Generación de analizador léxico

Lex (flex, . . .)

Expresiones regulares “tipo `grep`”

Expresiones simples (un sólo carácter):

c	carácter c
“c”	carácter c
.	cualquier carácter (excepto separador de línea)
[cdx-y]	cualquier carácter del conjunto
[^cdx-y]	cualquier carácter fuera del conjunto

Expresiones regulares “tipo `grep`”

Operadores para expresiones compuestas:

R?	R ó ε
R*	cero o más ocurrencias de R
R+	una o más ocurrencias de R
$R_1 R_2$	R_1 concatenada con R_2
R_1 | R_2	R_1 ó R_2
(R)	R (necesario para agrupar)

Nota: `?*+` tienen precedencia sobre la concatenación, y ésta sobre `|`

Gramáticas EBNF “tipo `lex`”

- Partes derechas son expresiones regulares “tipo `grep`”
- Sin recursividad (ni indirecta ni directa)
- Por tanto, genera sólo lenguajes regulares
- Podemos desarrollar las partes derechas (de tokens):
  - Sustituimos no-terminales por sus partes derechas
  - Si es necesario, insertamos paréntesis para los operadores `*+?`

Ejemplo

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

digito $\rightarrow$ [0-9]

subr $\rightarrow$ _

letdig $\rightarrow$ letra | digito

TOKEN_IDENT $\rightarrow$ letra letdig* (subr letdig+)*

Desarrollamos la parte derecha:

TOKEN_IDENT $\rightarrow$ [a-zA-Z] ([a-zA-Z] | [0-9])*
(_ ([a-zA-Z] | [0-9])+)*

Items

Una vez desarrolladas partes derechas de los tokens:

- Usaremos items con la forma $A \rightarrow \alpha \bullet \beta$
- El punto indica hasta dónde se ha leído (de la entrada)
- Al leer c (ó "c" ó $[\cdots]$) pasamos de $A \rightarrow \alpha \bullet c \beta$ a $A \rightarrow \alpha c \bullet \beta$
- Podemos ver los items como estados de un autómata finito no-determinista con transiciones ε (AFNε)

Transiciones ε (“cierre”)

$$
\begin{array}{rcll}
\bullet(\,R\,) & \rightsquigarrow & (\bullet R) & \\
\bullet R? & \rightsquigarrow & R?\bullet & \\
\bullet R* & \rightsquigarrow & R*\bullet & \\
\bullet R_1|\cdots|R_n & \rightsquigarrow & \cdots|\bullet R_i|\cdots & i=2,\ldots,n \\
(\,R\bullet\,) & \rightsquigarrow & (\,R\,)\bullet & \\
R\bullet? & \rightsquigarrow & R?\bullet & \\
R\bullet* & \rightsquigarrow & R*\bullet,\ \bullet R* & \\
R\bullet+ & \rightsquigarrow & R+\bullet,\ \bullet R+ & \\
\cdots|R_i\bullet|\cdots & \rightsquigarrow & R_1|\cdots|R_n\bullet &
\end{array}
$$

Al final, los items con $\bullet$ delante de ( )?*+| pueden eliminarse

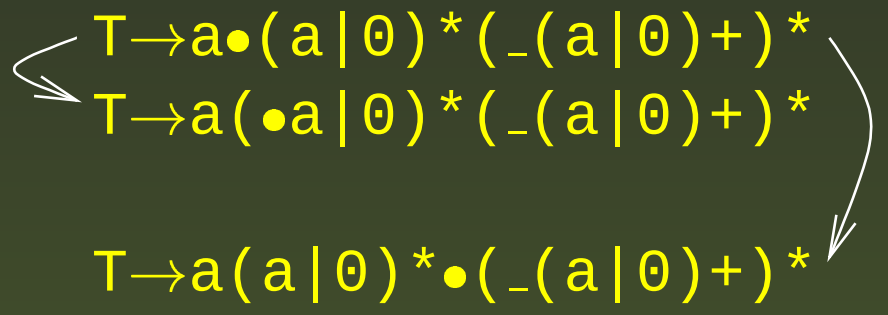
Ejemplo de transiciones ε

Partimos de:

