Compiladores e Intérpretes

2 Análisis Léxico

Paulo Félix Lamas
Área de Ciencias da Computación e Intelixencia Artificial
Departamento de Electrónica e Computación

[Índice]

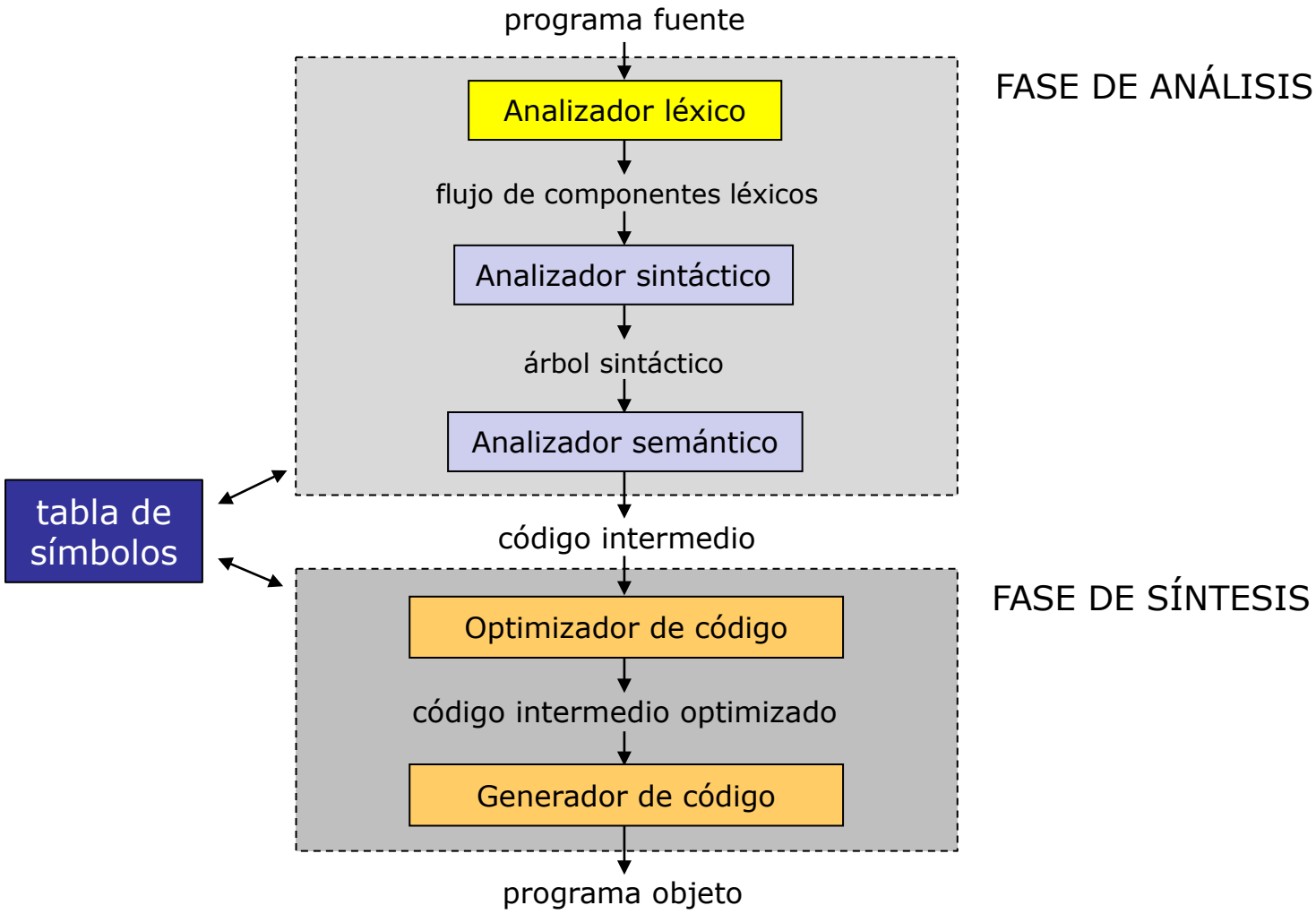
Introducción

- En este tema estudiaremos la primera fase del proceso de compilación: el análisis léxico del programa fuente.
- Comenzaremos contextualizando el análisis léxico en el proceso global de compilación, destacando su aportación e importancia.
- Formalizaremos este análisis mediante la teoría de autómatas y el uso de las nociones de expresión regular, autómata finito determinista y no determinista.
- Expondremos el problema de la gestión del código fuente mediante el estudio del sistema de entrada.
- Introduciremos la tabla de símbolos, que será de utilidad en todo el proceso de compilación.
- Introduciremos el problema del tratamiento de errores.

Objetivos

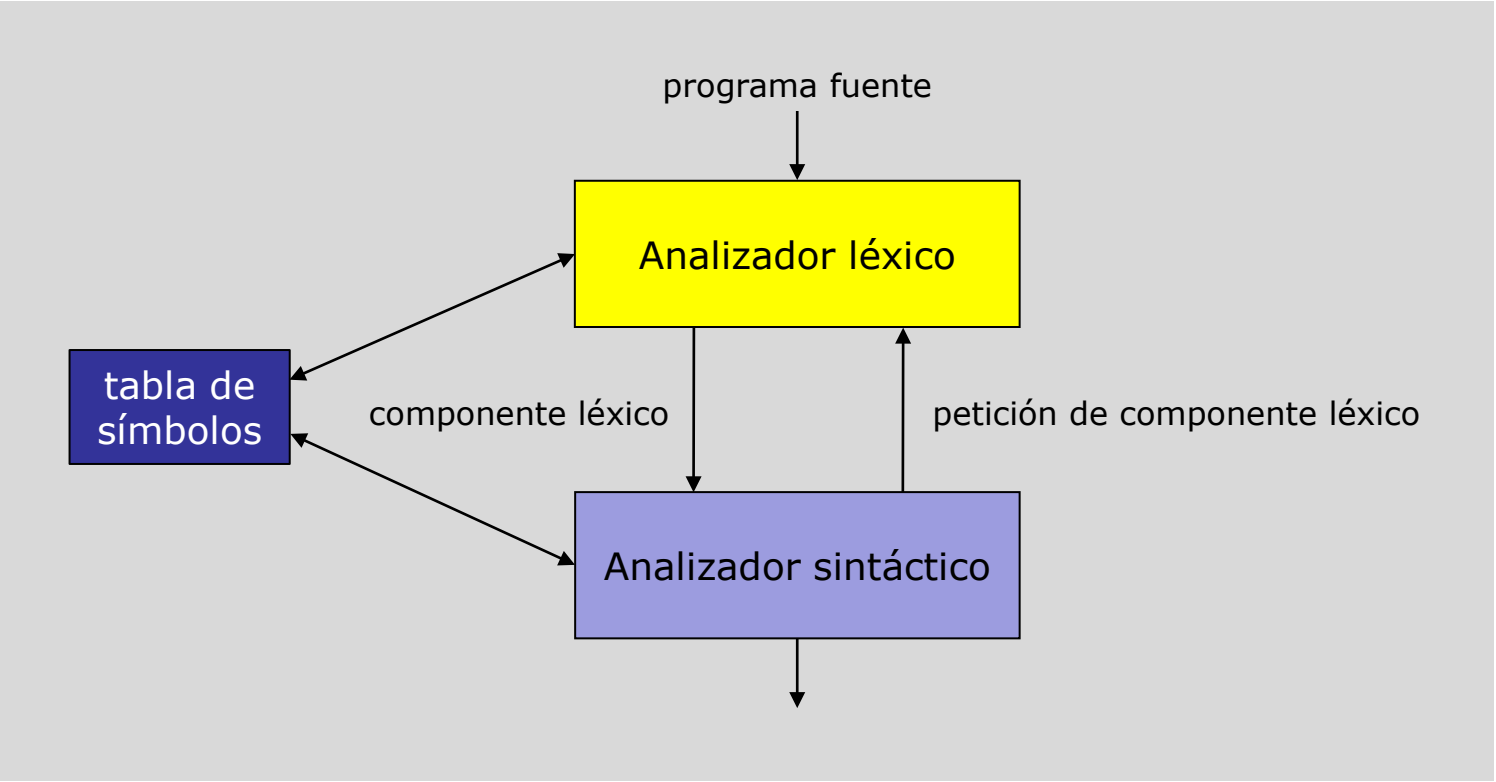
1. Comprender la estructura, organización y funcionamiento de un analizador léxico.
2. Saber definir los elementos léxicos de un lenguaje utilizando expresiones regulares.
3. Saber utilizar un generador automático de analizadores léxicos.
4. Construir las expresiones regulares de un lenguaje sencillo y programarlas en un generador automático de analizadores léxicos.
5. Comprender las ventajas de un sistema de entrada independiente y su funcionamiento.
6. Saber cómo se utiliza una tabla de símbolos y conocer su estructura.

1. Estructura



1. Estructura

- Los analizadores sintáctico y léxico funcionan según el patrón productor-consumidor, siguiendo el siguiente esquema:



1. Estructura.
Tareas del analizador léxico

1. Reconocer los componentes léxicos del lenguaje.

2. Eliminar aquellos caracteres del código fuente sin significado gramatical: los espacios en blanco, los caracteres de tabulación, los saltos de línea y de página,...

3. Eliminar los comentarios del programa fuente.

4. Reconocer los identificadores de variable, tipo, constantes, métodos, etc. y guardarlos en la tabla de símbolos.

5. Avisar de los errores léxicos detectados, y relacionar los mensajes de error con el lugar en el que aparecen en el programa fuente (línea, columna, etc.).

1. Estructura.
Ventajas del análisis léxico

- Las ventajas de separar los análisis léxico y sintáctico son:

  1. Se simplifica el análisis en general. Aislamos el análisis sintáctico de la llegada de caracteres inesperados, y así sólo ha de trabajar con la estructura del lenguaje. El diseño del compilador gana en claridad.

  2. Se mejora la eficiencia del compilador. Podemos aplicar técnicas específicas de mejora en cada fase.

  3. Se mejora la portabilidad del compilador. Si deseamos cambiar alguna característica del alfabeto de entrada, podemos cambiar el analizador léxico sin tocar el analizador sintáctico.

1. Estructura.
Diseño del análisis léxico

- Dividiremos el diseño del análisis léxico en etapas:
  1. Especificación de los términos del análisis léxico mediante el uso de expresiones regulares.
  2. Diseño de un autómata finito que permita reconocer las expresiones regulares propuestas.
  3. Realización de un autómata finito determinista mínimo que permita realizar un reconocimiento eficiente.
  4. Diseño y realización de un sistema de entrada.
  5. Diseño y realización de una tabla de símbolos.
  6. Diseño y realización de una estrategia de manejo de errores.

