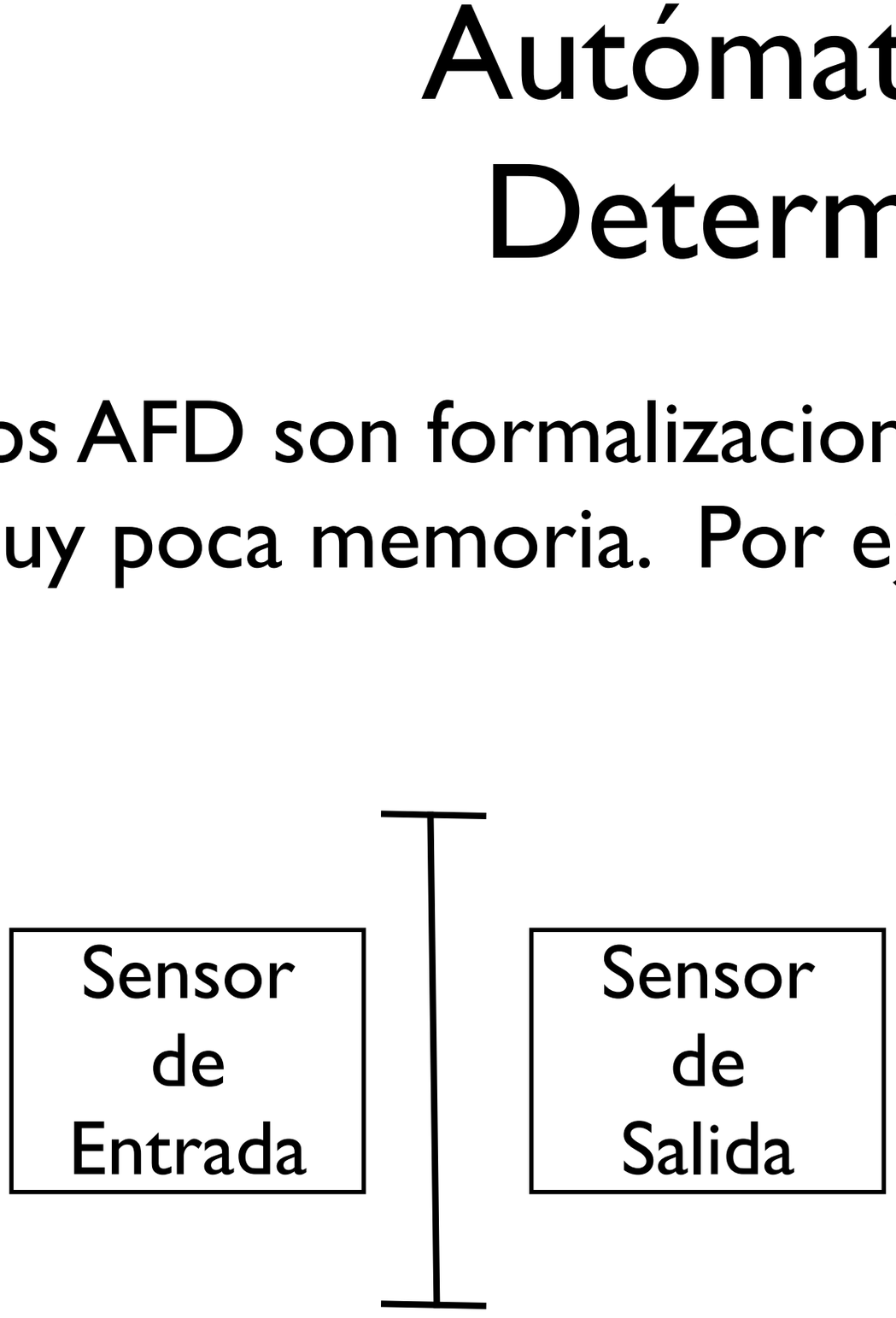
# Introducción a Automatas y Lenguajes

Pablo Castro

Programación Avanzada, 2018

# Autómatas Finitos Deterministicos

Los AFD son formalizaciones de dispositivos con muy poca memoria. Por ejemplo:

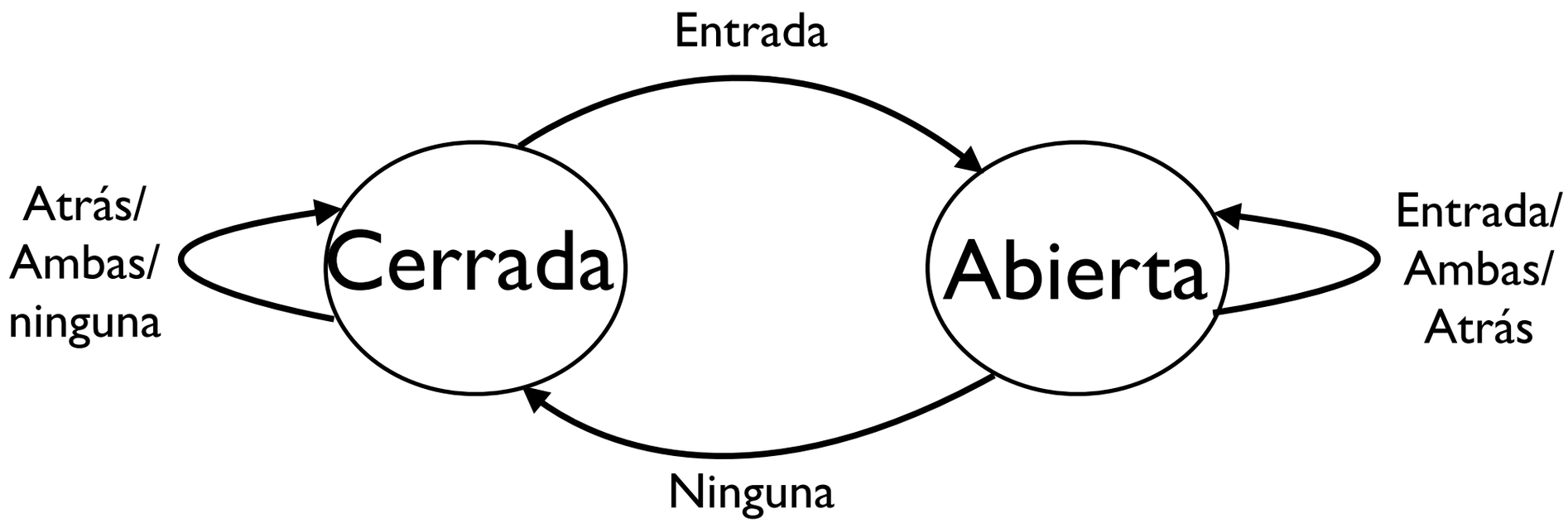



Pensemos en una puerta automática.

1. Si una persona se para en la entrada se abre.
2. Si no se detecta nada se cierra.
3. Si una persona está parada en la salida, si está abierta, se mantiene abierta, sino cerrada.

# AFD (cont)

- Tenemos dos estados: *Abierta* y *Cerrada*.
- Tenemos los siguientes cambios de estados: Alguien en la entrada, nadie en ningún lado.



# AFD (Cont)

Cada secuencia de eventos, produce una secuencia de transiciones de estados. Por ejemplo:

*Entrada,Atrás,Ninguno, Entrada,Ambas, Ninguna,Atrás, Ninguna*

Determina la siguiente secuencia de estados:

*Abierta,Abierta,Cerrada,Abierta,Abierta,Cerrada,Cerrada, Cerrada*

# AFD (cont)

Los AFD son abstracciones de estos mecanismos.
Un AFD es una tupla: $\langle Q, \Sigma, \delta, q_0, F \rangle$

- $Q$ es un conjunto finito de estados.
- $\Sigma$ es un conjunto finito, llamado alfabeto.
- $\delta : Q \times \Sigma \to Q$ es una función de transición.
- $q_0 \in Q$ es un estado de inicio.
- $F \subseteq Q$ es el conjunto de estados finales.

