APROBAR TC

FOR

DUMMIES®

A Reference for the Rest of Us!®

Contents

Chapter 1

Lenguajes formales

1.1 Alfabeto y motes

Alfabeto: conjunto finito no vacio, los elementos del cual se llaman **símbolos**

Utilizamos la letra Σ para referirnos a un alfabeto cualquiera.

Mote: dado un alfabeto, un mote es una secuencia finita de los símbolos del alfabeto.

- Longitud: el número de símbolos que lo componen
  - Se representa por $|w|$
- Mote vacio: mote de longitud 0.

Hay que distinguir la notación anterior de la que se usa para la cardinalidad de un conjunto. La cardinalidad de un conjunto s erepresenta con $||C||$.

Submote: es cualquier subcadena de símbolos consecutivos de un mote.

- Prefijos y sufijos son casos particulares.

Ocurrencias: el numero de ocurrencias de un sñimbolo v sobre un mote w es $|w|_v$

Σ^* representa el conjunto de todos los motes de un alfabeto. Su orden clasifica los motes por longitud y ordena lexicográficamente los motes de una misma longitud:

$$\lambda < a < b < aa < ba < bb < aaa < ...$$

1.2 Operaciones con motes

1.2.1 Concatenación

Sobre un alfabeto Σ, es la operación que hace corresponder a cada par de motes el mote formado por la juxtaposición del primero y el segundo:

$$\Sigma^* \times \Sigma^* \to \Sigma^*$$

$$(w_1, w_2) \to w_1.w_2$$

Si $w_1 = abb$ y $w_2 = ba$, tenemos que $w_1 w_2 = abbba$.

Propiedades:

- Operación asociativa
- El elemento neutro es λ
- Dota a Σ^* de estructura de monoide muy particular
  - Todos us elementos son simplificables

En algebra, un elemento x de un monoide M es simplificable si:

$$\forall y, z \in M \begin{cases} x.y = x.z \\ y.x = z.x \Rightarrow y = z \end{cases}$$

1.2.2 Revessament

El revessat de un mote w es el mote formado por los mismo símbolos escritos al inrevés.

1.3 Operaciones con lenguajes

1.3.1 Concatenación

1.3.2 Cierre de Kleene

1.3.3 Revessament

1.4 Morfismos y substituciones

1.4.1 Morfismos

Propiedades elementales

1.4.2 Substituciones

Otras propiedades elementales de las substituciones

Chapter 2

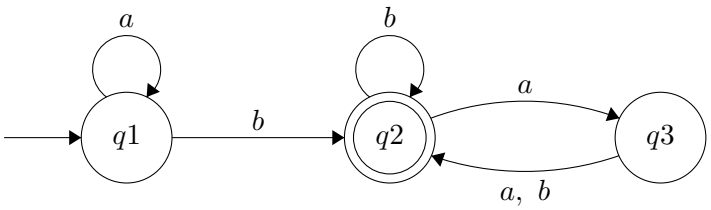
Lenguajes regulares

Introducción

- Ahora introducimos un modelo simple de ordenador con memoria finita.
- Este tipo de máquina será conocida como **autómata finito o máquina de estados finitos**.
- La idea básica de como un autómata finito funciona es:
  - Tenemos una palabra w definida en el alfabeto Σ; Entonces $w \in \Sigma^*$
  - La máquina irá leyendo los símbolos de w de izquierda a derecha.
  - Tras leer el último símbolo, sabremos si acepta o rechaza la palabra dada como entrada.
- Estas máquinas son útiles para saber si una palabra cumple unas condiciones impuestas, compiladores, etc.

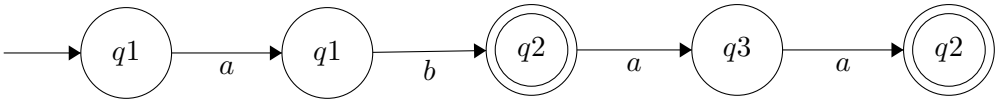
2.1 Autómata Finito Determinista (DFA)

Ejemplo: DFA con alfabeto $\Sigma = \{a, b\}$:

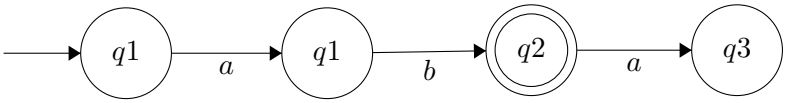


- $q1, q2, q3$ son los **estados**.
- $q1$ es el **estado inicial** ya que tiene una flecha que viene de la nada.
- $q2$ es el **estado aceptador** ya que tiene dibujado un doble círculo.

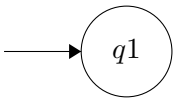
- Las flechas nos dicen como movernos cuando estamos en un estado y nos llega un símbolo de Σ.
- El DFA procesa cada símbolo de $w \in \Sigma^*$. Después de leer el último símbolo de w:
  - si el DFA se encuentra en un estado aceptador, entonces la palabra se **acepta**.
  - de lo contrario, se **rechaza**.
- veamos cómo se procesan las siguientes palabras sobre $\Sigma = \{a, b\}$ en la máquina de arriba:
  - *abaa* es aceptado



  - *aba* es rechazado



  - ε es rechazado

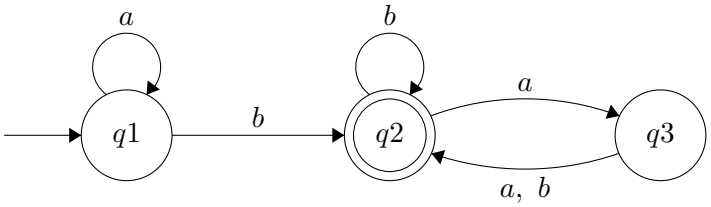


Definición: Un **autómata finito determinista (DFA)** es una tupla de 5 elementos:

$$M = (Q, \Sigma, \delta, q_0, F)$$

1. Q es un conjunto **finito** de estados.
2. Σ es un alfabeto, el DFA procesa palabras sobre Σ.
3. $\delta : Q \times \Sigma \to Q$ es la **función de transición**.
4. $q_0 \in Q$ es el **estado inicial**.
5. $F \subseteq Q$ es el conjunto de **estados aceptadores**.

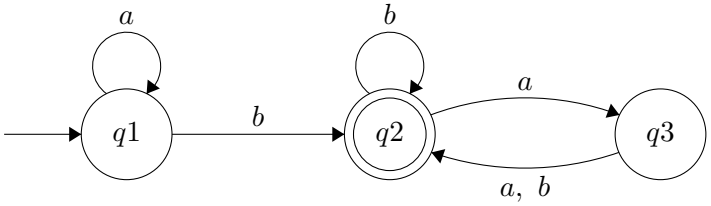
Función de transición de un DFA



La función de transición $\delta : Q \times \Sigma \to Q$ funciona de la siguiente manera:

- Por cada estado y por cada símbolo del alfabeto de entrada, la función δ nos da el siguiente estado.
- Concretamente, si $r \in Q$ y $l \in \Sigma$, entonces $\delta(r, l)$ es el estado del DFA al que nos vemos si estamos en el estado r y leemos l. Por ejemlo: $\delta(q2, a) = q3$.
- Por cada par de estado $r \in Q$ y símbolo $l \in \Sigma$,
  - hay **exactamente una** transición saliendo de r etiquetada l.
- Debido a eso solo hay un camino para procesar una palabra.
  - Por tanto, la máquina es **determinista**.

Ejemplo de un DFA



$M = (Q, \Sigma, \delta, q_1, F)$ con

- $Q = \{q_1, q_2, q_3\}$
- $\Sigma = \{a, b\}$
- $\delta : Q \times \Sigma \to Q$ se describe como

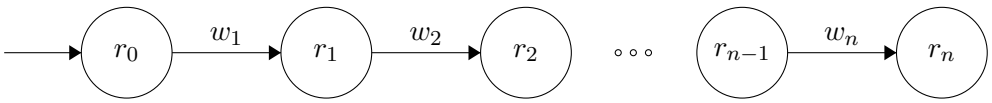
	a	b
q1	q1	q2
q2	q3	q2
q3	q2	q2

- q_1 es el estado inicial.
- $F = \{q_2\}$

Cómo se computa un DFA formalmente

- Sea $M = (Q, \Sigma, \delta, q_0, F)$ un DFA.
- La palabra $w = w_1 w_2 \cdots w_n \in \Sigma^*$, donde cada $w_i \in \Sigma$ y $n \geq 0$.
- Entonces M **acepta** w si existe una secuencia de estados $r_0, r_1, r_2, \cdots r_n \in Q$ tal que

1. $r_0 = q_0$
   - el primer estado r_0 de la secuencia es el principio del DFA.
2. $r_n \in F$
   - el último estado r_n de la secuencia es aceptador.
3. $\delta(r_i, w_{i+1}) = r_{i+1}$ por cada $i = 0, 1, 2, ..., n-1$
   - la secuencia de estados corresponde a transiciones válidas para la palabra w.

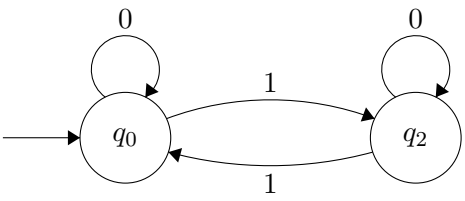


Lenguajes de una máquina

- **Definición:** Si A es un conjunto de todas las string que la máquina M acepta, entonces decimo
  - $A = L(M)$ es el **lenguaje de la máquina** M, y
  - M **reconoce** A.
- Si la máquina M tiene como entrada el alfabeto Σ, entonces $L(M) \subseteq \Sigma^*$.
- **Definición:** Un lenguaje es **regular** si es reconocido por algún DFA.

Ejemplo de un DFA

Ejemplo: Considera el siguiente DFA M_1 con el alfabeto $\Sigma = \{0, 1\}$:



Observaciones:

- 010110 es aceptado, pero 0101 es rechazado.
- $L(M_1)$ es el lenguaje sobre las palabras de Σ que tienen un número impar de 1's.

Construyendo el complementario de un DFA

- En general, dado un DFA M para el lenguaje A, podemos hacer un DFA $\overline{M}$ para $\overline{A}$ a partir de M mediante
  - cambiando todos los estados aceptadores en M a no aceptadores en $\overline{M}$,
  - cambiando todos los estados no aceptadores de M a aceptadores en $\overline{M}$,
- Más formalmente, suponemos que el lenguaje A con alfabeto Σ tiene un DFA

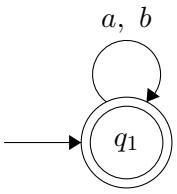
$$M = (Q, \Sigma, \delta, q_1, F).$$

- Entonces, el DFA para el lenguaje complementario $\overline{A}$ es

$$\overline{M} = (Q, \Sigma, \delta, q_1, Q - F).$$

  donde $Q, \Sigma, \delta, q_1, F$ son los mismos que en el DFA M.

Ejemplo: Consideremos el siguiente DFA M_6 con alfabeto $\Sigma = \{a, b\}$:



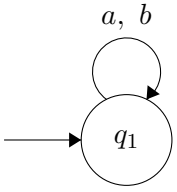
Observaciones:

- Este DFA acepta todos las posibles palabras sobre Σ, por ejemplo:

$$L(M_6) = \Sigma^*$$

- En general, cualquier DFA cuyos estados sean todos aceptadores reconoce el lenguaje Σ^*.

Ejemplo: Consideremos el siguiente DFA M_7 con alfabeto $\Sigma = \{a, b\}$:



Observaciones:

- Este DFA no acepta palabras sobre Σ, por ejemplo:

$$L(M_7) = \emptyset.$$

- En general,
  - un DFA puede no tener estados aceptadores, por ejemplo $F = \emptyset \subseteq \mathbb{Q}$.
  - cualquier DFA sin estados aceptadores reconoce el lenguaje $\emptyset$.

2.2 Algunas operaciones sobre lenguajes

- Sean A y B lenguajes-
- Antes hemos definido las operaciones:
  - **Unión:**

$$A \cup B = \{w | w \in A \text{ or } w \in B\}.$$

  - **Concatenación:**

$$A \cap B = \{vw | v \in A,\, w \in B\}.$$

  - **Estrella Kleen:**

$$A^* = \{w_1, w_2 \cdots w_k | k \geq 0 \text{ y cada } w_i \in A\}.$$

Cierre bajo operaciones

- Recordemos que dada una colección de objectos S, es **cerrada** bajo la operación f si al aplicar f a cada miembro de S siempre retorna un objecto dentro de S.
  - Por ejemplo, $\mathbb{N} = \{1, 2, 3, ...\}$ es cerrado bajo suma pero no resta.
- Antes vimos que dado un DFA M_1 para el lenguaje A, podemos construir un DFA M_2 para el lenguaje complementario $\overline{A}$.
- Además, la clase de los lenguajes regulares está cerrada por complemento.
  - Por ejemplo, si A es un lenguaje regular, entonces $\overline{A}$ también lo es.

2.2.1 Unión

Theorem 2.2.1 *La clase de lenguajes regulares es cerrada bajo unión.*

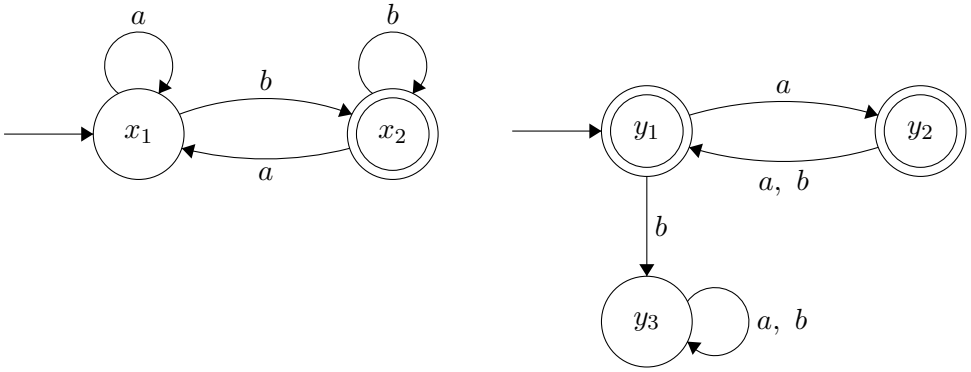
- Por ejemplo, si A_1 y A_2 son lenguajes regulares, entonces $A_1 \cup A_2$ también lo es.

Idea de demostración:

- Suponemos que A_1 es regular, por lo tanto tiene un DFA M_1.
- Suponemos que A_2 es regular, por lo tanto tiene un DFA M_2.
- $w \in A_1 \cup A_2$ si y solo si w es aceptado por M_1 o M_2.
- Necesitamos un DFA M_3 que acepte w si y solo si w es aceptado por M_1 o M_2.
- Construimos M_3 para llevar la cuenta de por dónde iría el input si estuviese corriendo simultáneamente en M_1 y M_2.
- Aceptamos la palabra si y solo si M_1 o M_2 acepta.

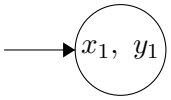
Ejemplo: Considremos los siguientes DFAs y el lenguaje sobre $\Sigma = \{a, b\}$:

- El DFA M_1 reconoce el lenguaje $A_1 = L(M_1)$
- El DFA M_2 reconoce el lenguaje $A_2 = L(M_2)$

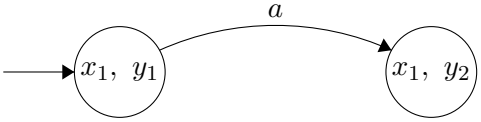


- Queremos un DFA M_3 para $A_1 \cup A_2$.

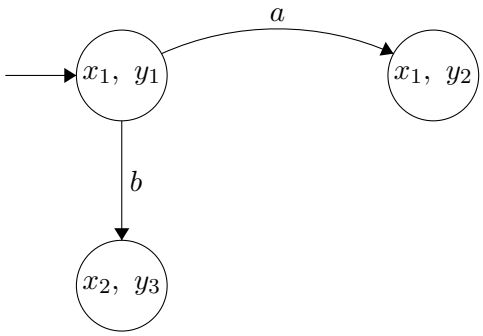
Paso 1: para construir un DFA M_3 para $A_1 \cup A_2$: Comenzamos en los estados iniciales de M_1 y M_2



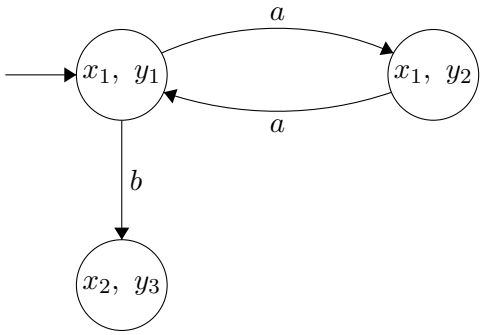
Paso 2: de (x_1, y_1) cuando llega una a, M_1 se mueve a x_1, y M_2 se mueve a y_2.



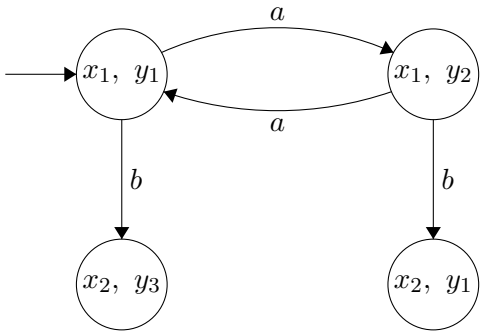
Paso 3: de (x_1, y_1) cuando llega una b, M_1 se mueve a x_2, y M_2 se mueve a y_3.



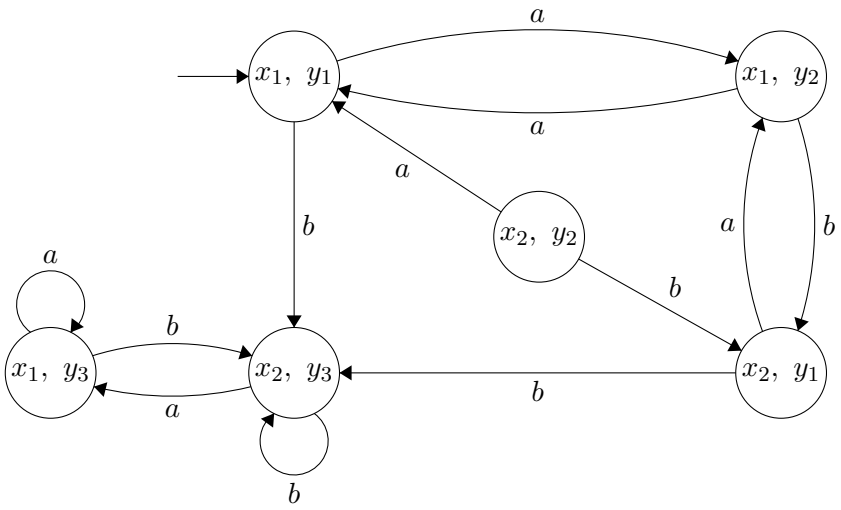
Paso 4: de (x_1, y_2) cuando llega una a, M_1 se mueve a x_1, y M_2 se mueve a y_1.



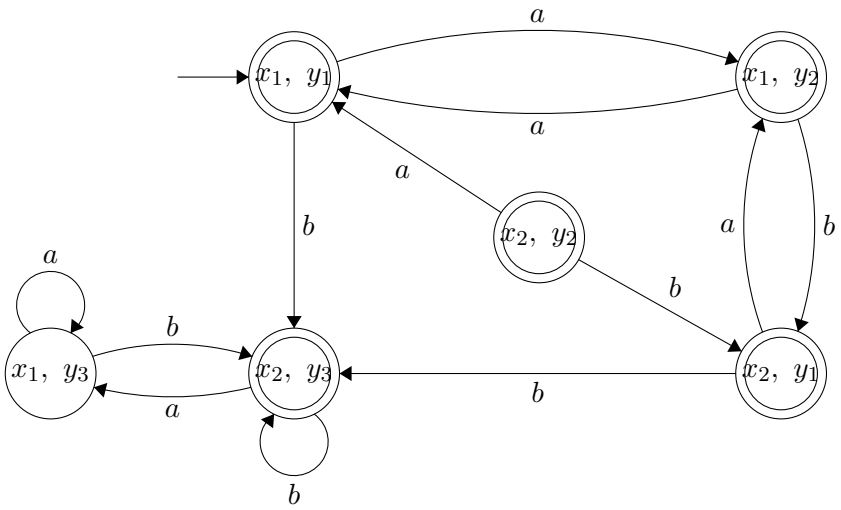
Paso 5: de (x_1, y_2) cuando llega una b, M_1 se mueve a x_2, y M_2 se mueve a y_1, ...



Continuamos hasta que cada nodo tenga transiciones de salida para cada símbolo de Σ.



Los estados aceptadores para el DFA M_3 para $A_1 \cup A_2$ son aquellos que lo eran de M_1 o M_2



Demostración de que los lenguajes regulares son cerrados bajo unión

- Suponemos que A_1 y A_2 están definidos bajo el mismo alfabeto Σ.
- Suponemos A_1 reconocido por el DFA $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$.
- Suponemos A_2 reconocido por el DFA $M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$.
- Definimos DFA $M_3 = (Q_3, \Sigma, \delta_3, q_3, F_3)$ para $A_1 \cup A_2$ como:
  - El conjunto de estados de M_3 es

$$Q_3 = Q_1 \times Q_2 = \{(x, y) | x \in Q_1, y \in Q_2\}.$$

  - El alfabeto de M_3 es Σ.
  - M_3 tiene la función de transición $\delta_3 : Q_3 \times \Sigma \to Q_3$ tal que para $x \in Q_1, y \in Q_2$, y $l \in \Sigma$,

$$\delta_3((x,y),l) = (\delta_1(x,l), \delta_2(y,l)).$$

  - El estado inicial de M_3 es

$$q_3 = (q_1, q_2) \in Q_3.$$

  - El conjunto de estados aceptadores de M_3 es

$$F_3 = \{(x,y) \in Q_1 \times Q_2 | x \in F_1 \text{ o } y \in F_2\} = [F_1 \times Q_2] \cup [Q_1 \times F_2].$$

- Ya que $Q_3 = Q_1 \times Q_2$,
  - el número de estados de la nueva máquina M_3 es $|Q_3| = |Q_1| \cdot |Q_2|$.
- Entonces, $|Q_3| < \infty$ ya que $|Q_1| < \infty$ y $|Q_2| < \infty$.

Observaciones:

- Podemos dejar fuera el estado $(x,y) \in Q_1 \times Q_2$ de Q_3 si (x,y) es inalcanzable desde el estado inicial de M_3, (q_1, q_2).
- Esto resultará en menos estados en Q_3, aún que seguimos teniendo un límite superior marcado por $|Q_1| \cdot |Q_2|$. Por ejemplo, $|Q_3| \leq |Q_1| \cdot |Q_2| < \infty$.

2.2.2 Intersección

Theorem 2.2.2 *La clase de lenguajes regulares es cerrada bajo intersección.*

- Por ejemplo, si A_1 y A_2 son lenguajes regulares, entonces $A_1 \cap A_2$ también lo es.

Idea de demostración:

- Suponemos que A_1 tiene un DFA M_1.
- Suponemos que A_2 tiene un DFA M_2.
- $w \in A_1 \cap A_2$ si y solo si w es aceptado por M_1 y M_2.
- Necesitamos un DFA M_3 que acepte w si y solo si w es aceptado por M_1 y M_2.
- Construimos M_3 para llevar la cuenta de por dónde iría el input si estuviese corriendo simultáneamente en M_1 y M_2.
- Aceptamos la palabra si y solo si M_1 y M_2 acepta.

2.2.3 Concatenación

Theorem 2.2.3 *La clase de lenguajes regulares es cerrada bajo concatenación.*

- Por ejemplo, si A_1 y A_2 son lenguajes regulares, entonces A_1 *o* A_2 también lo es.

Observaciones:

- Es posible (aún que pesado) contruir directamente un DFA para A_1 *o* A_2 dados los DFAs A_1 y A_2.
- Hay una forma más sencilla si introducimos un nuevo tipo de máquina, los NFA. Lo NFA son una generalización de las maquinas deterministas, por lo que cada DFA es automáticamente un NFA.

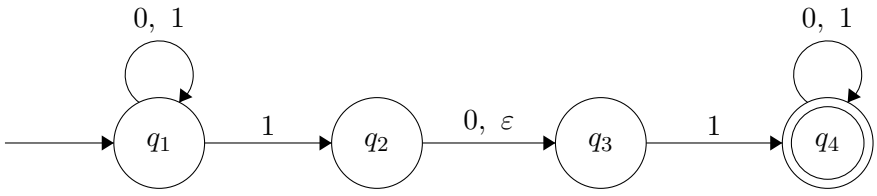
2.3 Autómata finito no determinista (NFA)

- Los NFAs nos permiten varias o ninguna elección tras leer un símbolo desde un estado en concreto
- Por cada estado q y símbolo $l \in \Sigma$, NFA puede tener
  - múltiples transiciones que salen del estado q tras leer l
  - ninguna transición sale del estado q tras leer l
  - transiciones que salen del estado q etiquetadas como ε
    - se puede coger una trasición ε sin necesidad de haber leído ningún símbolo de entrada.

Una forma de pensar en la computación no determinista es como un árbol de posibilidades. La raíz corresponde al inicio de la computación. Cada branca corresponde a un punto de la computación en la que la máquina puede tomar múltiples opciones. La máquina aceptara si una de estas opciones acepta.

Otra forma de verla es como una especie de computación paralela donde múltiples procesos se ejecutan de forma concurrente. Cuando el NFA se divide en multiples opciones corresponde a un proceso que hace "fork" en varios hijos.

Ejemplo: NFA N_1 con alfabeto $\Sigma = \{0, 1\}$.



- Supongamos que el NFA se encuentra en un estado con múltiples transiciones de salida. Por ejemplo el estado $q1$ tras leer un 1.
- La máquina se separa en múltiples instancias de ella misma (threads).
  - Cada copia procede a ejecutarse de forma independiente del resto.
  - El NFA puede estar en un **conjunto de estados** en vez de uno solo.
  - El NFA permite computar de forma paralela todos los posibles caminos.

  - Si una de esas copias lee un símbolo con el cual no puede proceder por ninguna transición de salida, entonces dicha copia (thread) **muere** o **crashea**.

- Si **cualquier** copia acaba en un estado aceptador tras leer la palabra de entrada completamente, entonces el NFA **acepta** la palabra.

- Si **ninguna** copia acaba en un estado aceptador tras leer la palabra de entrada completamente, entonces el NFA no acepta (**rechaza**) la palabra.

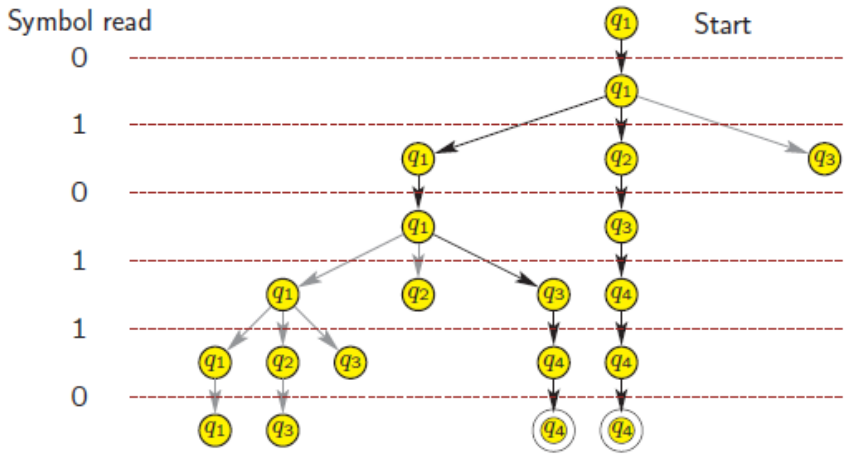
- De forma similar, si nos encontramos en un estado con una transición ε,

  - sin leer ningún símbolo el NFA se separa en múltiples copias, una por cada transición ε.

  - Cada copia procede independientemente del resto.

  - El NFA permite computar de forma paralela todos los posibles caminos.

  - El NFA procede de forma **no determinista** como antes.



Definición formal de un NFA

Definición: Para un alfabeto Σ, definimos $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$.

Definición: Un NFA es una tupla de 5 elementos $(Q, \Sigma, \delta, q_0, F)$, donde

1. Q es un conjunto finito de estados.

2. Σ es un alfabeto.

3. $\delta : Q \times \Sigma_\varepsilon \to \mathcal{P}(Q)$ es la función de transición, donde

   - $\mathcal{P}(Q)$ es el conjunto potencia de Q.

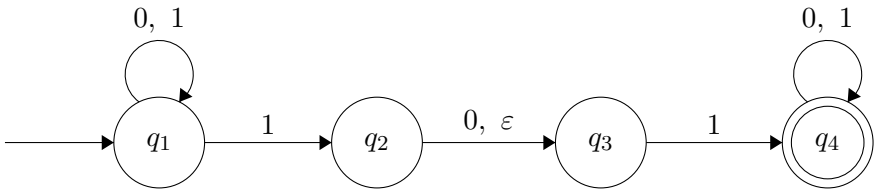
4. $q_0 \in Q$ es el estado inicial.

5. $F \subseteq Q$ es el conjunto de todos los estados aceptadores.

Diferencias entre DFA y NFA

- Los DFA tienen la función de transición $\delta : Q \times \Sigma \to Q$.
- Los NFA tienen la función de transición $\delta : Q \times \Sigma_\varepsilon \to \mathcal{P}(Q)$.
  - Retorna **un conjunto** de estados en vez de uno solo.
  - Permite transiciones ε ya que $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$.
  - Por cada estado $q \in Q$ y $l \in \Sigma_\varepsilon$, $\delta(q, l)$ es el conjunto de estados a los que se puede ir desde q tras leer l.

Observación: Todos los DFA son NFA.



Descripción formal del anterior NFA $N = (Q, \Sigma, \delta, q_1, F)$

- $Q = \{q_1, q_2, q_3, q_4\}$ es el conjunto de estados.
- $\Sigma = \{0, 1\}$ es el alfabeto.
- La función de transición $\delta : Q \times \Sigma_\varepsilon \to \mathcal{P}(Q)$

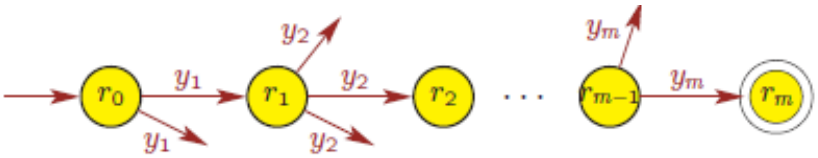
	0	1	ε
q_1	$\{q_1\}$	$\{q_1, q_2\}$	$\emptyset$
q_2	$\{q_3\}$	$\emptyset$	$\{q_3\}$
q_3	$\emptyset$	$\{q_4\}$	$\emptyset$
q_4	$\{q_4\}$	$\{q_4\}$	$\emptyset$

- q_1 es el estado inicial.
- $F = \{q_4\}$ es el conjunto de estados aceptadores.

