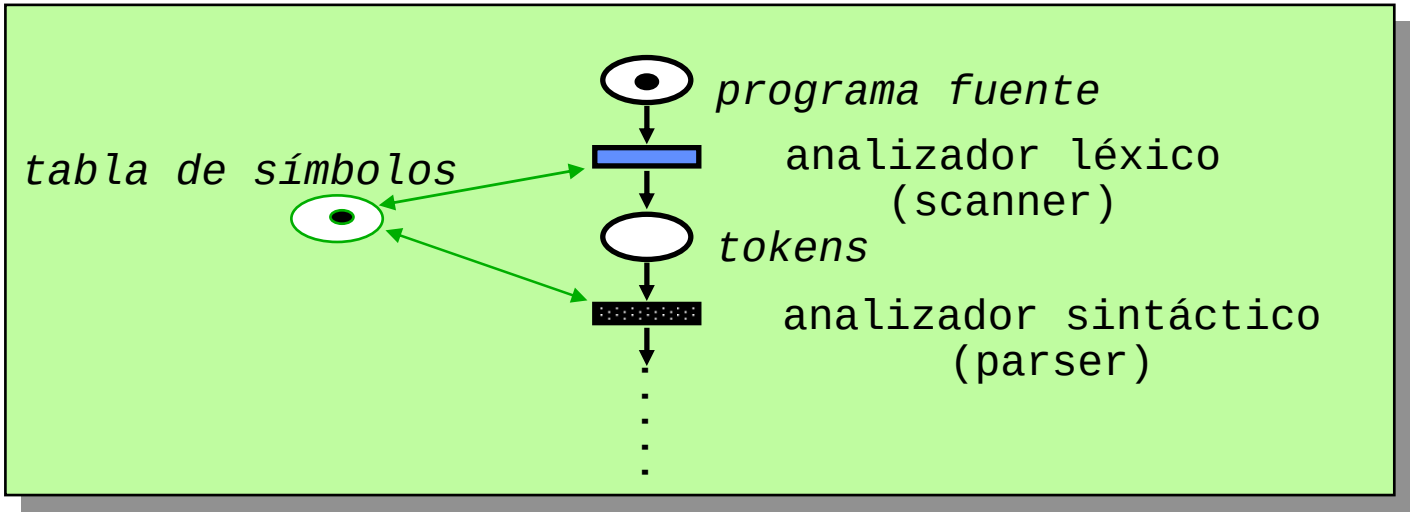
1) El papel de un analizador léxico (scanner)
2) Tokens, lexemas y patrones léxicos
3) Expresiones regulares
   - Su utilidad en compilación
   - Definición
   - Ejemplos
   - Notaciones
4) Autómatas finitos
   - Generalidades
   - Grafo de transiciones asociado a un AF
   - Aceptación de un string por un AF
5) Conversión de una expresión regular en un AFN
6) Transformación de un AFN en un AFD
7) Minimización de un AFD
8) Una introducción breve a la implementación de analizadores léxicos

El papel del analizador léxico

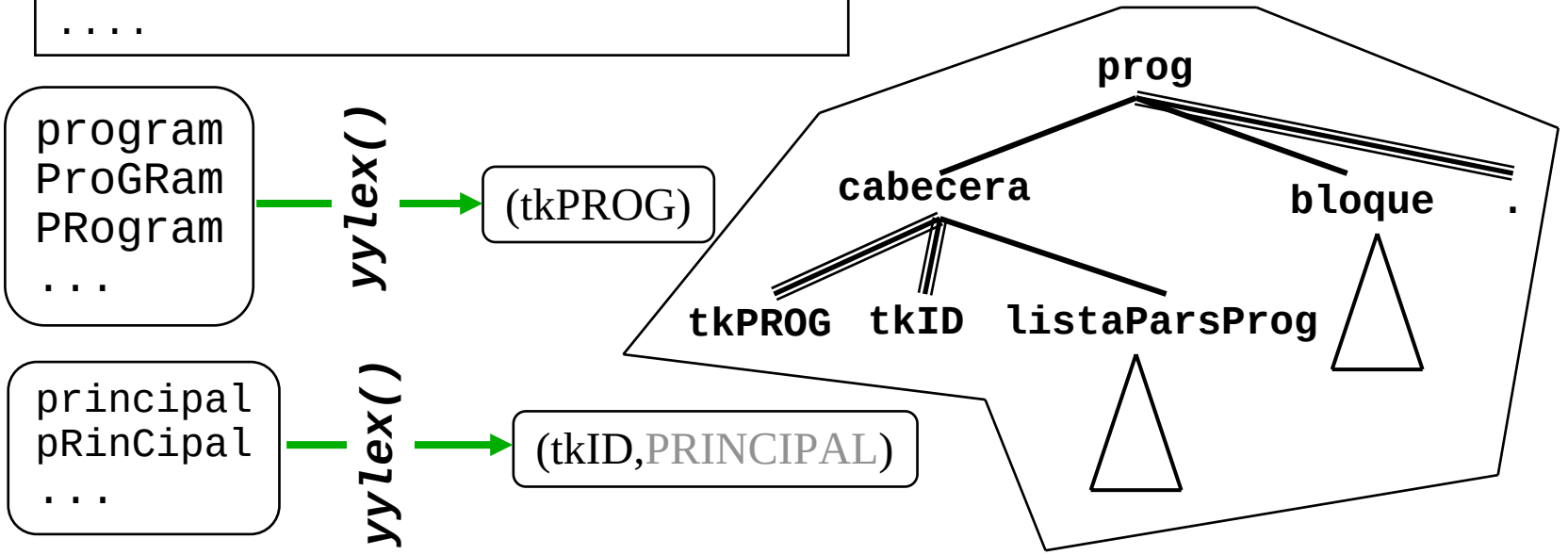
- La función primordial es agrupar caracteres de la entrada en tokens
- Estos tokens son suministrados ("bajo demanda") al analizador sintáctico



El papel del analizador léxico

```
prog: cabecera bloque '.'
cabecera: tkPROG tkID
cabecera: tkPROG tkID listaParsProg
listaParsProg: '(' listaID ')'
listaID: listaID ',' ID
listaID: ID
....
```

```
program principal(input,output);
....
```



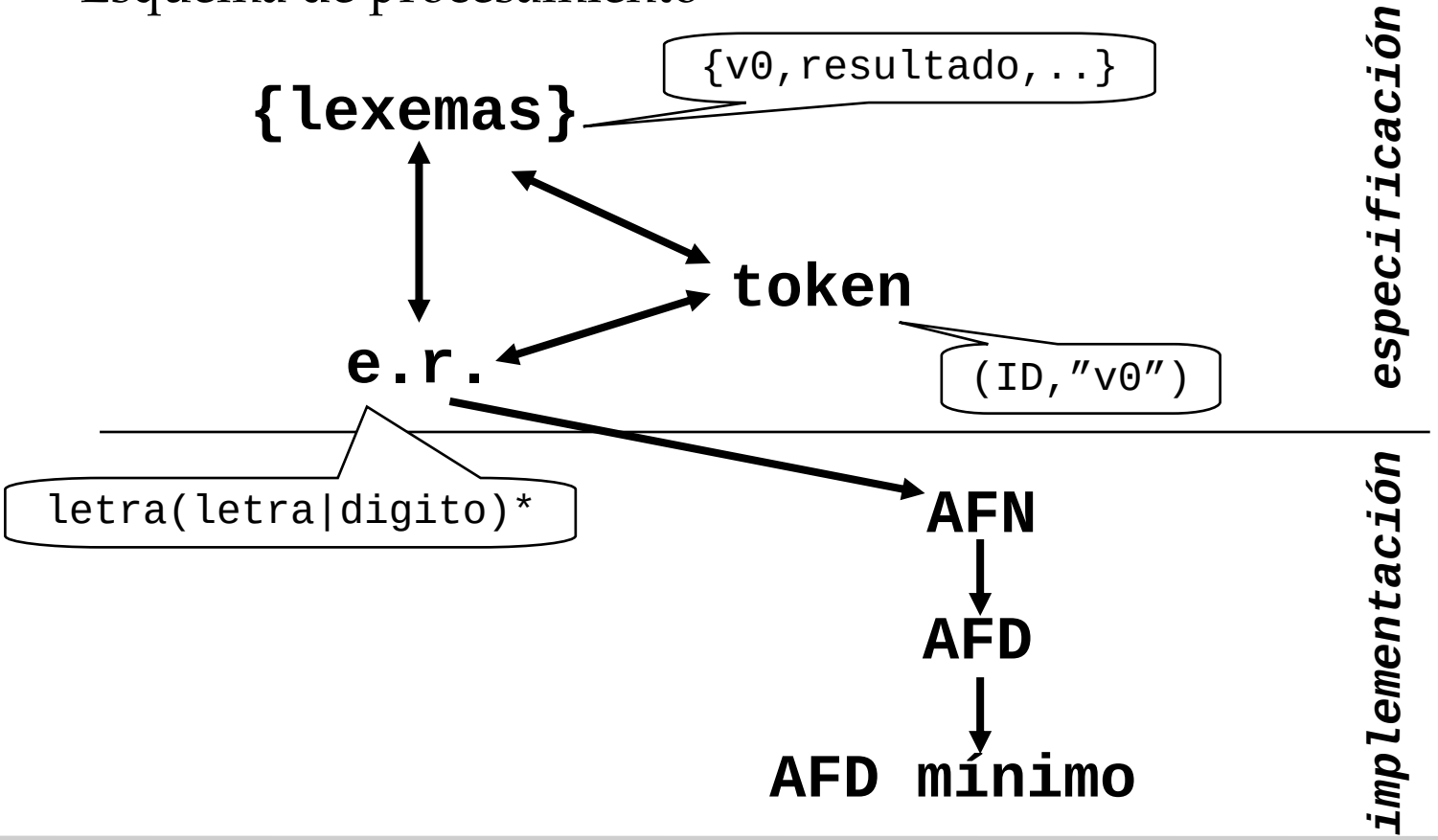
El papel del analizador léxico

- Pero, además
  - procesar directivas al compilador
  - introducir información preliminar en la tabla de símbolos (se verá)
  - eliminar separadores innecesarios (cuando no lo ha hecho un preprocesador)
  - sustituir macros
  - formatear y listar el fuente
- A, veces cuando el lenguaje es sintácticamente complejo, dos fases:
  - fase de examen
  - fase de análisis (propiamente dicho)

