Flex a la rápida

José Ignacio Medina

Octubre, 2008

Este documento se distribuye bajo una licencia **Creative Commons by-nc-sa 2.0**. Usted es libre de copiar, distribuir, comunicar y ejecutar públicamente este documento, y hacer documentos derivados, bajo las siguientes condiciones:

1. *Debes reconocer y citar la obra de la forma especificada por el autor o el licenciante.* En este caso, debes reconocer que la autoría del documento es mía y de los colaboradores, en caso de citarme en algún documento de tu autoría. No necesitas mi permiso explícito.

2. *No puedes utilizar esta obra para fines comerciales.*

3. *Si alteras o transformas esta obra, o generas una obra derivada, sólo puedes distribuir la obra generada bajo una licencia idéntica a ésta.*

Puedes contactarte conmigo a través del mail **kephalophoros@gmail.com**.

Si es que dispones de ejemplos como los descritos en este documento, y que quieras compartir con los demás, no dudes en enviármelos. Obviamente, te agregaré como colaborador al documento.

Índice general

1. Teoría **3**

1.1. ¿Qué es flex? 3

1.2. ¿Qué necesitamos? 4

1.3. ¿Cómo empezamos a trabajar? 6

2. Práctica **9**

2.1. Simulando un DFA 9

2.2. Reconociendo un texto simple 13

Capítulo 1

Teoría

1.1. ¿Qué es flex?

Flex es una herramienta para generar scanners (reconocedores). Un scanner es un programa que reconoce patrones léxicos en un texto. Flex recibe desde la línea de comandos una descripción del scanner a ser implementado desde un archivo especificado por el usuario. La descripción del scanner está formada por pares de expresiones regulares y código C, llamadas reglas.

Flex genera el código C equivalente a la descripción pedida por el usuario, código que está contenido en el archivo fuente "lex.yy.c". Este archivo contiene el código fuente de un autómata finito determinista, capaz de reconocer las expresiones regulares entregadas por el usuario en el archivo de entrada. Después de compilar el archivo, obtendremos un ejecutable que contendrá a nuestro reconocedor. Cuando es ejecutado, el reconocedor analizará su entrada (ya sea un archivo o texto por teclado) en busca de texto que calce con las expresiones regulares definidas en el archivo inicial. Cada vez que el reconocedor encuentre un calce entre un texto leído y una regla existente, se ejecutará el código en C correspondiente a esa regla.

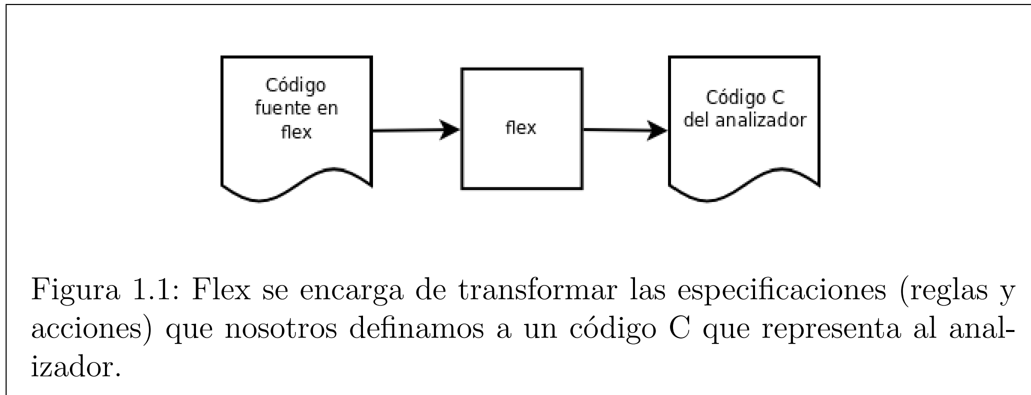

Figura 1.1: Flex se encarga de transformar las especificaciones (reglas y acciones) que nosotros definamos a un código C que representa al analizador.

Lo anteriormente descrito se puede entender mejor observando la Figura 1.1, la cual nos muestra el proceso necesario para obtener el código fuente del analizador.