# AFD (cont.)

Generalmente la función de transición se describe usando una tabla:

$\delta =$

| | Atrás | Entrada | Ambas | Ninguna |
|---|---|---|---|---|
| Cerrada | Cerrada | Abierta | Cerrada | Cerrada |
| Abierta | Cerrada | Abierta | Abierta | Abierta |

# AFD

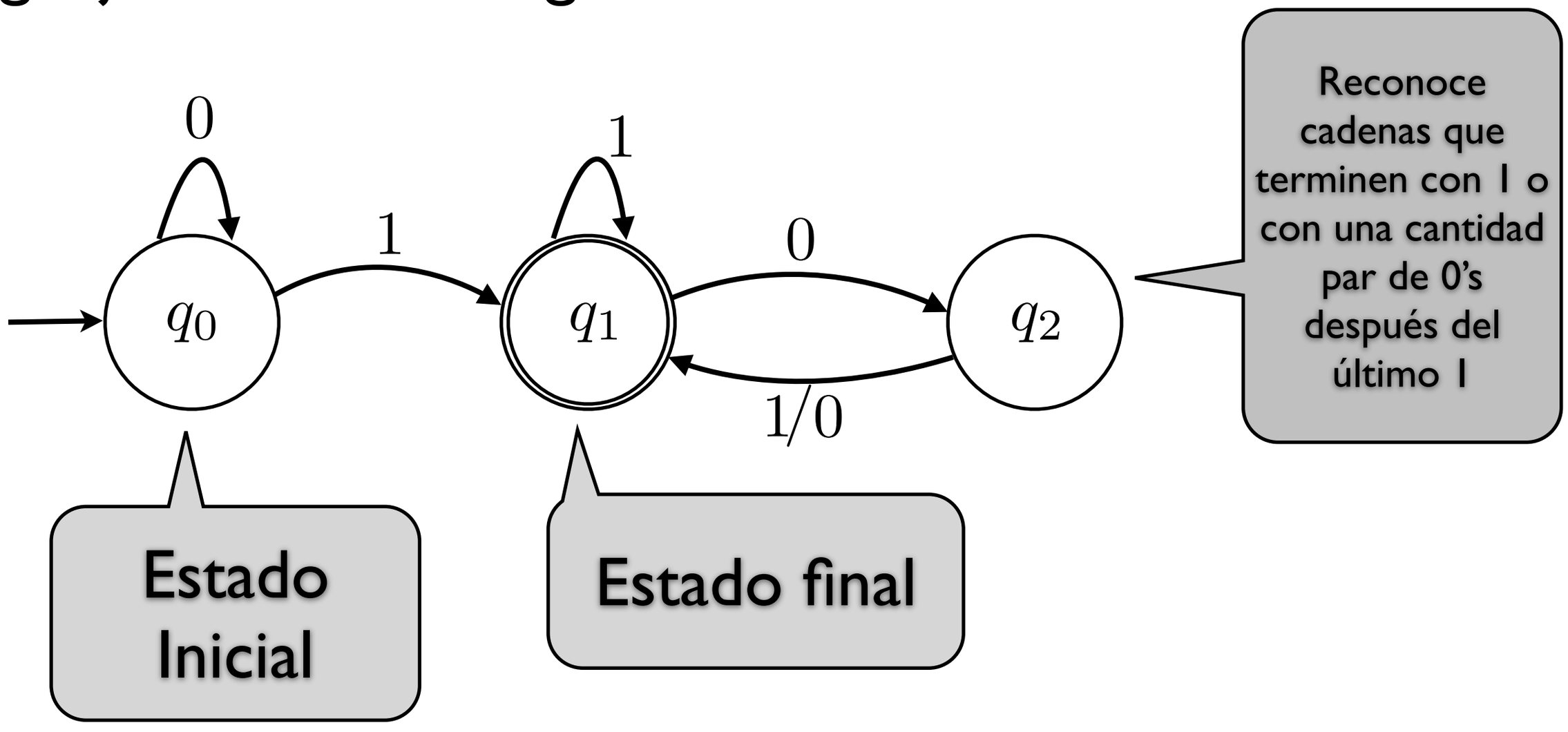
En general, los AFD se usan para capturar lenguajes.

Un lenguaje es un conjunto de secuencias de símbolos

- En computación consideramos lenguajes solo con 0 y 1's.
- Estos alcanzan para describir cualquier lenguaje con más símbolos (finitos).
- Por ejemplo: $\{0^k \mid k \geq 1\}$ es el lenguaje 0,00,000,000, 0000, ....