El papel del analizador léxico

- Los tokens se pasan como valores "simples"
- Algunos tokens requieren algo más que su propia identificación:
  - constantes: su valor
  - identificadores: el string
  - operadores relacionales: su identificación
  - .....
- Por lo tanto, el scanner debe realizar, a veces, una doble función:
  - identificar el token
  - "evaluar" el token

- Esquema de procesamiento



- Algunas definiciones:

token

"nombre" que se da a cada componente léxico

lexema

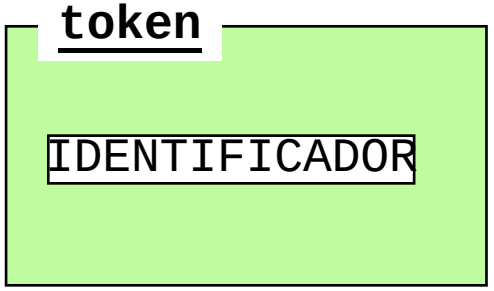
secuencia de caracteres de la entrada que corresponden a un token

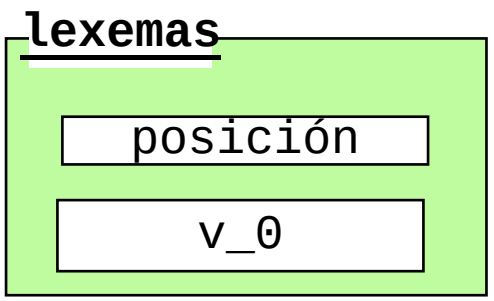
patrón

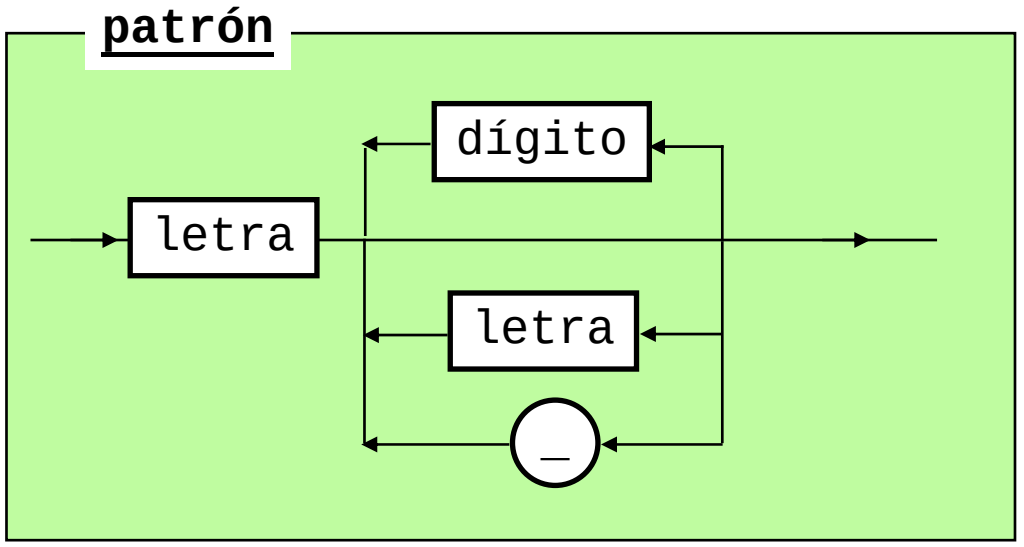
forma compacta de describir conjuntos de lexemas

Tokens, lexemas y patrones léxicos

- Además
  - un token se corresponde con un patrón
  - un token se puede corresponder con muchos lexemas
- Ejemplo: Identificadores





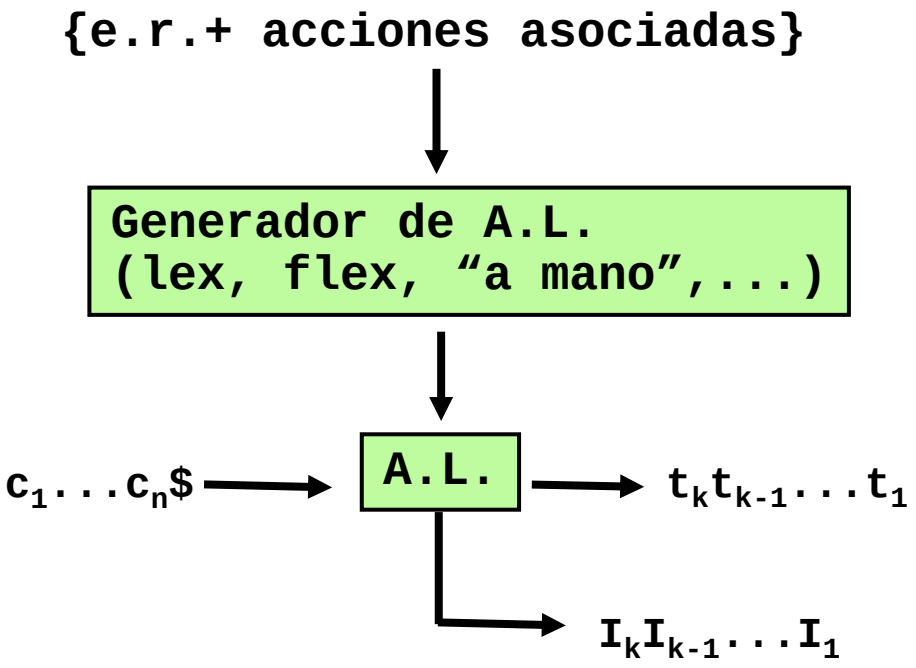


Tokens, lexemas y patrones léxicos

- Tokens más habituales:
  - palabras reservadas
  - identificadores
  - operadores
  - constantes
  - símbolos de puntuación: `;` `,` `.` `:`
  - símbolos especiales: ( ) [ ]
- Pero, a la vez que el propio token, el scanner puede (debe) devolver más información
  - si es un token CONSTANTE, su valor
  - si es un identificador, el string correspondiente
  - si es un símbolo de puntuación, cuál

- Esta información, se devuelve mediante "*atributos*"
- Pero aún puede hacer algo más:
  - puede detectar algunos (pocos) errores léxicos
    - » no hay concordancia con ningún patrón
  - puede llevar a cabo algunas recuperaciones de errores
    - » filtrado de caracters "extraños"
    - » completar algún patrón
    - » reemplazar algún carácter

- Proceso de construcción (con ayuda de herramientas)



$\underline{c_i}$:carácter
$\underline{t_i}$:token
$\underline{I_i}$:información asociada a $\underline{t_i}$

El papel del analizador léxico

- ¿Qué pinta tiene un scanner?
  - un conjunto de funciones
  - una para cada símbolo a reconocer
  - estas funciones son controladas/invocadas por una función que:
    - » selecciona, en función de los caracteres de entrada, qué función invocar
- Opcionalmente, también mediante tablas

- Pero puede ser complicado:
  - necesidad de recorrer varios caracteres antes de decidir el tipo de token
  - "pre-análisis" ó "look-ahead"
  - Ejemplo: reconocido "<" en C, puede corresponder a
    - » `MENOR` "<"
    - » `MENOR_O_IGUAL` "<="
    - » `SHIFT_LEFT` "<<"
  - "`IF`" vs. "`IFNI`"

alfabeto

conjunto finito de símbolos

ejemplo

{0,1}, letras y dígitos,.

cadena

secuencia finita de elementos del alfabeto

ejemplo

0010, estadoInicial, v_0,...

longitud de una cadena

número de elementos del alfabeto que la componen

ejemplos

|hola| = 4

|123456| = 6

ε = 0 (cadena vacía)

lenguaje

dado un alfabeto, cualquier conjunto de cadenas formadas con dicho alfabeto

ejemplo

Siendo Σ={0,1} {0,01,011,0111,01111,...}

- Operadores sobre lenguajes:
  - Sean L,M dos lenguajes

unión de lenguajes

$$L \cup M = \{c \mid c \in L \text{ ó } c \in M\}$$

concatenación de lenguajes

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

cerradura de Kleene

$$L^* = \bigcup_{i=0}^{\infty} L^i$$

CERO o MAS concatenaciones

cerradura positiva

$$L^+ = \bigcup^{\infty} L^i$$

UNA o MAS concatenaciones

Expresiones regulares. Definiciones

- Para un token dado, los lexemas correspondientes los expresaremos mediante expresiones regulares
- **Expresión regular**: forma compacta para definir un lenguaje regular
  - también llamado conjunto regular

lenguaje regular

El generado a partir de una expresión regular

- Una expresión regular r:
  - será definida a partir del lenguaje $L(r)$ que genera
  - operadores sobre expresiones regulares mediante operadores sobre lenguajes

- Sea Σ un alfabeto

expresión regular

1) ε es la expresión regular cuyo lenguaje es L(ε)={ε}

2) Si $a \in \Sigma$, a es la expresión regular cuyo lenguaje es L(a)={a}

3)Sean r,s exp. reg. con lenguajes L(r) y L(s)

a <> **a**

(r) es la exp. reg. cuyo lenguaje es L(r)

r|s es la exp. reg. cuyo lenguaje es L(r) $\cup$ L(s)

