

Procesadores de lenguaje

→ Tema 2 – Análisis léxico



Salvador Sánchez, Daniel Rodríguez
Departamento de Ciencias de la Computación
Universidad de Alcalá de Henares

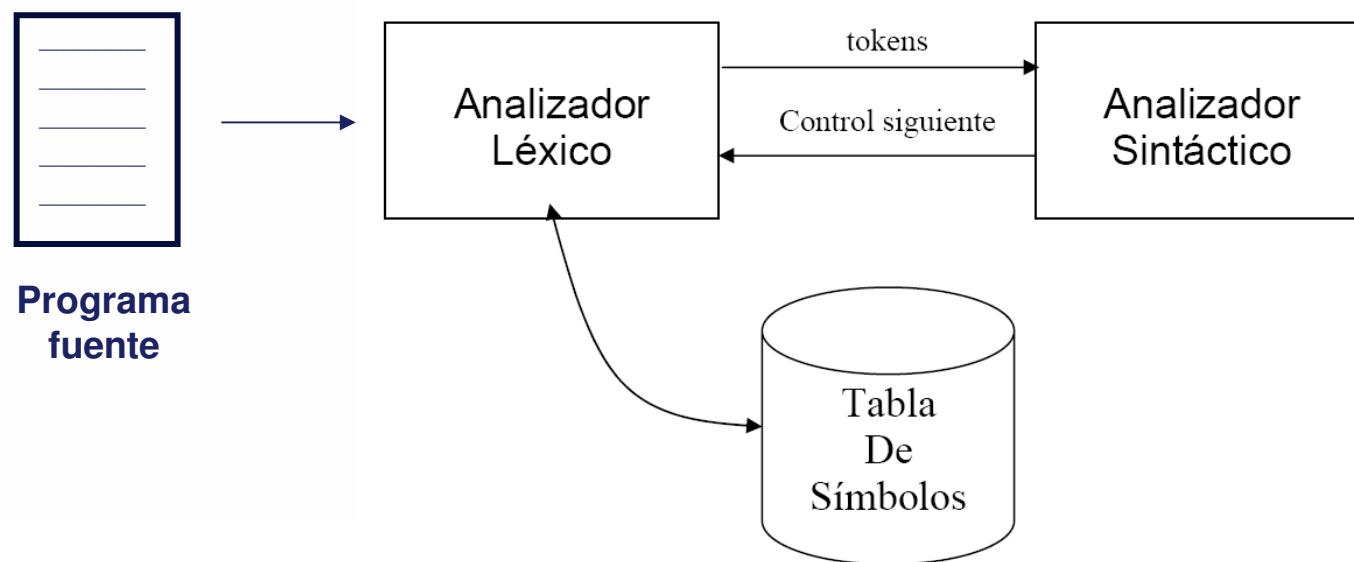
→ Resumen del tema

- Objetivo: comprender la estructura, organización y funcionamiento general del analizador léxico.
- Contenidos:
 - Formalismos de especificación léxica:
 - Expresiones regulares
 - Autómatas finitos
 - Tratamiento de errores léxicos y palabras reservadas
 - Construcción de un analizador léxico
 - Generadores automáticos de analizadores léxicos



→ Descripción de un analizador léxico

- Un analizador léxico lee carácter a carácter del programa fuente y genera una secuencia de componentes léxicos (*tokens*) que corresponden a unos patrones a los que asocia, si es necesario, unos atributos.



→ Tareas – Analizador Léxico

- Reconocer los componentes léxicos en la entrada.
- Eliminar los espacios en blanco, los caracteres de tabulación, los saltos de línea y de página y otros caracteres propios del dispositivo de entrada.
- Eliminar los comentarios de entrada.
- Detectar errores léxicos.
- En lenguajes de programación, además:
 - Reconocer los identificadores de variable, tipo, constantes, etc. y guardarlos en la tabla de símbolos.
 - Relacionar los mensajes de error del compilador con el lugar en el que aparecen en el programa fuente (línea, columna, etc.).



