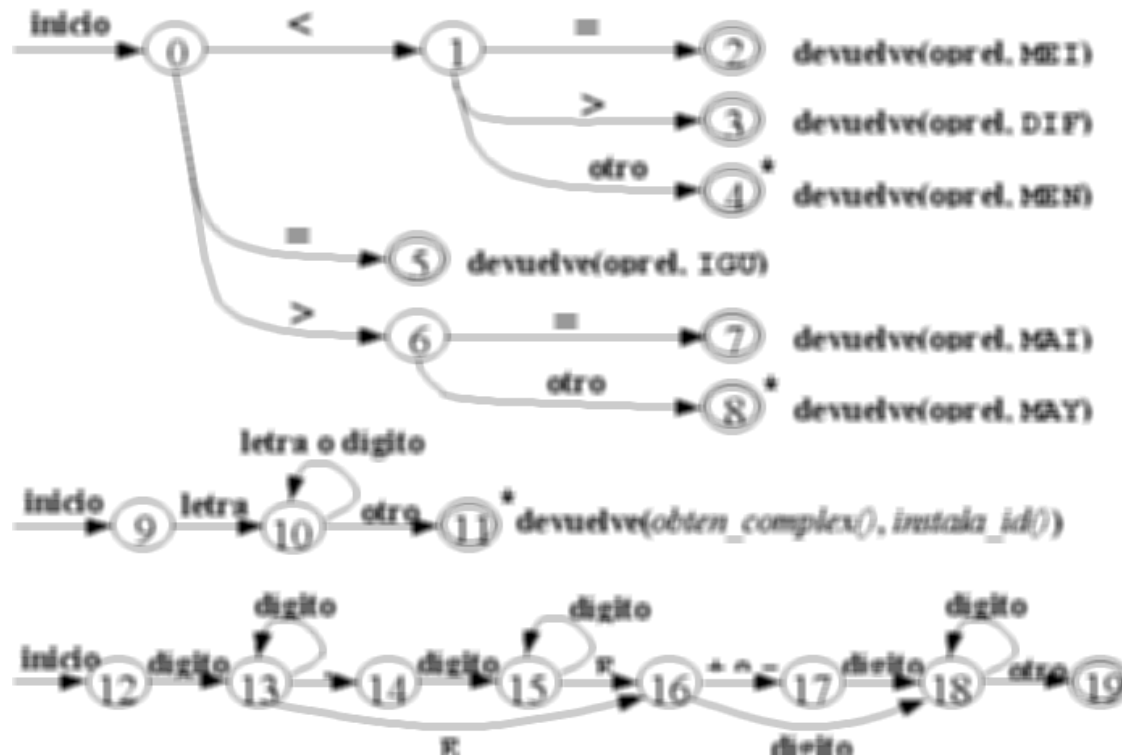


Análisis Léxico



Esquema

- Introducción
 - Conceptos básicos
 - Funciones del analizador léxico
 - Detección de errores
 - Esquema general de un analizador léxico
- Reconocimiento y de componentes léxicos
 - Reconocimiento de componentes léxicos
 - Expresiones y definiciones regulares
- Autómatas finitos
 - Definición y simulación
 - Obtención de AF a partir de ER
 - Transformación de un AFND en un AF
- Cuestiones de implementación
 - Manejo de buffers de entrada
 - Diseño de un generador de analizadores léxico
 - Compresión de tablas
 - Eficiencia en la simulación de AF

A decorative vertical bar on the left side of the slide, featuring a black background with glowing green binary code (0s and 1s) arranged in a grid-like pattern.

Introducción

Función del analizador léxico

(1)

- **Componente léxico (*token*)**: unidad mínima de información que significa algo a la hora de compilar; tiene una categoría léxica; las frases de un lenguaje son cadenas de componentes léxicos («palabras»).
- **Lexema**: Secuencia de caracteres de entrada que comprende un solo componente léxico; cadenas de caracteres de las que extraigo el concepto abstracto de componente léxico.
- **Patrón**: Descripción de la forma que han de tomar los lexemas para ajustarse a un componente léxico.
- **Atributos**: La información adicional sobre los *tókenes* proporcionada por el analizador léxico (lexema, valor numérico, puntero a TS).

Ejemplo:

COMP. LÉXICO	LÉXEMA	PATRÓN
const	const	const
if	if	if
oprel	>, <=, =, >, >=	< <= = > >=
id	pi, D2, velocidad	[a-zA-Z][a-zA-Z0-9]*
núm	3.141, 6.2E23, 120	-(?([0-9]+ [0-9]*\.[0-9]+)([eE][+-]?[0-9]+)?
literal	"hola mundo"	\["^\"]*\\"

- Un analizador léxico aísla al analizador sintáctico de la representación en lexemas de los componentes léxicos.

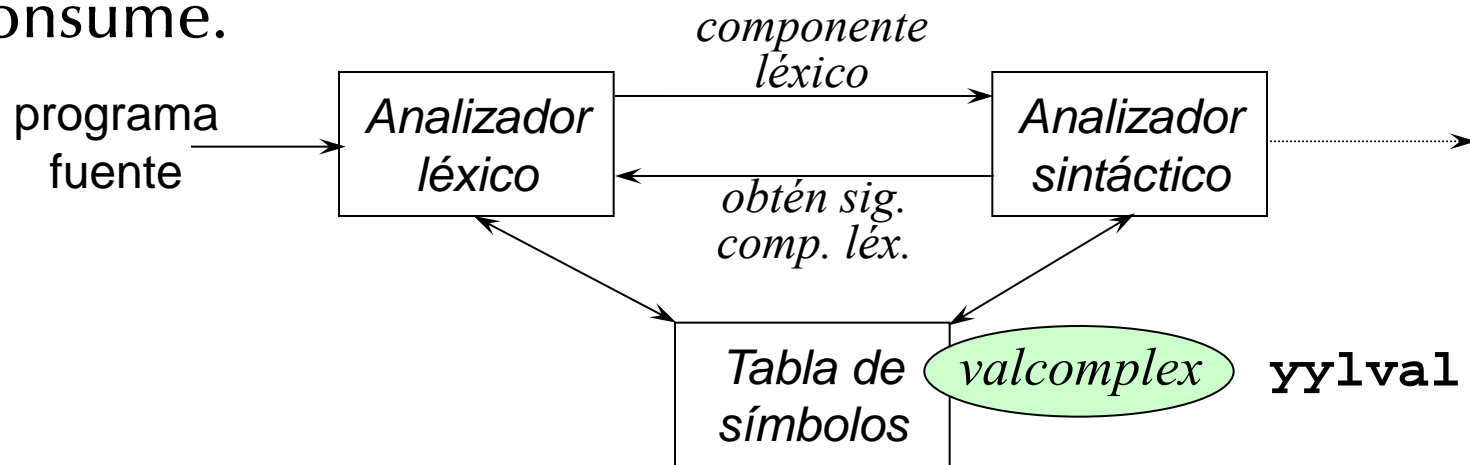
Tiene como funciones:

- **Eliminación de espacios en blanco y comentarios.** Si el analizador léxico elimina los espacios en blanco, el analizador sintáctico nunca tendrá que considerarlos. La alternativa de modificar la gramática para incorporar los espacios en blanco dentro de la situación no es tan fácil de implantar.
- **Reconocimiento de identificadores y palabras clave.** Es el encargado de construir los lexemas que constituyen los identificadores de los lenguajes de programación.
- **Reconocimiento de constantes.** La tarea de agrupar dígitos para formar enteros se le asigna, por lo general, a un analizador léxico, porque los números se pueden tratar como unidades simples durante la traducción.
- **Manejo del fichero del programa fuente.** Abrirlo leer sus caracteres y cerrarlo, gestionando posibles errores de lectura.

Función del analizador léxico

(3)

- Interfaz con el analizador sintáctico
- Es habitual que el analizador léxico y el sintáctico formen un par *productor-consumidor*. El analizador léxico produce componentes léxicos y el analizador sintáctico los consume.



- Hay varias razones para dividir la fase de análisis de la compilación en análisis léxico y análisis sintáctico.
 1. Separar el análisis léxico del análisis sintáctico a menudo permite simplificar una u otra de dichas fases.
 2. Se mejora la eficiencia del compilador.
 3. Se mejora la transportabilidad del compilador.

Atributos de los componentes léxicos

- Cuando más de un lexema concuerda con un patrón, el analizador léxico debe proporcionar a las siguientes fases del compilador información adicional sobre el lexema concreto que concordó.
- El analizador léxico recoge información sobre los componentes léxicos en sus atributos asociados.
- Los componentes léxicos influyen en las decisiones del análisis sintáctico, y los atributos, en la traducción de los componentes léxicos.
- En la práctica, los componentes léxicos suelen tener un solo atributo – un apuntador a la entrada de la tabla de símbolos donde se guarda la información sobre el componente léxico.