$$T \rightarrow a \bullet (a|0)^*(\_(a|0)^+)^*$$

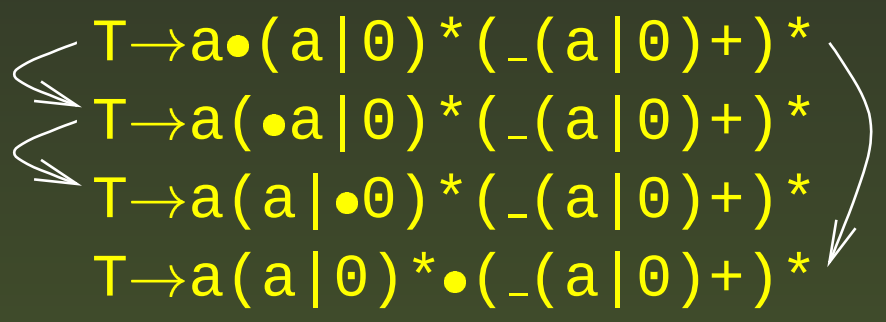
Ejemplo de transiciones ε

Obtenemos:



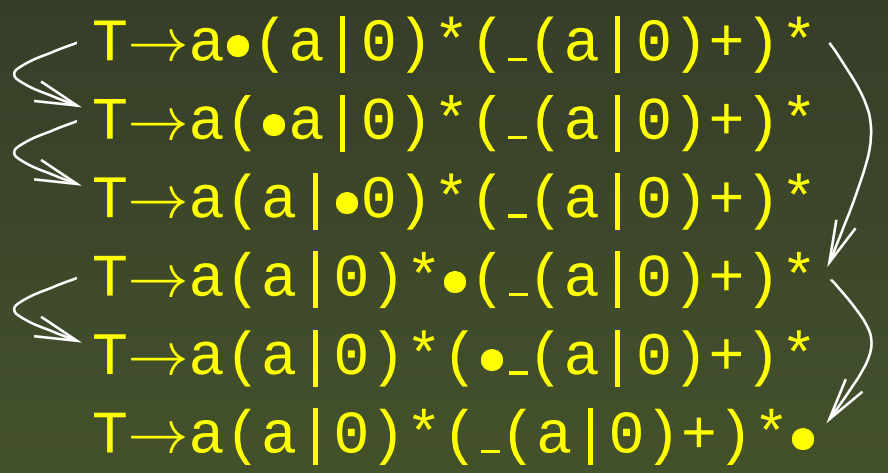
Ejemplo de transiciones ε

Obtenemos:



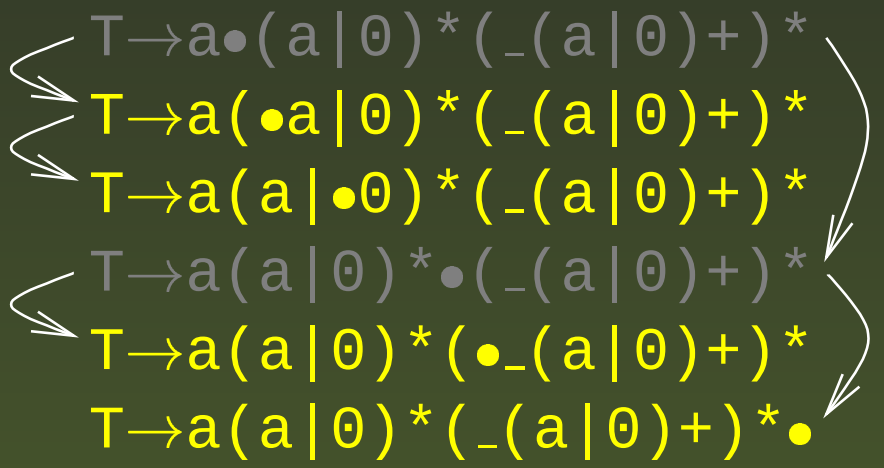
Ejemplo de transiciones ε

Cierre completado:



