8.1. Introdución

El lex es un generador de programas diseñado para el proceso léxico de cadenas de caracteres de input. El programa acepta una especificación, orientada a resolver un problema de alto nivel para comparar literales de caracteres, y produce un programa C que reconoce expresiones regulares. Estas expresiones las especifica el usuario en las especificaciones fuente que se le dan al lex. El código lex reconoce estas expresiones en una cadena de input y divide este input en cadenas de caracteres que coinciden con las expresiones. En los bordes entre los literales, se ejecutan las secciones de programas proporcionados por el usuario. El fichero fuente lex asocia las expresiones regulares y los fragmentos de programas. Puesto que cada expresión aparece en el input del programa escrito por el lex, se ejecuta el fragmento correspondiente.

El usuario proporciona el código adicional necesario para completar estas funciones, incluyendo código escrito por otros generadores. El programa que reconoce las expresiones se genera en forma de fragmentos de programa C del usuario, El lex no es un lenguaje completo sino un generador que representa una cualidad de un nuevo lenguaje que se añade al leguaje de programación C.

El lex convierte las expresiones y acciones del usuario (llamadas fuente en este capítulo) en un programa C llamado yylex. El programa yylex reconoce expresiones en un flujo (llamado input en este capítulo) y lleva a cabo las acciones especificadas para cada expresión a medida que se va detectando.

Considere un programa para borrar del input todos los espacios en blanco y todos los tabuladores de los extremos de las líneas. Las líneas siguientes:

```
%%
[b\ t]+ $     ;
```

es todo lo que se requiere. El programa contiene un delimitado %% para marcar el principio de las órdenes, y una orden. Esta orden contiene una expresión que coincide con una o más apariciones de los caracteres espacio en blanco o tabulador (escrito \ t para que se vea con mayor claridad, de acuerdo con la convención del lenguaje C) justo antes del final de una línea. Los corchetes indican la clase del carácter compuesto de espacios en blanco y tabuladores; el + indica uno o más del item anterior; y el signo de dólar ($) indica el final de la línea. No se especifica ninguna acción, por lo tanto el programa generado por el lex ignorará estos caracteres. Todo lo demás se copiará . Para cambiar cualquier cadena de caracteres en blanco o tabuladores que queden en un solo espacio en blanco, añada otra orden:

```
%%
[b\ t]+$      ;
[b\ t] +      printf (“     ”);
```

La automatización generada por este fuente explora ambas órdenes a la vez, observa la terminación de la cadena de espacios o tabuladores haya o no un carácter newline, y después ejecuta la acción de la orden deseada. La primera orden coincide con todas las cadenas de caracteres de espacios en blanco o tabuladores hasta el final de las líneas, y la segunda orden coincide con todos los literales restantes de espacios o tabuladores.

El lex se puede usar sólo para transformaciones sencillas, o por análisis o estadísticas buscando en un nivel léxico. El lex también se puede usar con un generador reconocedor para llevar a cabo la fase de análisis léxico; es especialmente fácil hacer que el lex y el yacc funcionen juntos. Los programas lex reconocen sólo expresiones regulares; yacc escribe reconocedores que aceptan una amplia clase de gramáticas de texto libre, pero que requieren un analizador de nivel bajo para reconocer tokens de input. Por lo tanto, a menudo es conveniente una combinación del lex y del yacc. Cuando se usa como un preprocesador para un generador, el lex se usa para dividir el input, y el generador de reconocimiento asigna una estructura a las piezas resultantes. Los programas adicionales, escritos por otros generadores o a mano, se pueden añadir fácilmente a programas que han sido escritos por el lex. Los usuarios del yacc se darán cuenta de que el nombre yylex es el que el yacc da a su analizador léxico, de forma que el uso de este nombre por el lex simplifica el interface.

El lex genera un autómata finito partiendo de expresiones regulares del fuente. El autómata es interpretado, en vez de compilado, para ahorrar espacio. El resultado es todavía un analizador rápido. En particular, el tiempo que utiliza un programa lex para reconocer y dividir una cadena de input es proporcional a la longitud del input. El número de órdenes lex o la complejidad de las órdenes no es importante para determinar la velocidad, a no ser que las órdenes que incluyan contexto posterior requieran una cantidad importante de exploración. Lo que aumenta con el número y complejidad de las órdenes es el tamaño del autómata finito, y por lo tanto el tamaño del programa generado por el lex.

En el programa escrito por el lex, los fragmentos del usuario (representando acciones que se van a llevar a cabo a medida que se encuentra cada expresión) se colectan como casos de un intercambio. El intérprete del autómata dirige el flujo de control. Se proporciona la oportunidad al usuario para insertar declaraciones o sentencias adicionales en la rutina que contiene las acciones, o para añadir subrutinas fuera de esta rutina de acción.