Ejemplo

```
printf("Resultado: %d\n", 25+'c');
```

<id, apuntador a la entrada de la tabla de símbolos para **printf** >

<(, >

<str, apuntador a la cadena "Resultado: %d\n">

<,,>

<núm, valor entero 25 >

<+,>

<car, c >

<),>

<!,>


Errores léxicos

- Son pocos los errores que se pueden detectar simplemente a nivel léxico porque un analizador léxico *tiene una visión muy restringida del programa fuente*.
- La estrategia de recuperación más sencilla es el «**modo pánico**». Se borran caracteres sucesivos de la entrada hasta que el analizador léxico pueda encontrar un componente léxico bien formado.
- Otras posibles acciones de recuperación son:
 1. Borrar un carácter extraño
 2. Insertar un carácter que falta
 3. Reemplazar un carácter incorrecto por otro correcto
 4. Intercambiar dos caracteres adyacentes.

Función del analizador léxico

(7)

```
function analex: integer;  
var buflex: array [0..100] of char; c: char;  
begin  
  loop begin  
    lee un carácter en c;  
    if c es un espacio o un tabulador then no hacer nada;  
    else if c es un carácter de línea nueva then numlínea:=numlínea+1;  
    else if c es un dígito then begin  
      asignar a valcomplex el valor de este y los dígitos siguientes;  
      return NUM  
    end  
    else if c es una letra then begin  
      poner c y las letras y dígitos sucesivos en buflex;  
      p:=busca(buflex);  
      if p=0 then p:=inserta(buflex, ID);  
      valcomplex:=p;  
      return el campo complex de la entrada p de la tabla  
    end  
    else begin  
      valcomplex:= NINGUNO;  
      return el número entero del código del carácter c  
    end  
  end  
end  
end
```



Reconocimiento y especificación de componentes léxicos

Reconocimiento de componentes léxicos (1)

Ejemplo

$prop \rightarrow \text{if } expr \text{ then } prop$
 $\quad | \text{if } expr \text{ then } prop \text{ else } prop$
 $\quad | \epsilon$

$expr \rightarrow \text{término oprel término}$
 $\quad | \text{término}$

$\text{término} \rightarrow \text{id} | \text{núm}$

$\text{if} \rightarrow \text{if}$

$\text{then} \rightarrow \text{then}$

$\text{else} \rightarrow \text{else}$

$\text{oprel} \rightarrow < | <= | = | <> | > | >=$

$\text{id} \rightarrow \text{letra} (\text{letra} | \text{dígito})^*$

$\text{núm} \rightarrow \text{dígito} + (. \text{dígito} +) ? (\text{E} (+ | -) ? \text{dígito} +) ?$

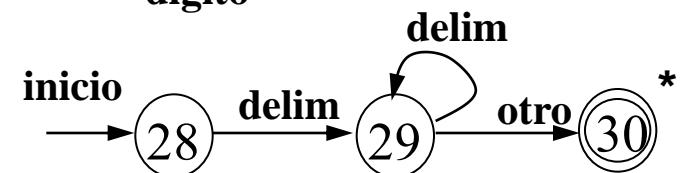
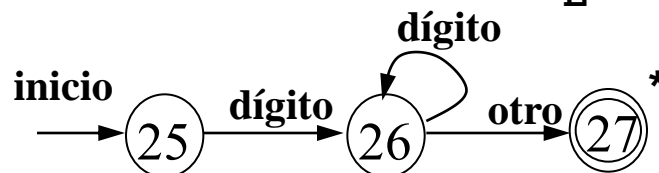
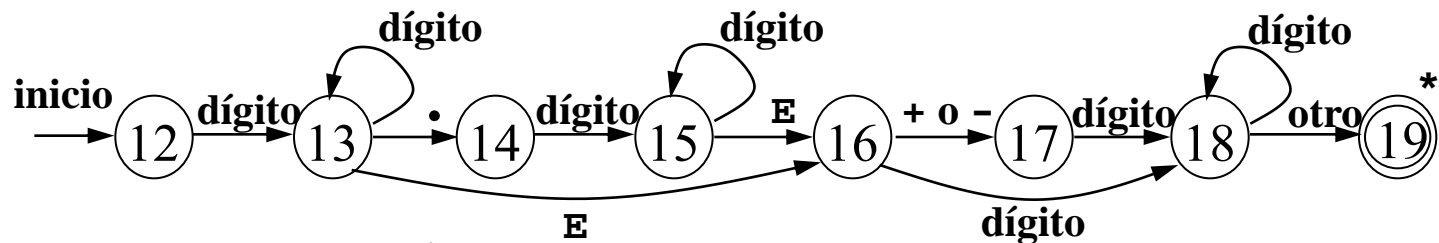
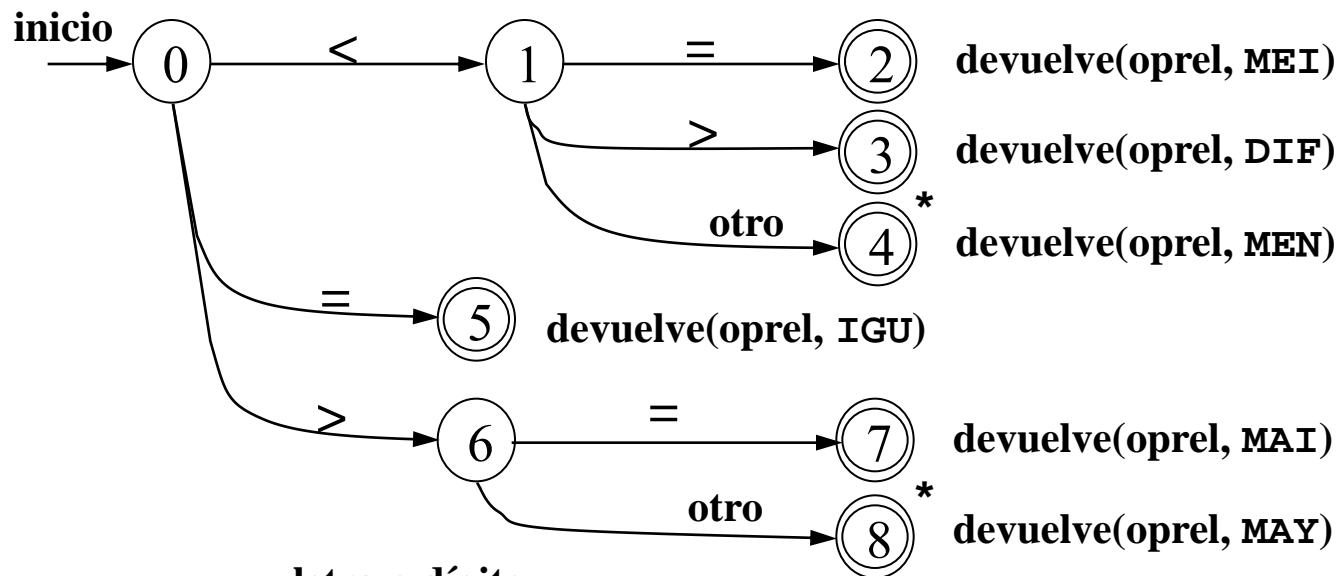
$\text{delim} \rightarrow [\ \backslash \text{t} \backslash \text{n}]$

$\text{eb} \rightarrow \text{delim} +$

Reconocimiento de componentes léxicos (2)

EXPR. REG.	COMP. LÉX.	VALOR DEL ATRIBUTO
eb	-	-
if	if	-
then	then	-
else	else	-
id	id	apuntador a la entrada en la tabla
núm	núm	apuntador a la entrada en la tabla
<	oprel	MEN
<=	oprel	MEI
=	oprel	IGU
<>	oprel	DIF
>	oprel	MAY
>=	oprel	MAI

Reconocimiento de componentes léxicos (3)



Reconocimiento de componentes léxicos (4)

- A partir del diagrama de transiciones se puede construir «a mano» un programa analizador léxico de un modo sistemático.
- En general puede haber varios diagramas de transiciones, cada uno de los cuales especifique un grupo de componentes léxicos.
 - Si surge un fallo mientras se esta siguiendo un diagrama de transiciones, se debe retroceder el apuntador **delantero** hasta donde estaba en el estado inicial de dicho diagrama, y activar el siguiente diagrama de transiciones (igualar el apuntador **delantero** al valor del apuntador **inicio_lexema**).
 - Si el fallo surge en todos los diagramas de transiciones, es que se ha detectado un error léxico y se invoca una rutina de recuperación de errores.

