sería imposible si $\mathcal{P} = \mathcal{NP}$. Si $\mathcal{NP} \neq$ co-$\mathcal{NP}$, puede mostrarse que un problema NP-completo no puede estar en co-$\mathcal{NP}$, y un problema co-NP-completo (que se define similarmente) no puede estar en $\mathcal{NP}$. Por ello, si un problema está en $\mathcal{NP} \cap$ co-$\mathcal{NP}$, se considera muy probable que no sea NP-completo.

El ejemplo favorito de un problema que estaba en $\mathcal{NP} \cap$ co-$\mathcal{NP}$ pero no se sabía si estaba en $\mathcal{P}$ era el lenguaje de los primos, pero éste se demostró polinomial el año 2004.

Otra pregunta interesante es si hay problemas que no se sepa si son NP-completos ni si están en $\mathcal{P}$. Hay pocos. Uno de ellos es el *isormorfismo de grafos*: Dados grafos G y G' del mismo tamaño, determinar si es posible mapear los nodos de G a los de G' de modo que las aristas conecten los mismos pares, $(u,v) \in E \Leftrightarrow (f(u), f(v)) \in E'$.

Existe una clase natural que contiene a todas éstas, y tiene que ver con el *espacio*, no el *tiempo*, que requiere una MT para resolver un problema. Llamaremos $\mathcal{P}$-time y $\mathcal{NP}$-time a las clases $\mathcal{P}$ y $\mathcal{NP}$, para empezar a discutir el espacio también.

Definición 6.20 *La clase $\mathcal{P}$-space es la de los lenguajes que pueden decidirse con una MTD usando una cantidad de celdas de la cinta que sea un polinomio del largo de la entrada.*

Está claro que $\mathcal{P}$-time $\subseteq \mathcal{P}$-space, pues una MTND no puede tocar más celdas distintas que la cantidad de pasos que ejecuta (más las $n+2$ que ya vienen ocupadas por la entrada, pero eso no viene al caso). Más aún: como la simulación de una MTND con una MTD requería espacio polinomial en el usado por la MTND (Sección 4.5), resulta que $\mathcal{NP}$-time $\subseteq$ $\mathcal{P}$-space, y lo mismo co-$\mathcal{NP}$-time (el cual no parece que se pueda decidir en tiempo polinomial con una MTND, pero sí en tiempo exponencial con la simulación de la MTND de la Sección 4.5, pues basta responder lo contrario de lo que respondería la simulación determinística). Sin embargo, no se sabe si $\mathcal{P}$-time $\neq$ $\mathcal{P}$-space. Existe incluso el concepto de Pspace-completo.

Definición 6.21 *Un lenguaje es Pspace-completo si pertenece a $\mathcal{P}$-space y, si es decidido en tiempo $T(n)$ por una MTD, entonces todo otro problema en $\mathcal{P}$-space se puede decidir en tiempo $T(p(n))$ para algún polinomio $p(n)$.*

Está claro que si se encuentra una solución determinística de tiempo polinomial para un problema Pspace-completo, entonces $\mathcal{P}$-time $=$ $\mathcal{P}$-space $=$ $\mathcal{NP}$-time $=$ co-$\mathcal{NP}$-time. Esto se ve aún más improbable que encontrar que $\mathcal{P}$-time $=$ $\mathcal{NP}$-time.

Un término no demasiado importante pero que aparece con frecuencia es *NP-hard*: son los problemas tales que todos los $\mathcal{NP}$ reducen a ellos, pero que no están necesariamente en $\mathcal{NP}$, de modo que pueden incluir problemas intratables incluso con una MTND. Es interesante que, a diferencia de lo que ocurre con $\mathcal{P}$ y $\mathcal{NP}$, sí es posible demostrar que existe una jerarquía estricta en términos de espacio y de tiempo, incluyendo problemas que demostrablemente requieren espacio y tiempo exponencial. La mayoría de los problemas interesantes están realmente en $\mathcal{NP}$, y no se sabe si requieren tiempo exponencial en una MTD, por eso la importancia del problema abierto ¿$\mathcal{P} \neq \mathcal{NP}$? Sin embargo, podemos mostrar un problema relativamente natural que es efectivamente difícil.

