Tema 2: Análisis léxico

Procesamiento de Lenguajes

Dept. de Lenguajes y Sistemas Informáticos
Universidad de Alicante

Fundamentos del análisis léxico

- El analizador léxico se encarga de suministrar al analizador sintáctico una serie de unidades lógicas llamadas *elementos léxicos* que resultan de agrupar los caracteres del fichero de entrada
- Cada uno de estos elementos léxicos se denomina token
- El analizador léxico es una *función o método* que es llamado por el analizador sintáctico cada vez que necesita conocer un nuevo token para continuar el proceso de traducción
- Es el único módulo del compilador que maneja el fichero de entrada

Los tokens

- Clases de tokens:
  - palabras reservadas (`if`, `then`,...)
  - símbolos especiales (operadores aritméticos, lógicos,...)
  - cadenas no específicas (identificador, número real,...)
  - `EOF` (fin de fichero)
- La cadena de caracteres concreta que se ha reconocido como un token determinado se denomina lexema
- El lexema no juega un papel desde el punto de vista estructural, pero sí desde el semántico
- A la información auxiliar que acompaña a un token se le llama atributos del token (lexema, fila, columna,...)
- Los tokens se especifican mediante expresiones regulares

Ejemplo de análisis léxico

Expr. Regular	Token
`[a-z][a-z0-9]*`	**id**
`if`	**if**
`[0-9]+`	**entero**
`>`	**mayor**
`>=`	**mayorig**
`/`	**div**

Traza de una cadena:

`␣/r2d2␣␣>=`↩
`if␣5=a`

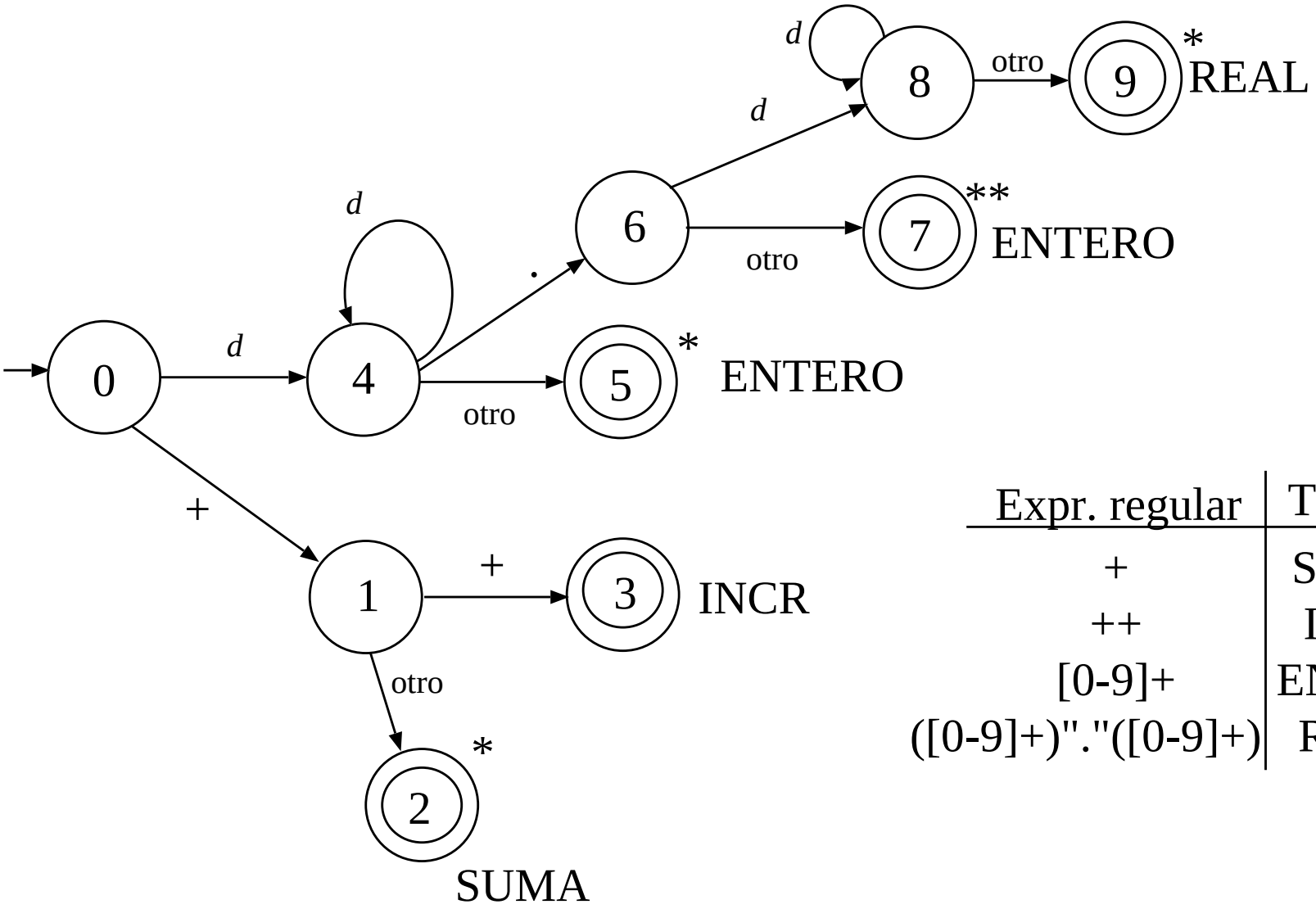
Token	Lexema	Fila	Columna
div	`/`	1	2
id	`r2d2`	1	3
mayorig	`>=`	1	9
if	`if`	2	1
entero	`5`	2	4
error léxico en '='			
id	`a`	2	6
`EOF`			

