Una Herramienta para el Análisis Léxico: Lex

Alejandro Viloria Lanero (aviloria@infor.uva.es)
Teoría de Autómatas y Lenguajes Formales
Universidad de Valladolid

Como hemos ido viendo, el shell de los sistemas UNIX/Linux nos ofrece varias herramientas para el tratamiento masivo de la información contenida en ficheros de texto, que nos permiten hacer búsquedas, reemplazos y múltiples procesamientos de más alto nivel a un nivel de 'campo' dentro de la 'línea'. Todas ellas se apoyaban en las expresiones regulares para llevar a cabo su cometido, y permitían encadenar su salida a modo de tubería con otras herramientas, para que de esta forma se aumentase su potencia de procesamiento.

Pero a menudo todo esto no es suficiente. En ocasiones se requiere poder realizar acciones de más alto nivel o bien mucho más complejas en cada ocurrencia de una expresión regular. Lex va a permitirnos definir nuestras propias acciones en un lenguaje mucho más común: C.

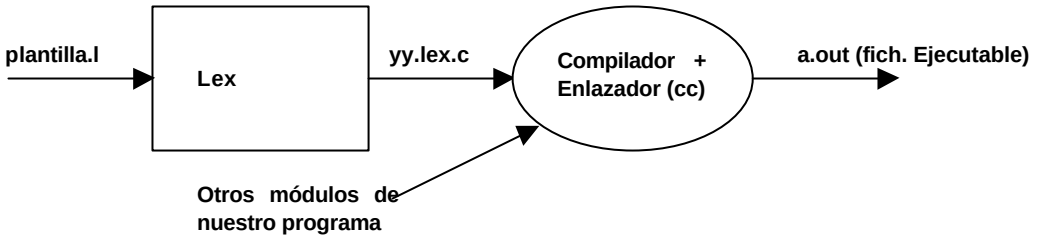
Introducción

Lex es una herramienta de los sistemas UNIX/Linux que nos va a permitir generar código C que luego podremos compilar y enlazar con nuestro programa.

La principal característica de Lex es que nos va a permitir asociar acciones descritas en C, a la localización de las Expresiones Regulares que le hayamos definido. Para ello Lex se apoya en una plantilla que recibe como parámetro, y que deberemos diseñar con cuidado.

Internamente Lex va a actuar como un autómata que localizará las expresiones regulares que le describamos, y una vez reconocida la cadena representada por dicha expresión regular, ejecutará el código asociado a esa regla.

Externamente podemos ver a Lex como una caja negra con la siguiente estructura:



En las siguientes secciones veremos todo lo correspondiente a la descripción de la plantilla, y el proceso de compilación.

La plantilla de Lex

La plantilla en la que Lex se va a apoyar para generar el código C, y donde nosotros deberemos describir toda la funcionalidad requerida, va a ser un fichero de texto plano con una estructura bien definida, donde iremos describiendo las expresiones regulares y las acciones asociadas a ella.

La estructura de la plantilla es la siguiente:

```
Declaraciones

%%

Reglas

%%

Procedimientos de Usuario
```

Se compone de tres secciones con estructuras distintas y claramente delimitadas por una línea en la que lo único que aparece es el carácter doble %.

Las secciones de ‘Declaraciones’ y la de ‘Procedimientos de Usuario’ son opcionales, mientras que la de ‘Reglas’ es obligatoria (aunque se encuentre vacía), con lo que tenemos que la plantilla más pequeña que podemos definir es:

```
%%
```

Esta plantilla, introducida en Lex, generaría un programa C donde el contenido de la entrada estándar sería copiado en la salida estándar por la aplicación de las reglas y acciones por defecto.

El formato de cada una de las secciones lo discutiremos en los apartados siguientes.

Lex va a actuar como un pre-procesador que va a trasformar las definiciones de esta plantilla en un fichero de código C.