Teorema 6.10 *El problema de determinar, dada una expresión regular R de largo $|R| = n$, si $\mathcal{L}(R) = \Sigma^*$:*

- *Es Pspace-completo con los operadores usuales para expresiones regulares (Def. 2.1).*
- *Requiere espacio exponencial en n (y por lo tanto tiempo exponencial en MTD o MTND) si permitimos el operador de intersección al definir R (además de la concatenación, unión y clausura de Kleene). Estas expresiones regulares se llaman* semiextendidas*.*
- *Requiere espacio superior a $2^{2^{2^{2^{\cdot^{\cdot^{\cdot^{2^n}}}}}}}$, para cualquier cantidad fija de 2's, si además de la intersección permitimos la operación de complementar una expresión regular. Estas expresiones regulares se llaman* extendidas *y esas complejidades se llaman* no elementales*.*

Prueba: Ver [AHU74, sec 10.6, 11.3 y 11.4]. $\square$

La jerarquía de complejidad en espacio está expresada en el siguiente teorema, donde puede verse que es bien fina, por ejemplo se distingue espacio n del espacio $n \log \log n$.

Teorema 6.11 *Sean $f(n) \geq n$ y $g(n) \geq n$ dos funciones tal que $\lim_{n \longrightarrow \infty} \frac{f(n)}{g(n)} = 0$. Entonces existen lenguajes que se pueden reconocer en espacio $g(n)$ y no se pueden reconocer en espacio $f(n)$, usando MTDs.*

Prueba: Ver [AHU74, sec 11.1]. □

El resultado es ligeramente menos fino para el caso del tiempo.

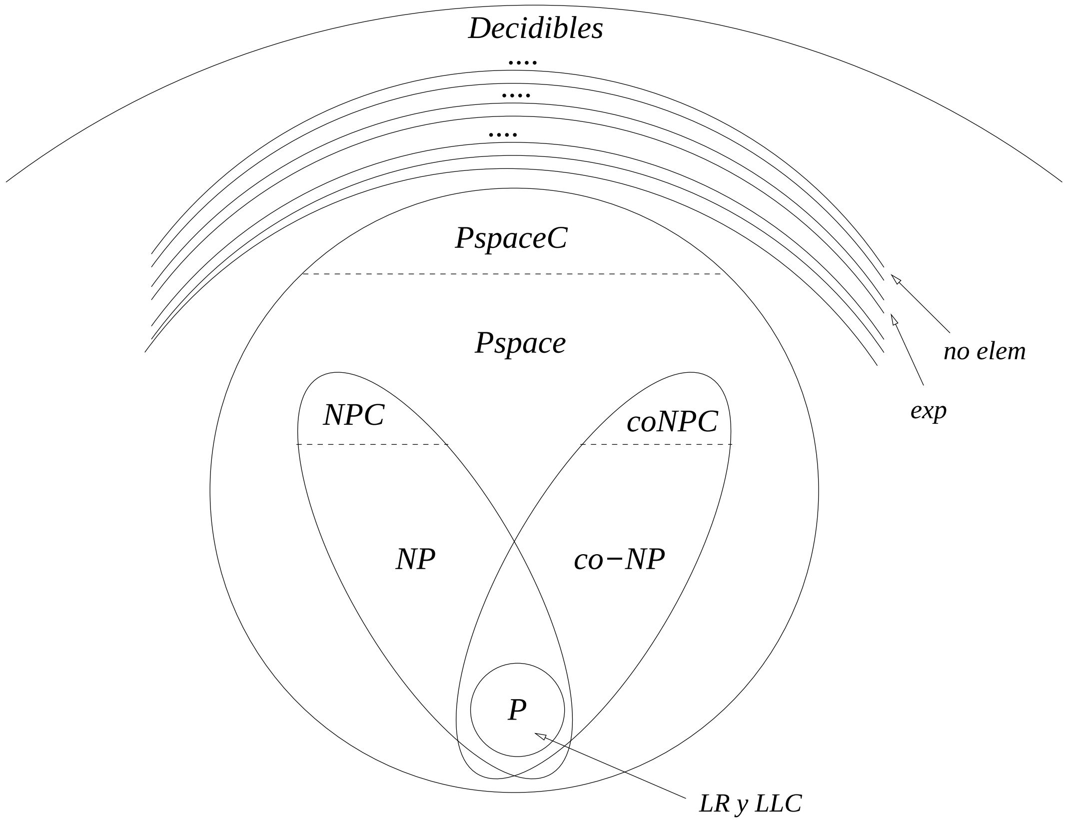
Teorema 6.12 *Sean $f(n) \geq n$ y $g(n) \geq n$ dos funciones tal que $\lim_{n \longrightarrow \infty} \frac{f(n) \log f(n)}{g(n)} = 0$. Entonces existen lenguajes que se pueden reconocer en tiempo $g(n)$ y no se pueden reconocer en tiempo $f(n)$, usando MTDs.*