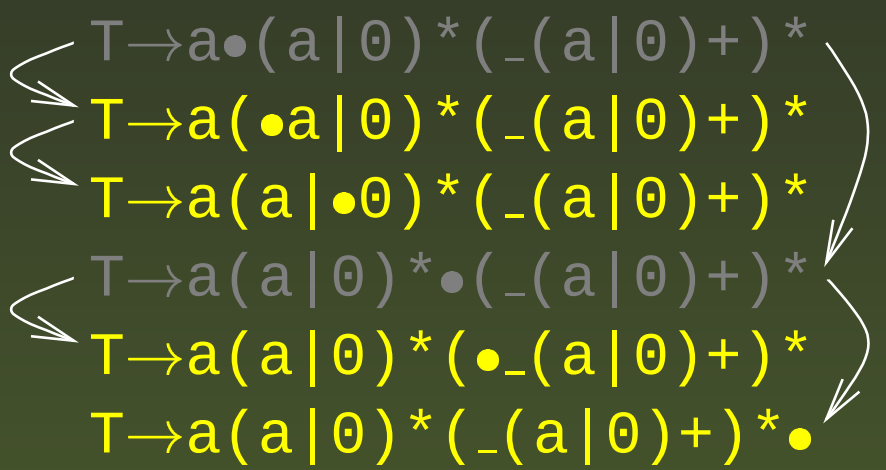
Ejemplo de transiciones ε

Eliminamos items inútiles:



Ejemplo de transiciones ε

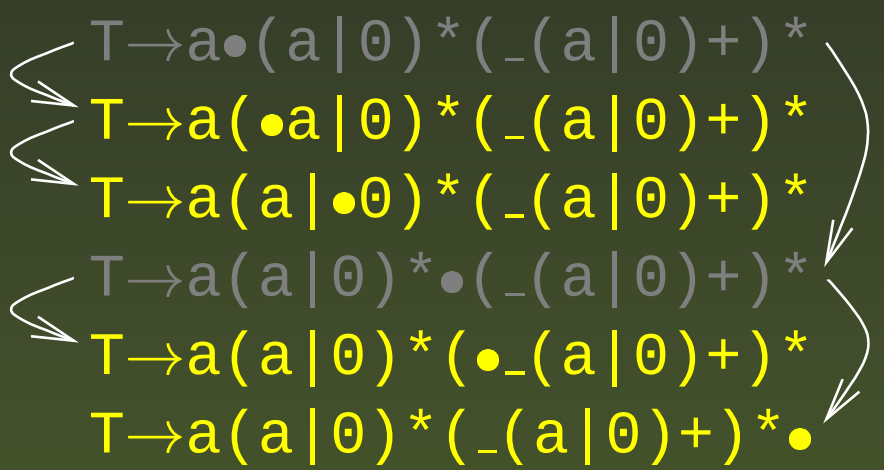
Eliminamos items inútiles:



Una transición con `a` nos daría `T→a(a•|0)*(_(a|0)+)*`

Ejemplo de transiciones ε

Eliminamos items inútiles:



Una transición con `a` nos daría `T→a(a•|0)*(_(a|0)+)*`

Tras cierre y eliminación, se obtiene el mismo conjunto

Construcción de AFD

- Estado inicial = cierre de items con el punto al principio
- Conjuntos de items simultáneamente válidos para los distintos tokens
- Asociamos a cada conjunto un estado del autómata finito determinista (AFD)
- Las transiciones del autómata corresponden a las de los items
- Los estados finales vienen indicados por el punto al final del ítem
- Se reconoce el token correspondiente a dicho ítem