→ Necesidad de un analizador léxico

- Al comparar las expresiones

$$a\emptyset := \emptyset b \emptyset * \emptyset 7 \emptyset + \emptyset 4 \emptyset ;$$
$$a \emptyset \emptyset \emptyset := \emptyset \emptyset b * 7 + 4 \emptyset \emptyset \emptyset ;$$

- La estructura de las dos expresiones es equivalente,
- La posición de los caracteres que las componen, aunque siguen el mismo orden, son diferentes.
- Si las distintas fases del compilador tuvieran que trabajar con los caracteres directamente sería más complicado descubrir la estructura de un programa.



→ Independencia de léxico y sintáctico

- El analizador léxico suele convertirse en una **subrutina** del analizador sintáctico.
- Varias razones para su independencia:
 - Se simplifica el diseño del analizador sintáctico.
 - Se consigue un compilador más eficiente
 - Un sistema de entrada optimizado aumenta la velocidad de lectura.
 - Añade portabilidad al compilador: independencia del alfabeto.
- Se comunican mediante:
 - Buffer intermedio.
 - Llamada a subrutina.



→ Conceptos básicos

- **Token (o componente léxico)**: Secuencia de caracteres con significado sintáctico propio.
- **Lexema**: Secuencia de caracteres cuya estructura se corresponde con el patrón de un token.
- **Patrón**: Regla que describe los lexemas correspondientes a un token.

COMPONENTE LÉXICO	LEXEMAS EJEMPLO	DESCRIPCIÓN DEL PATRÓN	OBSERVACIONES
IDENTIFICADOR	x, y, valor, x2	<letra> (<letra> <dígito>)*	Identificadores
CADENA	"Una cadena"	caracteres entre " y "	Constante de cadena



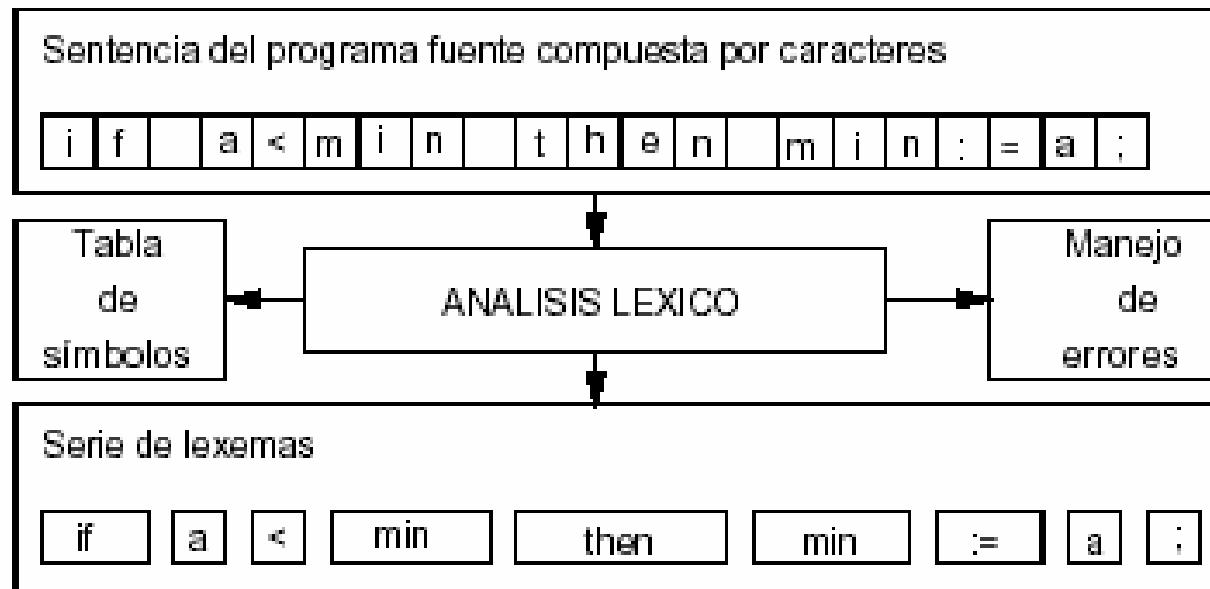
→ Componentes léxicos

- En la mayoría de los lenguajes de programación, se consideran componentes léxicos (tokens):
 - palabras reservadas
 - operadores (de comparación, asignación, lógicos, aritméticos ...)
 - identificadores
 - constantes
 - signos de puntuación (paréntesis, punto y coma ...)
 - marcas de comienzo y fin de bloque
- Los delimitadores no serán considerados, en general, tokens.
- Cuando un patrón puede concordar con más de un lexema es necesario conocer información adicional:
 - Esta información se almacena como atributos del token.



