



# Parseo y Generación de Código

24 de agosto de 2017

**Analizadores léxicos**

**Autómatas finitos**

**Expresiones regulares**

Licenciatura en Informática con Orientación en Desarrollo de Software  
Universidad Nacional de Quilmes

# Analizadores léxicos

# Problemas de análisis léxico

Problemas que uno quisiera resolver programando:

- ▶ Implementar herramientas de búsqueda tipo *glob* o *grep*.
  - ▶ \*.txt (*sintaxis tipo Unix*)
  - ▶ (Jorge Luis|J. L.) Borges
  - ▶ Jorge( )+Luis( )+Borges
  - ▶ [jJ]orge [lL]uis [bB]orges
  - ▶ Centro Cultural (General )?San Martín
  - ▶ [GJX]imena
  - ▶ [a-z]\*@unq.edu.ar
  - ▶ 192.168.[0-9]+.1

Usos comunes:

- ▶ Extraer datos de fuentes que no están normalizadas.
- ▶ Buscar y reemplazar en archivos de texto, bases de datos, código fuente, paquetes de red.

# Problemas de análisis léxico

Más problemas que uno quisiera resolver programando:

- ▶ Implementar analizadores léxicos.

- ▶ `[_a-zA-Z][_a-zA-Z0-9]*`
- ▶ `0x[0-9a-f]+`
- ▶ `//[^\n]+`

Usos comunes:

- ▶ Componente de un intérprete o compilador.
- ▶ Resaltado de sintaxis.
- ▶ Algunos preprocesadores como `cpp` trabajan a nivel léxico.
- ▶ Especificar o documentar la sintaxis de un lenguaje formal.

Por ejemplo, un protocolo:

- ▶ `((READ|WRITE) [a-z]+\n)*`

# Un analizador léxico sencillo

Para empezar, veremos cómo implementar manualmente un **analizador léxico** (alias **lexer**, **tokenizer**, **scanner**) para un lenguaje con las siguientes convenciones léxicas.

- ▶ Palabras clave: `if`, `else`.
- ▶ Símbolos: `{`, `}`, `=`, `==`.
- ▶ Identificadores: `[a-z]+` (excepto las palabras clave).
- ▶ Números: `[0-9]+`
- ▶ Se ignoran espacios, tabs y enters.
- ▶ Comentarios comienzan con `#` y terminan al final de la línea.

# Un analizador léxico sencillo – en C

Los símbolos terminales o *tokens* son un tipo enumerado:

```
typedef enum {  
    T_IF,  
    T_ELSE,  
    T_LBRACE,           // {  
    T_RBRACE,           // }  
    T_ASSIGN,           // =  
    T_EQUAL,            // ==  
    T_ID,  
    T_NUM,  
    T_EOF,  
} Token;
```

# Un analizador léxico sencillo – en C

Representación del analizador léxico:

```
#define MAX_BUFFER 1024
```

```
typedef struct {  
    FILE *archivo;  
    int linea;  
    int columna;  
    char buffer[MAX_BUFFER];  
} Tokenizador;
```

```
void inicializar(Tokenizador *t, FILE *archivo) {  
    t->archivo = archivo;  
    t->linea = 1;  
    t->columna = 1;  
}
```

## Un analizador léxico sencillo – en C

```
void comer_comentario(Tokenizador *t) {
    int c = fgetc(t->archivo);
    while (c != EOF && !es_fin_de_linea(c)) {
        t->columna++;
        c = fgetc(t->archivo);
    }
    ungetc(c, t->archivo);
}
```



## Un analizador léxico sencillo – en C

```
void comer_espacios(Tokenizador *t) {
    int c = fgetc(t->archivo);
    while (es_espacio(c) || c == '#') {
        if (c == '#') {
            t->columna++;
            comer_comentario();
        } else if (es_fin_de_linea(c)) {
            t->linea++;
            t->columna = 1;
        } else {
            t->columna++;
        }
        c = fgetc(t->archivo);
    }
    ungetc(c, t->archivo);
}
```

