


Teoría de la Computación

# Lenguajes y autómatas finitos

## Cadenas, lenguajes, DFA y la construcción del producto

**Gerson Garrido**

Semana 1

UNIVERSIDAD NACIONAL DE INGENIERÍA

# Hoy

**Sesión A**

1. Las tres preguntas del curso
2. Cadenas e inducción sobre cadenas
3. Lenguajes y sus operaciones
4. De problemas a lenguajes
5. Hay más lenguajes que programas

**Sesión B**

6. Máquinas con memoria fija: el DFA
7. Diseñar un DFA: qué recordar
8. Demostrar que es correcto
9. La construcción del producto
10. Complemento

**Lectura**

Apuntes de la semana 1. Apoyo: Sipser, cap. 0 (cadenas y lenguajes) y sec. 1.1; Erickson, *Models of Computation*, notas 1 y 3. Código en `code/week-01`.

SECCIÓN 01

# Las tres preguntas del curso

01

# De dónde venimos

- En Algoritmos resolvieron problemas: el camino más corto, el $k$-ésimo menor.
- Y quedó una sospecha: para algunos problemas nadie conoce un algoritmo rápido.

1. **¿Qué es calcular?** Una definición matemática de máquina, para poder hablar de *todas* las máquinas.
2. **¿Qué no se puede calcular?** Problemas bien definidos que ningún programa resuelve.
3. **¿Qué cuesta demasiado?** Lo que se puede resolver, pero no en un tiempo razonable. P y NP.

**El método del curso**

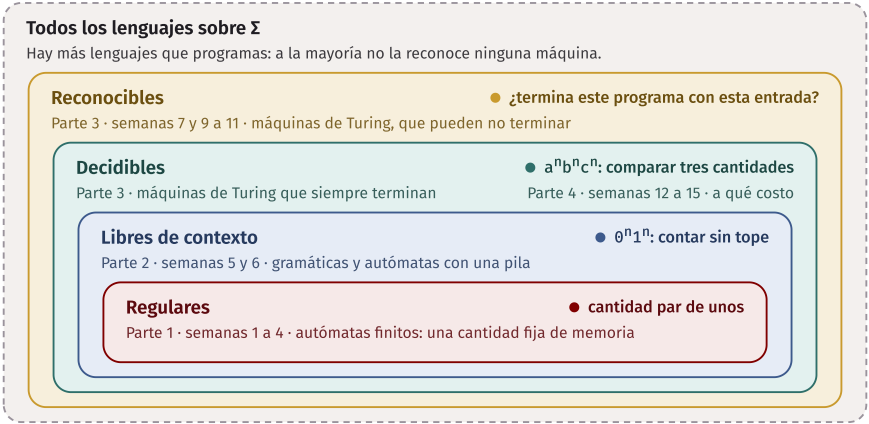
Cada modelo se define, se programa y se le busca el límite: algo que no puede hacer, con su demostración.

# El mapa: cuatro partes

| Parte | Semanas | El modelo | Su límite |
|---|---|---|---|
| 1. Lenguajes regulares | 1 a 4 | Autómatas finitos | No cuentan sin tope |
| 2. Lenguajes libres de contexto | 5 y 6 | Gramáticas y pila | No comparan tres cantidades |
| 3. Computabilidad | 7, 9 a 11 | Máquinas de Turing | Problemas sin solución |
| 4. Complejidad | 12 a 15 | Tiempo y espacio medidos | Lo que cuesta demasiado |

- En teoría: se construyen las máquinas y se demuestran los teoremas.
- En laboratorio: las mismas construcciones en C++, contra el juez.
- Hoy: el vocabulario de todo el curso y la primera máquina.

# El mapa: una clase dentro de otra



SECCIÓN 02

# Cadenas

02



# Alfabetos y cadenas

**Alfabeto** *(alphabet)*

Conjunto finito y no vacío de **símbolos**. Se escribe $\Sigma$.

**Cadena** *(string)*

Secuencia **finita** de símbolos de $\Sigma$.

- $\Sigma = \{0, 1\}$: `0110`, `1`, `000`
- $\Sigma = \{a, b\}$: `abba`
- **Longitud:** $|0110| = 4$
- **Cadena vacía** $\varepsilon$: la única de longitud 0. En C++, `""`.
- $\Sigma^*$: **todas** las cadenas sobre $\Sigma$.

**Ojo**

$\varepsilon$ no es un símbolo del alfabeto: es el nombre de una cadena que no se ve.

# $\Sigma^*$ es infinito, y se puede listar

terminal

```
$ build/strings_demo
Σ = {0, 1}. Las cadenas de Σ* con longitud a lo sumo 3, en orden:
  ε 0 1 00 01 10 11 000 001 010 011 100 101 110 111

 n   cadenas de longitud n   cadenas de longitud ≤ n
 0                         1                         1
 1                         2                         3
 2                         4                         7
 3                         8                        15
 4                        16                        31
```

Las primeras líneas de la salida.

- De longitud $n$: $|\Sigma|^n$.
- Binarias de longitud $\leq n$: $2^{n+1} - 1$.
- Cada cadena es finita.
- En este orden, cada cadena ocupa una posición finita: `101` es la número 13.

# Concatenación

**Concatenación** *(concatenation)*

$uv$ es la cadena $u$ seguida de $v$. Si $u = \texttt{ab}$ y $v = \texttt{ba}$: $uv = \texttt{abba}$, $vu = \texttt{baab}$.

