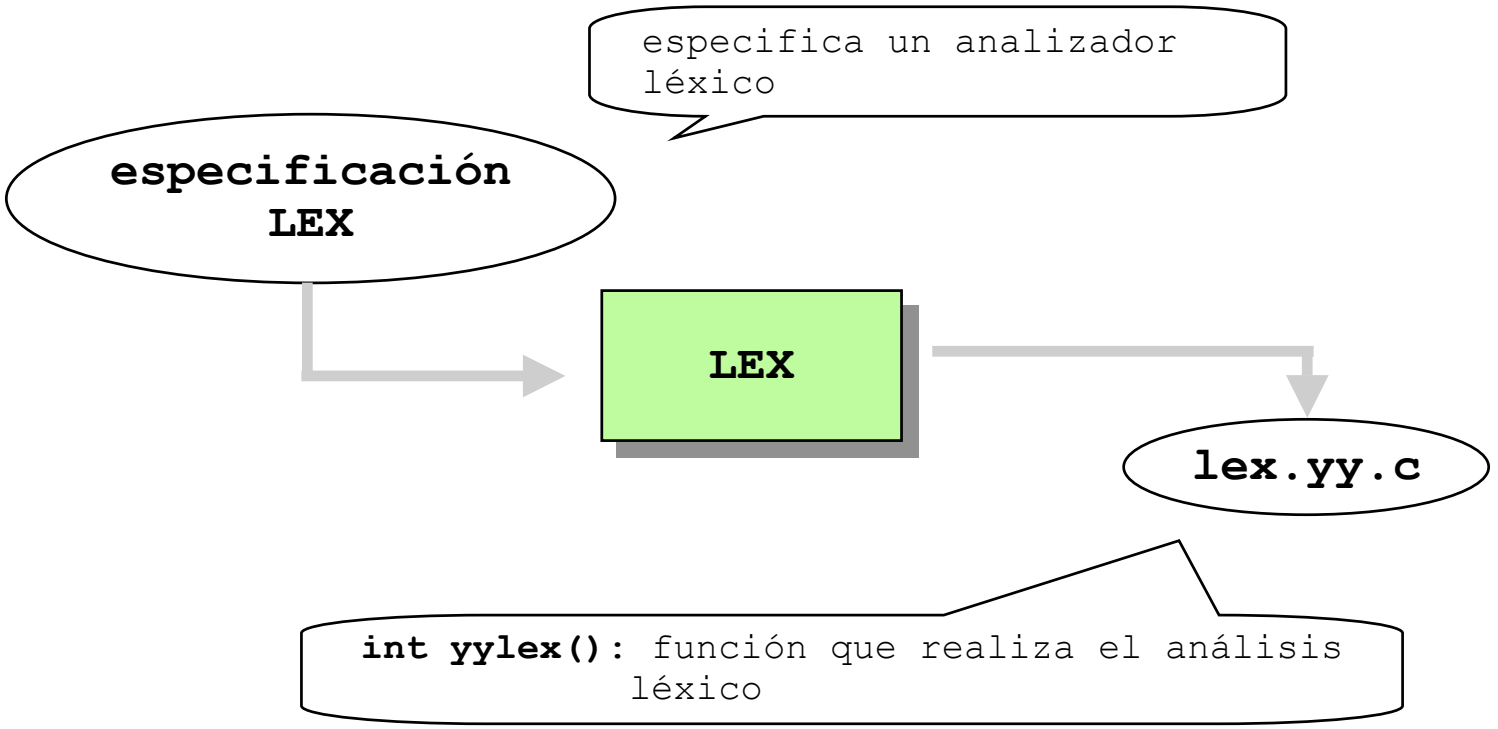
LEX: Un generador de analizadores léxicos

- Como hemos visto, el paso de una expresión regular a un AF se puede hacer de manera automatizada
- Existen varias herramientas que relizan dicho trabajo
- Generadores de analizadores léxicos
  - AT&T Lex (más común en UNIX)
  - MKS Lex (MS-Dos)
  - Flex
  - Abraxas Lex
  - Posix Lex
  - ScanGen
  - JLex
  - ...