Cómo se computa un NFA formalmente

- Sea $N = (Q, \Sigma, \delta, q_0, F)$ un NFA y $w \in \Sigma^*$.
- Entonces N **acepta** w si
  - podemos escribir w como $w = y_1 y_2 \cdots y_m$ para algún $m \geq 0$, donde cada símbolo $y_i \in \Sigma_\varepsilon$, y
  - hay una secuencia de estados $r_0, r_1, r_2, ..., r_m$ en Q tal que

1. $r_0 = q_0$
2. $r_{i+1} \in \delta(r_i, y_{i+1})$ para cada $i = 0, 1, 2, ..., m-1$
3. $r_m \in F$



Definición: El conjunto de todos las palabras aceptadas por el NFA N es **el lenguaje reconocido por** N y se denota como $L(N)$.

Equivalencia entre DFAs y NFAs

Definición: Dos máquinas (de cualquier tipo) son **equivalentes** si reconocen el mismo lenguaje.

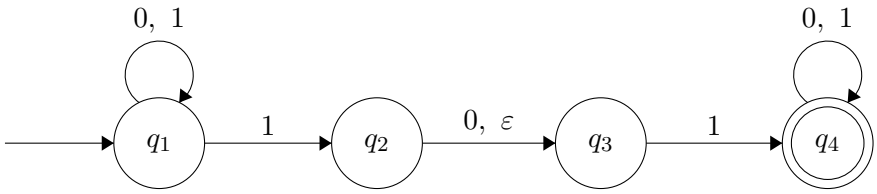
Theorem 2.3.1 *Cada NFA N tiene un DFA M que es equivalente.*

- Por ejemplo, si N es un NFA, entonces $\exists$ DFA M tal que $L(M) = L(N)$.

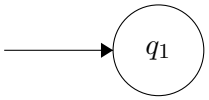
Idea de demostración:

- NFA N se divide en múltiples copias del mismo para movimientos no deterministas.
- El NFA puede estar en un conjunto de estados en un momento dado.
- Contrauir un DFA M cuyo conjunto de estados sea el **conjunto potencia** de los estados del NFA N, sabiendo en qué estado se encuentra N en un momento dado.

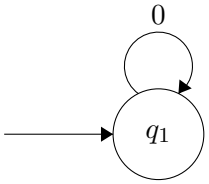
Ejemplo: Convertir el NFA N a un DFA equivalente.



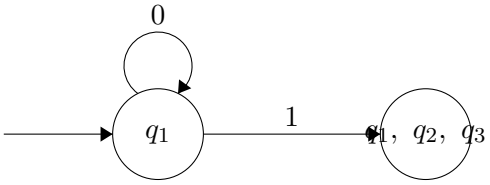
El estado inicial de N es q_1, no tiene ninguna transición ε de salida, por lo tanto, nuestro DFA tendrá de estado inicial $\{q_1\}$.



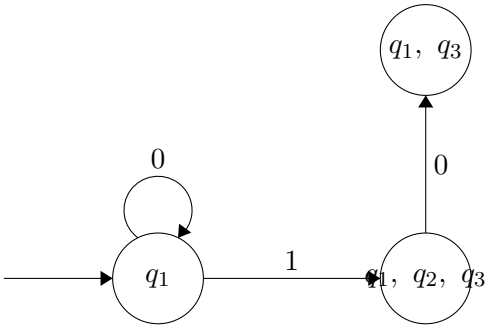
Cuando leemos 0 desde $\{q_1\}$, llegamos al mismo estado.



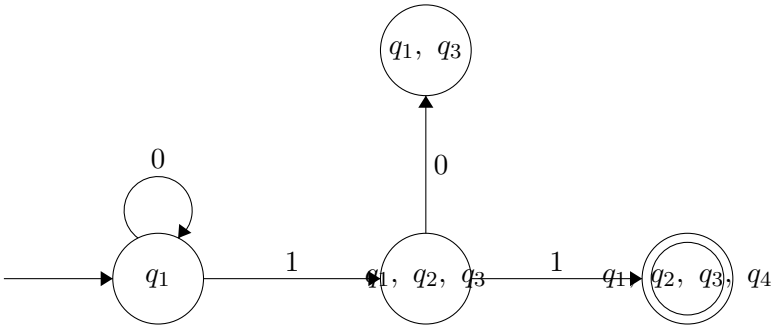
Al leer 1 desde el estado $\{q_1\}$, podemos ir a los estados $\{q_1, q_2, q_3\}$.



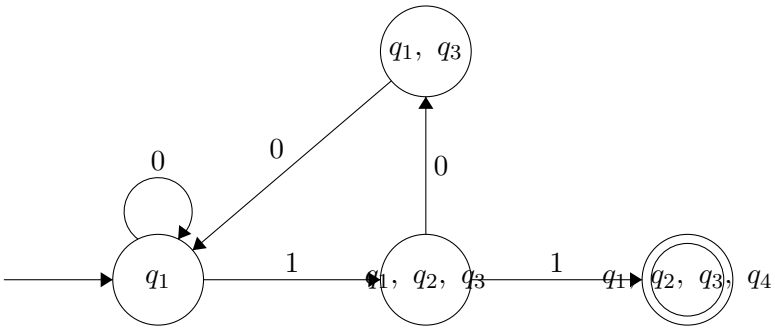
Al leer un 0 desde los estados $\{q_1, q_2, q_3\}$, podemos llegar a $\{q_1, q_3\}$.



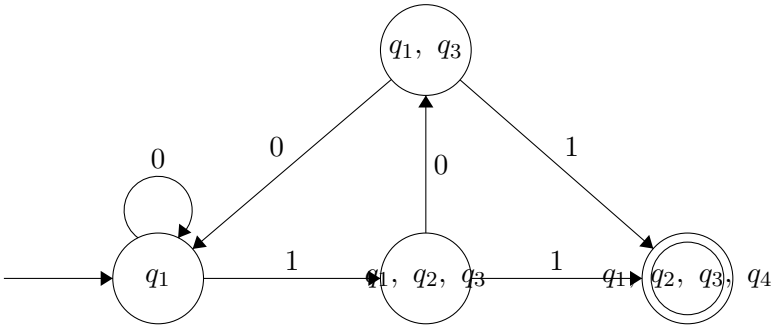
Al leer un 1 desde $\{q_1, q_2, q_3\}$, podemos ir a $\{q_1, q_2, q_3, q_4\}$.



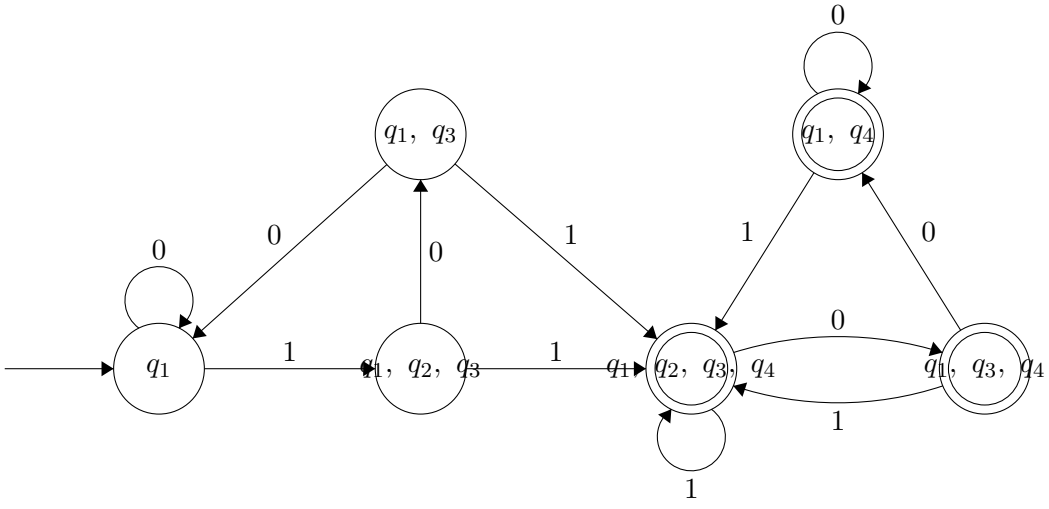
Al leer un 0 desde $\{q_1, q_3\}$, podemos ir a $\{q_1\}$.



Al leer un 1 desde $\{q_1, q_3\}$, podemos ir a $\{q_1, q_2, q_3, q_4\}$.

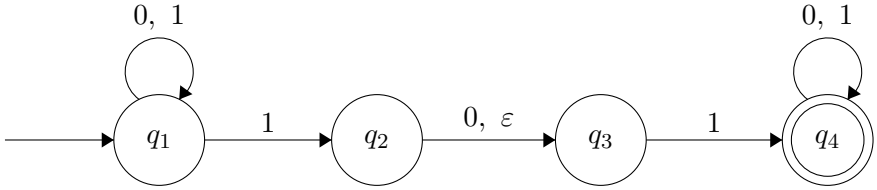


Continuamos hasta que todos los estados del DFA tengan transiciones de salida para 0 y 1. Los estados aceptadores del NFA son aquellos que tienen ≥ 1 estados aceptadores de N.



Demostración.

- Consideramos el NFA $N = (Q, \Sigma, \delta, q_0, F)$:



- **Definición:** El ε**-cierre** de un conjunto de estados $R \subseteq Q$ es

  $E(R) = \{q \mid$ tal que a q se puede llegar desde R moviéndose por 0 o más transiciones $\varepsilon\}$.

  – Por ejemplo, $E(\{q_1, q_2\}) = \{q_1, q_2, q_3\}$.

Convertir un NFA a su DFA equivalente

Dado un NFA $N = (Q, \Sigma, \delta, q_0, F)$, contruir un DFA equivalente $M = (Q', \Sigma, \delta', q_0', F')$ de la siguiente manera:

1. Calcular el ε**-cierre** de cada subconjunto $R \subseteq Q$.
2. Los estados del DFA M son $Q' = \mathcal{P}(Q)$.
3. El estado inicial del DFA M es $q_0' = E(\{q_0\})$.
4. El conjunto de estados aceptadores F' del DFA M son todos aquellos estados que continen alguno aceptador del NFA N.
5. Calculamos la función de transición $\delta' : Q' \times \Sigma \to Q'$ del DFA M' como:

   $$\delta'(R, l) = \{q \in Q | q \in E(\delta(r, l)) \text{ para algún } r \in R\}$$

   por $R \in Q' = \mathcal{P}(Q)$ y $l \in \Sigma$.
6. Podemos dejar fuera cualquier estado $q' \in Q'$ al que no podamos llegar desde q_0'. Por ejemplo, $\{q_2, q_3\}$ en nuestro ejemplo anterior.

Regular $\iff$ NFA

Corolario (1.40): El lenguaje A es regular si y solo algún NFA reconoce A.

Demostración.
($\Rightarrow$)

- Si A es regular, entonces hay un DFA que lo reconoce.
- Pero todos los DFAs son también NFAs, por tanto hay un NFA que reconoce A.

($\Leftarrow$)

- Lo hemos visto en el teorema anterior (2.3.1), el cual nos mostraba que cada NFA tenía un DFA equivalente.

2.4 Cierre de los lenguajes regulares

2.4.1 Unión

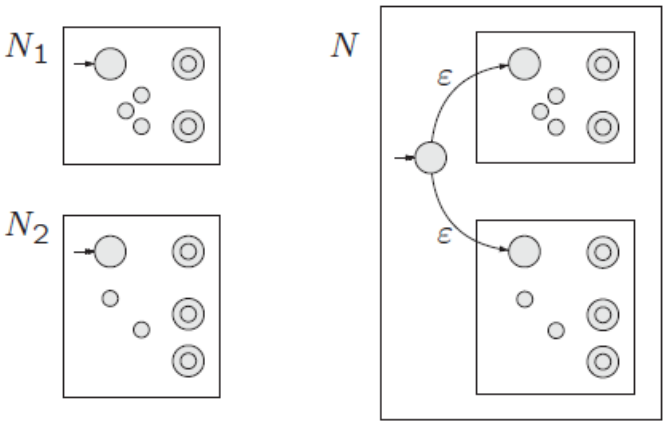
Observación: Podemos usar el hecho de que cada NFA tenga un DFA equivalente para simplificar al demostración de que la clase de los lenguajes regulares está cerrada bajo unión.

Observación: Recordemos la **unión**:

$$A_1 \cup A_2 = \{w | w \in A_1 \text{ o } w \in A_2\}.$$

Theorem 2.4.1 *La clase de los lenguajes regulares es cerrada bajo unión.*

Idea de demostración: Dado los NFAs N_1 y N_2 para A_1 y A_2 respectivamente, contruimos el NFA N para $A_1 \cup A_2$ de la siguiente forma:



Contruir un NFA $A_1 \cup A_2$ dado dos NFAs A_1 y A_2

- Sea A_1 el lenguaje reconocido por el NFA $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$.
- Sea A_2 el lenguaje reconocido por el NFA $N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$.
- Contruimos el NFA $N = (Q, \Sigma, \delta, q, F)$ para $A_1 \cup A_2$:
  - $Q = \{q_0\} \cup Q_1 \cup Q_2$ es el conjunto de estados de N.
  - q_0 es el estado inicial de N.
  - El conjunto de estados aceptadores es $F = F_1 \cup F_2$.
  - Por cada $q \in Q$ y $A \in \Sigma_\varepsilon$, la función de transición δ satisface

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & \text{si } q \in Q_1, \\ \delta_2(q,a) & \text{si } q \in Q_2, \\ \{q_1, q_2\} & \text{si } q = q_0 \text{ y } a = \varepsilon, \\ \emptyset & \text{si } q = q_0 \text{ y } a \neq \varepsilon. \end{cases}$$

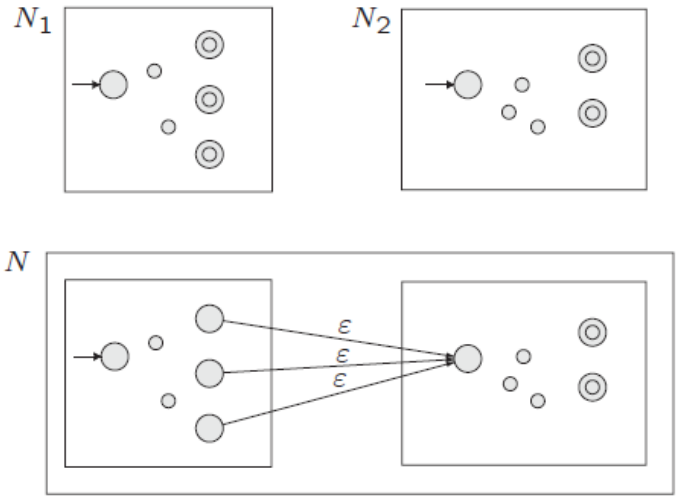
2.4.2 Concatenación

Observación: Recordemos la **concatenación**:

$$A_1 \; o \; A_2 = \{vw | v \in A, \, w \in B\}.$$

Theorem 2.4.2 *La clase de los lenguajes regulares es cerrada bajo concatenación.*

Idea de demostración: Dado los NFAs N_1 y N_2 para A_1 y A_2 respectivamente, contruimos el NFA N para $A_1 \; o \; A_2$ de la siguiente forma:



Contruir un NFA $A_1 \cup A_2$ dado dos NFAs A_1 y A_2

- Sea A_1 el lenguaje reconocido por el NFA $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$.
- Sea A_2 el lenguaje reconocido por el NFA $N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$.
- Contruimos el NFA $N = (Q, \Sigma, \delta, q, F)$ para $A_1 \; o \; A_2$:
  - $Q = Q_1 \cup Q_2$ es el conjunto de estados de N.
  - q_1 es el estado inicial de N ya que es el estado inicial de N_1.
  - El conjunto de estados aceptadores es F_2 ya que son los aceptadores de N_2.
  - Por cada $q \in Q$ y $A \in \Sigma_\varepsilon$, la función de transición δ satisface

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & \text{si } q \in Q_1 - F_1, \\ \delta_1(q,a) & \text{si } q \in F_1 \text{ y } a \neq \varepsilon, \\ \delta_1(q,a) \cup \{q_2\} & \text{si } q = F_1 \text{ y } a = \varepsilon, \\ \delta_2(q,a) & \text{si } q \in Q_2. \end{cases}$$

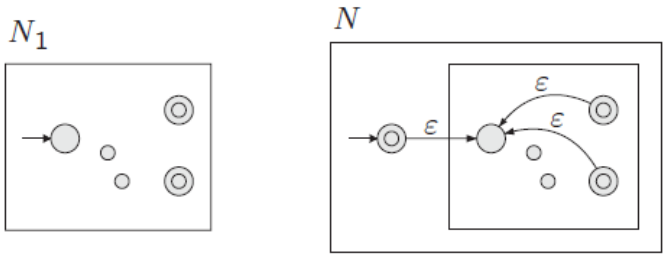
2.4.3 Estrella de Kleene

Observación: Recordemos la **estrella de Kleene**:

$$A^* = \{x_1 x_2 \cdots x_k | k \geq 0 \text{ y cada } x_i \in A\}.$$

Theorem 2.4.3 *La clase de los lenguajes regulares es cerrada bajo la operación de estrella de Kleene.*

Idea de demostración: Dado el NFA N_1 para A, construimos el NFA A^* de la siguiente forma:



Contruir un NFA A^* a partir del NFA para A

- Sea A el lenguaje reconocido por el NFA $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$.
- Contruimos el NFA $N = (Q, \Sigma, \delta, q, F)$ para A^*:
  - $Q = \{q_0\} \cup Q_1$ es el conjunto de estados de N.
  - q_0 es el estado inicial de N.
  - El conjunto de estados aceptadores es $F = \{q_0\} \cup F_1$.
  - Por cada $q \in Q$ y $A \in \Sigma_\varepsilon$, la función de transición δ satisface

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & \text{si } q \in Q_1 - F_1, \\ \delta_1(q,a) & \text{si } q \in F_1 \text{ y } a \neq \varepsilon, \\ \delta_1(q,a) \cup \{q_1\} & \text{si } q \in F_1 \text{ y } a = \varepsilon, \\ \{q_1\} & \text{si } q = q_0 \text{ y } a = \varepsilon, \\ \emptyset & \text{si } q = q_0 \text{ y } a \neq \varepsilon. \end{cases}$$

2.5 Expresiones regulares

- Las expresiones regulares son una forma de describir algunos lenguajes.
- Consideremos el alfabeto $\Sigma = \{0, 1\}$.
- Notación abreviada:
  - 0 quiere decir $\{0\}$
  - 1 quiere decir $\{1\}$
- Las expresiones regulares usan la notación abreviada y las operaciones
  - unión $\cup$
  - concatenación o
  - estrella de Kleene $*$
- Cuando usemos concatenación, normalmente ignoraremos el operador "o".

Interpretando expresiones regulares

Ejemplo: $0 \cup 1$ quiere decir $\{0\} \cup \{1\}$, equivale a $\{0, 1\}$.

Ejemplo:

- $(0 \cup 1)^*$ quiere decir $(\{0\} \cup \{1\})^*$.
- Esto equivale a $\{0, 1\}^*$, lo cual es el conjunto de todas las posibles palabras sobre el alfabeto $\Sigma = \{0, 1\}$.
- $\Sigma = \{0, 1\}$ normalmente usaremos notación abreviada Σ para denotar la expresión regular $(0 \cup 1)$.

Jerarquía de los operadores en expresiones regulares

- En la mayoría de lenguajes de programación,
  - la multiplicación tiene preferencia ante la suma.

$$2 + 3 \times 4 = 14$$

  - los paréntesis pueden cambiar el orden normal.

$$(2 + 3) \times 4 = 20$$

  - la exponenciación tiene preferencia ante la multiplicación y la suma.

- Orden de precedencia de operadores regulares:

  1. estrella de Kleene
  2. concatenación
  3. unión

- Los paréntesis pueden alterar el ordne.

Definición formal de expresión regular

Definición: R es una **expresión regular** con alfabeto Σ si R es

1. a para algún $a \in \Sigma$.
2. ε.
3. $\emptyset$.
4. $(R_1) \cup (R_2)$, donde R_1 y R_2 son expesiones regulares.
5. $(R_1)\ o\ (R_2)$, también escrito como $(R_1)(R_2)$ o R_1R_2, donde R_1 y R_2 son expresiones regulares.
6. $(R_1)^*$, donde R_1 es un expresión regular.
7. (R_1), donde R_1 es un expresión regular.

Se pueden quitar paréntesis redundantes. Por ejemplo, $((0) \cup (1))(1) \rightarrow (0 \cup 1)1$.

Definición: Si R es una expresión regular, entonces $L(R)$ es un lenguaje **generado** (o **descrito**) por R.

Algunos ejemplos de expresiones regulares

Ejemplos: Para $\Sigma = \{0, 1\}$,

1. $(0 \cup 1) = \{0, 1\}$
2. $0^*10^* = \{w | w$ tiene exactamente un $1\}$
3. $\Sigma^*1\Sigma^* = \{w | w$ tiene al menos un $1\}$
4. $\Sigma^*001\Sigma^* = \{w | w$ contiene la subpalabra $001\}$
5. $(\Sigma\Sigma)^* = \{w | |w|$ es par$\}$
6. $(\Sigma\Sigma\Sigma)^* = \{w | |w|$ es múltiplo de tres$\}$

7. $(0\Sigma^*0 \cup 1\Sigma^*1 \cup 0 \cup 1 = \{w|w$ empieza y acaba con el mismo símbolo$\}$

8. $1^*\emptyset = \emptyset$, cualquier cosa concatenada con $\emptyset$ es igual a $\emptyset$.

9. $\emptyset^* = \{\varepsilon\}$

Ejemplos:

1. $R \cup \emptyset = \emptyset \cup R = R$

2. $R \; o \; \varepsilon = \varepsilon \; o \; R = R$

3. $R \; o \; \emptyset = \emptyset \; o \; R = R$

4. $R_1(R_2 \cup R_3) = R_1R_2 \cup R_1R_3$. La concatenación se distribuye sobre la unión.

Ejemplo:

- Definimos PAR-PAR sobre el alfabeto $\Sigma = \{a, b\}$ como las palabras con un número par de a's y un número par de b's.

- Por ejemplo, $aababbaaababab \in$ PAR-PAR.

- Expresión regular:

$$(\; aa \cup bb \cup (ab \cup ba)(aa \cup bb)^*(ab \cup ba)\;)^*$$

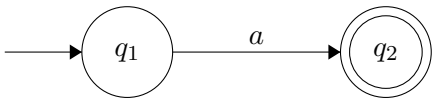
El teorema de Kleene

Theorem 2.5.1 *El lenguaje A es regular si y solo si A tiene una expresión regular.*

Lemma 2.5.1 *Si un lenguaje es descrito por una expresión regular, entonces es regular.*

Demostración:

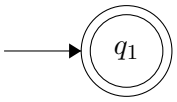
1. Si $R = a$ para algún $a \in \Sigma$, entonces $L(R) = \{a\}$, el cual tiene un NFA



   $N = (\{q_1, q_2\}, \Sigma, \delta, q_1, \{q_2\})$ donde la función de transición δ

   - $\delta(q_1, a) = \{q_2\}$,
   - $\delta(r, b) = \emptyset$ para cualquier estado $r \neq q_1$ o cualquier $b \in \Sigma_\varepsilon$ con $b \neq a$.

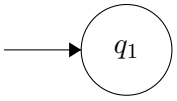
2. Si $R = \varepsilon$, entonces $L(R) = \{\varepsilon\}$, el cual tiene un NFA



$N = (\{q_1\}, \Sigma, \delta, q_1, \{q_1\})$ donde

- $\delta(r, b) = \emptyset$ para cualquier estado r y cualquier $b \in \Sigma_\varepsilon$.

3. Si $R = \emptyset$, entonces $L(R) = \emptyset$, el cual tiene un NFA



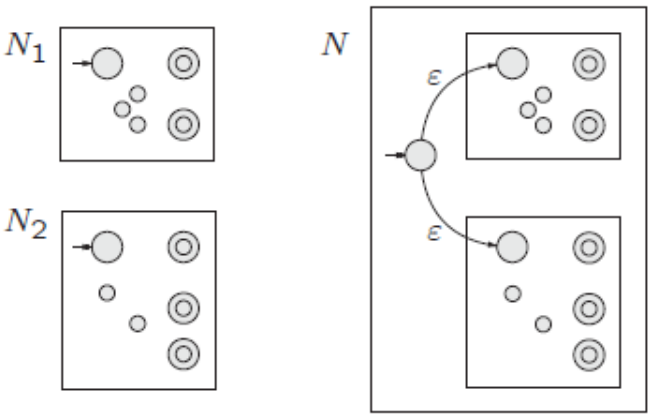
$N = (\{q_1\}, \Sigma, \delta, q_1, \emptyset)$ donde

- $\delta(r, b) = \emptyset$ para cualquier estado r y cualquier $b \in \Sigma_\varepsilon$.

4. Si $R = (R_1) \cup (R_2)$ y

- $L(R_1)$ tiene un NFA N_1.
- $L(R_2)$ tiene un NFA N_2.

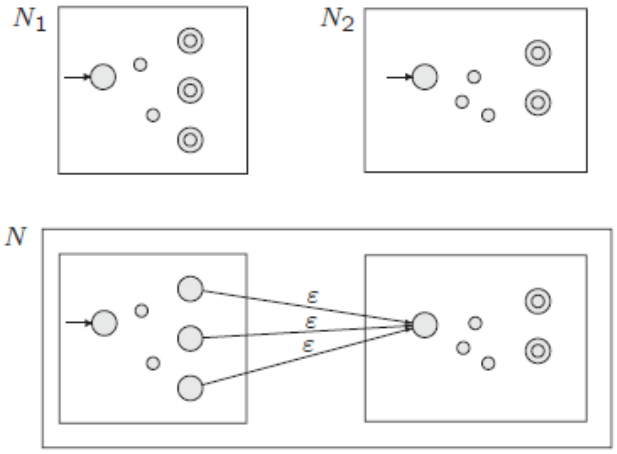
entonces $L(R) = L(R_1) \cup L(R_2)$ tiene el NFA N:



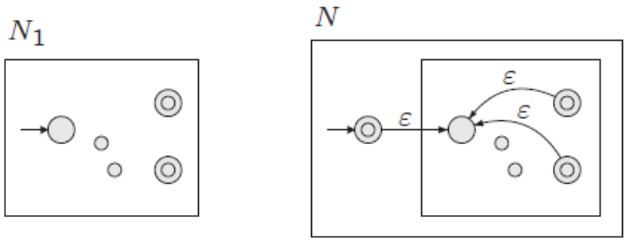
5. Si $R = (R_1)\ o\ (R_2)$ y

- $L(R_1)$ tiene un NFA N_1.
- $L(R_2)$ tiene un NFA N_2.

entonces $L(R) = L(R_1)\ o\ L(R_2)$ tiene el NFA N:

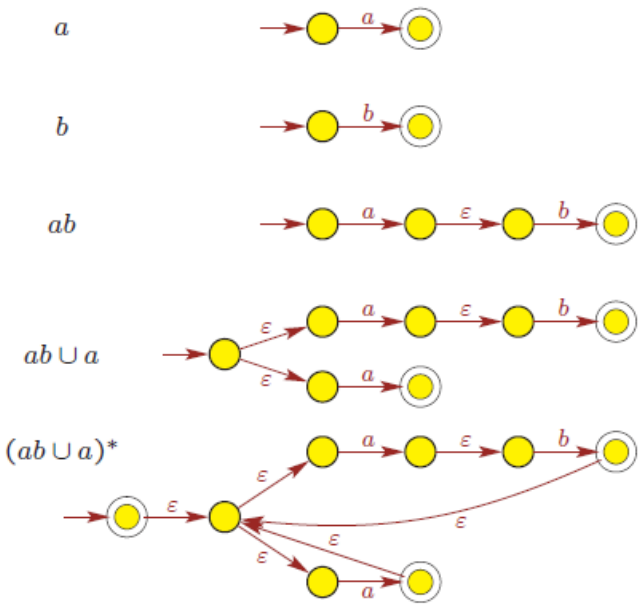


6. Si $R = (R_1)^*$ y $L(R_1)$ tiene un NFA N_1, entonces $L(R) = (L(R_1))^*$ tiene el NFA N:



- Por lo tanto, podemos convertir cualquier expresión regular en NFA.
- Entonces, por el corolario 1.40 podemos decir que el lenguaje $L(R)$ es regular.

Ejemplo: Contruir el NFA para $(ab \cup a)^*$



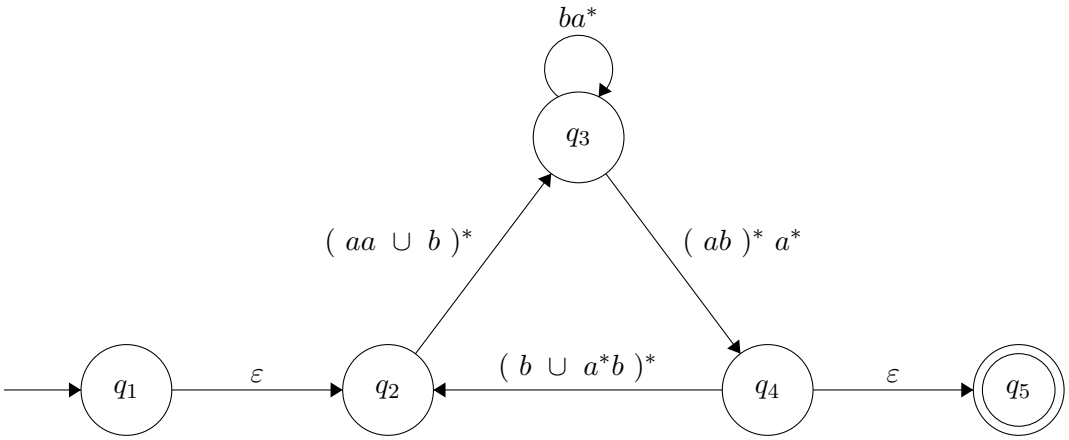
Más del teorema de Kleene

Lemma 2.5.2 *Si un lenguaje es regular, entonces tiene una expresión regular.*

Idea de demostración:

- Convertir un DFA en una expresión regular.
- Usar un **NFA generalizado (GNFA)**, que es un NFA con las siguientes modificaciones:
  - estado inicial sin transiciones de entrada.
  - un estado aceptador sin transiciones de salida.
  - las transiciones están etiquetadas con expresiones regulares en vez de elementos de Σ_ε.
    - se puede tomar una transición para cualquier palabra generada por su expresión regular.

Ejemplo: NFA



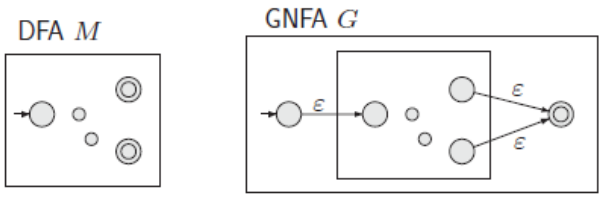
- Se puede mover desde
  - q_1 a q_2 con la palabra ε.
  - q_2 a q_3 con la palabra $aabaa$.
  - q_3 a q_3 con la palabra b o $baaa$.
  - q_3 a q_4 con la palabra ε.
  - q_4 a q_5 con la palabra ε.
- El GNFA acepta las palabras $\varepsilon\ o\ aabaa\ o\ b\ o\ baaa\ o\ \varepsilon\ o\ \varepsilon\ =\ aabaabbaaa$. (Los o son concatenación).