La Sección de Declaraciones

En la sección de Declaraciones podemos encontrarnos con 3 tipos de declaraciones bien diferenciados:

- Un bloque donde le indicaremos al pre-procesador que lo que estamos definiendo queremos que aparezca ‘tal cual’ en el fichero C generado. Es un bloque de copia delimitado por las secuencias ‘%{‘ y ‘%}’ donde podemos indicar la inclusión de los ficheros de cabecera necesarios, o la declaración de variables globales, o declaraciones procedimientos descritos en la sección de Procedimientos de Usuario:

```
%{
/* Este bloque aparecerá tal cual en el fichero yy.lex.c */
#include <stdio.h>
#include <stdlib.h>
#define  VALUE      33
int nl, np, nw
%}
```

- Un bloque de definición de ‘alias’, donde ‘pondremos nombre’ a algunas de las expresiones regulares utilizadas. En este bloque, aparecerá AL COMIENZO DE LA LÍNEA el nombre con el que bautizaremos a esa expresión regular y SEPARADO POR UN TABULADOR (al menos), indicaremos la definición de la expresión regular. Para utilizar dichos nombres en vez de las expresiones regulares debemos escribirlos entre llaves:

```
entero          [0-9]+
real            {entero}.{entero}
real2           {real}[eE][\+\-]?{entero}
numero          {entero}|{real}|{real2}
```

- Un bloque de definición de las condiciones de arranque del autómata. (Ver apartado de Condiciones sobre Reglas).

```
%s normal
```

Estos bloques pueden aparecer en cualquier orden, y pueden aparecer varios de ellos a lo largo de la sección de declaraciones. Recordemos que esta sección puede aparecer vacía.

La Sección de Reglas

En la sección de Reglas sólo permitiremos un único tipo de escritura. Las reglas se definen como sigue:

```
Exp.reg.tab->       {acciones escritas en C}
```

AL COMIENZO DE LA LÍNEA se indica la expresión regular, seguida inmediatamente por uno o varios TABULADORES, hasta llegar al conjunto de acciones en C que deben ir encerrados en un bloque de llaves.

A la hora de escribir las expresiones regulares podemos hacer uso de los acrónimos dados en la sección de Declaraciones, escribiéndolos entre llaves, y mezclándolos con la sintaxis general de las expresiones regulares.

Si las acciones descritas queremos que aparezcan en varias líneas debido a su tamaño, debemos comenzar cada una de esas líneas con al menos un carácter de tabulación.

Si queremos incorporar algún comentario en C en una o varias líneas debemos comenzar cada una de esas líneas con al menos un carácter de tabulación.

```
[a-zA-Z_][a-zA-Z0-9_]*    { printf (“ID: %s”, yytext); }
{numero}                  { printf (“NUM: %s”, yytext);}
^{numero}                 { printf (“NUM al ppio linea”);}
```

Como normas para la identificación de expresiones regulares, Lex sigue las siguientes:

- Siempre intenta encajar una expresión regular con la cadena más larga posible,
- En caso de conflicto entre expresiones regulares (pueden aplicarse dos o más para una misma cadena de entrada), Lex, se guía por estricto orden de declaración de las reglas.

Existe una regla por defecto, que es:

```
.                   {ECHO;}
```

es decir, que en caso de que la entrada no encaje con ninguna de las reglas, se aplicará esta: ‘cualquier carácter’->’lo imprime en la salida estándar’. Si queremos modificar este comportamiento tan solo debemos sobrescribir la regla ‘.’ con la acción deseada ({} si no queremos que haga nada).

```
.                   { }
```

La Sección de Procedimientos de Usuario

En la sección de Procedimientos de Usuario escribiremos en C sin ninguna restricción aquellos procedimientos que hayamos necesitado en la sección de Reglas. Todo lo que aparezca en esta sección será incorporado 'tal cual' al final del fichero *yy.lex.l*.

No debemos olvidar como concepto de C, que si la implementación de los procedimientos se realiza 'después' de su invocación (en el caso de Lex, lo más probable es que se hayan invocado en la sección de reglas), debemos haberlos declarado previamente. Para ello no debemos olvidar declararlos en la sección de Declaraciones.

```
%{
…
int func1 (int param);
void proc1 ();
}%
```

Como función típica a ser descrita en una plantilla Lex, aparece el método principal (main). Si no se describe ningún método *main* Lex incorpora uno por defecto donde lo único que se hace es fijar el fichero de entrada como la entrada estándar e invocar a la herramienta para que comience su procesamiento:

```
int main ()
{
  yyin = stdin;
  yylex ();
}
```

Un ejemplo de método *main* típico es aquel que acepta un nombre de fichero como fichero de entrada:

```
int main (int argc, char *argv[])
{
  if (argc == 2)
  {
    yyin = fopen (argv[1], "rt");
    if (yyin == NULL)
    {
      printf ("El fichero %s no se puede abrir\n", argv[1]);
      exit (-1);
    }
  }
  else yyin = stdin;
  yylex ();
  return 0;
}
```

Las Variables, Funciones, Procedimientos y Macros internas de Lex

Como hemos ido reflejando en algunos ejemplos, Lex incorpora algunas variables, funciones, procedimientos y macros que nos permiten realizar de una forma más sencilla todo el procesamiento.