- ¿Qué hace LEX?


especifica un analizador léxico
especificación LEX
LEX
lex.yy.c
int yylex(): función que realiza el análisis léxico

- LEX sigue el esquema:

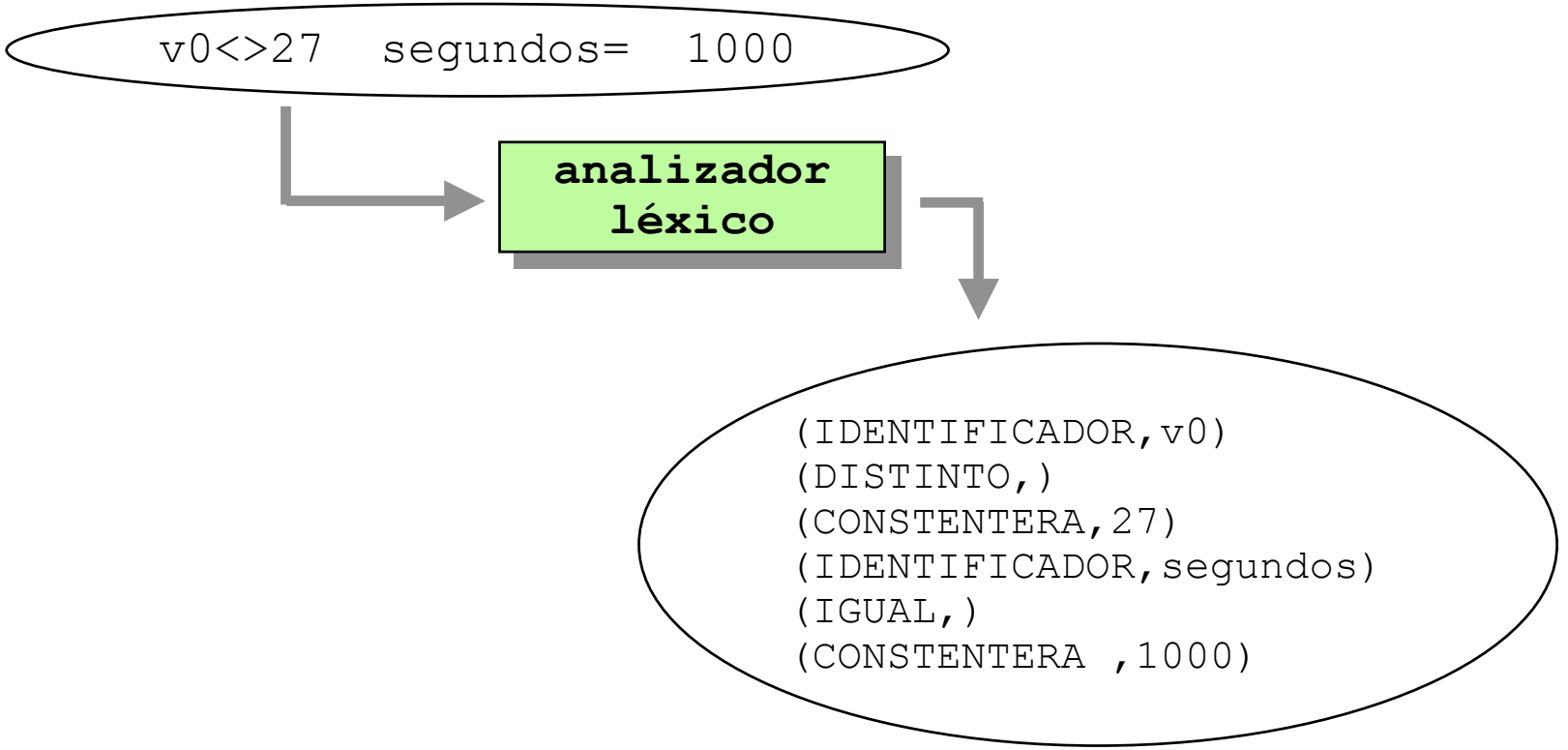
```
patrón1      { acción1 }
patrón2      { acción2 }
     .......
patrónk      { acciónk }
```

  donde:
  - patrón: expresión regular
  - acción: fuente C con las acciones a realizar cuando el patrón concuerde con un lexema
- Funcionamiento:
  - LEX recorre entrada estándar hasta que encuentra una concordancia
    - » un lexema correspondiente al lenguaje de algunas de las e.r. representadas por los patrones
  - entonces, ejecuta el código asociado (acción)
  - permite acceder a la información asociada al lexema (string, longitud del mismo, nº de línea en el fuente,...)

- Ejemplo: implementar en LEX, un analizador léxico para

```
menor →           <
mayor →           >
menorIgual →      <=
mayorIgual →      >=
igual →           =
distinto →        <>
letra →           a|b|...|z|A|B|....|Z
digito →          0|1|...|9
identificador →   letra (letra |digito)*
constEntera →     digito digito *
```

- Comportamiento deseado:



- Estructura de un programa LEX

```
sección de definiciones
%%
sección de reglas
%%
sección de rutinas del usuario
```