## Un analizador léxico sencillo – en C

```
Token siguiente_token(Tokenizador *t) {
    comer_espacios(t);
    int c = fgetc(t->archivo);
    if (c == EOF) {
        return T_EOF;
    } else if (c == '{') {
        t->columna++;
        return T_LBRACE;
    } else if (c == '}') {
        t->columna++;
        return T_RBRACE;
    } else if (c == '=') {
        t->columna++;
        c = fgetc(t->archivo);
        if (c == '=') {
            t->columna++;
            return T_ASSIGN;
        } else {
            ungetc(c, t->archivo);
            return T_EQUAL;
        }
    }
}
```

## Un analizador léxico sencillo – en C

```
} else if (es_numerico(c)) {  
    int i = 0;  
    t->columna++;  
    while (es_numerico(c) && i + 1 < MAX_BUFFER) {  
        t->buffer[i] = c;  
        i++;  
        c = fgetc(t->archivo);  
        t->columna++;  
    }  
    t->columna--;  
    t->buffer[i] = '\\0';  
    ungetc(c, t->archivo);  
    return T_NUM;
```

## Un analizador léxico sencillo – en C

```
} else if (es_alfabetico(c)) {
    int i = 0;
    t->columna++;
    while (es_alfabetico(c) && i + 1 < MAX_BUFFER) {
        t->buffer[i] = c;
        i++;
        c = fgetc(t->archivo);
        t->columna++;
    }
    t->columna--;
    t->buffer[i] = '\0';
    ungetc(c, t->archivo);

    if (!strcmp(t->buffer, "if")) {
        return T_IF;
    } else if (!strcmp(t->buffer, "else")) {
        return T_ELSE;
    } else {
        return T_ID;
    }
}
```

## Un analizador léxico sencillo – en C

```
    } else {  
        fprintf(stderr, "Simbolo desconocido "  
                      "en linea %u columna %u\n",  
                      t->linea, t->columna);  
        exit(1);  
    }  
}
```

# Limitaciones del analizador léxico sencillo

- ▶ **Limitación de la longitud de identificadores.** Se puede solucionar haciendo manejo dinámico de memoria. No se hace en el ejemplo anterior por una cuestión didáctica – para no complicar más el código.
- ▶ **Entrada/salida.** Leer de a un caracter de la entrada por vez puede llegar a ser muy lento. Un tokenizador realista debería leer la entrada de a fragmentos (p.ej. de a 64Kb). Esto requiere cierto cuidado. Ver Sec. 3.2 del libro del Dragón.
- ▶ **Estilo cuestionable: “if .. else if .. else if .. else”.** No es un gran problema en un analizador léxico<sup>1</sup>. Se puede mejorar el estilo usando un *dispatch* basado en una tabla o diccionario.

---

<sup>1</sup>Ejercicio: ¿cómo está hecho el analizador léxico de tu lenguaje favorito?

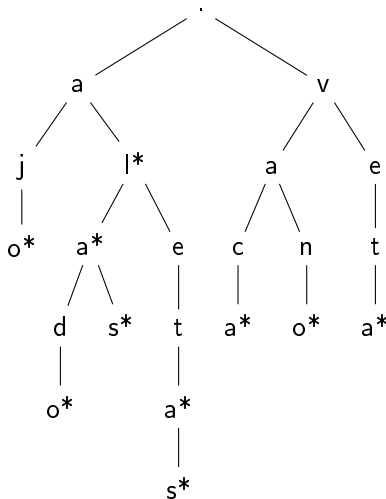
## Limitaciones del analizador léxico sencillo

- **Uso de “if .. else if ...” para distinguir palabras clave.**

Este es un problema algorítmico más importante/interesante.

Se puede solucionar usando un *trie*.

ajo  
al  
ala  
alado  
alas  
aleta  
aletas  
vaca  
vano  
veta



# Limitaciones del analizador léxico sencillo

- ▶ **Naturaleza ad hoc.**

El mayor problema del analizador léxico anterior es que es una solución *ad hoc*.