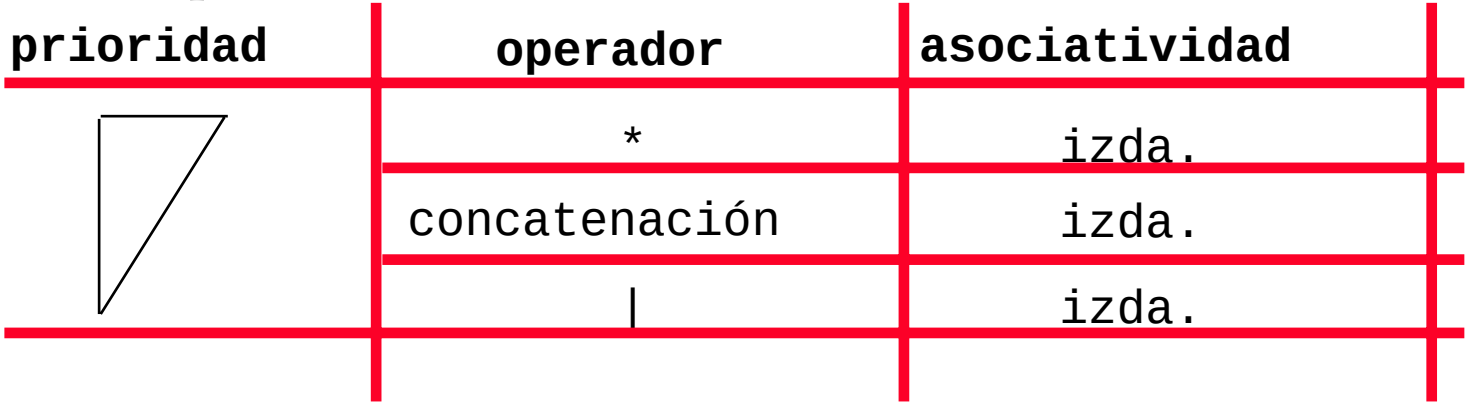
rs es la exp. reg. cuyo lenguaje es L(r)L(s)

r* es la exp. reg. cuyo lenguaje es (L(r))*

- Importante:

  3.4) da la posibilidad de uso de paréntesis para establecer prioridades

- Util, usar prioridades:



prioridad	operador	asociatividad
	*	izda.
	concatenación	izda.
	|	izda.

expresiones regulares equivalentes

Generan el mismo lenguaje

- Ejemplo 1: Sea Σ={a,b}

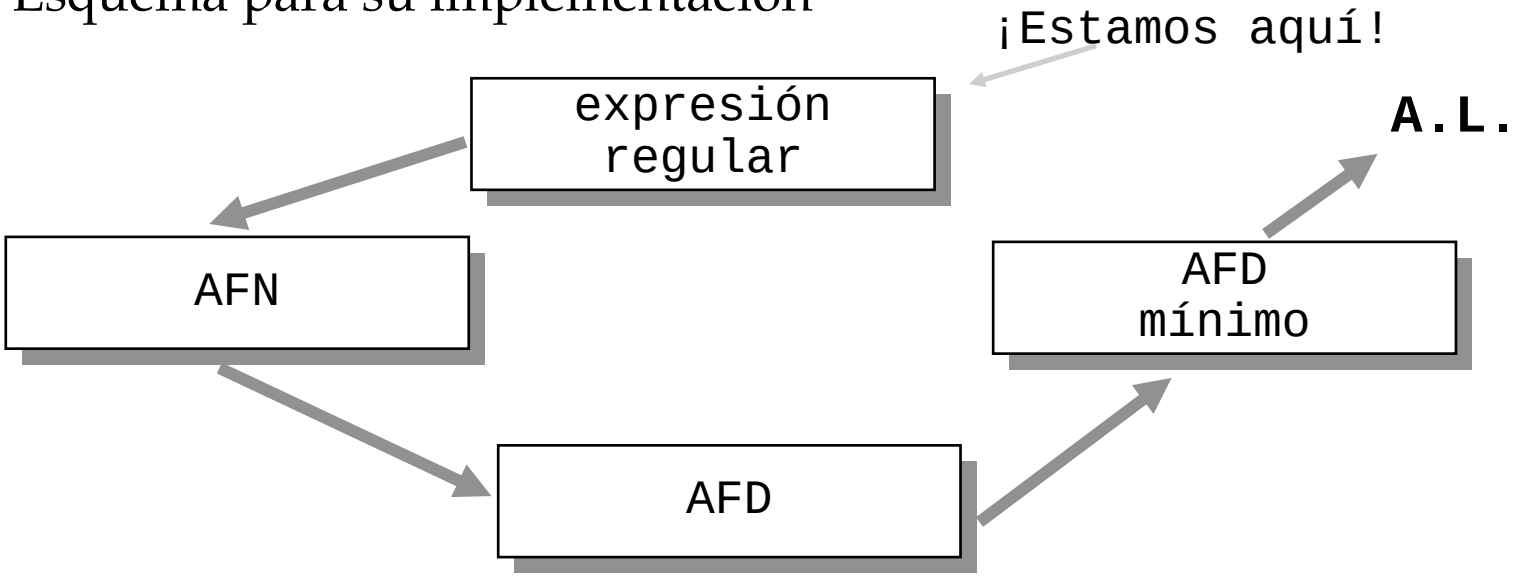
r	*L(r)*
ab	{ab}
a|b	{a,b}
a*	{ε,a,aa,aaa,aaaa,...}
ab*	{a,ab,ab,abbb,...}
(ab|c)*d	{d,abd,cd,abcd,ababcd,...}

Expresiones regulares. Ejemplos

- Ejemplo 2: Sea Σ={0,1}

00 **(00)**	el string '00'
(1|10)*	ε y todos los strings que empiezan con '1' y no tienen dos '0' consecutivos, ε
(0|1)*	todos los strings con 0 ó más '1' ó '0' y ε
(0|1)*00(0|1)*	todos los strings con al menos 2 '0' consecutivos
(0|ε)(1|10)*	todos los strings que no tengan dos '0' consecutivos
(0|1)*011	todos los strings que acaban en '011'

- Ya sabemos expresar los lexemas correspondientes a los tokens
- Necesitamos **implementar** el analizador léxico
- Esquema para su implementación



- Los autómatas finitos pueden ser utilizados para reconocer los lenguajes expresados mediante expresiones regulares
- Un autómata finito (AF) es una máquina abstracta que reconoce strings correspondientes a un conjunto regular
- También se denominan reconocedores
- Misión de un AF:
  - "reconocer" si un string de entrada respeta las reglas determinadas por una expresión regular
- Ejemplo:
  - e.r.: **`(ab|c)*d`**
  - ¿autómata?

Autómata Finito No Determinista

Un AFN es una 5-tupla (S,Σ,δ,s_0,F) donde:

1) S : conjunto de estados

2) Σ : conjunto de símbolos de entrada

3) δ : función de transición

$$\delta : S \times \Sigma \rightarrow P(S)$$

4) $s_0 \in S$: estado inicial

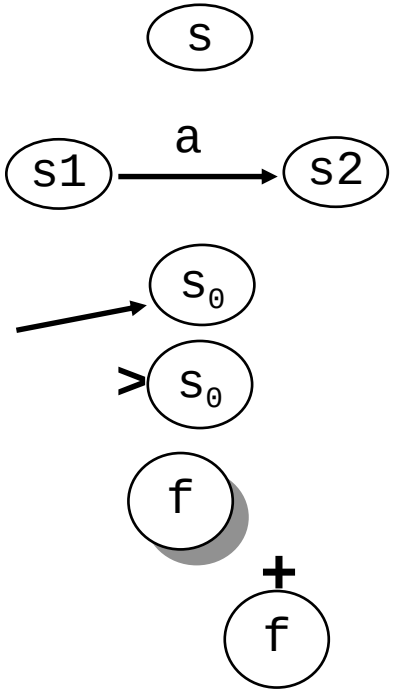
5) $F \subseteq S$: conjunto de estados finales
(o de aceptación)

¡¡ Es posible que $\varepsilon \in \Sigma$!!

¡¡ $\delta : S \times \Sigma \rightarrow P(S)$!!

¡N.D.!

- Notación gráfica:



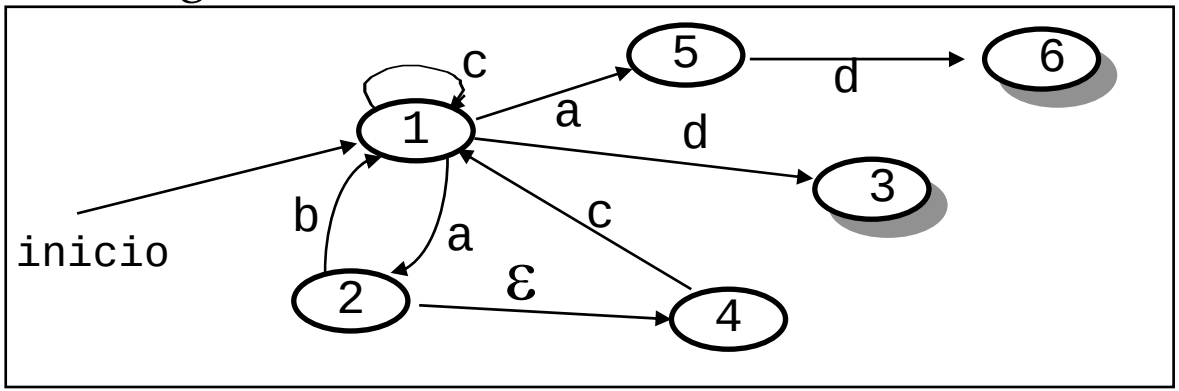
un estado

"transitar" de s1 a s2 cuando se reconozca "a"

s_0 es el estado inicial

f es un estado final (de aceptación)

- AFN como grafo de transiciones



1) S ={1,2,3,4,5,6}
2) Σ ={a,b,c,d}
3) δ(1,c)={1} δ(1,d)={3} δ(1,a)={2,5} δ(2,b)={1}
   δ(2,ε)={4} δ(4,c)={1} δ(5,b)={6}
4) s_0 = 1
5) F = {3,6}

- ¿Cómo funciona un AFN?
- Dado un string, debe determinar si lo acepta o no

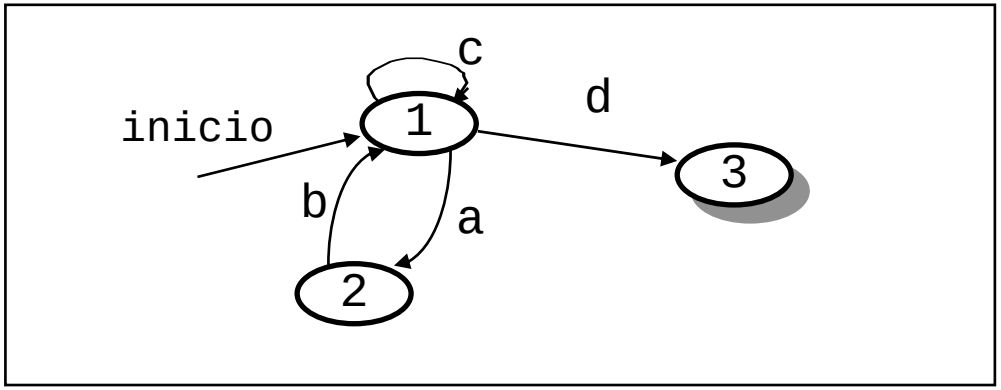
aceptación de un string por un AFN

El string $c_1c_2...c_n$ es aceptado por un AFN cuando existe, en el grafo de transiciones, un camino