2. Especificación del analizador.
Términos del análisis léxico

- Componente léxico (*token*): símbolo terminal de la gramática que define el lenguaje fuente. Pueden ser signos de puntuación, operadores, palabras reservadas, identificadores,…

- Patrón: expresión regular que define el conjunto de cadenas correspondientes a un componente léxico. Así, ´[0-9]$^{+}$´ identifica una cadena de una o más cifras, correspondiente al componente léxico NÚMERO_ENTERO.

- Lexema: cadena de caracteres presente en el código fuente y que coincide con el patrón de un componente léxico. Por ejemplo, ´453´ coincide con el patrón de NÚMERO_ENTERO.

- Atributos: acompañan a cada componente léxico encontrado y permiten su identificación y análisis posterior. En la práctica un único atributo apunta a una entrada de la tabla de símbolos.

2. Especificación del analizador.
Términos del análisis léxico

Ejemplos:

Componente léxico	Patrón	Lexemas
IDENTIFICADOR	[A-Za-z][A-Za-z0-9]*	imax, velocidad, rpm
NÚMERO_ENTERO	$[0\text{-}9]^+$	245, 0, 9384100
MAYOR_O_IGUAL	>=	>=

La sentencia FORTRAN `E = C ** 2` se traduce como:

1. <IDENTIFICADOR,apuntador en la tabla de símbolos a la entrada de E>
2. <OP_ASIGNACIÓN>
3. <IDENTIFICADOR,apuntador en la tabla de símbolos a la entrada de C>
4. <OP_EXPONENTE>
5. <NÚMERO_ENTERO,valor entero 2>

Ejercicio 2

- Sea el código fuente siguiente, escrito en PASCAL:

```
function maximo(num1,num2:integer):integer;
(* devuelve el maximo de dos numeros *)
begin
     if num1<num2
        then maximo:=num2
        else maximo:=num1
end; (* maximo *)
```

Se pide identificar los lexemas que forman los componentes léxicos del código.

2. Especificación del analizador. Expresiones regulares

- Una expresión regular r permite representar patrones de caracteres. El conjunto de cadenas representado por r recibe el nombre de lenguaje generado por r, y se escribe L(r).

- Para un alfabeto Σ diremos que:

  1. El símbolo $\emptyset$ (conjunto vacío) es una expresión regular y $L(\emptyset)=\{\}$.

  2. El símbolo ε (palabra vacía) es una expresión regular y $L(\varepsilon)=\{\varepsilon\}$.

  3. Cualquier símbolo $a \in \Sigma$ es una expresión regular y $L(a)=\{a\}$.

2. Especificación del analizador. Expresiones regulares

- A partir de estas expresiones regulares básicas pueden construirse expresiones regulares más complejas aplicando las siguientes operaciones:

  1. Concatenación: Si r y s son expresiones regulares, entonces rs también lo es, y L(rs)=L(r)L(s). Por ejemplo, si L_1={00,1} y L_2={11,0} entonces L_1L_2={0011,000,111,10}.

  2. Unión: Si r y s son expresiones regulares, entonces r|s también lo es, y L(r|s)=L(r)∪L(s). Por ejemplo, el lenguaje generado por ab|c es L(ab|c)={ab,c}.

  3. Cierre o clausura: Si r es una expresión regular, entonces r* también lo es, y L(r*)=L(r)*. Por ejemplo, el lenguaje generado por a*ba* es L(a*ba*)={b,ab,ba,aba,aab,…}

2. Especificación del analizador. Expresiones regulares

Ejemplos:

1. La expresión regular a|b designa el conjunto {a,b}.
2. La expresión regular (a|b)(a|b) designa {aa,ab,ba,bb}. Otra expresión regular para el mismo conjunto es aa|ab|ba|bb. Se dice que ambas expresiones regulares son equivalentes.
3. La expresión regular a* designa {ε,a,aa,aaa,…}.
4. La expresión regular (a|b)* designa el conjunto de todas las cadenas de a y b. Otra expresión regular equivalente es (a*b*)*.
5. La expresión regular a|a*b designa el conjunto que contiene la cadena a y todas aquellas con cero o más a seguidas de una b.

2. Especificación del analizador. Definiciones regulares

- Por conveniencia, daremos nombres a las expresiones regulares, utilizando dichos nombres como si fuesen símbolos:

$$d \rightarrow r$$

Ejemplo: Definimos los números sin signo en Pascal como:

```
digito → 0|1|…|9
digitos → digito digito*
fraccion → .digitos|ε
exponente → (E(+|-|ε)digitos)|ε
numero → digitos fraccion exponente
```

Números sin signo en Pascal son cadenas como 478, 23.12, o 3.14E-7.

2. Especificación del analizador. Simplificando la notación

- Cero o un caso: El operador unitario postfijo ? significa "cero o un caso de". Así r? abrevia r|ε, y designa el lenguaje L(r)∪{ε}.
- Uno o más casos: El operador unitario postfijo + significa "uno o más casos de". Si r designa el lenguaje L(r), entonces r^+ designa $(L(r))^+$. Se cumple que $r^*=r^+|\varepsilon$ y $r^+=rr^*$.
- Clases de caracteres: La notación [abc] designa la expresión regular a|b|c. [a-z] designa la expresión a|b|…|z.

Ejemplo:

```
digito → [0-9]
digitos → digito+
fraccion → (.digitos)?
exponente → (E(+|-)?digitos)?
numero → digitos fraccion exponente
```

Ejercicio 3

- Sea el lenguaje siguiente:

  Cadenas de caracteres formadas por una secuencia alfanumérica encerrada entre comillas dobles, sin otras comillas dobles en medio, a menos que aparezcan precedidas de \, como en \”. A su vez, de querer que aparezca \ también deberá venir precedido de \, como en \\.

Se pide proporcionar una definición regular para este lenguaje.

3. Autómatas finitos.
Autómata finito no determinista

- Utilizaremos autómatas finitos en el reconocimiento de expresiones regulares. Un autómata finito no determinista (AFN) es una quíntupla (Q,Σ,δ,q_0,F) donde:
  - Q es un conjunto finito de estados.
  - Σ es el alfabeto de entrada.
  - δ es la función de transición definida como:

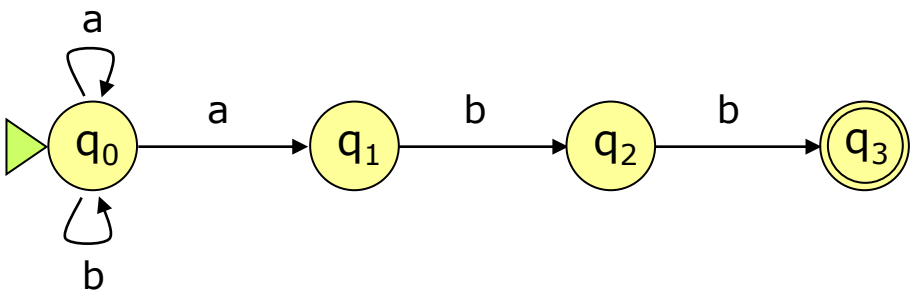
$$\delta: Q \times \Sigma \rightarrow P(Q)$$

  - $q_0 \in Q$ es el estado inicial.
  - $F \subseteq Q$ es el conjunto de estados finales.

3. Autómatas finitos.
Autómata finito no determinista

Ejemplo:

Diseñar un AFN que reconozca el lenguaje representado por la expresión regular (a|b)*abb:

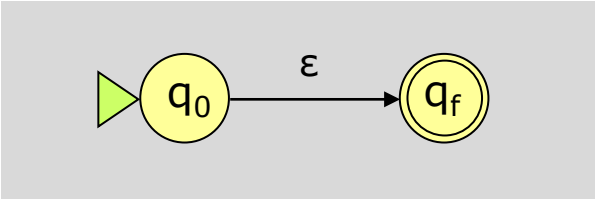


Un AFN se puede representar mediante su tabla de transiciones:

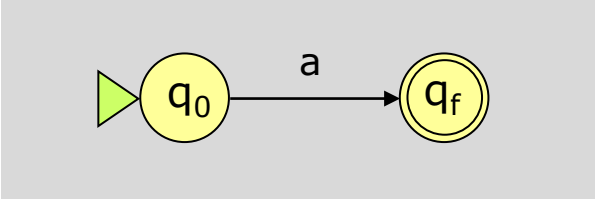
estado	símbolo de entrada	
	a	b
0	{0,1}	{0}
1	-	{2}
2	-	{3}

3. Autómatas finitos.
Diseño de un AFN

- ¿Cómo construir un AFN de un modo sistemático (esto es, programable) a partir de una expresión regular?
- Utilizaremos la denominada Construcción de Thompson, que utiliza tres reglas básicas:
  1. Para el símbolo ε, se construye el AFN siguiente:



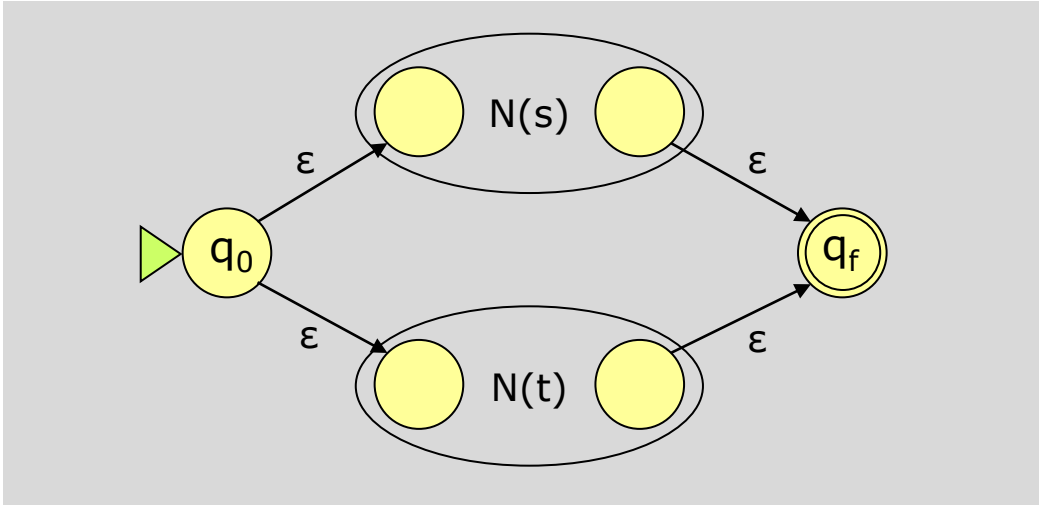
  2. Para el símbolo a∈Σ, se construye el AFN siguiente:



3. Autómatas finitos.
Diseño de un AFN

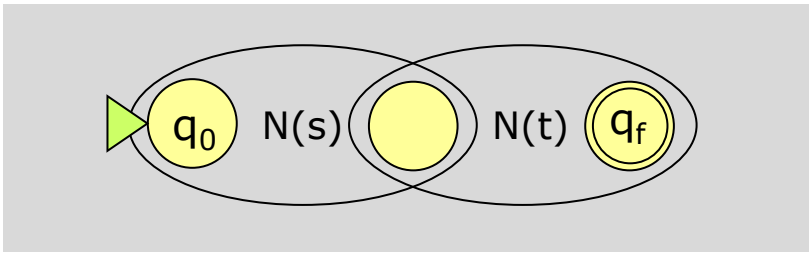
3. Supongamos que N(s) y N(t) son AFN para las expresiones regulares s y t:

   a) Para la expresión regular s|t se construye el AFN N(s|t) siguiente:

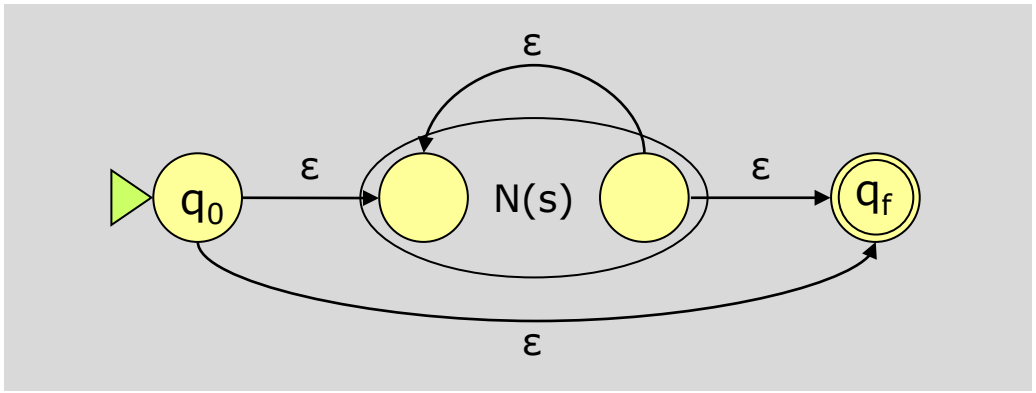


3. Autómatas finitos.
Diseño de un AFN

b) Para la expresión regular st se construye el AFN N(st) siguiente:



c) Para la expresión regular s* se construye el AFN N(s*) siguiente:



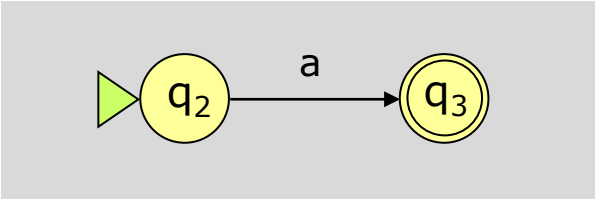
3. Autómatas finitos.
Diseño de un AFN

- Se puede demostrar que la aplicación sistemática de las reglas anteriores da lugar a un AFN N(r) que reconoce la expresión regular inicial r. Este AFN tiene las siguientes propiedades:
  1. N(r) tiene un estado inicial y un estado final. Esta propiedad la satisfacen asimismo todos los AFN que lo componen.
  2. N(r) tiene a lo sumo el doble de estados que de símbolos y operadores en r: cada vez que se aplica una regla se crean a lo sumo dos nuevos estados.
  3. Cada estado de N(r) tiene una transición saliente con un símbolo $a \in \Sigma$, o a lo sumo dos transiciones salientes con símbolos ε.

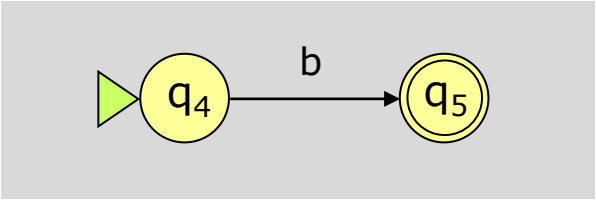
3. Autómatas finitos.
Diseño de un AFN

Ejemplo: Construir un AFN que reconozca el lenguaje representado por la expresión regular (a|b)*abb:

1. Para la primera a, N(a) es:

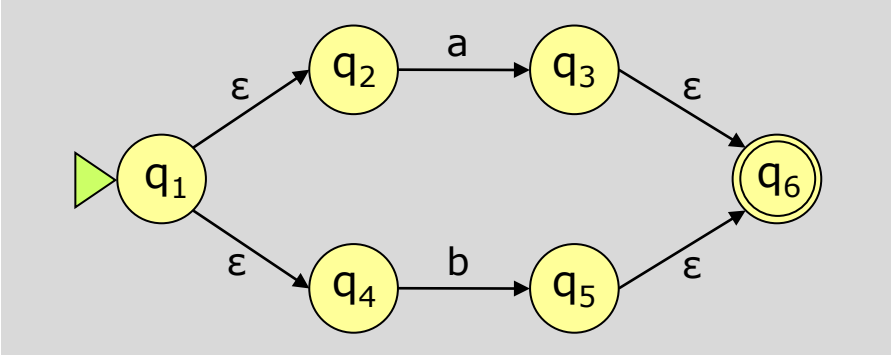


2. Para la primera b, N(b) es:

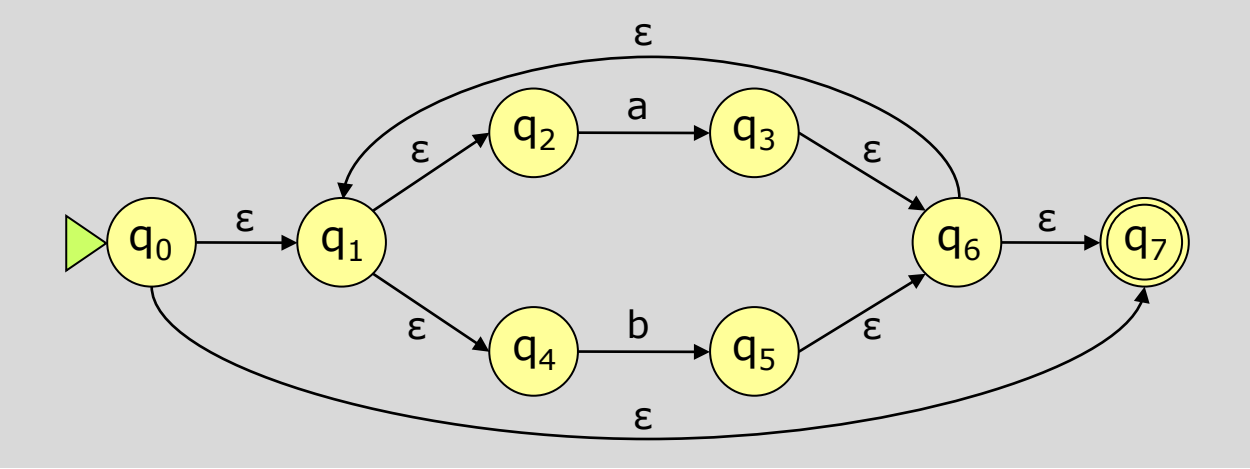


3. Autómatas finitos.
Diseño de un AFN

3. Combinamos N(a) y N(b) para obtener N(a|b):

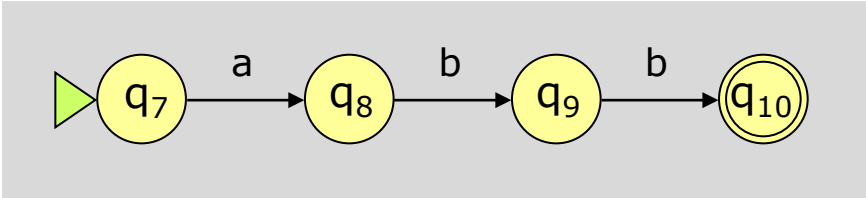


4. Para obtener N((a|b)*):

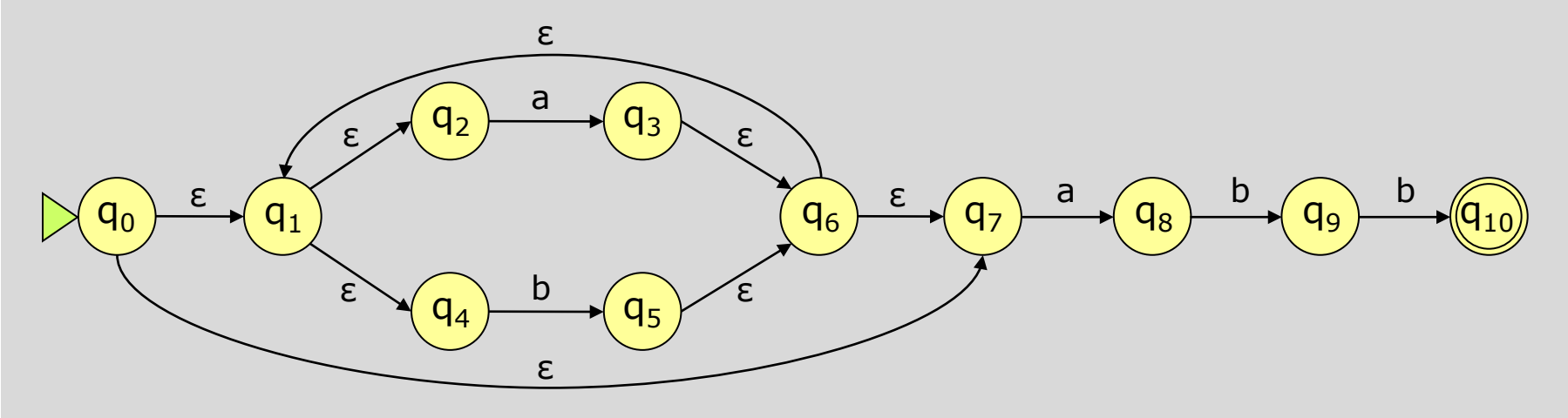


3. Autómatas finitos.
Diseño de un AFN

5. Simplificamos pasos para la expresión abb:



6. Para terminar concatenamos:



3. Autómatas finitos.
Autómata finito determinista

- Utilizaremos autómatas finitos deterministas para un reconocimiento eficiente de las expresiones regulares. Un autómata finito determinista (AFD) es una quíntupla (Q,Σ,δ,q_0,F) donde:
  - Q es un conjunto finito de estados.
  - Σ es el alfabeto de entrada.
  - δ es la función de transición definida como:

$$\delta: Q \times \Sigma \rightarrow Q$$

  - $q_0 \in Q$ es el estado inicial.
  - $F \subseteq Q$ es el conjunto de estados finales.

3. Autómatas finitos.
AFD equivalente a un AFN

- Utilizaremos el método de construcción de subconjuntos para obtener el AFD equivalente a un AFN dado. Este AFD será más sencillo de programar.
- En la tabla de transiciones de un AFN, a cada entrada le corresponde un conjunto de estados. En la de un AFD, a cada entrada le corresponde uno solo.
- La idea de este método es que a cada estado del AFD le corresponde un conjunto de estados del AFN equivalente.
- El AFD utiliza cada estado para localizar todos los posibles estados en los que puede estar el AFN equivalente después de leer cada símbolo de entrada.