Método para convertir un DFA en expresión regular

1. Primero convertimos el DFA en un GNFA equivalente.
2. Aplicamos de forma iterativa los siguientes pasos:
   - En cada paso eliminamos un estado del GNFA.
     * Cuando un estado es eliminado, hay que mirar todos los caminos que antes eran posible.
     * Se pueden eliminar los estados en cualquier orden pero el resultado será diferente.
     * Nunca se deben borrar el estado inicial o aceptador.
   - Se acaba cuando quedan dos estados restantes: el inicial y el aceptador.
     * Se etiqueta la transición del estado inicial al aceptador con la expresión regular obtenida del lenguaje del DFA original.

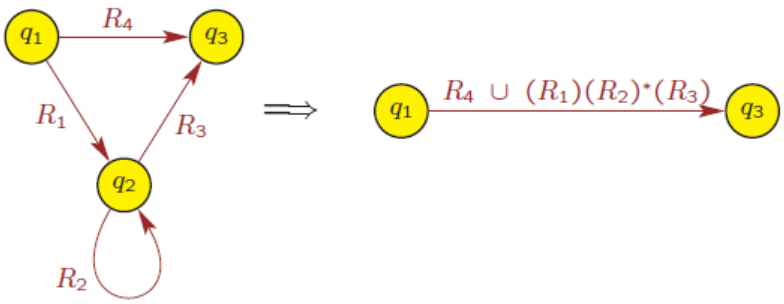
Observación: Este método también puede pasar de NFA a expresión regular.

1. Convertir el DFA $M = (Q, \Sigma, \delta, q_1, F)$ en un GNFA G equivalente.
   - Introducimos un nuevo estado inicial s.
     * Añadimos una transición de s a q_1 etiquetado como ε.
     * q_1 ya no es el estado inicial.
   - Introducimos un nuevo estado t.
     * Añadimos una transición etiquetada con ε por cada estado $q \in F$ a t.
     * Hacemos que cada estado en F ya no sea aceptador.
   - Cambiamos las etiquetas de las transiciones por expresiones regulares.
     * Por ejemplo, "a, b" a "$a \cup b$".

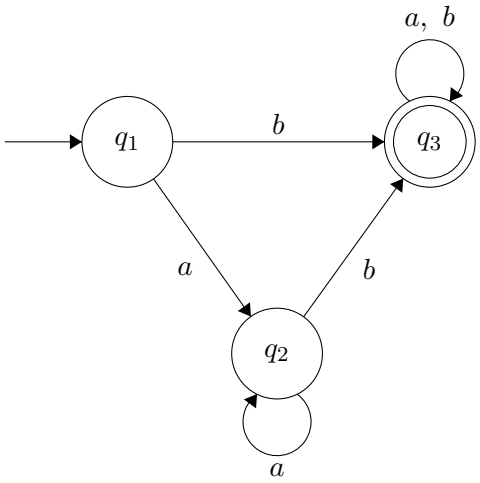


2. Iterativamente eliminamos un estado del GNFA G.
   - Tenemos que tener en cuenta todos los posibles caminos anteriores.
   - Nunca eliminamos el estado inicial s o aceptador t.

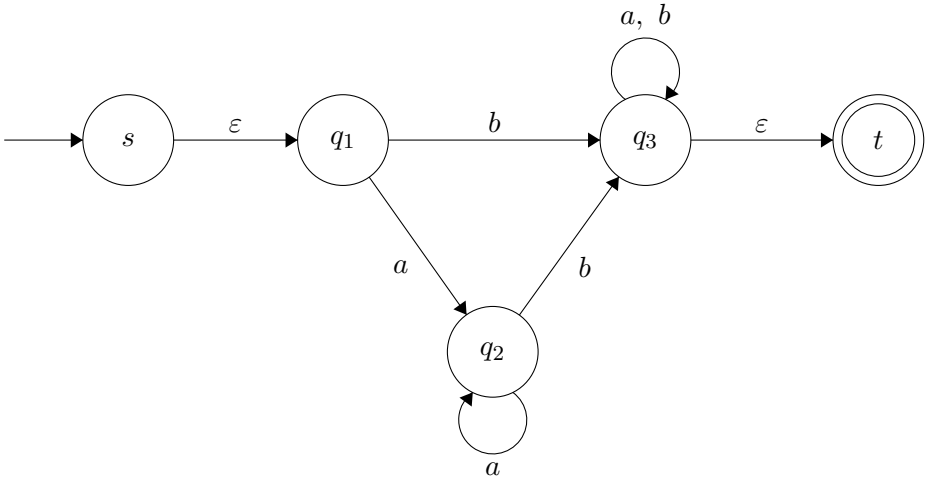
Ejemplo: Eliminar q_2 que no tiene otras transiciones de entrada/salida.



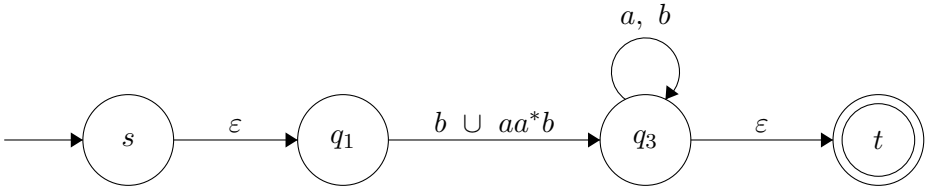
Ejemplo: Convertir el DFA M en una expresión regular.



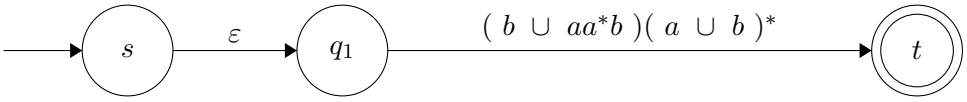
1. Pasamos de DFA a GNFA.



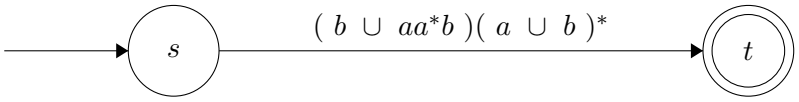
2.1 Eliminamos el estado q_2



2.2 Eliminamos el estado q_3



2.3 Eliminamos el estado q_1



Los lenguajes finitos son regulares

Theorem 2.5.2 *Si A es un lenguaje finito, entonces A es regular.*

Demostración:

- Ya que A es finito, podemos escribirlo como

$$A = \{w_1, w_2,..., w_n\}$$

  para algún $n < \infty$.

- Una expresión regular A es entonces

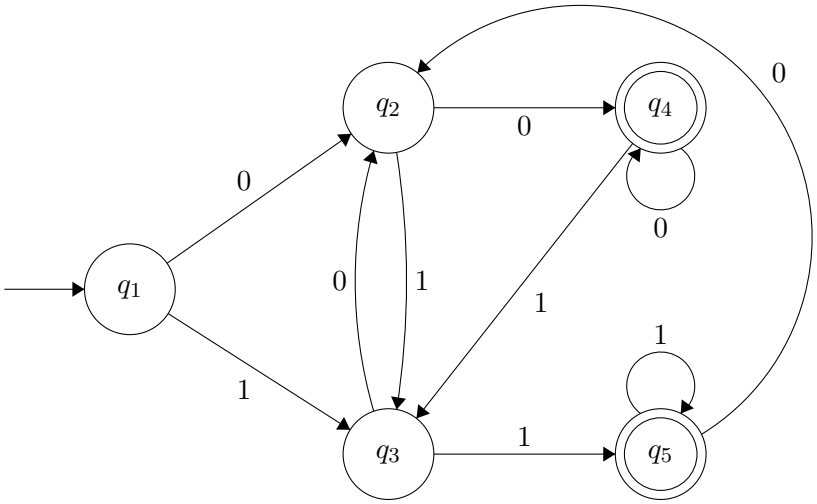
$$R = w_1 \cup w_2 \cup \cdots \cup w_n$$

- El teorema de Kleene implica A tiene un DFA, por lo tanto A es regular.

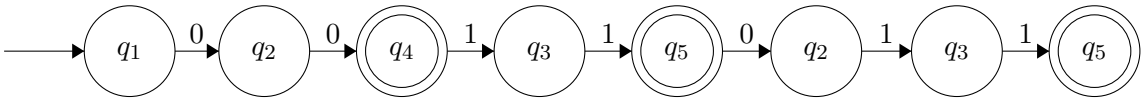
Observación: Lo contrario **no** es verdad. Por ejemplo, 1^* genera un lenguaje regular, pero es infinito.

Pumping Lemma para lenguajes regulares

Ejemplo: Dado un DFA con alfabeto $\Sigma = \{0, 1\}$ para el lenguaje A.



- El DFA tiene 5 estados.
- El DFA acepta la palabra $s = 0011$, que tiene longitud 4.
- Dada la palabra $s = 0011$, el DFA visita todos los estados.
- Para cualquier palabra con longitud ≥ 5, el principio del palomar garantiza que algún estado será visitado al menos dos veces.
- La palabra $s = 0011011$ es aceptada por el DFA.



- q_2 es el primer estado visitado dos veces.
- Usando q_2, dividimos la palabra s en tres partes x, y, z tal que $s = xyz$.
  - $x = 0$, los símbolos leídos hasta la primera visita a q_2.
  - $y = 0110$, los símbolos leídos entre la primera visita a q_2 y la segunda.
  - $z = 11$, los símbolos leídos después de la segunda visita a q_2.
- Recordemos que el DFA acepta la palabra

$$s = \underbrace{0}_{\text{x}}\underbrace{0110}_{\text{y}}\underbrace{11}_{\text{z}}.$$

- El DFA acepta también las palabras

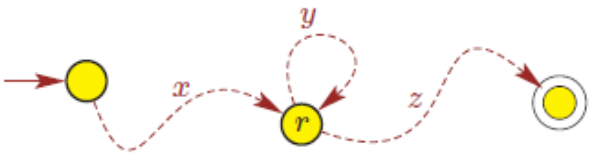
$$xyyz = \underbrace{0}_{\text{x}}\underbrace{0110}_{\text{y}}\underbrace{0110}_{\text{y}}\underbrace{11}_{\text{z}},$$

$$xyyyz = \underbrace{0}_{\text{x}}\underbrace{0110}_{\text{y}}\underbrace{0110}_{\text{y}}\underbrace{0110}_{\text{y}}\underbrace{11}_{\text{z}},$$

$$xz = \underbrace{0}_{\text{x}}\underbrace{11}_{\text{z}}.$$

- Palabras de la forma $xy^iz \in A$ por cada $i \geq 0$.

Pumping y



- Ya que y correponde a empezar en r y acabar en r,

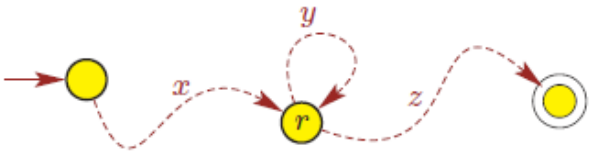
$$xy^iz \in A \text{ por cada } i \geq 1.$$

- También, nos damos cuenta de que $xy^0z = xz \in A$, por lo tanto

$$xy^iz \in A \text{ por cada } i \geq 0.$$

- $|y| > 0$ porque
  - y correponde a empezar en r y volver.
  - esto como mínimo consume un símbolo, por lo tanto y no puede estar vacío.

La longitud de xy



- $|xy| \leq p$, donde p es el número de estados del DFA, porque
  - xy son los símbolos leídos hasta la segunda visita a r.
  - Ya que r es el primer esstado que se visita dos veces, todos los estados visitados antes fueron únicos.

- Por lo tanto, antes de llegar a r por segunda vez, el DFA ha visitado como mucho p estados, lo que correponde a leer como mucho $p-1$ símbolos.

- La segunda visita a r (al leer un símbolo más) correponde a haber leído p símbolos como máximo.

Pumping Lemma

Theorem 2.5.3 ***Si*** *A es un lenguaje regular,* ***entonces*** *$\exists$ un número p (la longitud de pumping) donde,* ***si*** *$s \in A$ con $|s| \geq p$,* ***entonces*** *s se puede dividir en tres partes, $s = xyz$, satisfaciendo las condiciones*

1. *$xy^iz \in A$ para cada $i \geq 0$,*
2. *$|y| > 0$,* ***y***
3. *$|xy| \leq p$.*

Observaciones:

- y^i denota i copias de y concatenadas, y $y^0 = \varepsilon$.
- $|y| > 0$ significa que $y \neq \varepsilon$.
- $|xy| \leq p$ significa que x e y eon conjunto tienen menos de p símbolos.

2.6 Lenguajes no regulares

Definición: Un lenguaje es **no regular** cuando no tiene un DFA para el mismo.

Observaciones:

- El Pumping Lemma (PL) es el resultado de los lenguajes regulares.
- Aún así, el PL se usa principalmente para comprovar que el lenguaje A es **no regular**.
- Normalmente usamos **demostración por contradicción**.
  - Asumimos que A es regular.
  - El PL dice que todas las palabras $s \in A$ que tienen una longitud mínima, satisfacen unas condiciones.
  - Al coger un $s \in A$ correctamente, nos encontraremos con una contradicción.
  - El PL: **podemos** dividir s en $s = xyz$ satisfaciendo las condiciones 1-3.
  - Para llegar a la contraddición, llega a que **no se puede** dividir la palabra en $s = xyz$ satisfaciendo las condiciones 1-3.

– Ya que la condición 3 del PL dice que $|xy| \leq p$, normalmente cogemos $s \in A$ tal que todos los p símbolos son el mismo.

Lenguaje $B = \{ww \mid w \in \{0,1\}^*\}$ **es no regular**.

Demostración.

- Suponemos que B es regular, el PL imlica que B tiene una "longitud de pumping" p.
- Consideramos la palabra $s = 0^p 10^p 1 \in B$.
- $|s| = 2p + 2 \geq p$, de forma que podamos usar el PL.
- **Podemos** dividir en tres partes cumpliendo las condiciones anteriores.
- Para la contradicción, demostramos que **no** podemos satisfacer todas las condiciones a la vez.
  - Demostrar que **todas** las particiones $s = xyz$ que satisfacen las condiciones 2 y 3, no cumplen la 1.
- Ya que los primeros p símbolos de $s = \underbrace{00...0}_{\text{p}} 1 \underbrace{00...0}_{\text{p}} 1$ son todos 0's.
  - La condición 3 implica que tanto x como y tienen todo 0's.
  - z será el resto del primer set de 0's, seguidos por $10^p 1$.
- **La clave:** y contiene algunos de los primeros 0's, y z algunos de los segundos 0's, por lo tanto, haciendo pumping a y cambia el número de los primeros 0's.
- Tenemso entonces

$$x = 0^j \text{ para algún } j \geq 0,$$

$$y = 0^k \text{ para algún } k \geq 0,$$

$$z = 0^m 10^p 1 \text{ para algún } m \geq 0.$$

- $s = xyz$ implica

$$0^p 10^p 1 = 0^j 0^k 0^m 10^p 1 = 0^{j+k+m} 10^p 1,$$

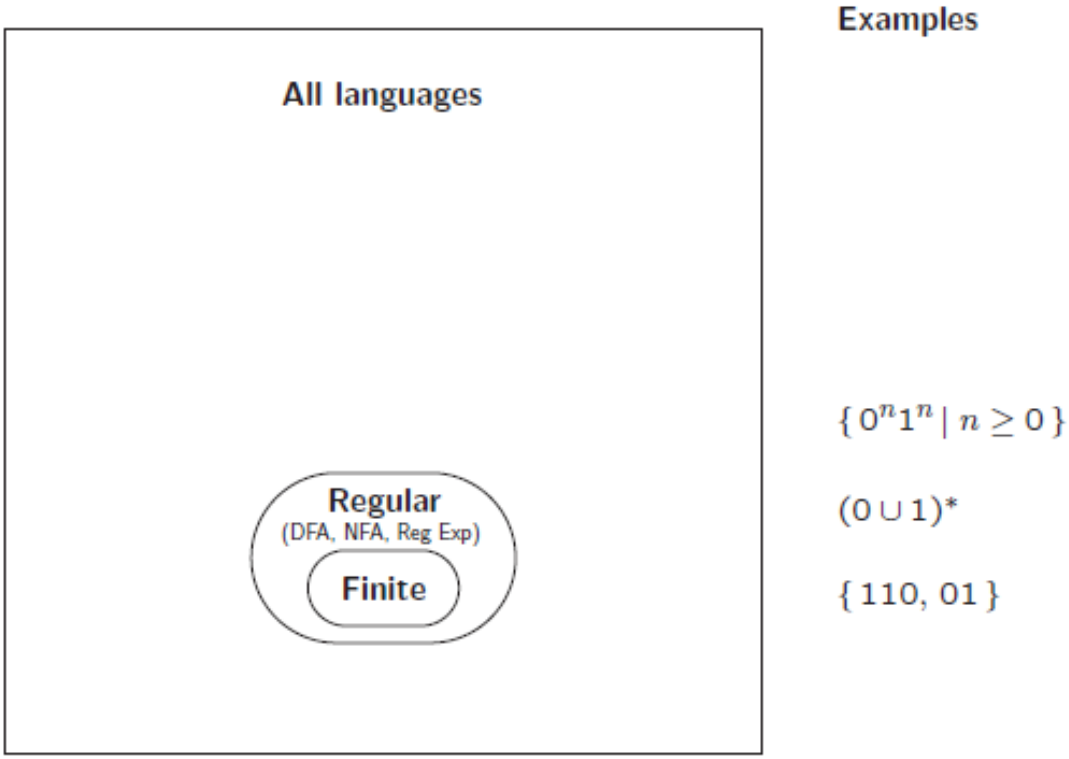
  por lo tanto $j + k + m = p$.
- La condición 2 implica que $|y| > 0$, por lo tanto $k > 0$.
- La condición 1 implica $xyyz \in B$, pero

$$xyyz = 0^j0^k0^k0^m10^p1 = 0^{j+k+k+m}10^p1 = 0^{p+k}10^p1 \notin B$$

ya que $j + k + m = p$ y $k > 0$.

- **Contradicción**, $B = \{ww \mid w \in \{0,1\}^*\}$ es no regular.

Jerarquía de los lenguajes (por ahora)



2.7 Resumen del capítulo

- Los DFA son máquinas deterministas que reconocen ciertos lenguajes.
- Un lenguaje es **regular** si es reconocido por un DFA.
- La clase de los lenguajes regulares es cerrada bajo unión, intersección, concatenación, estrella de Kleene y complementario.
- Los NFA pueden ser **no deterministas**: permiten elecciones dado un símbolo.
- Todo NFA tiene un DFA equivalente.
- Las expresiones regulares son una forma de generar algunos lenguajes.
- El teorema de Kleene: El lenguaje A tiene un DFA si y solo si tiene una expresión regular.

- Todo lenguaje finito es regular, pero no todo lenguaje regular es finito.
- Se usa el Pumping Lema para demostrar la no regularidad de algunos lenguajes.

Chapter 3

CFL - Context Free Languages

3.1 CFG - Context Free Grammars

Definicion 3.1.1 *Un CFG es una tupla de la forma* $G = (V, \Sigma, R, S)$*, donde:*

- V: es un conjunto finito de variables (no terminales)
- Σ: es un conjunto finito de terminales ($V \cap \Sigma = \emptyset$)
- R: es un conjunto finito de reglas de substitución (producciones)
- S: es la variable inicial, tal que $S \in V$

Las reglas de producción R son de la forma:

$$L \to X$$

Donde:

- $L \in V$
- $X \in (V \cup \Sigma)^*$

3.1.1 Ejemplo de CFG

Un ejemplo es el lenguaje $\{0^n 1^n | n \geq 0\}$

- Variables $V = S$
- Terminales $\Sigma = \{0, 1\}$
- Variable inicial S
- Reglas R:

$$S \to 0S1$$

$$S \to \lambda$$

Las reglas se pueden combinar con un or:

$$S \to 0S1|\lambda$$

3.1.2 Derivando strings usando CFG

Si tenemos que:

- $u, v, w \in (V \cup \Sigma)^*$
- $A \to w$ es una regla de la gramática

Entonces uAv produce uwv, lo que se escribe $uAv \Rightarrow uwv$

Una **derivación de un paso** consiste en la substitución de una variable por un string formado por terminales y variables de acuerdo a una regla de substitución. Por ejemplo, con la regla $A \to BC$:

$$01AD0 \Rightarrow 01BCD0$$

3.1.3 Lenguaje de un CFG

Definicion 3.1.2 *u **deriva** v, expresado de la forma $u \overset{*}{\Rightarrow} v$, si*

- $u = v$
- $\exists u_1, u_2, ..., u_k$ para alguna $k \geq 0$ tal que
  - $u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow ... \Rightarrow u_k \Rightarrow v$

$\overset{*}{\Rightarrow}$ denota una secuencia de ≥ 0 derivaciones de un paso.

Ejemplo: con las reglas $A \to B1|D0C$

$$0AA \overset{*}{\Rightarrow} 0D0CB1$$

Definicion 3.1.3 *El lenguaje de un CFG $G = (V, \Sigma, R, S)$ es:*

$$L(G) = \{w \in \Sigma^* | S \overset{*}{\Rightarrow} w\}$$

$\to$ se utiliza para definit reglas, $\Rightarrow$ para derivaciones.

3.1.4 Ejemplos de CFGs

Palindromos

```
S -> aSa | bSb | a | b | λ
```

CFG PAR-PAR

```
S -> aaS | bbS | XYXS | λ
X -> ab | ba
Y -> aaY | bbY | λ
```

CFG para expresiones aritméticas simples

```
S -> S + S | S - S | S x S | S/S | (S) | -S | 0 | 1 | ... | 9
```

Este CFG sería ambiguo porque se puede derivar la misma palabra de 2 formas diferentes:

3.1.5 Aplicaciones de CFLs

- Modelar lenguajes naturales (Noam Chomsky)
- Especificaciones de lenguajes de programación
  - Parsear un programa
- Describir estructuras matemáticas
- Clase intermedia entre lenguajes regulares y lenguajes computables.

3.1.6 Context-Free Languages

Definicion 3.1.4 *Cualquier lenguaje que puede ser generado por un CFG es un CFL.*

Algunos CFL son no regulares.

3.2 Forma normal de Chomsky

Definicion 3.2.1 *Un CFG $G = (V, \Sigma, R, S)$* ***esta en forma normal de Chomsky*** *si cada regla esta en alguna de las 3 formas siguientes*

$$A \to BC$$

$$A \to x$$

$$S \to \lambda$$

Con:

- variables $A \in V$ y $B, C \in V - \{S\}$
- terminal $x \in \Sigma$

Ejemplo:

```
S -> XX | XW | a | λ
X -> WX | b
W -> a
```

3.2.1 CFL a forma normal de Chomsky

Theorem 3.2.1 *Cualquier CFL puede ser descrito por una gramática en forma normal de Chomsky*

Demostración:

- Se comienza con un CFG $G = (V, \Sigma, R, S)$
- Se reemplaza, una por una, cada regla que no esta en la forma de Chomsky
- Hay que tener cuidado con:
  - La variable inicial (no permitida en las reglas RHS)
  - Reglas λ ($A \to \lambda$ no esta permitido cuando A no es una variable inicial)
  - Todas las otras violaciones de reglas ($A \to B$, $A \to aBc$, $A \to BCDE$)

3.2.2 Convertir CFG a la forma normal de Chomsky

1. Variable inicial no permitida en la RHS (right-hand side) de la regla
   - Nueva variable inicial S_0
   - Nueva regla $S_0 \to S$

2. Eliminar las $\lambda-$ producciones $A \to \lambda$

   - Antes: $B \to xAy$ $A \to \lambda|...$
   - Después: $B \to xAy|xy$ $A \to ...$

3. Eliminar las **reglas unitarias** $A \to B$

   - Antes: $A \to B$ y $B \to xCy$
   - Después: $A \to xCy$ y $B \to xCy$

4. Reemplazar las terminales problematicas a con una variable T_a con regla $T_a \to a$

   - Antes: $A \to ab$
   - Después: $A \to T_aT_b$ $T_a \to a$ $T_b \to b$

5. Acortar las RHS largas a secuencias con solo 2 variables cada una:

   - Antes: $A \to B_1B_2...B_k$
   - Después: $A \to B_1A_1$, $A_1 \to B_2A_2$ ... $A_{k-2} \to B_{k-1}B_k$
     Por lo que $A \Rightarrow B_1A_{11}B_2A_2 \Rightarrow ... \Rightarrow B_1B_2...B_k$

6. Tener cuidado a la hora de eliminar reglas

   - No introducir nuevas reglas que ya se han eliminado antes
   - Cuando se elimina $A \to \lambda$, insertar todos los nuevos reemplazos
     - Antes: $B \to AbA$ y $A \to \lambda|...$
     - Después: $B \to AbA|bA|Ab|b$ y $A \to ...$

Ejemplo de conversión a forma normal de Chomsky

CFG inicial:

```
S -> XSX | aY
X -> Y | S
Y -> b | λ
```

1. Introducir una nueva variable inicial y la nueva regla $S_0 \to S$

```
S0 -> S
S -> XSX | aY
X -> Y | S
Y -> b | λ
```

De la sección anterior:

```
S0 -> S
S -> XSX | aY
X -> Y | S
Y -> b | λ
```

2. Eliminar λ-reglas

Se elimina la λ producción de Y:

Eliminar $Y \rightarrow \lambda$

```
S0 -> S
S -> XSX | aY
X -> Y | S | λ
Y -> b
```

Eliminar $X \rightarrow \lambda$

```
S0 -> S
S -> XSX | aY | SX | XS | S
X -> Y | S
Y -> b
```

Por los pasos de transformación de la sección anterior:

```
S0 -> S
S -> XSX | aY | SX | XS | S
X -> Y | S
Y -> b
```

3. Eliminar las reglas unitarias:

(i) Eliminar la regla unitaria $S \rightarrow S$

(ii) Eliminar la regla $S0- > S$

```
S0 -> XSX | aY | a |SX | XS
S -> XSX | aY | SX | XS |
X -> Y | S
Y -> b
```

(iii) Eliminar la regla unitaria $X \rightarrow Y$

```
S0 -> XSX | aY | a |SX | XS
S -> XSX | aY | SX | XS |
X -> Y | S
Y -> b
```

```
S0 -> XSX | aY | a |SX | XS
S -> XSX | aY | SX | XS |
X ->S | b
Y -> b
```

(iv) Eliminar la regla unitaria $X \rightarrow S$

```
S0 -> XSX | aY | a |SX | XS
S -> XSX | aY | SX | XS |
X -> S | b
Y -> b
```

```
S0 -> XSX | aY | a |SX | XS
S -> XSX | aY | SX | XS |
X -> b | XSX | aY | a | SX | XS
Y -> b
```

4. Reemplazar las terminales problematicas a con la variable $U \to A$

```
S0 -> XSX | aY | a |SX | XS
S -> XSX | aY | SX | XS |
X -> b | XSX | aY | a | SX | XS
Y -> b
```

```
S0 -> XSX | UY | a |SX | XS
S -> XSX | UY | SX | XS |
X -> b | XSX | UY | a | SX | XS
Y -> b
U -> a
```

Acortar las RHS largas a secuencias RHS' con solo 2 variables cada una:

```
S0 -> XSX | UY | a |SX | XS
S -> XSX | UY | SX | XS |
X -> b | XSX | UY | a | SX | XS
Y -> b
```

```
S0 -> XX1 | UY | a |SX | XS
S -> XX1 | UY | SX | XS |
X -> b | XX1 | UY | a | SX | XS
Y -> b
U -> a
X1 → SX
```

Con esto finaliza la transformación.

3.3 PDAs (Pushdown Automata)

Los PDAs son a los CFL, lo que los DFAs a los lenguajes regulares:

- Esta presentado con un string w sobre un alfabeto Σ.
- Acepta o no acepta w.

Diferencias con los DFA:

- Tienen un stack.
- Pueden ser no deterministas.

Definicion 3.3.1 *Es una estructura de datos (**LIFO**) con tamaño ilimitado que tiene 2 operaciones:*

- **push**: añade un item a la pila
- **pop**: elimina un item de la pila

Propiedades:

- Tiene estados
- Pila con alfabeto Γ
- Transiciones entre estados basadas en:
  - Estado actual
  - Lectura de la entrada
  - Valor obtenido de hacer pop
- Al final de cada transición, se pueden pushear símbolos al stack.

Los CFLs son lenguajes que pueden ser reconocidos por un autómata que tiene una pila:

- $\{0^n 1^n | n \geq 0\}$ es un CFL
- $\{0^n 1^n 0^n | n \geq 0\}$ **no** es un CFL

El alfabeto de pila se denota con Γ. Los símbolos en este alfabeto pueden pushearse y popearse del stack, y hay un símbolo que marca el elemento final del stack Z.

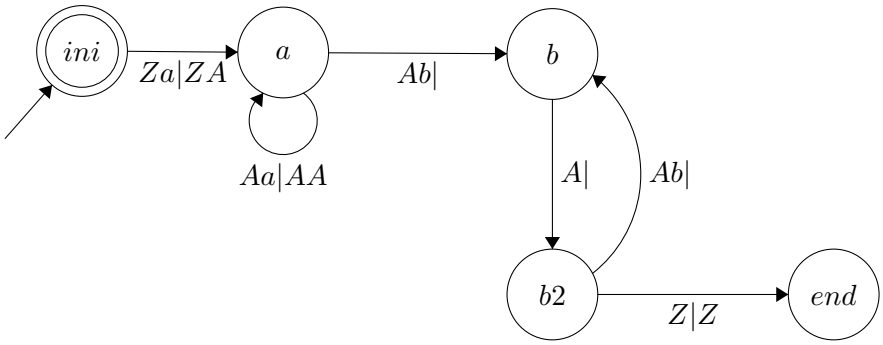
Un PDA es una tupla de la siguiente forma:

$$M = (Q, \Sigma, \Gamma, \delta, q_0, F$$

- Q es un conjunto finito de estados
- Σ es un alfabeto de entrada (finito)
- Γ es un alfabeto de pila (finito)
- q_0 es el estado incial, $q_0 \in Q$

3.3.1 Ejemplo de PDA

$L = \{a^{2n} b^n | n \geq 0\}$



Al inicio se cuenta que el stack esta vacío, indicado con el símbolo Z. Estando el stack vacío (símbolo Z) si entra una a, se pushea ZA. Se han de pushear los 2 símbolos, porque se ha leido Z haciendo un pop. La transición es de la forma $Za|ZA \rightarrow a$. Una vez se esta en el estado a pueden seguirse leyendo más $a's$. Cuando entra una b se salta al estado b, desempilando una A sin empilar nada y a continuación se salta al estado $b2$ sin leer nada, desempilando una segunda a. A partir de aquí se desempilan 2 $a's$ por cada b que entra, hasta que en la pila quede Z, que se pasa al estado end y finaliza.

3.4 Equivalencia entre PDAs y CFGs

Un lenguaje es CFL si y solo si hay un PDA que lo reconozca.