(si hay recuperación de errores)

Conclusiones tras la traza

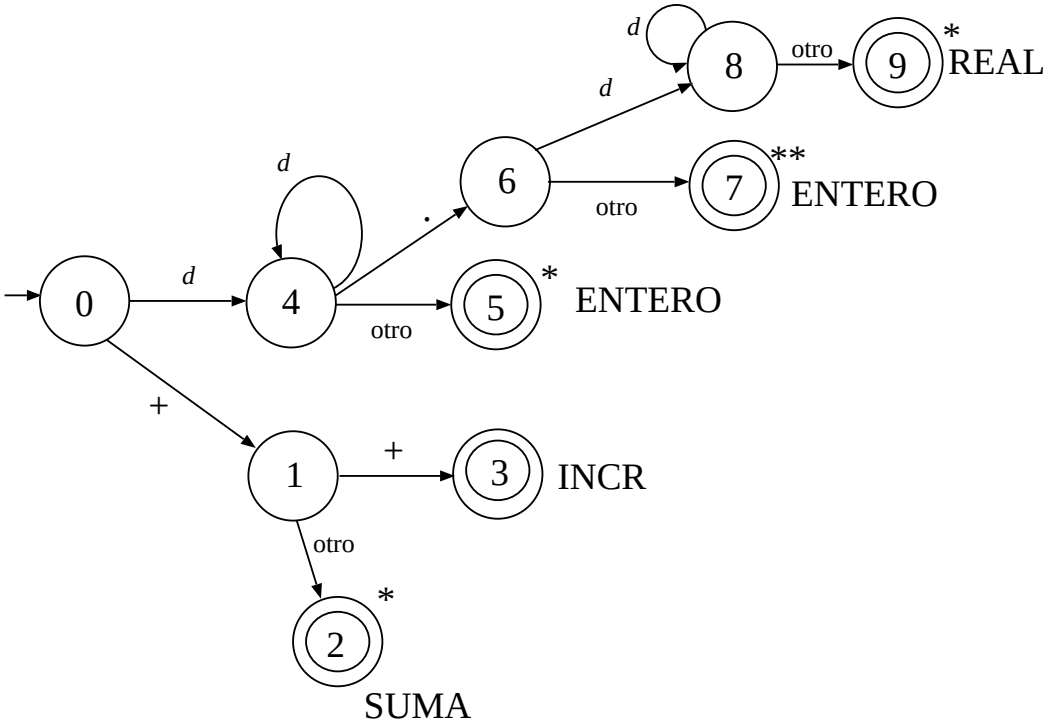
- Principio de la subcadena más larga
- Con algunos tokens nos tenemos que "pasar" leyendo la entrada (es decir, el fichero)
- El analizador léxico ignora los espacios en blanco, comentarios, saltos de línea,... pero sigue contando líneas y columnas
- Política de elección de palabra reservada sobre identificador
- Errores léxicos: caracteres no permitidos en el lenguaje (p.ej. "`$`") o no permitidos en un contexto determinado (p.ej. "`12.3`" vs "`.`" o "`.23`")

Diagrama de transiciones (DT)



Expr. regular	Token
+	SUMA
++	INCR
[0-9]+	ENTERO
([0-9]+)"."([0-9]+)	REAL

Funcionamiento del DT



Trazas

1. `12+`
2. `12.35`
3. `++123`
4. `1`
5. `1+2++`
6. `1.+`

Conclusiones tras la traza con DT

1. Mientras se analiza la cadena de entrada, el analizador debe *recordar* los *n* caracteres leídos anteriormente (donde *n* es el número máximo de asteriscos que aparecen en cualquier estado del DT)
2. Cuando se llega a un estado de aceptación, se devuelve el token asociado a ese estado, y se reintegran al buffer de entrada tantos caracteres como asteriscos tenga asociados el estado
3. IMPORTANTE: en el diagrama suelen estar previstas todas las posibles transiciones de salida de un estado (excepto en los de aceptación), para ello se usa la etiqueta "`otro`". Si no hay transición posible es un error (y no se indica en el DT).
4. En el DT se especifica lo que debe hacer el analizador léxico con cada carácter de la entrada. De los estados finales no salen transiciones
5. En algunos casos (números, id) es necesario leer un carácter más para estar seguros de haber reconocido el token completo