- $\varepsilon$ es el neutro: $\varepsilon w = w\varepsilon = w$.
- Es asociativa: $(uv)w = u(vw)$.
- No es conmutativa: $\texttt{abba} \neq \texttt{baab}$.
- Repetición: $w^k$ son $k$ copias de $w$, y $w^0 = \varepsilon$. $\quad 0^3 1^2 = \texttt{00011}$.
- Reverso: $(\texttt{abb})^R = \texttt{bba}$.

**Los símbolos no son números**

$0^3$ es la cadena $\texttt{000}$, no cero al cubo.

# Prefijos, sufijos y subcadenas

- $x$ es **prefijo** de $w$ si $w = xy$ para alguna $y$.
- $x$ es **sufijo** de $w$ si $w = yx$ para alguna $y$.
- $x$ es **subcadena** de $w$ si $w = yxz$ para algunas $y, z$. Los símbolos van *seguidos*.

terminal

```
w = abba, |w| = 4
prefijos (5):   {ε, a, ab, abb, abba}
sufijos (5):    {ε, a, ba, bba, abba}
subcadenas (9): {ε, a, b, ab, ba, bb, abb, bba, abba}
```

- $\varepsilon$ y $w$ siempre están en las tres listas. `aa` no es subcadena de `abba`.
- Longitud $n$: $n + 1$ prefijos, $n + 1$ sufijos, a lo sumo $n(n + 1)/2 + 1$ subcadenas.

# Las cadenas se definen por casos

Toda cadena $w$ es de una de dos formas:

- $w = \varepsilon$, o
- $w = xa$, con $x$ una cadena y $a$ un símbolo: el último.

**Longitud**

$|\varepsilon| = 0$
$|xa| = |x| + 1$

**Concatenación**

$u \cdot \varepsilon = u$
$u \cdot (xa) = (u \cdot x)a$

**Inducción sobre la longitud**

Caso base: la cadena vacía. Paso inductivo: quitar el último símbolo, usar la hipótesis, volver a ponerlo.

# Un ejemplo corto: $|uv| = |u| + |v|$

Por inducción sobre $|v|$, con $u$ fija.

**Caso base**, $v = \varepsilon$: $\quad |u\varepsilon| = |u| = |u| + 0 = |u| + |\varepsilon|$.

**Paso inductivo**, $v = xa$, y la afirmación vale para $x$:

$$
\begin{aligned}
|u \cdot (xa)| &= |(u \cdot x)a| && \text{definición de concatenación} \\
&= |u \cdot x| + 1 && \text{definición de longitud} \\
&= |u| + |x| + 1 && \text{hipótesis de inducción} \\
&= |u| + |xa| && \text{definición de longitud}
\end{aligned}
$$

**Error común**

Empezar en las cadenas de longitud 1. El caso base es $\varepsilon$: la entrada con la que más programas fallan.

SECCIÓN 03

# Lenguajes

03



# Lenguajes

**Lenguaje** *(language)*

Un conjunto de cadenas sobre $\Sigma$: cualquier subconjunto de $\Sigma^*$.

- $\{0,\ 01,\ 111\}$ — finito
- $\{\, w : w \text{ tiene una cantidad par de unos} \,\}$ — infinito
- $\{\, 0^n 1^n : n \geq 0 \,\} = \{\varepsilon,\ 01,\ 0011,\ 000111,\ \ldots\}$
- $\{\, w : w \text{ es un número primo en binario} \,\}$, $\Sigma^*$ y $\emptyset$

**La pregunta de todo el curso**

Dados $L$ y $w$: ¿se puede saber mecánicamente si $w \in L$? ¿Con qué máquina? ¿A qué costo?

# $\varepsilon$, $\emptyset$ y $\{\varepsilon\}$ no son lo mismo

| | Qué es | Tamaño | En C++ |
|---|---|---|---|
| $\varepsilon$ | una cadena | longitud 0 | `std::string{""}` |
| $\emptyset$ | un lenguaje sin cadenas | 0 cadenas | `std::set<std::string>{}` |
| $\{\varepsilon\}$ | un lenguaje con una cadena | 1 cadena | `std::set<std::string>{""}` |

**Error común**

Confundir $\varepsilon$, la cadena sin símbolos, con $\emptyset$, el lenguaje sin cadenas. Una caja vacía no es lo mismo que no tener caja.

"$M$ acepta $\varepsilon$": una cadena concreta está en su lenguaje. "$L(M) = \emptyset$": no acepta ninguna.

# Operaciones sobre lenguajes

### Las de conjuntos

- Unión: $A \cup B$
- Intersección: $A \cap B$
- Complemento: $\Sigma^* - A$. Siempre respecto de $\Sigma^*$.

### Las propias

- Concatenación: $AB = \{ uv : u \in A,\ v \in B \}$
- Potencias: $A^0 = \{\varepsilon\},\ A^{k+1} = A^k A$
- Estrella: $A^* = A^0 \cup A^1 \cup A^2 \cup \cdots$

**La estrella**

$w \in A^*$ si $w$ se parte en cero o más trozos de $A$. Cero trozos: $\varepsilon \in A^*$ **siempre**.