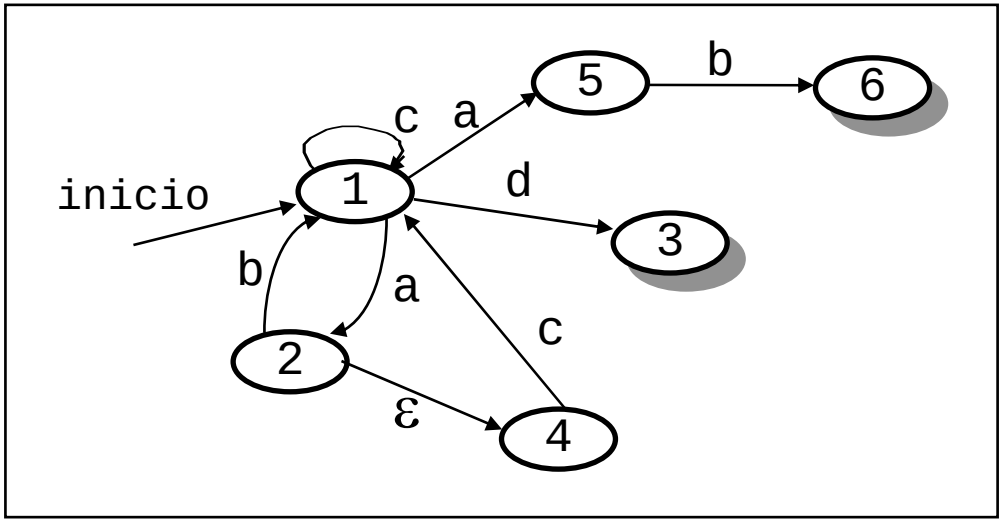
$$s_0 \xrightarrow{c_1} s_1 \xrightarrow{c_2} s_2 \xrightarrow{\varepsilon} \ldots \rightarrow s_{m-1} \xrightarrow{c_n} s_m$$

de manera que s_m es un estado final (de aceptación)

- Ejemplo:
  - ¿Aceptaría el autómata "abcd"? ¿Y "acd"?



- Ejemplo:
  - ¿Aceptaría el autómata "abcd"?
  - El indeterminismo puede generar problemas de eficiencia ("*backtraking*")



- Un Autómata Finito Determinista (AFD) es un caso particular de AFN

Autómata Finito Determinista

Un AFD es un AFN tal que:

1) ε **no etiqueta ningún arco**

2) $\delta : S \times \Sigma \rightarrow S$

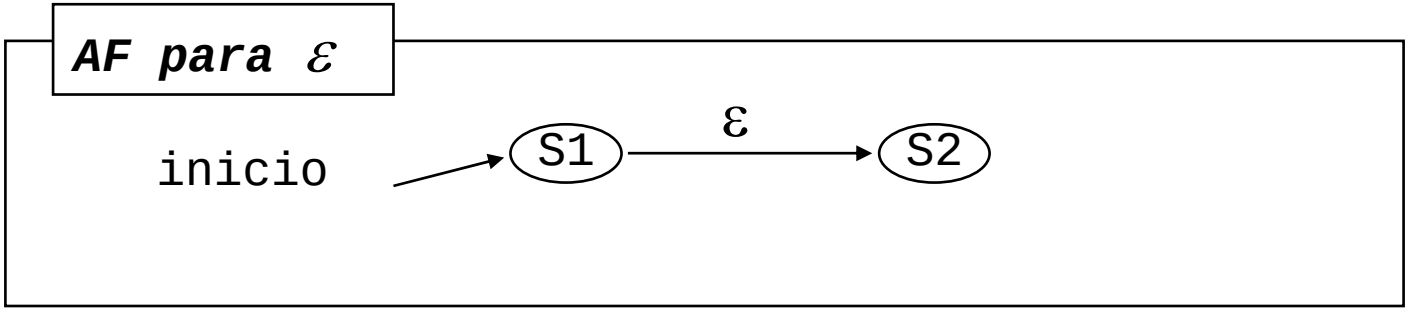
- Es decir:
  - toda transición exige un símbolo distinto de ε
  - desde un estado, no hay dos arcos etiquetados con el mismo símbolo

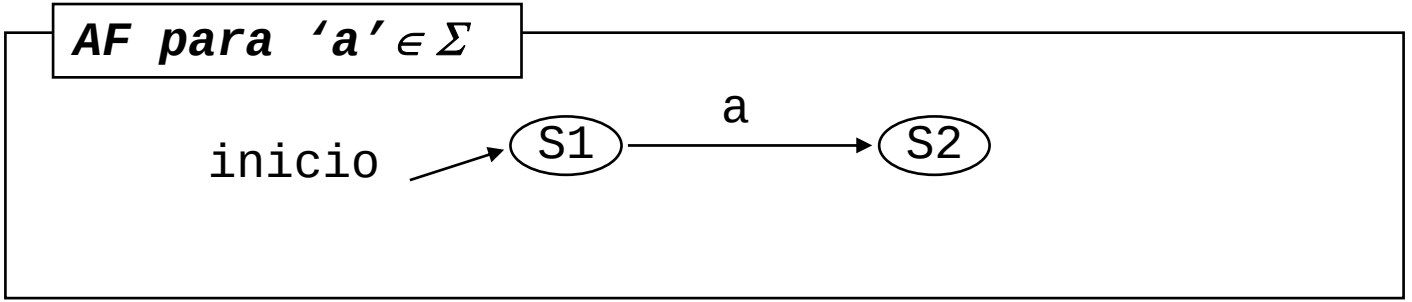
- Simular un AFD es muy sencillo y eficiente
- En lo que sigue:
  - dada una e.r., generar el AFN
  - Convertir el AFN en un AFD y minimizarlo
  - Implementar el AFD

```
Func simulaAFD(x,A) dev (acep:booleano)
/* Pre: x es la cadena a aceptar
        A=(S,Σ,δ,s0,F) es el AFD
   Post:acep = x es aceptado por A
*/
Variables sAct:estado ∪ {ERROR};i:entero
Principio
   <i,sAct>:=<1,s0>
   Mq sAct<>ERROR ∧ i<=length(x)
      sAct:=mueve(A,sAct,x[i]) /*único*/
      i:=i+1
   FMq
   acep:=(sAct∈F)
Fin
```

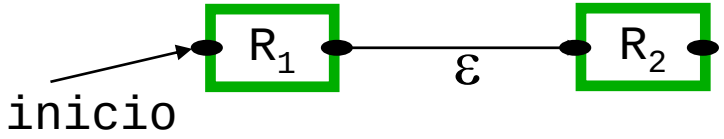
¿Complejidad?

- Objetivo: dada una expresión regular, generar un AFD que la reconozca
- Método: construcción de Thompson (Bell Labs.)