Como variables, las más utilizadas son:

Variable	Tipo	Descripción
yytext	char * o char []	Contiene la cadena de texto del fichero de entrada que ha encajado con la expresión regular descrita en la regla.
yylength	int	Longitud de yytext. yylength = strlen (yytext).
yyin	FILE *	Referencia al fichero de entrada.
yyval yylval	struct	Contienen la estructura de datos de la pila con la que trabaja YACC. Sirve para el intercambio de información entre ambas herramientas.

Como métodos (funciones y procedimientos), tenemos:

Método	Descripción
yylex ()	Invoca al Analizador Léxico, el comienzo del procesamiento.
yymore ()	Añade el yytext actual al siguiente reconocido.
yyless (n)	Devuelve los n últimos caracteres de la cadena yytext a la entrada.
yyerror ()	Se invoca automáticamente cuando no se puede aplicar ninguna regla.
yywrap ()	Se invoca automáticamente al encontrar el final del fichero de entrada.

Los procedimientos yyerror () e yywrap () pueden ser reescritos en la sección de Procedimientos de Usuario, cambiando de esta forma, el comportamiento ante la localización de errores, o del final del fichero de entrada.

Como macros y directivas característicos, existen:

Nombre	Descripción
ECHO	Escribe yytext en la salida estandar. ECHO = printf (“%s”, yytext)
REJECT	Rechaza la aplicación de la regla. Pone yytext de nuevo en la entrada y busca otra regla donde encajar la entrada. REJECT = yyless (yylength) + ‘Otra regla’
BEGIN	Fija nuevas condiciones para las reglas. (Ver Condiciones sobre Reglas).
END	Elimina condiciones para las reglas. (Ver Condiciones sobre Reglas).

Fijando Condiciones a las Reglas

Lex permite añadir condiciones de ejecución a las reglas. Esto es, condiciones bajo las cuales una regla puede ser ejecutada.

Esto es muy útil sobre todo cuando se da el caso de que unas reglas han de aplicarse bajo ciertas condiciones y no bajo otras. Por ejemplo la detección de palabras clave o de identificadores en un fichero de entrada con formato C: Si las palabras clave están dentro de un bloque de comentario o dentro de un par de comillas dobles (definición de constante de cadena), no han de interpretarse como palabras clave o identificadores, sino como componentes del comentario o de la constante:

```
if (i < 3) a++;
/* if (i < 3) a++; */
printf (“if (i < 3) a++;\n”);
```

Para ello, Lex incorpora la posibilidad de definir estas condiciones a las reglas, y los métodos (directivas) para cambiar de unas condiciones a otras.

Para definir que una regla está sujeta a una determinada condición, hay que anteceder dicha condición a la regla poniéndola entre pares ‘<>’:

```
<cond1 [, cond2]*>exp.reg.            { acciones }
```

Para indicar explícitamente que se ha de entrar en una condición, se utiliza la directiva BEGIN cond1 [, cond2]*; mientras que para salir de una condición se utiliza la directiva END cond1 [, cond2]*.

```
{id}                {  ECHO; }
“/*”                {  BEGIN comentario; }
<comentario>”*/”    {  END comentario; /* = BEGIN 0; */ }
<comentario>.       {  }
```

Las reglas que no son precedidas de ninguna condición se presupone que se ejecutan solamente baja la condición 0 (sin condiciones).

La invocación de BEGIN cambia la lista de condiciones por completo a la indicada (no las suma). Haciendo BEGIN 0, se vuelve al estado donde las reglas no necesitan ser precedidas de condiciones.

