Seminario de introducción a Flex

David Portolés Rodríguez

dporto@unizar.es

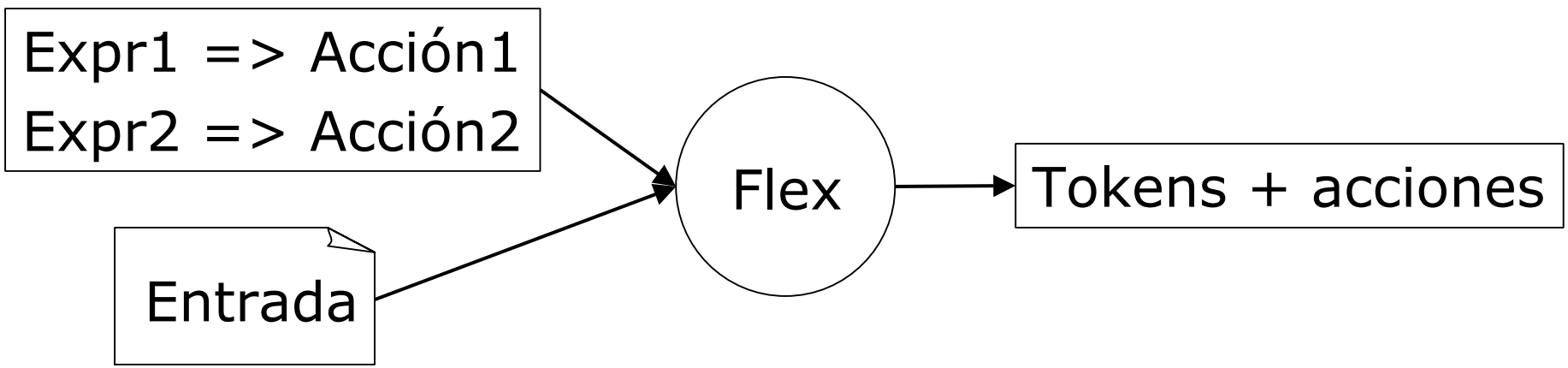
Lenguajes y Sistemas Informáticos

Dpto. de Informática e Ing. de Sistemas

Universidad de Zaragoza

¿Qué es Flex?

- Flex es un una herramienta que permite generar analizadores léxicos.
- A partir de un conjunto de expresiones regulares (patrones léxicos), Flex busca concordancias en un fichero de entrada y ejecuta acciones asociadas a estas expresiones.



Tokens, lexemas y patrones léxicos (1)

- *Token:* cada uno de los elementos reconocidos por el analizador léxico
- *Lexema:* secuencia de caracteres de la entrada que corresponden a un token
- *Patrón léxico:* forma compacta de describir conjuntos de lexemas

Tokens, lexemas y patrones léxicos (2)

- Token 1:1 patrón
- Token 1:N lexemas

- Los tokens se pasan como valores simples
- Pero algunos requieren algo más de información:
  - Ej: para una constante, habitualmente no sólo se necesita saber que es una constante sino también cuál es su valor

Expresiones regulares

- Definición formal de los lexemas que corresponden a un token
- Permitirá decidir cuándo se reconoce un token a partir de una secuencia de elementos de la entrada
- Se pueden "bautizar" para poder referenciarlas posteriormente

Expresiones regulares en Flex

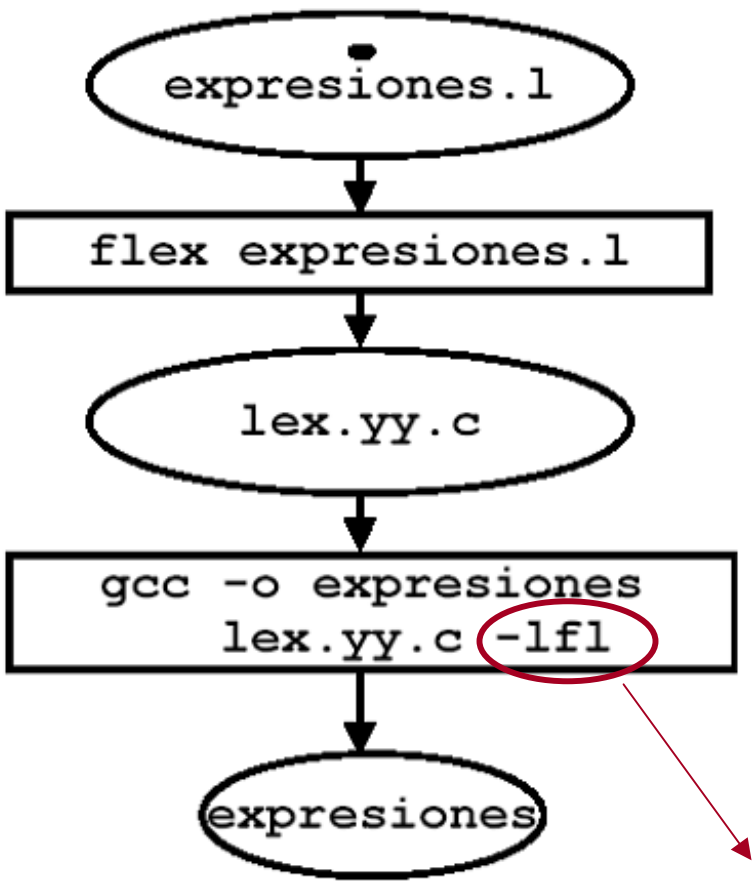
.	cualquier carácter excepto "\n"
r*	0 ó más concat. de **r**
r+	1 ó más concat. de **r**
r?	0 ó 1 veces **r**
[c_1...c_n]	conj.caracteres {c_1...c_n}
a-b	caracteres {a,succ(a),...}
^r	**r** debe concordar al principio de la línea
[^c_1...c_n]	cualquier car.$\notin${c_1...c_n}
r$	**r** debe concordar al terminar la línea
r{m,n}	entre m y n concatenaciones de **r**
****	para concordancia exacta de caracteres especiales: \\ \. \?
r1|r2	lo habitual
"c_1...c_n"	literalmente c_1...c_n
r1/r2	reconoce **r1** sólo si va seguida de **r2** **(¿e.r.?)**