# AFD

Los AFD pueden reconocer una colección simple de lenguajes, llamados regulares:



# Definición formal de Aceptación

Definimos: $\delta^* : Q \times [Bin] \to Q$

$$\delta^*(q, []) = q$$
$$\delta^*(q, x : xs) = \delta^*(\delta(q, x), xs)$$

Por ejemplo:

$$\delta^*(q_0, 100) = \delta^*(q_1, 00) = \delta^*(q_2, 0) = \delta^*(q_1, [])$$

En este caso decimos que A acepta el lenguaje.

# Usando Inv. para construir AFD

Supongamos que queremos reconocer:

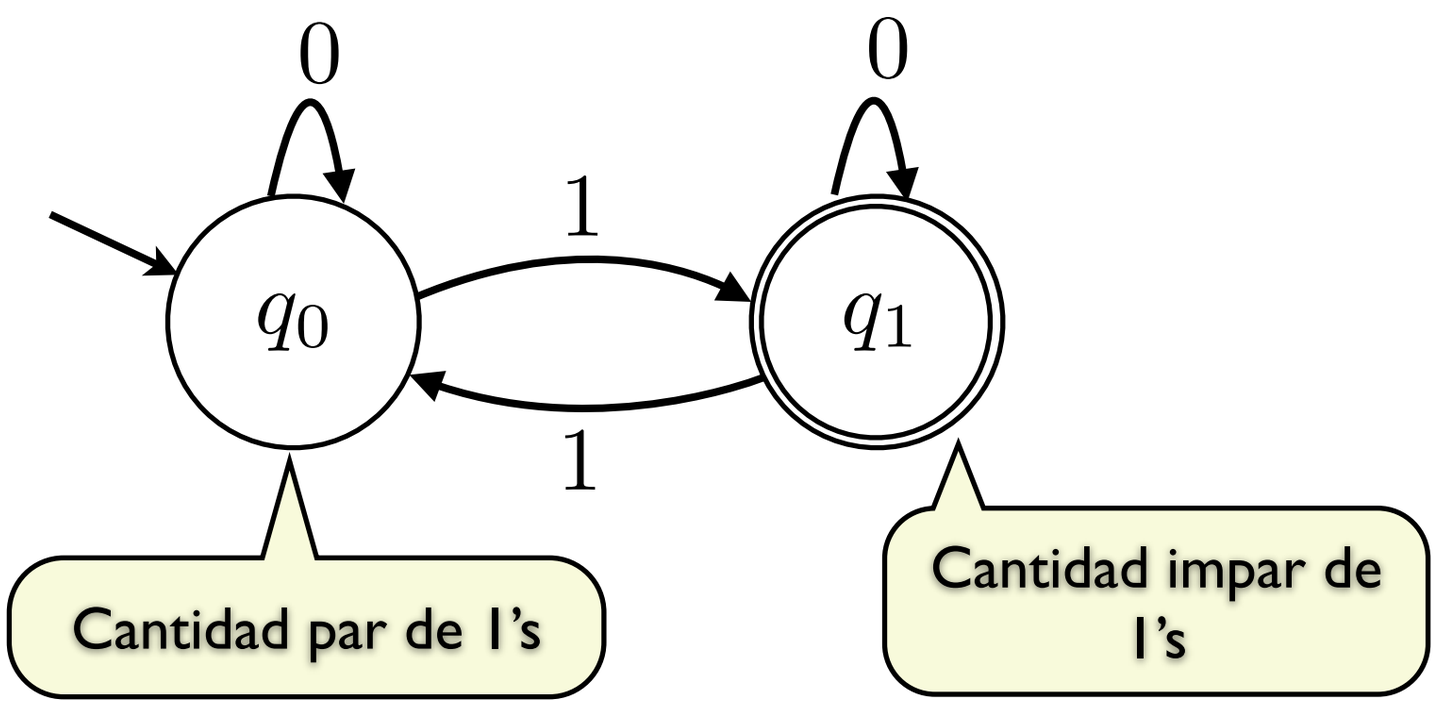
$$L = \{\{0,1\}^k \mid k \geq 0 \text{ y la cant. de 1's es impar}\}$$

Podemos hacerlo con invariantes:

- Un invariante por cada estado,
- q0: Hasta aquí tenemos una cantidad par de 1's.
- q1: Hasta aquí tenemos una cantidad impar de 1's.