# Las operaciones, calculadas

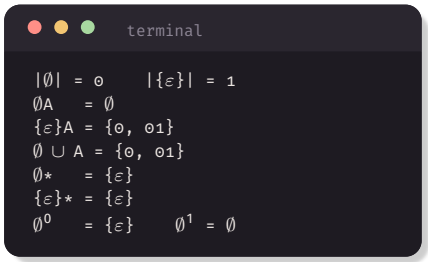
terminal

```
$ build/languages_demo
A = {0, 01}    B = {ε, 1, 01}

A ∪ B = {ε, 0, 1, 01}
A ∩ B = {01}
AB    = {0, 01, 001, 011, 0101}
BA    = {0, 01, 10, 010, 101, 0101}
A²    = {00, 001, 010, 0101}
A*, hasta longitud 3:              {ε, 0, 00, 01, 000, 001, 010}
complemento de A, hasta longitud 2: {ε, 1, 00, 10, 11}
```

- $AB$ tiene 5 cadenas y no 6: $0 \cdot 1 = 01 \cdot \varepsilon$.
- $AB \neq BA$: `10` está solo en $BA$.
- $A^*$ y el complemento son infinitos: se muestran las cadenas cortas.

# $\emptyset$ y $\{\varepsilon\}$ hacen cosas opuestas




- $\{\varepsilon\}A = A$: concatenar con $\varepsilon$ no cambia nada.
- $\emptyset A = \emptyset$: no hay ninguna $u$ que elegir.
- En la concatenación, $\{\varepsilon\}$ hace de 1 y $\emptyset$ hace de 0.
- $\emptyset^* = \{\varepsilon\}$: $\emptyset^0 = \{\varepsilon\}$ por definición, y las demás potencias son vacías.

**Pregunta**

¿Cuántas cadenas tiene $\{\varepsilon\}$? ¿Y $\emptyset$? ¿Qué es $\emptyset^*$?

# De problemas a lenguajes

**Problema de decisión** *(decision problem)*
Un problema en el que cada entrada tiene respuesta sí o no.

- Para una computadora, la entrada es una cadena.
- El problema parte $\Sigma^*$ en dos: las cadenas con respuesta sí, y las demás.

$$\text{PRIMES} = \{\, w \in \{0,1\}^* : w \text{ es la representación en binario de un primo} \,\}$$

¿Es 11 primo? es lo mismo que ¿1011 $\in$ PRIMES?

**Idea clave**
Un problema de decisión **es** un lenguaje: el de las entradas con respuesta sí.

# Lo que hay que fijar, y por qué basta con sí o no

**Al pasar a un lenguaje**

- Cómo se escribe la entrada: binario, lista de aristas…
- Una cadena mal formada no está en el lenguaje. No hay tercer caso.
- Los detalles cuentan: ¿ceros a la izquierda? ¿qué número es $\varepsilon$?

**Por qué basta decidir**

- Si no se puede *decidir* si existe, tampoco se puede *hallar*.
- Casi siempre se vuelve: "¿el bit $i$ de $f(x)$ es 1?"
- Un solo tipo de objeto para comparar todos los problemas.

# Hay más lenguajes que programas (1)

- Un programa es un archivo de texto: **una cadena**.
- Las cadenas se pueden listar, primero las más cortas.
- Entonces los programas se pueden listar: el primero, el segundo, el tercero…
- Cada programa decide a lo sumo un lenguaje: las entradas con las que responde sí.

**Consecuencia**

Los lenguajes que algún programa decide caben en una lista: $L_1, L_2, L_3, \ldots$

¿Están ahí *todos* los lenguajes?

# Hay más lenguajes que programas (2)

| | $\varepsilon$ | 0 | 1 | 00 | 01 |
|---|---|---|---|---|---|
| | $s_1$ | $s_2$ | $s_3$ | $s_4$ | $s_5$ |
| $L_1$ | sí | no | sí | sí | no |
| $L_2$ | no | no | sí | no | sí |
| $L_3$ | sí | sí | sí | no | no |
| $L_4$ | no | sí | no | no | sí |
| $L_5$ | sí | no | no | sí | sí |
| $D$ | **no** | **sí** | **no** | **sí** | **no** |

Una lista de ejemplo: ¿$s_j \in L_i$?

Dada **cualquier** lista de lenguajes $L_1, L_2, \ldots$:

$$D = \{ s_i : s_i \notin L_i \}$$

- $D$ hace lo contrario de $L_i$ con la cadena $s_i$.
- $D \neq L_i$ para todo $i$: difieren en $s_i$.
- $D$ no está en la lista.

Conclusión: hay lenguajes que **ningún programa decide**.

# Lo que el conteo no dice

- Con las 7 cadenas de longitud $\leq 2$ ya hay $2^7 = 128$ lenguajes. Con las 15 de longitud $\leq 3$, $2^{15} = 32\,768$.
- El argumento no dice **cuál** problema no tiene programa: $D$ no se parece a nada útil.
- Tampoco dijimos qué es exactamente un “programa” ni qué es “decidir”.
- Semana 7: la definición. Semana 9: un problema concreto y natural que ningún programa decide.

**Ahora, desde el otro extremo**

La máquina más simple, y los lenguajes que sí alcanza.

SECCIÓN 04

# Autómatas finitos deterministas

04

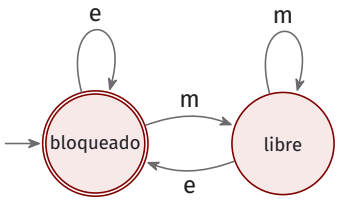
# Una máquina con memoria fija: la puerta automática

- Un sensor delante y otro detrás.
- El controlador solo sabe una cosa: si la puerta está **cerrada** o **abierta**.
- Con eso y la señal, decide.
- Toda su memoria es un bit: en qué fila está.

| | ninguno | delante | detrás | ambos |
|---|---|---|---|---|
| **cerrada** | cerrada | abierta | cerrada | cerrada |
| **abierta** | cerrada | abierta | abierta | abierta |

No cuenta personas ni guarda una historia de lo que pasó.