→ Descripción de un analizador léxico

- El análisis léxico es un análisis de los caracteres
 - Parte de éstos y por medio de patrones reconoce los lexemas
 - Envía al analizador sintáctico el componente léxico y sus atributos
 - Puede hacer tareas adicionales: eliminar blancos, control líneas ...



→ El analizador léxico como interfaz

- El analizador léxico y el sintáctico forman un par *productor-consumidor*.
- En algunas situaciones, el analizador léxico tiene que leer algunos caracteres por adelantado para decidir de qué *token* se trata.



→ Ejemplo

- Para la sentencia en Pascal:

```
IF x < 10 THEN x := x + y
```

- El analizador lexicográfico daría como resultado inicialmente la siguiente cadena de caracteres:

```
<79> <12> <28> <13> <80> <12> <65> <12> <34> <12>
```

- O lo que es lo mismo:

```
if identificador op_menor literal_entero then identificador  
asignación identificador op_suma identificador
```



→ Ejemplo

- El resultado final sería:

```
<79,-> <12,32> <28,-> <13,10> <80,-> <12,32>
       <65,-> <12,32> <34,-> <12,33>
```

- Es decir:

```
<if,-> <identificador,32> <op_menor,->
<literal_entero,10> <then,-> <identificador,32>
<asignación,-> <identificador,32> <op_suma,->
                           <identificador,33>
```



→ Atributos

- Los identificadores tienen asociados como atributos el lugar de la tabla de símbolos donde se han guardado:

< **identificador**, 32 >

- A veces el analizador sintáctico es el único que maneja la tabla de símbolos, en ese caso llevan asociado el propio lexema:

< **identificador**, x >

- Los literales tienen asociados como atributos el propio lexema:

< **literal_entero**, 10 >



→ Errores léxicos

- El analizador léxico rechaza texto con caracteres ilegales (no recogidos en el alfabeto) o combinaciones ilegales. Ejemplos:
 - “ñ”, “é” (caracteres que no pertenecen al alfabeto del lenguaje)
 - “:=”, “::” (no coinciden con ningún patrón de los *tokens* posibles)
- Se debe mostrar un mensaje de error claro y exacto.
 - En vez de...
 - Error 124
 - Falta declaración
 - Error en la línea 85
 - Se ha producido un error
 - Sería mejor...
 - int número = 1 ;
 ^

ERROR 124: línea 85, columna 6, carácter no válido



→ Recuperación de Errores léxicos

- Posibles acciones del analizador léxico para detectar errores, recuperarse y seguir trabajando:
 - *Modo de pánico*: Ignorar los caracteres no válidos hasta un carácter en el cual se considera que podría empezar un lexema correcto.
 - Distancia mínima de corrección de un error, i.e., recuperación “inteligente” con correcciones como:
 - Borrar los caracteres extraños.
 - Insertar un carácter que pudiera faltar.
 - Reemplazar un carácter incorrecto por uno correcto.
 - Comutar las posiciones de dos caracteres adyacentes.