Ejemplos de expresiones regulares en Flex

- Dígito => 0|1|2|3|4|5|6|7|8|9
- Dígito => [0-9]
- Número entero: [0-9]+
- Número entero: {Dígito}+
- Número real sin signo: {Dígito}+”.”{Dígito}*
- Identificador => [a-zA-Z][_a-zA-Z]*
- DNI => [0-9]{8}”-”[A-Z]
- Tfno. fijo/móvil nacional => [69][0-9]{8}
- Teléfono => (6[0-9]{8}) | (9[0-9]{8})
- Palabras que empiezan por efe: [fF][a-zA-Z]*
- Líneas que NO empiezan por efe: ^[^fF].*
- Literal “aNd” exacto: aNd
- Palabra “and”: [aA][nN][dD]

Proceso de compilación

- Ejemplo de uso de LEX:



sufijo '.l' para fuente FLEX

invocación a FLEX

fuente con el analizador

-lfl: biblioteca necesaria

analizador

-L/opt/flex/lib -lfl (Merlin)

Compilación con Makefile

```
#=================================================
#  Fichero: Makefile
#  Tema:    genera un analizador léxico para
#           introducir Flex/Lex
#  Fecha:   Septiembre-03
#  Uso:     make
#=================================================
LEX=flex
CC=gcc

expresiones: lex.yy.o
     $(CC) -o expresiones lex.yy.o -lfl
                    #-L/opt/flex/lib -lfl para Merlin
lex.yy.o: lex.yy.c
     $(CC) -c lex.yy.c

lex.yy.c: expresiones.l
     $(LEX) expresiones.l
```

Estructura de un fichero Flex

- Tres partes separadas por %%

```
Sección de definiciones
%%
Sección de reglas
%%
Sección  de código de usuario
```

Sección de definiciones

- Se incluye código C para declarar variables, funciones, tipos enumerados, include's, etc que se vayan a necesitar.
  - Se copia exactamente en lex.yy.c
  - Va entre %{ %}
- Se “bautizan” las E.R.
- Se pueden establecer condiciones iniciales del analizador léxico que se está construyendo
- Típicamente contiene una declaración de tipos enumerados para los posibles tokens
  - O mejor aún, un #include a un fichero que lo contenga

Sección de reglas

- Tiene el formato:

```
patrón1    { acción1 }
patrón2    { acción2 }
   ...
```

- Los patrones son E.R. y la acción es código C
- Dos tipos de líneas:
  - Fuente C: empiezan por blanco || “%{” || “%}”
  - Patrón-acción: empiezan por otra cosa
- El Fuente C se copia literalmente en lex.yy.c
- Si hay concordancia con el patrón se ejecuta la acción asociada
- Si hay varios patrones coincidentes se decide cuál corresponde y se ejecuta la acción
- Si no hay concordancia, ejecuta ECHO (copia el lexema en la salida)