# El torniquete




`m`: entra una moneda.

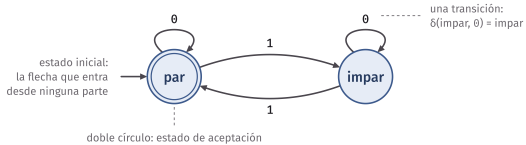
`e`: alguien empuja.

```
terminal

$ build/run_dfa machines/turnstile.dfa emme
w = emme
  bloqueado -e-> bloqueado -m-> libre -m-> libre -e-> bloqueado
  termina en bloqueado, que está en F: acepta
```

- Lo que le pasa en el día es una cadena sobre $\{m, e\}$.
- La segunda moneda se perdió: no tiene dónde anotarla.
- **Acepta** si termina en un estado marcado.

# Cantidad par de unos

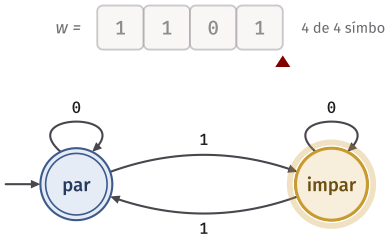



- No hace falta contar: basta la **paridad** de los unos leídos hasta ahora.
- Un 0 no la cambia. Un 1 la cambia.
- Sin leer nada: cero unos, y cero es par.

**Idea clave**

Lee una sola vez, de izquierda a derecha. Todo lo que sabe de lo leído es en qué estado está.

# Cantidad par de unos: la ejecución con 1101




**Lee 1: δ(par, 1) = impar. Tres unos, y la cadena se acabó.**

Termina en impar, que no está en F: rechaza. Pasó tres veces por par, pero solo cuenta el estado final.

# La definición formal

**Autómata finito determinista, DFA** ***(deterministic finite automaton)***

Una quíntupla $M = (Q, \Sigma, \delta, q_0, F)$, donde:

1. $Q$ es un conjunto finito, el de los **estados**;
2. $\Sigma$ es un alfabeto;
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 de aceptación**.

Los libros en español escriben **AFD**, **AFN** y **AP** donde este curso escribe DFA, NFA y PDA. Son los mismos objetos.

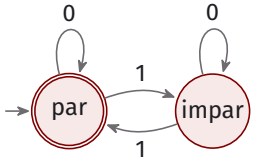
# Leer la definición despacio

- $\delta$ **es una función** de $Q \times \Sigma$ en $Q$:
  - definida para **todos** los pares: de cada estado sale una flecha con cada símbolo;
  - con **un solo** resultado: no hay dos flechas con el mismo símbolo. Eso es *determinista*.
- Hay **un** estado inicial.
- $F$ es cualquier subconjunto: uno, varios, todos o **ninguno**. $q_0$ puede estar en $F$.
- $Q$ es **finito**. Es la única limitación, y de ahí sale todo lo que un DFA no puede hacer.

**Pregunta**

¿Qué lenguaje reconoce un DFA con $F = \emptyset$? ¿Y uno con $F = Q$?

# El diagrama, la tabla y el archivo dicen lo mismo




| $\delta$ | 0 | 1 |
|---|---|---|
| par | par | impar |
| impar | impar | par |

even_ones.dfa

```
states: par impar
alphabet: 0 1
start: par
accept: par
delta:
par 0 par
par 1 impar
impar 0 impar
impar 1 par
```

terminal

```
$ build/show_dfa machines/even_ones.dfa
Q  = {par, impar}
Σ  = {0, 1}
q0 = par
F  = {par}

     δ      0      1
-> * par    par    impar
     impar  impar  par

estados alcanzables desde par: 2 de 2
```

- Una casilla de la tabla es una flecha del diagrama y una línea después de `delta:`.
- Una casilla vacía es una transición olvidada. El formato está en `code/formats.md`. (El archivo real empieza con una línea de comentario.)

# La función de transición extendida

$\delta^* : Q \times \Sigma^* \to Q$ ***(extended transition function)***

$\delta^*(q, \varepsilon) = q$

$\delta^*(q, xa) = \delta(\delta^*(q, x), a)$

$\delta^*(q, w)$: el estado en el que queda $M$ si empieza en $q$ y lee $w$ completa.

$$\begin{aligned}\delta^*(\text{par}, 110) &= \delta(\delta^*(\text{par}, 11), 0)\\ &= \delta(\delta(\delta(\delta^*(\text{par}, \varepsilon), 1), 1), 0)\\ &= \delta(\delta(\delta(\text{par}, 1), 1), 0) = \delta(\delta(\text{impar}, 1), 0) = \delta(\text{par}, 0) = \text{par}\end{aligned}$$

**Lema de la concatenación**

$\delta^*(q, uv) = \delta^*(\delta^*(q, u), v)$. Si $u$ y $u'$ llevan al mismo estado, $uv$ y $u'v$ también, para toda $v$.

# Aceptar una cadena, reconocer un lenguaje

**Aceptar y reconocer**

$M$ **acepta** $w$ si $\delta^*(q_0, w) \in F$. Si no, la **rechaza**.
$L(M) = \{ w \in \Sigma^* : \delta^*(q_0, w) \in F \}$ es el **lenguaje de** $M$, y $M$ lo **reconoce**.