# AFD

Obtenemos el siguiente autómata:

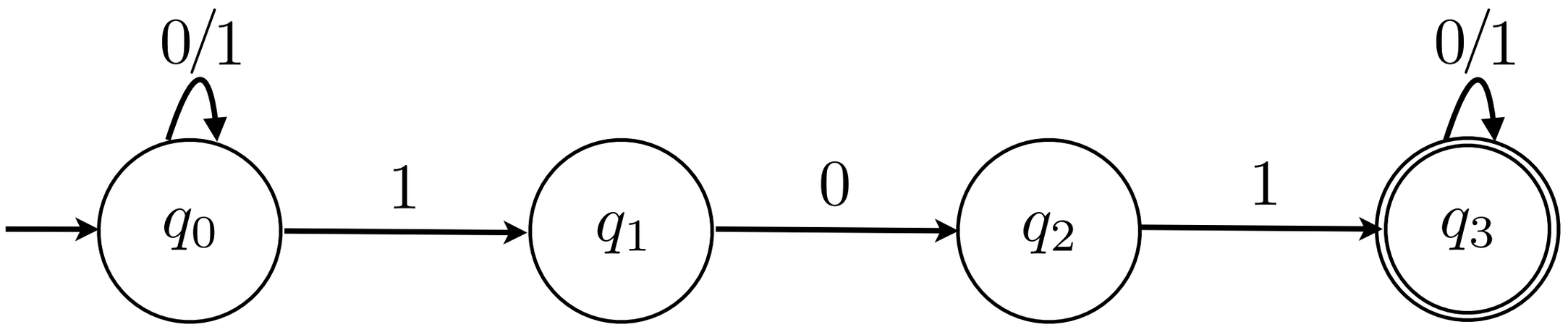


# Ejercicios

- Todas las cadenas con 001.
- Las cadenas que tienen al menos 3 1's y al menos 2 0's
- Las cadenas que tienen una cantidad par de 0's y una cantidad impar de 1's

# AFND

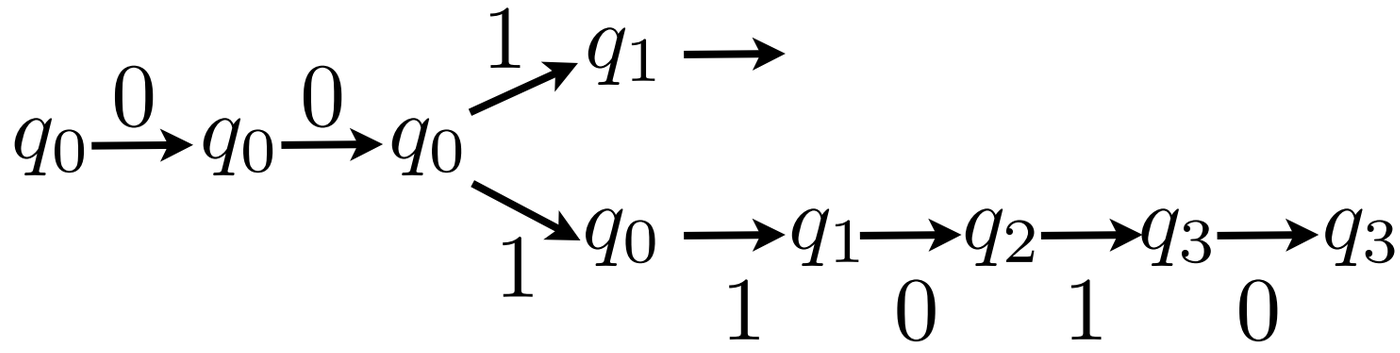
Los autómatas finitos no determinísticos permiten introducir no determinismo.