→ Especificación de componentes léxicos

- **Alfabeto:** Conjunto finito de símbolos.
- **Cadena** sobre un alfabeto: secuencia finita de símbolos de ese alfabeto.
 - Cadena vacía: ϵ
 - Operaciones con cadenas: concatenación y exponenciación
- **Lenguaje:** conjunto de cadenas sobre un alfabeto.
 - Operaciones con lenguajes: Unión, Concatenación, Cerradura de Kleene (Cierre $*$) y Cerradura positiva (Cierre $+$)



→ Expresiones regulares

- Notación que facilita la especificación de lenguajes.
- ϵ : cadena vacía
- a : cadena “ a ” si “ a ” \in alfabeto.
- Si r y s son cadenas que pertenecen respectivamente a los lenguajes $L(r)$ y $L(s)$, son expresiones regulares:
 - $r|s \equiv L(r) \cup L(s)$
 - $rs \equiv L(r)L(s)$
 - $r^* \equiv (L(r))^*$
- Abreviaturas:
 - 0 ó más $\rightarrow a^*$
 - 1 ó más $\rightarrow aa^* \equiv a^+$
 - 0 ó 1 $\rightarrow \epsilon | a \equiv a?$
 - clases de caracteres $\rightarrow a | b | c \equiv [abc]$. También: $a | b | c | \dots | z \equiv [a-z]$
 - Los paréntesis agrupan subexpresiones: $(a|b|c)^*$



→ Ejemplos

$0(0|1)0^*$: 010, 000, 010000, 01, etc.

$a(ab)^+b^*$: aab, aababb, aabbbbb, aababababbb, etc.

$[1-9]^?0$: 0, 10, 20, 30, . . . , 90

[a-zA-Z]

[0-9] +

[a-zA-Z] ([a-zA-Z] | [0-9]) *

[0-9] + (. [0-9] +) ?



→ Definiciones regulares

- Nombre que se da a una expresión regular por conveniencia, definiéndola como si fuera un símbolo:

Letra → [a-zA-Z]

Dígito → 0|1|2|3|4|5|6|7|8|9

Identificador → Letra (Letra|Digito)*

NúmeroReal → Digito⁺(.Digito⁺) ?



→ Definiciones regulares

Dígito → [0-9]

Dígitos → Dígito⁺

Fracción → (.Dígitos) ?

Exponente → (E (+|-) ?Dígitos) ?

NúmeroReal → Dígitos Fracción Exponente



→ Diagramas de transiciones

- Compuestos por estados y transiciones entre éstos
 - Círculos: estados
 - Arcos etiquetados: transiciones
- Sirven para representar lo que sucede a medida que se toman caracteres de la entrada
 - Cada círculo = un estado
 - Cada arista = un carácter en la entrada
 - Doble círculo = aceptación de un elemento léxico
 - Asterisco = el último carácter se debe devolver a la entrada (retroceso)
 - Otro = cualquier carácter no asignado a otra flecha
- No tiene estados de error (sin transición = error)



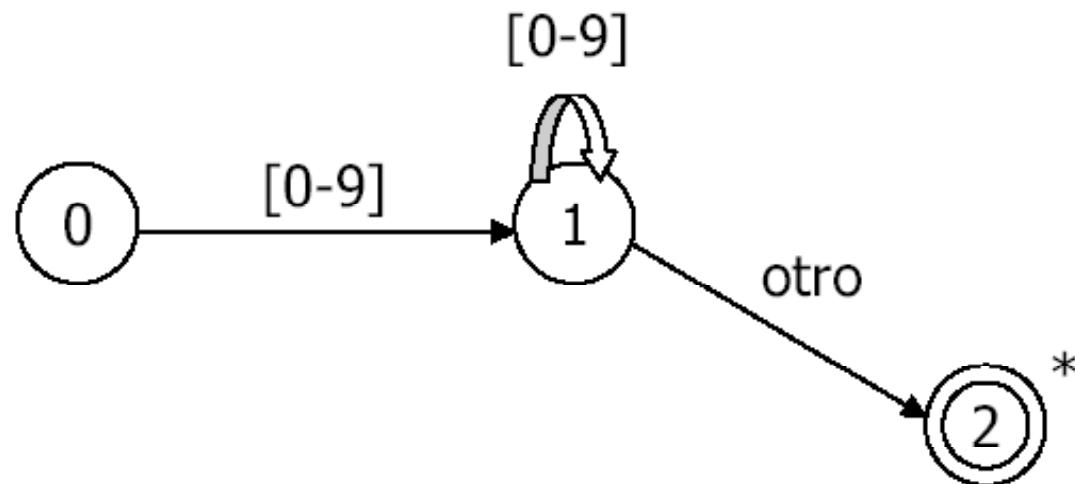
→ Autómatas finitos deterministas (AFD)