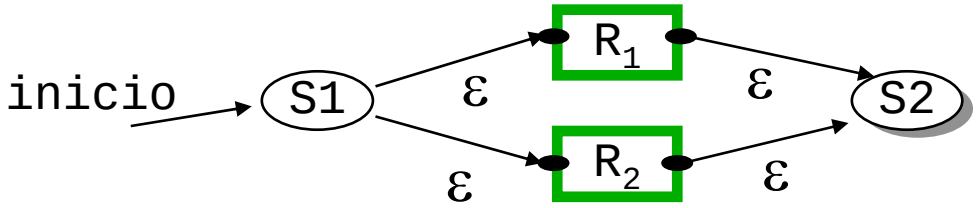


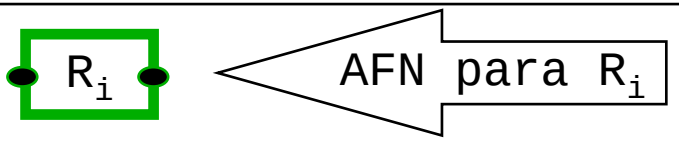


AF para la expresión regular R_1R_2

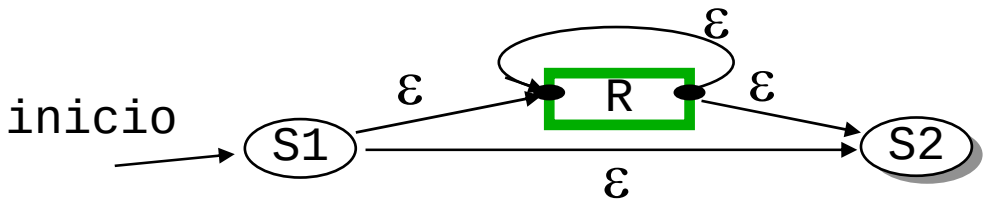


AF para la expresión regular $R_1|R_2$

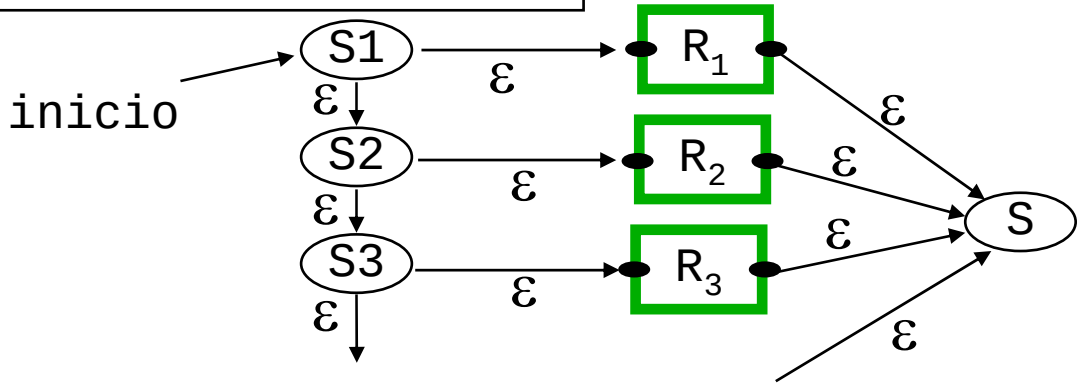




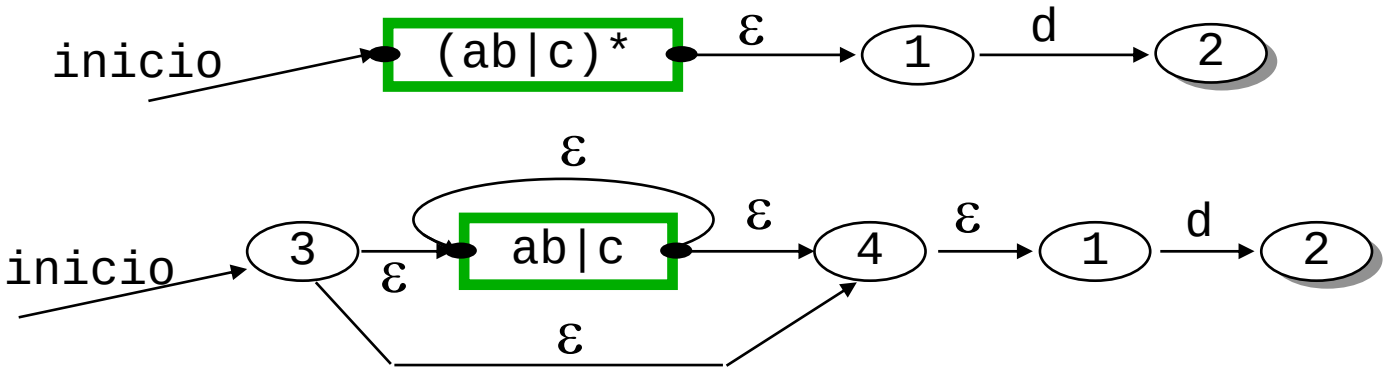
AF para la expresión regular R*

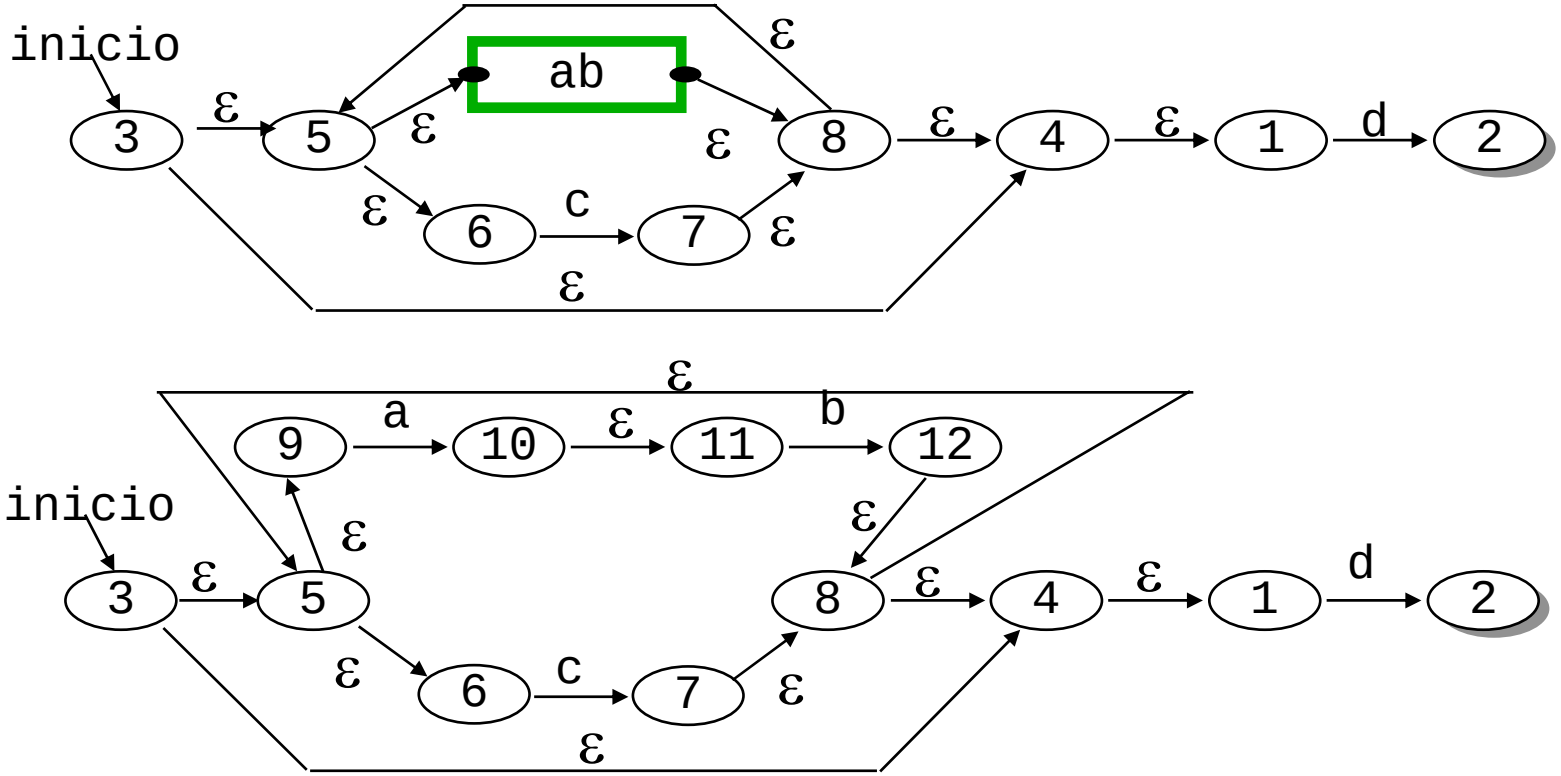


AF para un conjunto de ER



- Ejemplo: proceso de construcción para
  `(ab|c)*d`

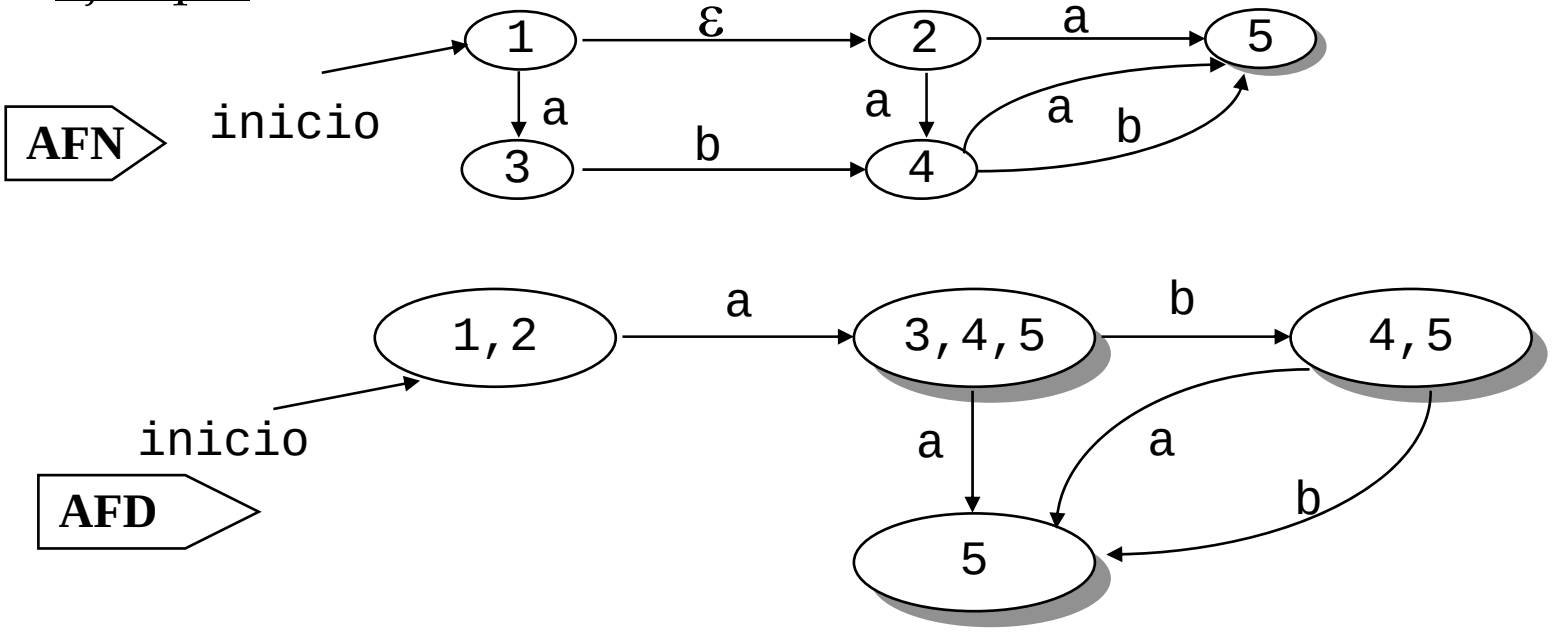



inicio
3
ε
5
ε
ab
ε
ε
8
ε
4
ε
1
d
2
ε
6
c
7
ε
ε
ε
9
a
10
ε
11
b
12
inicio
3
ε
5
ε
ε
8
ε
4
ε
1
d
2
ε
6
c
7
ε
ε
ε

Transformación de un AFN en un AFD

- Generar un AFN a partir de una e.r. es sencillo
- Implementar un AFD es más sencillo y eficiente que implementar un AFN
- Por lo tanto, es interesante saber generar, a partir de un AFN, un AFD equivalente
- El método se basa en la idea de ε-clausura (ver [HoUl 79])

- La idea básica es que un estado del AFD agrupa un conjunto de estados del AFN
- Ejemplo:



- Sea A=$(\mathbf{S}, \Sigma, \delta, \mathbf{s_0}, \mathbf{F})$ un AFN

ε-clausura de $s \in S$

Conjunto de estados de N alcanzables desde s usando transiciones ε

ε-clausura de $T \subseteq S$

$$\bigcup_{s \in T} \textbf{clausura}\varepsilon(s)$$

mueve(T,c)

Conjunto de estados a los que se puede llegar desde algún estado de de T mediante la transición c