En este ejemplo en el estado $q_0$ tenemos dos posibilidades. Podemos pensar la ejecución como un árbol

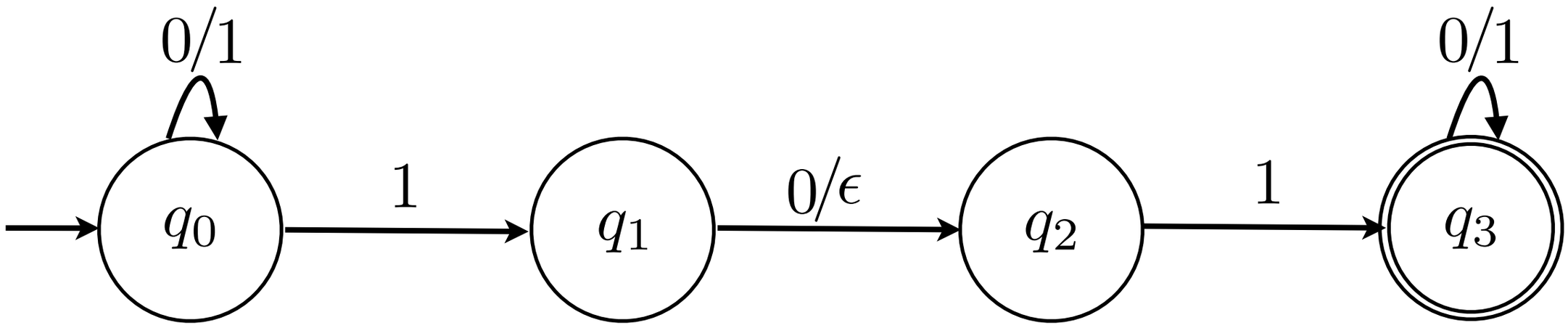
# AFND

Veamos las posibles ejecuciones de 0011010:



# AFND con transiciones lambda

Las transiciones lambda o epsilon son transiciones que no consumen símbolos




Cadenas conteniendo 101 ó 11

# Operaciones sobre autómatas

- Unión de autómatas, si tenemos $A_0$ y $A_1$ podemos construir $A_0 \cup A_1$
- Concatenación de autómatas, dado $A_0$ y $A_1$ podemos construir $A_0 \circ A_1$
- Iteración, dado $A$ podemos construir $A^*$ .

Estas operaciones se llaman operaciones regulares, y los lenguajes construidos con ellas se llaman leng. regulares

# Equivalencia de Autómatas

Todos estos formalismos son equivalentes, es decir, capturan los mismos lenguajes:

$$AFD = AFND = AFND+\epsilon$$

Los lenguajes regulares se utilizan para buscar en textos, para construir parsers y para diferentes operaciones sobre lenguajes.

# Gramáticas y Lenguajes

Algunas definiciones básicas:

Alfabeto: Es una colección de símbolos. Por ejemplo

$$\Sigma = \{0, 1\}$$

Alfabeto binario

Cadena: Es una secuencia de símbolos de algún alfabeto:

$$001 \qquad 101 \qquad 0 \qquad \epsilon$$

Secuencia vacía

Lenguaje: Es un conjunto de cadenas:

$$L = \{1, 11, 111, 1111, \dots\}$$

Conjunto de cadenas con todos unos

# Ejemplos

- El lenguaje castellano puede ser definido como el conjunto de aquellos textos que están bien escritos.
- El lenguaje de Haskell viene dado por el conjunto de todos los programas bien formados de Haskell.
- $L = \{10, 11, 101, \dots\}$, el lenguaje de todos los números primos en binario.

# Gramáticas

Una gramática es un conjunto de reglas que nos permiten definir lenguajes, por ejemplo:

$$EXP \rightarrow EXP + EXP \mid EXP * EXP \mid EXP - EXP$$
$$\mid EXP\ EXP \mid (EXP) \mid NUM$$
$$NUM \rightarrow CONS \mid CONS\ NUM$$
$$CONS \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

Define las expresiones numéricas:

$$15 * 5 \qquad 10/(3 * 3) \qquad 1252 \qquad ((3 + 5)/20) * 100$$

# Gramáticas (cont.)

Formalmente una gramática consta de:

1,2,3,4,5,6,7,8,9,0 en el ejemplo anterior

- Conjuntos de símbolos terminales (símbolos del alfabeto).

EXP, NUM, CONS en el ejemplo anterior

- Conjuntos de símbolos no-terminales, sirven para definir estructuras sintácticas validas.

- Conjunto de reglas, o producciones.