Algoritmo de análisis léxico

```
estado = 1;
c = leer();
do
{
     nuevo = delta(estado,c);
     if (nuevo == ERROR) errorLexico(...);
     if (esFinal(nuevo))
     {
        devolverCaracteres(nuevo);
        return tokenAsociado(nuevo);
     }
     else
     {
        estado = nuevo;
        c = leer();
     }
} while (true);
```

Construcción automática de analizadores léxicos

- lex/flex:

```
[a-z][a-z0-9]*    { return ID;}
```

- ANTLR:

```
ID : ('a'..'z')('a'..'z'|'0'..'9')+ ;
```

- ...

Lectura de ficheros carácter a carácter con Java

- En PL recomendamos utilizar `java.io.RandomAccessFile`
- Función recomendada para leer caracteres:

```
public char leerCaracter()
{
  char currentChar;

  try {
    currentChar = (char)fichero.readByte();
    return currentChar;
  }
  catch (EOFException e) {
     return EOF;   // constante estática de la clase
  }
  catch (IOException e) { ... // error lectura
  }
  return ' ';
}
```

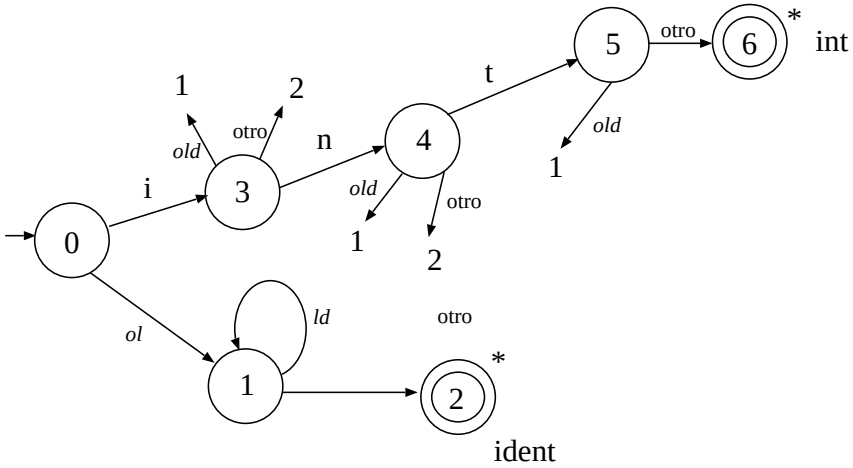
- ATENCIÓN: es posible que sea necesario *trucar* esta función (para tener en cuenta los caracteres leídos de más)

Caracteres leídos después del token

- En muchos casos, es necesario leer caracteres de más para asegurar el final del token. Ejemplo: `1234+34*`
- IMPORTANTE: Los caracteres leídos de más se deben *devolver* a la entrada para que sean leídos la siguiente vez que se invoque al analizador léxico (no se pueden perder)
- Hay dos formas de implementar esa *devolución*:
  - Retrocediendo el puntero de lectura del fichero (usando el método `.seek(...)`)
  - Haciendo buffering, almacenando en un vector (que funcionaría como una cola) los caracteres, y *trucando* la función de lectura de caracteres para consultar el buffer antes de leer del fichero

Palabras reservadas

1. Codificación explícita en el DT (no recomendable, excepto si se usa un generador automático de analizadores léxicos: `lex`, ANTLR, . . .)

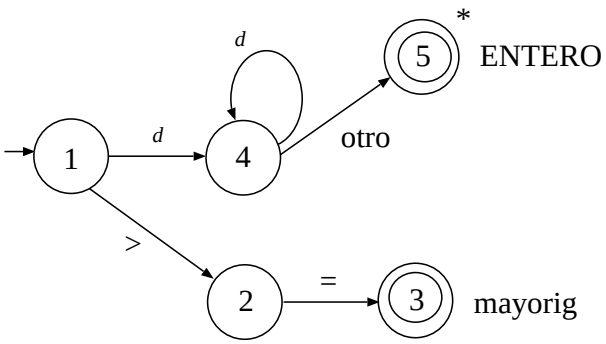


2. O bien, tras reconocer un identificador en el DT, se consulta en una lista de palabras reservadas (o en la tabla de símbolos) para decidir el token a devolver