- Algoritmo para simular un AFN

```
Func simulaAFN(x,A) dev (acep:booleano)
/* Pre: x es la cadena a aceptar
        A=(S,Σ,δ,s0,F) es el AFN
   Post:acep = x es aceptado por A
*/
Variables T:conjunto(estados ∪ {ERROR})
          i:entero
Principio
   <i,T>:=<1,clausuraε({s0})>
   Mq T<>{ERROR} ∧ i<=length(x)
      T:=clausuraε(mueve(A,T,x[i]))
      i:=i+1
   FMq
   acep:=(T∩F<>∅)
Fin
```

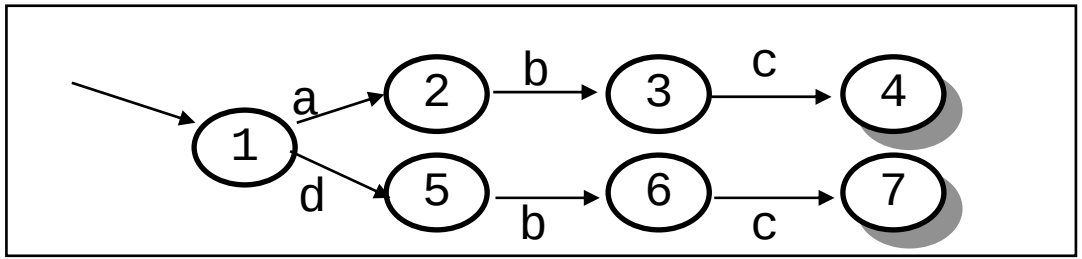
- Ejercicio:
  - Escribir el algoritmo "clausuraε"

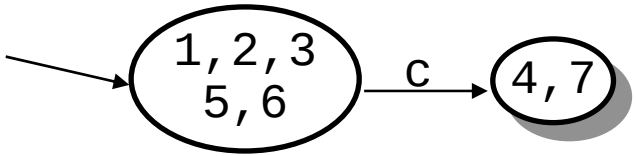
```
Func clausuraε(T,A) dev (c:conjEstados)
/* Pre: A es un AFN
        T es un conjunto de nodos de A

   Post:c= ε-clausura de T en A
*/
```

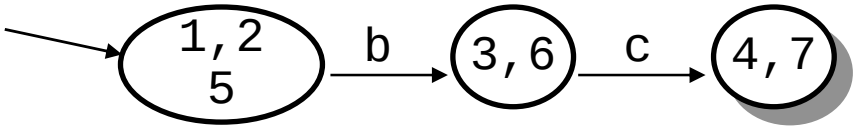
Minimización de AFD

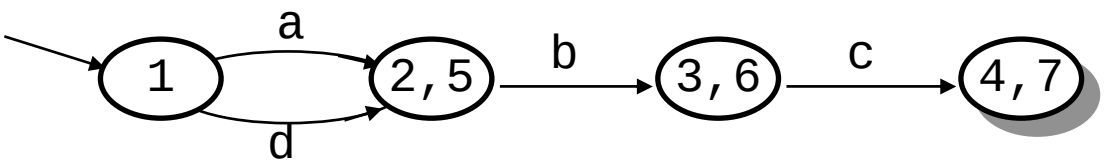
- Algunas cuestiones:
  - Como es lógico, cuantos más estados tiene un AF, más memoria es necesaria
  - El número de estados del AFD se puede/debe minimizar (ver [HoUl 79])
    - » inicialmente, dos estados:
      - uno con los de aceptación
      - otro con los demás
    - » sucesivas particiones de estados "globales" que no concuerdan con algún sucesor para un símbolo de entrada
  - El AFD mínimo equivalente es único

- Ejemplo:









Transformación de un AFN en un AFD

– Complejidad calculada:

	espacio	tiempo
AFN	$O(\|r\|)$	$O(\|r\| x \|x\|)$
AFD	$O(2^{\|r\|})$	$O(\|x\|)$

- r: expresión regular
  x: string a reconocer

– A destacar:

» AFN son mejores en espacio

- tener en cuenta que en su AFD, cada estado del AFN se puede "almacenar" varias veces

» AFD son mejores en cuanto a velocidad de reconocimiento

3ª conv. curso 95/96

Ejercicio 1 (3 ptos.): Los identificadores para un determinado lenguaje de programación se definen de acuerdo con la siguiente descripción:

Un identificador puede empezar con una letra o un "underscore" (carácter "_"), debe estar seguido por 0 ó más letras, dígitos o underscores, pero con las restricciones siguientes:

1) No pueden aparecer dos underscores seguidos

2) Un identificador no puede terminar con un underscore.

Además, no hay ninguna limitación en cuanto a la longitud de los identificadores.

1.1) (1.5 ptos.) Dibujar el Autómata Finito Determinista que corresponde a los identificadores descritos anteriormente.
Para etiquetar los arcos, en lugar de utilizar caracteres simples utilizar las siguientes clases de caracteres:

```
letra  [a-zA-Z]   digito  [0-9]   und  "_"
```

1.2) (1.5 ptos.) Dar una expresión regular correspondiente a los identificadores descritos anteriormente

Ejercicio 3 (0.75 ptos.): El libro *"Pascal: User Manual and Report"* de K. Jensen y N. Wirth, que establece la especificación ISO Pascal Standard, define un comentario del lenguaje como (sólo vamos a considerar comentarios válidos aquéllos que empiezan por "(*" y terminan por "*)"):

"(" seguido de cualquier secuencia de 0 ó más caracteres que no contenga "*)", y terminado por "*)"*

Escribir una expresión regular con sintaxis LEX para los comentarios Pascal así definidos.

Ejercicio: Dar una expresión regular para los strings de Ada

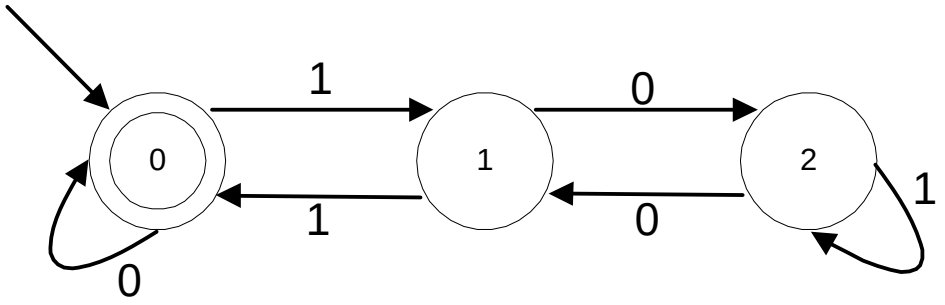
Ejercicio 3 (V2): El manual Pascal ISO 7185:1990, define un comentario Pascal como sigue:

("(“ | “{“) commentary (“*)” | “}”)*

where commentary is any sequence of characters and separations of lines containing neither *“*)”* nor *“}”* and *"(*“, “{“* do not occur inside commentary

Dar una expresión regular para esta versión de los comentarios

Ejercicio 4: Considerar el siguiente Autómata Finito Determinista. Dar una expresión regular que corresponda a dicho autómata.



Ejercicio 5: Sea Σ un alfabeto, y sea **r** una expresión regular sobre Σ. Vamos a denotar L(**r**) el lenguaje generado por la expresión **r**. Por otro lado, siendo **r** y **s** dos expresiones regulares,

* **rs** representa el lenguaje L(**rs**)={ vw | v ∈ L(r) y w ∈ L(s)}, formado por las concatenaciones de una cadena de **r** y una de **s**
* **r+s** representa el lenguaje formado las cadenas de L(**r**) ∪ L(**s**)
* siendo **n** un número natural, $\mathbf{r^n}$ representa el lenguaje generado la concatenación de **n** cadenas del lenguaje generado por **r**
* siendo **n** un número natural, **nr** representa el lenguaje generado la $(r+r+..._{\text{n veces}}...+r)$

El ejercicio pide razonar sobre la corrección o incorrección de las siguientes afirmaciones, donde **r, s, t** son expresiones regulares:

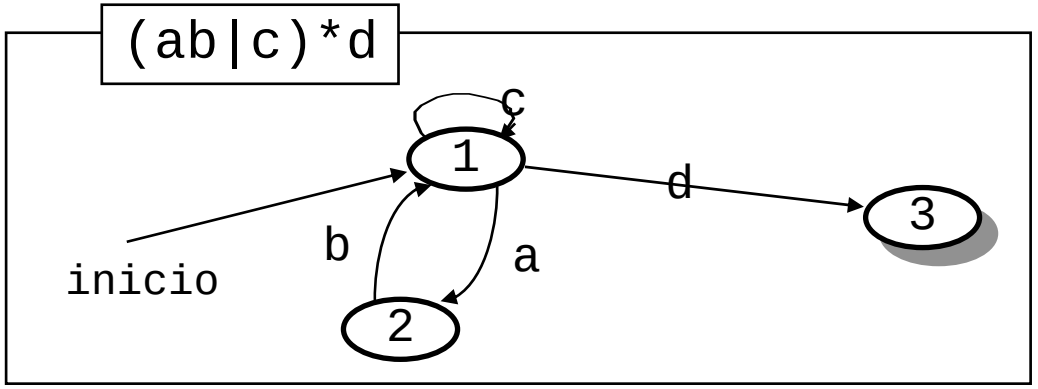
* **r(s+t) = rs+rt**
* $\varepsilon \in \mathbf{rs}$ si, y sólo si, $\varepsilon \in \mathbf{r+s}$
* $\mathbf{(r+s)^2 = r^2 + s^2 + 2rs}$

- Un AFN se suele implementar:
  - mediante una

    `tabla de transiciones`

    - » El grafo aparece explícitamente
    - » Entrada: `(estado, símbolo)`
    - » Salida: nuevo estado
  - "a pelo"
    - » el grafo de transiciones no está explícitamente implementado

- Ejemplo:



símbolo entrada

estado	a	b	c	d	
1	2	ERR	1	3	
2	ERR	1	ERR	ERR	
3	acep	acep	acep	acep	

Un analizador léxico muy simple

- Propuesta:

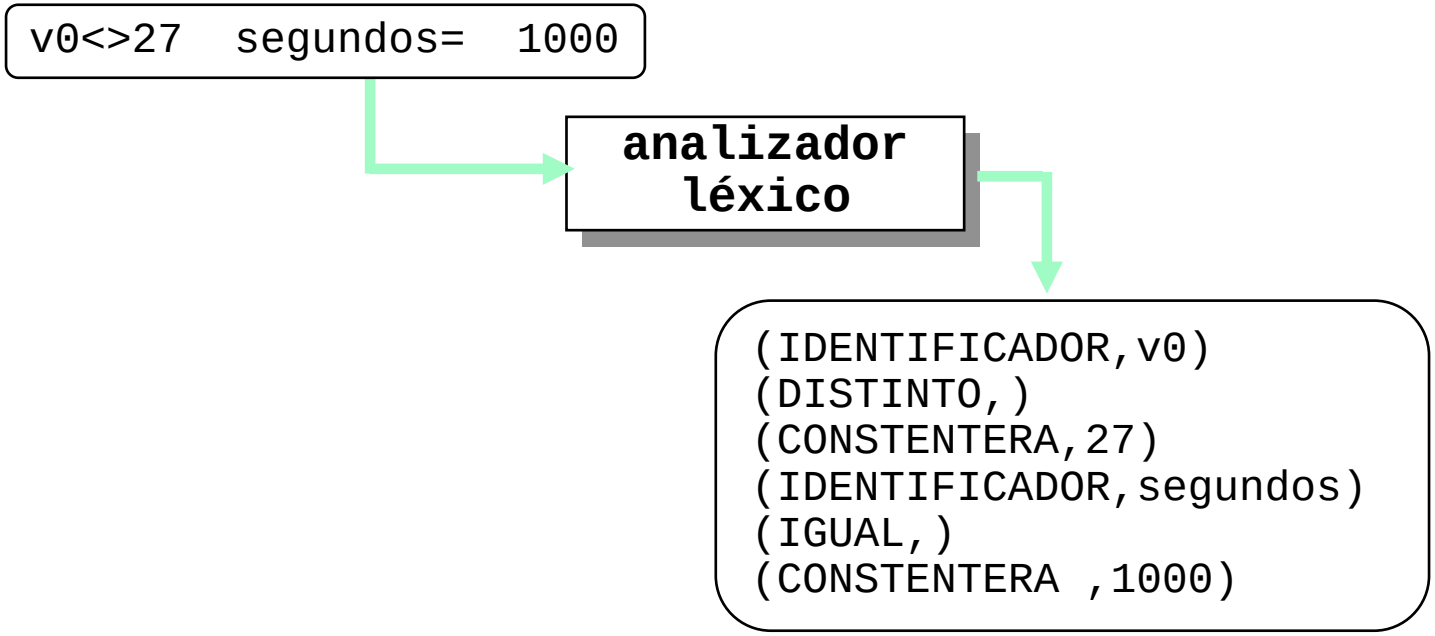
  Un analizador léxico para un evaluador de expresiones

- Involucra:
  - constantes enteras sin signo
  - operadores relacionales
    <,<=,>,>=,<>,=
  - identificadores de variables

- Componentes léxicos:

```
menor → <
mayor → >
menorIgual → <=
mayorIgual → >=
igual → =
distinto → <>
letra → a|b|...|z|A|B|....|Z
digito → 0|1|...|9
identificador →
    letra (letra |digito)*
constEntera →
    digito digito*
```

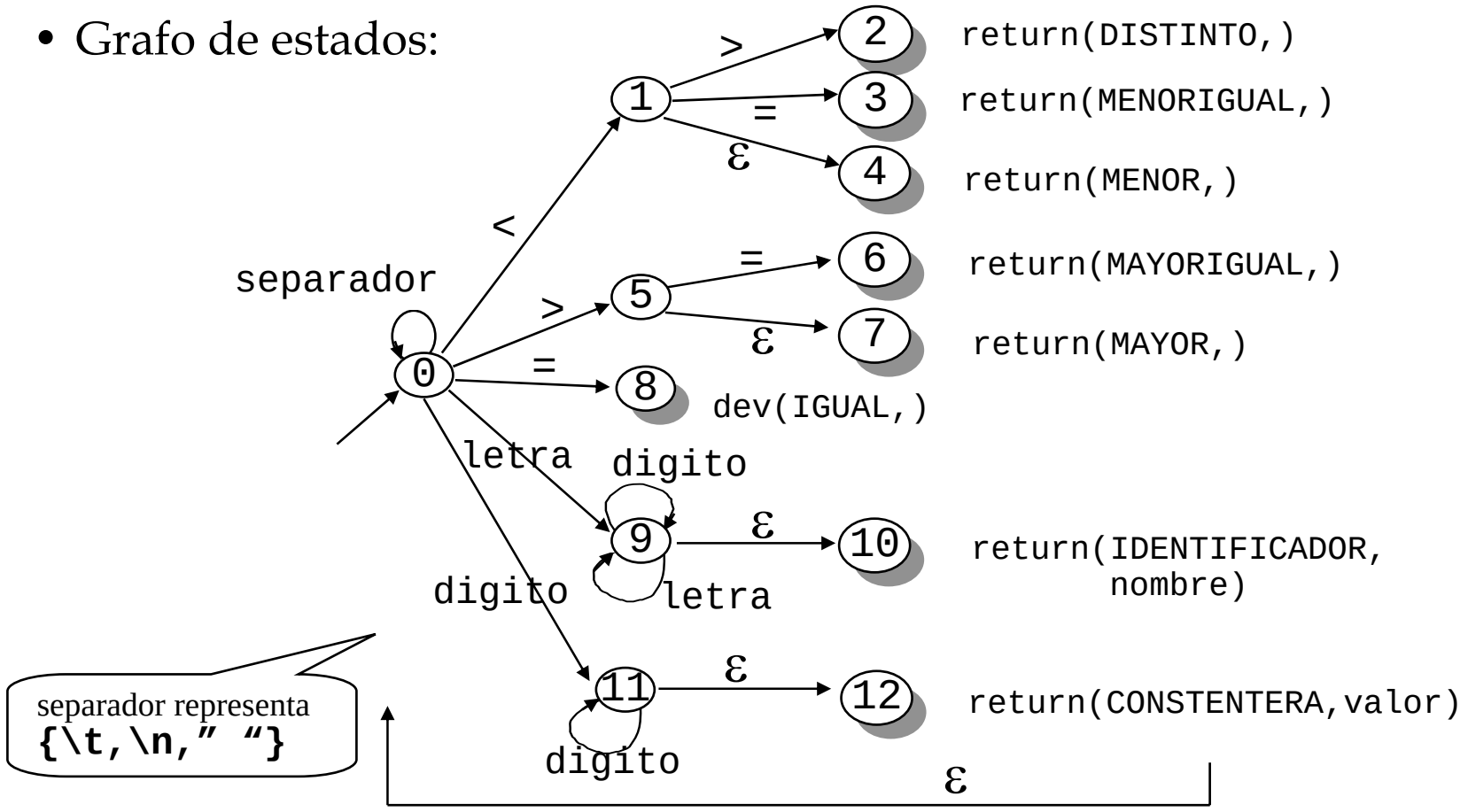
- Ejemplo de uso:



```
/*-----------------------------------------------
PRE:  el primer carácter en stdin corresponde al
      primer carácter del siguiente lexema.
       Cualquier valor para 'yylval', 'yytext',
      'yyleng'
POST: se ha procesado un lexema. El siguiente
      carácter en stdin corresponde al primer
      carácter de un nuevo lexema.
       * Devuelve el token correspondiente al
         lexema tratado
       * asignado a la variable global 'yylval'
         el valor adecuado (si necesario)
       * la var. global 'yytext' contiene el
         lexema tratado
       * la var. global 'yyleng' contiene la
         long. del lexema
-----------------------------------------------*/
token yylex()
{ ........ }

void reconocido(token elToken)
{ ........ }
```

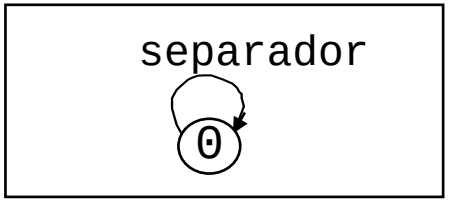
```
void main()
{  token elToken;

   do{
      elToken=yylex();
      reconocido(elToken);
   }while(elToken!=FIN);
}
```

- Grafo de estados:



- Consideraciones para su implementación

  1) Saltaremos separadores: `' '  '\t'  '\n'`



  2) Sobre la entrada:
    - » carácter a carácter: `buffer de entrada de tamaño 1`
    - » entrada de `stdin`
    - » termina con `'$'`
      - no corresponde a ningún lexema
      - devuelve un token "ficticio" `'FIN'`

3) Usaremos algunas variables globales

» ¿Por qué globales?

yytext

string con el lexema reconocido

yylval

entero con información necesaria
(valor de las ctes., p.e.)