Consejos para la sección de reglas de Flex

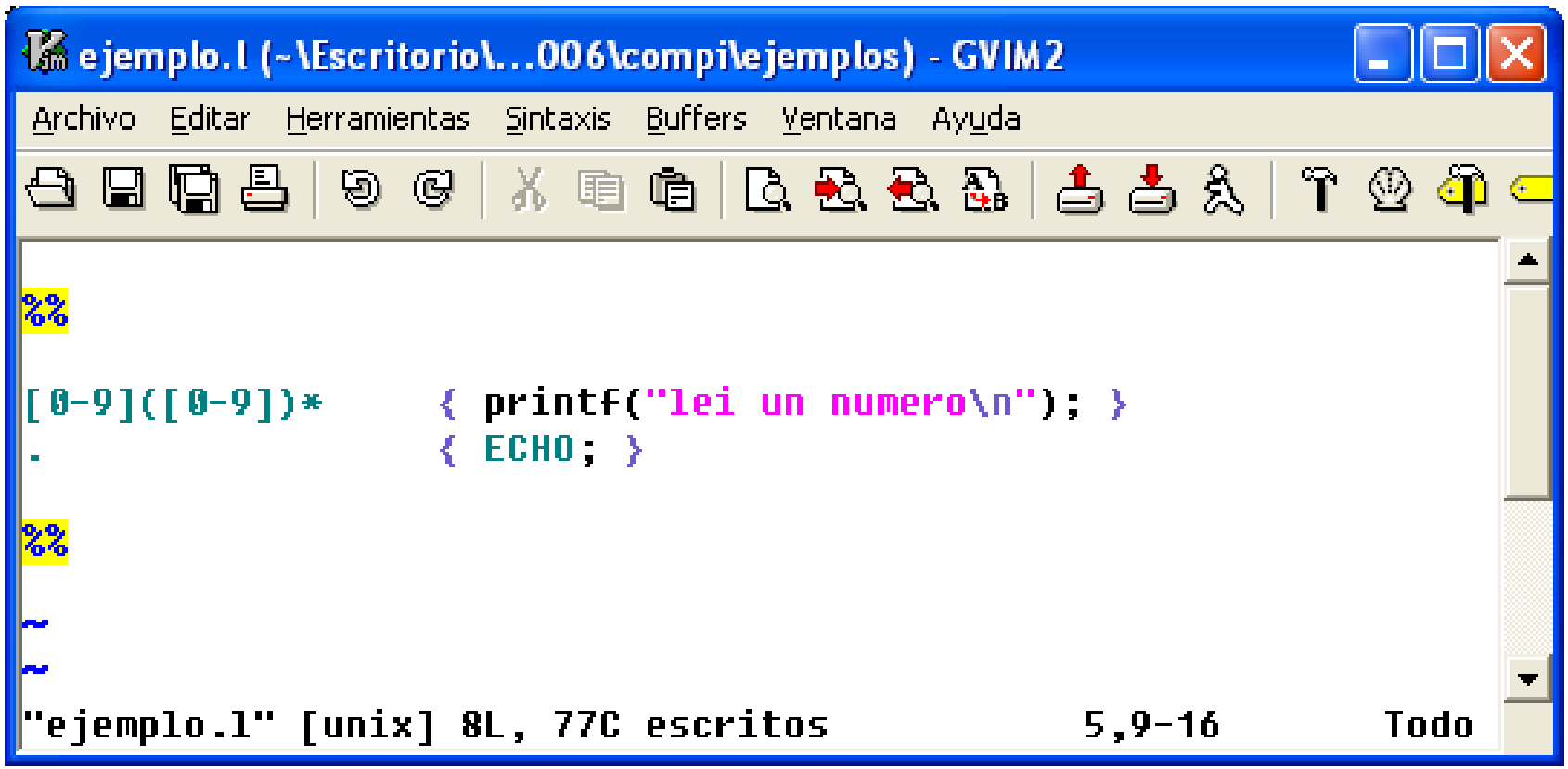
- En la definición de patrones:
  - Usar [] para conjuntos de caracteres
  - Usar () para agrupar.
    - Por ej, es típico que vayan seguidos de *
  - Usar {} para utilizar los “bautizos” de la primera sección
  - Cuidado con:
    - Comentarios de C
    - Caracteres especiales: . * ? % [etc
    - Espacios en blanco y tabuladores a principio de línea

Sección de código de usuario

- Código C creadas por necesidad del programador Flex
- Se copian literalmente en el lex.yy.c
- Aquí es donde situaremos gran parte de la lógica funcional de nuestras prácticas en lugar de hacerlo en la parte de acciones