Una vez que tenemos el código fuente del analizador, debemos compilarlo para obtener el ejecutable. Para esto, necesitamos un compilador de C/C++. Una vez que compilemos las fuentes, obtendremos un ejecutable que, dependiendo cómo fue programado desde flex, recibirá un archivo de entrada para ser analizado, o simplemente recibirá expresiones desde la línea de comandos, esperando a encontrar calces y ejecutando acciones cuando corresponda. El proceso completo puede entenderse mejor observando la Figura 1.2, en la cual se muestran los pasos a seguir para obtener un ejecutable desde un archivo fuente en flex.

1.2. ¿Qué necesitamos?

Por lo visto anteriormente, podemos deducir algunas cosas que necesitaremos antes de ponernos a trabajar. Condiciones obvias para trabajar con flex, además de las ganas y/o la obligación, serían:

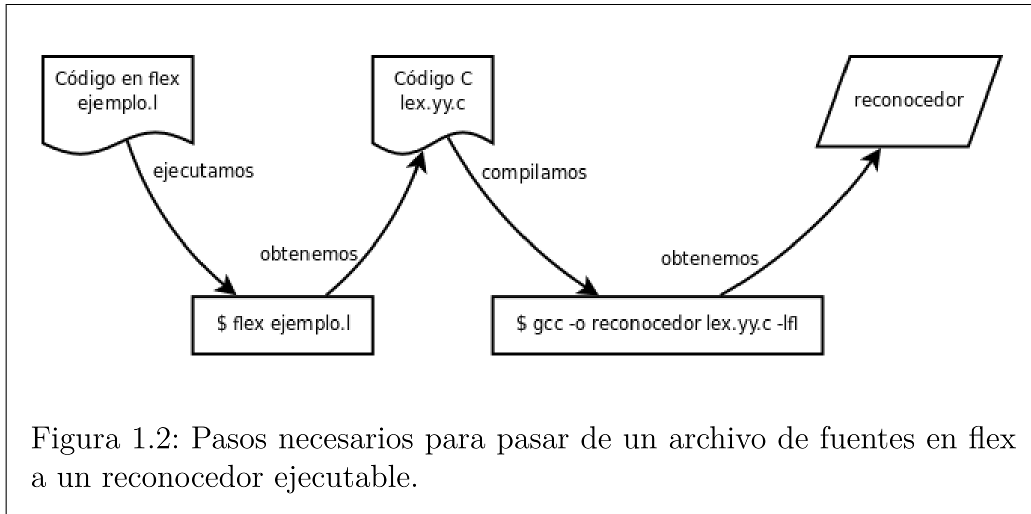

Figura 1.2: Pasos necesarios para pasar de un archivo de fuentes en flex a un reconocedor ejecutable.

1. Obviamente, flex instalado en nuestro computador. Los ejemplos del presente documento se procesaron con flex 2.5.34.
2. Un compilador de C/C++. En nuestro caso, usamos *gcc*, el cual viene incluído en casi todas las distribuciones de Linux. La versión utilizada es gcc 4.2.3.
3. Un editor de texto, ojalá con resaltado de sintaxis. Como recomendación pueden usar *gedit*, *emacs*, *kite*, etc.
4. Un sistema operativo en donde correr todo lo anterior. Es más fácil trabajar sobre Linux, ya que la instalación de software (y de sus dependencias) es más ordenada. En nuestro caso, usamos Ubuntu Hardy 8.04, con un kernel 2.6.24-21-generic.

Para empezar a trabajar, primero debemos cumplir con todos los requisitos descritos anteriormente. Una vez que tenemos Ubuntu instalado, podemos instalar todo lo necesario haciendo un

```
$ sudo aptitude install gcc flex
```

lo cual instalará los programas y todas las dependencias necesarias (incluso las recomendadas).

1.3. ¿Cómo empezamos a trabajar?

Para empezar a trabajar, debemos abrir un editor de texto y empezar a escribir las reglas de nuestro reconocedor. Para esto abrimos gedit, y escribimos lo siguiente:

```
/* definiciones */
%%
/* reglas */
%%
/* código de usuario */
```

En la *sección de definiciones* podemos incluir código C para declarar variables, funciones, definiciones de tipos, include's, etc. También podemos darles nombres a expresiones regulares que nosotros definamos (después veremos como hacer esto).