yyleng

longitud del lexema

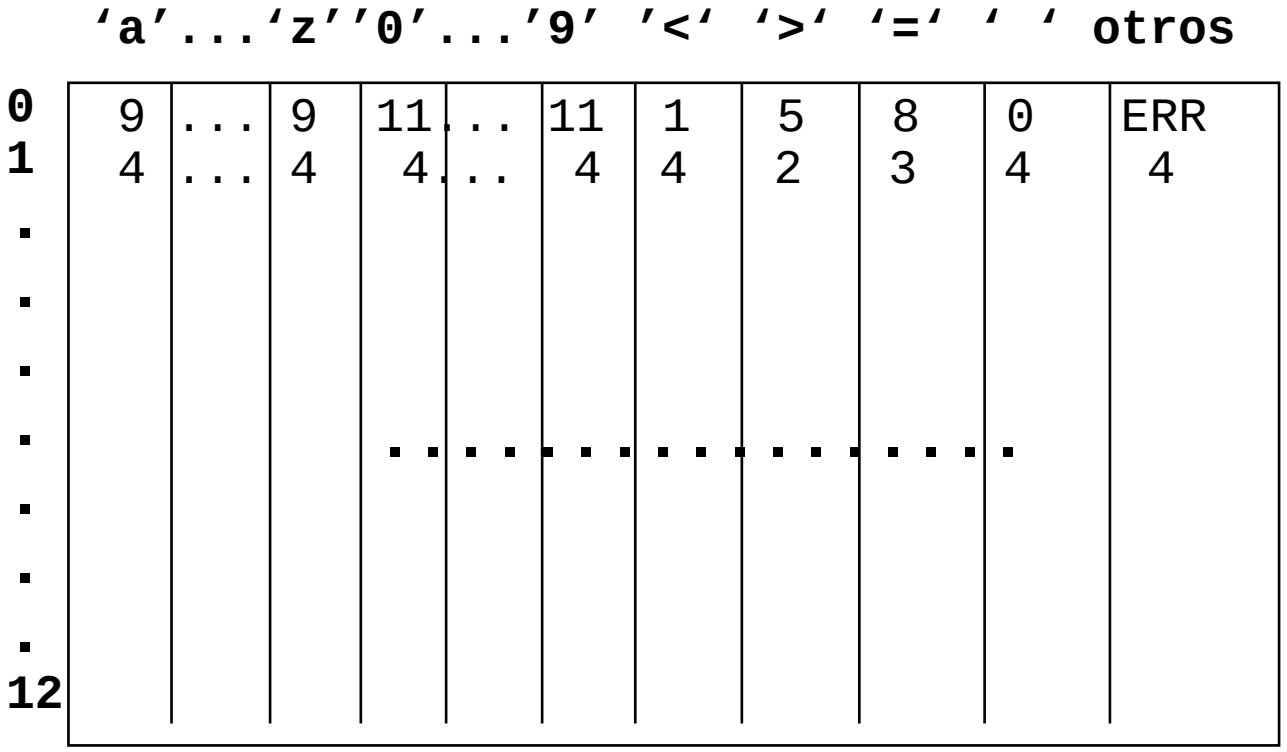
Implementación con TT

```
#include <stdio.h>
#include <ctype.h> /*isdigit(),isalpha()...*/

enum{ FIN=0,
      MENOR=256,
      MENORIGUAL,
      MAYOR,
      MAYORIGUAL,
      IGUAL,
      DISTINTO,
      IDENTIFICADOR,
      CONSTENTERA,

      NUMESTADOS=13,
      NUMCARS=41,
 /* 'a'...'z''0'...'9' '<' '>' '=' ' ' OTROS */
      ERROR=-1
};
#define carFin          '$'   /*fin de ent.*/

#define YYLMAX                 100
#define PONCAR(pC,C,L)
             {sprintf(pC++,"%c",C);L++;}
```

```
typedef int token;
char  sigCar,
      *delante,
      yytext[YYLMAX];        /*el lexema*/
int   yyleng,           /*long. del lexema*/
      yylval;            /*para atributos*/
int   estado;                /*el estado*/
```

hab., serán 128

- Las transi-ciones



- Parte del autó-mata

```
int T[NUMESTADOS][NUMCARS]={
{9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,
 9,9,9,9,9,9,11,11,11,11,11,11,11,11,11,
 11,1,5,8,0,ERROR},

     ........

}
```

```
int aceptacion[NUMESTADOS]=
      {0,0,1,1,1,0,1,1,1,0,1,0,1};
```

```
token yylex()
{  int sigEstado, seguir=1; /* posible nuevo estado */
   yyleng=0;

   yytext[0]='\0';      /* de momento, lexema vac'io */
   delante=yytext;
   estado=0;
   sigCar=getchar();
   if(sigCar==carFin) return FIN;
   do{
      sigEstado=T[estado][pos(sigCar)];
      if(sigEstado == ERROR)
         seguir=0;
      else{
         estado=sigEstado;
         if(!isspace(sigCar))
            PONCAR(delante, sigCar, yyleng);
         if(aceptacion[estado])
               seguir=0;
         else{ sigCar=getchar();
               if(sigCar==carFin)
                  seguir=0;
         }
      }
   }while(seguir);
   ...
}
```

gen., no necesario

saltar separadores

```
   ...
   if(aceptacion[estado])
      return tokenAceptacion();
   else if(sigCar==carFin)
      return FIN;
   else
      errorLexico();
}
```

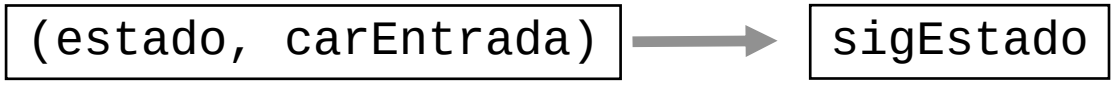
```
/*-----------------------------------------------
PRE:  global 'estado' con un estado de aceptación
      'yyleng','yytext' como siempre
POST: tokenAceptacion=token correspondiente.
      (lo da el estado)
      Además, ejecuta las acciones asociadas
      a los estados de aceptación
-----------------------------------------------*/
token tokenAceptacion(){
   switch(estado){
      case 2:  return DISTINTO;
      case 3:  return MENORIGUAL;
      case 4:  yytext[yyleng-1]='\0';
               yyleng--;
               ungetc(sigCar, stdin);
               return MENOR;
      case 6:  return MAYORIGUAL;
      case 7:  yytext[yyleng-1]='\0';
               yyleng--;
               ungetc(sigCar, stdin);
               return MAYOR;
      case 8:  return IGUAL;
      ...
   }
}
```

elimina ult. car. de yytext

```
case 10: yytext[yyleng-1]='\0';
    yyleng--;
    ungetc(sigCar, stdin);
    return IDENTIFICADOR;
case 12: yytext[yyleng-1]='\0';
    yyleng--;
    ungetc(sigCar, stdin);
    sscanf(yytext, "%d", &yylval);
    return CONSTENTERA;
```

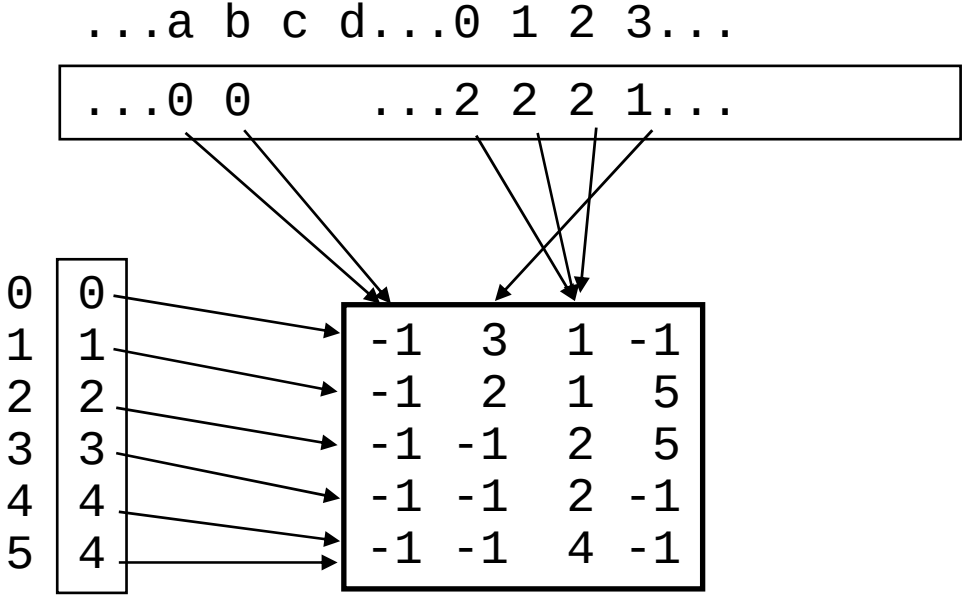
valor entero

- El objetivo básico de una implementación de un AF es dar la aplicación

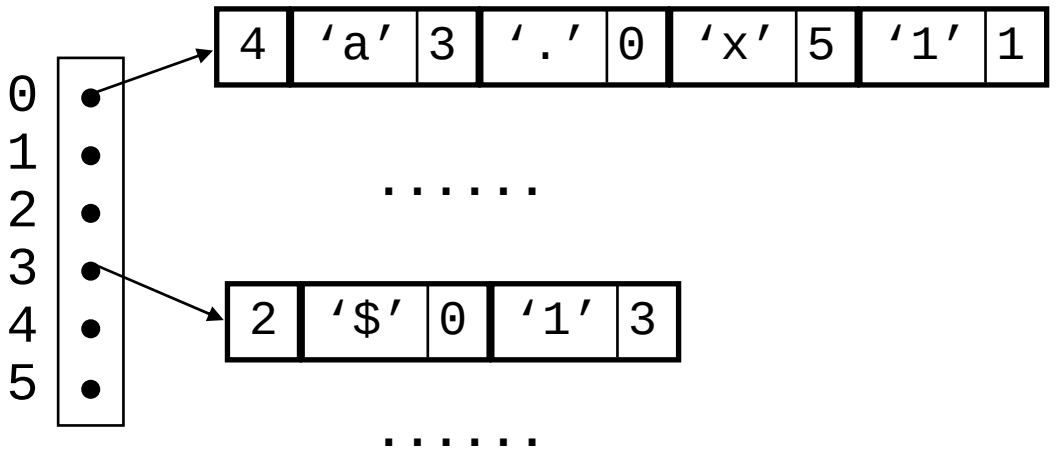


- La "más natural":
  - tabla como array
    - » una fila por cada estado
    - » una columna por cada carácter posible de entrada
  - en general, 128 columnas
- Ventajas:
  - fácil de programar (trivial)
  - acceso rápido
- Inconvenientes:
  - "despilfarro" de memoria

- Para evitar el problema de la memoria
  - técnicas de compactación de tablas
- **METODO 1**: basado en que muchas columnas/filas suelen ser idénticas
- Ejemplo: asumamos 6 estados $(S_0,...,S_5)$



- **METODO 2**: especialmente útil cuando la matriz de transición es muy dispersa
- Ejemplo: asumamos 6 estados **$(S_0,...,S_5)$**



Sobre recuperación de errores léxicos

- Cuando un scanner detecta un error léxico
  - NO abortar la compilación + RECUPERAR
- Tres aproximaciones:
  - ignorar los caracteres leídos hasta la detección del error
  - borrar el primer carácter leído y recomenzar el análisis
  - añadir algún carácter (si está claro)
- Errores más comunes:
  - debidos a un carácter extraño
  - suelen aparecer al principio del lexema
  - desbordamiento de variables/constantes
  - fin de línea antes de cerrar un string
  - problemas cerrar/abrir comentarios
- Opcionalmente: pasar al analizador sintáctico un token de ERROR, junto con el string que lo forma