Prueba: Ver [AHU74, ejercicios cap 11] (no resueltos), o [LP81, sec 7.7], para el caso más simple donde $\lim_{n \longrightarrow \infty} \frac{f(n)^2}{g(n)} = 0$. □

En el caso de las MTNDs los resultados son bastante menos finos: hay cosas que se pueden resolver en tiempo (o espacio) n^{k+1} pero no en tiempo n^k, para cada entero $k > 0$.

Esto implica, por ejemplo, que existen problemas que se pueden resolver en tiempo 2^n en una MTD y que definitivamente no están en $\mathcal{P}$ (ni en $\mathcal{NP}$). Nótese que no se sabe gran cosa sobre la relación entre espacio y tiempo, más allá de lo mencionado: Todo lo que se pueda hacer en tiempo $T(n)$ (determinístico o no) se puede hacer en espacio $T(n)$, pero en particular no se sabe mucho de en qué tiempo se puede hacer algo que requiera espacio $S(n)$.

El siguiente esquema resume lo principal que hemos visto.



Para terminar, algunas notas prácticas. En la vida real, uno suele encontrarse con problemas de *optimización* más que de *decisión*. Por ejemplo, no queremos saber si podemos iluminar el barrio con 40 faroles (VC) sino cuál es el mínimo número de faroles que necesitamos. Normalmente la dificultad de ambas versiones del problema es la misma. Si uno puede resolver el problema de optimización, es obvio que puede resolver el de decisión. Al revés es un poco más sutil, pues debo hacer una búsqueda binaria en el espacio de las respuestas: preguntar si me alcanzan 40 faroles; si me alcanzan, preguntar si me alcanzan 20; si no me alcanzan, preguntar si me alcanzan 30; y así. El tiempo se incrementa usualmente sólo en forma polinomial.

Otro tema práctico es: ¿qué hacer en la vida real si uno tiene que resolver un problema NP-completo? Si la instancia es suficientemente grande como para que esto tenga importancia práctica (normalmente lo será), se puede recurrir a algoritmos *aproximados* o *probabilísticos*. Los primeros, para problemas de optimización, garantizan encontrar una respuesta suficientemente cercana a la óptima. Los segundos, normalmente para problemas de decisión, se pueden equivocar con una cierta probabilidad (en una o en ambas direcciones). Por ejemplo, si la mitad de las ramas del árbol de configuraciones de una MTND me lleva a detenerme (o a obtener certificados válidos), entonces puedo intentar elecciones aleatorias en vez de no determinísticas para obtener certificados. Si lo intento k veces, la probabilidad de encontrar un certificado válido es $1 - 1/2^k$. Sin embargo, no siempre tenemos esa suerte. En particular, existe otra jerarquía de complejidad que se refiere a cuánto se dejan aproximar los problemas NP-completos. Algunos lo permiten, otros no. En los proyectos se dan algunos punteros.