La *sección de reglas* está compuesta por dos columnas; una de patrones y otra de acciones. Por cada patrón hay asociada una o más acciones, en donde patrones son expresiones regulares reconocidas, y acciones son códigos C definidos por el programador. Si existe una cadena que sea coincidente con más de un patrón, se escoge la que tenga un mayor número de coincidencias.

La *sección de código de usuario* contiene sentencias creadas por necesidad del programador, por ejemplo, código para la lectura de un fichero de texto.

Métodos, macros y variables	Acción o finalidad
variable **yytext**	almacena el token actual
variable **yyleng**	almacena la longitud de yytext
macro **ECHO**	muestra yytext en stdout
método **yylex()**	inicia el reconocimiento de lexemas

Cuadro 1.1: Métodos y macros frecuentes usados en flex.

Por mientras, guardemos este archivo con el nombre “ejemplo.l”. Luego lo llenaremos de reglas y códigos en el capítulo de práctica.

Primero, hay algunas cosas que debemos aprender de flex antes de ponernos a escribir código. Flex dispone de métodos y variables que hacen más fácil el análisis de las palabras reconocidas por el autómata. Por ejemplo, *yytext* contiene un puntero global al último lexema reconocido. Así mismo, *yyleng* contiene la longitud del lexema reconocido. *yyin* e *yyout* se encargan de los procesos de entrada y salida, respectivamente (p.e. al trabajar con ficheros, *yyin* se utiliza para el archivo de entrada). La función *yylex()* inicia el proceso de reconocimiento de lexemas, por lo cual debería ser lo primero que ejecutemos una vez que abramos el archivo de entrada. Una tabla de los principales métodos y macros usados en flex puede verse en el Cuadro 1.1.

Con respecto al archivo de entrada de flex, necesitamos saber algunas cosas respecto a las reglas y acciones que podemos utilizar. Como acción, podremos utilizar cualquier cantidad de código C, según sean nuestras necesidades. El código para las acciones puede utilizar los métodos y variables mencionadas en el párrafo anterior.

En el caso de las reglas, los patrones por los cuales se buscarán las palabras

Expresión regular	Significado
.	cualquier cosa excepto un salto de línea \n
e*	ninguna o cualquier número de e
e+	una o cualquier número de e
[abc...n]	a ∨ b ∨ c ∨ ... ∨ n (conjunción)
a-e	cualquier caracter de {a, b, c, d, e}
^e	e debe estar al principio de la línea
[^abc...n]	cualquier caracter excepto a, b, c ...
e$	e aparece al final de la línea
e{a,b}	entre a y b concatenaciones de e

Cuadro 1.2: Un subconjunto de las expresiones regulares disponibles en flex.

estarán definidos por expresiones regulares, algunas de las cuales se describen, junto con su significado, en el Cuadro 1.2.

Algo que siempre debemos recordar es que cuando más de alguna expresión regular sirva para describir una palabra, flex elije la expresión que tenga el mayor número de caracteres coincidentes. Si es que eso no soluciona el problema, flex opta por la regla escrita primero en el archivo fuente.

Un listado más amplio de expresiones regulares reconocidas por flex, y las variables, macros y métodos que implementa, puede ser encontrado en el manual oficial de la herramienta[1].

[1]Flex, versión 2.5: Un generador de analizadores léxicos rápidos, de Vern Paxson

Capítulo 2

Práctica

Ya con un poco de teoría encima, podemos empezar a probar con algunos ejemplos. Como dijimos en el Capítulo 1, la idea es que tengas los programas instalados, y un archivo de texto abierto con las secciones de definiciones, reglas y código de usuario comentadas.

Si es que hay algo que no entiendas, o que no te haya quedado claro del Capítulo 1, lee detenidamente la documentación oficial de flex. En todo caso, cada vez que en los ejemplos se ocupen comandos nuevos o algún método no explicado anteriormente, se hará una breve explicación de su uso.