Ejemplo de AFD

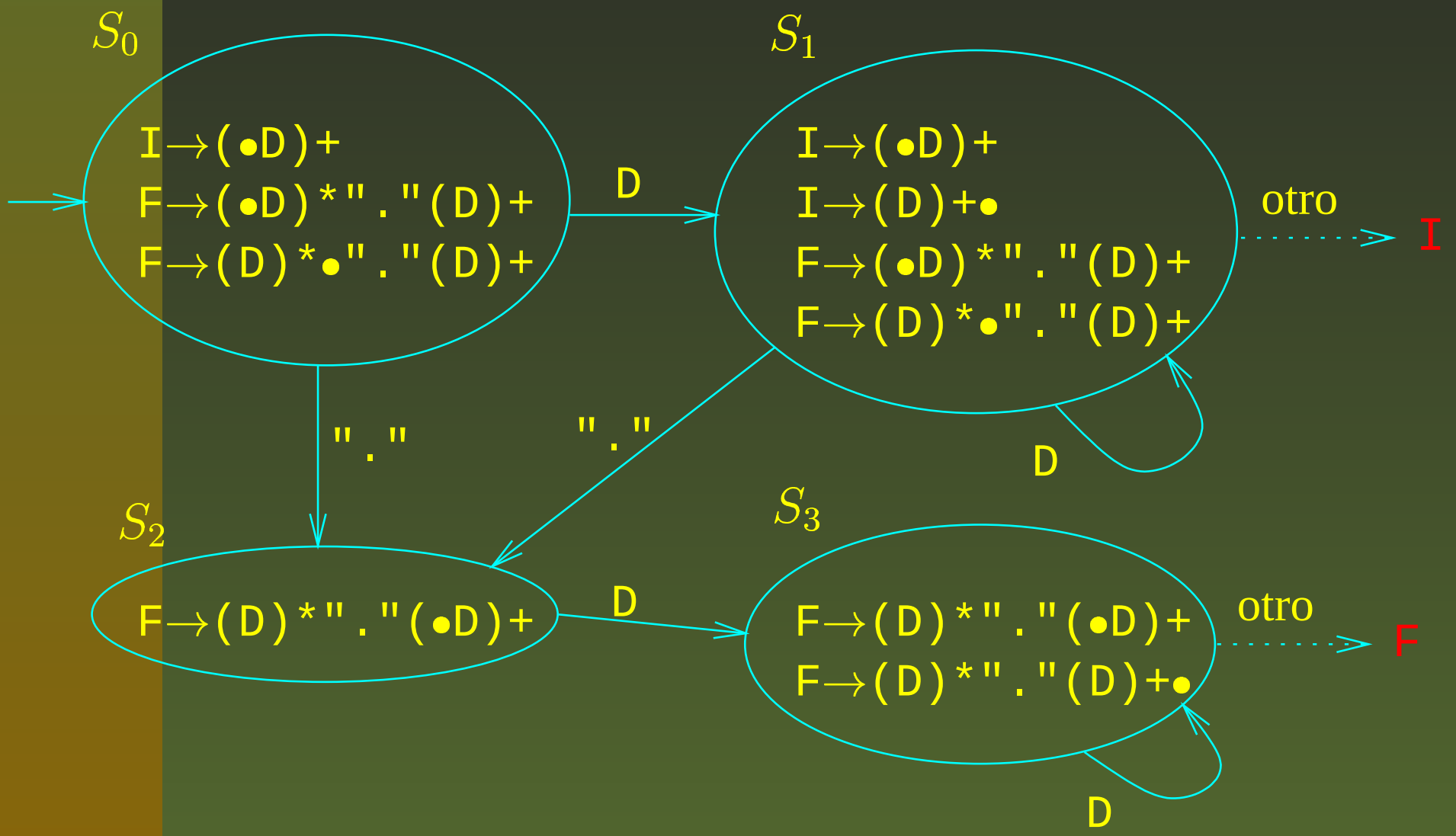


Tabla del AFD

	0	1	2
S_0	S_1		S_2
S_1	S_1	I	S_2
S_2	S_3		
S_3	S_3	F	F

Código símbolo:

0=D

1=otro

2=“.”