Reconocimiento de componentes léxicos (5)

- Una secuencia de diagramas de transiciones puede convertirse en un programa que busque los componentes léxicos especificados por los diagramas.
- A cada estado del le corresponde un segmento de código.
 - Si hay aristas que salen de un estado, entonces su código lee un carácter y selecciona una arista para seguir, si es posible. Si hay una arista etiquetada con el carácter leído, o etiquetada con una clase de caracteres que contenga el carácter leído, entonces el control se transfiere al código del estado apuntado por esa arista.
 - Si no hay tal arista y el estado en curso de ejecución no es el que indica que se ha encontrado un componente léxico, entonces se llama a la rutina **fallo()** para hacer retroceder el apuntador delantero a la posición del apuntador al comienzo e iniciar la búsqueda del componente léxico especificado por el siguiente diagrama de transiciones.
 - Si no hay que probar más diagramas de transiciones, **fallo()** llama a una rutina de recuperación de errores.
- Para devolver los componentes léxicos se usa una variable global **valor_lexico**.

Reconocimiento de componentes léxicos (listado)

```
int estado = 0, inicio = 0;
int valor_lexico;
int fallo(){
    delantero=inicio_lexema;
    switch(inicio){
        case 0: inicio=9; break;
        case 9: inicio=12; break;
        case 12: inicio=20; break;
        case 20: inicio=25; break;
        case 25: recupera(); break;
        default: /* error */
    } return inicio; }
complex sigte_complex(){
    while(1){
        switch(estado){
case 0:
            c = sigtecar();
            if(c==blanco||c==tab||
                c==newline){
                estado=0; inicio_lexema++;}
            else if (c=='<') estado=1;
            else if (c=='=') estado=5;
            else if (c=='>') estado=6;
            else estado=fallo();
            break;
```

```
case 9:
            c=sigtecar();
            if (isletter(c)) estado=10;
            else estado=fallo(); break;
case 10:
            c=sigtecar();
            if (isletter(c)) estado=10;
            else if (isdigit(c)) estado=10;
            else estado=11; break;
case 11:
            regresa(1);
            valor_lexico=instala_id();
            return ( obten_complex() );
        . . /* aquí los casos 12 al 24 */
case 25:
            c=sigtecar();
            if (isdigit(c)) estado=26;
            else estado=fallo(); break;
case 26:
            c=sigtecar();
            if (isdigit(c)) estado=26;
            else estado=27; break;
case 27:
            regresa(1);
            valor_lexico=instala_num();
            return( NUM );
        }}}}
```

Especificación de componentes léxicos (1)

- Las expresiones regulares son la notación mas generalizada para especificar los patrones de los lexemas de los componentes léxicos.
- Conviene tener claros los siguientes conceptos:
 - alfabeto; cadena sobre un alfabeto; cadena vacía; longitud de una cadena; concatenación y exponenciación de cadenas;
 - lenguaje; lenguaje vacío; unión, concatenación y cerraduras de Kleene y positiva de un lenguaje;
 - **expresión y definición regular**

Especificación de componentes léxicos (2)

- Las **expresiones regulares** se utilizan para denotar lenguajes regulares y se definen del siguiente modo recursivo (dado un alfabeto Σ):
 - ε la expresión regular que denota el conjunto $\{\varepsilon\}$
 - $\forall a \in \Sigma, a$ es una expresión regular que designa al conjunto regular $\{a\}$
 - Si α y β son expresiones regulares que designan los conjuntos regulares C_α y C_β respectivamente, entonces:
 - $(\alpha)|(\beta)$ es la expresión regular que designa al conjunto regular $C_\alpha \cup C_\beta$
 - $(\alpha)(\beta)$ es la expresión regular que designa al conjunto regular $C_\alpha \cdot C_\beta$
 - $(\alpha)^*$ es la expresión regular que designa al conjunto regular C_α^*
 - (α) es la expresión regular que designa al conjunto regular C_α
 - Φ es la expresión regular que representa la **conjunto vacío** (que es regular)
 - Ninguna otra cosa es un expresión regular.
- Además se utilizan las abreviaturas: $+$, $?$, $[c_1c_2... c_n]$, $[c_i-c_f]$

Especificación de componentes léxicos (4)

Ejemplo

- Una expresión regular para describir enteros:

`0|-?(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*`

- Una expresión regular para describir contenidos válidos para el elemento `table` de XHTML (suponiendo que el vocabulario está formado por los elementos: `caption`, `col`, `colgroup`, `thead`, `tfoot`, `tbody`, `tr`):

`caption?(col*|colgroup*)thead?tfoot?(tbody+|tr+)`

Especificación de componentes léxicos (4)

Definición regular: Conjunto de definiciones de la forma:

$d_1 \rightarrow r_1$ donde r_1 es una expresión regular sobre Σ

$d_2 \rightarrow r_2$ donde r_2 es una expresión regular sobre $\Sigma \cup d_1$

$d_3 \rightarrow r_3$ donde r_3 es una expresión regular sobre $\Sigma \cup d_1 \cup d_2$

• • • • • • • • • •

$$d_i \rightarrow r_i \quad \text{donde } r_i \text{ es una expresión regular sobre } \Sigma \cup \bigcup_{j=1}^{i-1} d_j$$

• • • • • • • • • • •

$d_n \rightarrow r_n$ donde r_n es una expresión regular sobre $\Sigma \cup \bigcup_{j=1}^{n-1} d_j$

Con flex no harían falta las flechas

Ejemplo

dígito → **[0-9]** **digito** ([0-9])


letra \rightarrow **[a-zA-Z]**

dígitos \rightarrow **dígito** + dígitos {dígito}+

fracción_opt \rightarrow (. dígitos) ?

exponente_opt \rightarrow (E (+ | -) ? dígitos) ?

núm → **dígitos** **fracción_opt** **exponente_opt**

A decorative vertical bar on the left side of the slide, featuring a black background with glowing green and blue binary code (0s and 1s) arranged in a grid-like pattern.

Autómatas finitos: definición, simulación y obtención

Autómata finito determinista: es una quintupla (Σ, Q, f, q_0, F) , donde:

- Σ es un alfabeto de entrada.
- Q es un alfabeto de estados.
- f es la función de transición:

$$f: Q \times \Sigma \rightarrow Q$$

$$q_0 \times A = q_0$$

- q_0 es el estado inicial.
- $F \subset Q$ es el conjunto de estados finales o estados de aceptación

Autómata finito NO determinista: es una quintupla $(\Sigma, \mathcal{Q}, f, q_0, \mathcal{F})$, donde:

- Σ es un **alfabeto de entrada**.
- \mathcal{Q} es un alfabeto de **estados**.
- f es la **función de transición**:

$$f: \mathcal{Q} \times (\Sigma \cup \{\epsilon\}) \rightarrow \mathcal{P}(\mathcal{Q})$$

- q_0 es el **estado inicial**.
- $\mathcal{F} \subset \mathcal{Q}$ es el conjunto de **estados finales** o **estados de aceptación**

Lo que cambia del NO determinista es la función de transición

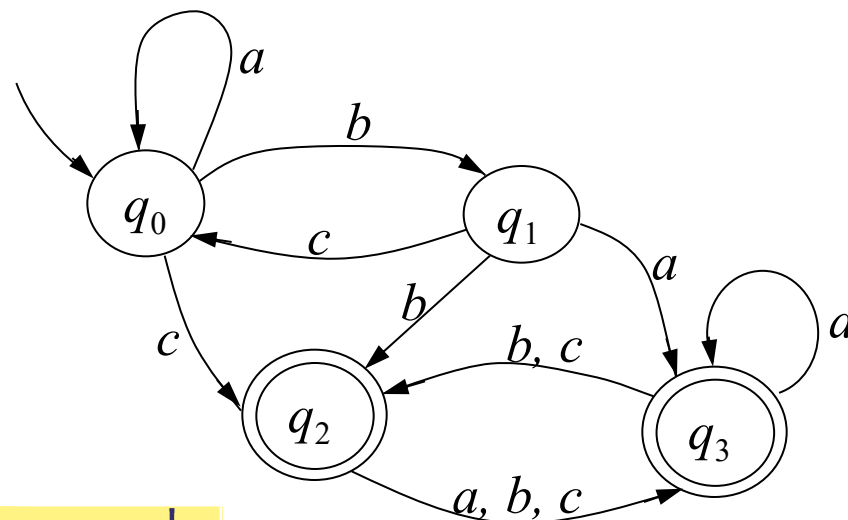
Ahora puedo ir a varios estados.

Autómatas finitos

(3)

- La parte fundamental de un autómata finito es su función de transición por eso es habitual representarlo mediante una tabla que da dicha función o mediante un grafo.