2.1. Simulando un DFA

En este ejemplo buscaremos reproducir un DFA, es decir, un autómata finito determinista. El DFA a simular es el de la Figura 2.1. La idea es que nuestro programa sea capaz de recibir por la línea de comandos una palabra, y en caso de que ocurra una condición de término una vez que hayamos terminado

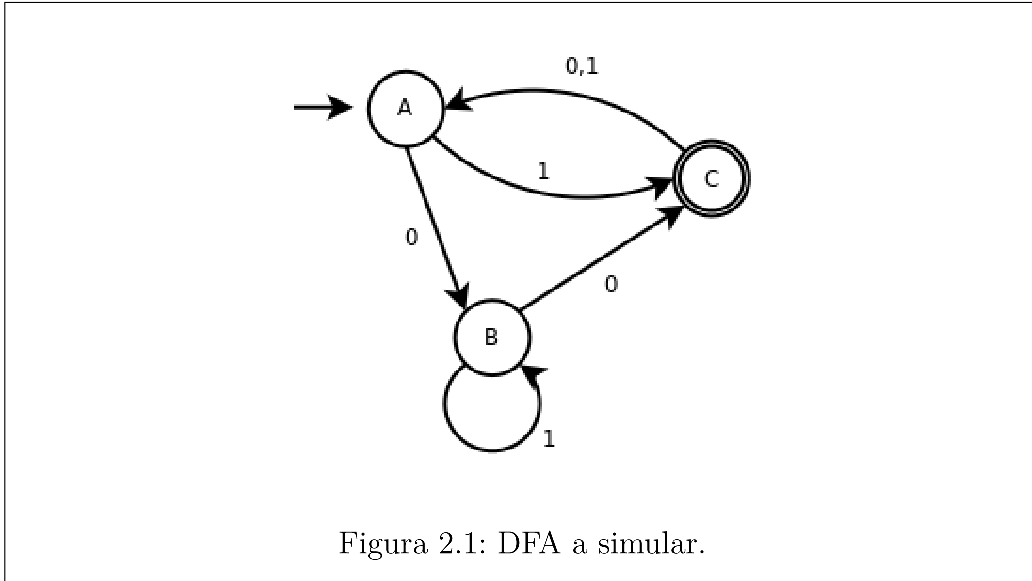

Figura 2.1: DFA a simular.

de escribir nuestra palabra (una vez que oprimamos ENTER), el programa nos devuelva la palabra reconocida por el DFA. En caso de no estar sobre una condición de término (estado final de nuestro autómata), el programa da aviso y termina. Así mismo, si es que ingresamos un caracter no reconocido por el autómata, el programa da aviso y termina su ejecución.

El programa en flex que simula al autómata sería:

```
%{ /* programa en flex para simular un DFA */
%}

%x B C

%%
<INITIAL>0          {yymore(); BEGIN(B);}
<INITIAL>1          {yymore(); BEGIN(C);}
<INITIAL>\n         {printf("No estás en un estado final\n");
```

```
                     exit(1);}
<INITIAL>[^01]   {printf("Sólo ceros y unos!\n");
                     exit(1);}
<B>0                 {yymore(); BEGIN(C);}
<B>1                 {yymore(); BEGIN(B);}
<B>\n                {printf("No estás en un estado final\n");
                     exit(1);}
<B>[^01]             {printf("Sólo ceros y unos!\n");
                     exit(1);}
<C>0                 {yymore(); BEGIN(INITIAL);}
<C>1                 {yymore(); BEGIN(INITIAL);}
<C>\n                {printf("Palabra reconocida: %s\n", yytext);
                     exit(1);}
<C>[^01]             {printf("Sólo ceros y unos!\n");
                     exit(1);}
```

Ok, ahora copiamos el código a un procesador de textos, y lo guardamos como “DFA.l” en la carpeta en la que estemos trabajando. Ahora, desde consola, nos movemos a la carpeta de trabajo y ejecutamos

```
$ flex DFA.l
```

con lo que deberíamos recibir un archivo llamado “lex.yy.c”. Ahora debemos compilar este archivo, haciendo

```
$ gcc -o DFA lex.yy.c -lfl
```

y recibiendo de parte del compilador un ejecutable de nombre “DFA”. Ahora, sólo nos resta probar que realmente funciona nuestro simulador de DFA. En la consola, ejecutamos el simulador escribiendo

```
$ ./DFA
```

y empezamos a escribir palabras. Puedes probar con las siguientes palabras: “01110”, “101”, “0000001”, “001” y “1b0”. Observa los mensajes de aprobación o rechazo de las palabras, e infiere los estados por los que pasó el autómata hasta que presionaste ENTER: ¿Los resultados son los esperados?.