Tabla del AFD

	0	1	2
S_0	S_1		S_2
S_1	S_1	I	S_2
S_2	S_3		
S_3	S_3	F	F

Código símbolo:

0=D

1=otro

2=“.”

Linearización por desplazamiento de filas, p.e.:

estado+símbolo	0	1	2	3	4	5	6	7	8
verificación									
transic/decisión									

Correspondencia estado-índice:

S_0	S_1	S_2	S_3

Tabla del AFD

	0	1	2
S_0	S_1		S_2
S_1	S_1	I	S_2
S_2	S_3		
S_3	S_3	F	F

Código símbolo:

0=D

1=otro

2=“.”

Linearización por desplazamiento de filas, p.e.:

estado+símbolo	0	1	2	3	4	5	6	7	8
verificación	S_0		S_0						
transic/decisión	S_1		S_2						

Correspondencia estado-índice:

S_0	S_1	S_2	S_3
0			

Tabla del AFD

	0	1	2
S_0	S_1		S_2
S_1	S_1	I	S_2
S_2	S_3		
S_3	S_3	F	F

Código símbolo:

0=D

1=otro

2=“.”

Linearización por desplazamiento de filas, p.e.:

estado+símbolo	0	1	2	3	4	5	6	7	8
verificación	S_0		S_0	S_1	S_1	S_1			
transic/decisión	S_1		S_2	S_1	I	S_2			

Correspondencia estado-índice:

S_0	S_1	S_2	S_3
0	3		

Tabla del AFD

	0	1	2
S_0	S_1		S_2
S_1	S_1	I	S_2
S_2	S_3		
S_3	S_3	F	F

Código símbolo:

0=D

1=otro

2="."

Linearización por desplazamiento de filas, p.e.:

estado+símbolo	0	1	2	3	4	5	6	7	8
verificación	S_0	S_2	S_0	S_1	S_1	S_1			
transic/decisión	S_1	S_3	S_2	S_1	I	S_2			

Correspondencia estado-índice:

S_0	S_1	S_2	S_3
0	3	1	

Tabla del AFD

	0	1	2
S_0	S_1		S_2
S_1	S_1	I	S_2
S_2	S_3		
S_3	S_3	F	F

Código símbolo:

0=D

1=otro

2="."

Linearización por desplazamiento de filas, p.e.:

estado+símbolo	0	1	2	3	4	5	6	7	8
verificación	S_0	S_2	S_0	S_1	S_1	S_1	S_3	S_3	S_3
transic/decisión	S_1	S_3	S_2	S_1	I	S_2	S_3	F	F

Correspondencia estado-índice:

S_0	S_1	S_2	S_3
0	3	1	6

Tabla del AFD

	0	1	2
S_0	S_1		S_2
S_1	S_1	I	S_2
S_2	S_3		
S_3	S_3	F	F

Código símbolo:

0=D

1=otro

2="."

Linearización por desplazamiento de filas, p.e.:

estado+símbolo	0	1	2	3	4	5	6	7	8
verificación	0	1	0	3	3	3	6	6	6
transic/decisión	3	6	1	3	I	1	6	F	F

Correspondencia estado-índice:

S_0	S_1	S_2	S_3
0	3	1	6

Formato de un fichero `flex`

Declaraciones en C entre `%{` y `%}`
Declaraciones especiales comenzadas por `%`
Reglas de “no-terminales” auxiliares