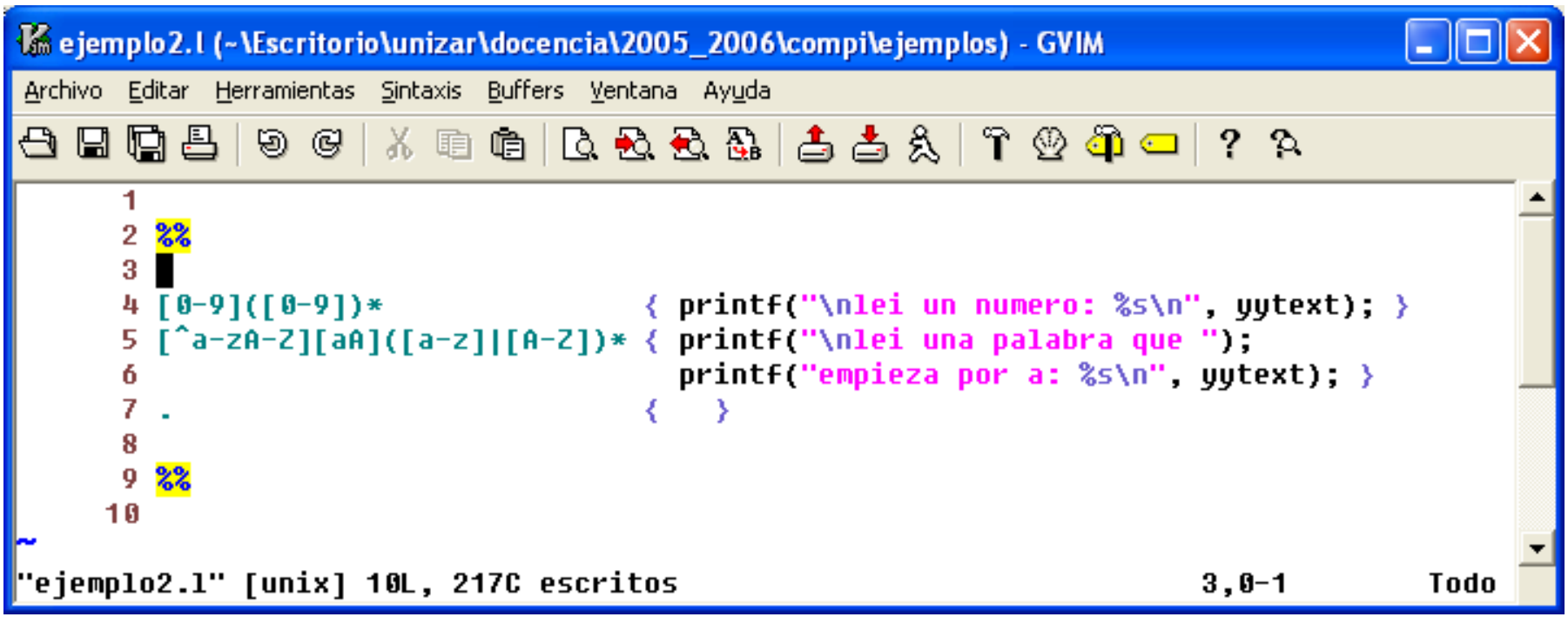
Primer ejemplo en Flex

- Reconocedor de números enteros
- Ejecutar: ejemplo1 < fich_entrada.txt



Accediendo al lexema

- Reconocedor de números enteros y palabras que empiezan por a.
- Mostrar los lexemas reconocidos



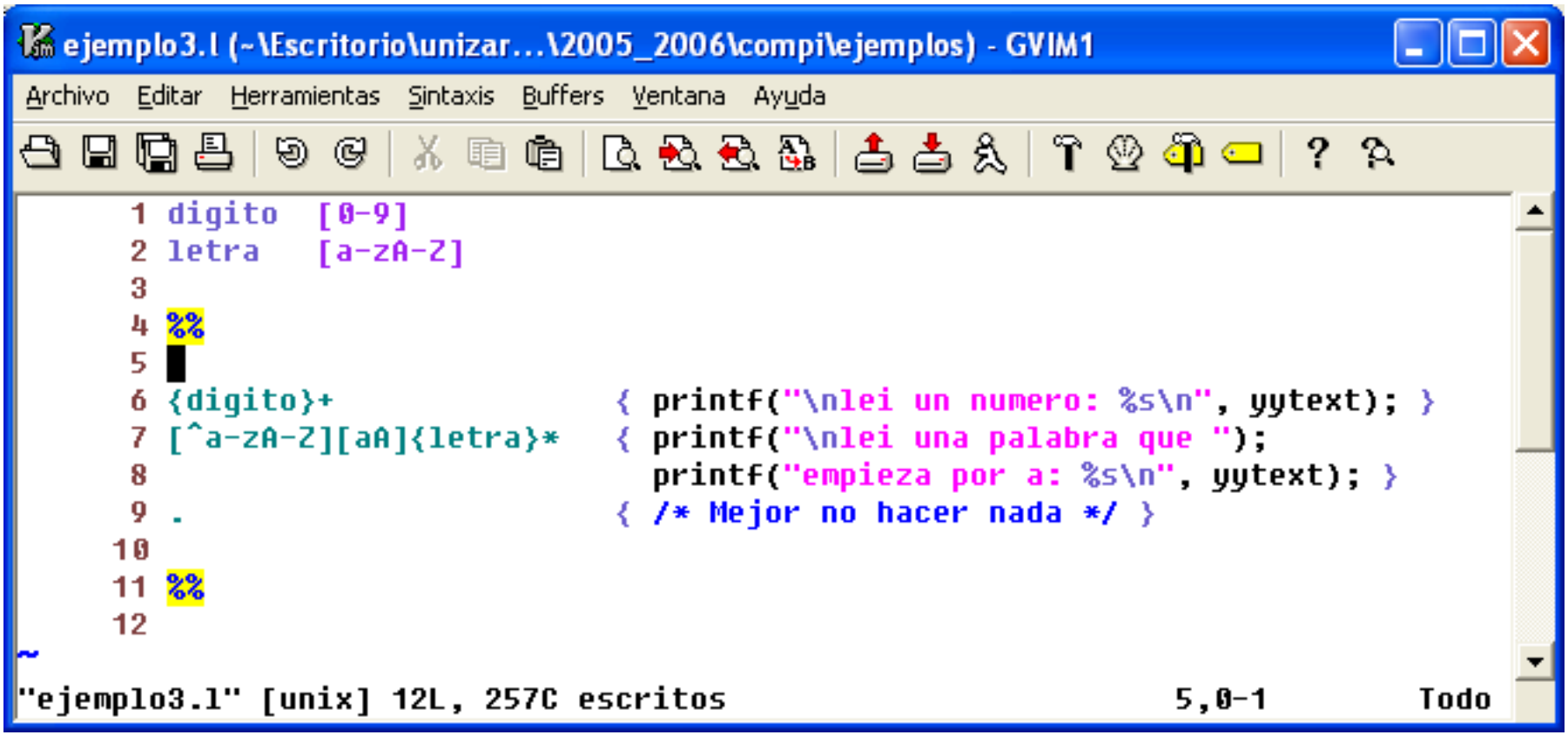
¿Es correcto en todos los casos?