- ▶ *Ideal utópico de la computación*: el programador describe el problema y la computadora lo resuelve sin que el programador indique cómo. La computadora no hace magia: utiliza técnicas generales de resolución de problemas.
- ▶ *Aproximación al ideal utópico*: programar soluciones generales a algunos problemas particulares.
- ▶ *Aproximación al ideal utópico en este caso*: si el programador especifica la sintaxis léxica se puede generar automáticamente un analizador léxico.



# Autómatas finitos

# Autómatas finitos determinísticos

Un **autómata finito determinístico** (AFD) es una 5-upla  $D = (Q, \Sigma, \delta, q_0, Q_F)$ , donde:

- ▶  $Q$  es un conjunto finito, el **conjunto de estados**.
- ▶  $\Sigma$  es un conjunto finito de símbolos, el **alfabeto**.
- ▶  $\delta : Q \times \Sigma \rightarrow Q$  es una función, la **función de transición**.
- ▶  $q_0 \in Q$  es un estado, el **estado inicial**.
- ▶  $Q_F \subseteq Q$  es un conjunto de estados, los **estados finales**.

# Autómatas finitos determinísticos

La función de transición se extiende a palabras, definiendo una relación ternaria  $\rightarrow_D \subseteq Q \times \Sigma^* \times Q$  que relaciona una tripla  $(q_1, \alpha, q_2)$  si se llega del estado  $q_1$  al estado  $q_2$  consumiendo la cadena  $\alpha$ . Se escribe  $q_1 \xrightarrow{\alpha}_D q_2$  si la tripla  $(q_1, \alpha, q_2)$  está relacionada. Más precisamente:

$$\begin{array}{ll} q \xrightarrow{\epsilon}_D q & \text{para todo estado } q \in Q \\ q \xrightarrow{a\alpha}_D q'' & \text{si y sólo si existe } q' \in Q \text{ tal que } q \xrightarrow{a}_D q' \xrightarrow{\alpha}_D q'' \end{array}$$

Es decir:

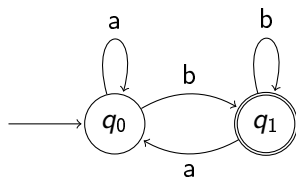
$$q_0 \xrightarrow{a_1 a_2 \dots a_n}_D q_n$$

si y sólo si existen  $q_1, \dots, q_{n-1}$  tales que

$$q_0 \xrightarrow{a_1}_D q_1 \xrightarrow{a_2}_D q_2 \dots \xrightarrow{a_n}_D q_n$$

# Autómatas finitos determinísticos

**Ejemplo.**



$$D = (\{q_0, q_1\}, \{a, b\}, \delta, q_0, \{q_1\})$$

| $\delta$ | $a$   | $b$   |
|----------|-------|-------|
| $q_0$    | $q_0$ | $q_1$ |
| $q_1$    | $q_0$ | $q_1$ |

$$q_0 \xrightarrow{abbabba} q_1 \text{?}$$

# Autómatas finitos determinísticos

Un AFD  $D = (Q, \Sigma, \delta, q_0, Q_F)$  **acepta** una cadena  $\alpha \in \Sigma^*$  si  $q_0 \xrightarrow{\alpha}_D q'$  donde  $q' \in Q_F$  es algún estado final.

El **lenguaje aceptado** por  $D$  es el conjunto de cadenas que acepta:

$$L(D) = \{\alpha \in \Sigma^* \mid q_0 \xrightarrow{\alpha}_D q', \quad \text{donde } q' \in Q_F\}$$

# Autómatas finitos determinísticos

**Ejercicio.** Definir un AFD en el alfabeto  $\{a, b\}$  que acepte el lenguaje de las cadenas terminadas en *abb*.

# Autómatas finitos determinísticos

## Algoritmo: simulación de un AFD.

*Entrada:* un AFD  $D = (Q, \Sigma, \delta, q_0, Q_F)$

una cadena  $\alpha \in \Sigma^*$

*Salida:* un booleano indicando si  $\alpha \in L(D)$

$q := q_0$

**foreach**  $x$  in  $\alpha$

$q := \delta(q, x)$

**end**

**return**  $q \in Q_F$

- ▶ ¿Cuánta memoria auxiliar necesita?
- ▶ ¿Cuánto tarda en decidir si una cadena  $\alpha$  está en  $L(D)$ ?