- Una máquina acepta *cadenas* y reconoce *un* lenguaje. Si no acepta nada, reconoce $\emptyset$.
- "$M$ reconoce $A$" es $L(M) = A$: acepta **todas** las de $A$ y **ninguna** otra.
- Solo cuenta el estado **final**.

**Error común**

"Acepta porque pasa por un estado de aceptación." Con 1101: par, impar, par, par, impar. Pasa tres veces por `par` y la rechaza.

# En C++: el `struct` es la quíntupla

dfa.h

```cpp
struct Dfa {
    std::vector<std::string> states;   // Q
    std::vector<char> alphabet;        // el alfabeto
    std::map<std::pair<std::string, char>,
             std::string> delta;       // la función de transición
    std::string start;                 // q0
    std::set<std::string> accept;      // F
};
inline std::string delta_star(const Dfa& m, std::string q,
                              const std::string& w) {
    for (char a : w) {
        q = step(m, q, a);
    }
    return q;
}
```

- Los cinco nombres del archivo `.dfa`.
- El ciclo es la segunda línea de $\delta^*$, una vez por símbolo.
- Con $\varepsilon$ no da vueltas: devuelve $q$.
- Memoria que no depende de $|w|$.

SECCIÓN 05

# Diseñar un DFA

05

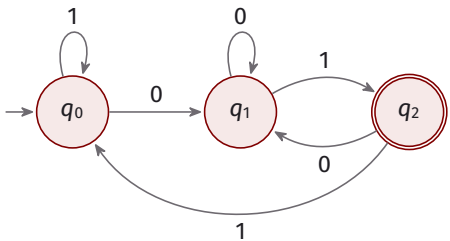
# El método: decidir qué recordar

1. **Pónganse en el lugar de la máquina.** Los símbolos llegan de uno en uno, y en cualquier momento pueden decirles “ahí terminó”.
2. **¿Qué hay que recordar** de lo leído? Una lista *finita* de posibilidades.
3. **Un estado por posibilidad.** Cada estado es un hecho sobre lo leído hasta ahora. Escríbanlo.
4. **Transiciones:** si el hecho es cierto y llega este símbolo, ¿qué hecho pasa a ser cierto? La tabla completa.
5. **Inicial:** el hecho que cumple $\varepsilon$. **Aceptación:** los hechos que implican estar en el lenguaje.
6. **Probar** con $\varepsilon$ y con cadenas cortas, antes de demostrar.

# Termina en 01

| Estado | Lo leído… |
|---|---|
| $q_0$ | no termina en 0 ni en 01 |
| $q_1$ | termina en 0 |
| $q_2$ | termina en 01 |

- Con 0: termina en 0, venga de donde venga.
- Con 1: termina en 01 solo si antes terminaba en 0.
- De $q_2$ **se sale**: el hecho deja de ser cierto.




Recordar los dos últimos símbolos también funciona, con 7 estados. Se recuerda de más.

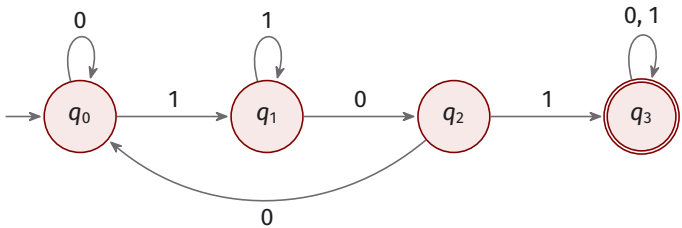
# Termina en 01: la ejecución

terminal

```
$ build/run_dfa machines/ends_01.dfa 110101 010
w = 110101
  q0 -1-> q0 -1-> q0 -0-> q1 -1-> q2 -0-> q1 -1-> q2
  termina en q2, que está en F: acepta
w = 010
  q0 -0-> q1 -1-> q2 -0-> q1
  termina en q1, que no está en F: rechaza
```

- La **ejecución** con $w$ tiene $|w| + 1$ estados, y el último es $\delta^*(q_0, w)$.
- Con 110101 la máquina llega a $q_2$, sale y vuelve a entrar.

# Contiene 101




| Estado | Lo leído… |
|---|---|
| $q_0$ | sin 101; no termina en 1 ni en 10 |
| $q_1$ | sin 101; termina en 1 |
| $q_2$ | sin 101; termina en 10 |
| $q_3$ | contiene 101 |

- $q_1$ con 1: termina en 11. El último 1 puede iniciar un 101: **se queda**.
- $q_2$ con 0: termina en 100. Nada sirve: a $q_0$.
- $q_3$: ya ganó, y no se sale.

# Contiene 101: cuando un intento falla

terminal

```
$ build/run_dfa machines/contains_101.dfa 11011 110011
w = 11011
  q0 -1-> q1 -1-> q1 -0-> q2 -1-> q3 -1-> q3
  termina en q3, que está en F: acepta
w = 110011
  q0 -1-> q1 -1-> q1 -0-> q2 -0-> q0 -1-> q1 -1-> q1
  termina en q1, que no está en F: rechaza
```

**Error común**

Mandar $q_1$ con 1 de vuelta a $q_0$, porque “se rompió el patrón”. Con ese cambio, 11011 se rechaza: después de 11 estaría en $q_0$ y ya no vería el 101 del final.

# Múltiplos de 3 en binario

- val($w$): el número que $w$ representa en base 2. $\text{val}(110) = 6$, $\text{val}(\varepsilon) = 0$.
- El número no cabe en los estados. Su **resto entre 3** sí: 0, 1 o 2.