Accediendo al lexema

- yytext contiene un puntero global al lexema reconocido
- yyleng contiene la longitud del lexema
- Hay otras más: yyin, yyout...
- También hay funciones: yylex(), yywrap(), yymore(), yyless(), yyterminate()...

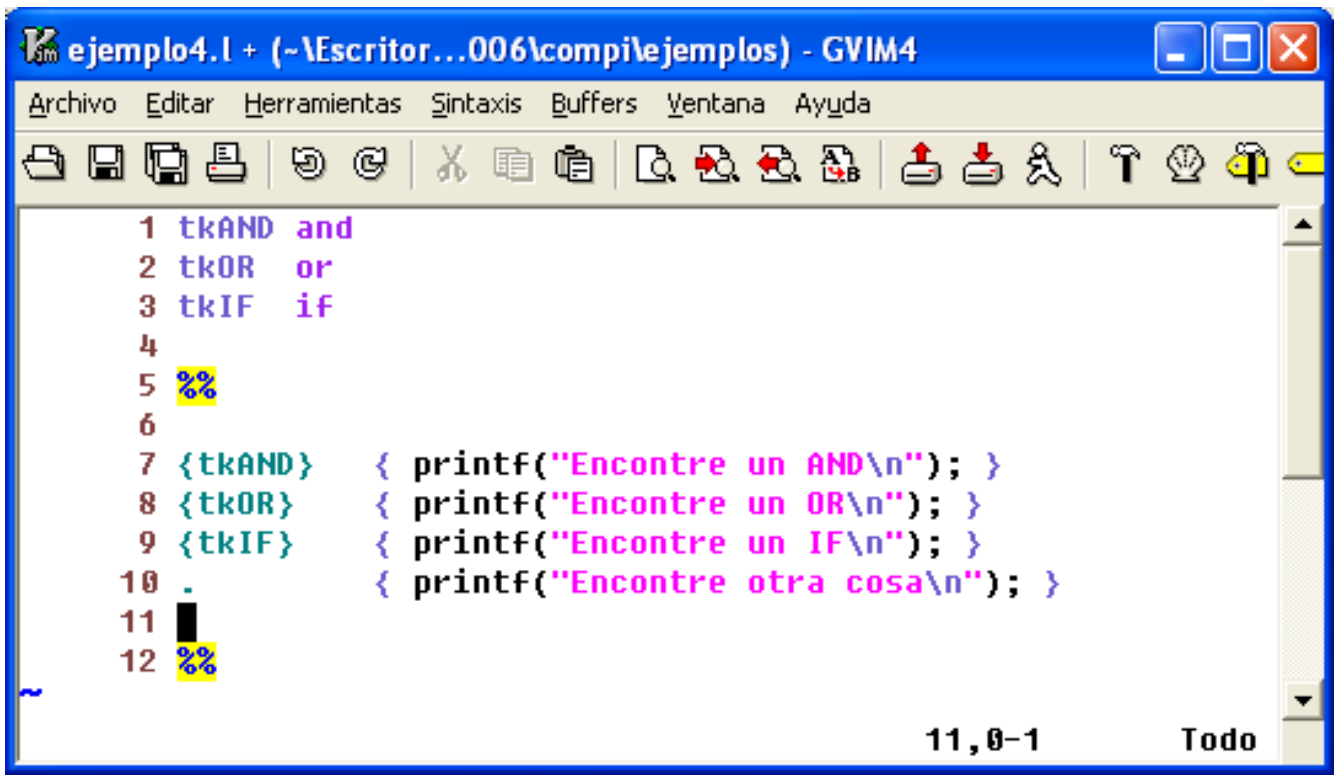
“Bautizando” una E.R.

- Ídem que el anterior
- “Bautizar” E.R.