# Autómatas finitos no determinísticos

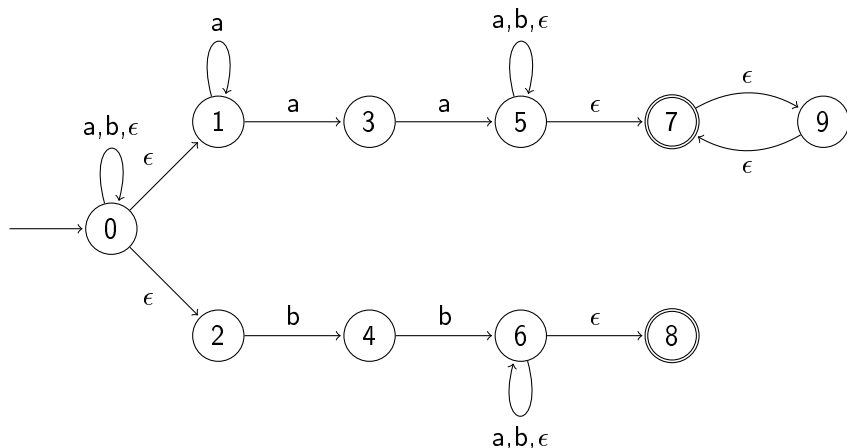
Un **autómata finito no determinístico** (AFN) es una 5-upla  $(Q, \Sigma, \delta, q_0, Q_F)$ , donde:

- ▶  $Q$  es un conjunto finito, el **conjunto de estados**.
- ▶  $\Sigma$  es un conjunto finito de símbolos, el **alfabeto**.
- ▶ Para cada estado  $q \in Q$  y cada símbolo  $x \in \Sigma \cup \{\epsilon\}$ , la expresión  $\delta(q, x)$  denota un conjunto de estados.  
Más precisamente,  $\delta : Q \times (\Sigma \cup \{\epsilon\}) \rightarrow \mathcal{P}(Q)$  es la **función de transición no determinística**.
- ▶  $q_0 \in Q$  es un estado, el **estado inicial**.
- ▶  $Q_F \subseteq Q$  es un conjunto de estados, los **estados finales**.

La función de transición no determinística  $\delta$  se extiende a palabras.  
Pero veamos primero un ejemplo.



# Autómatas finitos no determinísticos



Todavía no definimos formalmente aceptación para AFNs, pero podemos preguntarnos:

- ¿Este AFN acepta la cadena *abbab*?
- ¿Qué lenguaje acepta?

## Autómatas finitos no determinísticos

Dado un AFN  $N = (Q, \Sigma, \delta, q_0, Q_F)$ , se define la relación binaria  $\xRightarrow{\epsilon}_N \subseteq Q \times Q$  de tal manera que un par de estados  $(q_1, q_2)$  está relacionado si se puede llegar de  $q_1$  a  $q_2$  usando solamente transiciones etiquetadas con  $\epsilon$ . Más precisamente:

$$\begin{aligned} q &\xRightarrow{\epsilon}_N q && \text{para todo estado } q \in Q \\ q &\xRightarrow{\epsilon}_N q'' && \text{si existe un estado } q' \in Q \text{ tal que} \\ &&& q' \in \delta(q, \epsilon), \quad q' \xRightarrow{\epsilon}_N q'' \end{aligned}$$