Introducidas con flechas en el ejemplo dado

# Derivaciones

Una derivación es una secuencia de aplicaciones de reglas, que producen una secuencia de terminales:

$$EXP \to EXP + EXP \to EXP + (EXP) \to EXP + (EXP * EXP) \to NUM + (EXP * EXP) \to$$

$$CONS + (EXP * EXP) \to 5 + (EXP * EXP) \to 5 + (NUM + EXP) \to 5 + (CONS * EXP) \to$$

$$5 + (10 * EXP) \to 5 + (10 * NUM) \to 5 + (10 * 3)$$

Es decir, con esta derivación generamos la cadena:

$$5 + (10 * 3)$$

El lenguaje viene dado por todas las cadenas que podemos generar con la gramática.

# BNF

BNF es un lenguaje generalmente usado para describir gramáticas de los lenguajes de programación:

<vardecl> ::= var <vardecllist> ;
<vardecllist> ::= <varandtype> { ; <varandtype> }
<varandtype> ::= <ident> { , <ident> } : <typespec>
<ident> ::= <letter> { <idchar> }
<idchar> ::= <letter> | <digit>

Define declaraciones de variables:

var v: int;
var b:boolean;

# Jerarquias de Lenguajes

La jerarquía de Chomsky califica las gramáticas según la expresividad de los lenguajes que generan:

- Gramáticas tipo 0: Incluye todas las gramáticas de aquellos lenguajes reconocibles por una computadora.
- Gramáticas tipo 1: Gramática dependientes de contexto.
- Gramáticas tipo 2: Gramáticas libres o independiente de contexto.
- Gramáticas tipo 3: Gramáticas que pueden ser reconocidas por autómatas.

# Gramáticas Dependientes de Contexto

Son aquellos con producciones del estilo:

$$\alpha A \beta \rightarrow \alpha \gamma \beta$$

Donde, $\alpha$ y $\beta$ son cadenas de terminales o no terminales, y $\gamma$ es una cadena no vacía de símbolos, terminales o no terminales.

Algunos lenguajes naturales son descritos con gramáticas dependientes de contexto.

# Gramáticas Libres de Contexto

Intuitivamente, son aquellas en el que contexto no importa. Reglas del tipo:

$$N \rightarrow w$$

Donde, N es un no-terminal, y w una cadena de terminales y no terminales.

Muchos lenguajes de programación pueden ser descritos con gramáticas libres de contexto.

# Gramáticas Regulares

Son las gramáticas más simples de todas, generan aquellos lenguajes reconocidos por autómatas.

Producciones del tipo:

Recursivas a la derecha

$$A \to a$$
$$A \to aB$$

No terminal — Terminal

Recursivas a la izquierda

$$A \to a$$
$$A \to Ba$$

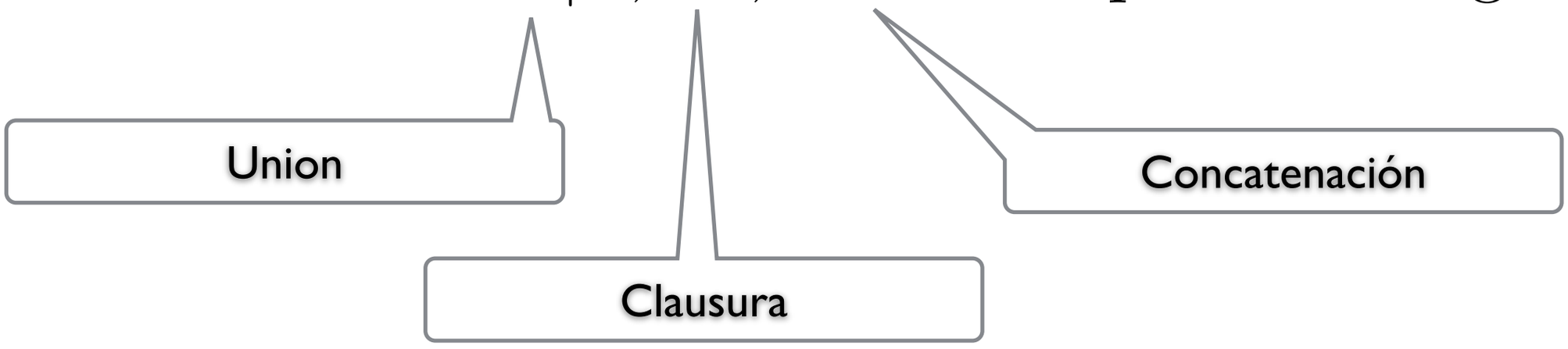
Utiles para definir lenguajes simples que pueden ser reconocidos de una forma eficiente.

