Flex, versión 2.5

Un generador de analizadores léxicos rápidos.
Edición 2.5, Abril 1995

Vern Paxson

1 Introducción

Este manual describe `flex`, una herramienta para la generación de programas que realizan concordancia de patrones en texto. El manual incluye a la vez secciones de tutorial y de referencia:

Descripción
: una breve introducción a la herramienta

Algunos Ejemplos Simples

Formato del Fichero de Entrada

Patrones
: las expresiones regulares extendidas que utiliza flex

Cómo se Empareja la Entrada
: las reglas para determinar lo que ha concordado

Acciones
: cómo especificar qué hacer cuando concuerde un patrón

El Escáner Generado
: detalles respecto al escáner que produce flex; cómo controlar la fuente de entrada

Condiciones de Arranque
: la introdución de contexto en sus escáneres, y conseguir "mini-escáneres"

Múltiples Buffers de Entrada
: cómo manipular varias fuentes de entrada; cómo analizar cadenas en lugar de ficheros.

Reglas de Fin-de-Fichero
: reglas especiales para reconocer el final de la entrada

Macros Misceláneas
: un sumario de macros disponibles para las acciones

Valores Disponibles para el Usuario
: un sumario de valores disponibles para las acciones

Interfaz con Yacc
: conectando escáneres de flex junto con analizadores de yacc

Opciones
: opciones de línea de comando de flex, y la directiva "%option"

Consideraciones de Rendimiento
: cómo hacer que sus analizadores vayan tan rápido como sea posible

Generando Escáneres en C++
: la facilidad (experimental) para generar analizadores léxicos como clases de C++

Incompatibilidades con Lex y POSIX
: cómo flex difiere del lex de AT&T y del lex estándar de POSIX

Diagnósticos
: esos mensajes de error producidos por flex (o por los escáneres que este genera) cuyo significado podría no ser evidente

Ficheros
: los ficheros usados por flex

Deficiencias / Errores
: problemas de flex conocidos

Ver También
: otra documentación, herramientas relacionadas

Autor
: incluye información de contacto

2 Descripción

`flex` es una herramienta para generar *escáneres*: programas que reconocen patrones léxicos en un texto. `flex` lee los ficheros de entrada dados, o la entrada estándar si no se le ha indicado ningún nombre de fichero, con la descripción de un escáner a generar. La descripción se encuentra en forma de parejas de expresiones regulares y código C, denominadas *reglas*. `flex` genera como salida un fichero fuente en C, '`lex.yy.c`', que define una rutina '`yylex()`'. Este fichero se compila y se enlaza con la librería '`-lfl`' para producir un ejecutable. Cuando se arranca el fichero ejecutable, este analiza su entrada en busca de casos de las expresiones regulares. Siempre que encuentra uno, ejecuta el código C correspondiente.

3 Algunos ejemplos simples

En primer lugar veremos algunos ejemplos simples para una toma de contacto con el uso de `flex`. La siguiente entrada de `flex` especifica un escáner que siempre que encuentre la cadena "username" la reemplazará por el nombre de entrada al sistema del usuario:

```
%%
username    printf( "%s", getlogin() );
```

Por defecto, cualquier texto que no reconozca el analizador léxico de `flex` se copia a la salida, así que el efecto neto de este escáner es copiar su fichero de entrada a la salida con cada aparición de "username" expandida. En esta entrada, hay solamente una regla. "username" es el *patrón* y el "printf" es la *acción*. El "%%" marca el comienzo de las reglas.

Aquí hay otro ejemplo simple:

```
        int num_lineas = 0, num_caracteres = 0;

%%
\n      ++num_lineas; ++num_caracteres;
.       ++num_caracteres;

%%
main()
        {
        yylex();
        printf( "# de líneas = %d, # de caracteres. = %d\n",
                num_lineas, num_caracteres );
        }
```

Este analizador cuenta el número de caracteres y el número de líneas en su entrada (no produce otra salida que el informe final de la cuenta). La primera línea declara dos variables globales, "num_lineas" y "num_caracteres", que son visibles al mismo tiempo dentro de '`yylex()`' y en la rutina '`main()`' declarada después del segundo "%%". Hay dos reglas, una que empareja una línea nueva ("\n") e incrementa la cuenta de líneas y la cuenta de caracteres, y la que empareja cualquier caracter que no sea una línea nueva (indicado por la expresión regular ".").

Un ejemplo algo más complicado:

```
/* escáner para un lenguaje de juguete al estilo de Pascal */

%{
/* se necesita esto para la llamada a atof() más abajo */
#include <math.h>
%}

DIGITO   [0-9]
ID       [a-z][a-z0-9]*

%%

{DIGITO}+   {
            printf( "Un entero: %s (%d)\n", yytext,
                    atoi( yytext ) );
            }
```

```
{DIGITO}+"."{DIGITO}*      {
            printf( "Un real: %s (%g)\n", yytext,
                    atof( yytext ) );
            }

if|then|begin|end|procedure|function        {
            printf( "Una palabra clave: %s\n", yytext );
            }

{ID}        printf( "Un identificador: %s\n", yytext );

"+"|"-"|"*"|"/"   printf( "Un operador: %s\n", yytext );

"{"[^}\n]*"}"     /* se come una linea de comentarios */

[ \t\n]+          /* se come los espacios en blanco */

.           printf( "Caracter no reconocido: %s\n", yytext );

%%

main( argc, argv )
int argc;
char **argv;
    {
    ++argv, --argc;  /* se salta el nombre del programa */
    if ( argc > 0 )
            yyin = fopen( argv[0], "r" );
    else
            yyin = stdin;

    yylex();
    }
```

Esto podría ser los comienzos de un escáner simple para un lenguaje como Pascal. Este identifica diferentes tipos de *tokens* e informa a cerca de lo que ha visto.

Los detalles de este ejemplo se explicarán en las secciones siguientes.

4 Formato del fichero de entrada

El fichero de entrada de `flex` está compuesto de tres secciones, separadas por una línea donde aparece únicamente un '`%%`' en esta:

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

La sección de *definiciones* contiene declaraciones de definiciones de *nombres* sencillas para simplificar la especificación del escáner, y declaraciones de *condiciones* de arranque, que se explicarán en una sección posterior. Las definiciones de nombre tienen la forma:

```
nombre definición
```

El "nombre" es una palabra que comienza con una letra o un subrayado ('`_`') seguido por cero o más letras, dígitos, '`_`', o '`-`' (guión). La definición se considera que comienza en el primer caracter que no sea un espacio en blanco siguiendo al nombre y continuando hasta el final de la línea. Posteriormente se puede hacer referencia a la definición utilizando "{nombre}", que se expandirá a "(definición)". Por ejemplo,

```
DIGITO   [0-9]
ID       [a-z][a-z0-9]*
```

define "DIGITO" como una expresión regular que empareja un dígito sencillo, e "ID" como una expresión regular que empareja una letra seguida por cero o más letras o dígitos. Una referencia posterior a

```
{DIGITO}+"."{DIGITO}*
```

es idéntica a

```
([0-9])+"."([0-9])*
```

y empareja uno o más dígitos seguido por un '`.`' seguido por cero o más dígitos.

La sección de *reglas* en la entrada de `flex` contiene una serie de reglas de la forma:

```
patrón   acción
```

donde el patrón debe estar sin sangrar y la acción debe comenzar en la misma línea.

Ver Capítulo 7 [Acciones], página 10, para una descripción más amplia sobre patrones y acciones.

Finalmente, la sección de código de usuario simplemente se copia a '`lex.yy.c`' literalmente. Esta sección se utiliza para rutinas de complemento que llaman al escáner o son llamadas por este. La presencia de esta sección es opcional; Si se omite, el segundo '`%%`' en el fichero de entrada se podría omitir también.

En las secciones de definiciones y reglas, cualquier texto *sangrado* o encerrado entre '`%{`' y '`%}`' se copia íntegramente a la salida (sin los %{}'s). Los %{}'s deben aparecer sin sangrar en líneas ocupadas únicamente por estos.

En la sección de reglas, cualquier texto o %{} sangrado que aparezca antes de la primera regla podría utilizarse para declarar variables que son locales a la rutina de análisis y (después de las declaraciones) al código que debe ejecutarse siempre que se entra a la rutina de análisis. Cualquier otro texto sangrado o %{} en la sección de reglas sigue copiándose a la salida, pero su significado no está bien definido y bien podría causar errores en tiempo de compilación (esta propiedad se presenta para conformidad con `POSIX`; para otras características similares) ver Capítulo 18 [Incompatibilidades con `lex` y POSIX], página 44)

En la sección de definiciones (pero no en la sección de reglas), un comentario sin sangría (es decir, una línea comenzando con "/*") también se copia literalmente a la salida hasta el próximo "*/".

5 Patrones

Los patrones en la entrada se escriben utilizando un conjunto extendido de expresiones regulares. Estas son:

'`x`' empareja el caracter '`x`'

'`.`' cualquier caracter (byte) excepto una línea nueva

'`[xyz]`' una "clase de caracteres"; en este caso, el patrón empareja una '`x`', una '`y`', o una '`z`'

'`[abj-oZ]`'
una "clase de caracteres" con un rango; empareja una '`a`', una '`b`', cualquier letra desde la '`j`' hasta la '`o`', o una '`Z`'

'`[^A-Z]`' una "clase de caracteres negada", es decir, cualquier caracter menos los que aparecen en la clase. En este caso, cualquier caracter EXCEPTO una letra mayúscula.

'`[^A-Z\n]`'
cualquier caracter EXCEPTO una letra mayúscula o una línea nueva

'*r*`*`' cero o más *r*'s, donde *r* es cualquier expresión regular

'*r*`+`' una o más *r*'s

'*r*`?`' cero o una *r* (es decir, "una *r* opcional")

'*r*`{2,5}`' donde sea de dos a cinco *r*'s

'*r*`{2,}`' dos o más *r*'s

'*r*`{4}`' exactamente 4 *r*'s

'`{nombre}`'
la expansión de la definición de "nombre" (ver más abajo)

'`"[xyz]\"foo"`'
la cadena literal: [xyz]"foo

'`\`*x*' si *x* es una '`a`', '`b`', '`f`', '`n`', '`r`', '`t`', o '`v`', entonces la interpretación ANSI-C de `\`*x*. En otro caso, un literal '*x*' (usado para indicar operadores tales como '`*`')

'`\0`' un caracter NUL (código ASCII 0)

'`\123`' el caracter con valor octal 123

'`\x2a`' el caracter con valor hexadecimal `2a`

'`(`*r*`)`' empareja una *R*; los paréntesis se utilizan para anular la precedencia (ver más abajo)

'*rs*' la expresión regular *r* seguida por la expresión regular *s*; se denomina "concatenación"

'*r*`|`*s*' bien una *r* o una *s*

'*r*/*s*' una *r* pero sólo si va seguida por una *s*. El texto emparejado por *s* se incluye cuando se determina si esta regla es el "emparejamiento más largo", pero se devuelve entonces a la entrada antes que se ejecute la acción. Así que la acción sólo ve el texto emparejado por *r*. Este tipo de patrones se llama "de contexto posterior". (Hay algunas combinaciones de *r*/*s* que flex no puede emparejar correctamente. Ver Capítulo 21 [Deficiencias / Errores], página 50, las notas a cerca del "contexto posterior peligroso".)

'^*r*' una *r*, pero sólo al comienzo de una línea (es decir, justo al comienzo del análisis, o a la derecha después de que se haya analizado una línea nueva).

'*r*$' una *r*, pero sólo al final de una línea (es decir, justo antes de una línea nueva). Equivalente a "*r*/\n".

Fíjese que la noción de flex de una "línea nueva" es exáctamente lo que el compilador de C utilizado para compilar flex interprete como '\n'; en particular, en algunos sistemas DOS debe filtrar los \r's de la entrada used mismo, o explícitamente usar *r*/\r\n para "r$".

'<*s*>*r*' una *r*, pero sólo en la condición de arranque *s* (Ver Capítulo 9 [Condiciones de arranque], página 16, para una discusión sobre las condiciones de arranque)

'<*s1*,*s2*,*s3*>*r*'
lo mismo, pero en cualquiera de las condiciones de arranque *s1*, *s2*, o *s3*

'<*>*r*' una *r* en cualquier condición de arranque, incluso una exclusiva.

'<<EOF>>' un fin-de-fichero

'<*s1*,*s2*><<EOF>>'
un fin-de-fichero en una condición de arranque *s1* ó *s2*

Fíjese que dentro de una clase de caracteres, todos los operadores de expresiones regulares pierden su significado especial excepto el caracter de escape ('\') y los operadores de clase de caracteres, '-', ']', y, al principio de la clase, '^'.

Las expresiones regulares en el listado anterior están agrupadas de acuerdo a la precedencia, desde la precedencia más alta en la cabeza a la más baja al final. Aquellas agrupadas conjuntamente tienen la misma precedencia. Por ejemplo,

```
foo|bar*
```

es lo mismo que

```
(foo)|(ba(r*))
```

ya que el operador '*' tiene mayor precedencia que la concatenación, y la concatenación más alta que el operador '|'. Este patrón por lo tanto empareja *bien* la cadena "foo" *o* la cadena "ba" seguida de cero o más r's. Para emparejar "foo" o, cero o más "bar"'s, use:

```
foo|(bar)*
```

y para emparejar cero o más "foo"'s o "bar"'s:

```
(foo|bar)*
```

Además de caracteres y rangos de caracteres, las clases de caracteres pueden también contener *expresiones* de clases de caracteres. Son expresiones encerradas entre los delimitadores '[:' y ':]' (que también deben aparecer entre el '[' y el ']' de la clase de caracteres; además pueden darse otros elementos dentro de la clase de caracteres). Las expresiones válidas son:

```
[:alnum:] [:alpha:] [:blank:]
[:cntrl:] [:digit:] [:graph:]
[:lower:] [:print:] [:punct:]
[:space:] [:upper:] [:xdigit:]
```

Todas estas expresiones designan un conjunto de caracteres equivalentes a la correspondiente función estándar '`isXXX`' de C. Por ejemplo, '`[:alnum:]`' designa aquellos caracteres para los cuales '`isalnum()`' devuelve verdadero —es decir, cualquier caracter alfabético o numérico. Algunos sistemas no ofrecen '`isblank()`', así que flex define '`[:blank:]`' como un espacio en blanco o un tabulador.