Theorem 3.4.1 *Un lenguaje es context free si y solo si hay un PDA que lo reconozca.*

Lemma 3.4.1 *Si $A = L(G)$ para algun CGF G, entonces $A = L(M)$ para algun PDA M*

Lemma 3.4.2 *Si $A = L(M)$ para algun PDA M, entonces $A = L(G)$ para algún PDA M*

Demostración:

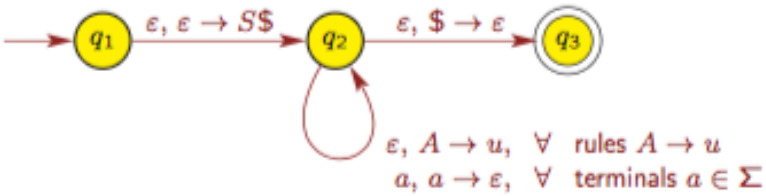
- Dado un CFG G, se convierte a un PDA M con $L(M) = L(G)$
- Idea básica: se crea un PDA que simula la derivación izquierda

Si por ejemplo se considera el $CFGG = (V, \Sigma, R, S$

- Variables V = $\{S, T\}$
- Terminales $\Sigma = \{0, 1\}$
- Reglas $S \rightarrow 0TS1|1T0$, $T \rightarrow 1$

La derivación izquierda del string $011101 \in L(G)$:

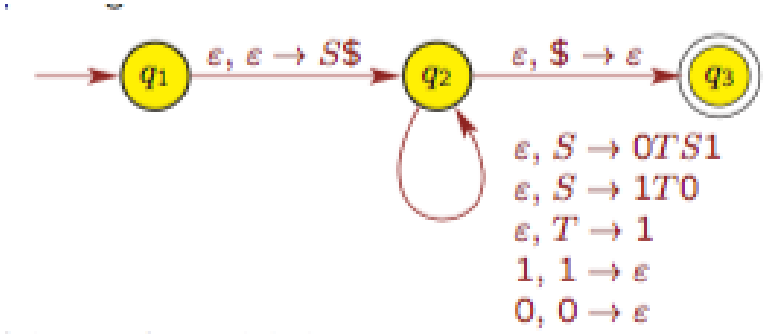
$$S \Rightarrow 0TS1 \Rightarrow 01S1 \Rightarrow 011T01 \Rightarrow 011101$$



Conversión de CFG a PDA

Se comienza con un PDA simple:

- Se pushea \$ y luego S en la pila, donde S es la variable inicial
- Se repite lo siguiente hasta que el stack este vacío
  - Si el top del stack es una variable $A \in V$, entonces reemplazar A por alguna $u \in (\Sigma \cup V)^*$, where $A \to u$ es una regla de derivación en R
  - Si el top del stack es terminal $a \in \Sigma$ y el siguiente símbolo es a, entonces se lee y se hace pop a.
  - Si el top de la pila es \$, entonces se hace pop y se acepta.



Con las reglas que hay del CFG:

```
S -> 0TS1 | 1T0
T -> 1
```

- El PDA es no determinista
- El alfabeto de entrada del PDA es el alfabeto terminal del CFG
  - $\Sigma = \{0, 1\}$
- El PDA simula la derivación izquierda del CFG
  - Pushea la regla RHS en orden inverso a la pila.

Lo siguiente es simular el string 011101 en el PDA. Cuando este en q_2 se mirará el top de la pila para determinar la siguiente transición.

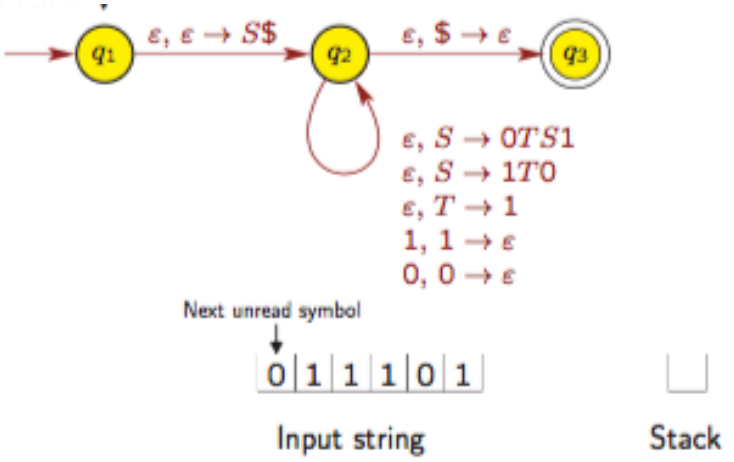


Figure 3.1: Se comienza en el estado q_1 con entrada 011101 y el stack vacío.

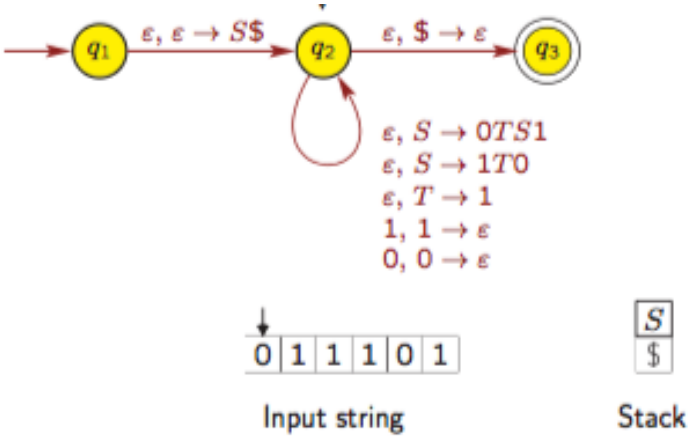


Figure 3.2: No se lee nada, no se hace pop, se salta a q_2 y se pushea \$ y después S.

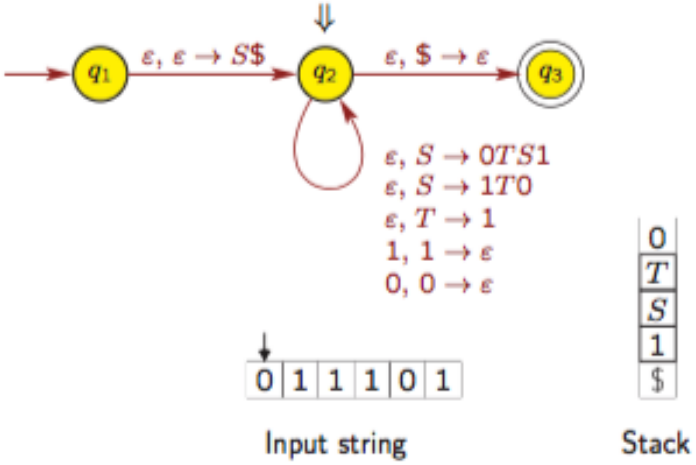


Figure 3.3: No se lee, se hace pop de S, se vuelve a q_2 y se hace push de $0TS1$

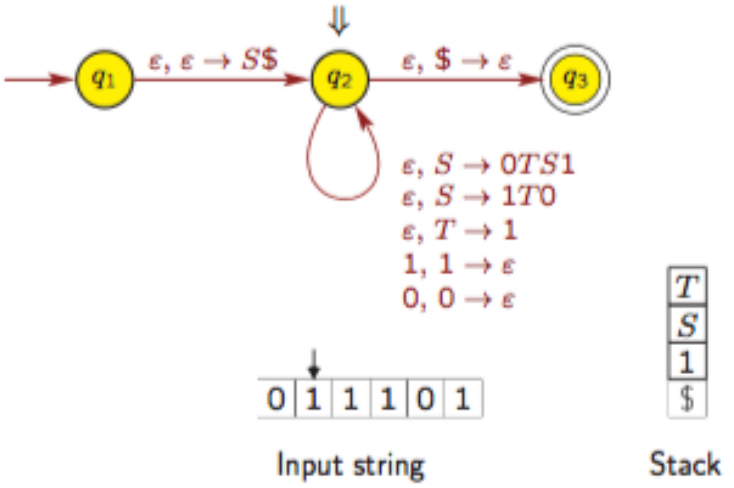


Figure 3.4: Se lee un 0, se hace pop 0, se vuelve a q_2 y no se pushea.

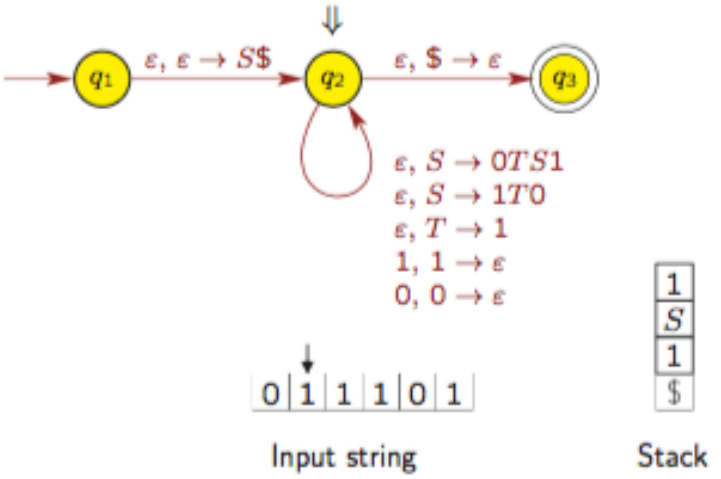


Figure 3.5: No se lee nada, se hace pop T, se vuelve q_2 y se pushea 1.

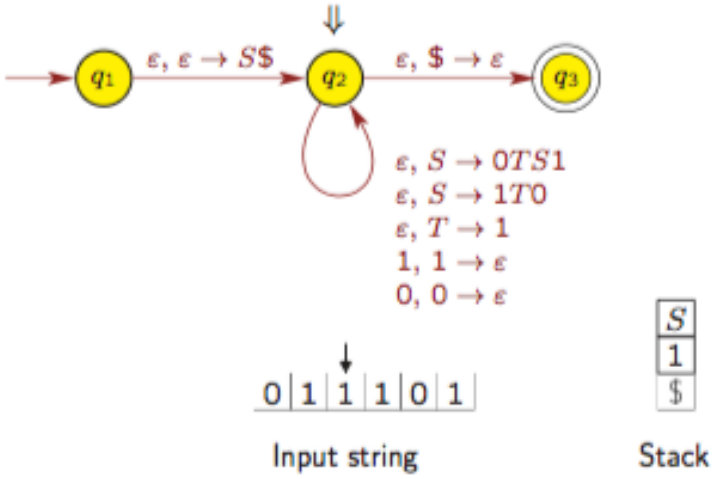


Figure 3.6: Se lee 1, se hace pop de 1, se vuelve a q_2 y no se pushea.

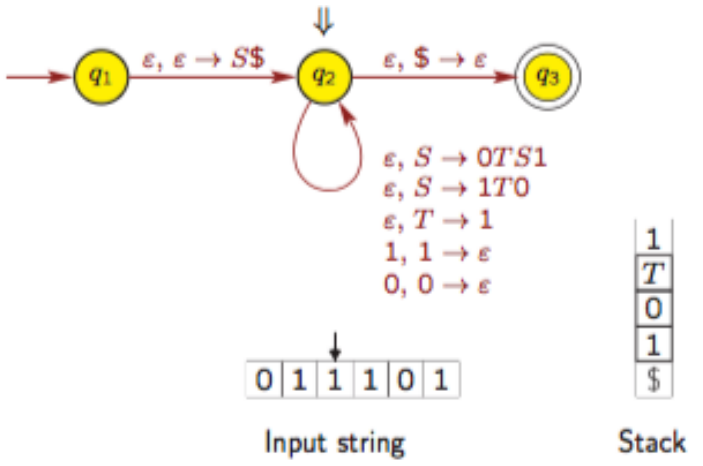


Figure 3.7: No se lee nada, se hace pop de S, se vuelve a q_2 y se pushea 1T0.

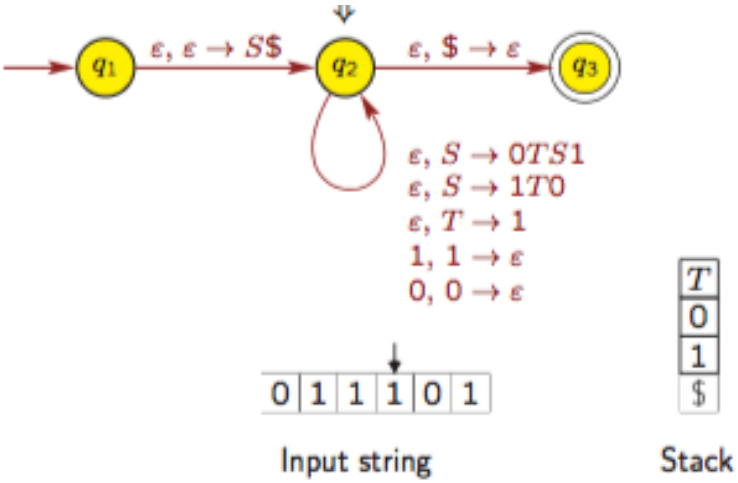


Figure 3.8: Se lee 1, se hace pop de 1, se vuelve a q_2 y no se pushea

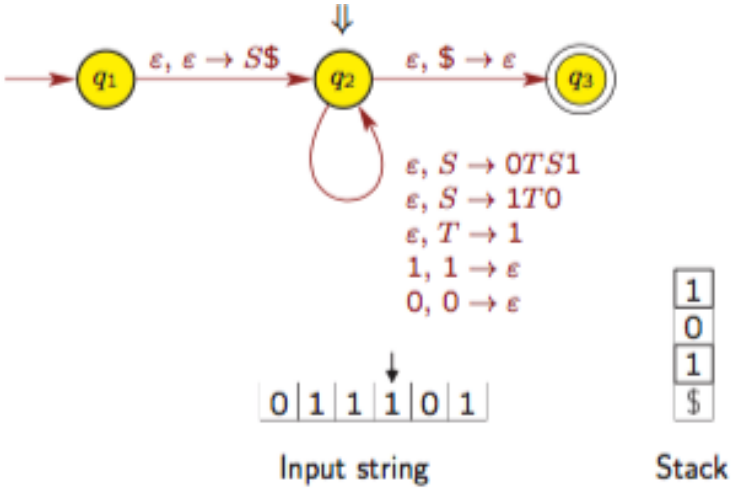


Figure 3.9: No se lee nada, se hace pop de T, se vuelve a q_2 y se pushea 1

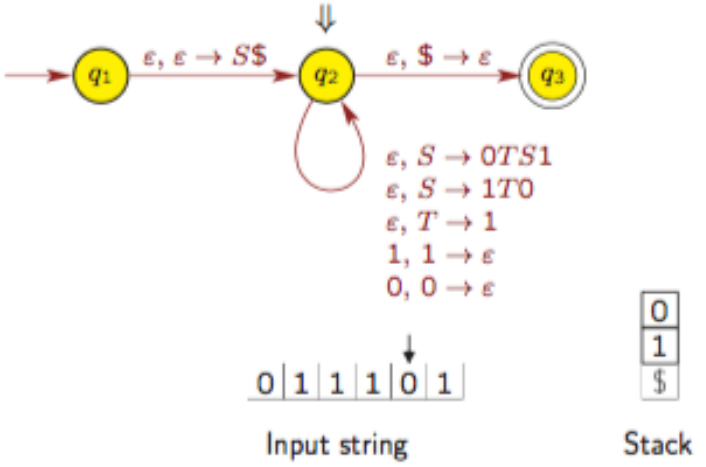


Figure 3.10: Se lee 1, se hace pop de 1, se vuelve a q_2 y no se pushea.

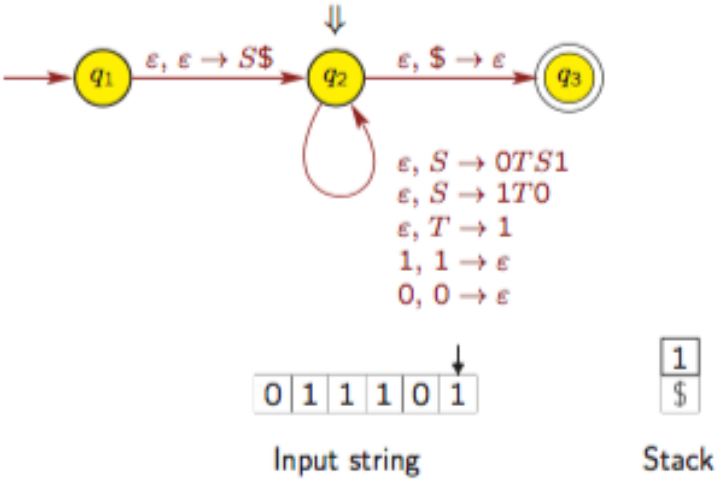


Figure 3.11: Se lee 0, se hace pop de 0, se vuelve a q_2 y no se pushea.

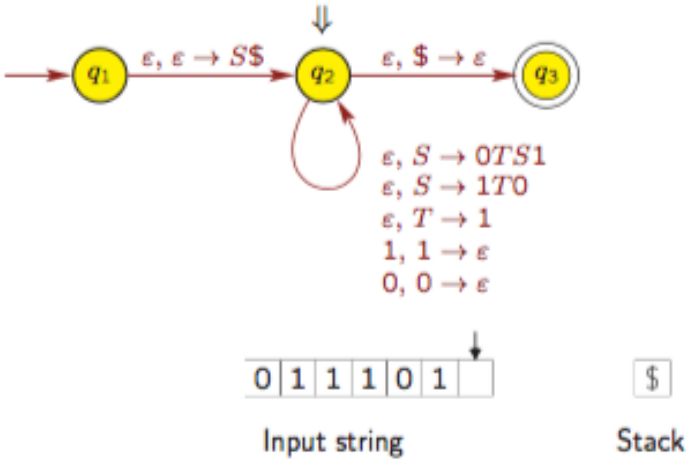


Figure 3.12: Se lee 1, se hace pop de 1, se vuelve a q_2 y no se pushea.

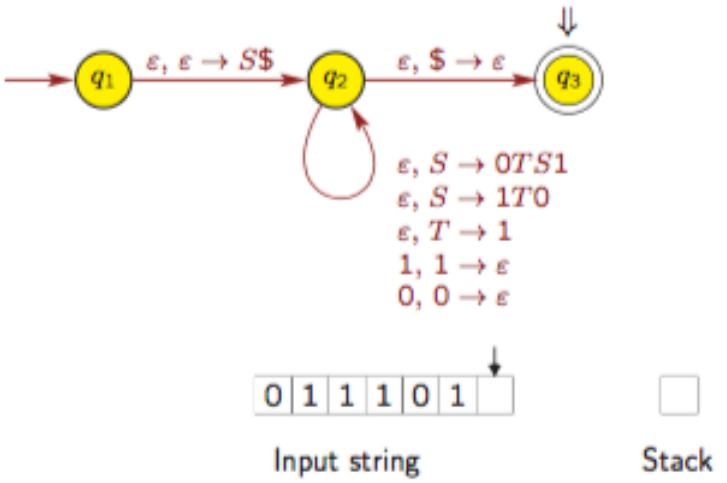


Figure 3.13: No se lee nada, se hace pop de \$, se mueve a q_3 y acepta

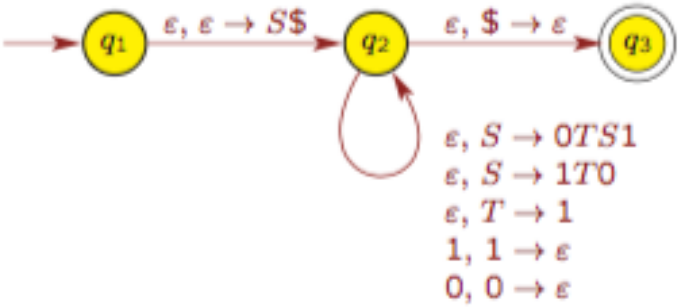
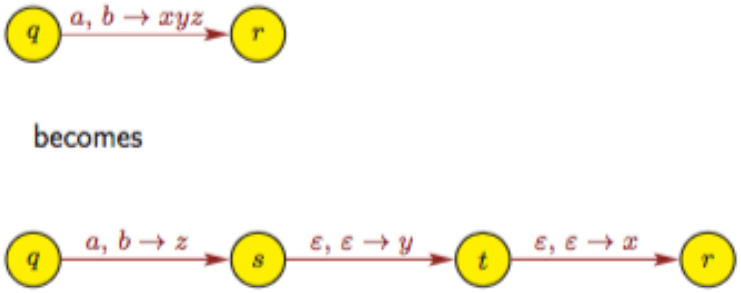


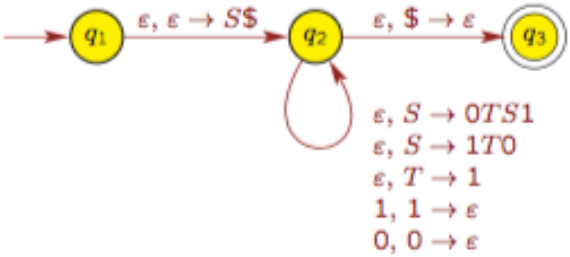
Figure 3.14: PDA resultante, pero no es un PDA final válido.

Ahora hay el problema de que se han pusheado strings al stack, en lugar de símbolos $\{0, 1\}$, lo que no esta permitido en la especificación de los PDAs.

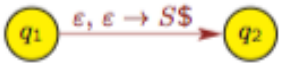
La solución pasa por añadir estados adicionales según se necesite.



El PDA de ejemplo queda:

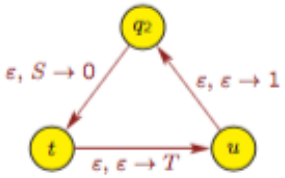


we replace



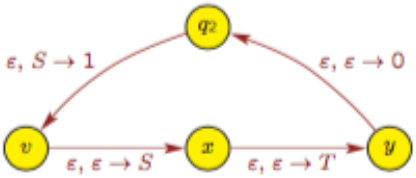
with

with

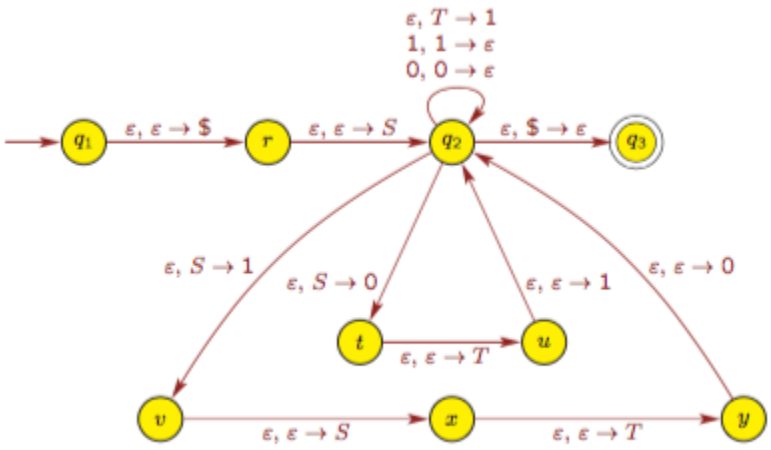


and replace

with



El PDA final queda de la forma:



Regular ⇒ CFL

Corollary 3.4.1.1 *Si A es un lenguaje regular, entonces A también es CFL*

Demostración:

- Se supone que A es regular
- Si A es regular, A tiene un NFA
- Un NFA es simplemente un PDA que ignora el stack.
- Por lo tanto A tiene un PDA
- Por lo tanto, el teorema 3.4.1 implica que A es incontextual.

La conversión contraria no es válida. Por ejemplo $\{0^n1^n|n \geq 0$ es CFL pero no regular.

3.5 Pumping Lemma para CFLs

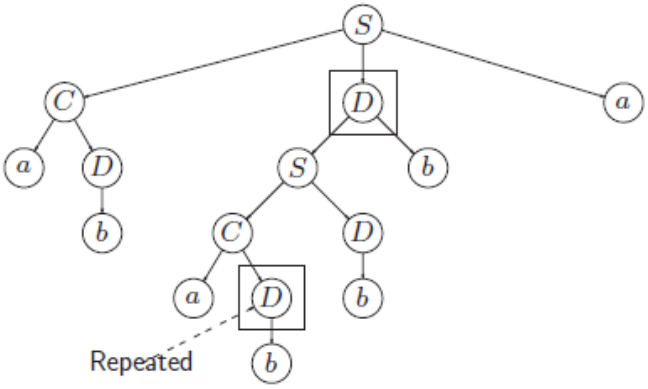
- Anteriormente hemos visto el Pumping Lemma para lenguajes regulares.
- El resultado es análogo para CFLs.
- **Idea básica:** La derivación de la palabra larga $s \in A$ contiene variables repetidas R.
  - El hecho de que la palabra sea larga, implica que tendrá un árbol de derivación largo, probablemente con variables repetidas.
  - Podemos dividir la palabra $s \in A$ en **cinco partes** $s = uvxyz$ basado en R.
  - $uv^ixy^iz \in A$ para todo $i \geq 0$.
- Consideramoe el lenguage A con CFG G

```
S -> CDa | CD
C -> aD
D -> Sb | b
```

- Aquí tenemos la derivación de G que repite las variables $R = D$:

$$S \Rightarrow CDa \Rightarrow aDDa \Rightarrow ab\underline{D}A \Rightarrow abSba \Rightarrow abCDba \Rightarrow aba\underline{D}Dba \Rightarrow ababDba \Rightarrow ababbba$$

- El árbol repite la variable D desde la raíz a la hoja.



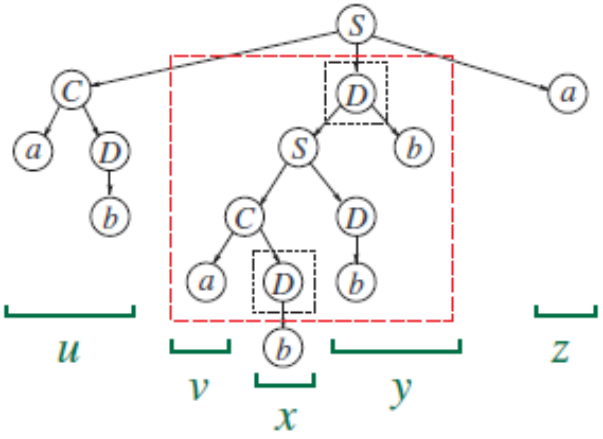
Dividiendo la palabra en 5 partes

- Dividimos la palabra $s \in A$ en

$$s = \underbrace{ab}_{\text{u}}\underbrace{a}_{\text{v}}\underbrace{b}_{\text{x}}\underbrace{bb}_{\text{y}}\underbrace{a}_{\text{z}}.$$

  usando la variable repetida D.

- Si leemos en profundidad primero
  - $u = ab$ está antes que la estructura $D - D$ del subárbol.
  - $v = a$ está antes que la segunda D en el subárbol $D - D$.
  - $x = b$ es en lo que el segundo D se convierte.
  - $y = bb$ está después que la segunda D en el subárbol $D - D$.
  - $z = a$ está después del subárbol $D - D$.



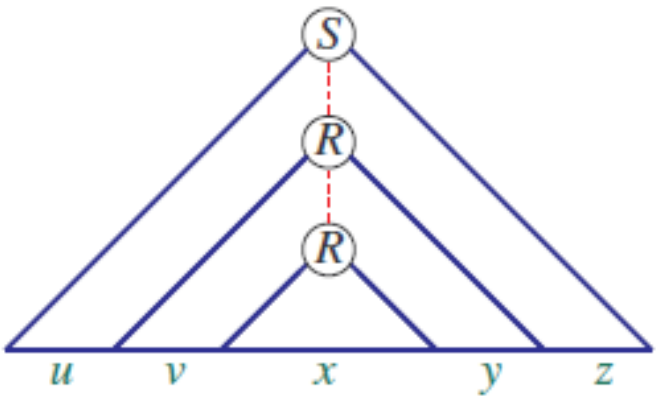
¿Cuándo es posible usar el Pumping Lemma?

- **La clave para hacer pumping**: variable R repetida en el árbol de derivación.
  - $S \overset{*}{\Rightarrow} uRz$ para $u, z \in \Sigma^*$
  - $R \overset{*}{\Rightarrow} vRy$ para $v, y \in \Sigma^*$
  - $R \overset{*}{\Rightarrow} x$ para $x \in \Sigma^*$
  - palabra $s = uvxyz \in A$
- Variable R repetida $R \overset{*}{\Rightarrow} vRy$, por lo que el pumping "$v - y$" es posible:

$$S \overset{*}{\Rightarrow} uRz \overset{*}{\Rightarrow} uvRyz \overset{*}{\Rightarrow} uv^iRy^iz \overset{*}{\Rightarrow} uv^ixy^iz \in A$$

- Si el árbol es **suficientemente alto**, entonces la variable repetida estará en el camino desde la raíz a la hoja.
  - ¿Cómo de alto tiene que ser el árbol para asegurarse el poder hacer pumping?
  - La **longitud** del camino entre dos nodos = al camino de transiciones entre ambos.
  - La **altura** és el camino más largo entre la raíz y una hoja.

Podemos hacer Pump si el árbol es suficientemente alto



- Camino desde la raíz S a una hoja
  - La hoja es un terminal $\in \Sigma$
  - Todos los otros nodos del camino son variables $\in V$.
- Si la altura del del árbol $\geq |V| + 1$, donde $|V|$ = número de variables del CFG

– entonces ∃ alguna variable repetida en el camino más largo a una hoja.

Pumping Lemma para CFG

Theorem 3.5.1 ***Si*** *A es CFL,* ***entonces*** ∃ *una longitud de pumping p donde,* ***si*** $s \in A$ *con* $|s| \geq p$*,* ***entonces*** *s puede dividirse en 5 partes.*

$$s = uvxyz$$

satisfaciendo las condiciones

1. $uv^ixy^iz \in A$ *para cada* $i \geq 0$,
2. $|vy| > 0$, *y*
3. $|vxy| \leq p$.

Observaciones:

- La condición 1 implica que $uxz \in A$ al coger $i = 0$.
- La condición 2 obliga a que vy no sea vacío.
- La condición 3 a veces es útil.

No CFL

Observación: El Pumping Lema (PL) para CFL se usa principalmente para demostrar que algunos lenguajes **no** son CFL.

Ejemplo: Demuestra que $B = \{a^nb^nc^n \mid n \geq 0\}$ no es CFL. **Demostración.**

- Suponemos que B es CFL, el PL implica que B tiene longitud de pumping $p \geq 1$.
- Consideramos la palabra $s = a^pb^pc^p \in B$, por tanto $|s| = 3p \geq p$.
- PL: s se divide en 5 partes siguiendo las condiciones anteriormente mencionadas.
- Por contradicción vemos que **no se puede** dividir satisfaciendo las condiciones 1-3.
  - Vemos que **todas** las palabras que cumplen la condición 2, violan la 1.
- Recordemos que $s = uvxyz = \underbrace{aa...a}_{\text{p}}\underbrace{bb...b}_{\text{p}}\underbrace{cc...c}_{\text{p}}$.
- Posibles divisiones para satisfacer la condición 2: $|vy| > 0$
  - Las palabras v e y son uniformes. Por ejemplo, $v = a...a$ e $y = b...b$.