La **clausura- $\epsilon$**  de un estado  $q$  es el conjunto de estados alcanzables por medio de transiciones  $\epsilon$ :

$$\text{cl}_{\epsilon}(q) = \{q' \in Q \mid q \xRightarrow{\epsilon}_N q'\}$$

La clausura- $\epsilon$  de un conjunto de estados  $\{q_1, \dots, q_n\} \subseteq Q$  se define como la unión de sus respectivas clausuras:

$$\text{cl}_{\epsilon}(\{q_1, \dots, q_n\}) = \bigcup_{i=1}^n \text{cl}_{\epsilon}(q_i)$$

# Autómatas finitos no determinísticos

## Algoritmo: cómputo de la clausura- $\epsilon$ .

*Entrada:* un AFN  $N = (Q, \Sigma, \delta, q_0, Q_F)$

un conjunto de estados  $A \subseteq Q$

*Salida:* el conjunto de estados  $C = \text{cl}_\epsilon(A)$

$C := A$

**while** existe un estado  $q \in C$  tal que  $\delta(q, \epsilon) \not\subseteq C$

$C := C \cup \delta(q, \epsilon)$

**end**

**return**  $C$

# Autómatas finitos no determinísticos

## Algoritmo: cómputo de la clausura- $\epsilon$ .

*Variante más concreta/explicita.*

*Entrada:* un AFN  $N = (Q, \Sigma, \delta, q_0, Q_F)$

un conjunto de estados  $A \subseteq Q$

*Salida:* el conjunto de estados  $C = \text{cl}_\epsilon(A)$

*pila* := []

$C := \emptyset$

meter todos los estados de  $A$  en *pila*

**while** *pila*  $\neq$  []

$q := \text{pila.pop}()$

**foreach**  $q' \in \delta(q, \epsilon)$

**if**  $q' \notin C$

$C := C \cup \{q'\}$

*pila.push*( $q'$ )

**end**

**end**

**end**

**return**  $C$

# Autómatas finitos no determinísticos

La función de transición no determinística  $\delta$  se extiende a palabras, definiendo una relación ternaria  $\rightarrow_N: Q \times \Sigma^* \times Q$  de tal manera que una tripla  $(q_1, \alpha, q_2)$  está relacionada si se puede llegar del estado  $q_1$  al estado  $q_2$  consumiendo la cadena  $\alpha$ . Se escribe  $q_1 \xrightarrow{\alpha}_N q_2$  si la tripla  $(q_1, \alpha, q_2)$  está relacionada. Más precisamente:

$$\begin{aligned} q &\xrightarrow{\epsilon}_N q' && \text{si } q \xRightarrow{\epsilon}_N q' \\ q &\xrightarrow{a\alpha}_N q''' && \text{si existen } q', q'' \in Q \text{ tales que} \\ &&& q \xRightarrow{\epsilon}_N q', \quad q'' \in \delta(q', a), \quad q'' \xrightarrow{\alpha}_N q''' \end{aligned}$$

# Autómatas finitos no determinísticos

Un AFN  $N = (Q, \Sigma, \delta, q_0, Q_F)$  **acepta** una cadena  $\alpha \in \Sigma^*$  si  $q_0 \xrightarrow{\alpha}_N q'$  donde  $q' \in Q_F$  es algún estado final.

El **lenguaje aceptado** por  $N$  es el conjunto de cadenas que acepta:

$$L(N) = \{\alpha \in \Sigma^* \mid q_0 \xrightarrow{\alpha}_N q', \quad \text{donde } q' \in Q_F\}$$

## Ejemplo.

- En el AFN de antes, determinar el conjunto de estados  $q'$  tales que:

$$q_0 \xrightarrow{abb}_N q'$$

# Autómatas finitos no determinísticos

## Algoritmo: simulación de un AFN.

*Entrada:* un AFN  $N = (Q, \Sigma, \delta, q_0, Q_F)$

una cadena  $\alpha \in \Sigma^*$

*Salida:* un booleano indicando si  $\alpha \in L(N)$

$S := \text{cl}_\epsilon(q_0)$

**foreach**  $x$  in  $\alpha$

$S := \text{cl}_\epsilon(\bigcup_{q \in S} \delta(q, x))$

**end**

**return**  $S \cap Q_F \neq \emptyset$

# Equivalencia entre AFDs y AFNs

- ▶ Dado un AFD  $D$ , es inmediato construir un AFN que acepte el mismo lenguaje que  $D$ .
- ▶ Dado un AFN  $N$ , ¿se puede construir un AFD que acepte el mismo lenguaje que  $D$ ?