- **Sección de definiciones**:
  - bloques literales
    - » se copian "tal cual" al fuente `lex.yy.c`
    - » entre '`%{`' y '`%}`'
  - definiciones regulares ("bautismo")
  - declaraciones propias para el manejo de tablas de LEX
  - condiciones iniciales
    - » cambiar el funcionamiento del reconocedor

también `enum{...}`

`yacc`
lo generará
automáticamente

```
%{   /* fichero: expresiones.l */
#include ....
#define MENOR                    128
#define MENORIGUAL               129
#define MAYOR                    130
#define MAYORIGUAL               131
#define IGUAL                    132
#define DISTINTO                 133
#define IDENTIFICADOR            134
#define CONSTENTERA              135

int yylval;
     /* para atributo cte. entera*/
typedef int token;
%}
%%
          sección de reglas
%%
     sección de rutinas de usuario
```

- **Sección de reglas**
  - Formato: **`patrón {acciones}`**
  - dos tipos de líneas:
    - » empiezan por blanco, '`%{`' ó '`%}`' fuente C
    - » empiezan por otra cosa: línea patrón-código
  - las líneas de fuente C se copian directamente en el fuente `lex.yy.c`
  - al encontrar concordancia, ejecutará código asociado
  - si no encuentra concordancia, ejecuta `ECHO` (copia el lexema en salida estándar)

```
%{
        sección de definiciones
%}
%%
(\t|\n|" ")+                          {/* no hace nada */};
[0-9]([0-9])*                         {sscanf(yytext,"%d",&yylval);
                                       return(CONSTENTERA);}
"<"                                   {return(MENOR);}
....
....
"<>"                                  {return(DISTINTO);}
[a-zA-Z]([a-zA-Z]|[0-9])*             {return(IDENTIFICADOR);}
.                                     {ECHO;}
%%
      sección de rutinas de usuario
```


patrón

acción

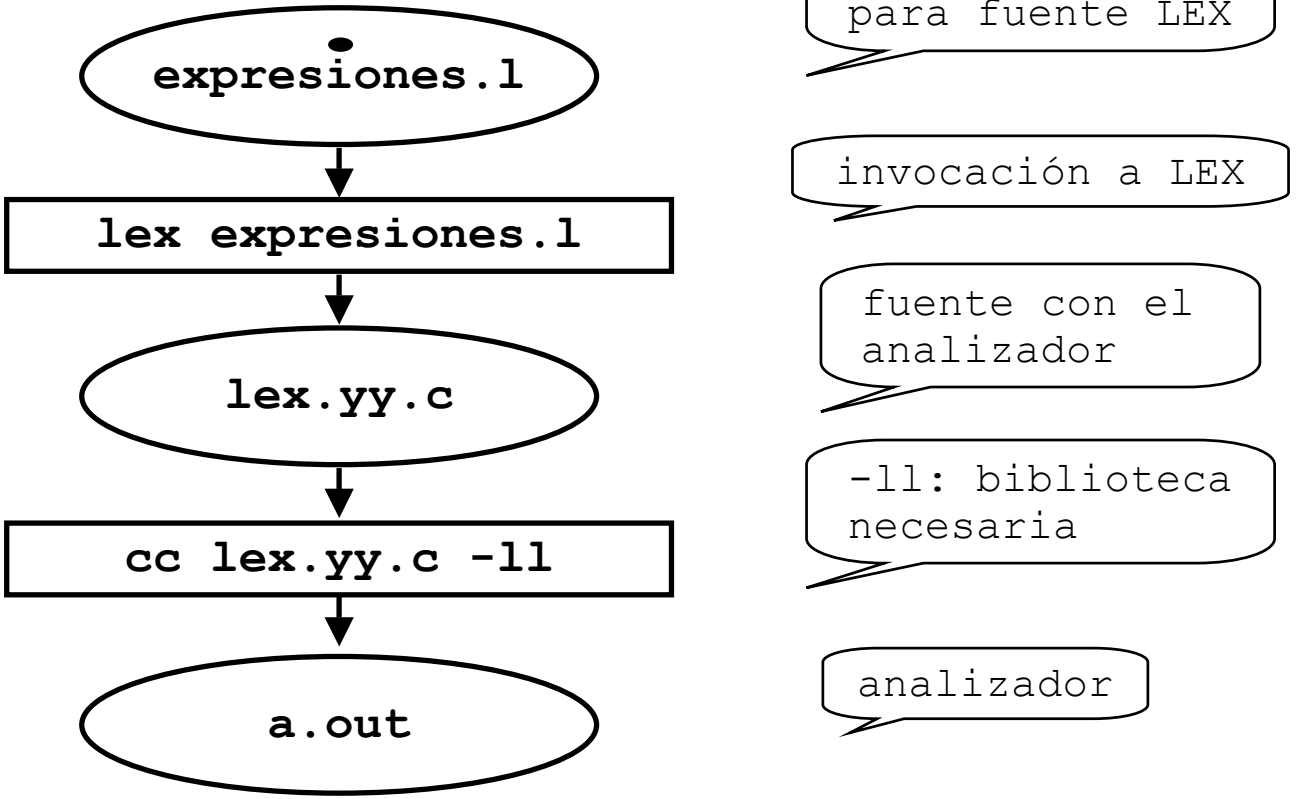
- **Sección de rutinas de usuario**
  - su contenido se copia literalmente en el fuente `lex.yy.c`
  - contiene funciones C escritas por el usuario y necesario para el analizador
    - » manejo de tabla de símbolos
    - » generación de salidas (cuando no es necesario generación de código)

```
%{
      sección de definiciones
%}
%%
      sección de reglas
%%
/*--------------------------------------
  Sólo para comprobación
--------------------------------------
*/
void escribeInfo(token elToken)
{  switch(elToken){
     case IDENTIFICADOR:
         fprintf(stdout,
             "ident.\t%s",yytext);
     break;
          ..........
   }
}
```

```
/*--------------------------------------
  Cuando EOF, yylex() dev. 0
--------------------------------------*/
int main()
{  token elToken;

   elToken=yylex();
   while(elToken){
      escribeInfo(elToken);
      elToken=yylex();
   }
}
```