- Son diagramas de transiciones generalizados.
- Un reconocedor de un lenguaje es un programa que dice si una cadena pertenece o no a un lenguaje.
- Toda expresión regular puede reconocerse mediante un autómata finito determinista (AFD)
- Se emplean autómatas por la sencillez de programar código que simule su funcionamiento.

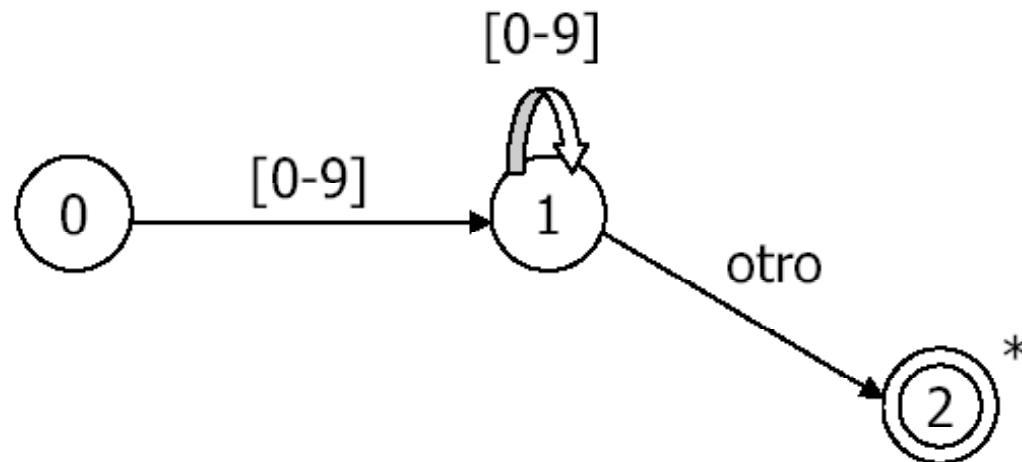


→ Diagramas de transiciones

NúmeroEntero → Digito⁺



→ Diagramas de transiciones



Estado	0-9	otro	Token	Retroceso
0	1	Error	-	-
1	1	2	-	-
2	-	-	Num_entero	1



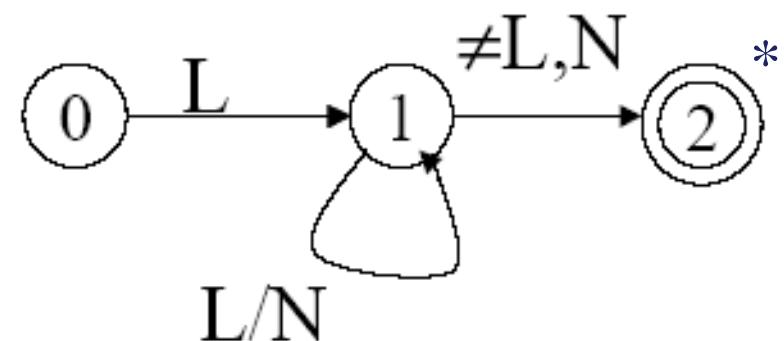
→ Diagramas de transiciones

Identificador → Letra(Letra|Digito)*

$L \rightarrow [a-zA-Z]$

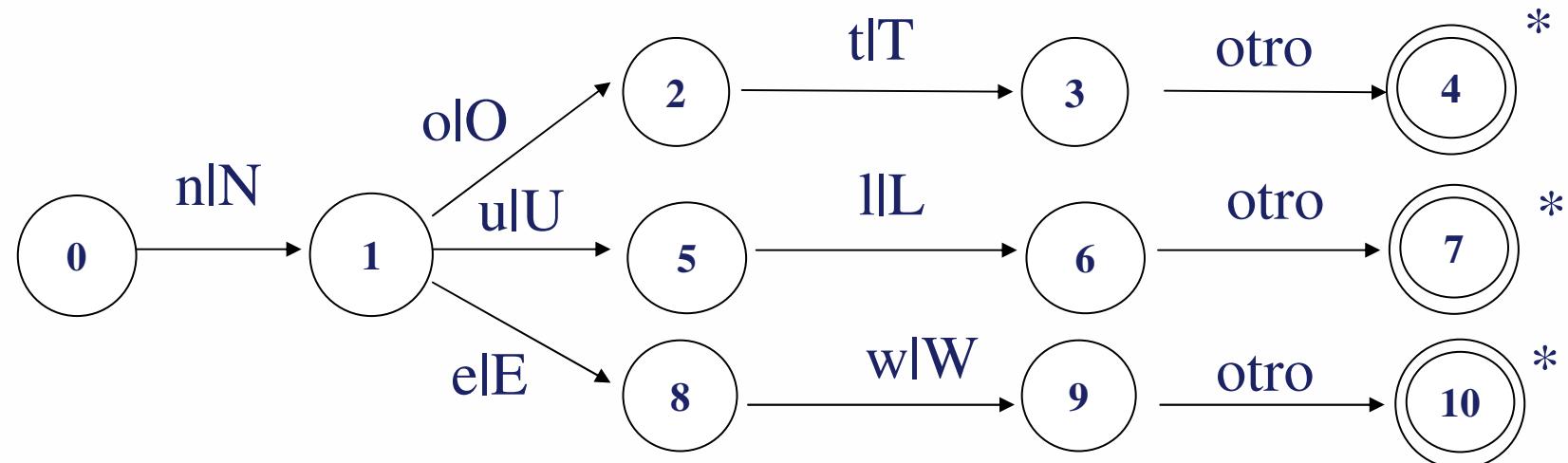
$N \rightarrow [0-9]$

$Id \rightarrow L (L \mid N)^*$



→ Diagramas de transiciones

Not–Nul–New (Pascal)



- 4 – Reconocer “not”
- 7 – Reconocer “nul”
- 10 – Reconocer “new”



→ Prioridad de los tokens

- Se da prioridad al *token* con el lexema más largo:
 - Si se lee “`>=`” y “`>`” se reconoce el primero.
- Si el mismo lexema se puede asociar a dos *tokens*, estos patrones estarán definidos en un orden determinado.
- Ejemplo:
 - `while` → palabra reservada “`while`”
 - `letra (letra | digito)*` → identificador
 - Si en la entrada aparece “`while`”, se elegirá la palabra reservada por estar primero.
 - Si estas especificaciones iniciales aparecieran en orden inverso, se reconocería un *token* identificador



→ Tratamiento de palabras reservadas

- Son aquellas que los lenguajes de programación “reservan” para usos particulares.
- ¿Cómo diferenciarlas de los identificadores?
 - Resolución **implícita**: reconocerlas todas como identificadores, utilizando una tabla adicional con las palabras reservadas que se consulta para ver si el lexema reconocido es un identificador o una palabra reservada.
 - Resolución **explícita**: se indican todas las expresiones regulares de todas las palabras reservadas y se integran los diagramas de transiciones resultantes de sus especificaciones léxicas en la máquina reconocedora.



→ Tratamiento de palabras reservadas

- Resolución explícita:

[fF][oO][rR] return (TOKEN_FOR)

[wW][hH][iI][lL][eE] return (TOKEN WHILE)

...

[a-zA-Z]([a-zA-Z0-9])* return (TOKEN_IDENTIFIER);

- Inconvenientes:

- Se alarga mucho el código.
- Se hace más difícil de leer.
- El autómata generado tiene muchos más estados.
- La ejecución es más lenta.