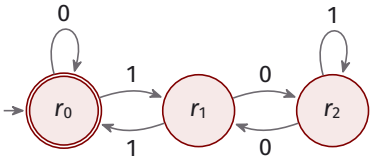
$$\text{val}(wb) = 2 \cdot \text{val}(w) + b$$

| **Estado** | **Lo leído…** | **Con 0: resto de** $2r$ | **Con 1: resto de** $2r+1$ |
|---|---|---|---|
| $r_0$ | deja resto 0 | $0 \to r_0$ | $1 \to r_1$ |
| $r_1$ | deja resto 1 | $2 \to r_2$ | 3, resto $0 \to r_0$ |
| $r_2$ | deja resto 2 | 4, resto $1 \to r_1$ | 5, resto $2 \to r_2$ |

**Idea clave**

Si lo que hay que recordar no cabe en una lista finita, no hay DFA. Aquí cabe: el resto se mantiene al día con cada bit.

# Múltiplos de 3: el autómata




Los prefijos de 10101 valen 1, 2, 5, 10, 21: restos 1, 2, 2, 1, 0.

terminal

```
$ build/run_dfa machines/multiple_of_3.dfa 10101 1101 0011
w = 10101
  r0 -1-> r1 -0-> r2 -1-> r2 -0-> r1 -1-> r0
  termina en r0, que está en F: acepta
w = 1101
  r0 -1-> r1 -1-> r0 -0-> r0 -1-> r1
  termina en r1, que no está en F: rechaza
w = 0011
  r0 -0-> r0 -0-> r0 -1-> r1 -1-> r0
  termina en r0, que está en F: acepta
```

SECCIÓN 06

# Demostrar que un DFA es correcto

06

# Una afirmación por estado

Probar con muchas cadenas no termina nunca. Se demuestra **más** de lo pedido: en qué estado queda $M$.

> **Afirmación (paridad)**
> Para toda $w$: $\delta^*(\text{par}, w) = \text{par}$ si $w$ tiene una cantidad par de unos, e impar si no.

**Caso base**, $w = \varepsilon$: cero unos, y $\delta^*(\text{par}, \varepsilon) = \text{par}$.
**Paso inductivo**, $w = xa$: una línea por casilla de la tabla.

- $x$ par, $a = 0$: $\delta(\text{par}, 0) = \text{par}$, y $w$ tiene los mismos unos.
- $x$ par, $a = 1$: $\delta(\text{par}, 1) = \text{impar}$, y $w$ tiene un uno más.
- $x$ impar, con $a = 0$ y con $a = 1$: igual, con la otra fila.

Demostrar más es más fácil: la hipótesis también dice más, dónde quedó la máquina con $x$.

# Termina en 01, demostrado

**Afirmación.** (a) Si $w$ termina en 01, queda en $q_2$. (b) Si termina en 0, en $q_1$. (c) Si no, en $q_0$.

| $\delta$ | 0 | 1 |
|---|---|---|
| $q_0$ | $q_1$ | $q_0$ |
| $q_1$ | $q_1$ | $q_2$ |
| $q_2$ | $q_1$ | $q_0$ |

**Base:** $\varepsilon$ cumple (c), y $\delta^*(q_0, \varepsilon) = q_0$. **Paso:** $w = xa$, $p = \delta^*(q_0, x)$.

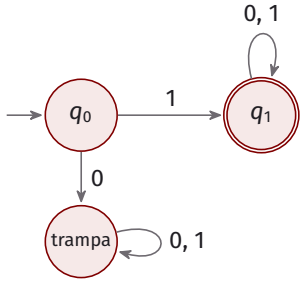
- $a = 0$: $w$ termina en 0. La columna del 0 es toda $q_1$.
- $a = 1$ y $x$ termina en 0: $w$ termina en 01. Por hipótesis $p = q_1$, y $\delta(q_1, 1) = q_2$.
- $a = 1$ y $x$ no termina en 0: a $w$ le toca (c). Por hipótesis $p$ es $q_0$ o $q_2$, y con 1 los dos van a $q_0$.

**Error común**

Demostrar una sola dirección. “Toda cadena que termina en 01 se acepta” también lo cumple un autómata que acepta todo. Con una afirmación por estado salen las dos.

# El estado trampa

**Empieza con 1.** ¿Y $q_0$ con 0?




- "Ahí la máquina se muere" no es un DFA: el archivo no se carga.
- "Ya perdió" también es un hecho sobre lo leído, y necesita su estado.
- Error común: olvidar el estado trampa y dejar transiciones sin definir. En un DFA, $\delta$ es total.

terminal

```
$ build/show_dfa errors/missing_trap.dfa
error: errors/missing_trap.dfa: falta la transición δ(q0, 0): en un DFA, δ es total
```

# Lenguajes regulares

**Lenguaje regular** *(regular language)*

Un lenguaje es regular si algún DFA lo reconoce.

- Hasta ahora: paridad de unos, termina en 01, contiene 101, múltiplos de 3, empieza con 1.
- Para demostrar que un lenguaje es regular: mostrar un DFA y demostrar que lo reconoce.
- Para demostrar que **no** lo es: probar que ningún DFA sirve. Semana 4.

**La pregunta que sigue**

Si $A$ y $B$ son regulares, ¿también $A \cap B$, $A \cup B$ y el complemento de $A$? ¿La clase es **cerrada**?

SECCIÓN 07

# La construcción del producto

07

# Dos máquinas a la vez

**Problema.** $M_1$ reconoce "cantidad par de unos". $M_2$ reconoce "termina en 01". Queremos un DFA para las cadenas que cumplen **las dos**.

