

# Tema 2. Análisis léxico

1. Introducción
2. Especificación de un analizador léxico
  - 2.1. Expresiones regulares
  - 2.2. Autómatas finitos
3. Implementación de un analizador léxico
  - 3.1. De una E.R. a un AF mínimo
  - 3.2. Problemas prácticos
4. Generadores de analizadores léxicos

# 1. Introducción

# Funciones del analizador léxico

- Lee secuencia de caracteres y devuelve secuencia de símbolos:
  - *Manejo de fichero de entrada.*
  - *Salta del código fuente comentarios, espacios en blanco, tabuladores,...*
  - *Hace corresponder los mensajes de error con la línea de código donde se produce.*
  - *A veces realiza labores de preprocesador.*
- No tiene porqué ser una fase individual.

# Definiciones

- **Token:**  
grupo o clase de caracteres con un significado colectivo.
- **Lexema:**  
Instancia particular de un token: Secuencia de caracteres en el programa fuente que se forman un token y casan con un patrón.
- **Patrón:**  
Regla que describe cómo se forma un token.  
  
¿Cómo se especifican los tokens (patrones)?  
Mediante expresiones regulares

## **2. Especificación de un A.L.**

### **2.1. Expresiones regulares**

# Repaso: algunas definiciones

**Alfabeto** : Conjunto finito de símbolos. Ej.  $\Sigma = \{a, b, c\}$

**Cadena** sobre un alfabeto: Secuencia finita y ordenada de símbolos de un alfabeto (también llamada **sentencia** o palabra).

**Lenguaje**: Conjunto de cadenas sobre un alfabeto.

Operaciones sobre lenguajes (conjuntos):

**Unión** de L y M,  $L \cup M = \{s \mid s \text{ está en } L \text{ o } s \text{ está en } M\}$

**Concatenación** de L y M

$LM = \{st \mid s \text{ está en } L \text{ y } t \text{ está en } M\}$

**Clausura de Kleene** de L,  $L^* = L^0 \cup L^1 \cup L^2 \cup \dots$

$L^0 = \{\varepsilon\}$ ,  $L^1 = L$ ,  $L^2 = LL$ ,  $L^3 = L^2 L$

**Clausura positiva** de L,  $L^+ = L^1 \cup L^2 \cup \dots$

Ejemplo:

$L = \{aa, bb, cc\}$ ,  $M = \{abc\}$

$\varepsilon$  representa la cadena vacía

# Expresión regular (ER)

- Dado un alfabeto  $\Sigma$ , una expresión regular (ER) sobre  $\Sigma$  se define como:
  - $\emptyset$  es una ER que denota al lenguaje  $\emptyset$ .
  - $\epsilon$  es una ER que denota al lenguaje  $\{\epsilon\}$ .
  - $a \in \Sigma$  es una ER que denota al lenguaje  $\{a\}$ .
  - Si  $r$  y  $s$  son ER denotando a los lenguajes  $L_r$  y  $L_s$  entonces:
    - $r \mid s$  es una ER que denota al lenguaje  $L_r \cup L_s$
    - $r \cdot s$  es una ER que denota al lenguaje  $L_r \cdot L_s$
    - $r^*$  es una ER que denota al lenguaje  $L_r^*$

# Definición regular

- Podemos dar nombres a las ER para construir ER más complejas

–  $d_1 \rightarrow r_1$        $r_1 \in \text{ER sobre } \Sigma^*$

–  $d_2 \rightarrow r_2$        $r_2 \in \text{ER sobre } \Sigma^* \cup \{d_1\}$

– ...

–  $d_n \rightarrow r$        $r_n \in \text{ER sobre } \Sigma^* \cup \{d_1, \dots, d_{n-1}\}$

- Ejemplo:

letra  $\rightarrow A \mid B \mid C \mid \dots \mid Z \mid a \mid b \mid \dots \mid z$

digito  $\rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

identificador  $\rightarrow \text{letra} (\text{letra} \mid \text{digito})^*$



# Ejemplo de definición regular

letra  $\rightarrow a \mid \dots \mid z \mid A \mid \dots \mid Z$

dígito  $\rightarrow 0 \mid \dots \mid 9$

identificador  $\rightarrow \text{letra} (\text{letra} \mid \text{dígito})^*$

constante\_numérica  $\rightarrow \text{constante\_entera} \mid \text{constante\_real}$

constante\_entera  $\rightarrow \text{dígito} (\text{dígito})^*$

constante\_real  $\rightarrow \text{constante\_entera} . (\text{dígito})^*$

símbolo\_especial  $\rightarrow \text{asignación} \mid \text{operador\_relacional}$   
 $\mid \text{subrango} \mid \text{operador\_aritmético}$   
 $\mid \text{separador} \mid \text{palabra\_reservada}$

símbolos  $\rightarrow \text{identificador} \mid \text{constante\_numérica}$   
 $\mid \text{símbolo\_especial}$