Por ejemplo, las siguientes clases de caracteres son todas equivalentes:

```
[[:alnum:]]
[[:alpha:][:digit:]]
[[:alpha:]0-9]
[a-zA-Z0-9]
```

Si su escáner ignora la distinción entre mayúsculas y minúsculas (la bandera '`-i`'), entonces '`[:upper:]`' y '`[:lower:]`' son equivalentes a '`[:alpha:]`'.

Algunas notas sobre los patrones:

- Una clase de caracteres negada tal como el ejemplo "[^A-Z]" anterior *emparejará una línea nueva* a menos que "\n" (o una secuencia de escape equivalente) sea uno de los caracteres presentes explícitamente en la clase de caracteres negada (p.ej., "[^A-Z\n]"). Esto es diferente a cómo muchas de las otras herramientas de expresiones regulares tratan las clases de caracteres negadas, pero desafortunadamente la inconsistencia está fervientemente enrraizada históricamente. Emparejar líneas nuevas significa que un patrón como [^"]* puede emparejar la entrada completa a menos que haya otra comilla en la entrada.
- Una regla puede tener lo más una instancia del contexto posterior (el operador '`/`' o el operador '`$`'). La condición de arranque, los patrones '`^`', y "<<EOF>>" pueden aparecer solamente al principio de un patrón, y, al igual que con '`/`' y '`$`', no pueden agruparse dentro de paréntesis. Un '`^`' que no aparezca al principio de una regla o un '`$`' que no aparezca al final de una regla pierde sus propiedades especiales y es tratado como un caracter normal.

  Lo siguiente no está permitido:

  ```
  foo/bar$
  <sc1>foo<sc2>bar
  ```

  Fíjese que la primera regla se puede escribir como "foo/bar\n".

  En el siguiente ejemplo un '`$`' o un '`^`' es tratado como un caracter normal:

  ```
  foo|(bar$)
  foo|^bar
  ```

  Si lo que se desea es un "foo" o un "bar" seguido de una línea nueva, puede usarse lo siguiente (la acción especial '`|`' se explica en la Capítulo 7 [Acciones], página 10.):

  ```
  foo      |
  bar$     /* la acción va aquí */
  ```

  Un truco parecido funcionará para emparejar un "foo" o, un "bar" al principio de una línea.

6 Cómo se empareja la entrada

Cuando el escáner generado está funcionando, este analiza su entrada buscando cadenas que concuerden con cualquiera de sus patrones. Si encuentra más de un emparejamiento, toma el que empareje más texto (para reglas de contexto posterior, se incluye la longitud de la parte posterior, incluso si se devuelve a la entrada). Si encuentra dos o más emparejamientos de la misma longitud, se escoge la regla listada en primer lugar en el fichero de entrada de `flex`.

Una vez que se determina el emparejamiento, el texto correspondiente al emparejamiento (denominado el *token*) está disponible en el puntero a caracter global `yytext`, y su longitud en la variable global entera `yyleng`. Entonces la *acción* correspondiente al patrón emparejado se ejecuta (Ver Capítulo 7 [Acciones], página 10, para una descripción más detallada de las acciones), y entonces la entrada restante se analiza para otro emparejamiento.

Si no se encuentra un emparejamiento, entonces se ejecuta la *regla por defecto*: el siguiente caracter en la entrada se considera reconocido y se copia a la salida estándar. Así, la entrada válida más simple de `flex` es:

```
%%
```

que genera un escáner que simplemente copia su entrada (un caracter a la vez) a la salida.

Fíjese que `yytext` se puede definir de dos maneras diferentes: bien como un *puntero* a caracter o como un *array* de caracteres. Usted puede controlar la definición que usa `flex` incluyendo una de las directivas especiales '`%pointer`' o '`%array`' en la primera sección (definiciones) de su entrada de flex. Por defecto es '`%pointer`', a menos que use la opción de compatibilidad '`-l`', en cuyo caso `yytext` será un array.

La ventaja de usar '`%pointer`' es un análisis substancialmente más rápido y la ausencia de desbordamiento del buffer cuando se emparejen tokens muy grandes (a menos que se agote la memoria dinámica). La desventaja es que se encuentra restringido en cómo sus acciones pueden modificar `yytext` (ver Capítulo 7 [Acciones], página 10), y las llamadas a la función '`unput()`' destruyen el contenido actual de `yytext`, que puede convertirse en un considerable quebradero de cabeza de portabilidad al cambiar entre diferentes versiones de `lex`.

La ventaja de '`%array`' es que entoces puede modificar `yytext` todo lo que usted quiera, las llamadas a '`unput()`' no destruyen `yytext` (ver más abajo). Además, los programas de `lex` existentes a veces acceden a `yytext` externamente utilizando declaraciones de la forma:

```
extern char yytext[];
```

Esta definición es errónea cuando se utiliza '`%pointer`', pero correcta para '`%array`'.

'`%array`' define a `yytext` como un array de `YYLMAX` caracteres, que por defecto es un valor bastante grande. Usted puede cambiar el tamaño símplemente definiendo con #define a `YYLMAX` con un valor diferente en la primera sección de su entrada de `flex`. Como se mencionó antes, con '`%pointer`' yytext crece dinámicamente para acomodar tokens grandes. Aunque esto signifique que con '`%pointer`' su escáner puede acomodar tokens muy grandes (tales como emparejar bloques enteros de comentarios), tenga presente que cada vez que el escáner deba cambiar el tamaño de `yytext` también debe reiniciar el análisis del token entero desde el principio, así que emparejar tales tokens puede resultar lento. Ahora `yytext` *no* crece dinámicamente si una llamada a '`unput()`' hace que se deba devolver demasiado texto; en su lugar, se produce un error en tiempo de ejecución.

También tenga en cuenta que no puede usar '`%array`' en los analizadores generados como clases de C++ (ver Capítulo 17 [Generando Escáneres en C++], página 40).

7 Acciones

Cada patrón en una regla tiene una acción asociada, que puede ser cualquier sentencia en C. El patrón finaliza en el primer caracter de espacio en blanco que no sea una secuencia de escape; lo que queda de la línea es su acción. Si la acción está vacía, entonces cuando el patrón se empareje el token de entrada simplemente se descarta. Por ejemplo, aquí está la especificación de un programa que borra todas las apariciones de "zap me" en su entrada:

```
%%
"zap me"
```

(Este copiará el resto de caracteres de la entrada a la salida ya que serán emparejados por la regla por defecto.)

Aquí hay un programa que comprime varios espacios en blanco y tabuladores a un solo espacio en blanco, y desecha los espacios que se encuentren al final de una línea:

```
%%
[ \t]+        putchar( ' ' );
[ \t]+$       /* ignora este token */
```

Si la acción contiene un '{', entonces la acción abarca hasta que se encuentre el correspondiente '}', y la acción podría entonces cruzar varias líneas. `flex` es capaz de reconocer las cadenas y comentarios de C y no se dejará engañar por las llaves que encuentre dentro de estos, pero aun así también permite que las acciones comiencen con '`%{`' y considerará que la acción es todo el texto hasta el siguiente '`%}`' (sin tener en cuenta las llaves ordinarias dentro de la acción).

Una acción que consista sólamente de una barra vertical ('`|`') significa "lo mismo que la acción para la siguiente regla." Vea más abajo para una ilustración.

Las acciones pueden incluir código C arbitrario, incuyendo sentencias `return` para devolver un valor desde cualquier rutina llamada '`yylex()`'. Cada vez que se llama a '`yylex()`' esta continúa procesando tokens desde donde lo dejó la última vez hasta que o bien llegue al final del fichero o ejecute un return.

Las acciones tienen libertad para modificar `yytext` excepto para alargarla (añadiendo caracteres al final–esto sobreescribirá más tarde caracteres en el flujo de entrada). Sin embargo esto no se aplica cuando se utiliza '`%array`' (ver Capítulo 6 [Cómo se empareja la entrada], página 9); en ese caso, `yytext` podría modificarse libremente de cualquier manera.

Las acciones tienen libertad para modificar `yyleng` excepto que estas no deberían hacerlo si la acción también incluye el uso de '`yymore()`' (ver más abajo).

Hay un número de directivas especiales que pueden incluirse dentro de una acción:

- `ECHO` copia yytext a la salida del escáner.
- `BEGIN` seguido del nombre de la condición de arranque pone al escáner en la condición de arranque correspondiente (ver Capítulo 9 [Condiciones de arranque], página 16).
- `REJECT` ordena al escáner a que proceda con la "segunda mejor" regla que concuerde con la entrada (o un prefijo de la entrada). La regla se escoge como se describió anteriormente en el Capítulo 6 [Cómo se Empareja la Entrada], página 9, y `yytext` e `yyleng` se ajustan de forma apropiada. Podría ser una que empareje tanto texto como la regla escogida originalmente pero que viene más tarde en el fichero de entrada de `flex`, o una que empareje menos texto. Por ejemplo, lo que viene a continuación contará las palabras en la entrada y llamará a la rutina `especial()` siempre que vea "frob":

```
            int contador_palabras = 0;
%%

frob        especial(); REJECT;
[^ \t\n]+   ++contador_palabras;
```

Sin el `REJECT`, cualquier número de "frob"'s en la entrada no serían contados como palabras, ya que el escáner normalmente ejecuta solo una acción por token. Se permite el uso de múltiples `REJECT's`, cada uno buscando la siguiente mejor elección a la regla que actualmente esté activa. Por ejemplo, cuando el siguiente escáner analice el token "abcd", este escribirá "abcdabcaba" a la salida:

```
%%
a        |
ab       |
abc      |
abcd     ECHO; REJECT;
.|\n     /* se come caracteres sin emparejar */
```

(Las primeras tres reglas comparten la acción de la cuarta ya que estas usan la acción especial '`|`'.) `REJECT` es una propiedad particularmente cara en términos de rendimiento del escáner; si se usa en *cualquiera* de las acciones del escáner esta ralentizará *todo* el proceso de emparejamiento del escáner. Además, `REJECT` no puede usarse con las opciones '`-Cf`' ó '`-CF`' (ver Sección 15.2 [Opciones], página 28 y Capítulo 16 [Consideraciones de rendimiento], página 35.)

Fíjese también que a diferencia de las otras acciones especiales, `REJECT` es una *bifurcación*; el código que la siga inmediatamente en la acción *no* será ejecutado.

- '`yymore()`' dice al escáner que la próxima vez que empareje una regla, el token correspondiente debe ser *añadido* tras el valor actual de `yytext` en lugar de reemplazarlo. Por ejemplo, dada la entrada "mega-klugde" lo que viene a continuación escribirá "mega-mega-kludge" a la salida:

```
%%
mega-    ECHO; yymore();
kludge   ECHO;
```

  El primer "mega-" se empareja y se repite a la salida. Entonces se empareja "kludge", pero el "mega-" previo aún está esperando al inicio de `yytext` asi que el '`ECHO`' para la regla del "kludge" realmente escribirá "mega-kludge".

Dos notas respecto al uso de '`yymore()`'. Primero, '`yymore()`' depende de que el valor de `yyleng` refleje correctamente el tamaño del token actual, así que no debe modificar `yyleng` si está utilizando '`yymore()`'. Segundo, la presencia de '`yymore()`' en la acción del escáner implica una pequeña penalización de rendimiento en la velocidad de emparejamiento del escáner.

- '`yyless(n)`' devuelve todos excepto los primeros *n* caracteres del token actual de nuevo al flujo de entrada, donde serán reanalizados cuando el escáner busque el siguiente emparejamiento. `yytext` e `yyleng` se ajustan de forma adecuada (p.ej., `yyleng` no será igual a *n*). Por ejemplo, con la entrada "foobar" lo que viene a continuación escribirá "foobarbar":

```
%%
foobar    ECHO; yyless(3);
[a-z]+    ECHO;
```

  Un argumento de 0 para `yyless` hará que la cadena de entrada actual sea analizada por completo de nuevo. A menos que haya cambiado la manera en la que el escáner procese de ahora en adelante su entrada (utilizando `BEGIN`, por ejemplo), esto producirá un bucle sin fin.

  Fíjese que `yyless` es una macro y puede ser utilizada solamente en el fichero de entrada de flex, no desde otros ficheros fuente.

- '`unput(c)`' pone el caracter `c` de nuevo en el flujo de entrada. Este será el próximo caracter analizado. La siguiente acción tomará el token actual y hará que se vuelva a analizar pero encerrado entre paréntesis.

```
{
```

```
int i;
/* Copia yytext porque unput() desecha yytext */
char *yycopia = strdup( yytext );
unput( ')' );
for ( i = yyleng - 1; i >= 0; --i )
    unput( yycopia[i] );
unput( '(' );
free( yycopia );
}
```

Fíjese que ya que cada '`unput()`' pone el caracter dado de nuevo al *principio* del flujo de entrada, al devolver cadenas de caracteres se debe hacer de atrás hacia delante.

Un problema potencial importante cuando se utiliza '`unput()`' es que si está usando '`%pointer`' (por defecto), una llamada a '`unput()`' *destruye* el contenido de `yytext`, comenzando con su caracter más a la derecha y devorando un caracter a la izquierda con cada llamada. Si necesita que se preserve el valor de yytext después de una llamada a '`unput()`' (como en el ejemplo anterior), usted debe o bien copiarlo primero en cualquier lugar, o construir su escáner usando '`%array`' (ver Capítulo 6 [Cómo se Empareja la Entrada], página 9).

Finalmente, note que no puede devolver `EOF` para intentar marcar el flujo de entrada con un fin-de-fichero.

- '`input()`' lee el próximo caracter del flujo de entrada. Por ejemplo, lo que viene a continuación es una manera de comerse los comentarios en C:

```
%%
"/*"        {
            register int c;

            for ( ; ; )
                {
                while ( (c = input()) != '*' &&
                        c != EOF )
                    ;    /* se come el texto del comentario */

                if ( c == '*' )
                    {
                    while ( (c = input()) == '*' )
                        ;
                    if ( c == '/' )
                        break;    /* encontró el final */
                    }

                if ( c == EOF )
                    {
                    error( "EOF en comentario" );
                    break;
                    }
                }
            }
```

(Fíjese que si el escáner se compila usando '`C++`', entonces a '`input()`' se le hace referencia con '`yyinput()`', para evitar una colisión de nombre con el flujo de '`C++`' por el nombre `input`.)