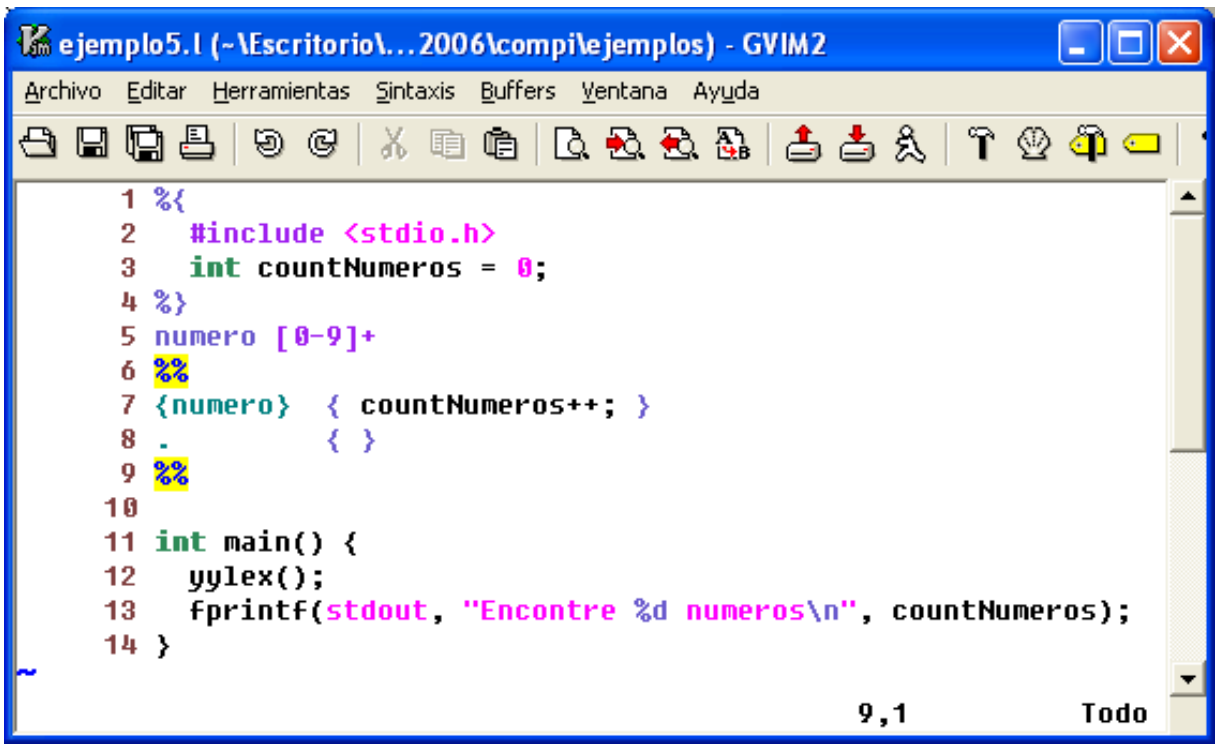
Ignorando mayúsculas en tokens

- Reconocedor de palabras reservadas
- Compilar con: flex –i (no existe siempre)
- Ejecución interactiva: Ctrl-D para acabar