→ Tratamiento de palabras reservadas

- Resolución implícita (1): se crea una tabla de palabras clave que se consulta cada vez que se encuentra un identificador para ver si es una palabra clave o no.

```
[a-zA-Z] ([a-zA-Z0-9])*      if is_keyword(yytext, kw) then  
                                return (kw)  
                            else  
                                return (TK_IDENTIFIER);
```

- Las palabras clave se tratan como un caso particular del patrón de los identificadores. Cuando se detecta un identificador:
 - Se comprueba si es una palabra clave.
 - Si no lo es, se trata de un identificador, y hay que comprobar si ya está introducido en la tabla de símbolos.
 - Si no está en la tabla de símbolos, hay que darle de alta y retornar un testigo identificador.



→ Tratamiento de palabras reservadas

- Resolución implícita (2): Se inicializan las primeras posiciones de la tabla de símbolos con las palabras clave.
- Cuando se encuentra un identificador, se consulta la tabla de símbolos:
 - Si se encuentra en la zona inicial reservada, es una palabra clave y se retornará su token.
 - Si se encuentra en la parte final de la tabla, se trata de un identificador que ya se había encontrado anteriormente.
 - Si no se encuentra, hay que añadirla como nuevo identificador.

Lexema	Testigo	
AND	_AND	Zona de palabras reservadas
ARRAY	_ARRAY	
BEGIN	_BEGIN	
CASE	_CASE	
.....		
ValorX	IDENTIFICADOR	Zona de identificadores
Test_B	IDENTIFICADOR	
.....		



→ Construcción de un analizador léxico

- Los analizadores léxicos pueden construirse:
 - *Usando generadores de analizadores léxicos*: Es la forma más sencilla pero el código generado por el analizador léxico es más difícil de mantener y puede resultar menos eficiente.
 - *Escribiendo el analizador léxico en un lenguaje de alto nivel*: permite obtener analizadores léxicos con más esfuerzo que con el método anterior pero más eficientes y sencillos de mantener.
 - *Escribiendo el analizador léxico en un lenguaje ensamblador*: Sólo se utiliza en casos específicos debido a su alto coste y baja portabilidad.



→ Implementación en leng. de alto nivel

- Consiste en implementar el diagrama de transiciones mediante la construcción de su tabla de transiciones.
 - filas: estados del diagrama de transiciones
 - columnas: posibles entradas
- Se puede también implementar directamente, utilizando estructuras de selección múltiple (*switch*) para leer caracteres hasta completar el *token*.
- **Modelo mixto:**
 - selección múltiple para los elementos léxicos de estructura más sencilla
 - diagrama de transiciones para el resto (cadenas no específicas, prefijos comunes, etc.)



→ Implementación en leng. de alto nivel

```
int explorador (void)
{ c = nuevocaracter (); /*función del sistema de entrada */
switch (c)
{ case ' ':
case '\t':
case '\n': break; /*no hacer nada para los separadores */
case '+':
case '-': return (OPERADOR_SUMA);
case '*':
case '/': return (OPERADOR_MULTIPLICAR);
/*.... aquí vendrían el resto de los operadores y
simbolos de puntuación por orden de precedencia */
default:   if (esnumero(c))
            {/*se leen caracteres mientras sean
            números y se retorna al flujo de entrada el último
            carácter leido, se guarda el lexema leído
            y se retorna con el testigo NÚMERO_ENTERO */
            return(NÚMERO_ENTERO);
        }
        else if (esletra(c))
            {/*se leen caracteres mientras sean letras o números
            y se retorna al flujo de entrada el último carácter
            leido, se guarda el lexema leído y se comprueba si
            es una palabra clave. Se retorna IDENTIFICADOR o
            la palabra clave*/
            return(testigo);
        }
    } /* del switch */
} /* del explorador */
```



→ Generador automático (LEX)

- Recibe la especificación de las expresiones regulares de los patrones que representan a los *tokens* del lenguaje y las acciones a tomar cuando los detecte.
- Genera los diagramas de transición de estados en código C , C++ o Java generalmente.
- Ventaja: Comodidad de desarrollo.
- Desventajas:
 - El mantenimiento del código generado resulta complicado.
 - La eficiencia del código generado depende del generador.



