Modelos de Computación

Práctica

Lex como localizador de expresiones regulares con acciones asociadas

1.- Introducción

Normalmente, *Lex* se usa en 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.

Otra utilidad de *Lex* consiste en ser una herramienta que nos permite ejecutar acciones tras la localización de cadenas de entrada que emparejan con expresiones regulares.

En esta prácticas nos centraremos en esta segunda utilidad de *Lex*. Se le pedirá al estudiante la realización de un trabajo práctico de procesamiento de un fichero que involucre el uso de *Lex* para localizar ciertas cadenas en el fichero y ejecutar una acción correspondiente con cada una de ellas. Por ejemplo, localizar y borrar direcciones de correo electrónico en una página web; cambiar de color los comentarios de un fichero con código fuente en C++; etc.

El alumno deberá plantear su propio trabajo práctico de procesamiento de ficheros usando *Lex*. A continuación se describirá una breve introducción sobre Lex y sus conceptos asociados que podrá servir de ayuda al estudiante para la realización de su práctica.

2.- Introducción a *Lex*

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.

Lex se apoya en una plantilla que recibe como parámetro y que deberemos diseñar con cuidado. En esta plantilla le indicaremos las expresiones regulares que debe localizar y las acciones asociadas a cada una de ellas.

A partir de esta plantilla, *Lex* genera código fuente en C. Este código contiene una función llamada yylex(), que localiza cadenas en la entrada que se ajustan a las expresiones regulares definidas, realizando entonces las acciones asociadas a dicho patrón.

3.- Expresiones regulares en Unix

Una expresión regular es un patrón que describe un conjunto de cadenas de caracteres. Por ejemplo, en MS-Dos el patrón aba*.txt describe el conjunto de cadenas de caracteres que comienzan con aba, contienen cualquier otro grupo de caracteres, luego un punto, y finalmente la cadena txt.

Una Expresión Regular en Unix nos sirve para definir lenguajes, imponiendo restricciones sobre las secuencias de caracteres que se permiten en este lenguaje. Por tanto una Expresión Regular estará formada por el conjunto de caracteres del alfabeto original, más un pequeño conjunto de caracteres extra (meta-caracteres) que nos permitirán definir estas restricciones.

El conjunto de metacaracteres para expresiones regulares es el siguiente:

\ ^ $. [ ] { } | ( ) * + ?

Estos caracteres, en una expresión regular, son interpretados de una manera especial y no como los caracteres que normalmente representan. Una búsqueda que implique alguno de estos caracteres obligará a usar el carácter \ . Por ejemplo, en una expresión regular, el caracter '.' representa "un caracter cualquiera"; si escribimos \. , estamos representando el caracter '.' tal cual, sin significado adicional.

Expresión Regular	**Significado**
Caracteres normales	Ellos mismos
.	Un carácter cualquiera excepto el carácter fin de línea
r*	r debe aparecer cero o más veces
r+	r debe aparecer una o más veces
r?	r debe aparecer cero o una vez
$r_1 \mid r_2$	La expresión regular r_1 o la r_2
^	Comienzo de línea
$	Fin de línea
-	Dentro de un conjunto de caracteres escrito entre corchetes, podemos especificar un rango (ej. [a-zA-Z0-9].
[...]	Un carácter cualquiera de los incluidos entre corchetes; acepta intervalos del tipo a-z, 0-9 A-Z, etc.
[^...]	Cualquier carácter menos el indicado o indicados entre corchetes; acepta intervalos del tipo a-z, 0-9 A-Z, etc.
(...)	Agrupación de los elementos dentro del paréntesis
\n	Carácter de salto de línea, fin de línea
\t	Carácter de tabulación
r_1r_2	La expresión regular r_1 seguida de la expresión regular r_2
r{n}	*n* ocurrencias de la expresión regular r
r{n,}	*n* o más ocurrencias de la expresión regular r
r{,m}	Cero o a los sumo *m* ocurrencias de la expresión regular r
r{n,m}	*n* o más ocurrencias de la expresión regular r, pero a lo sumo *m*
{*nombre*}	Se sustituye la expresión regular llamada *nombre*
"..."	Los caracteres entre comillas literalmente
\b	Indica un espacio en blanco

metacarácter	Literalmente el metacarácter

Ejemplos de expresiones regulares

Expresión Regular	**Significado**
a.b	axb aab abb aSb a#b ...
a..b	axxb aaab abbb a4$b ...
[abc]	a b c (cadenas de un carácter)
[aA]	a A (cadenas de un carácter)
[aA][bB]	ab Ab aB AB (cadenas de dos caracteres)
[0123456789]	0 1 2 3 4 5 6 7 8 9
[0-9]	0 1 2 3 4 5 6 7 8 9
[A-Za-z]	A B C ... Z a b c ... z
[0-9][0-9][0-9]	000 001 .. 009 010 .. 019 100 .. 999
[0-9]*	Cadena vacía 0 1 9 00 99 123 456 999 9999 ...
[0-9][0-9]*	0 1 9 00 99 123 456 999 9999 99999 99999999 ...
^1	Símbolo ‘1’ al comienzo de una línea
^[12]	Símbolo ‘1’ o ‘2’ al comienzo de una línea
^[^12]	Cualquier símbolo al comienzo de una línea menos ‘1’ o ‘2’
(123|124)$	Cadenas “123” o “124” al final de una línea
[0-9]+	0 1 9 00 99 123 456 999 9999 99999 99999999 ..
[0-9]?	Cadena vacía 0 1 2 .. 9
(ab)*	Cadena vacía ab abab ababab ...
^[0-9]?b	Cadenas b 0b 1b 2b ... 9b al comienzo de una línea
([0-9]+ab)*	Cadena vacía 1234ab 9ab9ab9ab 9876543210ab 99ab99ab ...
(ab|cd+)?(ef)*	Cadena vacía abef cdef cdddd efefef
[xy]	Representa el carácter ‘x’ o el carácter ‘y’
[^abc]	Representa cualquier carácter excepto ‘a’, ‘b’ o ‘c’
“$”	Carácter ‘$’, no indica fin de línea

4.- Estructura de un fichero *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 la cadena "%%".