3. Autómatas finitos.
AFD equivalente a un AFN

- Sea el AFN $N= (Q^N,\Sigma,\delta^N,q^N_0,F^N)$, deseamos obtener un AFD equivalente $D= (Q^D,\Sigma,\delta^D,q^D_0,F^D)$. Para ello, utilizaremos las tres operaciones siguientes:

  1. cierre-ε(q), es el conjunto de estados del AFN N que se pueden alcanzar desde el estado $q \in Q^N$ con transiciones ε solamente.
  2. cierre-ε(Q), es el conjunto de estados del AFN N que se pueden alcanzar desde los estados que pertenecen al conjunto $Q \subset Q^N$ con transiciones ε solamente.
  3. mueve(Q,σ), es el conjunto de estados del AFN N a los cuales llega una transición desde los estados que pertenecen al conjunto $Q \subset Q^N$ con el símbolo $\sigma \in \Sigma$.

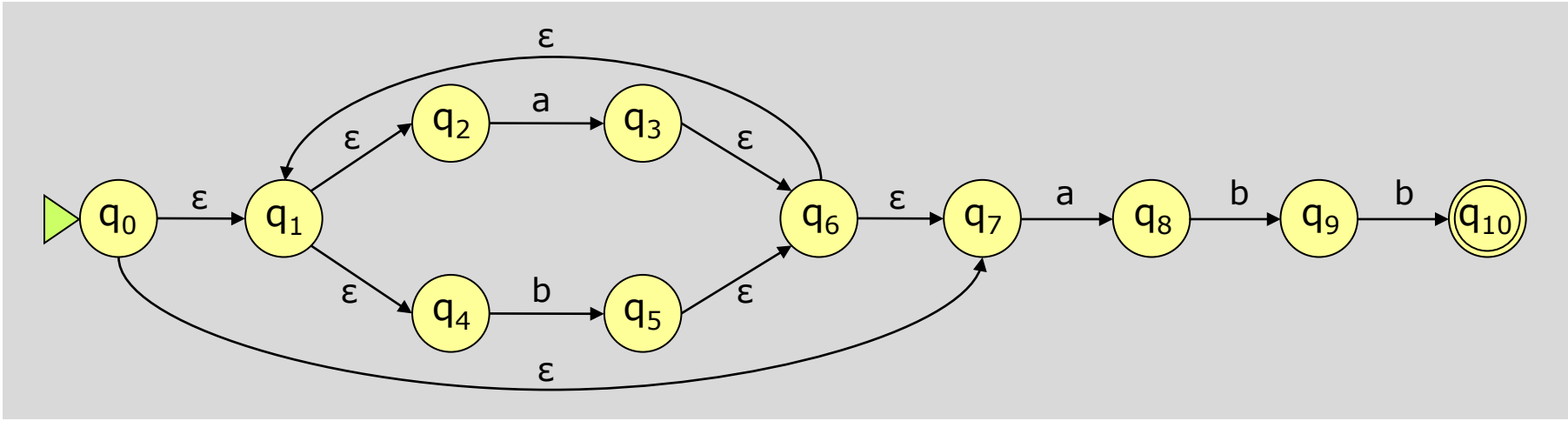
3. Autómatas finitos.
AFD equivalente a un AFN

- Seguiremos los siguientes pasos para la obtención del AFD D equivalente a un AFN N dado:
  1. Antes de detectar el primer símbolo de entrada, N se puede encontrar en cualquiera de los estados del conjunto cierre-ε(q^N_0). Por tanto, llamaremos q^D_0 al conjunto de esos estados $q \in Q^N$ tales que $q \in$ cierre-ε(q^N_0).
  2. Desde cualquier estado q^D_i hay una transición a un estado q^D_j con el símbolo $\sigma \in \Sigma$. Calculamos este estado q^D_j en dos pasos: 1) primero calculamos en el AFN N el conjunto mueve(q^D_i,σ) (recordemos que $q^D_i \subset Q^N$); 2) calculamos su cierre-ε. En definitiva, q^D_j =cierre-ε(mueve(q^D_i,σ)).
  3. En el AFD D, un estado será final si contiene algún estado final del AFN N.

3. Autómatas finitos.
AFD equivalente a un AFN

Ejemplo: Construir un AFD que reconozca el lenguaje representado por la expresión regular (a|b)*abb:

1. Utilizaremos como punto de partida el AFN obtenido:



2. El estado inicial del AFD equivalente es el cierre-ε(q_0):

$$q^D_0 = \{q_0, q_1, q_2, q_4, q_7\}$$

3. Autómatas finitos.
AFD equivalente a un AFN

3. Tenemos dos símbolos de entrada $\Sigma=\{a,b\}$. Comenzamos calculando q^D_1=cierre-ε(mueve(q^D_0,a)). En dos pasos:

   1) mueve(q^D_0,a) = mueve($\{q_0,q_1,q_2,q_4,q_7\}$,a) =$\{q_3,q_8\}$

   2) cierre-ε($\{q_3,q_8\}$) =$\{q_1,q_2,q_3,q_4,q_6,q_7,q_8\}$

   Por tanto, q^D_1 =$\{q_1,q_2,q_3,q_4,q_6,q_7,q_8\}$

4. Calculamos ahora q^D_2=cierre-ε(mueve(q^D_0,b)):

   1) mueve(q^D_0,b) = mueve($\{q_0,q_1,q_2,q_4,q_7\}$,b) =$\{q_5\}$

   2) cierre-ε($\{q_5\}$) =$\{q_1,q_2,q_4,q_5,q_6,q_7\}$

   Por tanto, q^D_2 =$\{q_1,q_2,q_4,q_5,q_6,q_7\}$

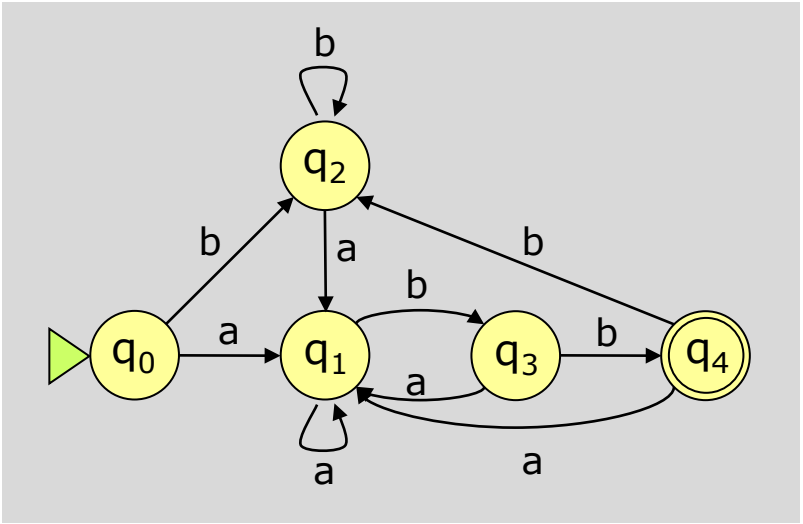
3. Autómatas finitos.
AFD equivalente a un AFN

5. Obtenemos otros 2 estados diferentes más:

   $q^D_3 = \{q_1, q_2, q_4, q_5, q_6, q_7, q_9\}$

   $q^D_4 = \{q_1, q_2, q_4, q_5, q_6, q_7, q_{10}\}$

6. Y con ello un AFD D equivalente al AFN N de partida:



3. Autómatas finitos.
AFD equivalente a un AFN

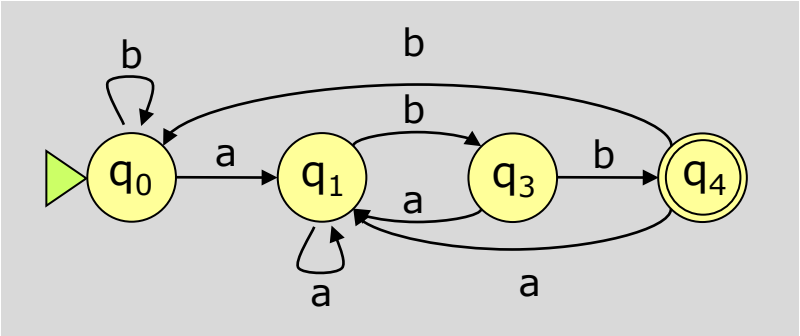
Con la siguiente tabla de transiciones asociada:

estado	símbolo de entrada	
	a	b
0	1	2
1	1	3
2	1	2
3	1	4
4	1	2

3. Autómatas finitos.
AFD mínimo equivalente

- Un analizador léxico será más eficiente cuanto menor sea el número de estados del AFD correspondiente.
- Para cualquier AFD existe un AFD mínimo equivalente.

Ejemplo: El siguiente AFD también reconoce (a|b)*abb y tiene un estado menos:

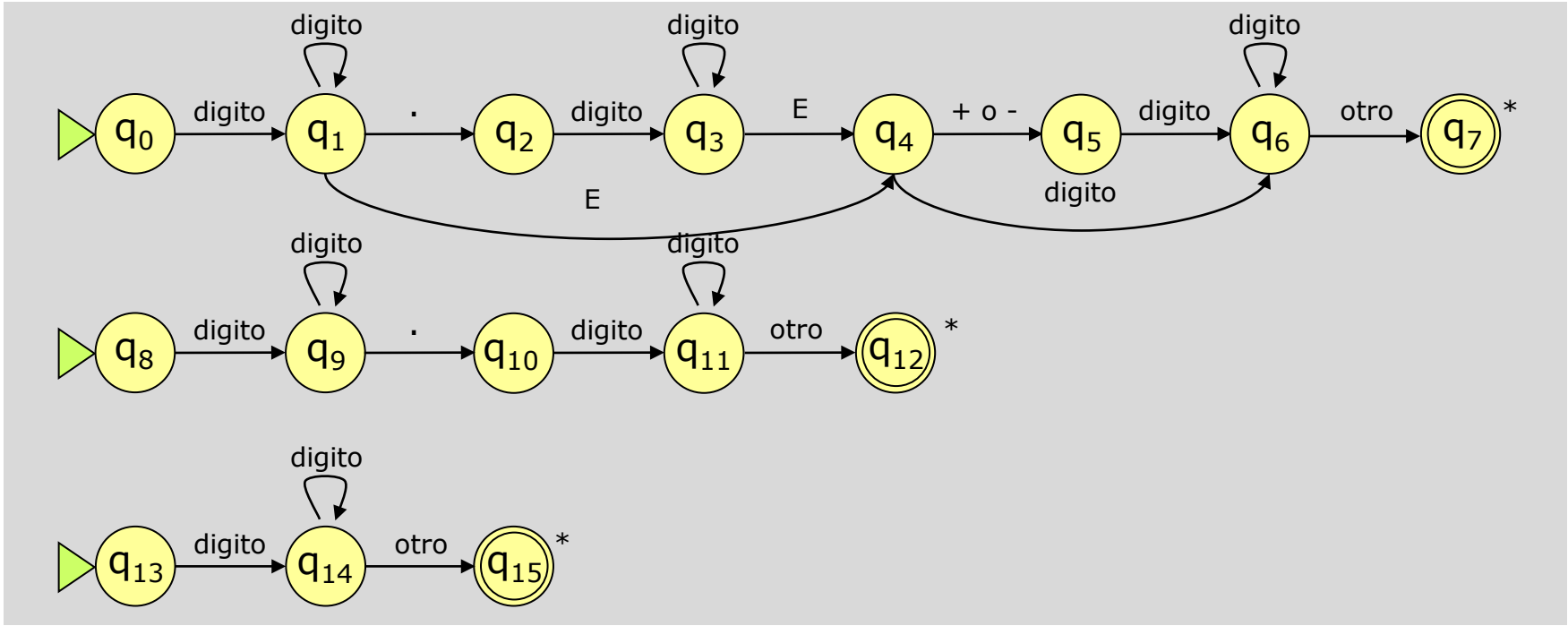


- La idea es identificar pares de estados equivalentes: q_0 y q_2 son equivalentes ya que el AFD se comporta igual para cualquier cadena que detecta a partir de ellos.