# Expresiones Regulares

Las expresiones regulares son *expresiones de texto* que pueden ser utilizados para definir lenguajes aceptados por AFD.

- Los carácteres y el símbolo $\epsilon$ son expresiones regulares.
- Si $E, F$ son expresiones regulares, entonces:

$E|F, E^*, EF$ son expresiones regulares.



# Ejemplos

Veamos Ejemplos:

| ER | Lenguaje que representa |
|---|---|
| $a$ | $\{a\}$ |
| $ab\|cd$ | $\{ab, cd\}$ |
| $a^*bc^*$ | $\{c, ab, bc, abc, aabc, \dots\}$ |

# ER y Programación

Las expresiones regulares son muy utilizadas en programación.

- La herramienta grep, permite buscar las ocurrencias de una expresión regular en un archivo,
- awk permite hacer scripts para el procesamiento de textos
- perl, usa expresiones regulares para hacer procesamiento de textos

# Grep

Grep es una herramienta simple para la búsqueda de expresiones regulares en archivos.

*Global Regular Expression and Print*

Se ejecuta de la siguiente forma:

```
grep regex file
```

expresión regular

archivo

# Grep

La forma más fácil de usar GREP es buscar palabras:

```
grep "GNU" GPL-3
```

General Public License

Muestra todas las lineas donde aparece "GNU" en el archivo

Con -i se pueden obviar mayúsculas/minúsculas

```
grep -i"license" GPL-3
```

Muestra lineas que contienen License o license

# Grep

Con -v podemos buscar lineas que no contienen cierto textos:

```
grep -v "the" GPL-3
```

Busca palabras que no contienen "the" en GPL

Con el carácter ^ grep busca patrones en el comienzo de la línea, con $ al final:

```
grep "^the" GPL-3
```

Líneas que empiecen con "the"

```
grep "the$" GPL-3
```

Líneas que terminen con "the"

# Grep

Con el carácter "." Podemos hacer matching con cualquier carácter:

```
grep "..re" GPL-3
```

Busca palabras que contienen dos letras cualesquiera seguidas de "re"

Con [ ] podemos indicar que grep busque palabra que tengan alguno de los carácteres dentro de los corchetes:

```
grep "^[A-Z]" GPL-3
```

Palabras que empiecen con mayúscula

```
grep "t[wo]o" GPL-3
```

Palabras que empiecen con "t" sigan con "w" ó "o" y terminen con "o"

# Grep

"*" se utiliza para indicar cero o más repeticiones de un patrón:

```
grep "([A-Za-z ]*)" GPL-3
```

Palabras entre paréntesis con espacios

```
grep "^[A-Z].*\." GPL-3
```

Líneas que empiecen con mayúscula, y terminen con un punto

# Grep - Extended Regular Expressions

Grep tiene una opción para usar expresiones regulares extendidas, es decir, en este caso se pueden usar operadores adicionales:

```
grep -E "([0-9]{1,3}[\.]){3}[0-9]{1,3}" myfile.txt
```

Uno a tres dígitos seguidos de un punto

Repetidos 3 veces

Con 1 o 3 dígitos finales

# Algunos Ejemplos

- `grep "^google" file`
- `grep "Limit$" file`
- `grep "...x" file`
- `grep "^[a-z]" myfile`
- `grep "([A-Z][a-z]*)" myfile`
- `grep "^[z-t][a-z]*|^[a-d][a-z]*"`

^ es comienzo de linea

$ fin de linea

tres caracteres y x

Grep tiene muchas más opciones, buscar en google!

En la biblioteca

- Introduction to the Theory of Computation. Michael Sipser (Cap. 1 y 2).
- Introduction to Automata Theory, Languages and Computation. Hopcroft, Motwani, Ullman. (Biblioteca)