        * Entonces uv^2xy^2z no tendrá el mismo número de a's, b's y c's porque $|vy| > 0$.
        * Por tanto, $uv^2xy^2z \notin B$.
    - Las palabras v e y no son ambas uniformes.
        * Entonces $uv^2xy^2z \notin L(a^*b^*c^*)$: símbolos no agrupados juntos.
        * Por tanto, $uv^2xy^2z \notin B$.
- Ninguna división que permite la condición 2, permite también la 1.
- **Contradicción**, $B = \{a^n b^n c^n \mid n \geq 0\}$ no es CFL.

Chapter 4

Máquinas de Turing

4.1 Definición

Son un lenguaje de programación de muy bajo nivel. Tienen tanta expresividad como cualquier lenguaje de programación.

Debido a que son demasiado básicas, no facilitan programar, pero sirven para ver que algo no se puede resolver con ellas.

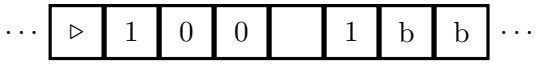
Nuestras máquinas tienen memoria finita. Por tanto son autómatas. Pero: una máquina con 1Gbyte de memoria tiene más 2^{233} estados posibles, cosa que es mucho más grande que el numero de átomos del universo: $2 * 10^{77}$.

Además, no es razonable afirmar que no podemos reconocer $\{a^n b^n\}$. Concluiriamos que este lenguaje no es decidible, pero todo informático dirá que es capaz de escribir un informático que compruebe si tenemos tantas a's como b's en una entrada.

Por lo anterior, y por el hecho de que con el tiempo la memoria disponible en las máquinas que usamos aumenta:

$$\text{Algoritmo} \equiv \text{control finito} + \text{memoria arbitraria (infinita)}$$

Una máquina de turing tiene un grafo de estados y transiciones, al igual que los DFA, pero además tienen lo que se llama **cinta**, que es una palabra con infinitas posiciones.



Para que el autómata maneje la cinta, en todo momento hay un apuntador a una de las posiciones de la cinta llamado **cabezal**.

Las transiciones del autómata tienen una condición para su ejecución:

- El símbolo guardado en la posición apuntada por el cabezal, ha de ser uno concreto.

Cada transición tendrá una acción indicando que símbolo habrá que escribir en la posición apuntada por el cabezal y si queremos mover el cabezal a izquierda, derecha o dejarlo quieto.

La máquina se encuentra al principio en el estado inicial. El cabezal apunta a la posición 1 y en la cinta

se haya la palabra de entrada de la máquina, empezando por la posición 1.

En la posición 0 se haya un símbolo especial que no forma del alfabeto de entrada y que llamaremos **símbolo de marca de inicio** $\triangleright$.

- Si lo vemos sabemos que estamos lo más a la izquierda posible de la entrada.

Después del último símbolo de la entrada aparece un símbolo llamado símbolo blanco, a partir del cual todos los símbolos que aparecen son blancos.

La máquina ejecutara transiciones cuando se cumplan sus condiciones. Como consecuencia:

- El cabezal se va desplazando.
- Se va modificando el contenido de la cinta.
- Los símbolos que se escriban no tienen porque ser del propio alfabeto de entrada.
- La palabra de entrada será aceptada si llegamos al único estado aceptador que tendrá la máquina.

Ejemplo

$\{w\#w|w \in \{0,1\}^*\}$

En el caso de que la palabra de entrada sea del lenguaje la cinta tendrá este aspecto y el cabezal apuntará al primer símbolo:

$\triangleright$ 011...#011...bb...

Lo que se hará será recordar mediante los estados, que símbolo se encuentra al principio, nos moveremos hasta el principio de la segunda parte y veremos si también se haya ese mismo símbolo al principio, y así sucesivamente con los siguientes símbolos.

Pero para que la máquina pueda saber que parte de las palabras ha sido ya comprobada, se irán substituyendo los símbolos ya tratados, por símbolos nuevos a modo de marca.

Si se lee el primer 0, se marca y se mueve hacia la derecha hasta saltarse el separador, recordando que hemos visto un 0. $\triangleright$ 0'11...#011...bb...

Cuando se ha comprobado que también hay un 0 en la segunda parte, se marca y se salta a la primera parte, saltandonos el separador, hasta encontrar nuevamente la marca inicial.

Ahora se lee un 1, se marca y se ve a la derecha saltando el separador y las marcas que encontremos, recordando que hemos visto un 1. Al comprobar que hay un 1, lo marcamos y volvemos atrás.

Así sucesivamente hasta comprobar que ambas palabras son iguales.

Hilo de ejecución de un recorrido:

$\triangleright q_0 011\#011 \rightarrow$ – δ debajo de la flecha

$\triangleright 0' q_{<0>} 11\#011 \rightarrow$

$\triangleright 0'1 q_{<0>} 1\#011 \rightarrow$

$\triangleright 0'11 q_{<0>} \#011 \rightarrow$

$\triangleright 0'11\#q_{0'}011 \rightarrow$

$\triangleright 0'11q_{i*}\#0'11 \rightarrow$

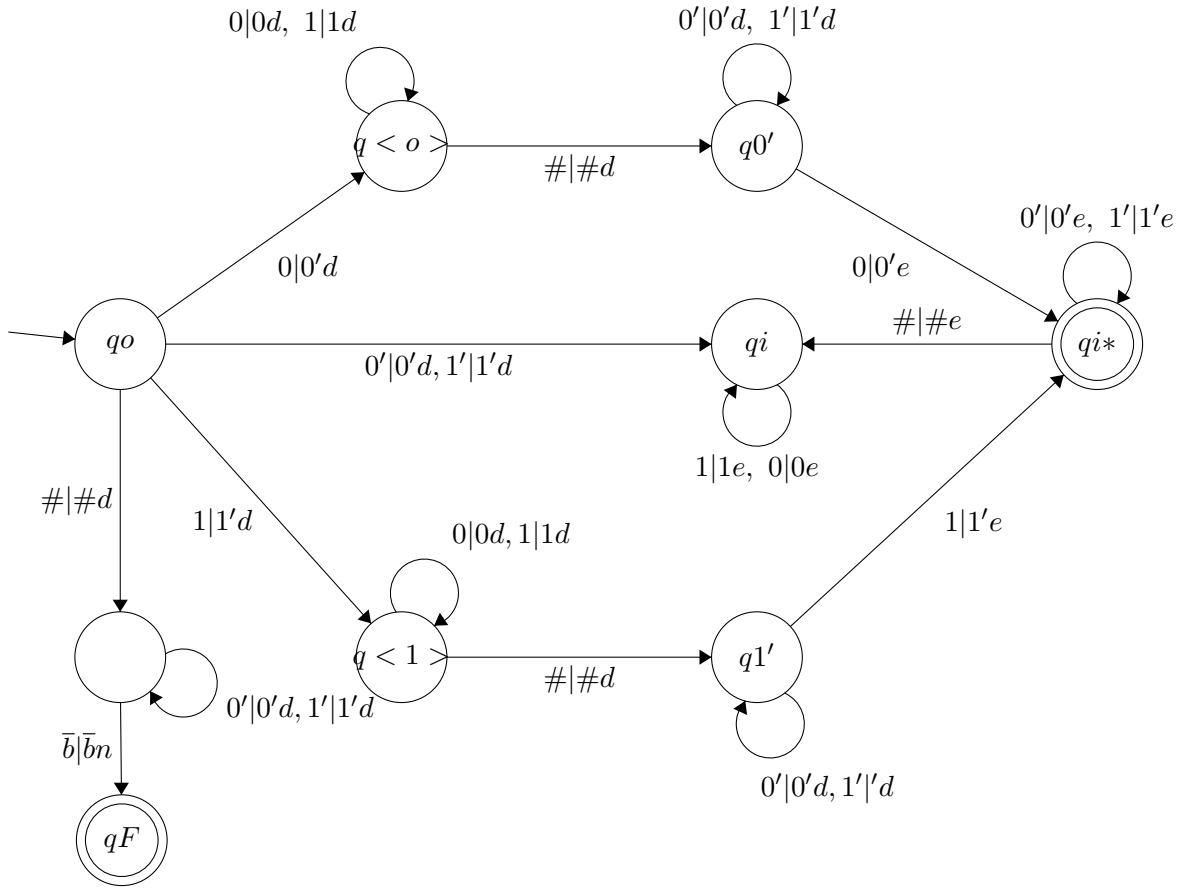
$\triangleright 0'1q_i1\#0'11 \rightarrow$

$\triangleright 0'q_i11\#0'11 \rightarrow$

$\triangleright q_i0'11\#0'11 \rightarrow$

$\triangleright 0'q_011\#0'11 \rightarrow$

La representación de la máquina de Turing es la siguiente:



Del estado inicial pasamos a q¡0¿ si leemos un 0, entonces vamos leyendo símbolos hasta encontrar el separador, y después del separador, se van saltando símbolos marcados q0' y al leer un 0 se salta a qi'. En qi' volvemos hacia atrás hasta el separador, y del separador volvemos al primer símbolo marcado que veamos al retroceder, volviendo a q0.

qending comprueba que todos los estados a la derecha del separador esten marcados.

4.1.1 Definición de una TM

Una TM se representa como la siguiente tupla:

$$M =< Q, \Sigma, \Gamma, \delta, q_0, q_F$$

Donde:

- Q: conjunto de estados
- Σ: alfabeto de entrada
- Γ: alfabeto de cinta
- δ: función de transición
- q_0, q_F: estado inicial y final

El alfabeto de cinta contiene el alfabeto de entrada y los símbolos de inicio y parada:

$$\Sigma \uplus \{\triangleright, \overline{b}\} \subseteq \Gamma$$

El alfabeto de cinta puede contener más símbolos que todos estos.

La función de transición: $\delta : Q - -\{q_F\} \times \Gamma \to Q \times \Gamma \times \{i, d, n\}$

Para cada estado que no sea el aceptador y cada símbolo de cinta, nos dice:

- A que estado vamos a parar Q
- Que nuevo simbolo se escribe en la cinta Γ
- Hacia donde se desplaza $\{i, d, n\}$

La función de transición es parcial, puede no estar definida para todos los estados y símbolos posibles.

También se puede considerar el caso no determinista, en el que la función de transición ya no sería necesariamente una función, si no un subconjunto del producto cartesiano de todos estos conjuntos.

Una **configuración** de una TM es un estado global en el que se puede encontrar en algún momento. Define no solo estado, si no al contenido de la cinta y la posición del cabezal.

El contenido de la cinta es infinito, pero solo un número finito de sus posiciones contendrá símbolos que no sean blanco, ya que en un tiempo finito de ejecución solo se puede modificar un número finito de símbolos.

- Para describir el **contenido** basta con una **palabra finita**

$$w_1 q w_2 \equiv w_1 q w_2 \overline{b} \equiv w_1 q w_2 \overline{b}\overline{b}$$

- w1 es el contenido de la cinta antes del cabezal
- q es el estado en el que se encuentra la máquina
- w2 es el contenido de la cinta, desde el cabezal en adelante.
  - el primer símbolo de w2 es el apuntado por el cabezal

- Más allá de w2 hay blancos
    - Por ello las configuraciones anteriores son equivalentas

La función de transición también se puede representar con un sistema de escritura, con reglas que se aplican sobre configuraciones.

una transición de desplazamiento a la derecha se puede representar con esta regla:

Una transición de desplazamiento a la derecha se puede representar con la siguiente regla:

$$\text{Cada } \delta(q,a) = (q',b,d) \text{ lo representamos con } qa \to bq'$$

La regla se aplica sobre una configuración

- Si el estado es q
- Si símbolo apuntado por el cabezal es a.

Como resultado:

- Reemplaza localmente el símbolo a por b
- Mueve el cabezal hacia la derecha
- Cambia el estado a q'

$$\text{Cada } \delta(q,a) = (q',b,n) \text{ lo representamos con } qa \to q'b$$

$$\text{Cada } \delta(q,a) = (q',b,d) \text{ lo representamos con } a_1qa \to q'a_1b, ..., a_nqa \to q'a_nb$$

Si desplazamos a la izquierda la cosa se complica ligeramente, porque la regla necesita saber que símbolo se encuentra a la izquierda del cabezal. Por ese motivo transiciones de este estilo se representan mediante tantas reglas cómo símbolos tiene el alfabeto de cinta. $a_1...a_n$ son los símbolos del alfabeto de cinta.

4.2 Lenguaje reconocido y función computada

Utilizamos el **estado aceptador** para determinar cuando una TM acepta una entrada w.

Definición: el conjunto de palabras que nos llevan a estado aceptador

$$L(M) = \{w \in \Sigma^* | \exists w_1, w_2 \in \Sigma^*, n \in \mathbb{N} : \triangleright w\overline{b}^n \to_\delta^* w_1qFw_2\}$$

Definimos la función computada por una TM del siguiente modo:

Un elemento x tiene como imagen y, si teniendo x como entrada, la máquina llega a estado aceptador dejando y en la cinta. Si x e y se forman sobre el alfabeto de entrada, y además tras añadir el símbolo

de inicio de cinta a la izquierda de x y suficientes blancos a su derecha, accedemos a una configuración con estado aceptador y que contenga y entre el primer y el segundo símbolo fuera del alfabeto de entrada, entonces decimos que y es la imagen de x por la función computada por la máquina.

$$\varphi M(x) = y \equiv x, y \in \Sigma^* \wedge \exists \alpha_1, \alpha_2 \in (\Gamma - \Sigma), w^{'} \in \Gamma^*, n \in \mathbb{N}$$

$$\triangleright x\bar{b}^n \rightarrow_\delta^* \rightarrow_{qF} \rightarrow_\lambda \alpha_1 y \alpha_2 w'$$

El dominio de la función computada por la máquina M es el lenguaje reconocido por M:

$$Dom(\varphi M = L(M)$$

De este modo denotamos que la imagen de x por φ_n esta definida, es decir que existe una y que es su imagen

$$\varphi_M(x) \downarrow \equiv \exists y : \varphi_M(x) = y \equiv x \in L(M)$$

Un mote $w \in \Sigma^*$ es aceptado por una TM M si M con entrada w se detiene en estado aceptador.

Decimos que **una configuración es terminal** si no se le puede aplicar ninguna regla:

$$\nexists w' \in \Gamma^* : w\bar{b} \rightarrow_\delta w'$$

Si para una entrada x, la imagen de x esta definida, entonces la máquina para con entrada x:

La implicación contraria no tiene porque ser cierta, puede haber configuraciones terminales que no sean aceptadoras. Simplemente puede ocurrir que no se pueda aplicar ninguna regla por no haber más transiciones definidas.

En el caso en que $\varphi_M(x)$ este definida, con M(x) también denotamos $\varphi(x)$

Definición: un lenguaje reconocido o aceptado por una TM M, representado por L(M), esta formado por el conjunto de motes aceptados por M.

- Dado un lenguaje L, decimos que M reconoce o acepta L si $L(M) = L$

Cuando una TM se detiene con una entrada, podemos extraer información de lo que ha quedado escrito en la cinta y asociarlo como salida.

- Esto nos permite introducir los conceptos de **función computada** y **función computable**

Función computada: representada por f_M se define así:

$f_M(w)$ es $M(w)$ si $M(w) \downarrow$ y esta indefinida en caso contrario. Dada una función

$$f : \Sigma^* \rightarrow \Sigma^*$$

Decimos que M computa f si $f = f_M$.

Una **función** es **computable** si **hay una TM** M para la cual $f = f_M$

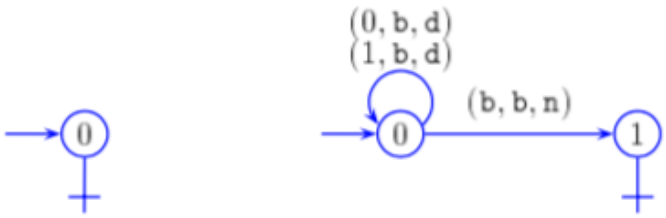


Fig. 1.3: TM *que reconeixen* Σ^*

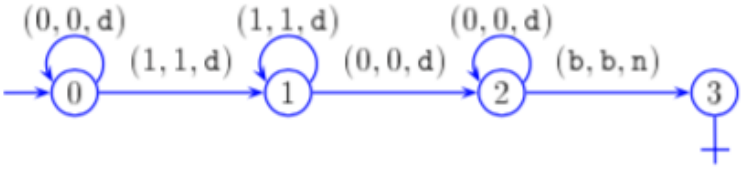


Fig. 1.4: TM *que reconeix* $0^*1^+0^+$

La primera computa la función identidad. La segunda computa la función constante $f(w) = \lambda$ para todo $w \in \Sigma^*$

Lenguaje decidible

$$(L \in Dec) \equiv existeTMMtalqueL(M) = LyMparacontodaentrada$$

Lenguaje semi-decidible (recursivamente numerable)

$$(L \in semi--Dec) \equiv existeTMMtalqueL(M) = LyMparacontodaentrada$$

Decimos que la máquina semidecide el lenguaje. Así pues **semi-decidir** y **reconocer** son sinónimos en este contexto. Esa máquina puede no parar con aquellas entradas que no sean del lenguaje.

Función computable

Si existe una máquina que la computa:

$$\exists M \in TM : \varphi_M = f$$

En tal caso, decimos que esa máquina es una implementación de la función o que computa la función.

4.3 Máquinas de Turing con parada segura. Tiempo de cálculo

4.4 Algunos porblemas sobre las máquinas de Turing

4.4.1 La máquina de Turing multicinta

4.4.2 La máquina de Turing indeterminista

4.5 Apuntes tomados en clase

4.6 Lenguaje reconocido por M

$L(M) = \{w \in \Sigma^* | \exists w1, w2 \in \Gamma^*, n \geq 0 \triangleright q_0 w\bar{b} \xrightarrow[\delta]{*} w_1 q_F w2\}$

$M(w) \downarrow$ M con entrada w se detiene

$M(w) \uparrow$ M con entrada w no se detiene

L es decidible: existe una TM M de parada segura tal que $L = L(M)$

L es semidecidible:

4.7 Tesis de Church Turing

Todo lo que es computable (algoritmo) es computable por una TM

TM $\equiv$ programas escritos LP: todo programa escrito en un lenguaje de programación (ej asm) se puede simular en una máquina de turing.

TM $\equiv$ K-TM (k cintes): toda maquina de turing con k cintas se puede simular en una máquina de turing en la que hacemos particiones

TM $\equiv$ RAM

1. TM reconoce lenguajes

2. TM computa funciones

M $= < Q, \Sigma, \Gamma, \delta, q_0, q_F >$

Función computada por M: φ_M

$\varphi_M = y \equiv .x, y, \in \Sigma^* \wedge \exists \alpha_1, \alpha_2 \in (\Gamma - \Sigma), w' \in \Gamma^*, n \geq 0 \triangleright q_0 x\bar{b} \rightarrow \alpha_1 y \alpha_2 w' \delta \cup \{q \rightarrow \lambda\}$

4.7.1 Computabilidad

f: $\Sigma^* \rightarrow \Sigma^*$ es computable si $f = \varphi_M$

$$\forall x \in dom(f) \Leftrightarrow M(x) \downarrow enq_F$$

$$x \in dom(f) \Rightarrow f(x) = M(x)$$

f es computable si existe una TM M tal que $\varphi_M = f$

$f : A \to B$

$*dom(f) \subseteq A$

$x \in dom(f) \Rightarrow \exists y \in B \ f(x) = y$

$dom(f) = A$ f es TOTAL

$f : \Sigma^* \to \Sigma*$

$\mathbb{N} \to \mathbb{N}$

$dom(f) = w \in \Sigma^* | valor_2(w) \in \dot{2}$

f es comoutable $\exists \varphi_M = f$

Chapter 5

Laboratorio TM

5.1 Indecibilidad

Todas las entradas son $\mathbb{N}$ ($o\Sigma^*$)

- Entrada w
- Entrada $< x, y >$

El conjunt de totes les TM també es poden codificar per $\mathbb{N}$

$$M =< Q, S, \Gamma, \delta, q_0, q_F >$$

$$M_n \text{ es la TM } < M >= n$$

$$M = M_n \Leftrightarrow < M >= n$$

Conjunt de tots els llenguatges

$$\{L|L \subseteq \Sigma^*\} = \delta(gma^*) = 2^{\Sigma^*}$$

$$\{L|L \subseteq \mathbb{N}\} = \delta(\mathbb{N}) = 2^{\mathbb{N}}$$

Hay más lenguajes que máquinas

5.2 Interprete de TM

La máquina de Turing universal. Es la máquina que con la entrada $< x, y >$ devuelve lo mismo que retornaría la máquina x con entrada y :

$$(M_x(y)) \text{ si } M_x(y) \downarrow$$

$$(< x, y >) \uparrow \text{ si } M_x(y) \uparrow$$

- Executa $M_x(y)$
-

$L \in Dec \Leftrightarrow \exists n L = L(M_n))$ i M_n es d'aturada segura

5.2.1 Propietats

- Dec
- Semidec

Son tancats respecte de $\cup, \cap, *$

Dec: es tancada respecte complementari

$$L \in Dec \Leftrightarrow \overline{L} \in Dec$$

5.3 Teorema del complement

5.4 Problema de l'autoaturada

Máquina que su entrada es ella misma.

$$K = \{x | M_x(x) \downarrow\}$$

M: entrada x executar $M_x(x)$; acceptar;

$x \in K \Rightarrow M(x) \downarrow$ i accepta $\Rightarrow x \in L(M)$

$x \notin K \Rightarrow M(x) \uparrow \Rightarrow X \notin L(M)$

Per tant $L(M) = K$

$K \notin Dec$

Suponemos que $K \in Dec$ Entonces $\exists M_k$

M de parada segura tal que $L(M_n) = k$

Definimos M:

- entrada x {
- si $M_k(x)$ aceptase entonces se cuelga
- si $M_k(x)$ rechaza entonces acepta

Sea y tal que $M_y = M$

$$y|inK \xrightarrow{defk} M_y(y)\downarrow \xrightarrow{defM_y} \text{rechaza} \Rightarrow y \notin K$$

$$y \notin K \xrightarrow{defk} M_y(y)\uparrow \xrightarrow{defM_y} acepta \Rightarrow y \in K$$

Lo anterior nos lleva a una **contradicción**

5.4.1 Semidecibilidad

$K \in SemiDecK \notin Dec$ Aplicando el teorema del complemento

$\overline{K} \notin SemiDec$

Problema de la parada

$H = \{< x, y > |M_x(y)\downarrow\}$

$x \in K \Leftrightarrow < x, x > \in H$

$x \xrightarrow{f} < x, x >$

$x \in K \Leftrightarrow < x, x > \in H$

Si $H \in Dec$, existe M_h TM de parada segura

$L(M_H) = H$ definimos M:

- entrada x
- ejecuta $M_n(< x, x >)$

$L(M) = K$ y M es de parada segura $\rightarrow M \in Dec$ **CONTRADICCIÖN**

5.5 Reducción

$C_1, C_2 \subseteq \Sigma^*$

$C_1 \leq C_2$ C1 se reduce a C2

Cuando existe $f : \Sigma^* \rightarrow \Sigma^*$ total y computable

Pora todo x, $x \in C_1 \Leftrightarrow f(x) \in C_2$

Proposición

- $C_1 \leq C_2 y C_2 \in Dec$ entonces $C_1 \in Dec$

Contrareciproco

Chapter 6

Máquinas de Turing y algoritmos

6.1 Esquemas de los algorítmos básicos

6.2 Una representación para las máquinas de turing

6.3 Interpretes y simuladores

6.4 La tesis de Church-Turing

Chapter 7

Computabilidad de funciones y decibilidad de lenguajes

7.1 Computabilidad de funciones

7.2 Decibilidad de lenguajes

Cuando queramos demostrar que un lenguaje es decidible bastará con mostrar un programa o algoritmo que decide L:

- Para siempre.
- Si la entrada es de L acepta.
- Si la entrada no es de L rechaza.

Un programa que semidecide L, es un pograma que:

- Si la entrada es de L **para y acepta**
- Si no es de L tanto
  - Puede parar y rechazar
- Puede quedarse colgado
- Es equivalente a decir que es reconocible por TM.

Para demostrar que un lenguaje no es decidible hay que hacer una reducción desde el lenguaje K a L

$$K \leq L$$

$$\left\{ \begin{smallmatrix} K= \\ x|M_x(x)\downarrow \end{smallmatrix} \right\}$$

Para demostrar que un programa no es semi-decidible basta con una reducción de K complementario a L

$$\overline{K} \leq L$$

$$\overline{K} = \{x | M_x(x) \uparrow\}$$

7.2.1 No decibilidad

Para empezar se utilizará el siguiente conjunto:

$$K = \{x | M_x(x) \downarrow\} \text{ no es decidible}$$

Es el conjunto de programas que apran con entrada ellos mismos. Con más precisión, es el conjunto de naturales tales que la máquina que codifican ejecutada con entrada el propio natural, termina.

Es un conjunto un tanto antiintuitivo, sin embargo es el que resulta más fácil demostrar que no es decidible.

Una vez demostrado que K no es decidible, podremos demostras que otros conjuntos no lo son.

Demostración

La demostración se hará por reducción al absurdo.

Suponemos que K es decidible. Entonces existe una máquina M_k que decide K, es decir, una máquina que para con cualquier entrada y acepta aquellas que son de K.

Usando M_k construimos una nueva máquina, a cuya codificación como número natural llamaremos n.

Entrada x

- Si $M_k(x)$ acepta, entonces se cuelga
- Si $M_k(x)$ rechaza entonces para.
  - Tanto si ponemos aquí aceptar como rechazar la demostración funcionara

Si ejecutamos la máquina con el natural n tenemos 2 opciones:

- $M_n(n) \downarrow$
- $M_n(n) \uparrow$

O bien para o bien no. Si para con entrada el mismo, esto implica:

$$M_n(n) \downarrow \Rightarrow n \in K \Rightarrow M_k(n) acepta \Rightarrow M_n(n) \uparrow$$

Entonces la condición de la primera rama del condicional se cumple, por lo que la máquina se cuelga con entrada n.

Supongamos ahora que la máquina con entrada su propia codificación se cuelga:

$$M_n(n) \uparrow \Rightarrow n \notin K \Rightarrow M_k(n) rechaza \Rightarrow M_n(n) \downarrow$$

$M_k(n)$ rechaza ya que M_k decide k. Entonces esta máquina con entrada n, solo cumple la segunda rama del condicional y la máquina parará. $M_n(n)$ se para y se contradice.

Por tanto:

$$(\neg M_n(n) \downarrow \wedge \neg M_n(n) \uparrow)$$

Lo que es una contradicción.

7.2.2 Demostraciones a partir de K

Ejemplo 1

Vamos a aprovechar el hecho de que K no es decidible, para demostrar que otros conjuntos tampoco lo son.

Tenemos el siguiente conjunto, que representa el conjunto de parejas de programa x entrada y, que el programa para con la entrada:

$$H = \{\langle x, y\rangle | M_x(y) \downarrow\} \text{ no es decidible}$$

Más concretamente, es el conjunto de naturales que codifican parejas de naturales (x,y), tales que la máquina codificada con x para con entrada y.

Supongamos que el conjunto es decidible, entonces hay una máquina que lo decide.

Llegamos a contradicción usandola para construir una máquina que decide K, que ya hemos visto que no es decidible.

Entrada x:

$$\text{Salida } M_H(\langle x, x\rangle)$$

Si x es de k, entonces M_x para con entrada x, de modo que M_H acepta.

Si x no es de k, entonces M_x no para con entrda x, de modo que M_H rechaza.

Así pues, esta máquina decide K y termina la demostración.

Ejemplo 2

$H' = \{x | \exists y : M_x(y) \downarrow\}$ no es decidible

Es el conjunto de los programas que paran con alguna entrada. Con más precisión, es el conjunto de naturales tales que la máquina que codifican para con alguna entrada.

Supongamos que es decidible y sea $M_{H'}$ una máquina que lo decide. Llegaremos a contradicción usando esta máquina para decidir K.

Entrada x

```
p:=
    Entrada y
        Ejecutar M_x(x)
        Parar
    Salida M_H'(p)
```

Con entrada x, nuestro programa construye internamente un subprograma, lo codifica en un natural p y finalmente da como salida lo que de M_H con entrda p. Puede genrar confusión la primera vez que se ve.

- Nuestro programa construye otro programa que recibe una entrada y, que no usa para nada, y ejecuta M_x con entrada x
  - En caso de que $M_x(x)$ pare, entonces para a continuación.

Hay que recalcar varias cosas:

- No estamos ejecutando el subprograma, solo lo estamos contruyendo.
- X es la variable de entrada del programa del programa principal, pero un valor constante en el subprograma.

Para verlo más claro vamos a ver una descripción del programa codificado por p más sencillo:

```
p:=TMprinf
    ''Entrada y
        Ejecutar M_\%d(\%d) , x, x
        Parar ''
Salida M_H'(p)
```

Con la instucción printf de c, convertimos un texto en otro substituyendo valores concretos (los $\%d$) por parametros (que son las x).

El programa que hemos creado, recibe una entrada, pasa de ella, y ejecuta otro programa que es constante independientemente de la entrada.

Veamos ahora que este programa decide k:

- Si x es de k el programa para con todas las entrada, por tanto $M_{H'}$ aceptará y nosotros aceptaremos
- Si x no es de k, entonces el programa no para con ninguna entrada, por tando $M_{H'}$ rechazará y nosotros rechazaremos.

Esto termina la demostración.

Estas argumentaciones se pueden simplificar ligeramente mediante el concepto de reducción. Una reducción entre 2 conjuntos, es una función computable y total que nos envia elementos de C1 a C2 y elementos que no son de C1 a elementos que no son de C2.

$$\forall x : (x \in C_1 \Leftrightarrow f(x) \in C_2)$$

En las demostraciones anteriores habían implicitas las reducciones siguientes:

Reducción de k al conjunto de parejas programa entrada, tales que el programa para con la entrada. Es una transformación computable porque es muy sencillo contruir un programa que nos transforme x en la pareja (x,x). Y es total porque todo x tiene pareja asociada. Si x es de k entonces este programa para con esta entrada y si no lo es el programa no para con la entrada. $K \leq \{\langle x,y\rangle | M_x(y)\downarrow\} x \mapsto \langle x,x\rangle$

Es una reducción de k al conjunto de programas que paran con alguna entrada. Es obvio que es una transformación computable y total. Además si x es de x, este programa para con alguna entrada y si x no es de k entonces este programa no para con alguna entrada. $K \leq \{x | \exists y : M_x(y)\downarrow\} : x \mapsto$

```
        Entrada y
            Ejecutar M_x(x)
            Parar
```

Cuando un conjunto se reduce a otro, la decibilidad del segundo implica la decibilidad del primero:

$$C_1 \leq C_2 : C_2 \in Dec \Rightarrow C1 \in Dec$$

Demostración: Sea M_{C2} una máquina que decide C_2. Sea M_f una máquina que computa $f : \mathbb{N} \to \mathbb{N}$ (La reduccion de C_1 en C_2)

Entonces la siguiente máquina decide C_1:

```
Entrada x
    Salida M_c2(M_f(x))
```

Para saber si un x es de x1 transformamos la pregunta en si f(x) es de C2 y como que se preserva la respuesta, la decibilidad de C2 nos ofrece la decibilidad de C1.