3. Autómatas finitos.
Construcción del analizador

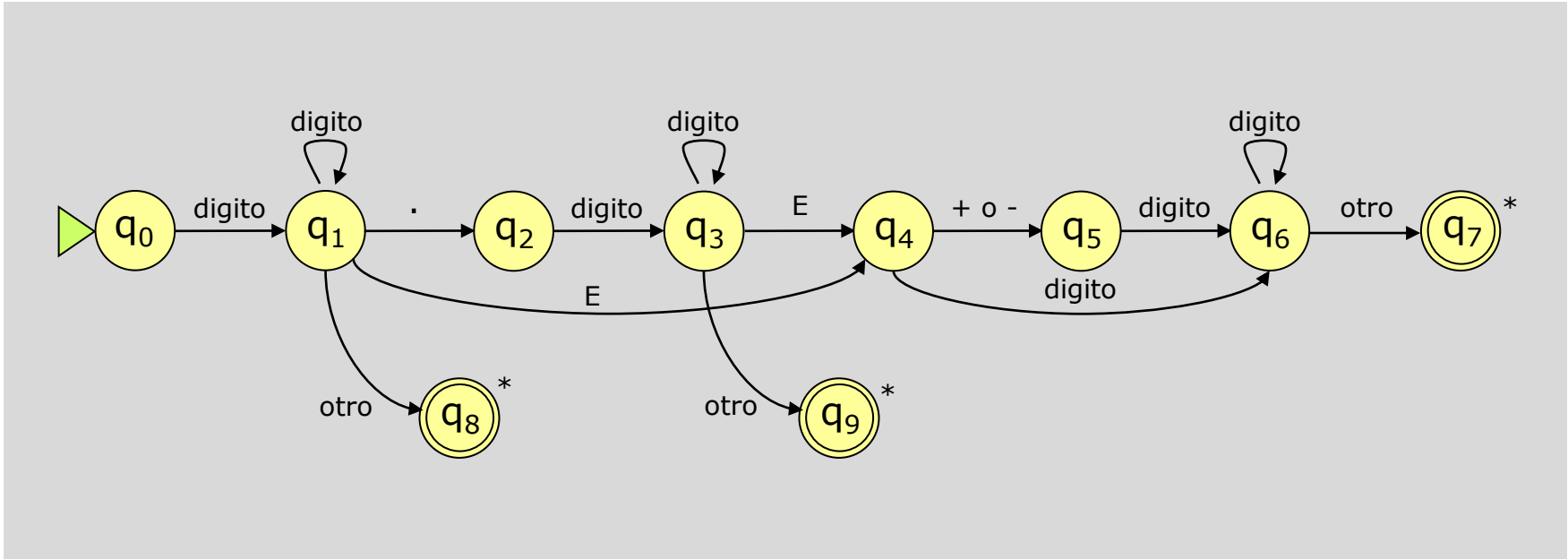
- El analizador léxico se construirá a partir de la agregación de AFD organizados según un orden conveniente.

Ejemplo: ¿Cómo aceptamos los lexemas 12, 12.3, 12.3E-4?



3. Autómatas finitos.
Construcción del analizador

- Podemos integrar los tres autómatas anteriores en el siguiente autómata:



Ejercicio 4

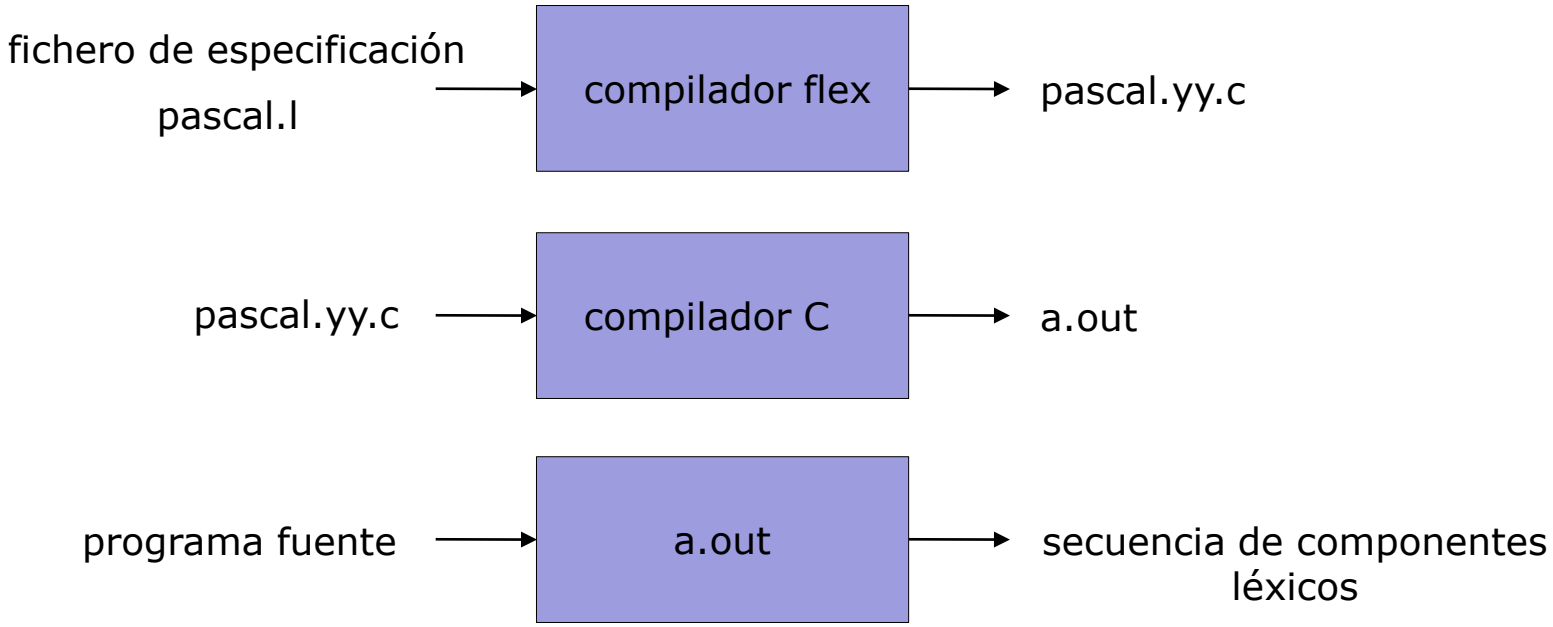
- Sea la expresión regular siguiente:
  a) (a|b)*abb(a|b)*

Se pide:

1. Diseñar un AFN que reconozca el lenguaje correspondiente.
2. Convertir dicho AFN en un AFD equivalente.
3. Mostrar para ambos autómatas la secuencia de cómputo en el procesamiento de la cadena de entrada bbabbab.

4. La herramienta flex

- Existen herramientas que generan analizadores léxicos de forma automática a partir de una especificación basada en el uso de expresiones regulares: por ejemplo, flex.
- El esquema de operación de flex es el siguiente:



4. La herramienta flex
El fichero de especificación

- El fichero de especificación permite describir los componentes léxicos del lenguaje fuente, y las acciones a realizar.
- La compilación de este fichero da lugar a una función `yylex()` que invoca el analizador sintáctico para leer componentes léxicos.
- Este fichero es de la siguiente forma:

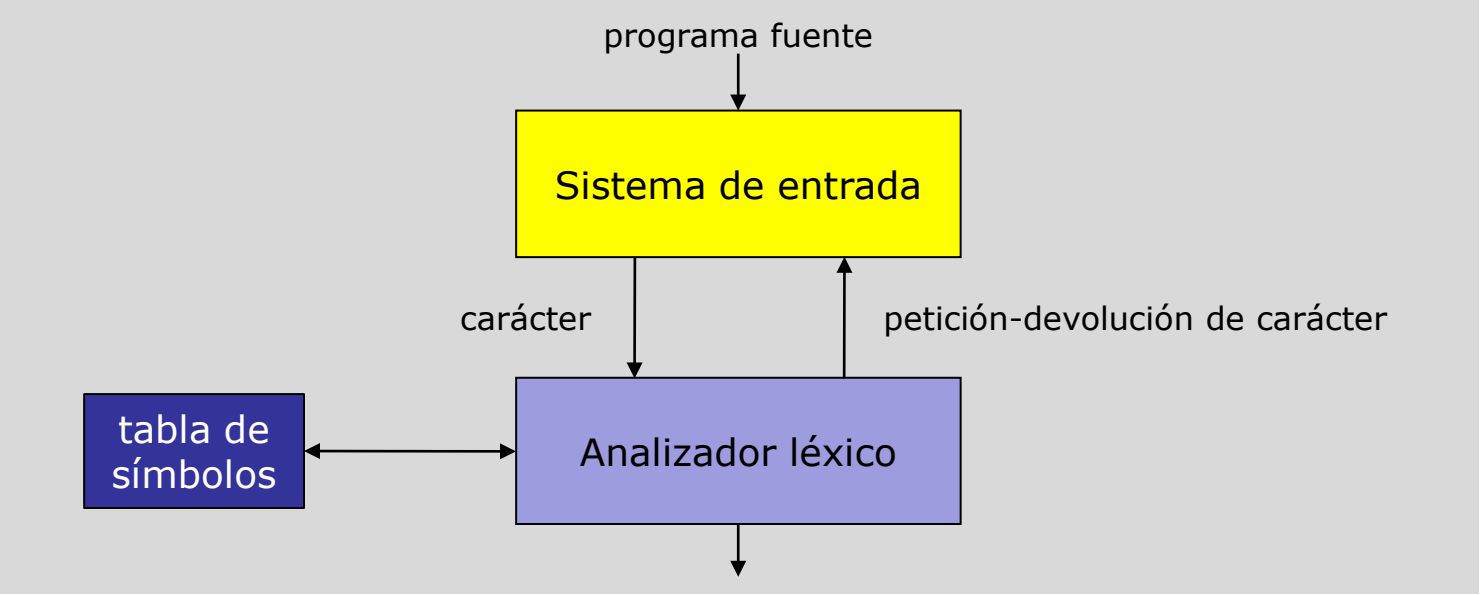
```
sección de definiciones

%%

sección de reglas de traducción

%%

sección de rutinas auxiliares
```

4. La herramienta flex
El fichero de especificación

```
delim           [ \t\n]
espacio         {delim}+
letra           {A-Za-z]
digito          [0-9]
id              {letra}({letra}|{digito})*

%%

{eb}
if              {return(IF);}
then            {return(THEN);}
else            {return(ELSE);}
{id}            {yylval = inserta_tabla(); return(ID);}

%%

inserta_tabla() {/* instala el componente léxico en la tabla de
   símbolos, y devuelve un apuntador a él */}
```

5. El sistema de entrada

- El sistema de entrada es un conjunto de rutinas que interactúan con el sistema operativo para la lectura de datos del programa fuente.
- El sistema de entrada y el analizador léxico funcionan según el patrón productor-consumidor, siguiendo el siguiente esquema:



5. El sistema de entrada

- La separación del sistema de entrada supone una mejora en:
  1. Eficiencia: supongamos un analizador léxico en C que tiene que acceder a un carácter de un fichero. Este carácter pasa 1) del disco a la memoria gestionada por el sistema operativo, 2) a una estructura `FILE`, 3) a una variable `string` del analizador. Urge soluciones.
  2. Portabilidad, ya que el sistema de entrada es el único componente del compilador que se comunica con el sistema operativo. Para cambiar de plataforma sólo tenemos que cambiar el sistema de entrada.