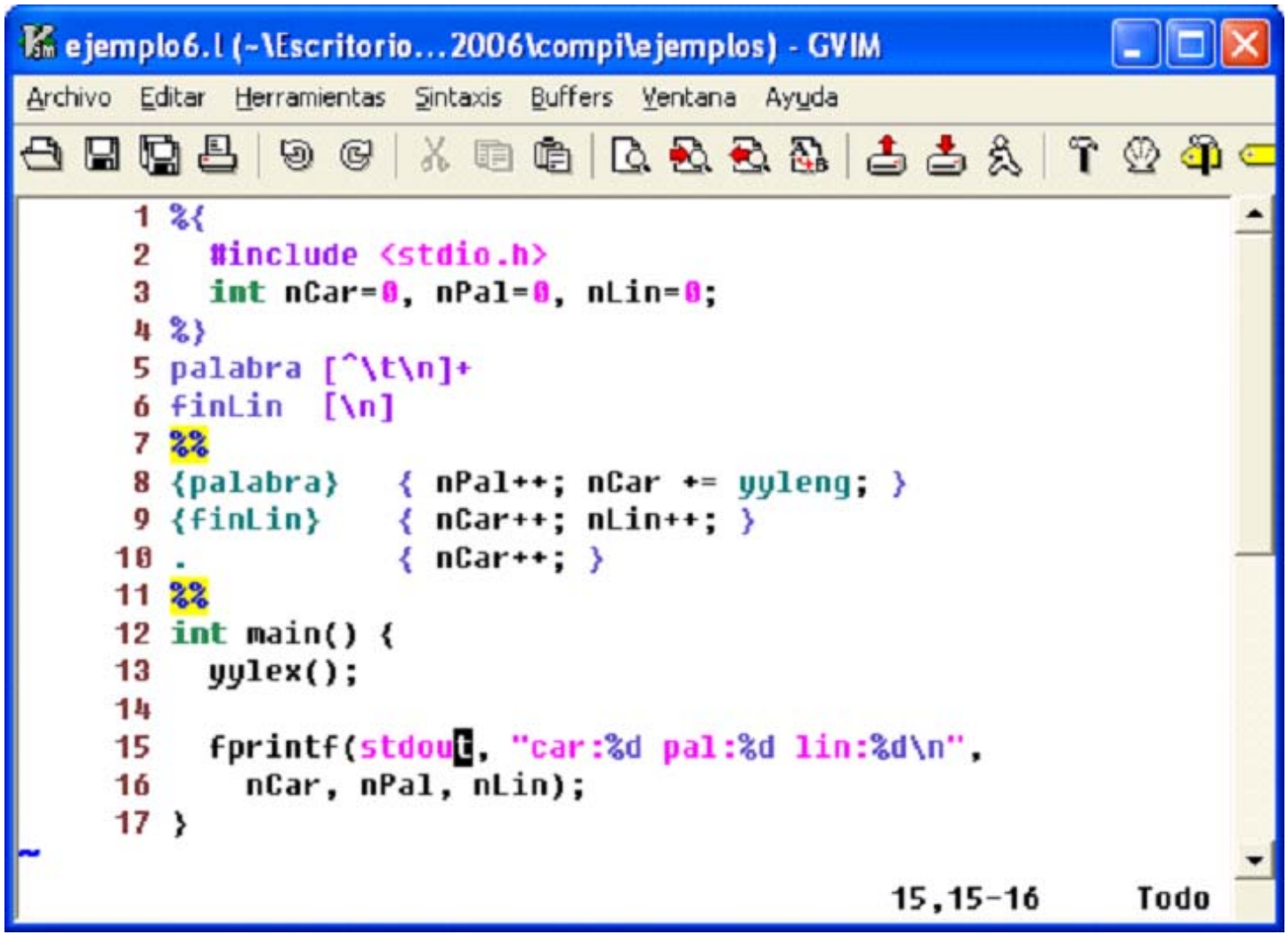


Definiendo variables C y redefiniendo main()

- Por defecto, Flex crea un main() que llama a yylex()
- Si redefinimos el main(), hay que llamar a yylex()
- yylex() entra en un bucle infinito hasta que no hay más tokens o se hace un return



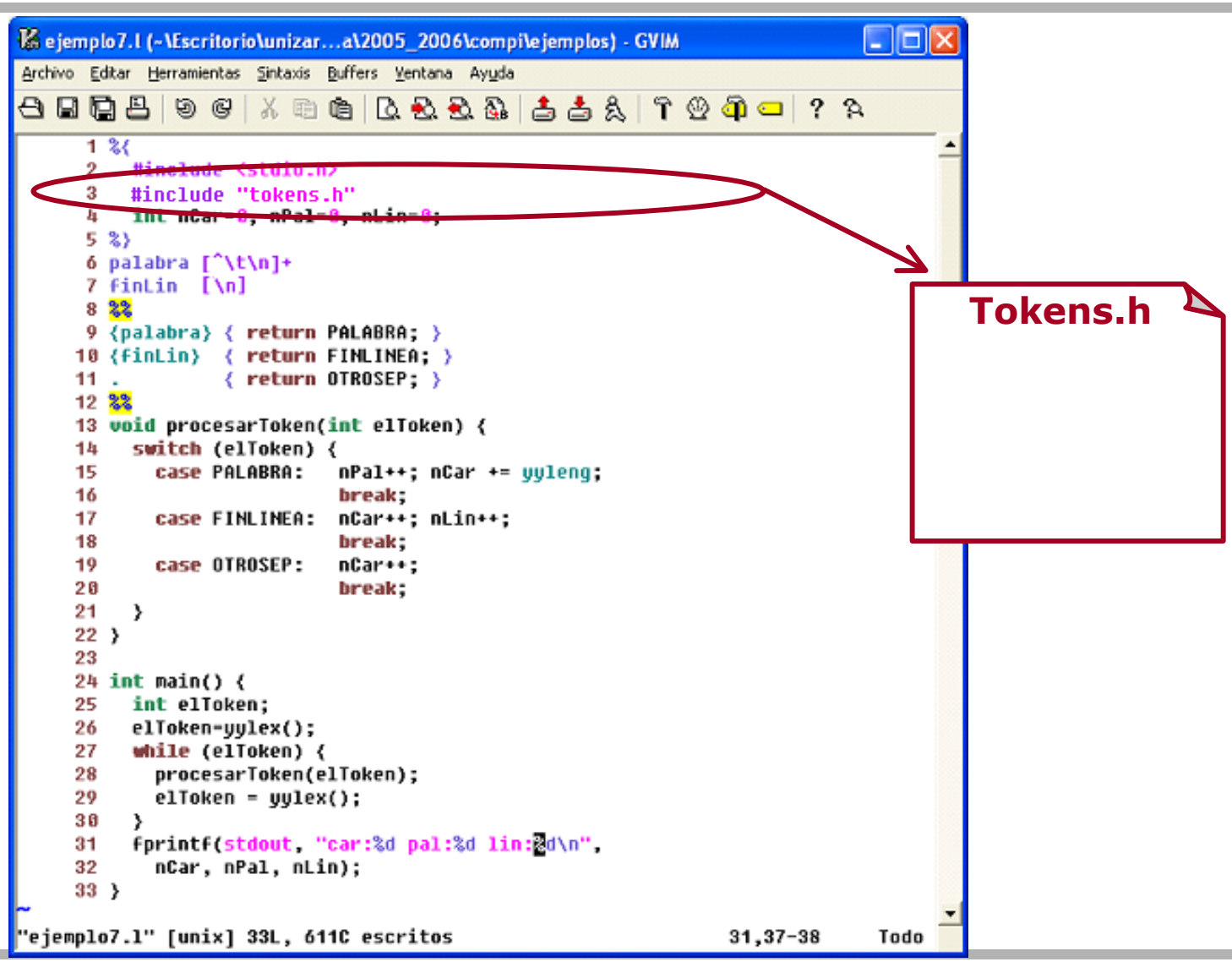
Ejecutando código en las acciones: ejemplo