6.7 Ejercicios

1. Suponga que $L_1 \leq L_2$ y que $\mathcal{P} \neq \mathcal{NP}$. Responda y justifique brevemente.

   (a) Si L_1 pertenece a $\mathcal{P}$, ¿L_2 pertenece a $\mathcal{P}$?

   (b) Si L_2 pertenece a $\mathcal{P}$, ¿L_1 pertenece a $\mathcal{P}$?

   (c) Si L_1 es $\mathcal{NP}$-Completo, ¿es L_2 a $\mathcal{NP}$-Completo?

   (d) Si L_2 es $\mathcal{NP}$-Completo, ¿es L_1 a $\mathcal{NP}$-Completo?

   (e) Si $L_2 \leq L_1$, ¿son L_1 y L_2 $\mathcal{NP}$-Completos?

   (f) Si L_1 y L_2 son $\mathcal{NP}$-Completos, ¿vale $L_2 \leq L_1$?

   (g) Si L_1 pertenece a $\mathcal{NP}$, ¿es L_2 $\mathcal{NP}$-Completo?

2. Considere una fórmula booleana en *Forma Normal Disyuntiva (FND)* (disyunción de conjunciones de literales).

   (a) Dé un algoritmo determinístico polinomial para determinar si una fórmula en FND es satisfactible.

(b) Muestre que toda fórmula en FNC se puede traducir a FND (leyes de Morgan, primer año).

(c) ¿Por qué entonces no vale que SAT-FNC $\leq$ SAT-FND y $\mathcal{P} = \mathcal{NP}$?

3. Muestre que las siguientes variantes del problema CH también son NP-completas.

   - El grafo es no dirigido.
   - El camino no necesita ser un circuito (es decir, volver al origen), pero sí pasar por todos los nodos.
   - El grafo es completo (tiene todas las aristas) y es no dirigido, pero cada arista tiene un costo $c(u, v) \geq 0$ y la pregunta es si es posible recorrer todos los nodos (aunque se repitan nodos) a un costo de a lo sumo C. Este es el *problema del vendedor viajero o viajante de comercio.*

4. Use CLIQUE para mostrar que los siguientes problemas son NP-completos.

   - El problema del *conjunto independiente máximo (maximum independent set)* es, dado un grafo $G = (V, E)$ y un entero $k \leq |V|$, determinar si existen k nodos en G tales que no haya aristas entre ellos.
   - El problema de *isomorfismo de subgrafos* es, dados grafos no dirigidos G y G', determinar si G' es isomorfo a algún subgrafo de G. Un subgrafo de G se obtiene eligiendo un conjunto de vértices y quedándose con todas las aristas de G que haya entre esos nodos.
   - El mismo problema anterior, pero ahora podemos elegir algunas aristas, no necesariamente todas, al generar el subgrafo de G.

5. Parta de KNAPSACK para demostrar que los siguientes problemas son NP-completos.

   (a) El problema de *partición de conjuntos (set partition)* es, dados dos conjuntos de naturales, ¿es posible dividirlos en dos grupos que sumen lo mismo?

   (b) El problema de *empaquetamiento (bin packing)* es, dado un conjunto de números y un repositorio infinito de paquetes de capacidad K, ¿puedo empaquetar todos los ítems en a lo sumo k paquetes sin exceder la capacidad de los paquetes? Reduzca de *set partition.*

6. El problema del *camino más largo* en un grafo no dirigido G es determinar si G tiene un camino de largo $\geq k$ que no repita nodos. Muestre que este problema es NP-completo.

7. Considere la siguiente solución al problema de la mochila, usando programación dinámica. Se almacena una matriz $A[0..n, 0..K]$, de modo que $A[i, j] = 1$ sii es posible sumar exactamente j eligiendo números de entre los primeros i de la lista $N_1,\ N_2, \ldots,\ N_n$.

(a) Muestre que $A[i+1,j] = 1$ sii $A[i,j] = 1$ ó $A[i,j-N_{i+1}] = 1$.