`%%`

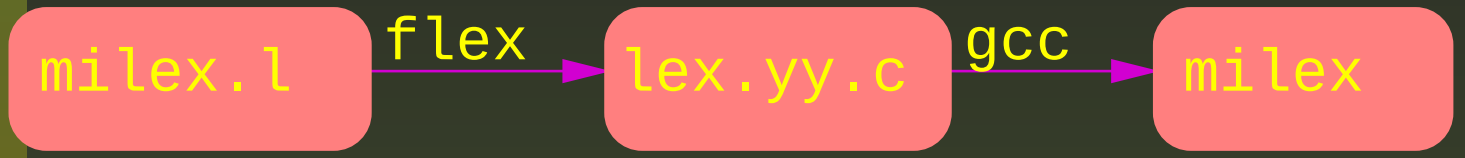
Secuencia de

- Expresiones regulares (*patrones*) de símbolos a reconocer
- Entre llaves, acciones en C asociadas, normalmente terminadas en `return` código-del-símbolo

`%%`

Rutinas auxiliares y/o `main` si programa independiente

Generación



1. Editamos un fichero de especificación `milex.l`
2. Procesamos con `flex milex.l`
3. Normalmente lo compilaremos junto con un analizador sintáctico producido por `bison`
4. Si introducimos `main()` que llame a `yylex()`, podremos tener un analizador léxico independiente:
   `gcc -o milex lex.yy.c -lfl`
   (usará ficheros e/s estándar)

Contenido de `lex.yy.c`

1. Declaraciones iniciales copiadas
2. Función `yylex()` con bucle de reconocimiento de patrones:
   - Cada vez que reconoce uno, almacena el texto en `yytext` y ejecuta el código asociado
   - Si más de uno, da preferencia al texto de mayor longitud
   - Si igual longitud, da preferencia al patrón indicado en primer lugar
3. Rutinas auxiliares copiadas

El patrón especial R_1/R_2 reconoce R_1 sólo si va seguido de R_2 (el texto de R_2 se volverá a leer)

Algunas acciones especiales

- `BEGIN` estado: pasan a considerarse las expresiones regulares comenzadas por <estado>
- `ECHO`: visualiza `yytext`
- `yymore()`: el texto actual se añadirá al del próximo token
- `yyless(n)`: se conservan los primeros `n` caracteres en `yytext` y el resto se volverán a procesar
- Si no hay acción, ignora el texto y reanuda la lectura para un nuevo patrón
- Si no se encuentra patrón alguno (error), se deja atrás (y visualiza) el primer carácter y vuelve a intentarlo

Tratamiento de errores en `flex`

Nada especialmente previsto, pero podemos (y debemos):

- Detectar caracteres no permitidos usando “`.`”
- Detectar patrones no permitidos, en su caso
- Para patrones erróneos frecuentes, reparar el texto/símbolo, p.e.: comillas o comentario sin cerrar
- Visualizar los correspondientes mensajes explicativos

Ejemplo `flex` (1/3)

```
%{
#define ENTERO 257
#define IDENTIF 258
int numlin = 1;
void error(char*);
%}

letra   [a-zA-Z]
digito  [0-9]
letdig  {letra}|{digito}
```

Ejemplo `flex` (2/3)

```
%%

("+"|-)?{digito}+  {return ENTERO;}
{letra}{letdig}*   {return IDENTIF;}
[-+*/]             {return yytext[0];}
\n                 {numlin++; /* continúa leyendo */}
[ \t] |
^#.*               {/* ignora */}
.                  {error("caracter raro");}
<<EOF>>            {return 0; /* por defecto */}
```

Ejemplo `flex` (3/3)

```
%%

int main(int argc, char** argv) {
  int s;
  if(argc>1) yyin=fopen(argv[1],"r"); /* else yyin=stdin */
  do printf("%i ",s=yylex()); while(s != 0);
  return 0;
}

void error(char* mens) {
  printf("Error lexico en linea %i: %s\n",numlin,mens);
}
```

Conclusión

- Un generador automático nos permite concentrarnos en el diseño léxico y acciones asociadas
- Desventajas: dependencia de la herramienta, coste de aprendizaje
- `flex` puede usarse como filtro potente de texto
- Más información en página `man flex` y en libro *"lex & yacc"*, de Levine et al. (681.3.06)