- YY_FLUSH_BUFFER vacía el buffer interno del escáner de manera que la próxima vez que el escáner intente emparejar un token, este primero rellenará el buffer usando `YY_INPUT` (ver Capítulo 8 [El Escáner Generado], página 14). Esta acción es un caso especial de la función más general '`yy_flush_buffer()`', descrita más abajo en el Capítulo 10 [Múltiples Buffers de Entrada], página 21.
- '`yyterminate()`' se puede utilizar en lugar de una sentencia de retorno en una acción. Esta hace que finalice el escáner y retorne un 0 a quien haya llamado al escáner, indicando que "todo está hecho". Por defecto, también se llama a '`yyterminate()`' cuando se encuentra un fin-de-fichero. Esta es una macro y podría ser redefinida.

8 El escáner generado

La salida de **flex** es el fichero 'lex.yy.c', que contiene la rutina de análisis 'yylex()', un número de tablas usadas por esta para emparejar tokens, y un número de rutinas auxiliares y macros. Por defecto, 'yylex()' se declara así

```
int yylex()
    {
    ... aquí van varias definiciones y las acciones ...
    }
```

(Si su entorno acepta prototipos de funciones, entonces este será "int yylex(void)"). Esta definición podría modificarse definiendo la macro "YY_DECL". Por ejemplo, podría utilizar:

```
#define YY_DECL float lexscan( a, b ) float a, b;
```

para darle a la rutina de análisis el nombre `lexscan`, que devuelve un real, y toma dos reales como argumentos. Fíjese que si pone argumentos a la rutina de análisis usando una declaración de función no-prototipada/tipo-K&R, debe hacer terminar la definición con un punto y coma (';').

Siempre que se llame a 'yylex()', este analiza tokens desde el fichero de entrada global `yyin` (que por defecto es igual a stdin). La función continúa hasta que alcance el final del fichero (punto en el que devuelve el valor 0) o una de sus acciones ejecute una sentencia `return`.

Si el escáner alcanza un fin-de-fichero, entonces el comportamiento en las llamadas posteriores está indefinido a menos que o bien `yyin` apunte a un nuevo fichero de entrada (en cuyo caso el análisis continúa a partir de ese fichero), o se llame a 'yyrestart()'. 'yyrestart()' toma un argumento, un puntero 'FILE *' (que puede ser nulo, si ha preparado a `YY_INPUT` para que analice una fuente distinta a `yyin`), e inicializa `yyin` para que escanee ese fichero. Esencialmente no hay diferencia entre la asignación a `yyin` de un nuevo fichero de entrada o el uso de 'yyrestart()' para hacerlo; esto último está disponible por compatibilidad con versiones anteriores de **flex**, y porque puede utilizarse para conmutar ficheros de entrada en medio del análisis. También se puede utilizar para desechar el buffer de entrada actual, invocándola con un argumento igual a `yyin`; pero mejor es usar `YY_FLUSH_BUFFER` (ver Capítulo 7 [Acciones], página 10). Fíjese que 'yyrestart()' *no* reinicializa la condición de arranque a `INITIAL` (ver Capítulo 9 [Condiciones de arranque], página 16).

Si 'yylex()' para el análisis debido a la ejecución de una sentencia `return` en una de las acciones, el analizador podría ser llamado de nuevo y este reanudaría el análisis donde lo dejó.

Por defecto (y por razones de eficiencia), el analizador usa lecturas por bloques en lugar de simples llamadas a 'getc()' para leer caracteres desde `yyin`. La manera en la que toma su entrada se puede controlar definienfo la macro `YY_INPUT`. La secuencia de llamada para YY_INPUT es "YY_INPUT(buf,result,max_size)". Su acción es poner hasta *max_size* caracteres en el array de caracteres *buf* y devolver en la variable entera *result* bien o el número de caracteres leídos o la constante YY_NULL (0 en sistemas Unix) para indicar EOF. Por defecto YY_INPUT lee desde la variable global puntero a fichero "yyin".

Una definición de ejemplo para YY_INPUT (en la sección de definiciones del fichero de entrada) es:

```
%{
#define YY_INPUT(buf,result,max_size) \
    { \
    int c = getchar(); \
    result = (c == EOF) ? YY_NULL : (buf[0] = c, 1); \
    }
%}
```

Esta definición cambiará el procesamiento de la entrada para que suceda un caracter a la vez.

Cuando el analizador reciba una indicación de fin-de-fichero desde YY_INPUT, entonces esta comprueba la función 'yywrap()'. Si 'yywrap()' devuelve falso (cero), entonces se asume que la función ha ido más allá y ha preparado yyin para que apunte a otro fichero de entrada, y el análisis continúa. Si este retorna verdadero (no-cero), entonces el analizador termina, devolviendo un 0 a su invocador. Fíjese que en cualquier caso, la condición de arranque permanece sin cambios; esta *no* vuelve a ser INITIAL.

Si no proporciona su propia versión de 'yywrap()', entonces debe bien o usar '%option noyywrap' (en cuyo caso el analizador se comporta como si 'yywrap()' devolviera un 1), o debe enlazar con '-lfl' para obtener la versión por defecto de la rutina, que siempre devuelve un 1.

Hay disponibles tres rutinas para analizar desde buffers de memoria en lugar de desde ficheros: 'yy_scan_string()', 'yy_scan_bytes()', e 'yy_scan_buffer()'. Las trataremos en la Capítulo 10 [Múltiples Buffers de Entrada], página 21. El analizador escribe su salida con 'ECHO' a la variable global yyout (por defecto, stdout), que el usuario podría redefinir asignándole cualquier otro puntero a FILE.

9 Condiciones de arranque

`flex` dispone de un mecanismo para activar reglas condicionalmente. Cualquier regla cuyo patrón se prefije con "<sc>" únicamente estará activa cuando el analizador se encuentre en la condición de arranque llamada "sc". Por ejemplo,

```
<STRING>[^"]*        { /* se come el cuerpo de la cadena ... */
            ...
            }
```

estará activa solamente cuando el analizador esté en la condición de arranque "STRING", y

```
<INITIAL,STRING,QUOTE>\. { /* trata una secuencia de escape ... */
            ...
            }
```

estará activa solamente cuando la condición de arranque actual sea o bien "INITIAL", "STRING", o "QUOTE".

Las condiciones de arranque se declaran en la (primera) sección de definiciones de la entrada usando líneas sin sangrar comenzando con '`%s`' ó '`%x`' seguida por una lista de nombres. Lo primero declara condiciones de arranque *inclusivas*, lo último condiciones de arranque *exclusivas*. Una condición de arranque se activa utilizando la acción `BEGIN`. Hasta que se ejecute la próxima acción `BEGIN`, las reglas con la condición de arranque dada estarán activas y las reglas con otras condiciones de arranque estarán inactivas. Si la condición de arranque es *inclusiva*, entonces las reglas sin condiciones de arranque también estarán activas. Si es *exclusiva*, entonces *sólamente* las reglas calificadas con la condición de arranque estarán activas. Un conjunto de reglas dependientes de la misma condición de arranque exclusiva describe un analizador que es independiente de cualquiera de las otras reglas en la entrada de `flex`. Debido a esto, las condiciones de arranque exclusivas hacen fácil la especificación de "mini-escáneres" que analizan porciones de la entrada que son sintácticamente diferentes al resto (p.ej., comentarios).

Si la distinción entre condiciones de arranque inclusivas o exclusivas es aún un poco vaga, aquí hay un ejemplo simple que ilustra la conexión entre las dos. El conjunto de reglas:

```
%s ejemplo
%%

<ejemplo>foo   hacer_algo();

bar            algo_mas();
```

es equivalente a

```
%x ejemplo
%%

<ejemplo>foo   hacer_algo();

<INITIAL,ejemplo>bar    algo_mas();
```

Sin el calificador '`<INITIAL,example>`', el patrón '`bar`' en el segundo ejemplo no estará activo (es decir, no puede emparejarse) cuando se encuentre en la condición de arranque '`example`'. Si hemos usado '`<example>`' para calificar '`bar`', aunque, entonces este únicamente estará activo en '`example`' y no en `INITIAL`, mientras que en el primer ejemplo está activo en ambas, porque en el primer ejemplo la condición de arranque '`example`' es una condición de arranque *inclusiva* ('`%s`').

Fíjese también que el especificador especial de la condición de arranque '`<*>`' empareja todas las condiciones de arranque. Así, el ejemplo anterior también pudo haberse escrito;

```
%x ejemplo
%%

<ejemplo>foo   hacer_algo();

<*>bar    algo_mas();
```

La regla por defecto (hacer un 'ECHO' con cualquier caracter sin emparejar) permanece activa en las condiciones de arranque. Esta es equivalente a:

```
<*>.|\n     ECHO;
```

'BEGIN(0)' retorna al estado original donde solo las reglas sin condiciones de arranque están activas. Este estado también puede referirse a la condición de arranque "INITIAL", así que '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.)

Las acciones BEGIN pueden darse también como código sangrado al comienzo de la sección de reglas. Por ejemplo, lo que viene a continuación hará que el analizador entre en la condición de arranque "ESPECIAL" siempre que se llame a 'yylex()' y la variable global entra_en_especial sea verdadera:

```
        int entra_en_especial;

%x ESPECIAL
%%
        if ( entra_en_especial )
            BEGIN(ESPECIAL);

<ESPECIAL>blablabla
...más reglas a continuación...
```

Para ilustrar los usos de las condiciones de arranque, aquí hay un analizador que ofrece dos interpretaciones diferentes para una cadena como "123.456". Por defecto este la tratará como tres tokens, el entero "123", un punto ('.'), y el entero "456". Pero si la cadena viene precedida en la línea por la cadena "espera-reales" este la tratará como un único token, el número en coma flotante 123.456:

```
%{
#include <math.h>
%}
%s espera

%%
espera-reales        BEGIN(espera);

<espera>[0-9]+"."[0-9]+      {
            printf( "encontró un real, = %f\n",
                    atof( yytext ) );
            }
<espera>\n           {
            /* este es el final de la línea,
             * así que necesitamos otro
             * "espera-numero" antes de
             * que volvamos a reconocer más
             * números
             */
            BEGIN(INITIAL);
```

```
            }

[0-9]+      {
            printf( "encontró un entero, = %d\n",
                    atoi( yytext ) );
            }

"."         printf( "encontró un punto\n" );
```

Aquí está un analizador que reconoce (y descarta) comentarios de C mientras mantiene una cuenta de la línea actual de entrada.

```
%x comentario
%%
        int num_linea = 1;

"/*"         BEGIN(comentario);

<comentario>[^*\n]*        /* come todo lo que no sea '*' */
<comentario>"*"+[^*/\n]*   /* come '*'s no seguidos por '/' */
<comentario>\n             ++num_linea;
<comentario>"*"+"/"        BEGIN(INITIAL);
```

Este analizador se complica un poco para emparejar tanto texto como le sea posible en cada regla. En general, cuando se intenta escribir un analizador de alta velocidad haga que cada regla empareje lo más que pueda, ya que esto es un buen logro.

Fíjese que los nombres de las condiciones de arranque son realmente valores enteros y pueden ser almacenados como tales. Así, lo anterior podría extenderse de la siguiente manera:

```
%x comentario foo
%%
        int num_linea = 1;
        int invocador_comentario;

"/*"         {
             invocador_comentario = INITIAL;
             BEGIN(comentario);
             }

...

<foo>"/*"    {
             invocador_comentario = foo;
             BEGIN(comentario);
             }

<comentario>[^*\n]*         /* se come cualquier cosa que no sea un '*' */
<comentario>"*"+[^*/\n]*    /* se come '*'s que no continuen con '/' */
<comentario>\n              ++num_linea;
<comentario>"*"+"/"         BEGIN(invocador_comentario);
```

Además, puede acceder a la condición de arranque actual usando la macro de valor entero `YY_START`. Por ejemplo, las asignaciones anteriores a `invocador_comentario` podrían escribirse en su lugar como

```
invocador_comentario = YY_START;
```

Flex ofrece `YYSTATE` como un alias para `YY_START` (ya que es lo que usa `lex` de AT&T).

Fíjese que las condiciones de arranque no tienen su propio espacio de nombres; los %s's y %x's declaran nombres de la misma manera que con #define's.

Finalmente, aquí hay un ejemplo de cómo emparejar cadenas entre comillas al estilo de C usando condiciones de arranque exclusivas, incluyendo secuencias de escape expandidas (pero sin incluir la comprobación de cadenas que son demasiado largas):

```
%x str

%%
        char string_buf[MAX_STR_CONST];
        char *string_buf_ptr;


\"      string_buf_ptr = string_buf; BEGIN(str);

<str>\"        { /* se vio la comilla que cierra - todo está hecho */
        BEGIN(INITIAL);
        *string_buf_ptr = '\0';
        /* devuelve un tipo de token de cadena constante y
         * el valor para el analizador sintáctico
         */
        }

<str>\n        {
        /* error - cadena constante sin finalizar */
        /* genera un mensaje de error */
        }

<str>\\[0-7]{1,3} {
        /* secuencia de escape en octal */
        int resultado;

        (void) sscanf( yytext + 1, "%o", &resultado );

        if ( resultado > 0xff )
                /* error, constante fuera de rango */

        *string_buf_ptr++ = resultado;
        }

<str>\\[0-9]+ {
        /* genera un error - secuencia de escape errónea;
         * algo como '\48' o '\0777777'
         */
        }

<str>\\n  *string_buf_ptr++ = '\n';
<str>\\t  *string_buf_ptr++ = '\t';
<str>\\r  *string_buf_ptr++ = '\r';
<str>\\b  *string_buf_ptr++ = '\b';
<str>\\f  *string_buf_ptr++ = '\f';
```

```
<str>\\(.|\n)  *string_buf_ptr++ = yytext[1];

<str>[^\\\n\"]+        {
        char *yptr = yytext;

        while ( *yptr )
                *string_buf_ptr++ = *yptr++;

        }
```

A menudo, como en alguno de los ejemplos anteriores, uno acaba escribiendo un buen número de reglas todas precedidas por la(s) misma(s) condición(es) de arranque. Flex hace esto un poco más fácil y claro introduciendo la noción de *ámbito* de la condición de arranque. Un ámbito de condición de arranque comienza con:

```
<SCs>{
```

Donde '`SCs`' es una lista de una o más condiciones de arranque. Dentro del ámbito de la condición de arranque, cada regla automáticamente tiene el prefijo '`<SCs>`' aplicado a esta, hasta un '`}`' que corresponda con el '`{`' inicial. Así, por ejemplo,

```
<ESC>{
    "\\n"   return '\n';
    "\\r"   return '\r';
    "\\f"   return '\f';
    "\\0"   return '\0';
}
```

es equivalente a:

```
<ESC>"\\n"  return '\n';
<ESC>"\\r"  return '\r';
<ESC>"\\f"  return '\f';
<ESC>"\\0"  return '\0';
```

Los ámbitos de las condiciones de arranque pueden anidarse.