Con el conocimiento que tenemos hasta ahora, hay algunas cosas del código que no quedan totalmente claras, como por ejemplo:

¿Qué quiere decir la línea “ %x B C”?

Es la definición de las condiciones de arranque, en este caso, B y C.

La idea de una condición de arranque es poder definir reglas que se activen condicionalmente, lo cual es conveniente para nuestro problema ya que deseamos que después de cada consumo de un caracter pasemos a otro estado, que a su vez tiene otras condiciones para pasar de un estado a otro.

Para saber más sobre las condiciones de arranque consulta el manual oficial de flex.

¿A qué se refiere con <INITIAL>?

Es el nombre de la primera condición de arranque por defecto. En el ejemplo, la hemos usado para representar el estado A (el estado inicial del DFA).

¿Para qué sirve “yymore()”?

yymore() le dice al reconocedor que la próxima vez que haga calzar una regla, el token correspondiente debe ser concatenado como prefijo al valor actual de yytext().

2.2. Reconociendo un texto simple

Se nos presenta el problema de reconocer un texto simple, con el fin de calcular la cantidad de palabras presentes en el texto, y la cantidad de líneas de éste. Tomemos como ejemplo el siguiente texto:

```
Las incoloras ideas verdes duermen furiosamente.
Pobre de aquel individuo que se atreva a despertarlas.
```

Primero, abrimos un editor de texto y copiamos el texto de ejemplo en él. Luego, lo grabamos con el nombre “ejemplo.txt”. Ahora, debemos crear un nuevo documento, y escribir un código parecido a éste:

```
%{ unsigned contadorPalabras = 0, contadorLineas = 1; %}

letra [a-zA-ZáéíóúÁÉÍÓÚ]
palabra {letra}+
espacio [ ]+
puntuacion [.,()¡!¿?:;]
eol \n

%%

{palabra}        contadorPalabras++;
{espacio}        /* nada */
{puntuacion}     /* nada */
{eol}            contadorLineas++;
.                {printf("%s no es válido\n", yytext);
                  printf("Sólo letras y puntuación.");
                  exit(1);}

%%
```

```
main(argc,argv)
int argc;
char **argv;
{
 if (argc > 1) {
      FILE *file;
      file = fopen(argv[1], "r");
      if (!file) {
           fprintf(stderr, "Error en archivo\n");
           exit(1);
      }
      yyin = file;
 }
 yylex();
 printf("El archivo tiene:\n");
 printf("%d palabras\n", contadorPalabras);
 printf("%d líneas\n", contadorLineas);
 return 0;
}
```

Lo guardamos con el nombre “textoSimple.l”. Luego, siguiendo los pasos del ejemplo anterior, hacemos:

```
$ flex textoSimple.l
$ gcc -o leer lex.yy.c -lfl
$ ./leer ejemplo.txt
```

Con lo que obtenemos como resultado que el texto está compuesto por 15 palabras y 2 líneas.

Este ejemplo nos permite observar que pueden generarse reglas en flex de una

gran complejidad: con las reglas para expresiones regulares que conocemos podríamos reconocer textos mucho más complejos, como un formulario de registro o una ficha médica. Ya sabemos formar palabras, y bajo reglas similares podemos formar números enteros, decimales, fracciones, RUT's, nombres con números en ellos (por ejemplo, variables en lenguajes de programación), operaciones matemáticas complejas, uso de paréntesis, etc.

Debemos darnos cuenta de que lo que reconocerá nuestro autómata será estrictamente lo que nosotros le ordenemos que reconozca. Por ejemplo, el texto de ejemplo puede haber estado compuesto por las siguientes líneas

```
asdf qwerty
LOL ROFLMAO
cuak!
```

y seguir siendo perfectamente "legal", ya que nuestra definición de reglas lo permite. En ningún caso estamos analizando si las palabras reconocidas tienen "significado", solamente estamos verificando su "legalidad a nivel léxico".