5. El sistema de entrada.
La memoria intermedia

- El analizador léxico debe detectar el componente léxico con el lexema más largo posible. Pensemos en los ejemplos “>” y “>=“, o “=“ y “==“. Necesitamos leer caracteres de un modo anticipado.
- Para resolver este problema se debe incorporar una memoria intermedia, de modo que:
  1. Se pueda almacenar un bloque de caracteres de disco y apuntar el fragmento ya analizado.
  2. En caso de devolución de caracteres al flujo de entrada se pueda mover un apuntador tantas posiciones como caracteres a devolver.

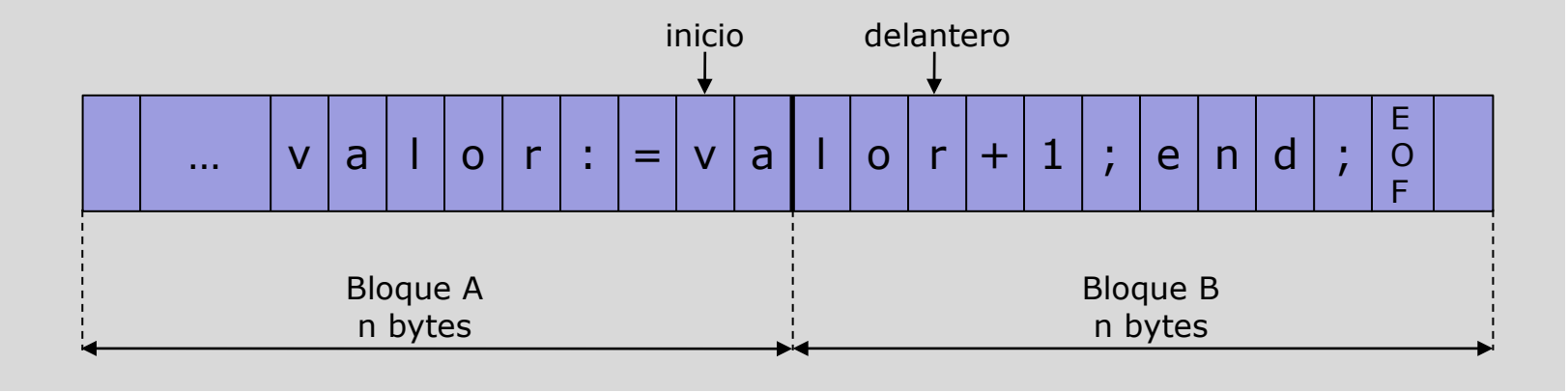
5. El sistema de entrada.
Métodos de gestión de la entrada

- Cualquier sistema de entrada debe satisfacer:
  1. Ser lo más rápido posible.
  2. Permitir un acceso eficiente a disco.
  3. Hacer un uso eficiente de la memoria.
  4. Soportar lexemas de longitud considerable.
  5. Tener disponibles el lexema actual y el anterior.

- Estudiaremos dos métodos de gestión del sistema de entrada:
  1. Método del par de memorias intermedias (*buffer*).
  2. Método del centinela.

5. El sistema de entrada.
Método del par de memorias intermedias

- Este método divide la memoria intermedia en dos mitades de n bytes cada una. n debe ser múltiplo de la longitud de la unidad de asignación.



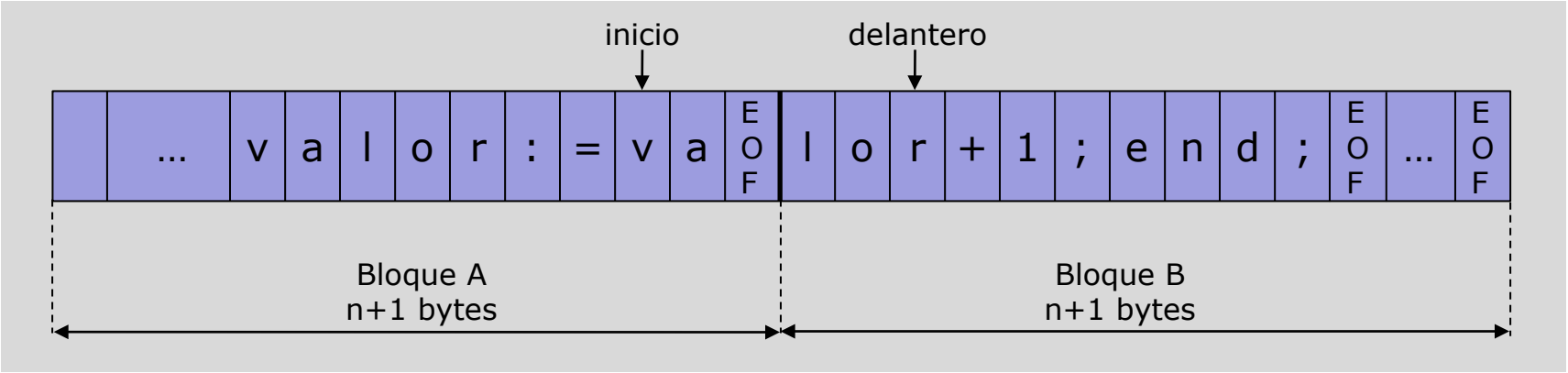
- Al principio, los apuntadores inicio y delantero apuntan al primer carácter de un lexema. A medida que el analizador pide caracteres delantero se mueve hacia adelante. Tras reconocer un componente léxico, inicio avanza hasta la posición de delantero y se inicia el análisis del siguiente lexema.

5. El sistema de entrada.
Método del par de memorias intermedias

- Cada vez que delantero se mueve:
  1. Se comprueba si se ha alcanzado el final del fichero (EOF).
  2. Se comprueba si se ha alcanzado el final de un bloque. Funciona como una lista circular: si se sobrepasa el bloque A se lee B, si se sobrepasa B se lee A. En esas operaciones se solicita al sistema operativo la carga del otro bloque.
- Este método tiene dos limitaciones:
  1. El tamaño del lexema está limitado por n. No es un problema importante.
  2. Es poco eficiente. Cada vez que delantero avanza hay que evaluar tres expresiones lógicas: fin de fichero, fin de bloque A y fin de bloque B.

5. El sistema de entrada.
Método del centinela

- Este método mejora al anterior añadiendo un byte más a cada bloque, en el que se guardará un carácter centinela (EOF).
- De este modo se hace una sola comprobación lógica cada vez que avanza delantero, la de EOF. Si la comprobación es positiva, se analiza cuál de los tres casos es.



6. La tabla de símbolos

- La tabla de símbolos es la estructura de datos utilizada por el compilador para gestionar los identificadores que aparecen en el programa fuente: constantes, variables, tipos, funciones,...
- Cuando el compilador encuentra un identificador, guarda en esta tabla la información que lo caracteriza: nombre, categoría (subrutina, variable, constante, clase, tipo,…), la dirección de memoria que se le asigna, su tamaño, etc.
- Cuando el identificador es referenciado en el programa, el compilador consulta la tabla de símbolos y obtiene la información que necesita.
- Una vez fuera del ámbito del identificador, se elimina de la tabla de símbolos.

6. La tabla de símbolos

- ¿Cómo se utiliza durante la compilación?
  - El analizador léxico, cuando encuentra un identificador, comprueba que está en la tabla de símbolos. Si no lo está, crea una nueva entrada para el mismo.
  - El analizador sintáctico añade información a los campos de los atributos. Pero también puede crear nuevas entradas si se definen nuevos tipos de datos como palabras reservadas.
  - El análisis semántico debe acceder a la tabla para consultar los tipos de datos de los símbolos.
  - El generador de código puede: 1) leer el tipo de dato de una variable para la reserva de espacio; 2) guardar la dirección de memoria en la que se almacenará una variable.

6. La tabla de símbolos.
Palabras reservadas

- Algunos lenguajes reservan algunas palabras que no pueden utilizarse como identificadores: `printf`, `for`, `if`, `while`, etc.
- ¿Cómo las distinguimos?
  1. Definirlas mediante expresiones regulares, y asociarles un componente léxico particular

     `[Ww][Hh][Ii][Ll][Ee]` `return(_WHILE)`
  2. Mediante una tabla de palabras clave. Cada vez que se encuentra un lexema correspondiente a un identificador se busca en esta tabla.
  3. Insertándolas al principio de la tabla de símbolos. Se gestionan en la misma tabla palabras clave e identificadores.

6. La tabla de símbolos

- La tabla de símbolos se inicializa en las primeras posiciones con las palabras reservadas del lenguaje, ordenadas por orden alfabético.
- Cuando se encuentra un nombre en el código fuente se consulta la tabla de símbolos:
  - Si se encuentra entre las palabras reservadas se devuelve su componente léxico al analizador sintáctico.
  - Si se encuentra después de las palabras reservadas es un identificador previamente encontrado.
  - Si no se encuentra en la tabla de símbolos, se añade como un nuevo identificador.
- De este modo se reduce el tamaño del AFD y se aumenta la eficiencia del análisis léxico.