Están disponibles tres rutinas para manipular pilas de condiciones de arranque:

'`void yy_push_state(int new_state)`'
: empuja la condición de arranque actual al tope de la pila de las condiciones de arranque y cambia a *new_state* como si hubiera utilizado '`BEGIN new_state`' (recuerde que los nombres de las condiciones de arranque también son enteros).

'`void yy_pop_state()`'
: extrae el tope de la pila y cambia a este mediante un `BEGIN`.

'`int yy_top_state()`'
: devuelve el tope de la pila sin alterar el contenido de la pila.

La pila de las condiciones de arranque crece dinámicamente y por ello no tiene asociada ninguna limitación de tamaño. Si la memoria se agota, se aborta la ejecución del programa.

Para usar pilas de condiciones de arranque, su analizador debe incluir una directiva '`%option stack`' (ver Sección 15.2 [Opciones], página 28).

10 Múltiples buffers de entrada

Algunos analizadores (tales como aquellos que aceptan ficheros "incluidos") requieren la lectura de varios flujos de entrada. Ya que los analizadores de `flex` hacen mucho uso de buffers, uno no puede controlar de dónde será leída la siguiente entrada escribiendo símplemente un `YY_INPUT` que sea sensible al contexto del análisis. A `YY_INPUT` sólo se le llama cuando el analizador alcanza el final de su buffer, que podría ser bastante tiempo después de haber analizado una sentencia como un "include" que requiere el cambio de la fuente de entrada.

Para solventar este tipo de problemas, `flex` provee un mecanismo para crear y conmutar entre varios buffers de entrada. Un buffer de entrada se crea usando:

```
YY_BUFFER_STATE yy_create_buffer( FILE *file, int size )
```

que toma un puntero a `FILE` y un tamaño "size" y crea un buffer asociado con el fichero dado y lo suficientemente grande para mantener *size* caracteres (cuando dude, use `YY_BUF_SIZE` para el tamaño). Este devuelve un handle `YY_BUFFER_STATE`, que podría pasarse a otras rutinas (ver más abajo). El tipo de `YY_BUFFER_STATE` es un puntero a una estructura opaca `struct yy_buffer_state`, de manera que podría inicializar de forma segura variables `YY_BUFFER_STATE` a '`((YY_BUFFER_STATE) 0)`' si lo desea, y también hacer referencia a la estructura opaca para declarar correctamente buffers de entrada en otros ficheros fuente además de los de su analizador. Fíjese que el puntero a `FILE` en la llamada a `yy_create_buffer` se usa solamente como el valor de `yyin` visto por `YY_INPUT`; si usted redefine `YY_INPUT` de manera que no use más a `yyin`, entonces puede pasar de forma segura un puntero `FILE` nulo a `yy_create_buffer`. Se selecciona un buffer en particular a analizar utilizando:

```
void yy_switch_to_buffer( YY_BUFFER_STATE nuevo_buffer )
```

conmuta el buffer de entrada del analizador de manera que los tokens posteriores provienen de *nuevo_buffer*. Fíjese que '`yy_switch_to_buffer()`' podría usarlo yywrap() para arreglar las cosas para un análisis continuo, en lugar de abrir un nuevo fichero y que `yyin` apunte a este. Fíjese también que cambiar las fuentes de entrada ya sea por medio de '`yy_switch_to_buffer()`' o de '`yywrap()`' *no* cambia la condición de arranque.

```
void yy_delete_buffer( YY_BUFFER_STATE buffer )
```

se usa para recuperar el almacenamiento asociado a un buffer. (El `buffer` puede ser nulo, en cuyo caso la rutina no hace nada.) Puede también limpiar el contenido actual de un buffer usando:

```
void yy_flush_buffer( YY_BUFFER_STATE buffer )
```

Esta función descarta el contenido del buffer, de manera que la próxima vez que el analizador intente emparejar un token desde el buffer, este primero rellenará el buffer utilizando `YY_INPUT`.

'`yy_new_buffer()`' es un alias de '`yy_create_buffer()`', que se ofrece por compatibilidad con el uso en C++ de `new` y `delete` para crear y destruir objetos dinámicos.

Finalmente, la macro `YY_CURRENT_BUFFER` retorna un handle `YY_BUFFER_STATE` al buffer actual.

Aquí hay un ejemplo del uso de estas propiedades para escribir un analizador que expande ficheros incluidos (la propiedad '`<<EOF>>`' se comenta en el Capítulo 11 [Reglas de fin-de-fichero], página 24):

```
/* el estado "incl" se utiliza para obtener el nombre
 * del fichero a incluir.
 */
%x incl

%{
#define MAX_INCLUDE_DEPTH 10
YY_BUFFER_STATE include_stack[MAX_INCLUDE_DEPTH];
```

```
int include_stack_ptr = 0;
%}

%%
include             BEGIN(incl);

[a-z]+              ECHO;
[^a-z\n]*\n?        ECHO;

<incl>[ \t]*      /* se come los espacios en blanco */
<incl>[^ \t\n]+   { /* obtiene el nombre de fichero a incluir */
        if ( include_stack_ptr >= MAX_INCLUDE_DEPTH )
            {
            fprintf( stderr, "Demasiados include anidados" );
            exit( 1 );
            }

        include_stack[include_stack_ptr++] =
            YY_CURRENT_BUFFER;

        yyin = fopen( yytext, "r" );

        if ( ! yyin )
            error( ... );

        yy_switch_to_buffer(
            yy_create_buffer( yyin, YY_BUF_SIZE ) );

        BEGIN(INITIAL);
        }

<<EOF>> {
        if ( --include_stack_ptr < 0 )
            {
            yyterminate();
            }

        else
            {
            yy_delete_buffer( YY_CURRENT_BUFFER );
            yy_switch_to_buffer(
                 include_stack[include_stack_ptr] );
            }
        }
```

Se dispone de tres rutinas para preparar buffers de entrada para el análisis de cadenas en memoria en lugar de archivos. Todas estas crean un nuevo buffer de entrada para analizar la cadena, y devuelven el correspondiente handle `YY_BUFFER_STATE` (que usted debería borrar con '`yy_delete_buffer()`' cuando termine con él). Estas también conmutan el nuevo buffer usando '`yy_switch_to_buffer()`', de manera que la próxima llamada a '`yylex()`' comenzará analizando la cadena.

'`yy_scan_string(const char *str)`'
: analiza una cadena terminada en nulo.

'`yy_scan_bytes(const char *bytes, int len)`'
: analiza `len` bytes (incluyendo posibles NUL's) comenzando desde el punto *bytes*.

Fíjese que ambas de estas funciones crean y analizan una *copia* de la cadena o bytes. (Esto podría ser deseable, ya que '`yylex()`' modifica el contenido del buffer que está analizado.) Usted puede evitar la copia utilizando:

'`yy_scan_buffer(char *base, yy_size_t size)`'
: que analiza in situ el buffer comenzando en *base*, que consiste de *size* bytes, donde los dos últimos bytes *deben* ser `YY_END_OF_BUFFER_CHAR` (ASCII NUL). Estos dos últimos bytes no se analizan; así, el análisis consta de '`base[0]`' hasta '`base[size-2]`', inclusive.

  Si se equivoca al disponer *base* de esta manera (es decir, olvidar los dos `YY_END_OF_BUFFER_CHAR` bytes finales), entonces '`yy_scan_buffer()`' devuelve un puntero nulo en lugar de crear un nuevo buffer de entrada.

  El tipo `yy_size_t` es un tipo entero con el que puede hacer una conversión a una expresión entera para reflejar el tamaño del buffer.

11 Reglas de fin-de-fichero

La regla especial "<<EOF>>" indica las acciones que deben tomarse cuando se encuentre un fin-de-fichero e yywrap() retorne un valor distinto de cero (es decir, indica que no quedan ficheros por procesar). La acción debe finalizar haciendo una de estas cuatro cosas:

- asignando a `yyin` un nuevo fichero de entrada (en versiones anteriores de flex, después de hacer la asignación debía llamar a la acción especial `YY_NEW_FILE`; esto ya no es necesario);
- ejecutando una sentencia `return`;
- ejecutando la acción especial '`yyterminate()`';
- o, conmutando a un nuevo buffer usando '`yy_switch_to_buffer()`' como se mostró en el ejemplo anterior.

Las reglas <<EOF>> no deberían usarse con otros patrones; estas deberían calificarse con una lista de condiciones de arranque. Si se da una regla <<EOF>> sin calificar, esta se aplica a *todas* las condiciones de arranque que no tengan ya acciones <<EOF>>. Para especificar una regla <<EOF>> solamente para la condición de arranque inicial, use

```
<INITIAL><<EOF>>
```

Estas reglas son útiles para atrapar cosas tales como comentarios sin final. Un ejemplo:

```
%x comilla
%%

...otras reglas que tengan que ver con comillas...

<comilla><<EOF>>   {
         error( "comilla sin cerrar" );
         yyterminate();
         }
<<EOF>>  {
         if ( *++filelist )
             yyin = fopen( *filelist, "r" );
         else
            yyterminate();
         }
```

12 Macros misceláneas

La macro `YY_USER_ACTION` puede definirse para indicar una acción que siempre se ejecuta antes de la acción de la regla emparejada. Por ejemplo, podría declararse con #define para que llame a una rutina que convierta yytext a minúsculas. Cuando se invoca a `YY_USER_ACTION`, la variable `yy_act` da el número de la regla emparejada (las reglas están numeradas comenzando en 1). Suponga que quiere medir la frecuencia con la que sus reglas son emparejadas. Lo que viene a continuación podría hacer este truco:

```
#define YY_USER_ACTION ++ctr[yy_act]
```

donde `ctr` en un vector que mantiene la cuenta para las diferentes reglas. Fíjese que la macro `YY_NUM_RULES` da el número total de reglas (incluyendo la regla por defecto, incluso si usted usa '`-s`'), así que una declaración correcta para `ctr` es:

```
int ctr[YY_NUM_RULES];
```

La macro `YY_USER_INIT` podría definirse para indicar una acción que siempre se ejecuta antes del primer análisis (y antes de que se haga la inicialización interna del analizador). Por ejemplo, este podría usarse para llamar a una rutina que lea una tabla de datos o abrir un fichero de registro.

La macro '`yy_set_interactive(is_interactive)`' se puede usar para controlar si el buffer actual se considera *interactivo*. Un buffer interactivo se procesa más lentamente, pero debe usarse cuando la fuente de entrada del analizador es realmente interactiva para evitar problemas debidos a la espera para el llenado de los buffers (ver el comentario de la bandera '`-I`' en la Sección 15.2 [Opciones], página 28). Un valor distinto de cero en la invocación de la macro marcará el buffer como interactivo, un valor de cero como no-interactivo. Fíjese que el uso de esta macro no tiene en cuenta '`%option always-interactive`' o '`%option never-interactive`' (ver Sección 15.2 [Opciones], página 28). '`yy_set_interactive()`' debe invocarse antes del comienzo del análisis del buffer que es considerado (o no) interactivo.

La macro '`yy_set_bol(at_bol)`' puede usarse para controlar si el contexto del buffer de análisis actual para el próximo emparejamiento de token se hace como si se encontrara al principio de una línea. Un argumento de la macro distinto de cero hace activas a las reglas sujetas a '`^`', mientras que un argumento igual a cero hacer inactivas a las reglas con '`^`'.

La macro '`YY_AT_BOL()`' devuelve verdadero si el próximo token analizado a partir del buffer actual tendrá activas las reglas '`^`', de otra manera falso.

En el analizador generado, las acciones están recogidas en una gran sentencia switch y separadas usando `YY_BREAK`, que puede ser redefinida. Por defecto, este es símplemente un `"break"`, para separar la acción de cada regla de las reglas que le siguen. Redefiniendo `YY_BREAK` permite, por ejemplo, a los usuarios de C++ que #define YY_BREAK no haga nada (¡mientras tengan cuidado para que cada regla finalice con un `"break"` o un `"return"`!) para evitar que sufran los avisos de sentencias inalcanzables cuando debido a que la acción de la regla finaliza con un `"return"`, el `YY_BREAK` es inaccesible.

13 Valores disponibles al usuario

Esta sección resume los diferentes valores disponibles al usuario en las acciones de la regla.

- '`char *yytext`' apunta al texto del token actual. Este puede modificarse pero no alargarse (no puede añadir caracteres al final).

  Si aparece la directiva especial '`%array`' en la primera sección de la descripción del analizador, entonces `yytext` se declara en su lugar como '`char yytext[YYLMAX]`', donde `YYLMAX` es la definicion de una macro que puede redefinir en la primera sección si no le gusta el valor por defecto (generalmente 8KB). El uso de '`%array`' produce analizadores algo más lentos, pero el valor de `yytext` se vuelve inmune a las llamadas a '`input()`' y '`unput()`', que potencialmente destruyen su valor cuando `yytext` es un puntero a caracter. El opuesto de '`%array`' es '`%pointer`', que se encuentra por defecto.

  Usted no puede utilizar '`%array`' cuando genera analizadores como clases de C++ (la bandera '`-+`').
- '`int yyleng`' contiene la longitud del token actual.
- '`FILE *yyin`' es el fichero por el que `flex` lee por defecto. Este podría redefinirse pero hacerlo solo tiene sentido antes de que el análisis comience o después de que se haya encontrado un EOF. Cambiándolo en medio del análisis tendrá resultados inesperados ya que `flex` utiliza buffers en su entrada; use '`yyrestart()`' en su lugar. Una vez que el análisis termina debido a que se ha visto un fin-de-fichero, puede asignarle a `yyin` el nuevo fichero de entrada y entonces llamar al analizador de nuevo para continuar analizando.
- '`void yyrestart( FILE *new_file )`' podría ser llamada para que `yyin` apunte al nuevo fichero de entrada. El cambio al nuevo fichero es inmediato (cualquier entrada contenida en el buffer previamente se pierde). Fíjese que llamando a '`yyrestart()`' con `yyin` como argumento de esta manera elimina el buffer de entradda actual y continúa analizando el mismo fichero de entrada.
- '`FILE *yyout`' es el fichero sobre el que se hacen las acciones '`ECHO`'. Este puede ser reasignado por el usuario.
- `YY_CURRENT_BUFFER` devuelve un handle `YY_BUFFER_STATE` al buffer actual.
- `YY_START` devuelve un valor entero correspondiente a la condición de arranque actual. Posteriormente puede usar este valor con `BEGIN` para retornar a la condición de arranque.