Para indicar las condiciones iniciales del procesamiento, se incorpora una sentencia con el formato siguiente en la sección de Declaraciones.

```
%start cond1 [, cond2]*         /* %start = %s */
```

El Proceso de Compilación

Una vez escrita la plantilla, invocamos a la herramienta mediante el comando de *shell*:

```
$ lex plantilla.l
```

Esto nos generará un fichero C, denominado *yy.lex.c* que contiene todo el código necesario para compilar nuestra aplicación.

Luego sólo queda compilar con las librerías adecuadas:

```
$ cc yy.lex.l -ll
```

En los sistemas gnu (Línux), existen ambas herramientas (lex y cc) pero denominadas con los nombres *flex, y gcc* aunque suelen incorporar enlaces a ellas a través de los nombres tradicionales.

Ejemplos

El siguiente ejemplo de plantilla, implementa el comando *wc* de lis sistemas UNIX / Linux, que nos permite contar los caracteres, palabras y líneas de un fichero.

```
%{
#include <stdio.h>
int nc, np, nl;
%}
%%
        /*----- Sección de Reglas ----------------*/
[^ \t\n]+           { np++; nc += yylength; }
[ \t]+              { nc += yylength; }
$                   { nl++; nc++; }
%%
        /*----- Sección de Procedimientos --------*/
int main (int argc, char *argv[])
{
  if (argc == 2)
  {
    yyin = fopen (argv[1], “rt”);
    if (yyin == NULL)
    {
      printf (“El fichero %s no se puede abrir\n”, argv[1]);
      exit (-1);
    }
  }
  else yyin = stdin;
  nc = np = nl = 0;
  yylex ();
  printf (“%d\t%d\t%d\n”, nc, np, nl);
  return 0;
}
```

Otro ejemplo interesante es el siguiente: Escribir un fichero HTML a partir de un fichero C, resaltando en otro color las palabras clave, las variables, las constantes numéricas y de cadena, y los comentarios.

```
%{
#include <stdio.h>

#define rojo        0xFF0000
#define verde       0x00FF00
#define azul        0x0000FF
#define morado      0xFF00FF
#define negro       0x000000

void escribirCabecera (char *nombre_fichero);
void cambiaClr (long color);
void nuevaLinea ();
void escribirFin ();
%}

id            [a-zA-Z_][a-zA-Z0-9_]*
entero        [0-9]+
real          {entero}.{entero}
real2         {real}[eE][\+\-]?{entero}
numero        {entero}|{real}|{real2}
pclaves       for|while|do|if|break|return

%%
        /*----- Sección de Reglas ----------------*/
{id}          { cambiaClr (azul);  ECHO; cambiaClr (negro); }
{numero}      { cambiaClr (verde); ECHO; cambiaClr (negro); }
\“            { cambiaClr (verde); ECHO; BEGIN cadena; }
<cadena>\”    { ECHO; cambiaClr (negro); END cadena; }
<cadena>.     { ECHO; }
“/*”          { cambiaClr (rojo);  ECHO; BEGIN coment; }
<coment>”*/”  { ECHO; cambiaClr (negro); END coment; }
<coment>.     { ECHO; }
{pclaves}     { cambiaClr (morado); ECHO; cambiaClr (negro); }
$             { nuevaLinea (); }
.             { ECHO; }
%%
        /*----- Sección de Procedimientos --------*/
int main (int argc, char *argv[])
{
  if (argc != 2)
  {
    printf (“Se necesita especificar un fichero de entrada\n”);
    exit (-1);
  }
  yyin = fopen (argv[1], “rt”);
  if (yyin == NULL)
  {
    printf (“El fichero %s no se puede abrir\n”, argv[1]);
    exit (-1);
  }

  escribirCabecera (argv[1]);
  yylex ();
  escribirFin ();
  return 0;
}
```

```
void escribirCabecera (char * fichero)
{
  printf (“<head>\n<title>%s</title>\n</head>\n”, fichero);
  printf (“\n<body>\n”\n);
}


void cambiaClr (long color)
{
  printf (“<font color=%x>”, color);
}


void nuevaLinea ()
{
  printf (“\n<br>\n”);
}


void escribirFin ()
{
  printf (“\n</body>”\n);
}
```

Lex como un Analizador Léxico

Lex puede verse desde otro punto de vista además de cómo una herramienta que nos permite ejecutar acciones tras la localización de cadenas de entrada que confrontan con expresiones regulares.