→ Generador automático (LEX)

- Parte de un conjunto de reglas léxicas (expresiones regulares) y produce un programa que reconoce las cadenas que cumplen dichas reglas
 - *JLex produce un programa Java que tras compilarse da como resultado la clase Yylex: implementación del AFD*
- A cada regla se asocian un conjunto de acciones.
 - Cuando *Yylex* encuentra una cadena que cumple un regla ejecuta las acciones asociadas a esa regla.



→ Sección de reglas léxicas

- La 3^a sección del fichero de especificación contiene las reglas que extraen tokens del fichero de entrada. Cada regla consta de:

- 1.Lista opcional de estados
- 2.Expresión regular
- 3.Acción léxica

[Lista_de_Estados] [Expresión_Regular] [Acción_Léxica]

- Ej:

```
[a-zA-Z] [a-zA-Z_0-9]* {System.out.println  
("Identificador");}
```



→ Acciones Léxicas

- Las acciones léxicas consiste en código Java entre llaves:
`{ Código_Java }`
- Valores léxicos:
 - `yytext()`: Función que nos retorna el lexema de entrada identificado.
 - `yychar`: Entero que contiene la posición, dentro el fichero de entrada, del primer carácter del lexema actual.
 - `yyline`: Entero con el número de línea donde se encuentra
 - Ej.: `[a-zA-Z] [a-zA-Z_0-9]*`
`{System.out.print ("Identificador:"+ yytext());}`



→ Estados

- Es posible encontrar varios componentes con diferentes significados en función del lugar donde aparezcan dentro del fichero de entrada.
Ej., en C: `int var1; /* var1 */`

```
[<ESTADO1 [,ESTADOn . . . ]>] regla_con_estado { acción }
```

- El comportamiento del analizador léxico será el siguiente:
 - Si el analizador se encuentra en el `ESTADO`, la regla `regla_con_estado` estará activada, i.e., se comparará su correspondencia con los lexemas de entrada.
 - Si el analizador no se encuentra en ninguno de los estados `<ESTADOi>`, la regla `regla_con_estado` no estará activada, i.e., no se comparará su correspondencia con los lexemas de entrada.
 - Si una regla no contiene ningún estado en su lista previa de estados, la regla estará siempre activa.



→ Estados - Ejemplo

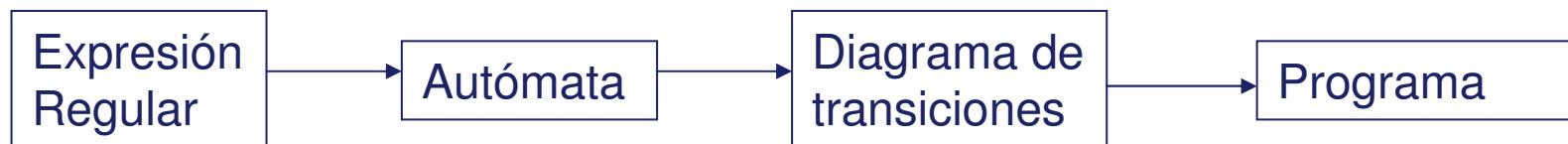
```
<YYINITIAL>"/*"      { yybegin (CCOMMENT); }
<CCOMMENT>[^*]*"/" { System.out.println ("Comentario: <" +
                      yytext ().substring(0,yytext ().length ()-
2)+"> );
                      yybegin (YYINITIAL); }
<YYINITIAL>[^/]*     { }
```

- Este analizador detecta comentarios del estilo /* ... */



→ El proceso de implementación

- Definir las expresiones regulares del analizador léxico
- Identificar los tokens (códigos y atributos)
- Construir los diagramas de transiciones (AFD)
- Completar los autómatas con acciones semánticas
- Definir todos los posibles errores
- Implementar el AFD y las acciones semánticas usando switch o tablas de transiciones



→ Bibliografía

- **Básica:**
 - *Compiladores: principios, técnicas y herramientas.* A.V. Aho, R. Sethi, J.D. Ullman. Addison-Wesley Iberoamerica. 1990.
- **Complementaria:**
 - *Construcción de compiladores: Principios y práctica.* K. C. Louden. Thomson-Paraninfo. 2004.