6. La tabla de símbolos.
La estructura de la tabla

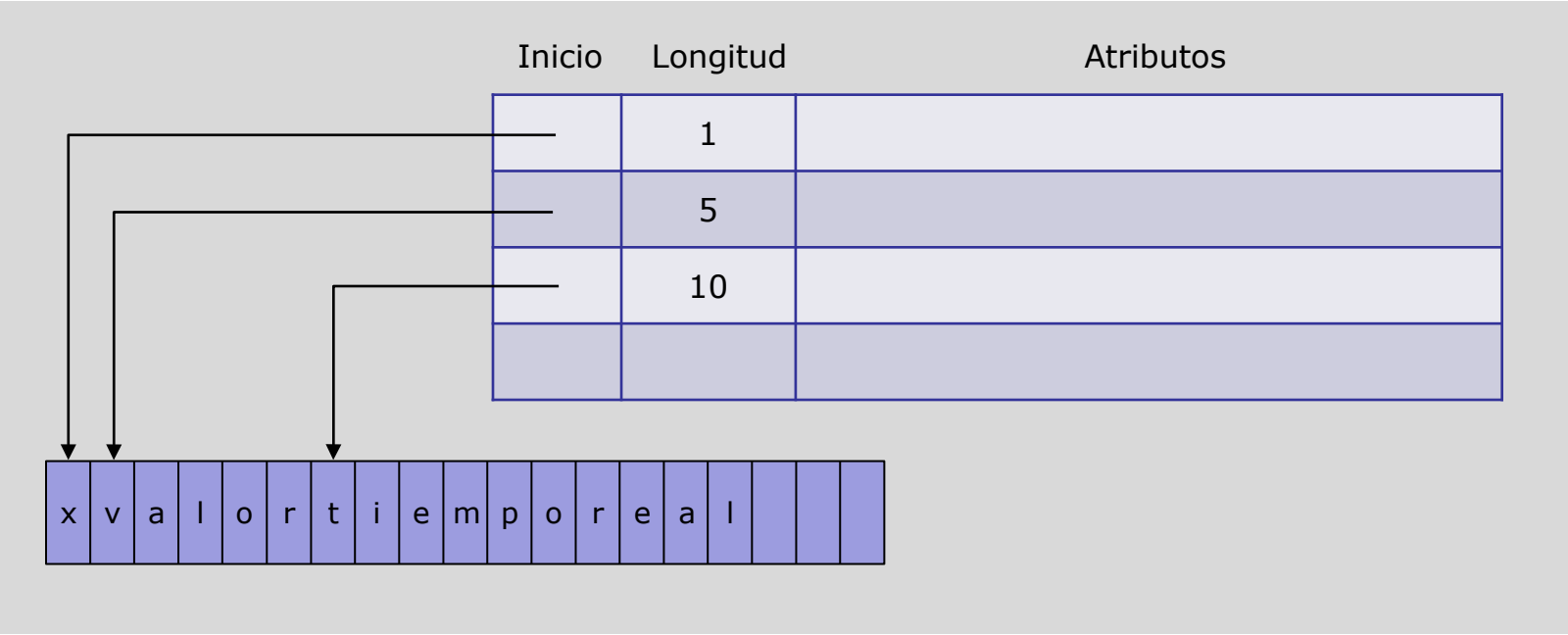
- La tabla de símbolos se estructura en un conjunto de registros, cuya longitud suele ser fija, conteniendo el lexema encontrado y un conjunto de atributos para su componente léxico.
- Hay dos formas características de almacenamiento:
  1. Estructura interna:

Nombres Atributos

Lexema	Dirección de memoria	Línea	Tipo	...

6. La tabla de símbolos.
La estructura de la tabla

2. Estructura externa:



- Esta estructura no exige una longitud fija para los identificadores, lo que permite aprovechar mejor el espacio de almacenamiento.

6. La tabla de símbolos. Operaciones

- La tabla de símbolos funciona como una base de datos en la que el campo clave es el lexema del símbolo.
- Las operaciones que se ejecutan sobre la tabla son:
  - Buscar un lexema y el contenido de sus atributos.
  - Insertar un nuevo registro previa comprobación de su no existencia.
  - Modificar la información contenida en un registro. En general, se realiza una operación de adición de información.
- Además, en lenguajes con estructura de bloque:
  - Nuevo bloque: comienzo de un nuevo bloque.
  - Fin de bloque: final de un bloque.

6. La tabla de símbolos.
Organización de la tabla

- Las tablas de símbolos se organizan de dos maneras:
  - Tablas no ordenadas. Generadas con vectores o listas. Poco eficientes pero fáciles de programar.
  - Tablas ordenadas. Permiten definir el tipo diccionario. Utilizan estructuras de tipo vector o lista ordenada, árboles binarios, árboles equilibrados (AVL), tablas de dispersión (*hash*), etc.
- Analizaremos el uso de algunas de estas estructuras para organizar una tabla de símbolos. Nos ocuparemos de su uso en la traducción de lenguajes con estructura de bloques.
- Las reglas de ámbito, propias de cada lenguaje, determinan la definición de cada lexema en cada momento de la compilación.

6. La tabla de símbolos.
Organización de la tabla

Ejemplo:

Programa	Nivel	Bloque
program uno	1	1
var dos: integer;	2	
procedure tres	2	2
var cuatro: char;	3	
procedure dos	3	3
var cuatro: integer;	4	
procedure cinco	3	4
var uno,dos: real; **(1)**	4	
procedure cinco	2	5
var uno,tres: real;	3	
procedure dos	3	6
var uno,cuatro:char; **(2)**	4	

6. La tabla de símbolos.
Tablas de símbolos no ordenadas

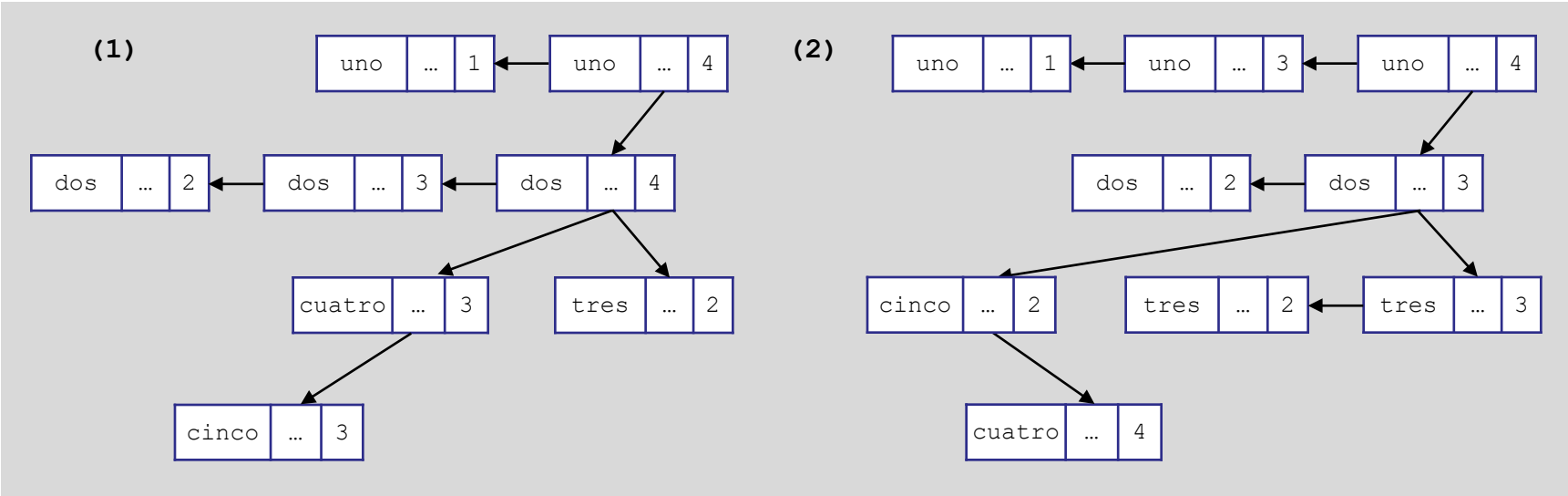
- Se utiliza un vector o una lista. Se añade una pila auxiliar de apuntadores de índice de bloque, para marcar el comienzo de los símbolos que corresponden a un bloque.
- Al terminar un bloque se eliminan todos los símbolos desde el siguiente al apuntado hasta el final de la tabla de símbolos.

6. La tabla de símbolos.
Tablas de símbolos no ordenadas

- Esta organización admite las siguientes operaciones:
  - Inserción. Cuando se encuentra la declaración de un símbolo se debe verificar que no se encuentra en el último bloque. Si no está se inserta en la última posición de la tabla, si no se devuelve un error (símbolo ya definido en ese ámbito).
  - Búsqueda. La búsqueda se hace desde el final de la tabla hacia el principio. Si se encuentra, se corresponde a la declaración realizada en el bloque más próximo.
  - Nuevo bloque. Cuando empieza un nuevo bloque se añade en la pila un apuntador al último símbolo de la tabla.
  - Fin de bloque. Cuando termina un bloque se eliminan todos los símbolos de dicho bloque desde el siguiente al de inicio, apuntado desde la pila. Luego se elimina el índice de la pila.

6. La tabla de símbolos.
TS con estructura de árbol

- Se mejora la eficiencia de la tabla de símbolos mediante un árbol binario ordenado, y se añade un nuevo campo a cada registro que indique el nivel al que pertenece el símbolo.
- Para evitar duplicidades se utilizan listas encadenadas.