Lex puede ser visto como la primera etapa necesaria a la hora de elaborar un compilador, un intérprete, un emulador o cualquier otro tipo de herramienta que necesite procesar un fichero de entrada para poder cumplir su misión.

Supongamos que queremos diseñar una calculadora. Para ello tomaremos las expresiones a evaluar de un fichero de entrada (o de la misma entrada estándar). Diseñar un programa en C que tome una expresión de la entrada estándar y la evalúe, no es nada sencillo. Pensad que la entrada puede ser algo sencillo o algo tan complejo como la siguiente expresión: *"sqrt (2/3.141592654) * sin (0.707)"*

Lo primero que deberemos hacer es reconocer los elementos que forman dicha entrada. Es decir, las constantes numéricas, los operadores, las funciones predefinidas (seno, coseno…)… A estos elementos individuales que llevan asociado un sentido propio, los denominaremos **tokens**.

Los tokens nos dan toda la información necesaria de este primer nivel, pues lo mismo nos da que la constante numérica reconocida sea un 3 que un 300, el procesamiento va a ser el mismo. Lo importante es que es una constante numérica (token).

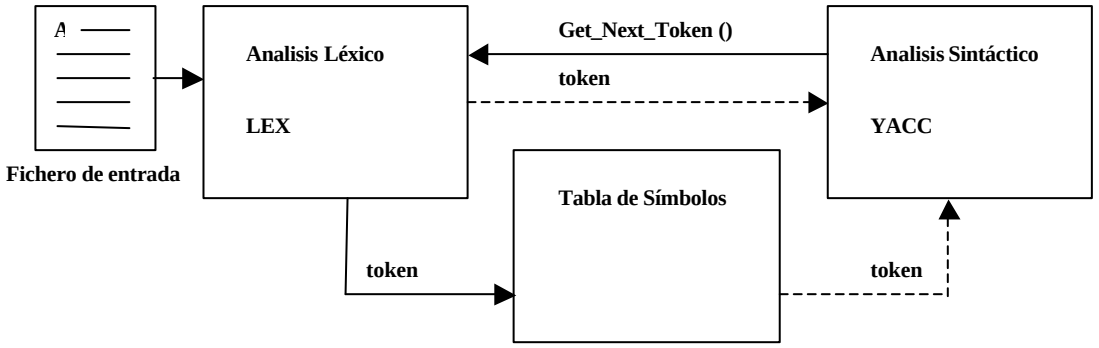
Esta etapa la podemos hacer con Lex sin ningún problema tal y como hemos visto. Tan solo debemos definir las reglas necesarias para reconocer esos tokens. A esta primera etapa se le da el nombre de **Análisis Léxico**, debido a que lo que el procesamiento fundamental se realiza en base a la detección de los lexemas (tokens) significativos.

Pero estos elementos individuales (token) no nos aportan toda la información necesaria para llevar a cabo el procesamiento final. Si nos fijamos sólo en los tokens, nos falta información: ¿Cómo se combinan los elementos? ¿Es posible la siguiente secuencia "+*2/(-4-*3)4sin"?. Desde el punto de vista de Lex, podemos reconocer todos los tokens correctamente, pero en cambio la sintaxis de la expresión es incorrecta. No tiene sentido.

Necesitamos entonces una segunda etapa que nos informe sobre la combinación de los elementos de entrada, y les de una interpretación según su combinación. Esta segunda etapa recibe el nombre de **Análisis Sintáctico**, donde lo importante el la estructura que tienen los datos (token) en la entrada.

Dicha estructura la vamos a definir entorno a un conjunto de reglas especiales, al que denominaremos 'gramática'. La herramienta que nos va a permitir llevar a cabo la implementación de esta segunda etapa va a ser YACC, que al igual que Lex, es un generador de código C basado en una plantilla similar a la de Lex, pero que va a permitirnos codificar estas nuevas reglas de combinación de los elementos.

Ambas etapas están muy interrelacionadas. Tal y como se ve en el esquema siguiente.



A continuación veremos cómo es el funcionamiento del sistema global. Para empezar diremos que el control del flujo pasa a la herramienta de Análisis Sintáctico, que es quien va a decidir lo que se hace en cada momento.