Ejecutando código en las acciones: consideraciones

- Si se pone toda la lógica funcional en las acciones el código queda poco elegante y complejo
- A veces no queda más remedio que ponerlo en las acciones

Lógica funcional en el main(): ejemplo



Lógica funcional en el main(): Consideraciones

- El código queda más elegante
- Esto nos permitirá enlazar el analizador léxico con el sintáctico (Yacc/Bison) de un modo más simple

Condiciones de arranque: Consideraciones

- Permite activar reglas condicionalmente
  - Una regla cuyo patrón se prefija con una condición de arranque, sólo se activa cuando el analizador se encuentre en un determinado estado
- Se declaran en la (primera) sección de definiciones, usando líneas sin sangrar comenzando con %s seguida por una lista de nombres.
- Se activa utilizando la acción BEGIN. Hasta que se ejecute la próxima acción BEGIN:
  - Las reglas con la condición de arranque dada estarán activas
  - Las reglas con otras condiciones de arranque estarán inactivas
  - Las reglas sin condiciones de arranque también estarán activas.
- BEGIN(0) retorna al estado original donde sólo las reglas sin condiciones de arranque están activas.
- BEGIN(INITIAL) es equivalente a BEGIN(0).
- No se requieren los paréntesis alrededor del nombre de la condición de arranque pero se considera de buen estilo.

Condiciones de arranque: ejemplo

ejemplo8.l (~\Escritorio\unizar...a\2005_2006\compi\ejemplos) - GVIM

Archivo Editar Herramientas Sintaxis Buffers Ventana Ayuda

```
 1 %s comentarioLargo comentarioCorto
 2
 3 %%
 4
 5 "/*"                          BEGIN(comentarioLargo);
 6 <comentarioLargo>[^*]*        /* come todo lo que no sea '*'     */
 7 <comentarioLargo>"*"+[^*/]*   /* come '*'s no seguidos por '/'   */
 8 <comentarioLargo>"*/"         BEGIN(INITIAL);
 9
10 "//"                          BEGIN(comentarioCorto);
11 <comentarioCorto>[^\n]*         /* come todo lo que no sea '*'   */
12 <comentarioCorto>\n            BEGIN(INITIAL);
13
14 .                             { ECHO; }
15
16 %%
17
18 int main() {
19   yylex();
20 }
~
                                                  20,1          Todo
```

¿Es correcto en todos los casos?
¿Qué reglas están activas en cada caso?

Consideraciones a tener en cuenta (1)

- Ser muy riguroso con la estructura:
  - Si comienza por blanco, ya no es línea de patrón-acción
  - Definir claramente las tres zonas delimitadas por %%
- Tener cuidado con los anidamientos de (),[],{}
- Tener presente la forma de resolución de múltiples patrones coincidentes
  - Elige el que concuerda con el string más largo
  - Si los patrones reconocen el mismo string, elige el que aparece primero en las declaraciones Flex

Consideraciones a tener en cuenta (2)

- Seguir principio “divide y vencerás”
  - Si falla el analizador léxico, lo más simple es ir eliminando patrón-acción hasta localizar el conflictivo
- Seguir principio KISS (Keep it simple, stupid!)
  - Pensar muy bien las E.R. a utilizar puede simplificar mucho la tarea
- Tener presente la diferencia entre mayúsculas y minúsculas de los lexemas:
  - Complicará el código con un montón de [aA][nN][dD]...
  - Se resuelve con flex –i
  - Consulta con tu profesor y/o compilador si estás autorizado a utilizarlo

Para saber más

- Consultar la página web de las asignaturas:
  - Compiladores I:

    http://webdiis.unizar.es/~ezpeleta/COMPI/compiladoresI.htm
  - Lenguajes, Gramáticas y Autómatas:

    http://webdiis.unizar.es/asignaturas/LGA/
- En ellas aparecen enlaces a documentación de utilidad (manuales online, apuntes, etc)

Preguntas