	<i>a</i>	<i>b</i>	<i>c</i>
$\rightarrow q_0$	q_0	q_1	q_2
q_1	q_3	q_2	q_0
(q_2)	q_3	q_3	q_3
(q_3)	q_3	q_2	q_2



estados = nodos

transiciones = arcos

estados finales = nodo con doble círculo

estado inicial = nodo con flecha de entrada

Simulación de un AFD

$s := q_0;$

$c := \text{sgtecar}();$

while $c \neq \text{eof}$ **do**

begin

$s := \text{mueve}(s, c);$

$c := \text{sgtecar}();$

end;

if $s \in \mathcal{F}$ **then return** «*SÍ*» **else return** «*NO*»

mueve (s, c) es una función que implementa la función de transición, dado un estado y un carácter nos devuelve el estado siguiente.

sgtecar () es una función que accede al siguiente carácter de la cadena de entrada.

Simulación de un AFND

$S := \text{cerr-}\varepsilon(\{q_0\});$

$c := \text{sgtecar}();$

while $c \neq \text{eof}$ **do**

begin

$S := \text{cerr-}\varepsilon(\text{mueve}(S, c));$

$c := \text{sgtecar}();$

end

if $S \cap \mathcal{F} \neq \emptyset$ **then return** « S' » **else return** « NO »

mueve (S, c) función de transición,
dado un cjto de estados y un carácter
nos devuelve **los** estados siguientes.

sgtecar () accede al siguiente
carácter de la entrada.

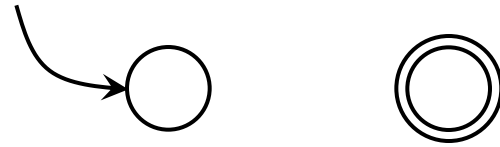
cerr- ε (S) conjunto de estados a los
que se puede llegar a partiendo de
estados de S y siguiendo arcos
etiquetados con ε .

Paso de una ER a un AFN

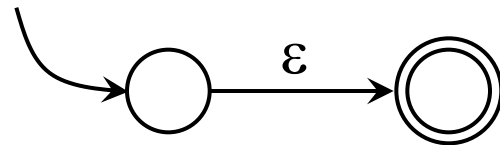
Algoritmo de McNaughton-Yamada-Thompson

(1)

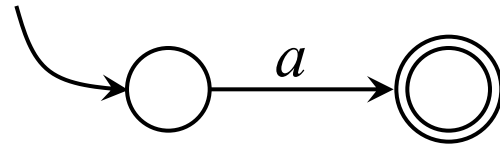
1 Φ



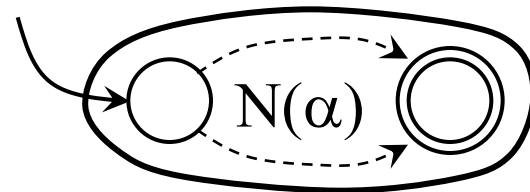
2 ε



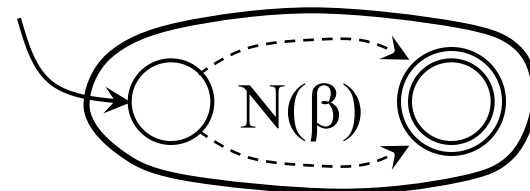
3 a



4 α



5 β

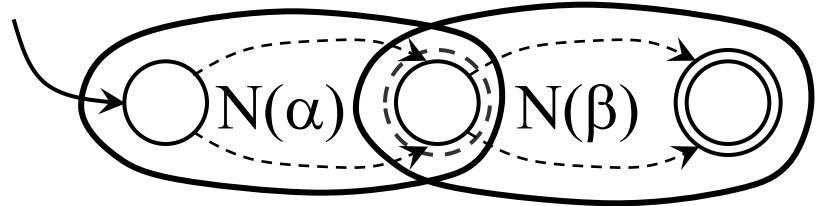


Paso de una ER a un AFN

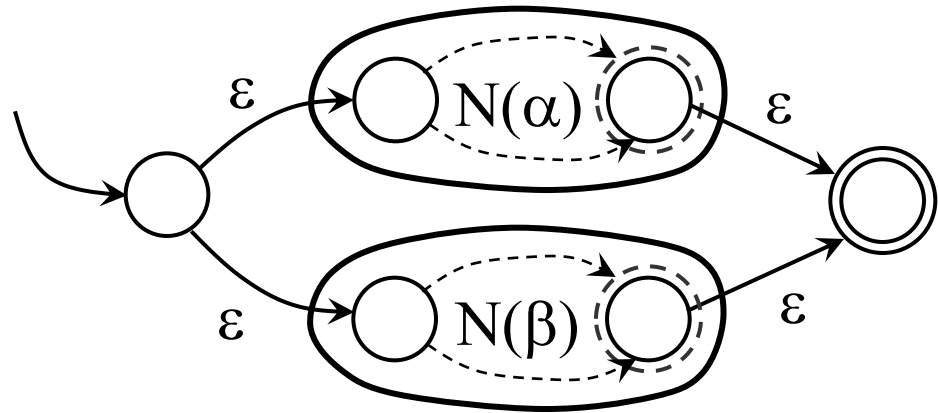
Algoritmo de McNaughton-Yamada-Thompson

(2)

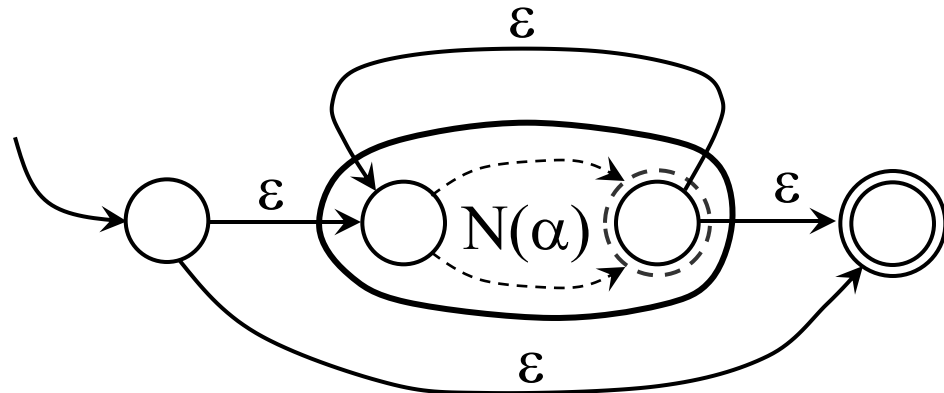
6 $(\alpha)(\beta)$



7 $(\alpha) \mid (\beta)$



8 $(\alpha)^*$



Paso de un AFND a un AFD

$\text{cerr-}\varepsilon(q) = \{\tilde{f}(q, \varepsilon^n)\} \cup \{q\}$. Es decir el conjunto de estados a los que puedo llegar a partir del estado q mediante transiciones ε (sin consumir entrada)

$$\text{cerr-}\varepsilon(T) = \bigcup_{q \in T} \text{cerr-}\varepsilon(q)$$

Algoritmo: Construcción de subconjuntos

$\text{estaD} \leftarrow \text{cerr-}\varepsilon(\{q_0\});$

while *haya un estado T sin marcar en estaD* **do**

marcar T ;

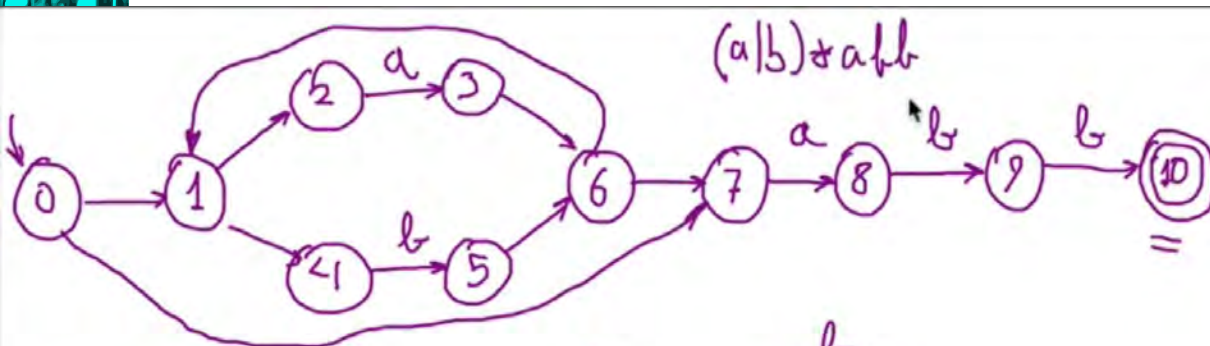
for *cada símbolo de entrada $a \in \Sigma$* **do**

$U := \text{cerr-}\varepsilon(\text{mueve}(T, a));$

if $U \notin \text{estaD}$ **then** *añadir U sin marcar a estaD*

$\text{tranD}[T, a] := U$

$\text{AFND}(\Sigma, Q, f, q_0, F) \rightarrow \text{AFD}(\Sigma, \text{estaD}, \text{tranD}, \text{cerr-}\varepsilon(q_0), F) \quad F = \{U \in \text{estaD} : U \cap F \neq \emptyset\}$