14 Interfaz con YACC

Uno de los usos principales de `flex` es como compañero del generador de analizadores sintácticos `yacc`. Los analizadores de `yacc` esperan invocar a una rutina llamada '`yylex()`' para encontrar el próximo token de entrada. La rutina se supone que devuelve el tipo del próximo token además de poner cualquier valor asociado en la variable global `yylval`. Para usar `flex` con `yacc`, uno especifica la opción '`-d`' de `yacc` para intruirle a que genere el fichero '`y.tab.h`' que contiene las definiciones de todos los '`%tokens`' que aparecen en la entrada de `yacc`. Entonces este archivo se incluye en el analizador de `flex`. Por ejemplo, si uno de los tokens es "TOK_NUMERO", parte del analizador podría parecerse a:

```
%{
#include "y.tab.h"
%}

%%

[0-9]+        yylval = atoi( yytext ); return TOK_NUMERO;
```

15 Invocando a Flex

15.1 Sinopsis

```
flex [-bcdfhilnpstvwBFILTV78+? -C[aefFmr] -osalida -Pprefijo -Sesqueleto]
[--help --version] [nombrefichero ...]
```

15.2 Opciones

`flex` tiene las siguientes opciones:

'-b' Genera información de retroceso en '`lex.backup`'. Esta es una lista de estados del analizador que requieren retroceso y los caracteres de entrada con los que la hace. Añadiendo reglas uno puede eliminar estados de retroceso. Si *todos* los estados de retroceso se eliminan y se usa '`-Cf`' ó '`-CF`', el analizador generado funcionará más rápido (ver la bandera '`-p`'). Únicamente los usuarios que desean exprimir hasta el último ciclo de sus analizadores necesitan preocuparse de esta opción. (ver Capítulo 16 [Consideraciones de Rendimiento], página 35)

'-c' es una opción que no hace nada, incluída para cumplir con POSIX.

'-d' hace que el analizador generado se ejecute en modo de *depuración*. Siempre que se reconoce un patrón y la variable global '`yy_flex_debug`' no es cero (que por defecto no lo es), el analizador escribirá en `stderr` una línea de la forma:

```
--accepting rule at line 53 ("el texto emparejado")
```

El número de línea hace referencia al lugar de la regla en el fichero que define al analizador (es decir, el fichero que se le introdujo a flex). Los mensajes también se generan cuando el analizador retrocede, acepta la regla por defecto, alcanza el final de su buffer de entrada (o encuentra un NUL; en este punto, los dos parecen lo mismo en lo que le concierne al analizador), o alcance el fin-de-fichero.

'-f' especifica un *analizador rápido*. No se realiza una compresión de tablas y se evita el uso de stdio. El resultado es grande pero rápido. Esta opción es equivalente a '`-Cfr`' (ver más abajo).

'-h' genera un sumario de "ayuda" de las opciones de `flex` por `stdout` y entonces finaliza. '`-?`' y '`--help`' son sinónimos de '`-h`'.

'-i' indica a `flex` que genere un analizador *case-insensitive*. Se ignorará si las letras en los patrones de entrada de `flex` son en mayúsculas o en minúsculas, y los tokens en la entrada serán emparejados sin tenerlo en cuenta. El texto emparejado dado en `yytext` tendrá las mayúsculas y minúsculas preservadas (es decir, no se convertirán).

'-l' activa el modo de máxima compatibilidad con la implementación original de `lex` de AT&T. Fíjese que esto no significa una compatibilidad *completa*. El uso de esta opción cuesta una cantidad considerable de rendimiento, y no puede usarse con las opciones '`-+`', '`-f`', '`-F`', '`-Cf`', ó '`-CF`'. Para los detalles a cerca de la compatibilidad que se ofrece, vea la Capítulo 18 [Incompatibilidades con `lex` y POSIX], página 44. Esta opción también hace que se defina el nombre `YY_FLEX_LEX_COMPAT` en el analizador generado.

'-n' es otra opción que no hace nada, incluída para cumplir con POSIX.

'-p' genera un informe de rendimiento en stderr. El informe consta de comentarios que tratan de las propiedades del fichero de entrada de `flex` que provocarán pérdidas serias de rendimiento en el analizador resultante. Si indica esta bandera dos veces, también obtendrá comentarios que tratan de las propiedades que producen pérdidas menores de rendimiento.

Fíjese que el uso de `REJECT`, '`%option yylineno`', y el contexto posterior variable (ver Capítulo 21 [Deficiencias / Errores], página 50) supone una penalización substancial del rendimiento; el uso de '`yymore()`', el operador '`^`', y la bandera '`-I`' supone penalizaciones del rendimiento menores.

'`-s`' hace que la *regla por defecto* (que la entrada sin emparejar del analizador se repita por `stdout`) se suprima. Si el analizador encuentra entrada que no es reconocida por ninguna de sus reglas, este aborta con un error. Esta opción es útil para encontrar agujeros en el conjunto de reglas del analizador.

'`-t`' indica a `flex` que escriba el analizador que genera a la salida estándar en lugar de en '`lex.yy.c`'.

'`-v`' especifica que `flex` debería escribir en `stderr` un sumario de estadísticas respecto al analizador que genera. La mayoría de las estadísticas no tienen significado para el usuario casual de `flex`, pero la primera línea identifica la versión de `flex` (la misma que se informa con '`-V`'), y la próxima línea las banderas utilizadas cuando se genera el analizador, incluyendo aquellas que se encuentran activadas por defecto.

'`-w`' suprime los mensajes de aviso.

'`-B`' dice a `flex` que genere un analizador *batch*, que es lo opuesto al analizador *interactivo* generador por '`-I`' (ver más abajo). En general, use '`-B`' cuando esté *seguro* de que su analizador nunca se usará de forma interactiva, y quiere con esto exprimir un *poco* más el rendimiento. Si por el contrario su objetivo es exprimirlo *mucho* más, debería estar utilizando la opción '`-Cf`' ó '`-CF`' (comentadas más abajo), que activa '`-B`' automáticamente de todas maneras.

'`-F`' especifica que se debe utilizar la representación de la tabla *rápida* (y elimina referencias a stdio). Esta representación es aproximadamente tan rápida como la representación completa de la tabla '`(-f)`', y para algunos conjuntos de patrones será considerablemente más pequeña (y para otros, mayor). En general, si el conjunto de patrones contiene "palabras clave" y una regla "identificador" atrápalo-todo, como la del conjunto:

```
"case"    return TOK_CASE;
"switch"  return TOK_SWITCH;
...
"default" return TOK_DEFAULT;
[a-z]+    return TOK_ID;
```

entonces será mejor que utilice la representación de la tabla completa. Si sólo está presente la regla "identificador" y utiliza una tabla hash o algo parecido para detectar palabras clave, mejor utilice '`-F`'.

Esta opción es equivalente a '`-CFr`' (ver más abajo). Esta opción no puede utilizarse con '`-+`'.

'`-I`' ordena a `flex` que genere un analizador *interactivo*. Un analizador interactivo es uno que solo mira hacia delante para decidir que token ha sido reconocido únicamente si debe hacerlo. Resulta que mirando siempre un caracter extra hacia delante, incluso si el analizador ya ha visto suficiente texto para eliminar la ambigüedad del token actual, se es un poco más rápido que mirando solamente cuando es necesario. Pero los analizadores que siempre miran hacia delante producen un comportamiento interactivo malísimo; por ejemplo, cuando un usuario teclea una línea nueva, esta no se reconoce como un token de línea nueva hasta que introduzca *otro* token, que a menudo significa introducir otra línea completa.

Los analizadores de `flex` por defecto son *interactivos* a menos que use la opción '`-Cf`' ó '`-CF`' de compresión de tablas (ver más abajo). Esto es debido a que si está buscando un rendimiento alto tendría que estar utilizando una de estas opciones, así que si no lo ha hecho `flex` asume que prefiere cambiar un poco de rendimiento en tiempo de ejecución en beneficio de un comportamiento iteractivo intuitivo. Fíjese también que *no puede* utilizar '`-I`' conjuntamente con '`-Cf`' ó '`-CF`'. Así, esta opción no se necesita realmente; está activa por defecto para todos esos casos en los que se permite.

Usted puede forzar al analizador que *no* sea interactivo usando '`-B`' (ver más arriba).

'`-L`' ordena a `flex` que no genere directivas '`#line`'. Sin esta opción, `flex` acribilla al analizador generado con directivas '`#line`' para que los mensajes de error en las acciones estén localizadas correctamente respecto al fichero original de `flex` (si los errores son debidos al código en el fichero de entrada), o a '`lex.yy.c`' (si los errores son fallos de `flex` —debería informar de este tipo de errores a la dirección de correo dada más abajo).

'`-T`' hace que `flex` se ejecute en modo de `traza`. Este generará un montón de mensajes en `stderr` relativos a la forma de la entrada y el autómata finito no-determinista o determinista resultante. Esta opción generalmente es para usarla en el mantenimiento de `flex`.

'`-V`' imprime el número de la versión en `stdout` y sale. '`--version`' es un sinónimo de '`-V`'.

'`-7`' ordena a `flex` que genere un analizador de 7-bits, es decir, uno que sólo puede reconocer caracteres de 7-bits en su entrada. La ventaja de usar '`-7`' es que las tablas del analizador pueden ser hasta la mitad del tamaño de aquellas generadas usando la opción '`-8`' (ver más abajo). La desventaja es que tales analizadores a menudo se cuelgan o revientan si su entrada contiene caracteres de 8-bits.

Fíjese, sin embargo, que a menos que genere su analizador utilizando las opciones de compresión de tablas '`-Cf`' ó '`-CF`', el uso de '`-7`' ahorrará solamente una pequeña cantidad de espacio en la tabla, y hará su analizador considerablemente menos portable. El comportamiento por defecto de `flex` es generar un analizador de 8-bits a menos que use '`-Cf`' ó '`-CF`', en cuyo caso `flex` por defecto genera analizadores de 7-bits a menos que su sistema siempre esté configurado para generar analizadores de 8-bits (a menudo este será el caso de los sistemas fuera de EEUU). Puede decir si flex generó un analizador de 7 u 8 bits inspeccionando el sumario de banderas en la salida de '`-v`' como se describió anteriormente.

Fíjese que si usa '`-Cfe`' ó '`-CFe`' (esas opciones de compresión de tablas, pero también el uso de clases de equivalencia como se comentará más abajo), flex genera aún por defecto un analizador de 8-bits, ya que normalmente con estas opciones de compresión las tablas de 8-bits completas no son mucho más caras que las tablas de 7-bits.

'`-8`' ordena a `flex` que genere un analizador de 8-bits, es decir, uno que puede reconocer caracteres de 8-bits. Esta bandera sólo es necesaria para analizadores generados usando '`-Cf`' ó '`-CF`', ya que de otra manera flex por defecto genera un analizador de 8-bits de todas formas.

Vea el comentario sobre '`-7`' más arriba a cerca del comportamiento por defecto de flex y la discusión entre los analizadores de 7-bits y 8-bits.

'`-+`' especifica que quiere que flex genere un analizador como una clase de C++. Vea la Capítulo 17 [Generando Escáneres en C++], página 40, para los detalles.

'`-C[aefFmr]`'
controla el grado de compresión de la tabla y, más generalmente, el compromiso entre analizadores pequeños y analizadores rápidos.

'-Ca' ("alinea") ordena a flex que negocie tablas más grandes en el analizador generado para un comportamiento más rápido porque los elementos de las tablas están mejor alineados para el acceso a memoria y computación. En algunas arquitecturas RISC, la búsqueda y manipulación de palabras largas es más eficiente que con unidades más pequeñas tales como palabras cortas. Esta opción puede doblar el tamaño de las tablas usadas en su analizador.

'-Ce' ordena a `flex` que construya *clases de equivalencia*, es decir, conjunto de caracteres que tienen identicas propiedades léxicas (por ejemplo, si la única aparición de dígitos en la entrada de `flex` es en la clase de caracteres "[0-9]" entonces los dígitos '0', '1', ..., '9' se pondrán todos en la misma clase de equivalencia). Las clases de equivalencia normalmente ofrecen notables reducciones en los tamaños de los ficheros finales de tabla/objeto (típicamente un factor de 2-5) y son juiciosamente bastante baratos en cuanto al rendimiento (una localización en un vector por caracter analizado).

'-Cf' especifica que se deben generar las tablas del analizador *completas* —`flex` no debería comprimir las tablas tomando ventaja de las funciones de transición similares para diferentes estados.

'-CF' especifica que debería usarse la representación del analizador rápido alternativo (descrito anteriormente en la bandera '-F') Esta opción no puede usarse con '-+'.

'-Cm' ordena a `flex` que construya *clases de meta-equivalencias*, que son conjuntos de clases de equivalencia (o caracteres, si las clases de equivalencia no se están usando) que comunmente se usan de forma conjunta. Las clases de meta-equivalencias son a menudo un gran ahorro cuando se usan tablas comprimidas, pero tienen un impacto moderado en el rendimiento (uno o dos tests "if" y una localización en un array por caracter analizado).

'-Cr' hace que el analizador generado *elimine* el uso de la librería de E/S estándar para la entrada. En lugar de llamar a 'fread()' o a 'getc()', el analizador utilizará la llamada al sistema 'read()', produciendo una ganancia en el rendimiento que varía de sistema en sistema, pero en general probablemente es insignificante a menos que también esté usando '-Cf' ó '-CF'. El uso de '-Cr' puede producir un comportamiento extraño si, por ejemplo, lee de `yyin` usando stdio antes de llamar al analizador (porque el analizador perderá cualquier texto que sus lecturas anteriores dejaron en el buffer de entrada de stdio).

'-Cr' no tiene efecto si usted define `YY_INPUT` (ver Capítulo 8 [El Escáner Generado], página 14).

Con solamente '-C' se especifica que las tablas del analizador deberían comprimirse pero no debería utilizarse ni las clases de equivalencia ni las clases de meta-equivalencias.

Las opciones '-Cf' ó '-CF' y '-Cm' no tienen sentido juntas —no hay oportunidad para las clases de meta-equivalencias si la tabla no está siendo comprimida. De otra forma las opciones podrían mezclarse líbremente, y son acumulativas.