(b) Muestre que A se puede llenar en tiempo $O(nK)$ usando la recurrencia anterior, de modo de resolver el problema de la mochila.

(c) ¿Esto implica que $\mathcal{P} = \mathcal{NP}$?

8. El problema de *programación entera* tiene varias versiones. Una es: dada una matriz A de $n \times m$ y un vector b de n filas, con valores enteros, determinar si existe un vector x de m filas con valores 0 ó 1 tal que $Ax = b$. Otra variante es $Ax \leq b$ (la desigualdad vale fila a fila). Otras variantes permiten que x contenga valores enteros.

   (a) Muestre que la primera variante que describimos (igualdad y valores de x en 0 y 1) es NP-completa (reduzca de KNAPSACK).

   (b) Muestre que la variante que usa $\leq$ en vez de $=$ es NP-completa (generalice la reducción anterior).

   (c) Muestre que la variante con $\leq$ y valores enteros para x es NP-completa (reduzca de la anterior).

6.8 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. En este capítulo en particular, para el que no existían guías previas, muchas de las preguntas de controles son ahora ejercicios, por eso no hay tantas aquí. Además este capítulo entró en el curso en 1999.

Ex 2000 Dibuje una jerarquía de inclusión entre los siguientes lenguajes: aceptables, decidibles, finitos, libres del contexto, regulares, $\mathcal{P}$, y $\mathcal{NP}$. Agregue también los *complementos* de los lenguajes en cada clase: los complementos de lenguajes finitos, los complementos de lenguajes regulares, etc. No confunda (por ejemplo) "complementos de lenguajes regulares" con "lenguajes no regulares". ¡Se pide lo primero!

Ex 2001 Un *computador cuántico* es capaz de escribir una variable binaria con ambos valores (0 y 1) *a la vez*, y la computación se separa en dos universos paralelos que no pueden comunicarse entre sí. Esto puede hacerse repetidamente para obtener secuencias de bits, subdividiendo las computaciones. Cada computación prosigue en su universo individual hasta que termina. El estado final del cálculo es una *superposición cuántica* de los resultados de todas las computaciones realizadas en paralelo. Luego, con un computador tradicional, es posible desentrañar algunos resultados en tiempo polinomial. En particular se puede descubrir si una determinada variable booleana está en cero en todos los resultados o hay algún 1.

La construcción real de computadores cuánticos está muy en sus comienzos. Suponiendo que esto se lograra, responda las siguientes preguntas, y si la respuesta es sí, indique qué impacto práctico tendría eso en el mundo.

(i) ¿Se podría decidir algún problema actualmente no decidible?
(ii) ¿Se podría aceptar algún problema actualmente no aceptable?
(iii) ¿Se podría resolver rápidamente algún problema NP-completo?

Ex 2003 Responda verdadero o falso a las siguientes afirmaciones, justificando en a lo sumo 3 líneas. Una respuesta sin justificación no tiene valor.

Sean P_1 y P_2 problemas (es decir, lenguajes).

(a) Si $P_1 \leq P_2$ y P_2 es NP-completo, entonces P_1 es NP-completo.

(b) Si P_1 y P_2 son NP-completos, entonces $P_1 \leq P_2$ y $P_2 \leq P_1$.

(c) Si P_1 está en $\mathcal{P}$, y $P_2 \leq P_1$, entonces P_2 está en $\mathcal{P}$.

(d) Si P_1 está en $\mathcal{NP}$ y $P_1 \leq P_2$, entonces P_2 está en $\mathcal{NP}$.

(e) Si P_2 está en $\mathcal{NP}$, P_1 es NP-completo y $P_1 \leq P_2$, entonces P_2 es NP-completo.

(f) Si P_1 es NP-completo, entonces es decidible.

Ex 2005 Los extraterrestres nos entregan una esfera metálica que recibe como entrada la descripción de una Máquina de Turing M y un input para M (en tarjetas perforadas). En menos de tres segundos la esfera se ilumina de verde si M se detiene frente a ese input y rojo sino. La esfera cambia la teoría de la computabilidad, porque

(a) Todos los problemas de decisión se vuelven computables.