$(a|b)^* a b$

n	$c-\varepsilon(n)$
0	0 1 2 4 7
3	1 2 3 4 6 7
5	1 2 4 5 6 7

a
2 3
7 8

b
4 5
8 9
9 10

	a	b	
0 \rightarrow A	B	C	0 1 2 4 7
3 8 B	B	D	1-4 6-8
5 \bar{C}	\bar{B}	\bar{C}	1 2 4-7
5 9 D	B	E	1 2 4-7 9
5 10 (\bar{E})	\bar{B}	\bar{C}	1 2 4-7 10

$$c-\varepsilon(m(A,a)) = c-\varepsilon(\{3,8\}) = c-\varepsilon(3) \cup c-\varepsilon(8) = 1-4 6-8$$

$$c-\varepsilon(m(A,b)) = c-\varepsilon(\{5\}) = 1, 2, 4-7$$

$$c-\varepsilon(m(B,a)) = c-\varepsilon(\{3,8\}) = B$$

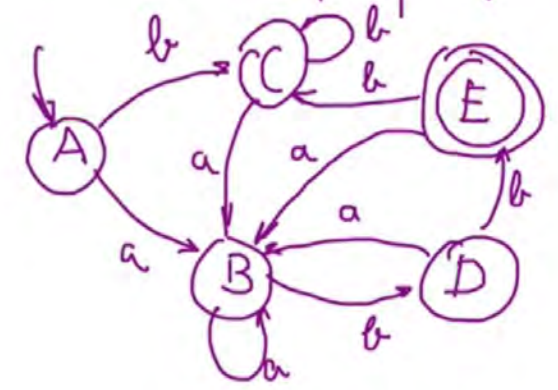
$$c-\varepsilon(m(B,b)) = c-\varepsilon(\{5,9\}) = 1 2 4-7 9$$

$$c-\varepsilon(m(C,a)) = c-\varepsilon(\{3,8\}) = B$$

$$c-\varepsilon(m(C,b)) = c-\varepsilon(\{5\}) = C$$

$$c-\varepsilon(m(D,a)) = c-\varepsilon(\{3,8\}) = B \quad c-\varepsilon(m(D,b)) = c-\varepsilon(\{5,10\}) = 1 2 4-7 10$$

$$c-\varepsilon(m(E,a)) = c-\varepsilon(\{3,8\}) = B \quad c-\varepsilon(m(E,b)) = c-\varepsilon(\{5\}) =$$



Construcción de un AFD a partir de una ER (1)

Pasos

- Construir la e.r. ampliada $\alpha\$$ $\$ \notin \Sigma$
- Obtener el árbol sintáctico de $\alpha\$$
- Responder a las preguntas:
 - ¿Que letras pueden aparecer como primera letra de la e.r. representada por α cuáles como últimas?
 - ¿Supuesto que yo se que letra acaba de aparecer, que letras pueden venir a continuación en las palabras representadas por α ?
 - La idea es tratar de numerar la posición de las letras que aparecen en la e.r. Una vez que tengo el árbol puedo hacer una numeración, lo cual me da una ordenación parcial. Se establece una equivalencia entre los **estados significativos** del autómata y las posiciones del árbol (nodos hoja).

ER --> AFN --> AFD

Pero vamos a aprender a hacer ER --> AFD directamente

Ejemplo: $(a|b)^*c$ --> $((a|b)^*c)\$$ -->

Construcción de un AFD a partir de una ER (2)

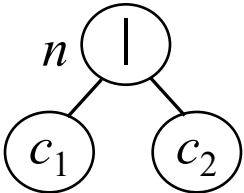
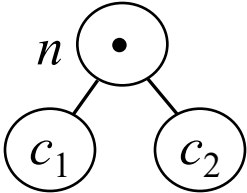
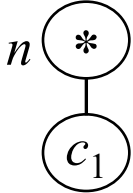
- Permite construir directamente un AFD a partir de una *expresión regular aumentada* $((\alpha)\$ \ \$ \notin \Sigma)$.
- Primero se construye un árbol sintáctico T para $\alpha\$$ con las posiciones de las hojas numeradas y después se calculan cuatro funciones: *anulable*, *primera-pos*, *última-pos* y *siguiente-pos* haciendo recorridos sobre T .
- Por último se construye el AFD a partir de la función *siguiente-pos*. Las funciones *anulable*, *primera-pos* y *última-pos* se definen sobre los nodos del árbol sintáctico y se usan para calcular *siguiente-pos*, que está definida en el conjunto de posiciones de las hojas en el árbol.

Construcción de un AFD a partir de una ER (3)

- La noción de una posición concordando con un símbolo de entrada será definida en cuanto a la función *siguiente-pos* en posiciones del árbol sintáctico. Si i es una posición, $\text{siguiente-pos}(i)$ es el conjunto de posiciones j tales que hay alguna cadena de entrada $\dots cd\dots$ Tal que i corresponde a esta aparición de c y j a esta aparición de d .
- Para calcular la función *siguiente-pos*, es necesario conocer qué posiciones pueden concordar con el primer o último símbolo de una cadena generada por una determinada subexpresión de una expresión regular. Para esto es también necesario conocer qué nodos son las raíces de las subexpresiones que generan lenguajes que incluyen la cadena vacía (nodos *anulables*)

Construcción de un AFD a partir de una ER (4)

- Cálculo de las funciones: *anulable*, *primera-pos* y *última-pos*

<i>Nodo</i>	<i>anulable(n)</i>	<i>primera-pos(n)</i>	<i>última-pos(n)</i>
<i>n</i> es una hoja con símbolo ε	true	\emptyset	\emptyset
<i>n</i> es una hoja con la posición <i>i</i>	false	$\{i\}$	$\{i\}$
	$anul(c_1) \vee anul(c_2)$	$pripos(c_1) \cup pripos(c_2)$	$últpos(c_1) \cup últpos(c_2)$
	$anul(c_1) \wedge anul(c_2)$	if $anul(c_1)$ then $pripos(c_1) \cup pripos(c_2)$ else $pripos(c_1)$	if $anul(c_2)$ then $últpos(c_1) \cup últpos(c_2)$ else $últpos(c_2)$
	true	$pripos(c_1)$	$últpos(c_1)$

Construcción de un AFD a partir de una ER (5)

- La función *siguiente-pos*(p) indica qué posiciones pueden seguir a la posición p en el árbol sintáctico. Dos reglas definen todas las formas en que una posición puede seguir a otra:
 - Si n es un *nodo-cat* con hijo izquierdo c_1 e hijo derecho c_2 , y p es una posición dentro de *última-pos*(c_1), entonces todas las posiciones de *primera-pos*(c_2) están en *siguiente-pos*(p).
$$\forall p \in \text{última-pos}(c_1) \quad \text{siguiente-pos}(p) \supset \text{primera-pos}(c_2)$$
 - Si n es un *nodo-ast*, e i es una posición dentro de *última-pos*(n), entonces todas las posiciones de *primera-pos*(n) están en *siguiente-pos*(p)
$$\forall p \in \text{última-pos}(c_1) \quad \text{siguiente-pos}(p) \supset \text{primera-pos}(c_1)$$

Construcción de un AFD a partir de una ER (6)

Algoritmo para obtener el AFD a partir de *siguiente-pos*

$estaD := primera-pos(raíz);$

while haya un estado T sin marcar en $estaD$ **do**
 marcar T ;

for cada símbolo de entrada $a \in \Sigma$ **do**

$U := \bigcup_{p \in T: p \text{ etiqueta una hoja con símbolo } a} siguiente-pos(p);$

if $(U \notin estaD) \wedge (U \neq \emptyset)$ **then** añadir U sin marcar a $estaD$

$tranD[T, a] := U$

end

end

AFD($\Sigma, estaD, tranD, primera-pos(raíz), F$) $F = \{U \in estaD: p \in U \text{ y } p \text{ la posición de \$}\}$

Minimización de los estados de un AFD

- Dado un AFD $(\Sigma, \mathcal{Q}, q_0, f, \mathcal{F})$, el AFD mínimo asociado $(\Sigma, \mathcal{Q}/\equiv, [q_0], [f], \mathcal{F}/\equiv)$ se puede construir mediante el siguiente algoritmo:
 1. $k = 0$. Construir la partición Π_0 de \mathcal{Q} constituida por las clases \mathcal{F} y $\mathcal{Q} - \mathcal{F}$
 2. repetir
 - a. incrementar k
 - b. construir Π_k partiendo de Π_{k-1} y manteniendo en la misma clase dos estados q y q' si y solamente si $\forall a \in \Sigma$ los estados $f(q, a)$ y $f(q', a)$ están en la misma clase de Π_{k-1}hasta que $\Pi_{k-1} = \Pi_k$ (lo cual ocurrirá antes de que $k = n-1$)
 3. Π_{k-1} es \mathcal{Q}/\equiv

A decorative vertical bar on the left side of the slide, featuring a dark teal background with a pattern of glowing green and yellow binary code (0s and 1s) arranged in a grid-like fashion.