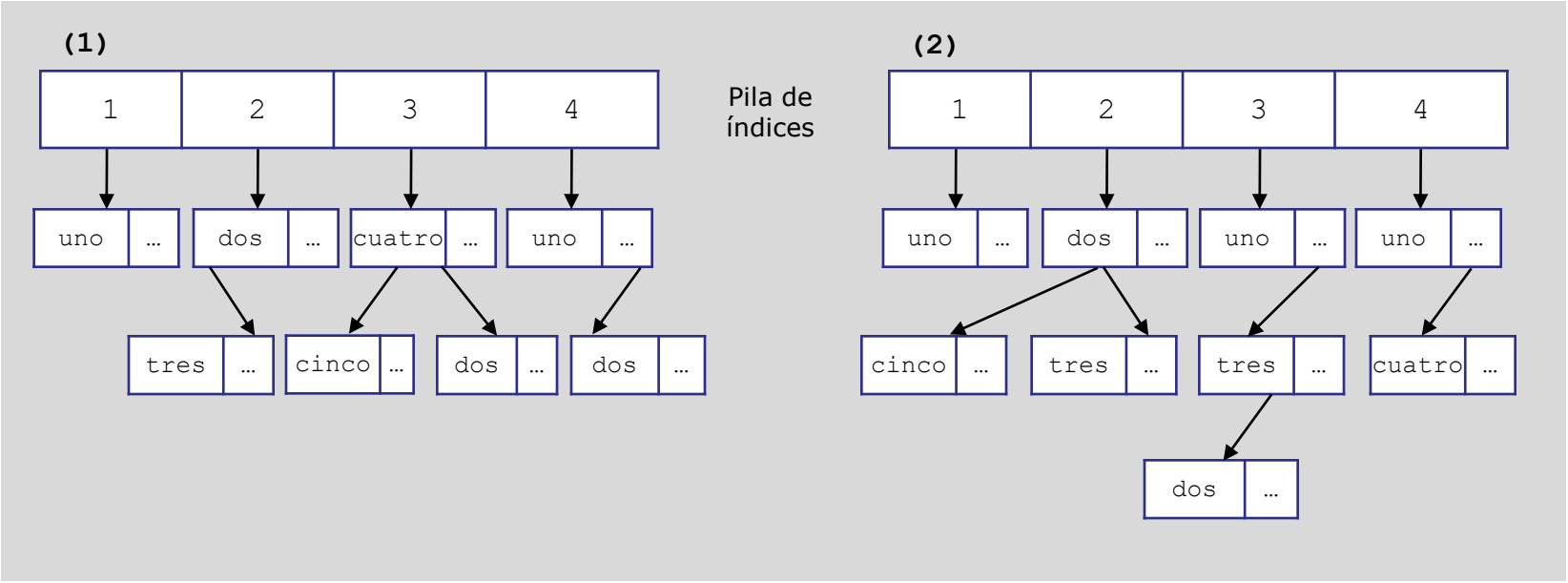


6. La tabla de símbolos.
TS con estructura de árbol

- Esta organización admite las siguientes operaciones:
  - Inserción. Para insertar un nuevo símbolo primero se busca la posición que le corresponde. Si ya hay un registro con el mismo lexema se comprueba que el campo de nivel no tiene el mismo valor que el bloque activo. Si fuese así se indica error. En caso contrario se añade el nuevo nodo a la lista del primero. Si no hay registro se inserta en el árbol.
  - Búsqueda. La propia de un árbol binario.
  - Nuevo bloque. Se incrementa en 1 el campo de nivel.
  - Fin de bloque. Al terminar un bloque se eliminan sus símbolos locales. Para ello: 1) se localizan los registros del árbol del bloque activo; 2) se borran; 3) se decrementa en 1 la variable con el número de nivel.

6. La tabla de símbolos.
TS con estructura de bosque

- Esta técnica utiliza un árbol para cada bloque del programa, y una pila de índices de nivel que apuntan a la raíz del árbol del nivel.



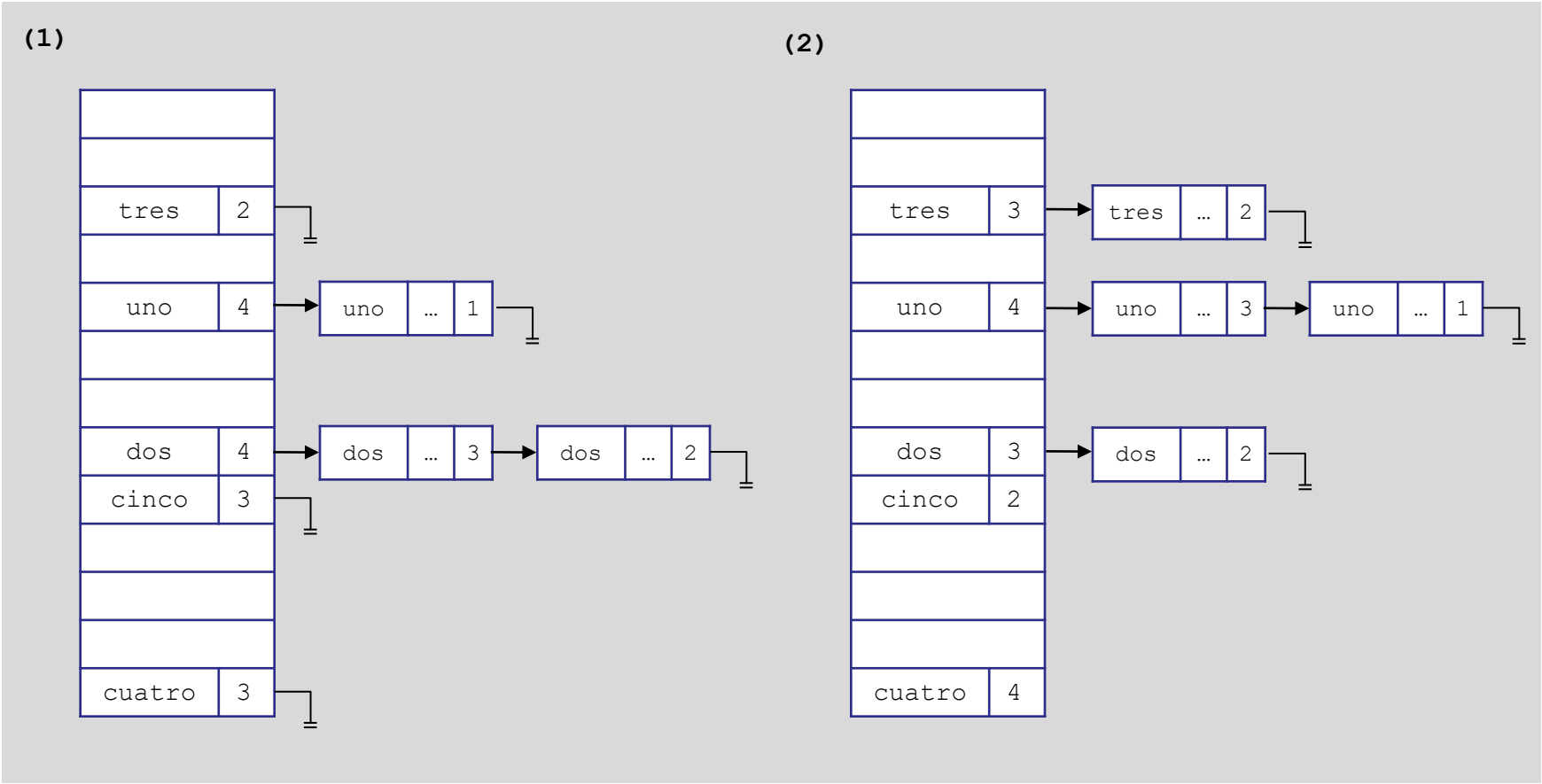
6. La tabla de símbolos.
TS con estructura de bosque

- Esta organización admite las siguientes operaciones:
  - Inserción. Consiste en hacer una inserción normal en el árbol del bloque activo.
  - Búsqueda. Primero se busca el lexema en el árbol del bloque activo. Si no se encuentra se busca en el árbol del bloque anterior, y así sucesivamente.
  - Nuevo bloque. Cuando empieza la compilación de un nuevo nivel, se crea un nuevo elemento en la pila de índices y una nueva estructura de tipo árbol.
  - Fin de bloque. Cuando se termina de compilar un bloque se destruye el árbol asociado y se elimina el último puntero de la pila de índices.

6. La tabla de símbolos.
Tablas de dispersión (*hash*)

- Una tabla de dispersión aplica una función matemática al lexema para determinar la posición de la tabla que le corresponde.
- El inconveniente de las tablas de dispersión son las colisiones que se producen cuando dos lexemas quieren ocupar la misma posición. Existen dos soluciones:
  - Tablas de dispersión cerradas. Cuando se produce una colisión se utiliza alguna técnica que permita acceder a una posición vacía (sondeo lineal, multiplicativo, cuadrático,...)
  - Tablas de dispersión abiertas. Se utiliza una lista encadenada de desbordamiento para resolver las colisiones. Cuando se produce una colisión se crea una lista encadenada con la cabeza en el registro de la tabla.

6. La tabla de símbolos.
Tablas de dispersión (*hash*)



6. La tabla de símbolos.
Tablas de dispersión (*hash*)

- Esta organización admite las siguientes operaciones:
  - Inserción. Como en cualquier tabla de dispersión. Se suelen utilizar listas de desbordamiento, colocando los símbolos más recientes en las primeras posiciones.
  - Búsqueda. Como en cualquier tabla de dispersión. En caso de colisión se debe buscar en la zona de desbordamiento, hasta encontrar el símbolo o una posición vacía.
  - Nuevo bloque. Sólo hay que almacenar el cambio de bloque activo.
  - Fin de bloque. Al terminar la compilación de un bloque se borrarán todos los registros correspondientes a este bloque. Para ello se recorre prácticamente toda la tabla. Después se cambia de bloque activo.

6. La tabla de símbolos.
Análisis de complejidad

- Comparamos las distintas formas de organización de una tabla de símbolos en cuanto a complejidad temporal:

	Búsqueda	Inserción
Lista no ordenada	O(n)	O(n)
Lista ordenada	O(log n)	O(n)
Árbol AVL	O(log n)	O(log n)
Tabla de dispersión	O(1)	O(1)

Ejercicio 5

- Sea el programa siguiente:

```
program uno
   var dos,tres: integer;
   procedure cinco
      var uno,dos:real;
      procedure seis
         var dos,tres:char;
      procedure cuatro
         var uno,dos:real;    (1)
   procedure cuatro
      var uno,dos:char;       (2)
```

Se pide mostrar en los puntos (1) y (2) el contenido de una tabla de símbolos construida según los métodos aquí explicados.

7. Tratamiento de errores

- Cuando en un momento del proceso de compilación se detecta un error:
  - Se debe mostrar un mensaje claro y exacto, que permita al programador encontrar y corregir fácilmente dicho error.
  - Debe recuperarse del error e ir a un estado que permita continuar analizando el programa en búsqueda de otros errores. Se debe evitar una cascada de errores, o pasar por alto otros.
  - No debe retrasar excesivamente el procesamiento de programas correctos.

7. Tratamiento de errores

- Errores más característicos de la fase de análisis léxico son:

Tipo de error	Explicación y subtipos	Ejemplos
Nombres ilegales de identificadores	Nombre de identificador que contiene caracteres inválidos	nen@12’
Números inválidos	Contiene caracteres inválidos Está formado incorrectamente Es demasiado grande y produce desbordamiento	2:13 3.14.58 9999999999999
Cadenas de caracteres incorrectas	Cadena demasiado larga, probablemente porque falta cerrar unas comillas	
Errores de ortografía en palabras reservadas	Caracteres omitidos Caracteres adicionales Caracteres incorrectos Caracteres mezclados	wile whhile whule wihle
Fin de fichero	Se detecta un fin de fichero durante el análisis de un componente léxico	

7. Tratamiento de errores

- El proceso de recuperación de un error puede suponer la adopción de diferentes medidas:
  - Ignorar los caracteres inválidos hasta formar un componente léxico correcto.
  - Eliminar caracteres que dan lugar a error.
  - Intentar corregir el error:
    - Insertar los caracteres que pueden faltar.
    - Reemplazar un carácter presuntamente incorrecto por uno correcto.
    - Intercambiar caracteres adyacentes.
- Cuidado, intentar corregir un error puede ser peligroso.

Bibliografía

- A.V. Aho, R. Sethi, J.D. Ullman. Compiladores. Principios, tecnicas y herramientas. 1a edición. Addison Wesley, 1990.
- M. Alfonseca, M. de la Cruz, A. Ortega, E. Pulido. Compiladores e intérpretes: teoría y práctica. Pearson Educación, 2006.