(b) Todos los lenguajes aceptables pasan a ser decidibles.

(c) Todos los lenguajes decidibles pasan a ser libres del contexto.

(d) Todos los problemas de $\mathcal{NP}$ pasan a estar en $\mathcal{P}$.

En cada una de las opciones indique si es cierto o falso, argumentando.

Ex 2005 El profesor Locovich descubre un algoritmo polinomial para resolver el k-coloreo. Explique paso a paso cómo utilizaría ese algoritmo para resolver el problema del circuito hamiltoniano en tiempo polinomial, con las herramientas que usted conoce.

Ex 2005 Dado un conjunto de conjuntos $\{S_1, S_2, \ldots, S_n\}$ y un número $1 \leq k \leq n$, llamemos $\mathcal{C} = S_1 \cup S_2 \ldots \cup S_n$. Queremos saber si existe un $S \subseteq \mathcal{C}$ tal que $|S| = k$ y que S contenga algún elemento de cada S_i, para todo $1 \leq i \leq n$.

Demuestre que el problema es NP-completo (hint: use vertex cover).

Ex 2006 Considere la MT robotizada del ejercicio Ex 2006 del capítulo 4.

- Si tomáramos esta MT como un modelo válido de computación, ¿cambiaría el conjunto de lenguajes aceptables? ¿el de los lenguajes decidibles? ¿el estatus de $\mathcal{NP}$?
- ¿Existen ejemplos del mundo real que se puedan asimilar, con cierta amplitud de criterio, a este modelo de computación?

6.9 Proyectos

1. Sólo hemos cubierto un pequeño conjunto de problemas NP-completos representativos. Hemos dejado fuera algunos muy importantes. Es bueno que dé una mirada a los cientos de problemas NP-completos que se conocen. Puede ver, por ejemplo, el excelente libro [GJ03] o, en su defecto, `http://en.wikipedia.org/wiki/List_of_NP-complete_problems`.

2. Investigue sobre algoritmos de aproximación para problemas NP-completos (realmente de optimización), y qué problemas se dejan aproximar. Una buena referencia es [ACG+99]. También hay algo en [CLRS01, cap 35] y en [GJ03, cap 6].

3. Investigue sobre algoritmos probabilísticos o aleatorizados, en particular para resolver problemas NP-completos, y qué problemas se dejan resolver probabilísticamente. Una buena referencia es [MR95], y hay algo de material en [HMU01, sec 11.4].

4. Investigue más sobre la jerarquía de complejidad. Además de las referencias que ya hemos usado, [AHU01, cap 10 y 11] y [LP81, cap 7] (los cuales hemos resumido sólamente), hay algo en [HMU01, cap 11], en [DW83, cap 13 a 15], y en [GJ03].

5. Investigue sobre técnicas para resolver en forma exacta problemas NP-completos, de la mejor forma posible en la práctica. Una fuente es [AHU83, sec 10.4 y 10.5].

Referencias

[ACG+99] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kahn, A. Marchetti-Spaccamela, M. Protasi. *Complexity and Approximation.* Springer-Verlag, 1999.

[AHU74] A. Aho, J. Hopcroft, J. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[AHU83] A. Aho, J. Hopcroft, J. Ullman. *Data Structures and Algorithms.* Addison-Wesley, 1983.

[CLRS01] T. Cormen, C. Leiserson, R. Rivest, C. Stein. *Introduction to Algorithms.* MIT Press, 2001 (segunda edición). La primera es de 1990 y también sirve.

[DW83] M. Davis, E. Weyuker. *Computability, Complexity, and Languages.* Academic Press, 1983.

[GJ03] M. Garey, D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, 2003 (edición 23). La primera edición es de 1979.

[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.

[MR95] R. Motwani, P. Raghavan. *Randomized Algorithms.* Cambridge University Press, 1995.

Índice de Materias