El analizador sintáctico va a solicitar al analizador léxico el siguiente token de la entrada. Con él va a poder decidir qué regla aplicar en cada caso, o bien, si no se puede aplicar ninguna regla, decidir que lo que ha ocurrido es un error sintáctico (gramatical).

El analizador léxico va a proporcionar cada token de la entrada bajo la petición del analizador sintáctico. Para ello el primer cambio que hay que hacer en la plantilla de Lex es incorporar sentencias de 'return' al final de cada regla que defina un token[1]:

```
exp.reg              { acciones; return token; }
```

Además del token, es importante que el analizador léxico mantenga otra serie de características, como por ejemplo el texto asociado al token (la cadena reconocida) porque aunque lo importante para el analizador sintáctico es la naturaleza de las entradas (tokens), no debemos olvidar que el resultado final ha de tener en cuenta qué valor es con el que hay que trabajar, o qué variable es a la que hacemos referencia…

Para ello entra en juego la **Tabla de símbolos**. En ella el analizador léxico deberá ir anotando cada nuevo símbolo reconocido, junto con sus propiedades (su naturaleza -> token, etc.) De esta forma, si un símbolo ya ha sido reconocido previamente, aparecerá en la tabla de símbolos y podemos adjudicarle la misma entrada.

Un ejemplo sencillo donde se ilustra el comportamiento del sistema y de la tabla de símbolos es el siguiente:

```
int a;
a = 0;
```

1) El analizador sintáctico (YACC) pide un token, y el analizador léxico (Lex) devuelve el token TOKEN_INT (palabra clave)

[1] En las herramientas que estamos utilizando los token se representan mediante constantes numéricas con valores por encima de 256. Cuando veamos YACC veremos que hay una forma de conseguir un listado con asignaciones automáticas de estos valores que podremos utilizar.

2) YACC pide el siguiente token (TOKEN_ID) y lo encaja en su gramática. YACC ya sabe que el ID reconocido es de tipo INTEGER. Puede ir a la tabla de símbolos y modificar su propiedad 'type' (por ejemplo) y asociarle el tipo entero. De esta forma cada vez que se haga referencia a 'a' sabremos que es un TOKEN_ID y es de tipo INTEGER. Y si en algún momento se intenta operar con 'a' utilizando otro tipo de datos, podemos saber que se comete un error. Este error es un error semántico, no sintáctico (la sintaxis gramatical puede ser correcta), pero ambos tipos de análisis pueden llevarse a cabo simultáneamente como se puede ver.

La estructura de la tabla de símbolos depende del tipo de procesamiento que queramos realizar. Entre sus campos podemos hacernos una idea de cuáles nos van a poder ser útiles para el procesamiento:

Campo	**Descripción**
Símbolo	Nombre del símbolo reconocido (p.e. 'a')
Tipo léxico	Tipo de token (p.e. 'TOKEN_ID')
Tipo sintáctico	Tipo lógico asignado por el contexto (p.e. 'INTEGER)
Tipo de retorno	En caso de funciones o arreglos, necesitaremos saber que tipo devuelve.
Contexto	Contexto sobre el cual se ha definido (p.e. la función donde se ha definido)
Valor del símbolo	Si estamos realizando un emulador o un intérprete, necesitaremos contar con el valor que se le ha asignado a cada símbolo. (p.e. '0')
…	…

Como hemos visto en el ejemplo, la tabla de símbolos es accesible a todos los analizadores, y cada uno va a tomar de ella los datos que necesita en cada caso, y va a aportar su granito de arena en la definición completa de ese símbolo.

Con respecto a qué símbolos han de introducirse en la tabla, diremos que tampoco hay una regla concreta. Podemos introducirlos todos, incluso los operadores o los signos de puntuación, o bien sólo introducir aquellos que nos sean útiles (identificadores y/o constantes).

Es recomendable que la tabla de símbolos esté indexada para poder realizar las búsquedas más rápidamente. Para ello debemos de tener en cuenta que un símbolo puede ser distinto aunque tenga el mismo nombre, si se define bajo contextos distintos (una variable global y una redefinición dentro de una función), con lo que la clave de búsqueda debería contemplar el contexto como componente junto con el nombre del símbolo.