La configuración por defecto es '-Cem', que especifica que `flex` debería generar clases de equivalencia y clases de meta-equivalencias. Esta configuración provee el mayor grado de compresión. Puede llegarse a un compromiso entre analizadores de ejecución más rápida con el coste de tablas mayores siendo generalmente verdadero lo siguiente:

```
lo más lento y pequeño
        -Cem
        -Cm
        -Ce
        -C
        -C{f,F}e
```

```
        -C{f,F}
        -C{f,F}a
    lo más rápido y grande
```

Fíjese que los analizadores con tablas más pequeñas normalmente se generan y compilan de la forma más rápida posible, así que durante el desarrollo usted normalmente querrá usar como viene por defecto, compresión máxima.

'-Cfe' a menudo es un buen compromiso entre velocidad y tamaño para la producción de analizadores.

'-osalida'
: ordena a flex que escriba el analizador al fichero 'salida' en lugar de a 'lex.yy.c'. Si combina '-o' con la opción '-t', entonces el analizador se escribe en stdout pero sus directivas '#line' (vea la opción '-L' más arriba) hacen referencia al fichero 'salida'.

'-Pprefijo'
: cambia el prefijo 'yy' usado por defecto por flex para todas las variables visibles globalmente y nombres de funciones para que sea *prefijo*. Por ejemplo, '-Pfoo' cambia el nombre de yytext a 'footext'. Este también cambia el nombre por defecto del fichero de salida de 'lex.yy.c' a 'lex.foo.c'. Aquí están todos los nombres afectados:

```
yy_create_buffer
yy_delete_buffer
yy_flex_debug
yy_init_buffer
yy_flush_buffer
yy_load_buffer_state
yy_switch_to_buffer
yyin
yyleng
yylex
yylineno
yyout
yyrestart
yytext
yywrap
```

(Si usted está utilizando un analizador en C++, entonces únicamente yywrap y yyFlexLexer se ven afectados.) Dentro de su analizador, puede aún hacer referencia a las variables globales y funciones usando cualquier versión de su nombre; pero externamente, estas tienen el nombre modificado.

Esta opción le deja enlazar fácilmente múltiples programas flex conjuntamente en el mismo ejecutable. Fíjese, sin embargo, que usando esta opción también se renombra 'yywrap()', de manera que ahora *debe* o bien proveer su propia versión de la rutina (con el nombre apropiado) para su analizador, o usar '%option noyywrap', ya que enlazar con '-lfl' no podrá proveerle una por defecto.

'-Sfichero_esqueleto'
: ignora el fichero de esqueleteo por defecto con el que flex construye sus analizadores. Usted probablemente nunca necesitará utilizar esta opción a menos que este haciendo mantenimiento o un desarrollo de flex.

flex también ofrece un mecanismo para controlar las opciones dentro de la propia especificación del analizador, en vez de a partir de la línea de comando. Esto se hace incluyendo las directivas '%option' en la primera sección de la especificación del analizador. Usted puede especificar varias opciones con una sola directiva '`%option`', y varias directivas en la primera sección de su fichero de entrada de flex.

La mayoría de las opciones vienen dadas simplemente como nombres, opcionalmente precedidos por la palabra "no" (sin intervenir un espacio) para negar su significado. Las banderas de flex o su negación son equivalentes a un número:

```
7bit            opción -7
8bit            opción -8
align           opción -Ca
backup          opción -b
batch           opción -B
c++             opción -+

caseful o
case-sensitive  opuesto de -i (por defecto)

case-insensitive o
caseless        opción -i

debug           opción -d
default         opuesto de la opción -s
ecs             opción -Ce
fast            opción -F
full            opción -f
interactive     opción -I
lex-compat      opción -l
meta-ecs        opción -Cm
perf-report     opción -p
read            opción -Cr
stdout          opción -t
verbose         opción -v
warn            opuesto de la opción -w
                (use "%option nowarn" para -w)

array           equivalente a "%array"
pointer         equivalente a "%pointer" (por defecto)
```

Algunas directivas '`%option`' ofrecen propiedades que de otra manera no están disponibles:

'`always-interactive`'
: ordena a flex que genere un analizador que siempre considere su entrada como "interactiva". Normalmente, sobre cada fichero de entrada nuevo el analizador llama a '`isatty()`' como intento para determinar si la entrada del analizador es interactiva y por lo tanto debería leer un caracter a la vez. Cuando esta opción se utilice, sin embargo, entonces no se hace tal llamada.

'`main`'
: ordena a flex que facilite un programa '`main()`' por defecto para el analizador, que simplemente llame a '`yylex()`'. Esta opción implica `noyywrap` (ver más abajo).

'`never-interactive`'
: ordena a flex que genere un analizador que nunca considere su entrada como "interactiva" (de nuevo, no se hace ninguna llamada a '`isatty()`'). Esta es la opuesta a '`always-interactive`'.

'`stack`'
: activa el uso de pilas de condiciones de arranque (ver Capítulo 9 [Condiciones de arranque], página 16).

'stdinit' si se establece (es decir, '%option stdinit') inicializa yyin e yyout a stdin y stdout, en lugar del que viene por defecto que es nil. Algunos pogramas de lex existentes dependen de este comportamiento, incluso si no sigue el ANSI C, que no requiere que stdin y stdout sean constantes en tiempo de compilación.

'yylineno'
ordena a flex a generar un analizador que mantenga el número de la línea actual leída desde su entrada en la variable global yylineno. Esta opción viene implícita con '%option lex-compat'.

'yywrap' si no se establece (es decir, '%option noyywrap'), hace que el analizador no llame a 'yywrap()' hasta el fin-de-fichero, pero simplemente asume que no hay más ficheros que analizar (hasta que el usuario haga apuntar yyin a un nuevo fichero y llame a 'yylex()' otra vez).

flex analiza las acciones de sus reglas para determinar si utiliza las propiedades REJECT o 'yymore()'. Las opciones reject e yymore están disponibles para ignorar sus decisiones siempre que use las opciones, o bien estableciendolas (p.ej., '%option reject') para indicar que la propiedad se utiliza realmente, o desactivándolas para indicar que no es utilizada (p.ej., '%option noyymore').

Tres opciones toman valores delimitados por cadenas, separadas por '=':

```
%option outfile="ABC"
```

es equivalente a '-oABC', y

```
%option prefix="XYZ"
```

es equivalente a '-PXYZ'. Finalmente,

```
%option yyclass="foo"
```

sólo se aplica cuando se genera un analizador en C++ (opción '-+'). Este informa a flex que ha derivado a 'foo' como una subclase de yyFlexLexer, así que flex pondrá sus acciones en la función miembro 'foo::yylex()' en lugar de 'yyFlexLexer::yylex()'. Este también genera una función miembro 'yyFlexLexer::yylex()' que emite un error en tiempo de ejecución (invocando a 'yyFlexLexer::LexerError()') si es llamada. Ver Capítulo 17 [Generando Escáneres en C++], página 40, para información adicional.

Están disponibles un número de opciones para los puristas de lint que desean suprimir la aparición de rutinas no necesarias en el analizador generado. Cada una de la siguientes, si se desactivan (p.ej., '%option nounput'), hace que la rutina correspondiente no aparezca en el analizador generado:

```
input, unput
yy_push_state, yy_pop_state, yy_top_state
yy_scan_buffer, yy_scan_bytes, yy_scan_string
```

(aunque 'yy_push_state()' y sus amigas no aparecerán de todas manera a menos que use '%option stack').

16 Consideraciones de rendimiento

El principal objetivo de diseño de `flex` es que genere analizadores de alto rendimiento. Este ha sido optimizado para comportarse bien con conjuntos grandes de reglas. Aparte de los efectos sobre la velocidad del analizador con las opciones de compresión de tablas '`-C`' anteriormente introducidas, hay un número de opciones/acciones que degradan el rendimiento. Estas son, desde la más costosa a la menos:

```
REJECT
%option yylineno
contexto posterior arbitrario

conjunto de patrones que requieren retroceso
%array
%option interactive
%option always-interactive

'^' operador de comienzo de línea
yymore()
```

siendo las tres primeras bastante costosas y las dos últimas bastante económicas. Fíjese también que '`unput()`' se implementa como una llamada de rutina que potencialmente hace bastante trabajo, mientras que '`yyless()`' es una macro bastante económica; así que si está devolviendo algún texto excedente que ha analizado, use '`yyless()`'.

`REJECT` debería evitarse a cualquier precio cuando el rendimiento es importante. Esta es una opción particularmente cara.

Es lioso deshacerse del retroceso y a menudo podría ser una cantidad de trabajo enorme para un analizador complicado. En principio, uno comienza utilizando la bandera '`-b`' para generar un archivo '`lex.backup`'. Por ejemplo, sobre la entrada

```
%%
foo        return TOK_KEYWORD;
foobar     return TOK_KEYWORD;
```

el fichero tiene el siguiente aspecto:

```
El estado #6 es no-aceptar -
 números de línea asociados a la regla:
       2       3
 fin de transiciones: [ o ]
 transiciones de bloqueo: fin de archivo (EOF) [ \001-n  p-\177 ]

El estado #8 es no-aceptar -
 números de línea asociados a la regla:
       3
 fin de transiciones: [ a ]
 transiciones de bloqueo: fin de archivo (EOF) [ \001-`  b-\177 ]

El estado #9 es no-aceptar -
 números de línea asociados a la regla:
       3
 fin de transiciones: [ r ]
 transiciones de bloqueo: fin de archivo (EOF) [ \001-q  s-\177 ]

Las tablas comprimidas siempre implican un retroceso.
```

Las primeras líneas nos dicen que hay un estado del analizador en el que se puede hacer una transición con una 'o' pero no sobre cualquier otro caracter, y que en ese estado el texto recientemente analizado no empareja con ninguna regla. El estado ocurre cuando se intenta emparejar las reglas encontradas en las líneas 2 y 3 en el fichero de entrada. Si el analizador está en ese estado y entoces lee cualquier cosa que no sea una 'o', tendrá que retroceder para encontrar una regla que empareje. Con un poco de análisis uno puede ver que este debe ser el estado en el que se está cuando se ha visto "fo". Cuando haya ocurrido, si se ve cualquier cosa que no sea una 'o', el analizador tendrá que retroceder para simplemente emparejar la 'f' (por la regla por defecto).

El comentario que tiene que ver con el Estado #8 indica que hay un problema cuando se analiza "foob". En efecto, con cualquier caracter que no sea una 'a', el analizador tendrá que retroceder para aceptar "foo". De forma similar, el comentario para el Estado #9 tiene que ver cuando se ha analizado "fooba" y no le sigue una 'r'.

El comentario final nos recuerda que no mecere la pena todo el trabajo para eliminar el retroceso de las reglas a menos que estemos usando '-Cf' ó '-CF', y que no hay ninguna mejora del rendimiento haciéndolo con analizadores comprimidos.

La manera de quitar los retrocesos es añadiendo reglas de "error":

```
%%
foo         return TOK_KEYWORD;
foobar      return TOK_KEYWORD;

fooba       |
foob        |
fo          {
            /* falsa alarma, realmente no es una palabra clave */
            return TOK_ID;
            }
```

La eliminación de retroceso en una lista de palabras clave también puede hacerse utilizando una regla "atrápalo-todo":

```
%%
foo         return TOK_KEYWORD;
foobar      return TOK_KEYWORD;

[a-z]+      return TOK_ID;
```

Normalmente esta es la mejor solución cuando sea adecuada.

Los mensajes sobre retrocesos tienden a aparecer en cascada. Con un conjunto complicado de reglas no es poco común obtener cientos de mensajes. Si uno puede descifrarlos, sin embargo, a menudo sólo hay que tomar una docena de reglas o algo así para eliminar los retrocesos (ya que es fácil cometer una equivocación y tener una regla de error que reconozca un token válido. Una posible característica futura de `flex` será añadir reglas automáticamente para eliminar el retroceso).

Es importante tener en cuenta que se obtienen los beneficios de eliminar el retroceso sólo si elimina *cada* instancia del retroceso. Dejar solamente una significa que no ha ganado absolutamente nada.

El contexto posterior *variable* (donde la parte delantera y posterior no tienen una longitud fija) supone casi la misma pérdida de rendimiento que `REJECT` (es decir, substanciales). Así que cuando sea posible una regla como esta:

```
%%
raton|rata/(gato|perro)   correr();
```

es mejor escribirla así:

```
%%
raton/gato|perro          correr();
rata/gato|perro           correr();
```

o así

```
%%
raton|rata/gato           correr();
raton|rata/perro          correr();
```

Fíjese que aquí la acción especial '|' *no* ofrece ningún ahorro, y puede incluso hacer las cosas peor (ver Capítulo 21 [Deficiencias / Errores], página 50).

Otro área donde el usuario puede incrementar el rendimiento del analizador (y una que es más fácil de implementar) surge del hecho que cuanto más tarde se empareje un token, más rápido irá el analizador. Esto es debido a que con tokens grandes el procesamiento de la mayoría de los caracteres de entrada tiene lugar en el (corto) bucle de análisis más interno, y no tiene que ir tan a menudo a hacer el trabajo de más para constituir el entorno del analizador (p.ej., `yytext`) para la acción. Recuerde el analizador para los comentarios en C:

```
%x comentario
%%
        int num_linea = 1;

"/*"         BEGIN(comentario);

<comentario>[^*\n]*
<comentario>"*"+[^*/\n]*
<comentario>\n             ++num_linea;
<comentario>"*"+"/"        BEGIN(INITIAL);
```

Esto podría acelerarse escribiéndolo como:

```
%x comentario
%%
        int num_linea = 1;

"/*"         BEGIN(comentario);

<comentario>[^*\n]*
<comentario>[^*\n]*\n      ++num_linea;
<comentario>"*"+[^*/\n]*
<comentario>"*"+[^*/\n]*\n ++num_linea;
<comentario>"*"+"/"        BEGIN(INITIAL);
```

Ahora en lugar de que cada línea nueva requiera el procesamiento de otra regla, el reconocimiento de las líneas nuevas se "distribuye" sobre las otras reglas para mantener el texto reconocido tan largo como sea posible. ¡Fíjese que el *añadir* reglas *no* ralentiza el analizador! La velocidad del analizador es independiente del número de reglas o (dadas las consideraciones dadas al inicio de esta sección) cuán complicadas sean las reglas respecto a operadores tales como '*' y '|'.