## **2.2. Autómatas finitos**

# Reconocedores: AFN

- Un **reconocedor** es un programa que toma una cadena  $x$  como entrada y responde “si” si es una sentencia del lenguaje, o “no” en otro caso.
- Una ER puede ser convertida en un reconocedor construyendo un autómata finito, que podrá ser determinista o no-determinista.
- Un **autómata finito no-determinista (AFN)** es un modelo matemático que consiste en:

$$\text{NFA} = (Q, \Sigma, \delta, q_0, F)$$

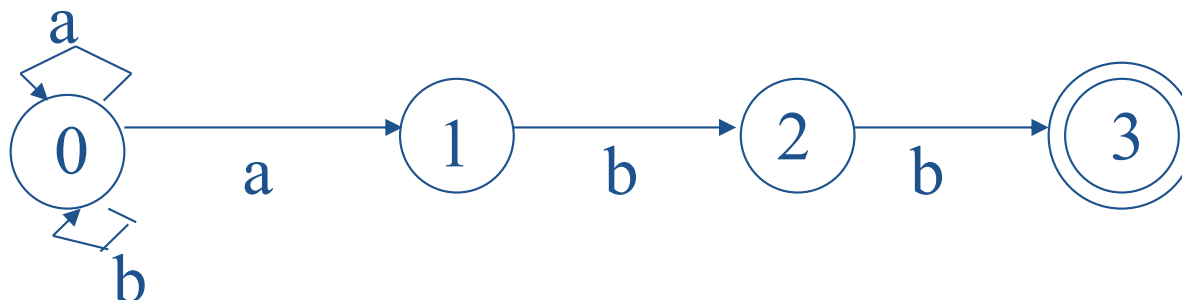
$$F \subseteq Q; \quad q_0 \in Q; \quad \delta: Q \times (\Sigma \cup \{\epsilon\}) \rightarrow P(Q)$$

- Un conjunto de estados  $Q$
- Un conjunto de símbolos de entrada
- Una función de transición.
- Un estado  $q_0$  que se distingue como estado inicial
- Un conjunto de estados  $F$  que se distinguen como estados finales.

$P(Q)$  representa “partes de  $Q$ ”: todos los posibles subconjuntos formados con elementos de

# AFN

- Un AFN es **no-determinista** si:
  - Un mismo símbolo puede etiquetar dos o más transiciones que salen del mismo estado, o
  - La cadena vacía etiqueta alguna transición.
- Un AFN **acepta** una cadena de entrada  $x$  si y solo si hay algún camino en el grafo de transición etiquetado con los símbolos de la cadena desde el estado inicial a algún estado de aceptación.
- Ejemplo:
  - ¿Qué lenguaje reconoce el AFN?