Función de transición en DT

1. Matriz (dispersa) gestionada por rangos
2. Implementación en código mediante instrucciones de selección
3. Hay que almacenar qué token se debe devolver para cada estado final, y en el caso del diagrama de transiciones, el número de asteriscos (que es el número de caracteres que hay que devolver a la entrada o almacenar en el buffer)

Codificación de la tabla de transiciones




```
int delta (int estado, int c) {
  switch (estado) {
    case 1: if (c=='>') return 2;
            else if (c>='0' && c<='9') return 4;
            else return -1;
            break;
    case 2: if (c=='=') return 3;
            else return -1;
            break;
    case 3: return -1; // estado final
            break;
    case 4: if (c>='0' && c<='9') return 4;
            else return 5; // otro
            break;
    case 5: return -1; // estado final
            break;
    default: // error interno
  }
}
```

Representación de los tokens

```
public class Token {

    public int fila;
    public int columna;

    public String lexema;

    public int tipo;    // tipo es: ID, ENTERO, REAL ...

    public static final int
        ID       = 1,
        ENTERO   = 2,
        REAL     = 3,
        ...
        FINFICHERO = ...;
...
}
```

Ejercicio 1

Diseña un diagrama de transiciones (DT) para reconocer los siguientes componentes léxicos:

`numentero`	un número entero sin signo con dígitos entre 0 y 9, que no puede empezar por 0
`numoctal`	un número entero en octal, que empieza por `0` y con dígitos entre 0 y 7 (p.ej. `015` es 13)
`numhex`	un número entero en hexadecimal, que empieza por `0x` y con dígitos entre 0 y 9, y letras entre la A y la F (mayúsculas siempre)
`id`	identificador: secuencia de letras y dígitos que empieza por letra

El "`0`" se debe considerar que es un número octal. A continuación, indica cómo un analizador léxico construido sobre el DT anterior separaría en *tokens* la secuencia de caracteres "`09 120x021 0190xAF 0x9AGF 0xaF`".

Ejercicio 2

Diseña un diagrama de transiciones para reconocer los siguientes componentes léxicos:

`if`	la palabra reservada '`if`'
`ifthen`	la palabra reservada '`ifthen`'
`iftrue`	la palabra reservada '`iftrue`'
`oftrue`	la palabra reservada '`oftrue`'
`ident`	cualquier secuencia de letras y dígitos que empiece por una letra y no coincida con ninguna de las palabras reservadas
`mulop`	el símbolo '`/`'
`assop`	el símbolo '`=`'
`mulassop`	el símbolo '`/=`'
`addassop`	el símbolo '`+=`'
`identico`	el símbolo '`/==/`'

Indica cómo separa este analizador la secuencia de caracteres "`if5true/==/iftrue+=true/==false`" en *tokens*.

Ejercicio 3

Diseña un diagrama de transiciones para reconocer los siguientes componentes léxicos:

`numentero`	un número entero sin signo con uno o más dígitos entre 0 y 9
`numexpo`	un número que tiene uno o más dígitos, una "`E`" seguida de un signo "+" o "−" opcional, y que acaba con una secuencia de uno o más dígitos. Ejemplos: `1E1`, `112E+112`
`id`	identificador: secuencia de letras y dígitos que empieza por letra
`min`	la palabra reservada "`min`"
`mon`	la palabra reservada "`mon`"
`dosp`	el símbolo "`:`"
`igual`	el símbolo "`=`"
`asig`	el símbolo "`:=`"
`equiv`	el símbolo "`:==:`"

A continuación, indica cómo un analizador léxico construido sobre el diagrama de transiciones anterior separaría en *tokens* la secuencia de caracteres "`09 :==mono 12E-34 123E+3a3E27`".

Ejercicio 4

Diseña un diagrama de transiciones para reconocer los siguientes componentes léxicos:

`numentero`	un número entero sin signo con uno o más dígitos entre 0 y 9
`numfija`	un número que tiene cero o más dígitos, un punto, y una secuencia de uno o más dígitos. Ejemplos: `.233`, `1.1`, `112.112`
`id`	identificador: secuencia de letras y dígitos que empieza por letra
`del`	la palabra reservada "`del`"
`delete`	la palabra reservada "`delete`"
`postdec`	el símbolo "`--`"
`flecha`	el símbolo "`-->`"
`mayor`	el símbolo "`>`"

Ejercicio 5

Diseña un diagrama de transiciones para reconocer los siguientes componentes léxicos:

Expresión regular	Elemento léxico
<>	**distinto**
<	**menor**
>	**mayor**
+	**opsuma**
−	**opsuma**
++	**incremento**
−−	**decremento**
−>	**desref**
−−−>	**dobleref**
:=	**asig**