- Un programa correría $M_1$, luego $M_2$, y juntaría las respuestas.
- Un DFA lee la entrada **una sola vez**: tiene que correr las dos a la vez.
- Para eso debe recordar en qué estado va cada una: un **par** de estados.
- Hay una cantidad finita de pares. Cada par puede ser un estado.

# La construcción

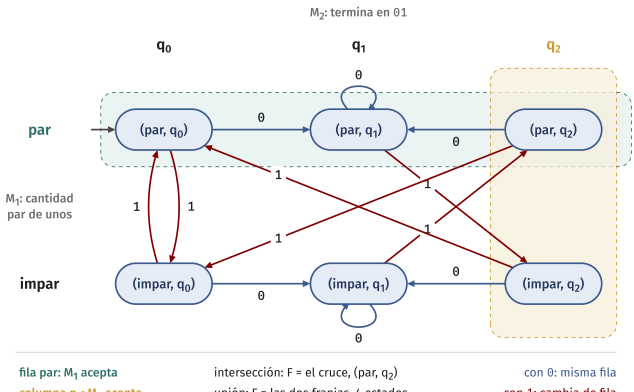
Sean $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ y $M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$, con el mismo alfabeto.

**El producto** $M = (Q, \Sigma, \delta, q_0, F)$ ***(product construction)***

- $Q = Q_1 \times Q_2$
- $\delta((p, q), a) = (\delta_1(p, a),\ \delta_2(q, a))$
- $q_0 = (q_1, q_2)$
- Intersección: $F = F_1 \times F_2$. Los dos aceptan.
- Unión: $F = \{\, (p, q) : p \in F_1 \text{ o } q \in F_2 \,\}$. Alguno acepta.

Es un DFA: $|Q_1| \cdot |Q_2|$ estados, y $\delta$ es total porque $\delta_1$ y $\delta_2$ lo son.

# El producto de la paridad y “termina en 01”




Fila: el estado de $M_1$. Columna: el de $M_2$. $2 \times 3 = 6$ estados.

# El producto, ejecutado

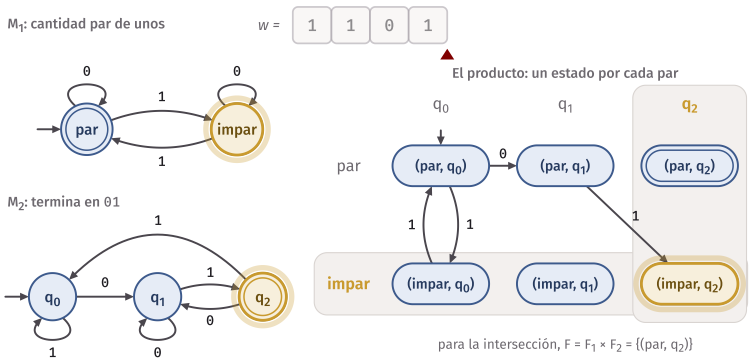
terminal

```
$ build/product inter machines/even_ones.dfa machines/ends_01.dfa | build/run_dfa - 1101 1001
w = 1101
  (par,q0) -1-> (impar,q0) -1-> (par,q0) -0-> (par,q1) -1-> (impar,q2)
  termina en (impar,q2), que no está en F: rechaza
w = 1001
  (par,q0) -1-> (impar,q0) -0-> (impar,q1) -0-> (impar,q1) -1-> (par,q2)
  termina en (par,q2), que está en F: acepta
```

| **Leído** | $M_1$ | $M_2$ | **Producto** |
|---|---|---|---|
| $\varepsilon$ | par | $q_0$ | (par, $q_0$) |
| 1 | impar | $q_0$ | (impar, $q_0$) |
| 11 | par | $q_0$ | (par, $q_0$) |
| 110 | par | $q_1$ | (par, $q_1$) |
| 1101 | impar | $q_2$ | (impar, $q_2$) |

- La columna del producto es el par de las otras dos. Eso hay que demostrar.
- 1101 termina en 01 y tiene tres unos: $M_2$ sí, $M_1$ no.

# Las dos máquinas, en paralelo




**Leído 1101: $M_1$ queda en impar, $M_2$ queda en $q_2$, y el producto en (impar, $q_2$).**

$M_2$ acepta y $M_1$ no: (impar, $q_2$) no está en $F_1 \times F_2$, y el producto rechaza 1101.

# Por qué funciona: el lema del producto

**Lema**

Para todo par $(p, q)$ y toda cadena $w$: $\delta^*((p,q), w) = (\delta_1^*(p, w),\ \delta_2^*(q, w))$.

**Caso base**, $w = \varepsilon$: los dos lados son $(p, q)$.
**Paso inductivo**, $w = xa$:

$$
\begin{aligned}
\delta^*((p,q), xa) &= \delta(\delta^*((p,q), x),\ a) && \text{definición de } \delta^* \\
&= \delta((\delta_1^*(p,x), \delta_2^*(q,x)),\ a) && \text{hipótesis de inducción} \\
&= (\delta_1(\delta_1^*(p,x), a),\ \delta_2(\delta_2^*(q,x), a)) && \text{definición de } \delta \text{ en el producto} \\
&= (\delta_1^*(p, xa),\ \delta_2^*(q, xa)) && \text{definición de } \delta_1^* \text{ y } \delta_2^*
\end{aligned}
$$

# Los regulares son cerrados bajo intersección y unión

**Teorema**

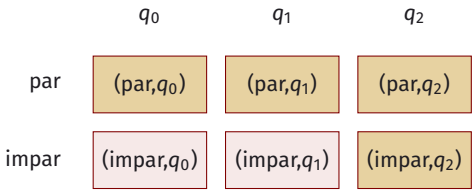
Si $A_1$ y $A_2$ son regulares, $A_1 \cap A_2$ y $A_1 \cup A_2$ también.