La propiedad anterior tiene **contrareciproco**. La no-decibilidad de C1 implica la no-decibilidad de C2. Para demostrar que un conjunto es indecidible es suficiente con que reduzcamos K a él.

$$C1 \leq C2 : C_1 \notin Dec \Rightarrow C2 \notin Dec$$

7.3 No semi-decidibilidad

Empezamos viendo que el complementario de K, es decir el conjunto de programas que se cuelgan con entrada ellos mismos, no es ni tan siquiera semi-decidible:

$$\overline{K} = \{x | M_x(x)\uparrow\}$$

Por demostraciones anteriores tenemos que K no es decidible.

Es fácil ver por este programa que K es semi-decidible, viendo el siguiente programa, que lo semi-decide:

```
Entrada x
    Ejecutar M_x(x)
    aceptar
```

El programa acepta justamente aquellas x de entrada, tales que la máquina que codifican con entrada x, para. Es decir las x de K. Así pues K es semi-decidible, pero no decidible.

Cuando un conjunto C y su complementario $\overline{C}$ son semidecidibles, entonces C es decidible:

$$C, \overline{C} \in semi - Dec \Rightarrow C \in Dec$$

Como K es semi-decidible, pero no decidible, entonces su complementario no puede ser semi-decidible también, porque entonces K sería decidible.

Con esta técnica podemos demostrar la no semi-decibilidad de muchos otros conjuntos.

7.3.1 Ejemplo

$$\{x | \forall y : M_x(y) \uparrow\}$$

El conjunto de programas que se cuelga con toda entrada.

De la sección anterior tenemos que su complementario $M_{H'}$ es indecidible:

$$H' = \{x | \exists y : M_x(y) \downarrow\}$$

Para terminar, con la misma técnica, solo queda demostrar que H' si es semi-decidible. Mediante el siguiente programa:

```
Entrada x
    t:= 0
    Ejecutar indefinidamente:
        Desde y:= 0 hasta t hacer:
            Si M_x(y) para en t pasos entonces aceptar
        t ++
```

Mediante t recorremos todos los tiempos de ejecución posibles, y para cada t recorremos todas las entradas y entre 0 y t, y ejecutamos $M_x(y)$ con cada uno de los y con tiempo t. Si x es de H' entonces existe una y con el que M_x para al cabo de un cierto número de pasos. En el momento que t sea mayor que ese número de pasos, y que ese y concreto, entonces pararemos y aceptaremos. En caso contrario si x no es de H', M_x no para con ninguna entrada y entonces este programa no aceptara.

Esta técnica de demostrar la no semidecibilidad de un conjunto a base de demostrar la semi-decibilidad pero no decibilidad de su complementario no sirve en todos los casos. Por ejemplo:

El conjunto de los programas que se cuelga con alguna entrada, no es ni semi-decidible ni el ni su complementario. Por suerte el método de reducción de la sección anterior también sirve.

$$C_1 \leq C_2 : C2 \in semi - Dec \Rightarrow C1 \in semi - Dec$$

7.4 Propiedades de cierre

7.4.1 Reunión e intersección

Complementación

Chapter 8

Reductibilidad y completeza

8.1 Reducciones

8.2 Propiedades de las reducciones

8.3 Reducciones e indecibilidad

8.3.1 Teorema s-m-n

Conjuntos de índices. Teorema de Rice

8.4 Ejercicios de ejemplo

1. Conjunto de programas tales que para una entrada dan como salida la propia entrada no es decidible.

$$L = \{p | \exists y : M_p(y) = y\}$$

Basta con mostrar una reducción de K a L

$x \mapsto p$

```
        entrada y
                ejecutar Mx(x)
                salida(y)
```

La transformación es computable y total, ya que se puede programar la transformación de x a p mediante un programa que recibe x y da como salida la codificación del programa en terminos de x. En el programa codificado por p resultante de la reducción, x no es una variable del programa, es una constante fija.

Se conserva la respuesta para las correspondientes entradas:

- Si x pertenece a k, p pertenece a L

- Si x no pertenece a k, p no pertenece a L

$x \in K \Rightarrow M_x(x) \downarrow \Rightarrow \forall y : M_p(y) = y$

$\Rightarrow \exists y : M_p(y) = y \Rightarrow p \in L$

Esto implica que el comportamiento del programa es el siguiente:

- Recibe una entrada
- Pierde un poco el tiempo con Mx(x) que acaba parando
- Da salida y
- El programa codificado por p, da por salida la propia entrada para cualquier entrada. En particular se cumple la condición que define L

$x \notin K \Rightarrow\Rightarrow M_x(x) \uparrow \Rightarrow \forall y : M_p(y) \uparrow$

$\Rightarrow \neg\exists y : M_p(y) = y$

2. El lenguaje anterior es semidecidible

Para ello basta con mostrar un programa, que dado un p de entrada si p cumple la condición, nuestro programa acepta. Si no cumple la condición, tanto puede rechazar como quedarse colgado.

$L = \{p | \exists y : M_p(y) = y\}$

Llamamos M a la máquina siguiente:

```
input y {
        t:= 1
        ejecutar indefinidamente:
                desde y:= 0 hasta t hacer:
                        si Mp(y) = y en t pasos
                                entonces aceptar
                t ++
```

$p \in L \Rightarrow \exists y : M_p(y) = y \Rightarrow \exists T : M_p(y) = y en T pasos$

cuando $t \geq max(y, T)$ M acepta $\Rightarrow M(p)$ acepta.

$p \notin L \Rightarrow \neg\exists y : M_p(y) = y \Rightarrow M(p)$ no acepta. El programa se queda colgado.

M semidecide L y por tanto L es semi-decidible, pero no decidible (por el anterior).

3. Conjunto de programas tales que para alguna entrada dan algo distinto de la propia entrada

$L = \{p | \exists y : M_p(y) \neq y\} = \{p | \exists y : (M_p(y) \text{ da salida distinta de } y)\}$

Vamos a demostrar que L es semi-decidible pero no decidible. La demostración es muy parecida para el problema anterior. En lugar de reescribir todo, se va a retocar el problema anterior.

$x \mapsto p$

```
        entrada y
```

```
                ejecutar Mx(x)
                salida 0
```

En este programa, existe una entrada que la salida es distinta que la entrada:

$x \in K \Rightarrow M_x(x) \downarrow \Rightarrow M_p(1) = 0 \Rightarrow \exists y : M_p(y) \neq y \Rightarrow p \in L$

$x \notin K \; M_x(x) \uparrow \Rightarrow \forall y : M_p(y) \uparrow \Rightarrow \neg \exists y : M_p(y) \neq y \Rightarrow p \notin L$

```
        input y {
                t:= 1
                ejecutar indefinidamente:
                        desde y:= 0 hasta t hacer:
                                si Mp(y) != y en t pasos
                                        entonces aceptar
                        t ++
```

Se cambia el Mp(y) = y por Mp(y) != y.

$p \in L \Rightarrow \exists y : M_p(y) \neq y \Rightarrow \exists T : M_p(y) \neq yenTpasos$

cuando $t \geq max(y, T)$ M acepta $\Rightarrow M(p)$ acepta.

$p \notin L \Rightarrow \neg \exists y : M_p(y) \neq y \Rightarrow M(p) noacepta$. El programa se queda colgado.

4. Para alguna entrada el programa se cuelga o da una salida distinta de la propia entrada

$L = \{p | \exists y : (M_p(y) \uparrow \vee_p(y) \neq y)$

Empezamos comprobando si la reducción de K que servía en problemas anteriores sirve:

$(k \leq L)\ x \mapsto p =$

```
        entrada y
                ejecutar Mx(x)
                salida 0
```

Para la respuesta afirmativa, si x pertenece a k, entocnes Mx(x) para y por lo tanto este programa, por ejemplo para entrada 1, da salida 0, de modo que existe una entrada para la cual el programa da salida distinta de la entrada, de modo que $p \in L$ en el caso que $x \in K$.

Si $x \notin K$ entonces Mx(x) se cuelga y por lo tanto este programa se cuelga con todas las entradas. Por lo tanto nuestro programa p también cumple que exista alguna entrada con la cual se cuelga y por lo tanto $p \in L$ aun cuando $x \neq K$. La respuesta **negativa no se preserva.**

Es posible encontrar una reducción de K a L para este problema, pero puede resultar complicado.

Para este problema concreto hay una reducción sencilla desde $\overline{K}$ a L. Es suficiente con cambiar el 0 por y.

```
        entrada y
                ejecutar Mx(x)
                salida y
```

$x \in \overline{K} \Rightarrow M_x(x) \uparrow \Rightarrow \forall y : M_p(y) \uparrow \Rightarrow p \in L$

$x \notin \overline{K} \Rightarrow M_x(x) \downarrow \Rightarrow \forall y : M_p(y) = y$. Como no existe una entrada con la que el programa se cuelgue o de una salida distinta a la entrada, tenemos que $p \notin L$.

Al ser una reducción desde $\overline{K}$, L no es ni semi-decidible.

4. El conjunto de programas tales que para toda entrada dan como salida la propia entrada

$(K \leq L)$

$x \mapsto L$

```
        entrada y
                ejecutar Mx(x)
                salida y
```

Si $x \in K$, entonces Mx(x) para, de modo que este programa para cualquier entrada, pierde un rato el tiempo y da como salida la propia entrada. Y por lo tanto p cumple la condición de modo que $p \in L$

Si $x \notin K$, entonces Mx(x) se cuelga de modo que este programa se cuelga con toda entrada y no es cierto que con toda entrada se de como salida la propia entrada, y p no es de L. Se preserva la respuesta negativa.

Con esto se concluye que L no es decidible.

Si intentamos demostrar que es semi-decidible nos va a resultar imposible. No hay manera de considerar todas las entradas para poder comprobar para cada una de ellas si acaban parando y dan como salida la propia entrada. Vamos a demostrar que no es **semi-decidible**.

$\overline{K} \leq L$

Si $x \in \overline{K}$, si $M_x(x) \uparrow$, entonces este programa ha de parar con todas las entradas y para cada una de ellas dar como salida la propia entrada.

Si $x \notin \overline{K}$, es decir $M_x(x) \downarrow$, p no debe cumplir la condición.

Parecen condiciones opuestas. Pero se utilizará un truco que servirá en otros problemas también.

$x \mapsto p =$

```
        entrada y
                Si Mx(x) para en y pasos
                        salida 0
                sino salida y
```

La condición de parada simula la ejecución de $M_x(x)$ durante y pasos, y si ha parado en y pasos o menos se cumple la condición y damos salida 0. Esta instrucción no se puede quedar colgada, nuestro programa nunca se para y siempre da salida o bien 0 o bien y.

En caso contrario damos salida y.

Si $x \in \overline{K} \Rightarrow M_x(x) \uparrow \Rightarrow \forall y : M_p(y) = y \Rightarrow p \in L$.

Si $x \notin \overline{K} \Rightarrow Mx(x) \downarrow$ en un cierto número de pasos T. Para entradas mayores estrictas de t, vamos a entrar por la rama de salida 0 y esos y cumpliran que son distintos de 0.

Para $y > Y, M_p(y) = 0 \neq y \Rightarrow p \in L$

Con esto concluimos que no es ni tan si quiera semi-decidible.

Reducciones alternativas.

$x \mapsto p =$

```
        entrada y
                Si Mx(x) para en y pasos
                        colgarnos
                sino salida y
```

Si $x \notin \overline{K} \Rightarrow Mx(x) \downarrow$ en un cierto número de pasos T. Para entradas mayores estrictas de t, el programa se cuelga. Para $y > Y, M_p(y) \uparrow \Rightarrow p \in L$

5. conjunto de programas que dan salida 1 con toda entrada. Demostrar que no es semi-decidible

$L = \{p | \forall y : M_p(y) = 1\}$

$x \mapsto p =$

```
        entrada y
                Si Mx(x) para en y pasos
                        colgarnos
                sino salida 1
```

Si $x \in \overline{K} \Rightarrow M_x(x) \uparrow \Rightarrow \forall y : M_p(y) = 1 \Rightarrow p \in L$.

Si $x \notin \overline{K} \Rightarrow Mx(x) \downarrow$ en un cierto número de pasos T. Para entradas mayores estrictas de t, el programa se cuelga. Para $y > Y, M_p(y) \uparrow \Rightarrow p \in L$

6. Conjunto de programas tales que para alguna entrada da salida distinta de la salida que da el propio programa con la misma entrada

$L = \{\exists y : M_p(y) \neq M_p(y)\}$

Se trata de una condición incumplible. Es una definición del conjunto vacío y el conjunto vacío si es decidible.

```
        entrada p
                rechazar
```

Un progama que para siempre y rechaza para todas las entradas. El lenguaje del enunciado es decidible.

8.4.1 Ejemplos 2

Tenemos un programa:

```
    input y
        output y + 1
```

Da como salida la propia entrada + 1. Sea n el natural que codifica este programa, entonces M_n, la máquina o programa codificada por n, es el programa anterior.

φ_n es la función implementada por el programa. Una función no es más que un conjunto de parajas que relaciona cada posible entrada del programa con su correspondiente salida, si es que esta definida.

$$\varphi_n = \{(0,1),(1,2),(2,3)...\{$$

φ_n y M_n son objetos muy distintos.

Para el siguiente programa:

```
input y
    if y ∈ 2̇ entonces
        salida y
    else infiniteloop
```

$\varphi_n = \{(0,0),(2,2),(4,4)\}$. Se trata de una función que no es total, pues no esta definida para los números naturales impares.

La función implementada por el siguiente programa esta totalmente indefinida y es el conjunto de parejas vacío, que se denota $\emptyset$

```
input y
    infiniteloop
```

8.4.2 Teorema de Rice

Sea $L \subseteq \mathbb{N}$ que cumple:

- L es un conjunto de indices, ha de ser un conjunto de naturales tales que la función que implementan cumplan una cierta condición. $L = \{n|P(\varpi n)\}$
- $L \neq \emptyset$ y $L \neq \mathbb{N}$. No es ni el vacío ni el total.

Bajo estas condicones se puede concluir que el conjunto L no es decidible.

Además, si L contiene a los naturales que implementan la función totalmente indefinida, se puede concluir que L no es ni tan siquiera **semi-decidible**:

$$\varphi_m = \emptyset \Rightarrow n \in L$$

$$P(\emptyset)$$

Ejemplos:

- $\{n|\varphi_n(1) = 2\}$: el conjunto de naturales n, tales que la función implementada por la máquina n con entrada 1, da salida 2.
- $\{n|\varphi_n$ es biyectiva$\}$

En todos los casos, que un natural pertenezca a un conjunto o no depende de como es la función que implementa, y no del natural en si. Si 2 naturales implementan la misma función, o bien los 2 pertenecen a L o bien ninguno de los dos pertenece a L.

Imaginemos que tenemos un programa x que cumple la siguiente condición: $\{n|\varphi_n(n) = 1\}$.

$$\varphi_x(x) = 1$$

También tenemos un programa y, distinto de x, pero que se comporta igual que x, implementa la misma función.

$$\varphi_y(x) = 1$$

Suponemos también que tanto x como y, que implementan la misma función, con entrada y no dan salida 1:

$$\varphi_x(y) \neq 1$$

$$_y(y) \neq 1$$

Bajo estas condiciones, x si que cumple la condición y es del conjunto n, pero y no la cumple y por lo tanto no es del conjunto, aún cuando ambos implementan la misma función. Por lo tanto esto no es un **conjunto de índices**.

Conjunto de programas que con entrada 1, dan salida 2:

$$L = \{p|\varphi_p(1) = 2\}$$

Empezamos viendo que L si que es semi-decible:

```
input p
    Si Mp(1) = 2
        aceptar
    else rechazar
```

Se puede quedar colgado si la simulación con entrada 1 no termine.

L **no es decidible**:

$K \leq L$

$x \mapsto p$

```
input y
    ejecutar Mx(x)
    output 2
```

Si $x \in K$ el programa para y para cualquier entrada da salida 2. Por lo que para entrada 1, da salida 2 y cumple con la condición de ser de L.

$$x \in K \rightarrow M_x(x) \downarrow \rightarrow p \in L$$

Si x no pertenece a k, se cuelga para todas las entradas y para entrada 1 no dará salida 2.

$$x \notin k \rightarrow M_x(x) \uparrow \rightarrow p \notin L$$

Utilizando el teorema de Rice se ve que L si que es un conjunto de indices, para que un p pertenezca al conjunto L o no, debe cumplirse una condición definida únicamente sobre φ_p. L es diferente al conjunto vacío porque hay programas que con entrada 1 dan salida 2, por ejemplo:

```
input y
    output 2
```

Además L es diferente del total, porque también existen programas que con entrada 1 no dan salida 2.

```
input y
    salida 1
```

Como se cumplen todas las propiedades del teorema de Rice, se conlcuye que L no es decidible.

Conjunto de programas que definen la función totalmente indefinida

$L = \{p | \varphi_p = \emptyset\}$

Este conjunto no es ni semi-decible. Basta con ver que se cumplen todas las propiedades del teorema de Rice:

- L es un conjunto de índices. Es obvio por la propia descripción del conjunto.
- L no es el vacío se muestra implementando un porgrama que imprementa la función totalmente indefinida
- L no es el total se justifica mostrando un programa que no implemente la función totalmente indefinida.

En este punto el teorema nos dice que el conjunto no es decidible. Para demostrar que no es semi-decidible, hace falta justificar la última condición, si un programa implementa la función totalmente indefinida, entonces su codificación es del conjunto. Esto es obvio, porque p es el conjunto de naturales, que implementan programas tales que la función implementada por el programa que codifican es la función totalmente indefinida.

Conjunto de programas que implementan funciones biyectivas

Una función biyectiva implica que la imagen de la función es todos los naturales. El teorema de Rice nos permite concluir que esto no es decidible.

L es un conjunto de indices.

$L \neq \emptyset$. Existen programas que implementan funciones biyectivas. Para 2 entradas distintas tenemos 2 salidas distintas.

```
input y
    salida y
```

$L \neq \mathbb{N}$. Tampoco es el total, porque existen programas que no implementan una función biyectiva.

```
input y
    salida 1
```

No podemos concluir por el teorema de Rice que no es semi-decidible porque la función totalmente indefinida no es exhaustiva, su imagen no contiene nada. Sin embargo el lenguaje no es ni semi-decidible. Este es un ejemplo para el que el teorema de Rice es insuficiente y una mala elección para clasificar el problema.

Hay que plantear de la manera tradicional:

$\overline{K} \leq L$

$x \mapsto p$

```
input y
    si Mx(x) para en y pasos
        output 1
    else output y
```

$x \in \overline{K} \rightarrow M_x(x) \uparrow \Rightarrow \forall y : \varphi_p(y) = y \Rightarrow \varpi_p esbiyectiva.$ Calcula la función identidad, que es biyectiva.

$x \notin \overline{K} \Rightarrow M_x(x) \downarrow$ en un cierto número de pasos t$\Rightarrow \forall y > T : \varphi_p(y) = 1 \Rightarrow \varphi_p$ no es biyectiva $p \notin L$

Conjunto de parejas de programas, tales que para una entrada, ambos programas acaban dando la misma salida

$L = \{\langle p, q\rangle | \exists z : M_p(z) = M_q(z)\}$

L es semi-decidible:

```
input <p,q>
    t:= 1
    ejecutar indefinidamente:
        desde z:= 0 hasta t hacer:
            si Mp(z) = Mq(z) en t pasos
                entonces aceptar
        t++
```

El programa se queda colgado si no se llega a verificar la condición, cosa que no importa porque se esta semi-decidiendo. Y si cumple la condición para ser de L acepta.

L no es decidible:

$x \mapsto p$

```
input y
    Ejecutar Mx(x)
    salida 1
```

Da salida 1 para todas las entradas.

$x \mapsto q$

```
input y
    salida 1
```

Ambos programas dan la misma salida para todas las entradas, y la pareja $\langle p, q\rangle$ pertenece a L. Se preserva la respuesta afirmativa.

Si x no pertenece a K, el programa p se cuelga para cualquier entrada, mientras que q da salida 1. Por lo tanto no hay una entrada para que ambos den una misma salida para una entrada.

Otro ejemplo:

$L = \{\langle p, q\rangle | \forall z : M_p(z) = M_q(z)\}$

L no es semidecidible:

$\overline{K} \leq L$

$x \mapsto p$

```
    input z
        Si Mx(x) para en z pasos
            ouput 2
        else output 1
```

$x \mapsto q$

```
    input z
        output 1
```

$x \in \overline{K} \Rightarrow M_x(x) \uparrow \Rightarrow \forall z : M_p(z) = 1 \wedge M_q(z) = 1 \Rightarrow \forall z : M_p(z) = M_q(z) \Rightarrow \langle p, q \rangle \in L$

$x \notin \overline{K} \rightarrow M_x(x) \downarrow$ en un cierto número de pasos T $\Rightarrow M_(T) = 2 \wedge M_q(T) = 1 \Rightarrow \langle p, q \rangle \notin L$

Chapter 9

Decibilidad

9.1 Lenguajes decidibles

Para responder a la pregunta de que pueden y que no pueden hacer los ordenadores debemos considerar las siguientes preguntas sobre los lenguajes:

- ¿Que lenguajes son Turing-decidibles?
- ¿Qué lenguajes son Turing-reconocibles?
- ¿Qué lenguajes no son ninguna de las 2 cosas anteriores?

La decibilidad tiene que ser estudiada para saber que problemas no se pueden resolver por ordenadores.

9.1.1 Niveles para describir algoritmos

Existen 3 niveles para describir los algoritmos:

- Formal: DFA, CFGs...
- Implementación: pseudo-código
- Alto nivel: por ejemplo en inglés

9.1.2 Formato de entrada y salida

Las máquinas de Turing únicamente aceptan strings sobre un cierto alfabeto como entrada.

Si el input X e Y esta en cualquier otro formato (Grafo, TM, polinómico) usamos la siguiente notación para codificarlo como strings:

$$\langle X, Y \rangle$$

9.1.3 Problema de aceptación para DFAs

Dado un DFA B, acepta un string w?

La entrada para la máquina de Turing será:

$$\langle B, w \rangle$$

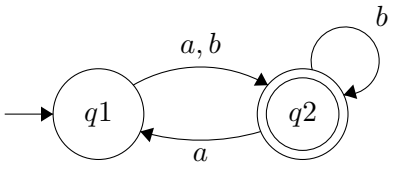
El universo contiene todas las instancias posibles:

$$\Omega = \{\langle B, w \rangle | B \text{ es un DFA y } w \text{ es un string } \}$$

El lenguaje comprende todas las instancias que dan SI:

$$A_{DFA} = \{\langle B, w \rangle | B \text{ es un DFA que acepta el string} w\} \subseteq \Omega$$

DFA D_1



$\langle D_1, abb \rangle \in A_{DFA}$

$\langle D_1, \lambda \rangle \notin A_{DFA}$

Theorem 9.1.1 *A_{DFA} es un lenguaje decidible*

$$A_{DFA} = \{\langle B, w \rangle | B \text{ es un DFA que acepta el string } w\}$$

Para demostrar que es decidible, es necesario mostrar $\exists$ TM M que decide A_{DFA}

- Toma cualquier entrada $\langle B, w \rangle \in \Omega$ como entrada
- Se para y acepta si $\langle B, w \rangle \in A_{DFA}$
- Se para y rechaza si $\langle B, w \rangle \notin A_{DFA}$

Demostración

M = Con input $\langle B, w \rangle \in A_{DFA}$, donde

- $B = (Q, \Sigma, \delta, q_0, F$ es un DFA
- w = w1w2...wn $\in \Sigma^*$ es una entrada a ser procesada por B

```
Si no <B,w> correctamente codificado
        rechaza
Simular B en w con la ayuda de punteros q e i:
        q = q0
        q in Q apunta al estado actual del DFA B
        i in 1...|w| apunta la posici n actual del string w
        while i < |w|
                q cambia de acuerdo al simbolo de entrada wi y la funci
                i ++
Si B acaba en estado q in F
        aceptar
else
        rechazar
```

Problema de aceptación de NFAs

¿Dado un NFA B, acepta un string w?

$$A_{NFA} = \{\langle B, w\rangle | B \text{ es un NFA que acepta } w\}$$

Es un lenguaje decidible

Demostración:

TM: con entrada $\langle B, w\rangle \in \Omega = \{\langle B, w\rangle | B$ es un NFA w es un string $\}$

- B $= /Q, \Sigma, \delta, q_0, F$ es un NFA
- $w \in \Sigma^*$ es un sting de entrada B.

```
Si input mal codificado
        rechaza
Se convierte el NFA a DFA
Se corre la m aquina ADFA que decide M con entrada <C,w>
Si M acepta <C,w>
        aceptar
else
        rechazar
```

Problema de aceptación sobre RegEx es decidible

Dada una reg exp R, ¿genera el string w?

$$A_REX = \{\langle R, w\rangle R \text{ es una expresión regular que genera el string } w\}$$

```
Si <R,w> no es una codificaci n correcta
        rechazar
Convertir R a un DFA B
Correr la TM que decide ADFA con input <B,w> y dar output
```

Problema del vacío sobre DFAs

Dado un DFA, ¿reconoce el lenguaje vacío?

$E_{DFA} = \{\langle B \rangle | B$ es un DFA y $L(B) = 0$, un subset del universo $\Omega = \{\langle B \rangle | B$ es un DFA $\}$

Es **decidible**

Demostración:

- Comprobar si algún estado aceptador es accesible desde el estado incial.
- Si pasa eso, rechazar.

Con entrada $\langle B \rangle \in \Omega$, donde $B = (Q, \Sigma, \delta, q_0, F$ es un DFA:

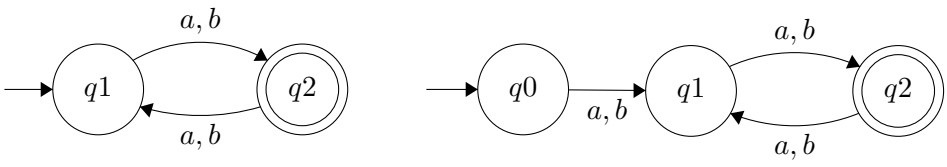
```
Si <B> no es codificacion de DFA correcta
        rechazar
S:= conjunto de estados accesibles desde q0. Inicialmente S = {q0}
while i:=0..|Q| veces:
        Si S tiene un elemento de F
                rechazar
        Else, a adir a S los elementos accesibles desde S usando func d
                Si exist qi in S && l in Sigma con ftrans
                        S.push_back(qj)
Si S intersec F = 0
        acepta
else
        reject
```

9.1.4 Problema de equivalencia de DFA es decidible

Dados 2 DFAs, ¿son equivalentes?

$$EQ_{DFA} = \{\langle A, B \rangle | \text{ A y B son DFAs y } L(A) = L(B)\}$$

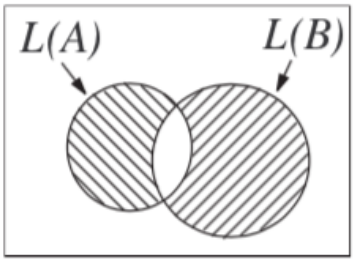
Un subset del universo $\Omega = \{\langle A, B \rangle | \text{ A y B son DFAs } \}$



Los automatas A_1 y B_1 no reconocen el mismo lenguaje, por lo que $\langle A_1, B_1 \rangle \notin EQ_{DFA}$

$$EQ_{DFA} = \{\langle A, B \rangle | \text{ A y B son DFAs y } L(A) = L(B)\}$$

Dados A y B, se construye un **DFA C** tal que C acepta cualquier string aceptado por A o B, pero no por los dos:



- $L(C)$ es la diferencia simetrica de L(A) y L(B)

Por lo que $L(A) = L(B)$ si $L(C) = 0$

Se construye C usando el agoritmo para la intersección del complemento de los DFA, y uniones **(Teorema 1.25)**.

- Teorema 1.25: La clase de los lenguajes regulares es cerrada bajo la unión.

Con input $\langle A, B\rangle \in \Omega$ donde A y B son DFAs

```
Comprobar si <A,B> esta bien codificado, si no rechazza
Contruir DFA C tal que
    L(C) = [L(A) ∩ ¬L(B)] ∪ [¬L(A) ∩ L(B)]

Correr la TM que decide EDFA con input <C>

Si <C> in Edfa
    Acepta
Si <C> notin Edfa
    Reject
```

9.1.5 El problema de aceptación de CFGs es decidible

Dado un CFG G, genera un string w?

$$A_{CFG} = \{\langle G, w\rangle | \text{ G es un CFG y w un string}\}$$

Para cada pareja de entrada especifica:

- $\langle G, w\rangle \in A_{CFG}$ si G genera w
- $\langle G, w\rangle \notin A_{CFG}$ si G no genera w

Theorem 9.1.2 *A_{CFG} es decidible*

```
Se comprueba si <G,w> es una codificaci n correcta.
    Si no, rechaza
Se convierte el CFG G a la forma normal de Chomsky G'
```

```
Si w = λ
    Comprueba si S -> λ es una regla de G'
        Si lo es, acepta
        Si no, rechaza
Si w /= λ, enumera todas las derivaciones con 2|w|-1 pasos
    Si alguna genera w, acepta
    Else rechaza
```

9.1.6 Problema del lenguaje vacío para CFGs

Theorem 9.1.3 *El lenguaje* $E_{CFG} = \{\langle G \rangle | G \text{ es un CFG con } L(G) = 0\}$ *es decidible*

Demostración:

Con entrada $\langle G \rangle$ donde G es un CFG

```
Comprobar si <G> es una codificaci n correcta
    Si no -> rechaza
```

```
Define un set T ⊆ V ∪ Σ tal que u ∈ T implica u ⇒* w para alguna w ∈ Σ*.
Inicialmente T = Σ

Repetir |V| veces
    Comprobar cada regla B -> X1...Xk en R
    Si B ∉ T y cada X_i ∈ T
        T.push_back(B)

Si S ∈ T
    Reject
Else
    Accept
```

9.1.7 ¿Son equivalentes 2 CFGs?

Definimos el lenguaje siguiente:

$$EQ_{CFG} = \{\langle G, H \rangle | G, H \text{ son CFGs y } L(G) = L(H)\}$$

Para DFAs podriamos usar el procedimiento de decisión del vacio para resolverlo.

Tratamos de contruir el CFG C a partir de G y H, tal que:

$$L(C) = [L(G) \cap \overline{L(H)}] \cup [\overline{L(G)} \cap L(H)]$$

Y a partir de el comprobar si L(C) es vacio usando la TM que decide E_{CFG}

Pero **no podemos definir** CFG C a como la **diferencia simetrica**, porque los **CFLs no son cerrados bajo el complemento ni intersección**

Por lo que EQ_{CFG} **no es un lenguaje decidible**

9.1.8 Los CFLs son decidibles

Theorem 9.1.4 *Cada CFL L es un lenguaje decidible.*

Demostración:

L es un CFL

- G' es un CFG para el lenguaje L
- S es la TM A_{CFG}
  - $A_{CFG} = \{\langle G, w\rangle | G \text{ es un CFG que genera el string w}\}$

Se construye la TM $M_{G'}$ para el lenguaje L teniendo $CFGG'$, de la siguiente forma:

```
MG' input w
Correr la TM qu decide S para input <G',w>
Si S acepta
    Aceptar
Else
    Reject
```

¿En qué se diferencian las TM S y $M_{G'}$?

- TM S tiene input $\langle G, w\rangle$
- TM $M_{G'}$ tiene input w para una G' fijada

9.1.9 La TM universal U

¿Es una TM capaz de simular todas las otras máquinas?