Cuestiones de implementación

Manejo de *buffers* de entrada

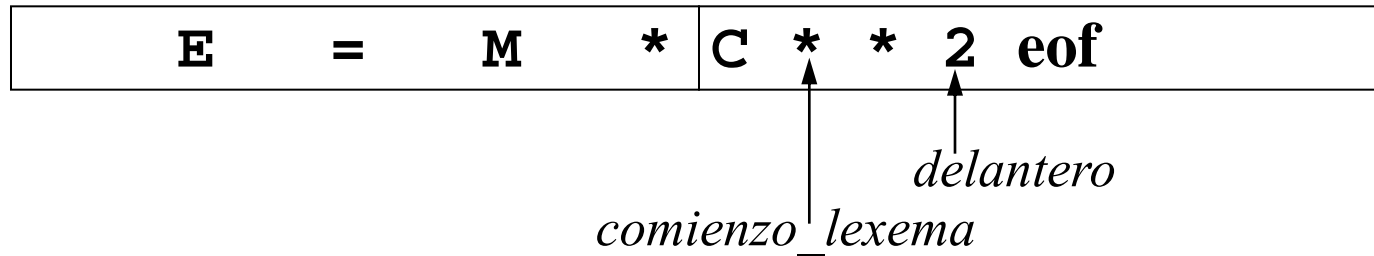
(1)

- Existen tres métodos generales de implantación de un analizador léxico:
 - Utilizar un generador de analizadores léxicos.
 - Escribir el analizador léxico en un lenguaje convencional de programación de sistemas.
 - Escribir el analizador léxico en lenguaje ensamblador.
- Tienen un orden de dificultad creciente, los enfoques más difíciles consiguen analizadores léxicos más rápidos.
- Como el analizador léxico es la única fase que lee el programa carácter a carácter, su velocidad es un problema en el diseño de compiladores.
- Además, en muchos lenguajes el analizador léxico necesita pre-analizar varios caracteres antes de poder anunciar una concordancia. Los caracteres pre-analizados se tienen que devolver después a la entrada. Como se puede consumir mucho tiempo moviendo caracteres, se han desarrollado técnicas especializadas en el manejo de *buffers* para reducir el número de operaciones necesarias.

Manejo de *buffers* de entrada

(2)

- Se utiliza un *buffer* dividido en dos mitades de N caracteres cada una.



- Se leen N caracteres en cada mitad del *buffer* con una orden de lectura de sistema, en vez de invocar una instrucción de lectura para cada carácter. Si quedan menos de N caracteres en la entrada, entonces se lee un carácter especial **eof** en el *buffer* después de los caracteres de entrada.
- Se mantienen dos apuntadores al *buffer* de entrada. La cadena de caracteres entre los dos apuntadores es el lexema en curso. Al principio, los dos apuntadores apuntan al primer carácter del próximo lexema que hay que encontrar.
- El apuntador *delantero* examina hacia delante hasta encontrar una concordancia con un patrón. Una vez determinado el siguiente lexema, el apuntador *delantero* se coloca en el carácter de su extremo derecho.

Manejo de *buffers* de entrada

(3)

- Después de haber procesado el lexema, ambos apuntadores se colocan en el carácter situado inmediatamente después del lexema.
- Cuando el apuntador *delantero* esta a punto de sobrepasar la marca intermedia del *buffer*, se llena la mitad derecha.
- Cuando el apuntador *delantero* está a punto de sobrepasar el extremo derecho del *buffer*, se llena la mitad izquierda y el apuntador *delantero* se regresa al principio del *buffer*.

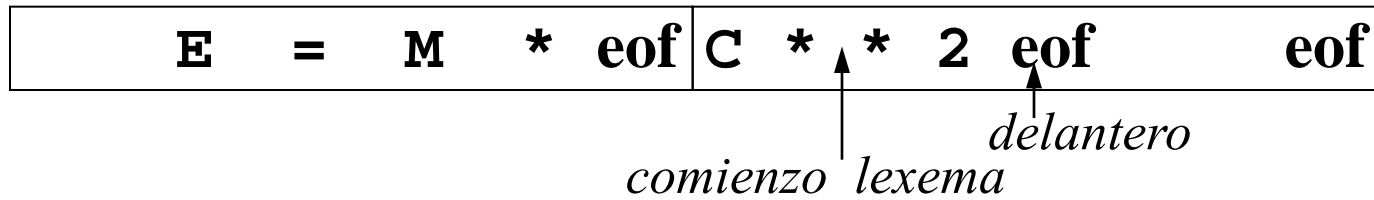
```
if delantero está al final de la primera mitad then begin
    recargar la segunda mitad;
    delantero := delantero + 1;
end
else if delantero está al final de la segunda mitad then begin
    recargar la primera mitad;
    pasar delantero al principio de la primera mitad
end
else delantero := delantero + 1;
```

- Este esquema funciona muy bien, pero limita la cantidad de caracteres de pre-análisis, esto puede imposibilitar el reconocimiento de los componentes léxicos cuando la distancia recorrida por el apuntador *delantero* sea mayor que la longitud del *buffer*.

Manejo de *buffers* de entrada

(4)

- El algoritmo, tal como se ha expuesto, necesita dos comparaciones para cada avance del apuntador *delantero*. Se pueden reducir a una las comparaciones si se amplía cada mitad del *buffer* para admitir un carácter *centinela* al final.



- Este código realiza sólo una comparación en la mayoría de las ocasiones

```
delantero := delantero + 1;  
if delantero = eof then begin  
  if delantero esta al final de la primera mitad then begin  
    recargar la segunda mitad;  
    delantero := delantero + 1;  
  end  
  else if delantero al final de la segunda mitad then begin  
    recargar la primera mitad;  
    pasar delantero al principio de la primera mitad  
  end  
  else /* eof dentro de buffer*/ terminar la entrada  
end
```

Diseño de un generador de analizadores léxicos (1)

- Supóngase que se tiene una especificación de un analizador léxico de la forma:

p_1 $\{ acción_1 \}$

p_2 $\{ acción_2 \}$

...

p_n $\{ acción_n \}$

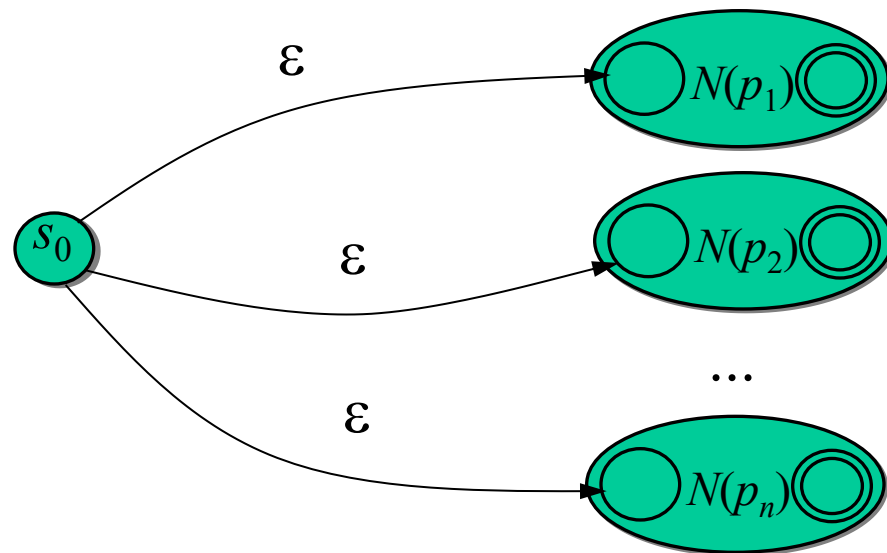
p_i es una expresión regular

$acción_i$ es un fragmento de programa que debe ejecutarse siempre que se encuentre en la entrada un lexema que concuerde con p_i

- El problema es construir un reconocedor que busque lexemas en el *buffer* de la entrada. Si concuerda más de un patrón, el reconocedor elegirá el lexema más largo que haya concordado. Si hay dos o más patrones que concuerden con el lexema más largo, se elige el primer patrón que haya concordado de la lista.

Diseño de un generador de analizadores léxicos (2)

- Un método es construir la tabla de transiciones de un autómata finito no determinista N para el patrón compuesto $p_1|p_2| \dots | p_n$. Esto se puede hacer creando primero un AFND $N(p_i)$ para cada patrón p_i utilizando el algoritmo de Thompson, añadiendo después un nuevo estado de inicio s_0 , y por último enlazando s_0 al estado de inicio de cada $N(p_i)$ con una transición ε .