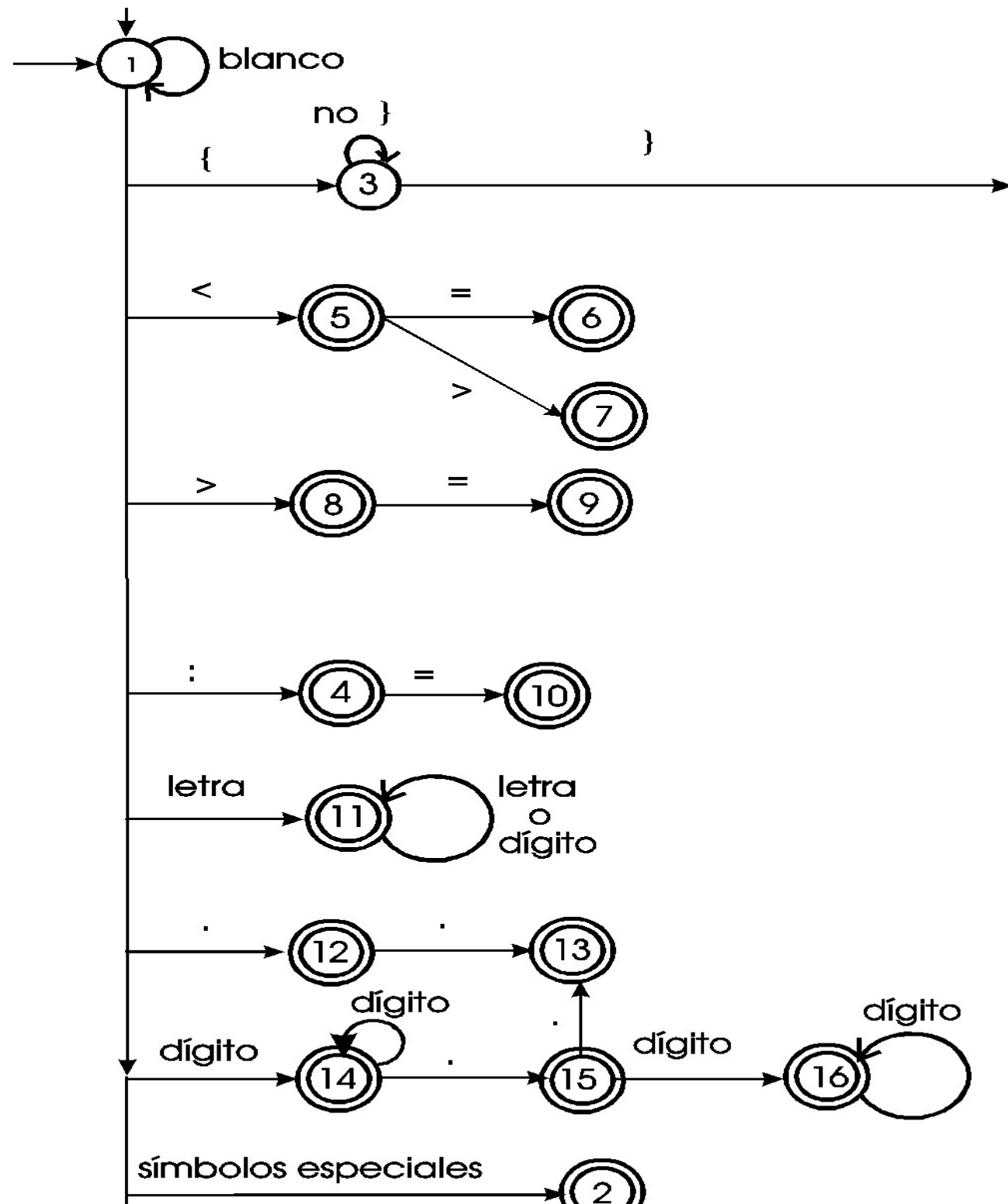
# AFD

- Se puede mejorar la complejidad temporal usando autómatas finitos deterministas en lugar de AFN.
- Un AF es **determinista** (AFD) si
  - No hay transiciones vacías, y
  - No hay ningún estado con más de un arco de salida etiquetado con el mismo símbolo

$$\text{AFD} = (Q, \Sigma, \delta, q_0, F)$$

$$F \subseteq Q; \quad q_0 \in Q; \quad \delta: Q \times \Sigma \longrightarrow Q$$

# Ejemplo



## **3. Implementación de un A.L.**

### **3.1. De ER a AF mínimo**

# Equivalencias

E.R.  $\leftrightarrow$  AFND con  $\varepsilon$ -trans  $\leftrightarrow$  AFND  $\leftrightarrow$  AFD

- Construir un AFN para reconocer la unión de todos los patrones
- Convertir el AFN en un AFN sin  $\varepsilon$ -trans
- Convertir el AFN en un AFD.
- Minimizar el AFD.
- Implementar el AFD

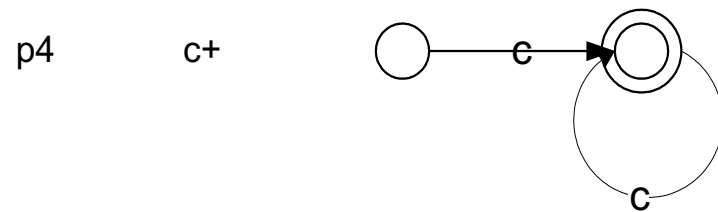
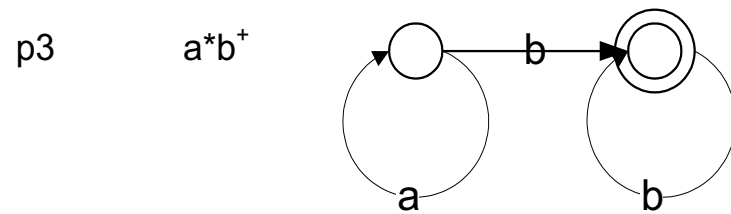
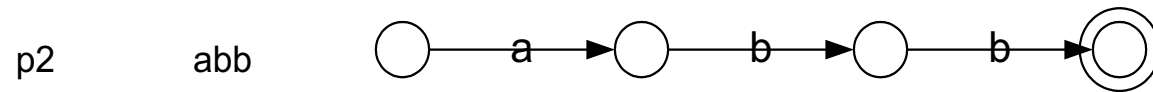