# Equivalencia entre AFDs y AFNs

- ▶ Dado un AFD  $D$ , es inmediato construir un AFN que acepte el mismo lenguaje que  $D$ .
- ▶ Dado un AFN  $N$ , ¿se puede construir un AFD que acepte el mismo lenguaje que  $D$ ? ¡Sí!

# Equivalencia entre AFDs y AFNs

## Construcción de subconjuntos.

Si  $N = (Q, \Sigma, \delta, q_0, Q_F)$  es un AFN, podemos construir el siguiente AFD  $D = (\mathcal{P}(Q), \Sigma, \Delta, \text{cl}_\epsilon(q_0), S_F)$ :

- ▶ Un estado de  $D$  es un **conjunto**  $S \subseteq Q$ .  
( $S$  es subconjunto de los estados de  $N$ ).
- ▶ El alfabeto es el mismo.
- ▶ La función de transición

$$\Delta : \mathcal{P}(Q) \times \Sigma \rightarrow \mathcal{P}(Q)$$

está dada por:

$$\Delta(S, x) = \text{cl}_\epsilon(\cup_{q \in S} \delta(q, x))$$

- ▶ El estado inicial es el conjunto  $\text{cl}_\epsilon(q_0)$ .
- ▶ Un conjunto de estados del AFN es un estado final para el AFD si contiene algún estado final:

$$S_F = \{S \subseteq Q \mid S \cap Q_F \neq \emptyset\}$$

## Equivalencia entre AFDs y AFNs

**Teorema.** Si  $N$  es un AFN y  $D$  es el AFD que resulta de la construcción de subconjuntos de  $N$ , entonces  $N$  y  $D$  aceptan el mismo lenguaje.

*Demostración.* No vamos a probarlo rigurosamente, pero la observación esencial es la siguiente propiedad técnica<sup>2</sup>:

$$\begin{array}{ll} \text{cl}_\epsilon(q_0) \xrightarrow{\alpha}_D S & \text{en el AFD construido} \\ \text{si y sólo si} & \\ S = \{q' \in Q \mid q_0 \xrightarrow{\alpha}_N q'\} & \text{en el AFN original} \end{array}$$

Por lo tanto dada una cadena cualquiera  $\alpha$ :

$$\begin{array}{ll} \text{El AFD } D \text{ acepta } \alpha & \text{si y sólo si } \text{cl}_\epsilon(q_0) \xrightarrow{\alpha}_D S \in S_F \\ & \text{si y sólo si } q_0 \xrightarrow{\alpha}_N q \text{ para algún } q \in Q_F \\ & \text{si y sólo si } \text{el AFN } N \text{ acepta } \alpha. \end{array}$$

---

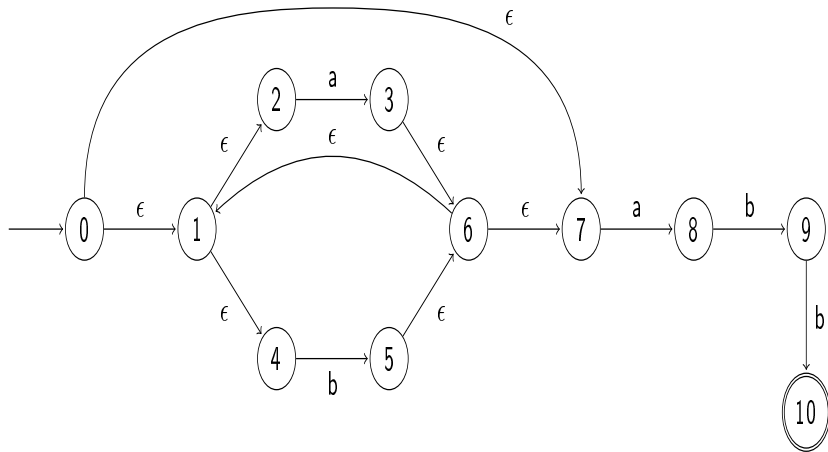
<sup>2</sup>Se puede ver demostrando algo un poco más general:

$$\text{cl}_\epsilon(S) \xrightarrow{\alpha}_D S' \iff S' = \{q' \in Q \mid \exists q \in S. q \xrightarrow{\alpha}_N q'\}$$

Por inducción en la longitud de la cadena  $\alpha$ .

## Equivalencia entre AFDs y AFNs

**Ejercicio.** Construir un AFD que acepte el mismo lenguaje que el siguiente AFN en el alfabeto  $\{a, b\}$ .



## Equivalencia entre AFDs y AFNs

- ▶ Si un AFN tiene  $n$  estados, ¿cuántos estados tiene el AFD que resulta de la construcción de subconjuntos?
- ▶ **Observación:** no es necesario incluir todos estos subconjuntos de estados en la construcción del AFD, basta con incluir los subconjuntos de estados **alcanzables**.

# Expresiones regulares

# Operaciones entre lenguajes

Si  $\Sigma$  es un alfabeto y  $L, L' \subseteq \Sigma^*$  son lenguajes, definimos las siguientes operaciones:

- ▶ **Concatenación.**  $L \cdot L' = \{\alpha\beta \mid \alpha \in L, \beta \in L'\}$ .
- ▶ **Unión.**  $L \cup L' = \{\alpha \mid \alpha \in L \vee \alpha \in L'\}$ .  
(Es la unión de conjuntos).
- ▶ **Concatenación de un lenguaje consigo mismo.**

$$\begin{aligned} L^0 &= \{\epsilon\} \\ L^{n+1} &= L \cdot L^n \end{aligned}$$

- ▶ **Clausura de Kleene.**  $L^* = \bigcup_{i=0}^{\infty} L^i$   
Es decir,  $\alpha \in L^*$  si existe  $n \geq 0$  y existen palabras  $\alpha_1, \dots, \alpha_n$  tales que  $\alpha = \alpha_1 \dots \alpha_n$  y  $\alpha_i \in L$  para cada  $i$ .  
Notar que  $\epsilon \in L^*$  siempre.
- ▶ **Clausura positiva.**  $L^+ = \bigcup_{i=1}^{\infty} L^i$   
Es decir,  $\alpha \in L^+$  si existe  $n \geq 1$  y existen palabras  $\alpha_1, \dots, \alpha_n$  tales que  $\alpha = \alpha_1 \dots \alpha_n$  y  $\alpha_i \in L$  para cada  $i$ .  
Notar que  $\epsilon \in L^+$  si y solamente si  $\epsilon \in L$ .

# Operaciones entre lenguajes

**Ejemplo.** Si el alfabeto es  $\Sigma = \{a, b, c, d\}$  y tenemos los lenguajes:

$$L_1 = \{aa, bb\}$$

$$L_2 = \{ccc, d\}$$

Entonces:

$$L_1 \cdot L_2 = \{aaccc, aad, bbccc, bbd\}$$

$$L_1 \cup L_2 = \{aa, bb, ccc, d\}$$

$$(L_1)^2 = \{aaaa, aabb, bbaa, bbbb\}$$

$$(L_2)^* = \{\epsilon, ccc, d, cccccc, cccd, dccc, dd, ccccccccc, \dots\}$$



# Expresiones regulares

Dado un alfabeto  $\Sigma$ , las **expresiones regulares** en el alfabeto  $\Sigma$  son expresiones (es decir, *árboles*) que se construyen inductivamente con las siguientes reglas:

- ▶ El símbolo  $\emptyset$  es una expresión regular.
- ▶ El símbolo  $\epsilon$  es una expresión regular.
- ▶ Cualquier símbolo  $x \in \Sigma$  es una expresión regular.
- ▶ Si  $R$  y  $S$  son expresiones regulares,  $R \cdot S$  es una expresión regular. Se abrevia  $RS$ .
- ▶ Si  $R$  y  $S$  son expresiones regulares,  $R \mid S$  es una expresión regular.
- ▶ Si  $R$  es una expresión regular,  $R^*$  es una expresión regular.