- Ejemplo de uso de LEX:



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

analLexExpSimples: lex.yy.o
     $(CC) -o analLexExpSimples 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: primer.l
     $(LEX) primer.l
```

- Ejemplo de invocación:
  - el fichero 'entrada' es

```
v0<>27
segundos=       1000
```

  - invocación:

```
a.out < entrada
```

  - genera la siguiente salida

```
identificador  v0
distinto       <>
numero         27
identificador  segundos
igual          =
numero         1000
```

- También se pueden dar nombres a expresiones regulares

```
%{     /* fichero: expresiones.l */
      ......
int yylval;     /* hab. los define Yacc */
typedef int token;
%}
separadores     [\n\t" "]+
.....
letra           [a-zA-Z]
digito          [0-9]
identificador {letra}({letra}|{digito})*
constEntera   {digito}({digito})*
%%
          sección de reglas
%%
     sección de rutinas de usuario
```

Lex

Nota: yytext definido en *ll* como

```
char yytext[YYLMAX]
```

Nota: De necesitar declararlo como variable externa, usar

```
extern char yytext[]
```

pero no

```
extern char *yytext
```

```
%{
        sección de definiciones
%}
%%
{separadores}   {/* no hace nada */};
{constEntera}   {sscanf(yytext,"%d",&yylval);
                 return(CONSTENTERA);}
....
{identificador} {return(IDENTIFICADOR);}
.               {ECHO;}
%%
      sección de rutinas de usuario
```

Nota: el nombre de la ER en la parte de patrones debe ponerse entre '{' y '}' para distinguirlo del patrón literal

{separadores}<>separadores

- Extensiones de LEX a e.r.

.	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.?)**

- ¿Qué pasa cuando hay ambigüedad?
- LEX aplica dos reglas:

```
.....
letra           [a-zA-Z]
digito          [0-9]
identificador {letra}({letra}|{digito})*
%%
"if"                {return(IF);}
{identificador}   {return(IDENT);}
%%
```

Regla 1

Aplica el patrón que concuerde con el string más largo

Regla 2

Si dos patrones representa el mismo string, aplica el que aparece antes en las declaraciones de LEX

Un ejemplo: contar caracteres, líneas y palabras

```
#include <stdio.h>
int nCar=0, nPal=0, nLin=0;
%}
palabra  [^ \t\n]+
finLin   \n
%%
{palabra} {nCar+=yyleng;nPal++;}
{finLin}  {nCar++;nLin++;}
.         {nCar++;}
%%
int main(){
   yylex();
   fprintf(stdout,
        "car:%d  pal:%d  lin:%d\n",
         nCar, nPal, nLin
   );
}
```

Una versión "a las bravas"

Un ejemplo: contar caracteres, líneas y palabras

```
#include <stdio.h>
enum {PALABRA=128,FINLINEA,
      OTROSEP};
int nCar=0, nPal=0, nLin=0;
%}
palabra  [^ \t\n]+
finLin   \n
%%
{palabra} {return PALABRA;}
{finLin}  {return FINLINEA;}
.         {return OTROSEP;}
%%
int main(){

}
```

un poco más estructurada y elegante

```
int elToken;
elToken=yylex();

while(elToken){
  switch(elToken){
    case PALABRA: nPal++;nCar+=yyleng;
                  break;
    case FINLINEA:nCar++;nLin++;
                  break;
    case OTROSEP: nCar++;
  }
  elToken=yylex();
}
fprintf(stdout,
    "car:%d  pal:%d  lin:%d\n",
    nCar, nPal, nLin
);
```