**Qué se construye:** el producto de $M_1$ y $M_2$ con $F = F_1 \times F_2$. **Por qué reconoce** $A_1 \cap A_2$**:**

$$
\begin{aligned}
M \text{ acepta } w &\iff \delta^*((q_1, q_2), w) \in F_1 \times F_2 \\
&\iff (\delta_1^*(q_1, w),\ \delta_2^*(q_2, w)) \in F_1 \times F_2 && \text{lema} \\
&\iff \delta_1^*(q_1, w) \in F_1 \ \text{ y } \ \delta_2^*(q_2, w) \in F_2 \\
&\iff w \in A_1 \cap A_2
\end{aligned}
$$

Cada línea es un “si y solo si”: salen las dos direcciones. Para la unión, la misma cadena con “o”.

# La unión: la misma máquina con distinto $F$




- Los estados, las transiciones y el inicial son los mismos.
- Unión: toda la fila *par* y toda la columna $q_2$.

terminal

```
$ build/product union machines/even_ones.dfa machines/ends_01.dfa | grep accept
accept: (par,q0) (par,q1) (par,q2) (impar,q2)
```

Error común: usar $F_1 \times F_2$ para la unión, o escribir $F = F_1 \cup F_2$, que no son pares. Es $(F_1 \times Q_2) \cup (Q_1 \times F_2)$.

# ¿Todos los pares se alcanzan?

terminal

```
$ build/product inter machines/even_ones.dfa machines/even_ones.dfa --alcanzables | head -6
# Producto de machines/even_ones.dfa y machines/even_ones.dfa: intersección.
# 2 × 2 = 4 estados, 2 alcanzables.
states: (par,par) (impar,impar)
alphabet: 0 1
start: (par,par)
accept: (par,par)
```

- **Alcanzable:** alguna cadena lleva hasta él. Una máquina consigo misma va siempre en $(q, q)$.
- "Termina en 01" con "contiene 101": $3 \times 4 = 12$ estados, **8** alcanzables.
- Los que faltan son hechos que se contradicen: $(q_1, q_1)$ diría "termina en 0" y "termina en 1".

En la práctica: un programa parte del par inicial y construye solo los pares que encuentra.

# Complemento

**Teorema**

Si $A$ es regular, su complemento $\Sigma^* - A$ también.

**Qué se construye:** $M' = (Q, \Sigma, \delta, q_0, Q - F)$. La misma máquina, con la aceptación intercambiada.
**Por qué:** para toda $w$, el estado $\delta^*(q_0, w)$ existe, es uno solo, y está en $F$ o en $Q - F$.

$$M' \text{ acepta } w \iff \delta^*(q_0, w) \in Q - F \iff \delta^*(q_0, w) \notin F \iff w \notin A$$

Error común: complementar sin estado trampa. En “empieza con 1” con dos estados, la cadena 0 no se acepta ni antes ni después del intercambio. Primero se completa $\delta$.

# Lo que un DFA no puede hacer

- Todo lo de hoy recuerda algo acotado: una paridad, un resto, los últimos símbolos.
- $\{ 0^n 1^n : n \geq 0 \}$ pide recordar **cuántos** ceros hubo. No tiene tope.
- Un DFA con $n$ estados lee $\varepsilon, 0, 00, \ldots, 0^n$: son $n + 1$ cadenas.
- Dos de ellas, $0^i$ y $0^j$, lo dejan en el mismo estado. Desde ahí hace lo mismo con cualquier continuación.
- Entonces trata igual a $0^i 1^i$ y a $0^j 1^i$. Una está en el lenguaje y la otra no.

**Es solo la idea**

La demostración, y dos herramientas generales, en la semana 4.

# Laboratorio: un simulador de DFA

1. Leer un DFA y una lista de cadenas, y responder `acepta` o `rechaza` por cada una.
2. Representarlo con los cinco componentes, con los mismos nombres.
3. Pasar los casos de borde: la cadena vacía, un DFA sin estados de aceptación, uno que acepta todo.
4. Probar con su simulador los DFA de la clase, y escribir uno propio.
5. Opcional: el producto de dos DFA.

Dónde: `code/week-01/lab/dfa_sim`. `make test` corre el juez. Una línea vacía es $\varepsilon$: lean con `std::getline`.

casos/01.in

```
states: par impar
alphabet: 0 1
start: par
accept: par
delta:
par 0 par
par 1 impar
impar 0 impar
impar 1 par
strings: 3
0110

111
```

Sin la línea de comentario inicial.

# Para la próxima clase

- Leer los apuntes de la semana 1 y terminar el laboratorio.
- Preguntas de repaso de la guía: un DFA para “no contiene `aa`”; cuántos estados tiene un producto y cuántos se alcanzan; qué falla en el complemento sin estado trampa.
- Próxima semana: máquinas que pueden elegir. NFA, transiciones $\varepsilon$ y la construcción de subconjuntos.

**Pregunta**

“El tercer símbolo desde el final es 1.” ¿Qué tendría que recordar un DFA para reconocerlo? ¿Cuántos estados son?