Dada una codificación $M, w\rangle$ de una TM M con entrada w, podemos similar M con entrada w?

Podemos hacer esto a través de una TM universal llamada U:

```
U = Con input <M,w> donde M es una TM y w un string
    Simula M con input w
    Si M entra en estado aceptador
        Acepta
    Si M entra en estado de rechazo
        Rechaza
```

Se puede pensar en U como un **emulador**.

9.2 El problema de aceptación de una TM es indecidible

Dada una máquina de TM M y un string w, la máquina M acepta w?

$$A_{TM} = \{\langle M, w\rangle | M \text{ es una TM que } \textbf{acepta} \text{ el string w}\}$$

Con universo:

$$\Omega = \{\langle M, w\rangle | M \text{ es una TM y w un string}\}$$

Para una dupla específica $\langle M, w\rangle \in \Omega$ de una TM M y un string w:

- $\langle M, w\rangle \in A_{TM}$ si M **acepta** w
- $\langle M, w\rangle \notin A_{TM}$ si M **rechaza** w

La TM universal U:

- Reconoce A_{TM}, por lo que A_{TM} es Turing-reconozible
- U no decide A_{TM}
  - Si M hace loop en w, entonces U hace loop en $\langle M, w\rangle$

Pero, ¿podemos decidir A_{TM}?

- Esto es indecidible

9.3 Algunos lenguajes no son Turing-reconocibles

9.4 Algunos problemas indecidibles

Los ordenadores son limitados en un sentido fundamental. Algunos problemas comunes no son resolvibles:

- Un programa ordena un array de enteros?
- Son los programas y especificaciones objetos matemáticos precisos?
  - Uno podría pensar entonces que es posible desarrollar un algoritmo que determine si un programa cumple con una especificación. Pero es **imposible**.

Por lo anterior se tienen que introducir algunas ideas nuevas.

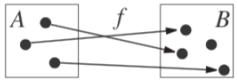
9.4.1 Mapeo y funciones

Consideremos fcn $f : A \to B$ mapea objetos de un set A a un set B

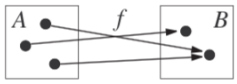
9.5 Sets contables e incontables

Conjunto de enteros:

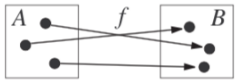
- **Definition:** f is **one-to-one** (aka **injective**) if every $x \in A$ has a unique image $f(x)$:
  - If $f(x) = f(y)$, then $x = y$.
  - Equivalently, if $x \neq y$, then $f(x) \neq f(y)$.



- **Definition:** f is **onto** (aka **surjective**) if every $z \in B$ is "hit" by f:
  - If $z \in B$, then there is an $x \in A$ with $f(x) = z$.



- **Definition:** f is a **correspondence** (aka **bijection**) if it both one-to-one and onto.
  - Inverse fcn $f^{-1} : B \to A$ then exists.



1/1	1/2	1/3	1/4	1/5	...
2/1	2/2	2/3	2/4	2/5	...
3/1	3/2	3/3	3/4	3/5	...
4/1	4/2	4/3	4/4	4/5	...
...	...	...	...	...	...

Conjunto de naturales:

Un conjunto S es infinito, por lo que no existe una función exhaustiva $f : S \to N$

- El set S es al menos tan grande como el set N

Un conjunto S es contable si es finito o tiene el mismo tamaño que N

- Podemos enumerar todos los elementos en un conjunto contable.
- Cada elemento esta enumerado.

9.5.1 Conjunto de los racionales es contable

El conjunto de los racionales definido por: $Q = \{\frac{m}{n} | m, n \in N\}$ es contable.

Demostración: Si se escriben los elementos de Q como una array 2-dimensional infinita:

Si intentamos:

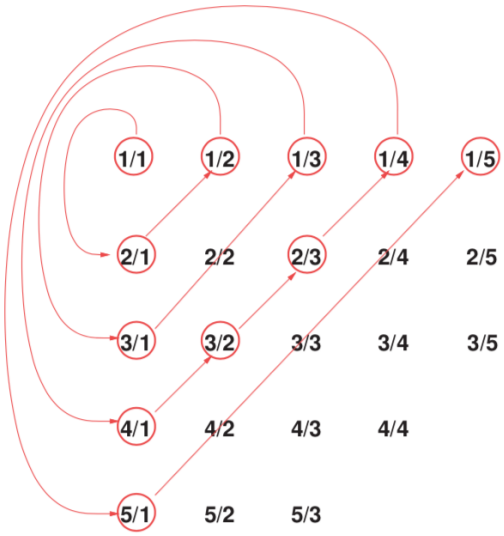
- Enumerar los elementos de la primera fila
- Enumerar después los de la segunda
- Y así sucesivamente

Nunca llegaremos a la segunda fila, ya que la primera es infinita.

Si por otro lado intenatmos:

- Enumerar los elementos desde las diagonales sur-oeste hacia el nor-este

- Saltamos los duplicados



9.5.2 Más sets contables

Existe una correspondencia entre $N = \{1, 2, 3...\}$ y cada uno de los conjuntos:

- $Z = \{... - 2, -1, 0, 1, 2...\}$
- $N * 2 = \{(i, j)|i, j \in N\}$
- $\{a\}$
- Σ^* para cualquier alfabeto Σ
  - Ej: $\Sigma = \{a, b\}$

9.5.3 Conjuntos no enumerables

Los conjuntos no contables son más grandes que los contables.

Definición: un numero real es un numero con representación decimal:

- $\pi = 3.1415926...$
- $\sqrt{2} = 1.3132136...$
- $2 = 2.0000...$

Theorem 9.5.1 *El conjunto $\mathbb{R}$ de todos los número reales es incontable*

Demostración: Supongamos que hay una correspondencia entre $\mathbb{N}$ y $\mathbb{R}$:

n	f(n)
1	3.14159...
2	0.5555...
3	40.00000....
4	15.20361...
...	...

Como la correspondencia existe, se supone que contendrá cada número real.

Cada número esta escrito como una expansión decimal infinita.

Si ahora construimos un número x entre 0 y 1 que no este en la lista usando la diagonalización de **Cantor**, tenemos $x = 0.d_1d_2d_3...$ donde

- d_n es el n-esimo digito despues del punto decimal en expansión decimal de x
- d_n es diferente que el n-esimo digito en el n-esimo número de la lista.

n	f(n)
1	3.14159...
2	0.5$\underline{5}$55...
3	40.00$\underline{0}$00....
4	15.203$\underline{6}$1...
...	...

Por ejemplo podemos tomar $\mathbf{x = 0.2617}$

$\forall n, x$ difiere del n-esimo numero en la lista en almenos posición n

- Por lo que x no esta en la lista
- Es contradictorio, puesto que $\mathbb{R}$ debería contener todos los número, incluido x

Por lo tanto:

$$\nexists \text{ correspondencia } f : \mathbb{N} \to \mathbb{R} \text{ por lo que } \mathbb{R} \text{ es incontable}$$

9.5.4 El conjunto de todas las TMs es contable

Por cada alfabeto finito ψ, el conjunto ψ^*

Demostración: enumerar strings en orden de strings.

Corolario: el conjunto de las TMs es contable

Demostración:

- Cada TM tiene una descripción finita
  - Podemos describirla con la codificación $\langle M \rangle$
  - La codificación es un string finito de símbolos sobre un alfabeto *psi*

- Si enumeramos todos los strings sobre ψ
  - Omitomos todos los que no son codificaciones válidas de TM
- Como que ψ^* es contable
  - Existe solo un número contable de TMs

9.5.5 El conjunto de todos los lenguajes es incontable

Corolario: El set β sobre todas las secuencias binarias es incontable

Demostración: Usar el argumento de diagonalización como prueba de que $\mathbb{R}$ es incontable

Corolario: El conjunto L de todos los lenguajes sobre el alfabeto Σ es incontable

Demostración:

- **Idea:** mostrar $\exists$ correspondencia x entre L y β
- Secuencia caracteristica del lenguaje esta definida por:
  - $x : L \to \beta$
  - Se escriben los elementos de Σ^* en orden: s1,s2,s3...
  - Cada lenguaje $A \in L$ tiene una secuencia unica $X(A) \in B$
  - El n-esimo bit de $X(A)$ es 1 si y solo si $s_n \in A$

Ejemplo: Para $\Sigma = \{0, 1\}$

$$\Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, ..\}$$

$$A = \{0, 00, 01, 000\}$$

$$X(A) = 01011001$$

El mapeo $X : L \to \beta$ es una correspondencia porque

- Es injectivo: lenguajes diferentes A_1 y A_2 se diferencian en como mínimo un string s_i, por lo que los i-esimos bits de $X(A_1)$ y $X(A_2)$ son diferentes.
- Exhaustiva: para cada secuencia $b \in \beta$, $\exists$ lenguaje A para el cual $X(A) = b$

Por lo tanto, L tiene el mismo tamaño que el set **incontable** β, por lo que L es incontable.

9.6 Algunos lenguajes no son Turing-reconocibles

Como cada TM reconoce algún lenguaje, el conjunto de las TM es contable y el conjunto de todos los lenguajes es incontable, **existen más lenguajes que TMs que los puedan reconocer.**

Corollary 9.6.0.1 *Algunos lenguajes no son Turing-reconocibles*

¿Que lenguajes son Turing-reconocibles?

Recordando el problema de aceptación de las TM

Dada una TM M y un string w, ¿M acepta w?

$$A_{TM} = \{\langle M, w\rangle | M \text{ es una TM que } \textbf{acepta} \text{ el string w}\}$$

$$\text{Sobre el universo } \Omega = \{\langle M, w\rangle | M \text{ es una TM y w es un string}\}$$

El universo Ω contiene todos los posibles pares de TM y w, no solo una instancia específica.

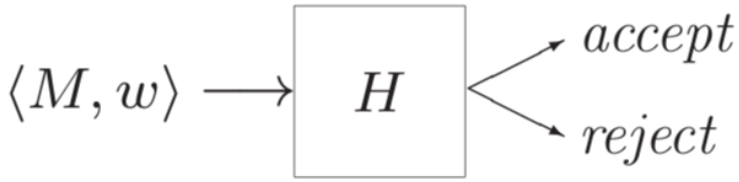
Para una TM M y string w específicos:

- Si M acepta w, entonces $\langle M, w\rangle \in A_{TM}$ es una instancia positiva
- Si M rechaza w, entonces $\langle M, w\rangle \notin A_{TM}$ es una instancia negativa

Theorem 9.6.1 *A_{TM} es indecidible*

Demostración por contradicción

Supongamos que A_{TM} es decidida por un TM H, con entrada $\langle M, w\rangle \in \Omega$

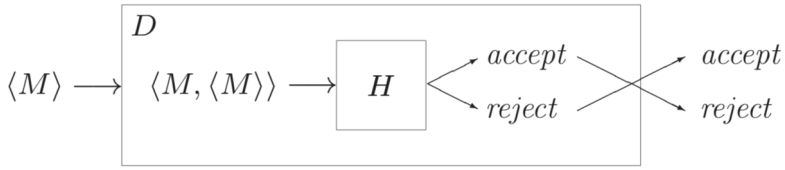


Se usa H como subrutina para definir otra TM D, con input $\langle M\rangle$

¿Qué pasa cuando corremos D con entrada $\langle D\rangle$?

- D acepta $\langle D\rangle$ si y solo si D no acepta $\langle D\rangle$
  - Lo que es imposible.

Explicación:



- Si existe una TM H que decide A_{TM}
  - TM H toma como entrada $\langle M, w\rangle \in \Omega$, donde M es una TM y w string
  - H acepta $\langle M, w\rangle \in A_{TM}$ si M acepta W
  - H rechaza $\langle M, w\rangle \notin A_{TM}$, si M no acepta w
- Consideramos el lenguaje $L = \{\langle M\rangle | M$ es una TM que no acepta $\langle M\rangle\}$
- Usando TM H como una subrutina, podemos construir TM D que decide L:

```
input <M>
    Correr H con input <M<M>>
    Si H acepta, rechazar
    Si H rechaza, aceptar
```

- Cuando corremos D con entrada $\langle D\rangle$
  - Parte 1 de D corre H con entrada $\langle D, \langle D\rangle\rangle$
  - D acepta $\langle D\rangle$ si y solo si D no acepta $\langle D\rangle$
    - **Contradicción**

Por lo tanto TM H no existe y A_{TM} es indecidible.

9.7 Lenguajes co-Turing-Reconozibles

$A_{TM} = \{\langle M, w\rangle \ | M$ es una TM que acepta el string w

Esta máquina no es decidible, pero si reconozible.

- Usamos la TM U universal para simular TM M con entrada w
  - Si M acepta w, entonces U acepta $\langle M, w\rangle$
  - Si M rechaza w, entonces U rechaza $\langle M, w\rangle$
  - Si M se cuelga con w, entonces U se cuelga con $\langle M, w\rangle$

Definición: un lenguaje A es co-Turing-reconocible si su complemento $\overline{A}$ es turing reconozible

Decidible $\Leftrightarrow$ *Turing* y $co - Turing - reconicible$

Theorem 9.7.1 *Un lenguaje solo es decidible si es a la vez*

- *Turing reconocible*
- *Co-turing-reconozible*

Demostración

Decidible $\Rightarrow$ *Turing* y $co - Turing - reconicible$ Supongamos que el lenguaje A es decidible. Entonces es Turing-reconozible.

- Como es decidible, $\exists TM$ M que
  - Siempre para
  - Acepta strings $w \in A$
  - Rechaza string $w \notin A$

Definimos la TM M' igual que M, exceptuando que cambiamos los estados de aceptación y rechazo

- M' rechaza cuando M acepta
- M' acepta cuando M rechaza

TM M' siempre se cuelga, puesto que M siempre se cuelga, por lo que M' decide $\overline{A}$

- $\overline{A}$ es Turing-reconozible
- A es co-Turing-reconozible

Turing y $co - Turing - reconicible$ $\Rightarrow$ Decidible

Supongamos que A es TM-reconocible y co-TM-reconocible. Entonces existe:

- TM M reconociendo A
- TM M' reconociendo $\overline{A}$

Para cualquier strin $w \in \Sigma^*$, o bien $w \in A$ o $w \notin A$, por lo que M o M' aceptan w.

Contruimos otra TM D a partir de M y M' de la siguiente forma:

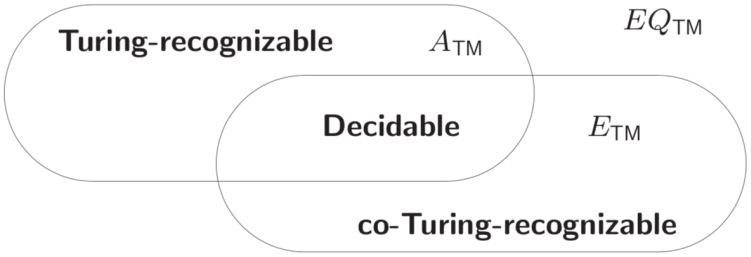
```
D =
    input w
    Se alterna el correr una instruccion de M y M' con entrada w.
    Se espera a que M o M' acepten

    Si M acepta acptar
    Si M' acepta rechazar
```

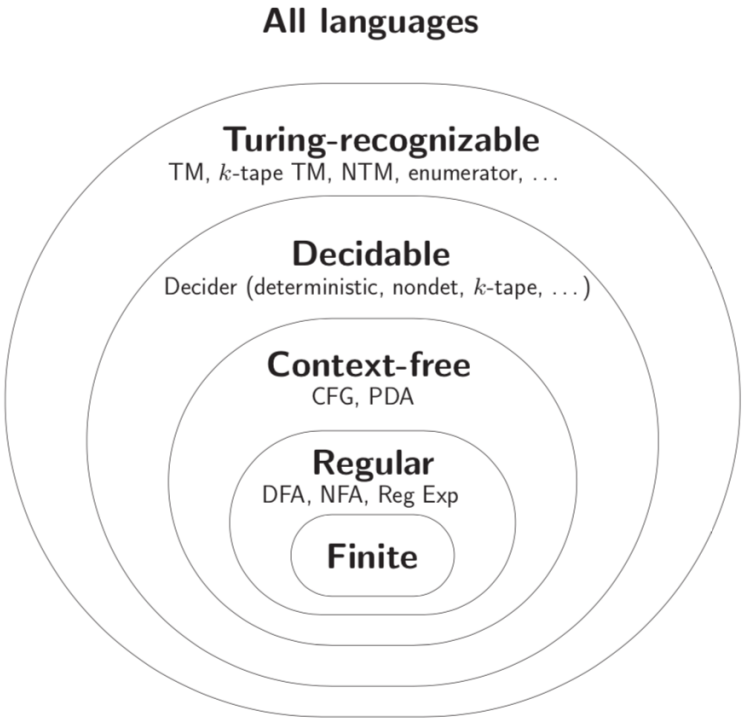
Por lo tanto D decide A, por lo que A es **decidible**.

9.7.1 complemento ATM no es Turing-reconozible

9.7.2 Otros lenguajes No-turing-reconocibles



9.8 Jerarquía de los lenguajes



9.9 Sumario

- Lenguajes decidibles: $A_{DFA}, A_{NFA}, A_{REX}, E_{DFA}, EQ_{DFA}, A_{CFG}, E_{CFG}, CFL$
- TM universal: puede simular cualquier TM para una entrada
- El problema de aceptación de las TM (A_{TM}) es reconocible, pero no decidible

- Sets contables e incontables:
  - Método de diagonalización para demostrar que algunos sets son incontables
  - El set de todas las TMs es contable
  - El set de todos los lenguajes es incontable
  - Algunos lenguajes no son Turing-reconocibles ($\overline{A_{TM}}$)
- Un lenguaje es Turing-reconocible si su complementario es Turing-reconocible
- Decidible $\Leftrightarrow$ Turing-reconocible y co-turing-reconocible.

Chapter 10

Algunos problemas indecidibles

10.1 El problema de los mtes de Thue

10.2 Gramáticas de tipo 0

10.3 El problema de la correspondencia de Post

10.4 Problemas sobre gramáticas incontextuales

10.4.1 Problemas decidibles en DCFL

10.4.2 Problemas indecidibles en CFL

Chapter 11

Reducibilidad

11.1 Introducción

Anteriormente se ha visto que el problema A_{TM} no se puede resolver con TMs. Utilizaremos reducciones para demostrar que otros problemas tampoco lo son.

Las reducciones utilizan dos problemas, A y B.

Definicion 11.1.1 *Si A se reduce a B, entonces cualquier solución de B resuelve A.*

Remarcar que:

- Ya hemos hecho una reducción de A_{NFA} a A_{DFA} para demostrar que A_{NFA} es decidible.
- La definición de recursión por si sola no dice nada sobre como resolver A ni B.
- Si A es reducible a A, A no puede ser más difícil que B
- $p \Rightarrow q$ es equivalente a $\not{p} \Rightarrow \not{q}$
- Suponiendo que A se reduzca a B, tenemos:
  - Si **podemos resolver** B, podemos resolver A
  - Si **no podemos resolver A**, no podemos resolver B
  - Si A es **indecidible**, B es indecidible

A partir de la demostración que hemos hecho de que A_{TM} es indecidible, podemos demostrar la indecibilidad de otros problemas.

Para demostrar que un lenguage L es indecidible, demostraremos que A_{TM} se reduce a L. Si L es decidible, implicaría que A_{TM} es decidible, que es una contradicción con la definción de la propia máquina A_{TM} y por tanto L no es decidible.

- Suponemos que L es decidible

- Creamos una máquina R que decide L
- Usando R como subrutina, creamos otra TM S que decide A_{TM}
- Pero como A_{TM} no es decidible, L tampoco lo es.

11.2 Problema de la parada (Halting problem)

Definimos el problema:

$$HALT_{TM} = \{\langle M, w\rangle | \text{ M es una TM y M se para con string w}\}$$

A_{TM} y $HALT_{TM}$ tienen el mismo univero:

$$\Omega = \{\langle M, w\rangle\} \text{ M es una TM, w es un string }\}$$

Dada una pareja concreta $\langle M, w\rangle \in \Omega$ de TM y string:

- Si M para con entrada w $\rightarrow \langle M, w\rangle \in HALT_{TM}$
- Si M no para con entrada w $\rightarrow \langle M, w\rangle \notin HALT_{TM}$

Theorem 11.2.1 *$HALT_{TM}$ no es decidible*

La idea básica para demostrarlo por contradicción consiste en reducir A_{TM} a $HALT_{TM}$

- Se supone que $\exists TM$ R que decide $HALT_{TM}$
- Se utiliza la TM universal para construir una TM a partir de R que decida A_{TM}

TM U:

```
input <M,w>
    Simular M con input w
    Si M entra en estado aceptador
        aceptar
    Si M entra en restado rechazador
        rechaza
```

La TM U no decide A_{TM} ya que en primer paso M puede colgarse con entrada w. Para solucionarlo primero se correra R con entrada $\langle M, w\rangle$ para saber si es seguro correr M con entrada w:

Maquina S:

```
input <M,w>
    Correr R con entrada <M,w>
    Si R rechaza
        rechazar
    Si R acepta
```

```
            Simular M con entrada w hasta que pare
        Si M acepta
            aceptar
        Else
            Rechazar
```

La TM S siempre para y decide A_{TM}

- S acepta $\langle M, w\rangle \in A_{TM}$ y rechaza $\langle M, w\rangle \notin A_{TM}$

Por lo tanto, decidir A_{TM} se reduce a decidir $HALT_{TM}$, y como A_{TM} es indecidible, $HALT_{TM}$ también lo es.

11.3 El problema del vacío para TM es indecidible

Dada una TM M, reconoce el lenguaje vacío?

$$E_{TM} = \{\langle M\rangle | \text{M es una TM y } L(M) = \emptyset\}$$

- Si M acepta como mínimo un string, $\langle M\rangle \notin E_{TM}$
- Si M no acepta ningún string, $\langle M\rangle \in E_{TM}$

Theorem 11.3.1 *E_{TM} es indecidible.*

Idea de demostración:

- Suponemos que E_{TM} es decidible
- Existe una máquina R que decide E_{TM}
- Se utiliza R para construir otra TM S que decide A_{TM}
- Como A_{TM} no es decidible, E_{TM} tampoco.

Demostración: suponemos $\exists$ TM R que decide E_{TM}

```
    input <M,w>
        Construir TM M1 a partir de M
            M1 := input x
                si x /= w
                    reject
                si x == w
                    correr M(x)
                    Si acepta
                        accept
        Correr R(M1)
        Si R acepta
            reject
```

```
        Si R rechaza
            accept
```

$$\langle M_1 \rangle \notin E_{TM} \Leftrightarrow L(M_1) \neq \emptyset \Leftrightarrow \text{M acepta w} \Leftrightarrow \langle M, w \rangle \in A_{TM}$$

Pero entonces TM S decide A_{TM}, lo que es indecidible, por lo tanto la TM R no puede existir y E_{TM} es indecidible.

11.4 La TM que reconoce lenguages regulares es indecidible

Dada una TM M, reconoce un lenguaje regular?

$$REG_{TM} = \{\langle M \rangle | \text{M es una TM y L(M) es un lenguaje regular}\}$$

- Si L(M) es regular, entonces $\langle M \rangle \in REG_{TM}$
- Si L(M) es no-regular, entonces $\langle M \rangle \notin REG_{TM}$

Theorem 11.4.1 *REG_{TM} es indecidible.*

Idea de demostración:

- Se asume que REG_{TM} es decidible
- R es una TM que decide REG_{TM}
- Se utiliza R para contruir una TM S que decide A_{TM}

Construcción de la TM R:

- Recibe una entrada $\langle M, w \rangle$
- S construye una TM M' usando $\langle M, w \rangle$, de forma que TM M' reconoce un lenguaje regular si y solo si M acepta w
  - Si M no acepta w, entonces reconoce el siguiente lenguaje:
    - $\{0^n 1^n | n \geq 0\}$
- Se construye M' de la siguiente manera:
  - M' acepta automáticamente todos los strings de la forma $\{0^n 1^n | n \geq 0\}$
  - A parte, si M acepta w, entonces M' acepta cualquier otro string.

Demostración: suponiendo que REG_{TM} es decidible, R es una TM que lo decide y usamos R para construir la TM S que decide A_{TM}

```
Contruir M'
M' :=
    input x
        Si x ∈ {o^n 1^n | n ≥ 0}
            accept
        Si x ∉ {o^n 1^n | n ≥ 0}
            reject
Correr R con entrada <M'>
Si R acepta
    accept
Si R rechaza
    reject
```

$\langle M' \rangle \in REG_{TM} \Leftrightarrow \langle M, w \rangle \in A_{TM}$, por lo que S decide A_{TM}, lo que es imposible.

11.5 La equivalencia de 2 TM es indecidible

Dadas 2 TM, reconocen el mismo lenguaje?

$$EQ_{TM} = \{\langle M_1, M_2 \rangle | M_1, M_2 \text{son TMs y} L(M_1) = L(M_2)\}$$

Para cada tupla:

- Si $L(M_1) = L(M_2)$, entonces $\langle M_1, M_2 \rangle \in E!_{TM}$
- Si $L(M_1) \neq L(M_2)$, entonces $\langle M_1, M_2 \rangle \notin E!_{TM}$

Theorem 11.5.1 *EQ_{TM} no es decidible.*

Se reduce E_{TM} a EQ_{TM} de la siguiente forma:

- $M2 = M_\emptyset$ es una TM con $L(M_\emptyset) = \emptyset$
- Una TM que decida EQ_{TM} también puede decidir E_{TM} decidiendo si $\langle M_1, M_\emptyset \in EQ_{TM}$
  - $\langle M_1 \rangle \in E_{TM} \Leftrightarrow \langle M_1, M_\emptyset \rangle \in EQ_{TM}$

Como que E_{TM} es indecidible, EQ_{TM} también ha de serlo.

Más tarde ser verá que EQ_{TM}:

- No es Turing-reconocible
- No es co-turing-reconocible

11.6 Teorema de Rice

Toda propiedad no trivial P de los lenguajes de las TM es indecidible.

Sea P un lenguaje que consite de descripciones de TM tal que

- P contiene algunas, pero no todas, las descripciones de TMs
- Cuando $L(M_1) = L(M_2)$, tenemos $\langle M_1 \rangle \in P$ si y solo si $\langle M_2 \rangle \in P$.
- Por lo que P es indecidible.

11.6.1 Demostración del teorema de Rice

Supongamos que P es decidido por un TM Rp:

- $T_\emptyset$ es una TM que siempre rechaza, por lo qu $L(T_\emptyset) = \emptyset$
- Se asume que $\langle T_\emptyset \rangle \notin P$. Por el caso contrario tenemos $\overline{P}$
- Como hemos asumido que P no es trivial, $\exists$ TM T con $\langle T \rangle \in P$
- Se diseña una TM S que decide A_{TM} usando la capacidad de $Rp's$ para distinguir entre $T_\emptyset$ y T

Máquina S:

```
input <M,w>
    Maquina Mw {
        input x
            Simular M con entrada w. Si se cuelga reject
            Simular T con entrada x. Si acepta accept
    }
    Usar la TM Rp para decidir si <Mw> in P
        Si esta accept
        Si no reject
```

Como que Tm M_w simula T si M acepta w, tenemos que:

- $L(M_w = L(T)$ si M acepta w
- $L(M_w) = \emptyset$ si M no acepta w

Por lo tanto, $\langle M_w \rangle \in P$ si y solo si M acepta w. Y entonce S decide A_{TM}, lo que es imposible. Como conclusión tenemos que P es indecidible.

11.6.2 Historial de computación

Definicion 11.6.1 *Un historial de computación aceptador para una TM M con entrada w es una secuencia de configuraciones $C_1, C_2, ..C_k$ para alguna $k \geq 1$ tal que se cumplen las siguientes propiedades:*

- C_1 es la configuración inicial de M con entrada w
- Cada C_j produce C_{j+1}

- C_k es una configuración aceptadora

Definicion 11.6.2 *Un historial de computación rechazador para M con entrada w, es lo mismo que el aceptador, cambiando que C_k es una configuración que rechaza.*

Propieades de los historiales de computación:

- Los historiales aceptadores y rechazadores son finitos.
- Si M no para con w no existe ningún historial, ni aceptador ni rechazador
- Es útil tanto para TMs deterministas (un único historial) como no deterministas (varios historiales)
- $\langle M, w\rangle \notin A_{TM}$ es equivalente a
  - $\nexists$ historial de computación aceptador $C_1, ..., C_k$ para M con entrada w
  - Todos los historiales $C_1, ..., C_k$ son no aceptadores para M con entrada w

11.7 Reducciones con mapeo

11.7.1 Funciones computables

Supongamos que tenemos 2 lenguajes:

- A esta definido sobre el alfabeto Σ_1, tal que $A \subseteq \Sigma_1^*$
- B esta definido sobre el alfabeto Σ_2, tal que $B \subseteq \Sigma_2^*$

A será reducible a B, si podemos usar una caja negra de B para construir un algoritmo que resuelva A.

Definicion 11.7.1 *Una función $f : \Sigma_1^* \to \Sigma_2^*$ es una función computable si hay alguna TM M, que para cada entrada $w \in \Sigma_1^*$ para con $f(w) \in \Sigma_2^*$ en su cinta.*

Con funciones computables podemos transformar de una TM a otra. Ejemplo:

Función T:

```
input w
    Si w = <M>, donde M es una TM
        Construir <M'>, donde M' es una TM tal que
            L(M') = L(M), pero M' nunca trata de mover
            la cabeza del extremo izquierdo de la tapa.
```

Si A es reducible a través de un mapeo a B:

$$A \leq_m B$$

Si existe una función computable:

$$f : \Sigma_1^* \to \Sigma_2^*$$

Tal que, para cada $w \in \Sigma_1^*, w \in A \Leftrightarrow f(w) \in B$

11.7.2 Ejemplo: reducción de ATM a HALT TM

Anteriormente se había demostrado que $HALT_{TM}$ era indecidible reduciendo A_{TM} a ella.

Para demostrar que $A_{TM} \leq HALT_{TM}$ necesitamos una función computable que con entrada $\langle M, w \rangle$, que es una instancia del problema de aceptación,genere una salida $f(\langle M, w \rangle) = \langle M', w' \rangle$ que será una instancia de HALT.

$$\langle M, w \rangle \in A_{TM} \Leftrightarrow f(\langle M, w \rangle) = \langle M', w' \rangle \in HALT_{TM}$$

La siguiente TM F computa esa función:

```
input <M,w>
    Contruir TM M'
        M' := input x
            Correr M con entrada x
            Si acepta
                accept
            Si rechaza
                reject
    output <M',w'>
```

11.7.3 Decidability obeys ¡= Ordering

Decidability obeys $\leq_m$ *Ordering*

Theorem 11.7.1 *Si* $A \leq_m B$ *y* B *es decidible, entonces* A *es decidible*

Demostración:

- M_B es una TM que decide B
- f es una función de reducción de A a B

Tenemos la siguiente TM M_A:

```
input w
    Computar f(w)
    Correr MB con entrada f(w) y dar el mismo resultado
```

Como f es una función de reducción, $w \in A \Leftrightarrow f(w) \in B$

- Si $w \in A$, entonces $f(w) \in B$, por lo que M_B y M_A aceptan

- Si $w \notin A$, entonces $f(w) \notin B$, por lo que M_B y M_A rechazan

Por lo tanto M_A decide A.