Un ejemplo final sobre la aceleración de un analizador: suponga que quiere analizar un fichero que contiene identificadores y palabras clave, una por línea y sin ningún caracter extraño, y reconocer todas las palabras clave. Una primera aproximación natural es:

```
%%
asm      |
auto     |
break    |
```

```
... etc ...
volatile |
while    /* es una palabra clave */

.|\n     /* no es una palabra clave */
```

Para eliminar el retroceso, introduzca una regla atrápalo-todo:

```
%%
asm      |
auto     |
break    |
... etc ...
volatile |
while    /* es una palabra clave */

[a-z]+   |
.|\n     /* no es una palabra clave */
```

Ahora, si se garantiza que hay exáctamente una palabra por línea, entonces podemos reducir el número total de emparejamientos por la mitad mezclando el reconocimiento de líneas nuevas con las de los otros tokens:

```
%%
asm\n    |
auto\n   |
break\n  |
... etc ...
volatile\n |
while\n  /* es una palabra clave */

[a-z]+\n |
.|\n     /* no es una palabra clave */
```

Uno tiene que ser cuidadoso aquí, ya que hemos reintroducido retroceso en el analizador. En particular, aunque *nosotros* sepamos que ahí nunca habrán otros caracteres en el flujo de entrada que no sean letras o líneas nuevas, `flex` no puede figurarse eso, y planeará la posible necesidad de retroceder cuando haya analizado un token como "auto" y el próximo caracter sea algo distinto a una línea nueva o una letra. Previamente este podría entonces emparejar la regla "auto" y estar todo hecho, pero ahora este no tiene una regla "auto", solamente una regla "auto\n". Para eliminar la posibilidad de retroceso, podríamos o bien duplicar todas las reglas pero sin línea nueva al final, o, ya que nunca esperamos encontrar tal entrada y por lo tanto ni cómo es clasificada, podemos introducir una regla atrápalo-todo más, esta que no incluye una línea nueva:

```
%%
asm\n    |
auto\n   |
break\n  |
... etc ...
volatile\n |
while\n  /* es una palabra clave */

[a-z]+\n |
[a-z]+   |
.|\n     /* no es una palabra clave */
```

Compilado con '`-Cf`', esto es casi tan rápido como lo que uno puede obtener de un analizador de `flex` para este problema en particular.

Una nota final: `flex` es lento cuando empareja NUL's, particularmente cuando un token contiene múltiples NUL's. Es mejor escribir reglas que emparejen *cortas* cantidades de texto si se anticipa que el texto incluirá NUL's a menudo.

Otra nota final en relación con el rendimiento: tal y como se mencionó en el Capítulo 6 [Cómo se Empareja la Entrada], página 9, el reajuste dinámico de `yytext` para acomodar tokens enormes es un proceso lento porque ahora requiere que el token (inmenso) sea reanalizado desde el principio. De esta manera si el rendimiento es vital, debería intentar emparejar "grandes" cantidades de texto pero no "inmensas" cantidades, donde el punto medio está en torno a los 8K caracteres/token.

17 Generando escáneres en C++

`flex` ofrece dos maneras distintas de generar analizadores para usar con C++. La primera manera es simplemente compilar un analizador generado por `flex` usando un compilador de C++ en lugar de un compilador de C. No debería encontrarse ante ningún error de compilación (por favor informe de cualquier error que encuentre a la dirección de correo electrónico dada en el Capítulo 23 [Autor], página 52). Puede entonces usar código C++ en sus acciones de las reglas en lugar de código C. Fíjese que la fuente de entrada por defecto para su analizador permanece como `yyin`, y la repetición por defecto se hace aún a `yyout`. Ambos permanecen como variables '`FILE *`' y no como `flujos` de C++.

También puede utilizar `flex` para generar un analizador como una clase de C++, utilizando la opción '`-+`' (o, equivalentemente, '`%option c++`'), que se especifica automáticamente si el nombre del ejecutable de flex finaliza con un '`+`', tal como `flex++`. Cuando se usa esta opcióx, flex establece por defecto la generación del analizador al fichero '`lex.yy.cc`' en vez de '`lex.yy.c`'. El analizador generado incluye el fichero de cabecera '`FlexLexer.h`', que define el interfaz con las dos clases de C++.

La primera clase, `FlexLexer`, ofrece una clase base abstracta definiendo la interfaz a la clase del analizador general. Este provee las siguientes funciones miembro:

'`const char* YYText()`'
: retorna el texto del token reconocido más recientemente, el equivalente a `yytext`.

'`int YYLeng()`'
: retorna la longitud del token reconocido más recientemente, el equivalente a `yyleng`.

'`int lineno() const`'
: retorna el número de línea de entrada actual (ver '`%option yylineno`'), ó 1 si no se usó '`%option yylineno`'.

'`void set_debug( int flag )`'
: activa la bandera de depuración para el analizador, equivalente a la asignación de `yy_flex_debug` (ver Sección 15.2 [Opciones], página 28). Fíjese que debe construir el analizador utilizando '`%option debug`' para incluir información de depuración en este.

'`int debug() const`'
: retorna el estado actual de la bandera de depuración.

También se proveen funciones miembro equivalentes a '`yy_switch_to_buffer()`', '`yy_create_buffer()`' (aunque el primer argumento es un puntero a objeto '`istream*`' y no un '`FILE*`'), '`yy_flush_buffer()`', '`yy_delete_buffer()`', y '`yyrestart()`' (de nuevo, el primer argumento es un puntero a objeto '`istream*`').

La segunda clase definida en '`FlexLexer.h`' es `yyFlexLexer`, que se deriva de `FlexLexer`. Esta define las siguientes funciones miembro adicionales:

'`yyFlexLexer( istream* arg_yyin = 0, ostream* arg_yyout = 0 )`'
: construye un objeto `yyFlexLexer` usando los flujos dados para la entrada y salida. Si no se especifica, los flujos se establecen por defecto a `cin` y `cout`, respectivamente.

'`virtual int yylex()`'
: hace el mismo papel que '`yylex()`' en los analizadores de flex ordinarios: analiza el flujo de entrada, consumiendo tokens, hasta que la acción de una regla retorne un valor. Si usted deriva una subclase S a partir de `yyFlexLexer` y quiere acceder a las funciones y variables miembro de S dentro de '`yylex()`', entonces necesita utilizar '`%option yyclass="S"`' para informar a `flex` que estará utilizando esa subclase en lugar de `yyFlexLexer`. Es este caso, en vez de generar

'yyFlexLexer::yylex()', flex genera 'S::yylex()' (y también genera un substituto 'yyFlexLexer::yylex()' que llama a 'yyFlexLexer::LexerError()' si se invoca).

'virtual void switch_streams(istream* new_in = 0, ostream* new_out = 0)'
: reasigna yyin a new_in (si no es nulo) e yyout a new_out (idem), borrando el buffer de entrada anterior si se reasigna yyin.

'int yylex(istream* new_in, ostream* new_out = 0)'
: primero conmuta el flujo de entrada via 'switch_streams(new_in, new_out)' y entonces retorna el valor de 'yylex()'.

Además, yyFlexLexer define las siguientes funciones virtuales protegidas que puede redefinir en clases derivadas para adaptar el analizador:

'virtual int LexerInput(char* buf, int max_size)'
: lee hasta 'max_size' caracteres en *buf* y devuelve el número de caracteres leídos. Para indicar el fin-de-la-entrada, devuelve 0 caracteres. Fíjese que los analizadores "interactivos" (ver las banderas '-B' y '-I') definen la macro YY_INTERACTIVE. Si usted redefine LexerInput() y necesita tomar acciones distintas dependiendo de si el analizador está analizando una fuente de entrada interactivo o no, puede comprobar la presencia de este nombre mediante '#ifdef'.

'virtual void LexerOutput(const char* buf, int size)'
: escribe a la salida *size* caracteres desde el buffer *buf*, que, mientras termine en NUL, puede contener también NUL's "internos" si las reglas del analizador pueden emparejar texto con NUL's dentro de este.

'virtual void LexerError(const char* msg)'
: informa con un mensaje de error fatal. La versión por defecto de esta función escribe el mensaje al flujo cerr y finaliza.

Fíjese que un objeto yyFlexLexer contiene su estado de análisis *completo*. Así puede utilizar tales objetos para crear analizadore reentrantes. Puede hacer varias instancias de la misma clase yyFlexLexer, y puede combinar varias clases de analizadores en C++ conjuntamente en el mismo programa usando la opción '-P' comentada anteriormente.

Finalmente, note que la característica '%array' no está disponible en clases de analizadores en C++; debe utilizar '%pointer' (por defecto).

Aquí hay un ejemplo de un analizador en C++ simple:

```
// Un ejemplo del uso de la clase analizador en C++ de flex.

%{
int mylineno = 0;
%}

string  \"[^\n"]+\"

ws      [ \t]+

alpha   [A-Za-z]
dig     [0-9]
name    ({alpha}|{dig}|\$)({alpha}|{dig}|[_.\-/$])*
num1    [-+]?{dig}+\.?([eE][-+]?{dig}+)?
num2    [-+]?{dig}*\.{dig}+([eE][-+]?{dig}+)?
number  {num1}|{num2}
```

```
%%

{ws}    /* evita los espacios en blanco y tabuladores */

"/*"    {
        int c;

        while((c = yyinput()) != 0)
            {
            if(c == '\n')
                ++mylineno;

            else if(c == '*')
                {
                if((c = yyinput()) == '/')
                    break;
                else
                    unput(c);
                }
            }
        }

{number}  cout << "número " << YYText() << '\n';

\n        mylineno++;

{name}    cout << "nombre " << YYText() << '\n';

{string}  cout << "cadena " << YYText() << '\n';

%%

int main( int /* argc */, char** /* argv */ )
    {
    FlexLexer* lexer = new yyFlexLexer;
    while(lexer->yylex() != 0)
        ;
    return 0;
    }
```

Si desea crear varias (diferentes) clases analizadoras, use la bandera '`-P`' (o la opción '`prefix=`') para renombrar cada `yyFlexLexer` a algún otro `xxFlexLexer`. Entonces puede incluir '`<FlexLexer.h>`' en los otros ficheros fuente una vez por clase analizadora, primero renombrando `yyFlexLexer` como se presenta a continuación:

```
#undef yyFlexLexer
#define yyFlexLexer xxFlexLexer
#include <FlexLexer.h>

#undef yyFlexLexer
#define yyFlexLexer zzFlexLexer
#include <FlexLexer.h>
```

si, por ejemplo, usted utilizó '`%option prefix="xx"`' para uno de sus analizadores y '`%option prefix="zz"`' para el otro.

IMPORTANTE: la forma actual de la clase analizadora es *experimental* y podría cambiar considerablemente entre versiones principales.

18 Incompatibilidades con lex y POSIX

`flex` es una reescritura de la herramienta `lex` del Unix de AT&T (aunque las dos implementaciones no comparten ningún código), con algunas extensiones e incompatibilidades, de las que ambas conciernen a aquellos que desean escribir analizadores aceptables por cualquier implementación. Flex sigue completamente la especificación POSIX de `lex`, excepto que cuando se utiliza '`%pointer`' (por defecto), una llamada a '`unput()`' destruye el contenido de `yytext`, que va en contra de la especificación POSIX.

En esta sección comentaremos todas las áreas conocidas de incompatibilidades entre flex, lex de AT&T, y la especificación POSIX.

La opción '`-l`' de `flex` activa la máxima compatibilidad con la implementación original de `lex` de AT&T, con el coste de una mayor pérdida de rendimiento en el analizador generado. Indicamos más abajo qué incompatibilidades pueden superarse usando la opción '`-l`'.

`flex` es totalmente compatible con `lex` con las siguientes excepciones:

- La variable interna del analizador de `lex` sin documentar `yylineno` no se ofrece a menos que se use '`-l`' ó '`%option yylineno`'. `yylineno` debería gestionarse por buffer, en lugar de por analizador (simple variable global). `yylineno` no es parte de la especificación POSIX.
- La rutina '`input()`' no es redefinible, aunque podría invocarse para leer los caracteres que siguen a continuación de lo que haya sido reconocido por una regla. Si '`input()`' se encuentra con un fin-de-fichero se realiza el procesamiento de '`yywrap()`' normal. '`input()`' retorna un fin-de-fichero "real" como `EOF`.

  La entrada en su lugar se controla definiendo la macro `YY_INPUT`.

  La restricción de `flex` de que '`input()`' no puede redefinirse va de acuerdo a la especificación POSIX, que simplemente no especifica ninguna manera de controlar la entrada del analizador que no sea haciendo una asignación inicial a `yyin`.
- La rutina '`unput()`' no es redefinible. Esta restricción va de acuerdo a POSIX.
- Los analizadores de `flex` no son tan reentrantes como los analizadores de `lex`. En particular, si tiene un analizador interactivo y un gestor de interrupción con long-jumps fuera del analizador, y el analizador a continuación se invoca de nuevo, podría obtener el siguiente mensaje:

  ```
  fatal flex scanner internal error--end of buffer missed
  ```

  Para volver al analizador, primero utilice

  ```
  yyrestart( yyin );
  ```

  Vea que esta llamada eliminará cualquier entrada en el buffer; normalmente esto no es un problema con un analizador interactivo.

  Dese cuenta también de que las clases analizadoras en C++ *son* reentrantes, así que si usar C++ es una opción para usted, debería utilizarla. Ver Capítulo 17 [Generando Escáneres en C++], página 40, para los detalles.
- '`output()`' no se provee. La salida desde la macro '`ECHO`' se hace al puntero de fichero `yyout` (por defecto a `stdout`).

  '`output()`' no es parte de la especificación POSIX.
- `lex` no acepta condiciones de arranque exclusivas (%x), aunque están en la especificación POSIX.
- Cuando se expanden las definiciones, `flex` las encierra entre paréntesis. Con lex, lo siguiente:

  ```
  NOMBRE    [A-Z][A-Z0-9]*
  %%
  foo{NOMBRE}?      printf( "Lo encontró\n" );
  %%
  ```

no reconocerá la cadena "foo" porque cuando la macro se expanda la regla es equivalente a "foo[A-Z][A-Z0-9]*?" y la precedencia es tal que el '?' se asocia con "[A-Z0-9]*". Con flex, la regla se expandirá a "foo([A-Z][A-Z0-9]*)?" y así la cadena "foo" se reconocerá.