Diseño de un generador de analizadores léxicos (3)

- Cuando se simula el AFND, se construye la secuencia de conjuntos de estados donde puede estar el AFND combinado después de leer cada carácter de entrada.
- Siempre que se añada un estado de aceptación al conjunto de estados en curso, se registran la posición del puntero delantero y el patrón p_i correspondiente a este estado de aceptación. Si el conjunto de estados en curso ya contiene un estado de aceptación, entonces solo se registra el patrón que aparezca primero en la especificación de LEX.

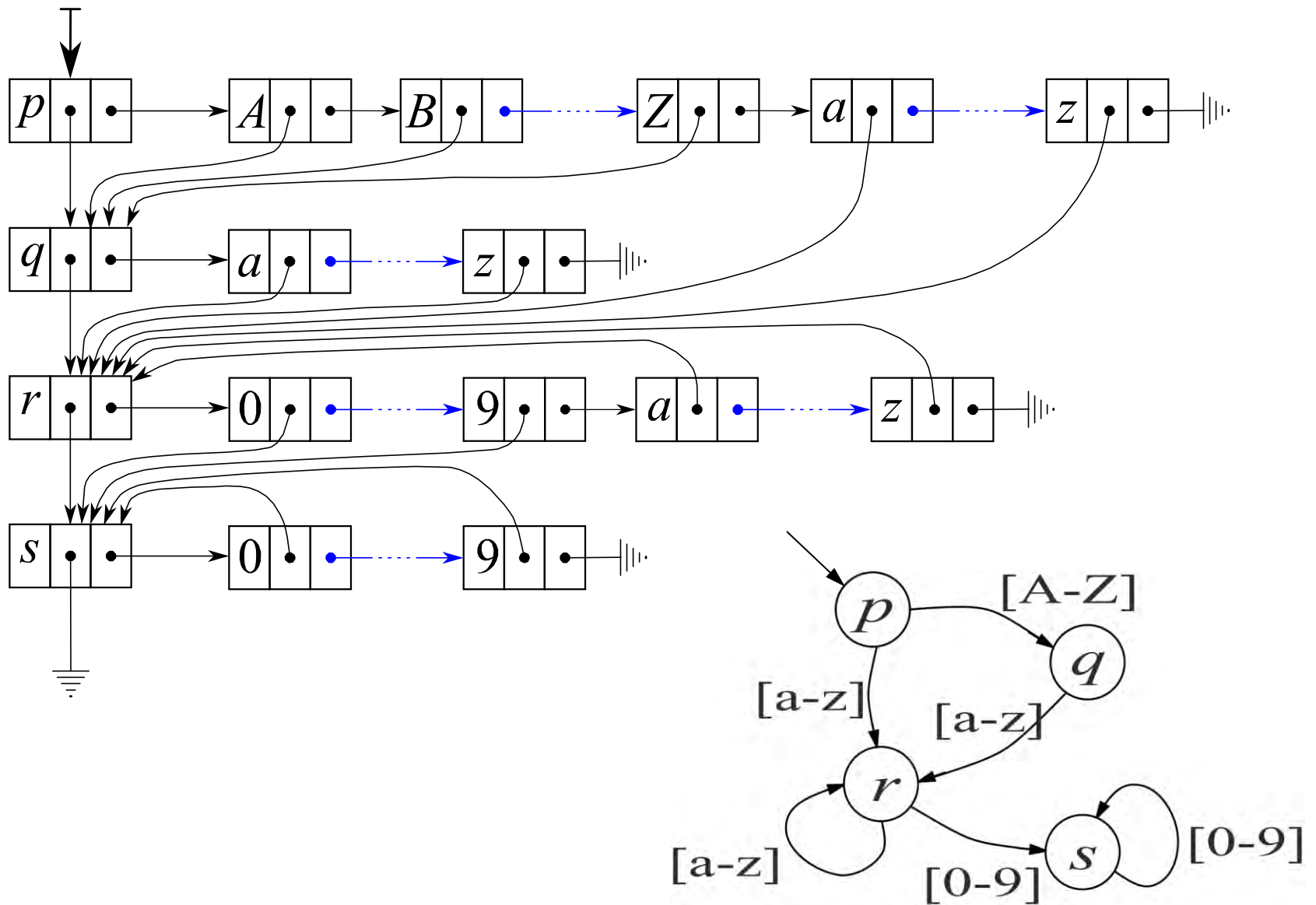
Diseño de un generador de analizadores léxicos (4)

- Incluso si se encuentra un conjunto de estado que contenga un estado de aceptación, para encontrar la concordancia más larga se debe seguir simulando el AFND hasta alcanzar *terminación*, es decir, un conjunto de estados desde el que no hay transiciones con el símbolo de entrada en curso.
- En la terminación, se retrocede el apuntador delantero a la posición en que ocurrió la última concordancia. El patrón que hizo dicha concordancia identifica al componente léxico encontrado, y el lexema emparejado es la cadena entre los apuntadores de *comienzo_lexema* y *delantero* (además existe un patrón de error).

- Dado que el proceso de análisis léxico ocupa una parte considerable del tiempo del compilador, si se utiliza un AFD para ayudar a implantar el analizador léxico, es aconsejable una representación eficiente de la función de transición.
- Existen muchas formas de implantar la función de transición de un autómata finito:
 - Matriz bidimensional, indexada por estados y caracteres. Proporciona el acceso más rápido, pero puede ocupar demasiado espacio.
 - Lista enlazada para almacenar las transiciones de salida de cada estado, con una transición «por omisión» al final de la lista: es un esquema mas compacto, pero más lento.

Compresión de tablas

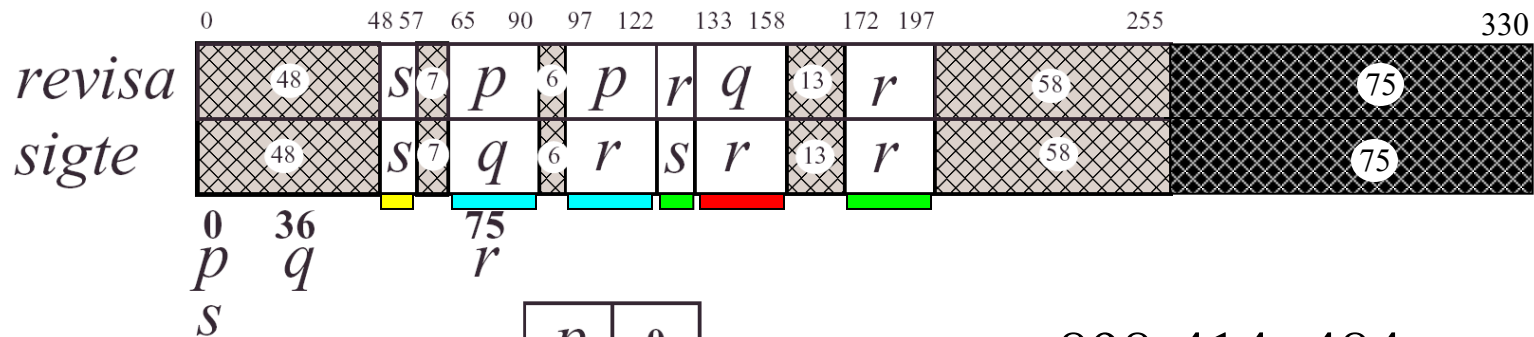
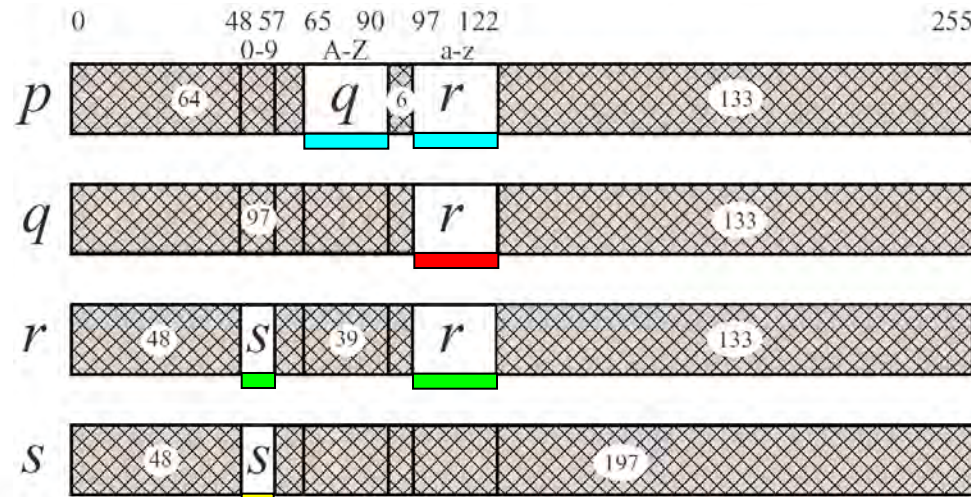
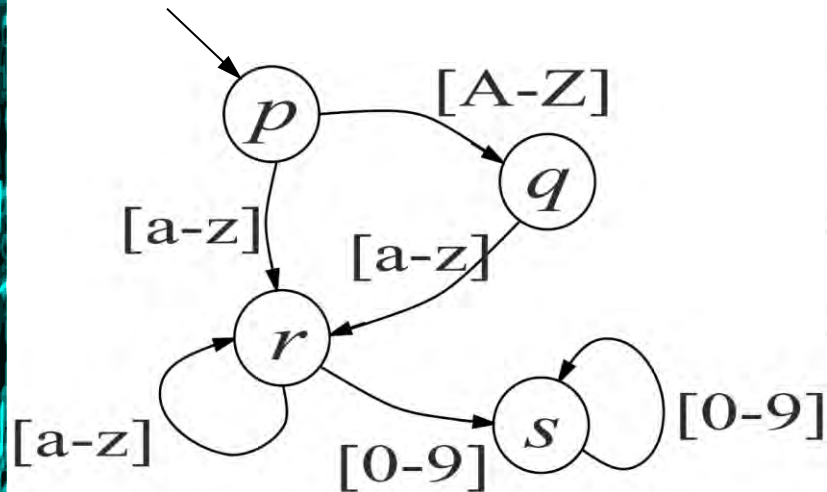
(2)