## 3.2. Problemas prácticos

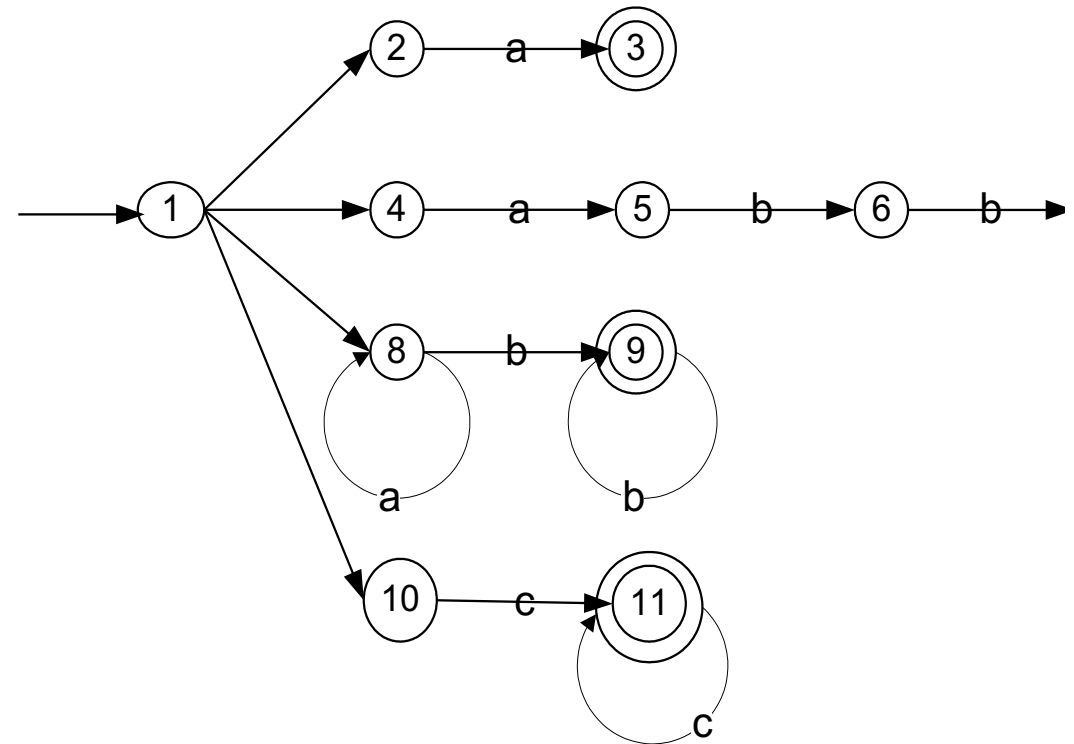
# Problemas prácticos

- El analizador sintáctico **llama** al léxico y éste último lo devuelve un token.
- Cuando el analizador léxico encuentra "**==**" ¿Qué token o tokens reconoce?
  - Reconoce siempre el token más largo
  - Usa varios caracteres de anticipación
- Las **palabras clave** pueden:
  - Escribirse como expresiones regulares y ser incluidas en el AFD. Esto incrementará el tamaño del AFD
  - Tratarlas como excepciones de los identificadores (buscándolas en una tabla de **palabras reservadas**)

# Problemas prácticos



$\downarrow$        $\downarrow$   
a a b a b b a c c a  
            $\uparrow$



# Identación y a. léxico: Python

Un programa Python está dividido en **líneas lógicas** acabadas con el token "NEWLINE"

Cada línea lógica puede estar formada por varias **líneas físicas** unidas por un barra "\"

Ej.

```
if Parcial_1 >= 5 and Parcial_2 >= 5 \
    and Evaluacion_continua >= 5 : return 1
```

- Se lleva la cuenta de los espacios en blanco (y tabuladores) al principio de una línea lógica para determinar la indentación y por lo tanto la agrupación de instrucciones.

# Identación y a. léxico: El caso de Python

- El a. léxico se apoya en una **pila de enteros**. Inicialmente apila el valor cero.

Para cada línea lógica:

- Si número de espacios en blanco a principio de la línea es mayor que el número en la cima de la pila
  - Apilar el número de espacios en blanco
  - General el token IDENT
- Si número de espacios en blanco a principio de la línea es menor que el número en la cima de la pila
  - Mientras la cima de la pila sea mayor que el número de espacios en la línea
    - Desapilar el número
    - General el token DEDENT

Al llegar al final del fichero desapilar todos los números de la pila generando un token DEDENT para cada uno de ellos.

## 4. Generadores de a. léxicos

# Flex

---

Ver seminario de prácticas