# Expresiones regulares

Cada expresión regular  $R$  **denota** un lenguaje. Inductivamente:

- ▶  $L(\emptyset) = \emptyset$
- ▶  $L(\epsilon) = \{\epsilon\}$
- ▶ Si  $x \in \Sigma$ , entonces  $L(x) = \{x\}$ .
- ▶  $L(R \cdot S) = L(R) \cdot L(S)$
- ▶  $L(R \mid S) = L(R) \cup L(S)$
- ▶  $L(R^*) = L(R)^*$

# Expresiones regulares

Convenciones:

- ▶ El operador de mayor precedencia es la clausura ( $\star$ ), seguido por la concatenación ( $\cdot$ ), seguido por la unión ( $|$ ).

- ▶ **Ejemplo:**

$$aba^\star | bab^\star = ((ab)(a^\star)) | ((ba)(b^\star))$$

- ▶ Las expresiones regulares:

$$R | (S | T) \qquad (R | S) | T$$

son distintas, pero generalmente se pueden identificar.

- ▶ Las expresiones regulares:

$$R \cdot (S \cdot T) \qquad (R \cdot S) \cdot T$$

son distintas, pero generalmente se pueden identificar.

# Expresiones regulares

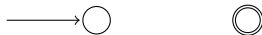
**Ejercicio.** Describir el lenguaje generado por la siguiente gramática en el alfabeto  $\{a, b, c, d, e, f\}$  usando una expresión regular:

$$\begin{aligned} S &\rightarrow A \mid C \\ A &\rightarrow abA \mid B \\ B &\rightarrow cB \mid dB \mid \epsilon \\ C &\rightarrow Ce \mid f \end{aligned}$$

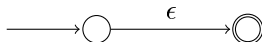
# Construcción de Thompson

Dada una expresión regular  $R$ , se puede construir un AFN  $N(R)$  que acepta el lenguaje denotado por  $R$ . Inductivamente, se puede construir un autómata que tiene un único estado final:

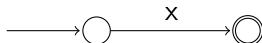
- Construcción de  $N(\emptyset)$ :



- Construcción de  $N(\epsilon)$ :

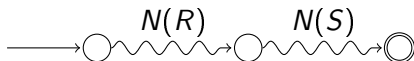


- Construcción de  $N(x)$  si  $x \in \Sigma$ :

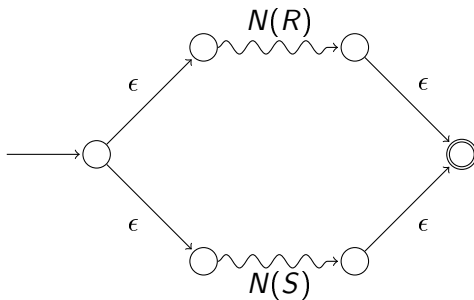


# Construcción de Thompson

- Construcción de  $N(RS)$ :

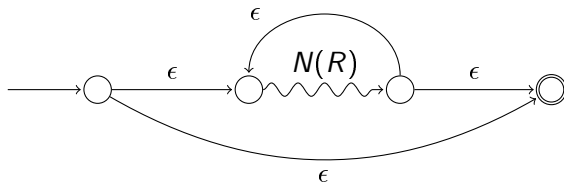


- Construcción de  $N(R \mid S)$ :



# Construcción de Thompson

- Construcción de  $N(R^*)$ :



# Construcción de Thompson

**Ejercicio.** Usando la construcción de Thompson, construir un AFN que acepte el lenguaje denotado por:

$$0^*1^* \mid 1^*0^*$$