Compresión de tablas

(3)

Ejemplo



base

p	0
q	36
r	75
s	0

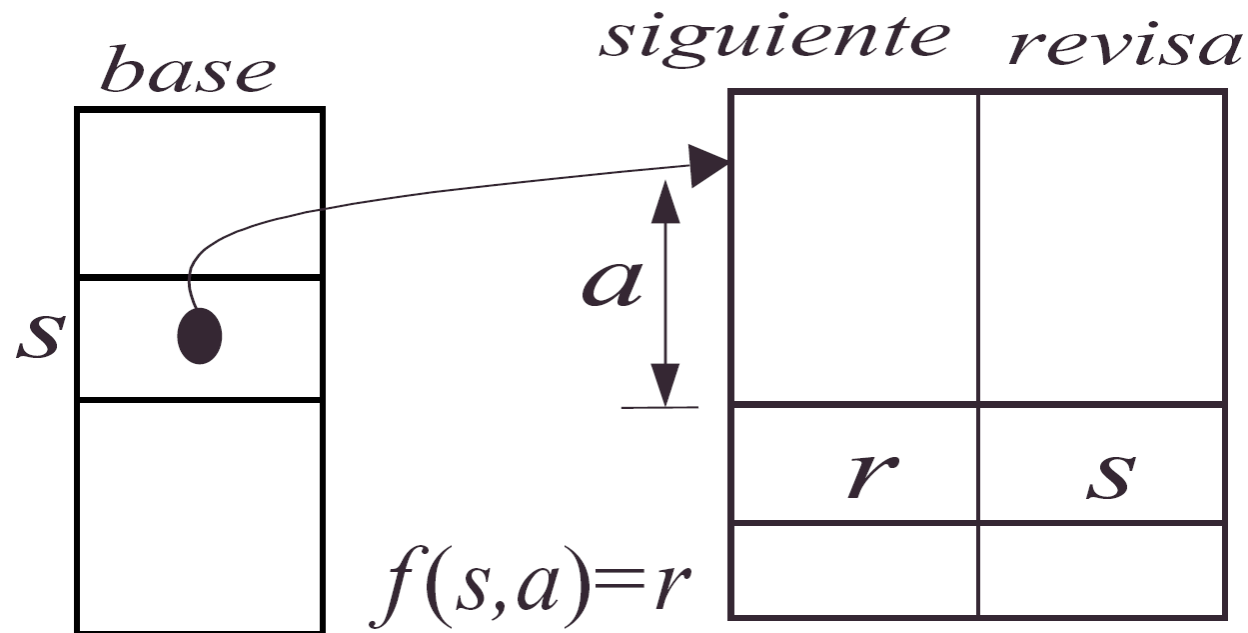
$$898 - 414 = 484$$

$$\approx 54\%$$

Compresión de tablas

(4)

- Existe una representación que combina el acceso rápido de la representación por medio de matrices con la compacidad de las estructuras de listas. Se utiliza una estructura de datos que consta de tres matrices indexadas por números de estados.



- Se utiliza la matriz *base* para determinar la posición base de las entradas para cada estado almacenado en las matrices *siguiente* y *revisa*.

```
function sigte_estado(s, a);  
  if revisa[base[s] + a] = s then  
    return siguiente[base[s] + a]  
  else return -1
```

- Para iniciar la matriz *siguiente* puede usarse una estrategia de primero el que mejor ajuste. La siguiente fila a almacenar se «mueve» sobre la siguiente sin que se produzca «colisión».

Simulación eficiente de AFND

(1)

- 1) $S = \epsilon\text{-closure}(s_0);$
- 2) $c = \text{nextChar}();$
- 3) **while** ($c \neq \text{eof}$) {
- 4) $S = \epsilon\text{-closure}(\text{move}(S, c));$
- 5) $c = \text{nextChar}();$
- 6) }
- 7) **if** ($S \cap F \neq \emptyset$) **return** "yes";
- 8) **else return** "no";

Figure 3.37: Simulating an NFA

- c es el siguiente carácter de entrada leído por la función *nextChar()*
- Primero se calcula $\text{move}(S, c)$ y a continuación se hace la cerradura épsilon

- Una implementación eficiente del algoritmo requiere las siguientes estructuras de datos:
 1. Dos pilas para almacenar el conjunto de estados del AFND.
 - Una de las pilas, *oldStates*, mantiene el conjunto de estados «en curso» – el valor de *S* a la derecha de la asignación (4).
 - La otra, *newStates*, almacena el siguiente conjunto de estados, el valor de *S* a la izquierda de la asignación. Tras cada iteración los valores de *newStates* se transfieren a *oldStates*.
 2. Un array de booleanos, *alreadyOn*, indexado por los estados del ANFD, que indica que estados hay en *newStates* (es más rápido hacer la comprobación *alreadyOn[s]*, que buscar el estado en *newStates*)
 3. Una matriz *move[s, a]* que almacena la función de transición del AFND. Sus entradas son conjuntos de estados (representados, por ejemplo, como una lista enlazada).

Simulación eficiente de AFND

(3)

- El pseudocódigo para añadir estados sería

```
9)  addState(s) {  
10)      push s onto newStates;  
11)      alreadyOn[s] = TRUE; {  
12)      for ( t on move[s,  $\epsilon$ ] )  
13)          if ( !alreadyOn[t] )  
14)              addState(t);  
15)  }
```

Figure 3.38: Adding a new state s , which is known not to be on *newStates*

```
push all states of T onto stack;  
initialize  $\epsilon$ -closure(T) to T  
while ( stack is not empty ) {  
    pop t, the top elemento, of stack;  
    for ( each state u with an edge from t to u labeled  $\epsilon$  )  
        if ( u is not in  $\epsilon$ -closure(T) ) {  
            add u to  $\epsilon$ -closure(T);  
            push u onto stack;  
        }  
}
```

Figure 3.33: Computing ϵ -closure(*T*)

- La implementación del paso 4:

```
16)  for ( s on oldStates ) {  
17)      for ( t on move[s, c] )  
18)          if ( !alreadyOn[t] )  
19)              addState(t);  
20)      pop s from oldStates;  
21)  }  
22)  for ( s on newStates ) {  
23)      pop s from newStates;  
24)      push s onto oldStates;  
25)      alreadyOn[s] = FALSE;
```

Figure 3.39: Implementation of step (4) of Fig. 3.37

Complejidades de los AFs

AUTOMATON	SPACE	TIME
NFA	$O(r)$	$O(r \times x)$
DFA (worst case)	$O(2^{ r })$	$O(x)$
DFA (lazy evaluation)	$O(c + x)$	$O(x)$

$|x|$ string length

$|r|$ regular expresion length

$|c|$ cache size

Complejidades de los AFs

AUTOMATON	INITIAL	PER STRING
NFA	$O(r)$	$O(r \times x)$
DFA typical case	$O(r ^3)$	$O(x)$
DFA worst case	$O(r ^2 \times 2^{ r })$	$O(x)$
DFA lazy evaluation	$O(c + x)$	$O(x)$

$|x|$ string length

$|r|$ regular expresion length

$|c|$ cache size