11.7.4 Indecidibilidad obeys ¡= Ordering

Undecidability *obeys* $\leq_m$ *Ordering*

Corollary 11.7.1.1 *Si $A \leq_m B$ y A es indecidible, entonces B es indecidible también.*

Demostración: Si el lenguaje A es indecidible y B decidible, contracide el teorema anterior.

$\overline{A} = \Sigma_1^* - A$ y $\overline{B} = \Sigma_2^* - B$

Si $A \leq_m B$, entonces $\overline{A} \leq_m \overline{B}$

11.7.5 ETM no es Turing-reconocible

Tomando como base $\overline{A_{TM}}$, que no es Turing-reconocible.

Se reduce $\overline{A_{TM}} \leq_m E_{TM}$. Se define la función $f(\langle M, w\rangle) = \langle M'\rangle$, donde M' es la TM siguiente:

```
input x
    Ignorar input x, y correr M con entrada w
    Si M acepta
        aceptar
```

Si M acepta w, entonces $L(M') = \Sigma^*$.

Si M no acepta w, entonces $L(M') = \emptyset$

Por lo que $\langle M, w\rangle \in \overline{A}_{TM} \Leftrightarrow f(\langle M, w\rangle) = \langle M'\rangle \in E_{TM}$

11.7.6 EQTM no es Turing-reconocible

$EQ_{TM} = \{\langle M_1, M_2\}|M_1, M_2$ son TMs con $L(M_1) = L(M_2)$

Demostración: Se reduce $\overline{A_{TM}} \leq_m EQ_{TM}$

M1 = reject con todos los inputs.

M2=

```
input x
    Ignorar input x, y correr M con entrada w
    Si M acepta, aceptar
```

$L(M_1) = \emptyset$

Si M acepta w, entonces $L(M_2) = \Sigma^*$

Si M no acepta w, entonces $L(M_2) = \emptyset$

Por lo tanto, $\langle M, w\rangle \in \overline{A_{TM}} \Leftrightarrow f(\langle M, w\rangle) = \langle M_1, M_2\rangle \in EQ_{TM}$

Como $\overline{A_{TM}}$ no es Turing-reconocible, EQ_{TM} tampoco.

11.7.7 EQTM no es co-turing-reconocible

Se reduce $A_{TM} \leq_m EQ_{TM}$

Se define la función de reducción $f(\langle M, w\rangle) = \langle M_1, M_2\rangle$ donde:

M_1=

```
    input x
        accept
```

M_2=

```
    input x
        Ignorar input, correr M con input w
        Si M acepta
            accept
```

$L(M_1) = \Sigma^*$

Si M acepta w, entonces $L(M_2) = \Sigma^*$

Si M no acepta w, entonce $L(M_2) = \emptyset$

Como A_{TM} no es co-Turing-reconozible, EQ_{TM} no es co-Turing-reconozible.

Chapter 12

Ejercicios

12.1 Tema 1

1.Formalitzeu els seguents llenguatges utilitzant la notació clàssica de conjunts, com a parell variable de mot i propietat sobre aquest. Feu servir quantificadors universals i existencials, operadors booleans i les notacions sobre mides de mots que hem introdüit.

(a) Llenguatge dels mots sobre $\{a, b\}$ que contenen el submot ab.

$\{w \in \{a, b\}^* | w_{|ab|} \geq 1\}$

(b) Llenguatge dels mots sobre a,b tals que a la dreta de cada submot ab hi ha algun submot ba.

(c) Llenguatge dels mots sobre a, b que contenen el submot ab i el submot ba.

(d) Llenguatge dels mots sobre a, b, c tals que entre cada dues b's hi ha alguna a.

(e) Llenguatge dels mots sobre a, b tal que tota ocurr'encia d'una b és en posició parella (el primer símbol d'un mot ocupa la posició 1).

(f) Llenguatge dels mots sobre a,b amb algun prefix amb més b's o igual que a's.

(g) Llenguatge dels mots sobre a, b tals que qualsevol prefix seu té més b's o igual que a's.

(h) Llenguatge dels mots sobre a, b amb algun prefix de mida parella amb més b's o igual que a's.

(i) Llenguatge dels mots sobre a, b tals que qualsevol prefix seu de mida parella té més b's o igual que a's.

(j) Llenguatge dels mots sobre a, b que tenen un prefix i un sufix id'entics de mida més gran o igual que 0 i més petita o igual que la mida del propi mot.

2. Argumenteu si són certes (amb una justificació) o falses (amb un contraexemple) les afirmaciones següents sobre mots x, y, z i llenguatges A,B,C en general

(a) $xy = yx$

(b) $xy = xz \Rightarrow y = z$

(c) $A(BC) = (AB)C$

(d) $AB = BA$

(e) $A \neq \emptyset \wedge AB = AC \Rightarrow B = C$

(f) $A \neq \emptyset \wedge B \neq \emptyset \wedge AB = CD \wedge (\forall u \in A, v \in C | u| = |v|) \Rightarrow A = C \wedge B = D$

(g) $(A \cup B)C = AC \cup BC$ i $A(B \cup C) = AB \cup AC$

(h) $(A \cap B)C \subseteq AC \cap BC$ i $A(B \cap C) \subseteq AB \cap AC$

(i) $(A \cap B)C \subseteq AC \cap BC$ i $A(B \cap C) \supseteq AB \cap AC$

4. Argumenteu si són certes (amb una justificació) o falses (amb un contraexemple) les següents afirmacions en general

b) Cert

Teoria: La concatenació de dos llenguatges es el llenguatge format per tots els mots obtinguts concatenant un mot del primer llenguatge amb un del segon.

$(L_1 L_2)^R = L_2^R L_1^R$

$L_1 L_2 = \{xy \,|\, x \in L_1 \bigwedge y \in L_2\}$

$\{(x, y)\}^R | x \in L_1 \bigwedge y \in L_2$

$\{y^R x^R | x^R \in L_1^R \bigwedge y^R \in L_2^R\} = L_2^R L_1^R$

2. Argumenteu si són certes (amb una justificació) o falses (amb un contraexemple) les següents afirmacions sobre els mots x, y,z i llenguatges A, B, C en general)

b) $xy = xz => y = z$

Cert

$xy = xz$ són el mateix mot

$|xy| = |xz| = |x| + |y| = |x| + |z|$

$|x| = |x|$

$|xy| - |x| = |xz| - |x|$

$y = z$

2 e) Fals

h) $(A \cap B)C \subseteq AC \cap BC$ **i** $A(B \cap C) \subseteq AB \cap AC$

$(A \cap B)C = \{xy | x \in (A \cap B) \bigwedge y \in C\}$

$\{xy | x \in A \wedge x \in B \wedge y \in C \; rbrace = \{xy | (x | inA \wedge y \in C) \wedge (x | inA \wedge y \in B)\}$

$AC \cap BC = \{xy | x \in A \wedge y \in C\} \cap \{xy | x \in B \wedge y \in C\} = \{xy | x \in A \wedge y \in C \wedge \exists u, v | u \in B \wedge v \in C \wedge w = uv\}$

3. Argumenteu si són certes (amb una justificació) o falses (amb un contraexemple) les següents afirmacions en general

b) Fals

$L_1^* L_2^* \subseteq (L_1 L_2)^*$ Cert

Según la profe:

$w \in L_1^* L_2^* => \exists n, m \geq 0 we L_1^n L_2^m => \{xy | x \in A \wedge x \in B\}$

$w = u_1, u_2 ... u_n, v_1, ... v_m$

- $u_1 \cap L_1$
- $u_n \cap L_1$
- $v_1 \cap L_2$
- $v_m \cap L_2$

$w \in (L_1 L_2)^* => \exists r \geq 0 w \in (L_1 L_2)^r => w = x_1 ... x_r \in L_1 L_2$

3e)

$(L_1^* \cup L_2^*) \subseteq (L_1 \cup L_2)^*$

$L_1^* = \underset{n \geq 0}{U}$

3h)

$(L_1 \cap L_2)^* \supseteq (L_1^* \cap L_2^*))$

3k)

$\overline{L^*} \supseteq \overline{L} \supseteq \overline{L}^*$

Contraexemple

Si $x \in (L)^* => x \in \overline{L}$?

$x \in (\overline{L})^* sempre(perdefestrelladekleene)$

Però, si $y \in L(L = \{\lambda\})$ llavors $\lambda \notin \overline{L}$

3n)

$L^+ L^+ \supseteq L^+$ Fals

3t

$L = L^2 \Rightarrow (L = L^*) \vee (L = 0)$

$L = L^2 \vee (L = 0)$

$L = L^2 => L = L^2 L$

$L = L^4 = L^2 L^2 = LL = L^2$

$L^* = L^0$

4c)

$(L_1 \cup L_2)^R = L_1^R \cup L_2^R$

$L_1 \cup L_2 = \{w | w \in L_1 \vee w \in L_2\}$

$(L_1 \cup L_2)^R = \{w^R | w \in L_1 \vee w \in L_2\}$

$L^R\{u | u \in L^R\}$

$L_1^R \cup L_2^R = \{u | u \in L_1^R \vee u \in L_2^R\} \ - u = w^R \rightarrow L_1^R \cup L_2^R = \{w^R | w^R \in L_1^R \vee w^R \in L_2^R\}$

$L_1^R \cup L_2^R = \{w^R | w \in L_1 \vee w \in L_2\}$

4e) $\overline{L}^R = \overline{L^R}$

4g) $(L_1 L_2)^R = L_1^R L_2^R = L_1^R L_2^R \Rightarrow L1 = L2$?

Fals. Contraexemple

$L1 = a$

$L2 = \wedge = \{\lambda\}$

$(L_1 L_2)^R = \{a\}^R = \{a\}$

$L_1^R L_2^R = \{a\}^R\{\lambda\} = \{a\}$

Però, $L_1 \neq L_2$

5b)

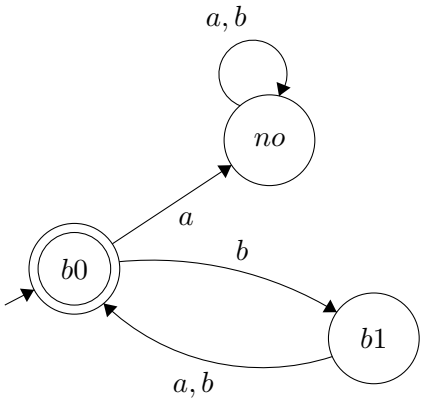
σ

Chapter 13

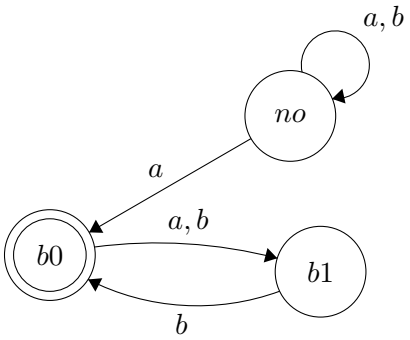
Autòmats finits

13. Obtén el DFA mínimo para $L = \{w \in \{a,b\}^* | \forall w1, w2(w = w1aw2 \Rightarrow |w1|_b \in \dot{2}\}$ y calcula explíticamente el NFA reverso A^R, determinízalo y minimízalo

DFA:

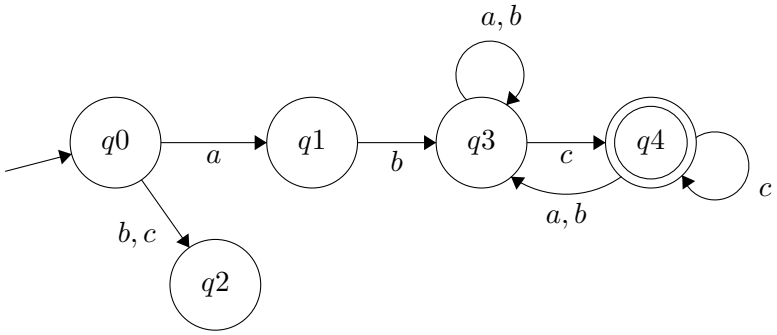


NFA:



17. Obten el DFA mínimo para $L = \{axbyc | x, y \in \{a,b,c\}^*\}$ y, dado el morfismo definido por $\sigma(a) = ab$, $\sigma(b) = b$, $\sigma(c) = \lambda$, calcula explícitamente el NFA imagen $\sigma(A)$, determinízalo y minimízalo

REVISAR ESTE DFA, ESTA FATAL.



19. Sea A el DFA mínimo que reconoce los números binarios múltiplos de 3. Calcula $\sigma^{-1}(A)$ tomando como σ los morfismos:

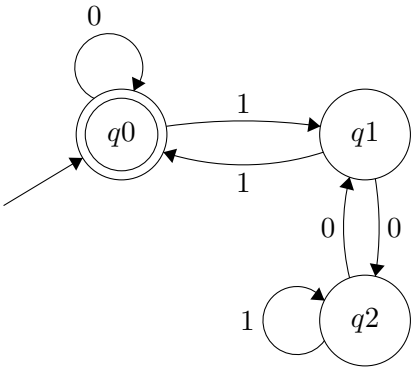
c) $\sigma(a) = 00$, $\sigma(b) = 11$, $\sigma(c) = \lambda$

$L(A) = \{w \in \{0,1\}^* | value_2(w) \in \dot{3}\}$

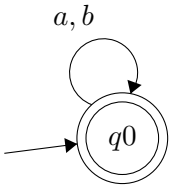
$A =< Q, \{0,1\}, \delta.q_0, F >$

$\sigma^-1(A) =< Q, \{a,b,c\}, \delta^{'}, q_0, F$

$\delta^{'}(q,x) = \delta(q, \sigma(x)$



$\sigma^{-1}(L) = \{w\{a,b,c\}^* | 0(w) \in L\}$



20. Diseña un algoritmo de coste razonable para encontrar los estados no accesibles de un DFA de entrada

BFS + recorrido de los Vertices visitados

21. Diseña un algoritmo de coste razonable para decidir si un DFA de entrada acepta alguna palabra

Queremos saber si alguna palabra, ∃ un nodo q aceptador que es accesible desde el estado inicial q0.

Q accesibles ∩ F $\neq 0$

$$T(n) = \begin{cases} \text{O}(1) & \text{si } q \in F \text{ o } F == 0 \\ \text{O}(|Q|\Sigma|) \end{cases}$$

29

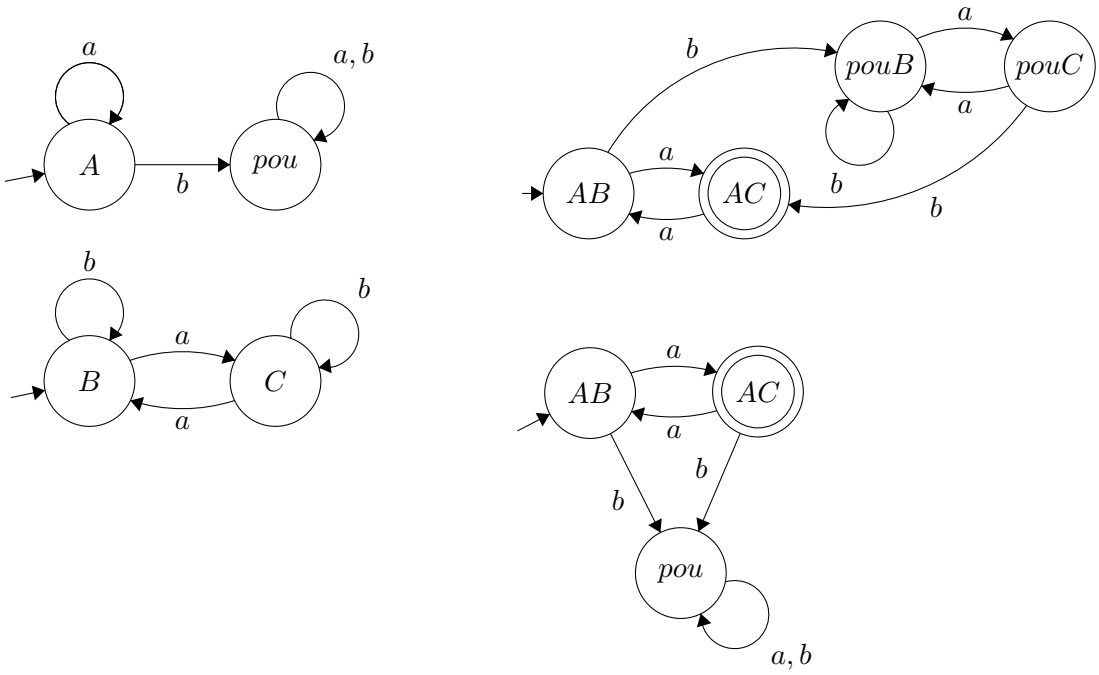
a) $A_1 \cap A_2$ és un DFA minim

Falso: contraejemplo

$A_1 = \{w \in \{a,b\}^* || w|_b = 0\}$

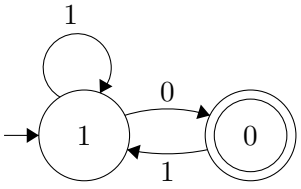
$A_2 = \{w \in \{a,b\}^* || w|_a \in \dot{2}\}$

CREO QUE EN EL SEGUNDO DFA ESTA DEL REVÉS LO DE PAR E IMPAR DE A

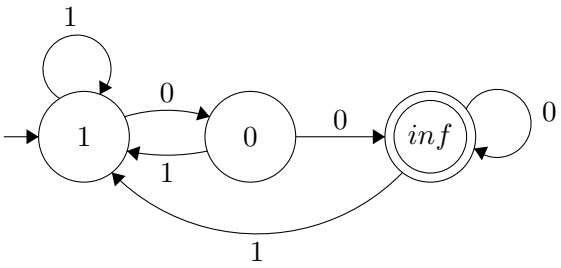


32. Dado un natural n definimos $L_n = \{w \in \{0,1\}^* | \exists k valor_2(2) = k*2^n\}$. Justifíca que cualquier L_n es regular. ¿Cuántos estados tiene el DFA mínimo que reconoce L_n

$L1 = k * 2$



$L2 = k * 4$


1
0
1
0
0
inf
0
1
1

Chapter 14

Gramatiques incontextuals

3. Justifica l'ambigüitat de les CFG següents

c)

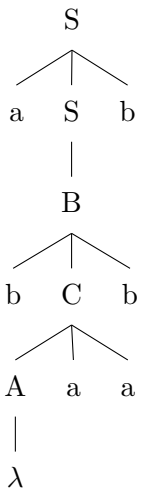
$A- > aSb|B$

$B \rightarrow bAa|bCb|\lambda$

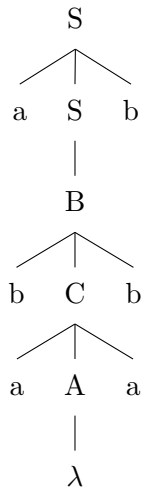
$A \rightarrow aAbA|bAaA|\lambda$

$C \rightarrow Aaa|aAa|aaA$

abbbbb



abaabb



d)

$S \to AaBa|ABaA|ACA|AbabA$

$B \to bb$

$C \to bB$

$A \to aA|bA|\lambda$

$B \stackrel{*}{=}> w \in \Sigma^{'} \Leftrightarrow w = bb$

$C \stackrel{*}{=}> w \in \Sigma^{*} \Leftrightarrow w = bbb$

$A \stackrel{*}{=}> w \in \Sigma^{*} \Leftrightarrow w = \{a, b\}^{*}$

$S \stackrel{*}{=}> \in \Sigma^{*} \Leftrightarrow w \in (a, b)^{*}abb(a+b)^{*} + (a+b)^{*}bba(a+b)^{*} + (a+b) * bbb(a+b) * + (a+b)^{*}bab(a+b)^{*}$

5)

G_1

Chapter 15

Problemas Tema 8

5. Sea B un conjunto semidecidible y sea C un conjunto que cumple $C = \{x|\exists y\langle x, y\rangle \in B\}$. Demuestra que C es semidecidible. Queremos demostrar B sem-dec y $C\{x|\exists y < x, y > B\} \Rightarrow C$

Sea $M_B : L(M_B) = B$

```
Entrada x
  t = 0
  ejecutar indef
    para y = 0 hasta t
    si (M_B⟨x,y⟩ acepta en t pasos)
      Entonces ACEPTAR
    t ++
```

b) $C \subseteq Ndec \Leftrightarrow \exists f computable, total, injectiva, creciente, tqIm(f) = C$

$\Leftarrow$ Sea Mx la Tm que computa f

```
M: Entrada y
  i = 1
  z = run Mx(i)
  while z /= y
    ++ 1
    z = run Mx(i)
  ACEPTAR
```

$L(M) = \{y|\exists x f(x) = y\} = Im(f)$

$\Rightarrow ComCdec \Rightarrow \exists M_x(y) \downarrow \forall y, quedecideixC$

$C = L(M_x) \uparrow = \{w_1, w_2..., w_1, ...\} \subseteq \mathbb{N}$

$\exists f Im(f) = C$ Total, computable, injectiva

$F(x) =$ i-esim z tal que

$M_x(z) s'atura <\leq tpassos$

8. Sea f una función computable e inyectiva. ¿Es f^{-1} computable e inyectiva?

9.

$f : \mathbb{N} \to \mathbb{N}$ decreciente. ¿Es f computable?

$$dom(f) es finit \Rightarrow dom(f) es decidible$$

$$dom(f) es finito \Rightarrow f \subseteq \mathbb{N} \times \mathbb{N} es finito$$

Chapter 16

Problemas tema 9

Teorema de Rice

Sea A un conjunto de indices:

$$\text{A es decidible} \Leftrightarrow A = \emptyset \vee A = \mathbb{N}$$

En el caso que $A \neq \emptyset$ y $A \neq \mathbb{N}$, si $i \in A$ donde $dom\varphi_i = \emptyset$, entonces A no es semi-decidible.

1. Classifica com a decidibles, indecidibles però semidecidibles, o no semidecidibles, els conjunts següents.

k) $\{p|\varphi$ **es injectiva y total**$\}$

A conjunt ind $\Leftrightarrow \forall x, y(x \in A \wedge \varphi_x = \varphi_y \rightarrow y \in A)$

A es recursio $A = \emptyset$ o $A = \mathbb{N}$

C = $\{p\ \varphi_p es$

F identitat (inj y total) : $\exists x|M_x$ computa f identitat y $x \in C \rightarrow C \notin \emptyset$

F vacía (computable): $\exists y|M_y$:

```
entrada y:
    infiniteloop
```

Por lo que $Dom(\varphi y = 0) \rightarrow y \notin C \rightarrow C \neq \mathbb{N}$

No sirve porque la función vacia no esta en el conjunto y por tanto no es semidecidible.

Se hace una reducción a $\overline{K} \leq \{p|\varphi_p$ inyectiva y total

```
input y
    simular maquina x para y pasos
    si para
        output y
    si no para
        infiniteloop
```

$x \in \overline{K} : \forall y M_x(x) \uparrow$

$\forall y M_p(y) = y \Rightarrow \varphi p$ es inyectiva y total $\Rightarrow p \in C$

$x \notin \overline{K} : \exists t M_x(x) \downarrow$ en exactamente t pasos

$\exists t \forall y \geq t \ M_x(x) \downarrow$ en y pasos

$\exists t \forall y \geq t \ M_p(y) \uparrow \Rightarrow Dom(\varphi_p) \subseteq \{y | y \leq t\} \Rightarrow \varphi_p$ es total $\Rightarrow p \notin C$

otro apartado

$\{p|$ p es finito$\}$

$B = \{p | \{w | |w| \leq n\} \subseteq L_p\}$ es semidecidible

$L(M_p) = \{w | |w| < n\}$ no es decidible

Sea $x, |x| = m \ M_x(x) \downarrow$

Definimos M:

```
entrada p
    para cada x, |x| <= n do
        simular Mp(x)
        si Mp(x) rechaza
            Reject
    end

    Aceptar
```

$L(M) = B$ es semidecidible. Soy capaz de construir una máquina que reconoce exactamente ese lenguaje. Sea M la máquina, el lenguaje reconocido por M es B.

$p \in B \Rightarrow \{w | |w| \leq n\} \subseteq L(M_p) \Rightarrow M(p)$ acepta $p \in L(M)$

$p \notin B \Rightarrow \exists w, |w| \leq n \ w \notin L(M_p)$

$M_p(w) \uparrow \Rightarrow p \notin L(M)$

$M_p(w) \downarrow \Rightarrow M(p) \downarrow$ rebutja $p \notin L(M)$

Demostrar que no es semidecidible

$p \in A$ y $L_p = L_q \Rightarrow$ A es un conjunto de indices

$p \in A \rightarrow L(Mp)$ es finito $= q \in A$

otro

$L = \{p | \varphi_p$ creciente y total$\}$ No es semi-decidible

$x \mapsto p$

```
input y
    Si (Mx(x) no acepta en y pasos)
        output y
    inifiniteloop
```

$x \in \overline{K} \Leftrightarrow p \in L$

$x \in \overline{K} \Rightarrow M_x(x) \uparrow \Rightarrow \forall y M_p(y) = y \Rightarrow p \in L$

$x \notin \overline{K} \Rightarrow \exists t \forall y \geq y \ M_x(x) \downarrow$ en y pasos $\exists t \forall y \geq t M_p(y) \uparrow \Rightarrow \varphi_p$ no es total $\Rightarrow p \notin L$

n) $L = \{p | \varphi_p$ es total y estrictamente decreciente$\}$

Si es estrictamente decreciente el dominio es finito, si es total el dominio es infinito. La propiedad es imposible, no hay función que lo satisfaga.

$\varphi_p : \mathbb{N} \to \mathbb{N}$

$f : \mathbb{N} \to \mathbb{N}$ estrictamente decreciente $\Rightarrow Dom(f) = finito \Rightarrow$ f no es total

$$f(x) > f(x+1)$$

2. Classifica com a decidibles, indecidibles però semidecidibles, o no semidecidibles, els conjunts següents.

a) $L = \{\langle p, q \rangle | \forall_z ((M_p(z) \downarrow \wedge M_q(z) \uparrow) \vee (M_p(z) \uparrow \wedge M_q(z) \downarrow))\}$ No es semi-decidible

$\overline{K} \leq L$

$x \mapsto \langle p, 1 \rangle$

Maquina p

```
    Entrada z
        parar
```

Maquina q:

```
    Entrada z
        Ejecutar Mx(x)
        Parar
```

$x \in \overline{K} \Rightarrow M_X(x) \uparrow \Rightarrow \forall z (M_q(z) \uparrow \wedge M_p(z) \downarrow) \Rightarrow \langle p, q \rangle \in L$

$x \notin \overline{K} \Rightarrow M_X(x) \downarrow \Rightarrow \forall z (M_p(z) \downarrow \wedge M_q(z) \downarrow) \Rightarrow \langle p, q \rangle \notin L$

b) $L = \{\langle p, z \rangle | \exists y M_p(y) = z\}$ Semidecidible, pero no decidible

```
    input <p,z>
        t := 0
        while(1){
            from y=0 to t{
                if(Mp(y) == z
                    accept
            }
            t ++
        }
```

Por lo que $L \in Semi - dec$

$K \leq L$

$x \mapsto \langle p, 1 \rangle$ a la z se le pone un número, porque es constante.

```
input x
    corre Mx(x)
    output 1
```

$x \in K \Rightarrow M_x(x) \downarrow \Rightarrow M_p(y) = q \forall y \Rightarrow \langle p, 1 \rangle \in L$

$x \notin K \Rightarrow M_x(x) \uparrow \Rightarrow M_p(y) \uparrow \Rightarrow \langle p, 1 \rangle \notin L$

Por lo que $L \notin Dec$

d) $\{p | L_p$ es incontextual $\}$

Si L_p es un CFL, entonces existe un CGF que llamaremos G que crea el lenguaje L_p.

Tenemos el siguiente lenguaje:

$$A_{CFG} = \{\langle G, w \rangle | \text{G es un CGF que genera } w\}$$

Se puede construir un algoritmo en una TM para decidir este lenguaje:

```
N:= Buscar(no-terminales que generan λ)
  Si el estado inicial S esta entre ellos, entonces se acepta
  Else para cada B en N
    Por cada produccion del tipo C -> xBy (xy ≠ λ)
      Add produccion C -> xy
    Eliminar las producciones A -> λ
    // Ahora en N tenemos una gramatica G' tal que
    // L(G') = L(G) − λ. Solo falta comprobar si
    // w es generable por G
    DFS en las derivaciones izquierdas de G', comenzando por S
      Si se consigue derivar w, aceptamos
    //Para hacerlo mas eficiente, si una palabra derivada es mas
    //larga que w, paramos.
    //Para evitar bucles de la forma S->X->S
    guardamos las palabras ya generadas
```

Otra opción es transformar el CFG en la forma normal de Chomsky, de forma que la palabra w se derivaría en $2 * |w| - 1$ pasos, por lo que la máquina se pararía.

Por lo anterior, tenemos que decidir si una palabra w es generable por CFG es un problema decidible sobre la máquina A_{CFG} Entonces si para cada palabra w, si le pasamos a la máquina A_{CFG} la entrada entrada $\langle G, w \rangle$, tenemos:

```
entrada w
    Correr la maquina
        If (TM A_CFG acepta) aceptamos
        Else no aceptamos
```

Por lo tanto es posible decidir si el lenguaje es generado por un CFG a través de una TM, y por tanto es decidible.

CORRECCIÓN. No es ni semi-decidible. Dentro de Lp pueden estar las gramáticas incontextuales, pero cualquier otra cosa, también el conjunto vacío. No podemos hacer un TM que nos diga si es incontextual.

f) $C = \{p | Dom(\varphi) \in Dec\}$

El indice de la función vacia esta dentro. C no es semi-decidible ni decidible.

h) $C = \{p | Dom(\varphi) \notin Semi - dec\}$

C es el conjunto vacio.

i) $C = \{p | Dom(\varphi) \notin Dec\} = \text{J}$

$\overline{K \leq J}$

$x \mapsto p$

$x \in \overline{K} \Rightarrow M_x(x) \uparrow \Rightarrow Im(\varphi_p) \notin Dec$

$x \notin \overline{K} \Rightarrow M_x(x) \downarrow \Rightarrow Im(\varphi_p) \in Dec$

$x \mapsto p$

```
entrada y
si no (Mx(x) se para en y pasos) do
    si Mi(y) acepta
        retorna y //y pertenece a K
    si no infiniteloop

si no infiniteloop
```

$x \in \overline{K} \Rightarrow M_x(x) \uparrow \forall y(M_p(y) = y \Leftrightarrow y \in K) \Rightarrow Im(\varphi_p) = K$ no es decidible

$x \notin \overline{K} \Rightarrow Mx(x) \downarrow \exists$

Chapter 17

Examenes

17.1 2016

n	bin	reverse bin dec	n.reverse bin	n.reverse dec
3	11	3	1111	15
6	110			
9	1001	9	1001 1001	153
12	1100			
15	1111	15	1111 1111	255
18	10010			
21	10101	21	10101 10101	693
24	11000			
27	11011	27	11011 11011	891
30	11110			

Table 17.1: Multiplos de 3 y palindromos de ellos