Fíjese que si la definición comienza con '^' o finaliza con '$' entonces *no* se expande con paréntesis, para permitir que estos operadores aparezcan en las definiciones sin perder su significado especial. Pero los operadores '<s>, /', y '<<EOF>>' no pueden utilizarse en una definición de flex.

El uso de '-l' produce en el comportamiendo de lex el no poner paréntesis alrededor de la definición.

La especificación de POSIX dice que la definición debe ser encerrada entre paréntesis.

- Algunas implementaciones de lex permiten que la acción de una regla comience en una línea separada, si el patrón de la regla tiene espacios en blanco al final:

  ```
  %%
  foo|bar<espacio aquí>
    { foobar_action(); }
  ```

  flex no dispone de esta propiedad.
- La opción '%r' de lex (generar un analizador Ratfor) no se ofrece. No es parte de la especificación de POSIX.
- Después de una llamada a 'unput()', el contenido de yytext está indefinido hasta que se reconozca el próximo token, a menos que el analizador se haya construido usando '%array'. Este no es el caso de lex o la especificación de POSIX. La opción '-l' elimina esta incompatibilidad.
- La precedencia del operador '{}' (rango numérico) es diferente. lex interpreta "abc{1,3}" como "empareja uno, dos, o tres apariciones de 'abc'", mientras que flex lo interpreta como "empareja 'ab' seguida de una, dos o tres apariciones de 'c'". Lo último va de acuerdo con la especificación de POSIX.
- La precedencia del operador '^' es diferente. lex interpreta "^foo|bar" como "empareja bien 'foo' al principio de una línea, o 'bar' en cualquier lugar", mientras que flex lo interpreta como "empareja 'foo' o 'bar' si vienen al principio de una línea". Lo último va de acuerdo con la especificación de POSIX.
- Las declaraciones especiales del tamaño de las tablas tal como '%a' que reconoce lex no se requieren en los analizadores de flex; flex los ignora.
- El identificador FLEX_SCANNER se #define de manera que los analizadores podrían escribirse para ser procesados con flex o con lex. Los analizadores también incluyen YY_FLEX_MAJOR_VERSION y YY_FLEX_MINOR_VERSION indicando qué versión de flex generó el analizador (por ejemplo, para la versión 2.5, estas definiciones serán 2 y 5 respectivamente).

Las siguientes propiedades de flex no se incluyen en lex o la especificación POSIX:

```
analizadores en C++
%option
ámbitos de condiciones de arranque
pilas de condiciones de arranque
analizadores interactivos/no-interactivos
yy_scan_string() y sus amigas
yyterminate()
yy_set_interactive()
yy_set_bol()
YY_AT_BOL()
<<EOF>>
<*>
YY_DECL
```

```
YY_START
YY_USER_ACTION
YY_USER_INIT
directivas #line
%{}'s alrededor de acciones
varias acciones en una línea
```

más casi todas las banderas de flex. La última propiedad en la lista se refiere al hecho de que con `flex` puede poner varias acciones en la misma línea, sepradas con punto y coma, mientras que con `lex`, lo siguiente

```
foo    handle_foo(); ++num_foos_seen;
```

se trunca (sorprendentemente) a

```
foo    handle_foo();
```

`flex` no trunca la acción. Las acciones que no se encierran en llaves simplemente se terminan al final de la línea.

19 Diagnósticos

'aviso, la regla no se puede aplicar'
: indica que la regla dada no puede emparejarse porque sigue a otras reglas que siempre emparejarán el mismo texto que el de esta. Por ejemplo, en el siguiente ejemplo "foo" no puede emparejarse porque viene después de una regla "atrápalotodo" para identificadores:

```
[a-z]+    obtuvo_identificador();
foo       obtuvo_foo();
```

El uso de `REJECT` en un analizador suprime este aviso.

'aviso, se ha especificado la opción -s pero se puede aplicar la regla por defecto'
: significa que es posible (tal vez únicamente en una condición de arranque en particular) que la regla por defecto (emparejar cualquier caracter simple) sea la única que emparejará una entrada particular. Ya que se indicó '-s', presumiblemente esto no es lo que se pretendía.

'definición no definida reject_used_but_not_detected'
'definición no definida yymore_used_but_not_detected'
: Estos errores pueden suceder en tiempo de compilación. Indican que el analizador usa `REJECT` o 'yymore()' pero que `flex` falló en darse cuenta del hecho, queriendo decir que `flex` analizó las dos primeras secciones buscando apariciones de estas acciones y falló en encontrar alguna, pero que de algún modo se le han colado (por medio de un archivo #include, por ejemplo). Use '%option reject' ó '%option yymore' para indicar a flex que realmente usa esta funcionalidad.

'flex scanner jammed'
: un analizador compilado con '-s' ha encontrado una cadena de entrada que no fue reconocida por niguna de sus reglas. Este error puede suceder también debido a problemas internos.

'token too large, exceeds YYLMAX'
: su analizador usa '%array' y una de sus reglas reconoció una cadena más grande que la constante `YYLMAX` (8K bytes por defecto). Usted puede incrementar el valor haciendo un #define `YYLMAX` en la sección de definiciones de su entrada de `flex`.

'el analizador requiere la opción -8 para poder usar el carácter '*x*''
: La especificación de su analizador incluye el reconocimiento del caracter de 8-bits *x* y no ha especificado la bandera -8, y su analizador por defecto está a 7-bits porque ha usado las opciones '-Cf' ó '-CF' de compresión de tablas. Ver Sección 15.2 [Opciones], página 28, el comentario de la bandera '-7' para los detalles.

'flex scanner push-back overflow'
: usted utilizó 'unput()' para devolver tanto texto que el buffer del analizador no pudo mantener el texto devuelto y el token actual en `yytext`. Idealmente el analizador debería ajustar dinámicamente el buffer en este caso, pero actualmente no lo hace.

'input buffer overflow, can't enlarge buffer because scanner uses REJECT'
: el analizador estaba intentando reconocer un token extremadamente largo y necesitó expandir el buffer de entrada. Esto no funciona con analizadores que usan `REJECT`.

'fatal flex scanner internal error--end of buffer missed'
: Esto puede suceder en un analizador que se reintroduce después de que un longjump haya saltado fuera (o sobre) el registro de activación del analizador. Antes de reintroducir el analizador, use:

```
yyrestart( yyin );
```

o, como se comentó en el Capítulo 17 [C++], página 40, cambie y use el analizador como clase de C++.

'`too many start conditions in <> construct!`'
: ha listado más condiciones de arranque en una construcción `<>` que las que existen (así que tuvo que haber listado al menos una de ellas dos veces).

20 Ficheros

'-lfl' librería con la que los analizadores deben enlazarse.

'lex.yy.c'
: analizador generado (llamado 'lexyy.c' en algunos sistemas).

'lex.yy.cc'
: clase generada en C++ con el analizador, cuando se utiliza '-+'.

'<FlexLexer.h>'
: fichero de cabecera definiendo la clase base del analizador en C++, FlexLexer, y su clase derivada, yyFlexLexer.

'flex.skl'
: esqueleto del analizador. Este fichero se utiliza únicamente cuando se construye flex, no cuando flex se ejecuta.

'lex.backup'
: información de los retrocesos para la bandera '-b' (llamada 'lex.bck' en algunos sistemas).

21 Deficiencias / Errores

Algunos patrones de contexto posterior no pueden reconocerse correctamente y generan mensajes de aviso ("contexto posterior peligroso"). Estos son patrones donde el final de la primera parte de la regla reconoce el comienzo de la segunda parte, tal como "zx*/xy*", donde el 'x*' reconoce la '`x`' al comienzo del contexto posterior. (Fíjese que el borrador de POSIX establece que el texto reconocido por tales patrones no está definido.)

Para algunas reglas de contexto posterior, partes que son de hecho de longitud fija no se reconocen como tales, resultando en la pérdida de rendimiento mencionada anteriormente. En particular, las partes que usan '`|`' o `{n}` (tales como "foo{3}") siempre se consideran de longitud variable.

La combinación de contexto posterior con la acción especial '`|`' puede producir que el contexto posterior *fijo* se convierta en contexto posterior *variable* que es más caro. Por ejemplo, en lo que viene a continuación:

```
%%
abc      |
xyz/def
```

El uso de '`unput()`' invalida yytext e yyleng, a menos que se use la directiva '`%array`' o la opción '`-l`'.

La concordancia de patrones de NUL's es substancialmente más lento que el reconocimiento de otros caracteres.

El ajuste dinámico del buffer de entrada es lento, ya que conlleva el reanálisis de todo el texto reconocido hasta entonces por el (generalmente enorme) token actual.

Debido al uso simultáneo de buffers de entrada y lecturas por adelantado, no puede entremezclar llamadas a rutinas de <stdio.h>, tales como, por ejemplo, '`getchar()`', con reglas de `flex` y esperar que funcione. Llame a '`input()`' en su lugar.

La totalidad de las entradas de la tabla listada por la bandera '`-v`' excluye el número de entradas en la tabla necesarias para determinar qué regla ha sido emparejada. El número de entradas es igual al número de estados del DFA si el analizador no usa `REJECT`, y algo mayor que el número de estados si se usa.

`REJECT` no puede usarse con las opciones '`-f`' ó '`-F`'.

El algoritmo interno de `flex` necesita documentación.

22 Ver también

`lex`(1), `yacc`(1), `sed`(1), `awk`(1).

John Levine, Tony Mason, and Doug Brown: Lex & Yacc, O'Reilly and Associates. Esté seguro de obtener la 2a. edición.

M. E. Lesk and E. Schmidt, LEX - Lexical Analyzer Generator

Alfred Aho, Ravi Sethi and Jeffrey Ullman: Compilers: Principles, Techniques and Tools; Addison-Wesley (1986) —Edición en castellano: Compiladores: Principios, Técnicas y Herramientas, Addison-Wesley Iberoamericana, S.A. (1990). Describe las técnicas de concordancia de patrones usadas por `flex` (autómata finito determinista).

23 Autor

Vern Paxson, con la ayuda de muchas ideas e inspiración de Van Jacobson. Versión original por Jef Poskanzer. La representación de tablas rápidas es una implementación parcial de un diseño hecho por Van Jacobson. La implementación fue hecha por Kevin Gong y Vern Paxson.

Agradecimientos a los muchos `flex` beta-testers, feedbackers, y contribuidores, especialmente a Francois Pinard, Casey Leedom, Robert Abramovitz, Stan Adermann, Terry Allen, David Barker-Plummer, John Basrai, Neal Becker, Nelson H.F. Beebe, '`benson@odi.com`', Karl Berry, Peter A. Bigot, Simon Blanchard, Keith Bostic, Frederic Brehm, Ian Brockbank, Kin Cho, Nick Christopher, Brian Clapper, J.T. Conklin, Jason Coughlin, Bill Cox, Nick Cropper, Dave Curtis, Scott David Daniels, Chris G. Demetriou, Theo Deraadt, Mike Donahue, Chuck Doucette, Tom Epperly, Leo Eskin, Chris Faylor, Chris Flatters, Jon Forrest, Jeffrey Friedl, Joe Gayda, Kaveh R. Ghazi, Wolfgang Glunz, Eric Goldman, Christopher M. Gould, Ulrich Grepel, Peer Griebel, Jan Hajic, Charles Hemphill, NORO Hideo, Jarkko Hietaniemi, Scott Hofmann, Jeff Honig, Dana Hudes, Eric Hughes, John Interrante, Ceriel Jacobs, Michal Jaegermann, Sakari Jalovaara, Jeffrey R. Jones, Henry Juengst, Klaus Kaempf, Jonathan I. Kamens, Terrence O Kane, Amir Katz, '`ken@ken.hilco.com`', Kevin B. Kenny, Steve Kirsch, Winfried Koenig, Marq Kole, Ronald Lamprecht, Greg Lee, Rohan Lenard, Craig Leres, John Levine, Steve Liddle, David Loffredo, Mike Long, Mohamed el Lozy, Brian Madsen, Malte, Joe Marshall, Bengt Martensson, Chris Metcalf, Luke Mewburn, Jim Meyering, R. Alexander Milowski, Erik Naggum, G.T. Nicol, Landon Noll, James Nordby, Marc Nozell, Richard Ohnemus, Karsten Pahnke, Sven Panne, Roland Pesch, Walter Pelissero, Gaumond Pierre, Esmond Pitt, Jef Poskanzer, Joe Rahmeh, Jarmo Raiha, Frederic Raimbault, Pat Rankin, Rick Richardson, Kevin Rodgers, Kai Uwe Rommel, Jim Roskind, Alberto Santini, Andreas Scherer, Darrell Schiebel, Raf Schietekat, Doug Schmidt, Philippe Schnoebelen, Andreas Schwab, Larry Schwimmer, Alex Siegel, Eckehard Stolz, Jan-Erik Strvmquist, Mike Stump, Paul Stuart, Dave Tallman, Ian Lance Taylor, Chris Thewalt, Richard M. Timoney, Jodi Tsai, Paul Tuinenga, Gary Weik, Frank Whaley, Gerhard Wilhelms, Kent Williams, Ken Yap, Ron Zellar, Nathan Zelle, David Zuhn, y aquellos cuyos nombres han caído bajo mis escasas dotes de archivador de correo pero cuyas contribuciones son apreciadas todas por igual.

Agradecimientos a Keith Bostic, Jon Forrest, Noah Friedman, John Gilmore, Craig Leres, John Levine, Bob Mulcahy, G.T. Nicol, Francois Pinard, Rich Salz, y a Richard Stallman por la ayuda con diversos quebraderos de cabeza con la distribución.

Agradecimientos a Esmond Pitt y Earle Horton por el soporte de caracteres de 8-bits; a Benson Margulies y a Fred Burke por el soporte de C++; a Kent Williams y a Tom Epperly por el soporte de la clase de C++; a Ove Ewerlid por el soporte de NUL's; y a Eric Hughes por el soporte de múltiples buffers.

Este trabajo fue hecho principalmente cuando yo estaba con el Grupo de Sistemas de Tiempo Real en el Lawrence Berkeley Laboratory en Berkeley, CA. Muchas gracias a todos allí por el apoyo que recibí.

Enviar comentarios a Vern Paxson (`vern@ee.lbl.gov`).

Sobre esta traducción enviar comentarios a Adrián Pérez Jorge (`alu1415@csi.ull.es`).

Índice

%

-

<

A

B

C

D

E

F

G

I

L

M

N

O

P

R

S

T

U

V

Y

Tabla de Contenido