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.

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 encontrar dos 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 constantes, o declarar la cabecera de procedimientos descritos en la sección "Procedimientos de Usuario". Por ejemplo:

```
%{
/* Este bloque aparecerá tal cual en el fichero lex.yy.c */

#include <stdio.h>
#include <stdlib.h>
#define VALUE  3

int  nl, np, nw;
void escribir_datos (int dato1, int dato2, int dato3);

%}
```

- 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. Por ejemplo:

```
letra       [a-zA-Z]
digito      [0-9]
entero      {digito}+
real        {entero}.{entero}
real2       {real}[eE][\+\-]?{entero}
numero      {entero}|{real}|{real2}
```

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.

Estos 'alias' deben de ir entre llaves para su posterior uso. Por ejemplo, {numero}

La sección de Reglas

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

Expresión_Regular Tabulador {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. Por ejemplo, *^{numero}*.

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.

Un ejemplo del contenido de la sección de Reglas podría ser:

{letra}({letra}|{digito}|_)*	{ printf (“ID: %s”, yytext); }
{numero}	{ printf (“NUM: %s”, yytext);}
^{numero}	{ printf (“NUM al ppio linea”);}

Nota: la expresión {letra}({letra}|{digito}|_)* es equivalente a [a-zA-Z_][a-zA-Z0-9_]*

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;}

Esta regla se aplica en el caso de que la entrada no encaje con ninguna de las reglas. Lo que hace es imprimir en la salida estándar el carácter que no encaja con ninguna regla. 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 lex.yy.c.

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. Por ejemplo:

```
%{
…
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, esto es:

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

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

```
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;
}
```

5.- Variables, funciones, procedimientos y macros de *Lex*

Variables

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.
yyleng	int	Longitud de yytext.
yyin	FILE *	Referencia al fichero de entrada.

Métodos

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.

Macros

Nombre	Descripción
ECHO	Escribe yytext en la salida estandar.
REJECT	Rechaza la aplicación de la regla. Pone yytext de nuevo en la entrada y busca otra regla donde encajar la entrada.

6.- Ejemplo

El siguiente ejemplo de plantilla nos permite contar los caracteres, palabras y líneas de un fichero:

```
            /*----- Sección de Declaraciones --------------*/

%{
#include <stdio.h>

int  nc, np, nl;
void escribir_datos (int dato1, int dato2, int dato3);
%}

%%
            /*----- Sección de Reglas ----------------*/

[^ \t\n]+           { np++; nc += yyleng; }
[ \t]+              { nc += yyleng; }
\n                  { 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 ();
escribir_datos(nc,np,nl);

return 0;
}

void escribir_datos (int dato1, int dato2, int dato3) {
  printf  ("Num_caracteres=%d\tNum_palabras=%d\tNum_líneas=%d\n",
                                                    dato1,dato2,dato3);
}
```

7.- El proceso de compilación

Los pasos para obtener un fichero ejecutable usando *Lex* son los siguientes:

1. Crear la plantilla *Lex*. Por ejemplo, "plantilla.l".

2. Invocar la herramienta *Lex* mediante el comando shell de Unix:

   -) lex plantilla.l

   (Si en nuestra distribución no está disponible *lex* podemos invocar el comando *flex*).

   Este paso nos genera un fichero C, denominado "lex.yy.c" que contiene todo el código necesario para compilar nuestra aplicación.

3. Compilar el programa que genera *Lex* ("lex.yy.c") junto a las librerías adecuadas:
   -) gcc lex.yy.c –o prog –ll

   (gcc lex.yy.c –o prog –lfl en el caso de usar *flex*)

4. Ya sólo nos queda ejecutar el programa *prog* sobre un fichero de entrada.
   -) ./prog <Nombre_Fichero>

Tareas a realizar

1. Formar un grupo de trabajo compuesto por una, dos o tres personas.
2. Cada grupo de trabajo debe pensar un problema original de procesamiento de textos. Para la resolución de este problema debe ser apropiado el uso de *Lex*, o sea, se debe resolver mediante el emparejamiento de cadenas con expresiones regulares y la asociación de acciones a cada emparejamiento.
3. Cada grupo debe resolver el problema propuesto usando *Lex*. Se deberá realizar una memoria donde se presente una descripción del problema y su solución, además de entregar electrónicamente los ficheros de texto con la implementación de la solución.
4. Esta práctica deberá ser entregada antes del día 31 de Diciembre de 2020. Se entregará a través de la plataforma PRADO en un fichero .zip conteniendo todos los archivos de esta práctica. Sólo es necesario que lo entregue uno de los componentes del grupo.