El lex no está limitado a fuente que se puede interpretar sobre la base de un posible carácter. Por ejemplo, si hay dos órdenes una que busca ab y la otra que busca abcdefg, y la cadena de caracteres del input es abcdefh, el lex reconocerá ab y dejará el puntero del input justo delante de cd. Tal precaución es más costosa que el proceso de lenguajes más sencillos.

8.2. Formato fuente del lex

El formato general de la fuente lex es:

```
{definiciones}
%%
{órdenes}
%%
{subrutinas del usuario}
```

donde las definiciones y las subrutinas del usuarios se omiten a menudo. El segundo %% es opcional, pero el primero se requiere para marcar el principio de las órdenes. El programa lex mínimo absoluto es por lo tanto

```
%%
```

(sin definiciones, ni órdenes) lo cual se traduce en un programa que copia el input en el output sin variar.

En el formato del programa lex que se mostró anteriormente, las órdenes representan las decisiones de control del usuario. Estas forman una tabla en la cual la columna izquierda contiene expresiones regulares y la columna derecha contiene decisiones, fragmentos de programas que se ejecutarán cuando se reconozcan las expresiones. Por lo tanto la siguiente orden individual puede aparecer:

```
integer   printf(“ localizada palabra reservada INT ”) ;
```

Esto busca el literal “integer” en el input e imprime el mensaje:

```
localizada palabra reservada INT
```

siempre que aparezca en el texto de input. En este ejemplo la función de librería printf() se usa para imprimir la cadena de caracteres o literal. El final de la expresión regular lex se indica por medio del primer carácter espacio en blanco o tabulador. Si la acción es simplemente una sola expresión C, se puede especificar en el lado derecho de la línea; si es compuesta u ocupa más de una línea, deberá incluirse entre llaves. Como ejemplo un poco más útil, suponga que se desea cambiar un número de palabras de la ortografía Británica a la Americana. Las órdenes lex tales como

```
colour          printf(“color”);
mechanise       printf(“mechanize”);
petrol          printf(“gas”);
```

será una forma de empezar. Estas órdenes no son suficientes puesto que la palabra petroleum se convertirá en gaseum; una forma de proceder con tales problemas se describe en una sección posterior.

8.3. Expresiones del lex

Una expresión especifica un conjunto de literales que se van a comparar. Esta contiene caracteres de texto (que coinciden con los caracteres correspondientes del literal que se está comparando) y caracteres operador (estos especifican repeticiones, selecciones, y otras características). Las letras del alfabeto y los dígitos son siempre caracteres de texto. Por lo tanto, la expresión

integer

Coincide con el literal "integer" siempre que éste aparezca y la expresión

a57d

busca el literal a57d.

Los caracteres operador son:

" \ [ ] ^ - ? . * + | ( ) $ / { } % < >

Si cualquiera de estos caracteres se va a usar literalmente, es necesario incluirlos individualmente entre caracteres barra invertida (\ ) o como un grupo dentro de comillas (").

El operador comillas **(")** indica que siempre que esté incluido dentro de un par de comillas se va a tomar como un carácter de texto. Por lo tanto

xyz"++"

coincide con el literal xyz++ cuando aparezca. Nótese que una parte del literal puede estar entre comillas. No produce ningún efecto y es innecesario poner entre comillas caracteres de texto normal; la expresión

"xyz++"

es la misma que la anterior. Por lo tanto poniendo entre comillas cada carácter no alfanumérico que se está usando como carácter de texto, no es necesario memorizar la lista anterior de caracteres operador.

Un carácter operador también se puede convertir en un carácter de texto poniéndole delante una barra invertida **(\ )** como en

xyz\+\+

el cual, aunque menos legible, es otro equivalente de las expresiones anteriores. Este mecanismo también se puede usar para incluir un espacio en blanco dentro de una expresión; normalmente, según se explicaba anteriormente, los espacios en blanco y los tabuladores terminan una orden. Cualquier carácter en blanco que no esté contenido entre corchete tiene que ponerse entre comillas.

Se reconocen barios escapes C normales con la barra invertida (\ ):

\ n	newline
\ t	tabulador
\ b	backspace
\ \	barra invertida

Puesto que el carácter newline es ilegal en una expresión, es necesario usar n; no se requiere dar escape al carácter tabulador y el backspace. Cada carácter excepto el espacio en blanco, el tabulador y el newline y la lista anterior es siempre un carácter de texto.

8.4. Llamar al lex.

Hay dos pasos al compilar un programa fuente lex Primero, el programa lex fuente tiene que ser convertido en un programa regenerado en el lenguaje de propósito general. Entonces éste programa tiene que ser compilado y cargado, normalmente con una librería de subrutinas lex. El programa generado está en un fichero llamado lex.yy.c. La librería I/O está definida en términos de la librería estándar C.

A la librería se accede por medio del flag de la carga -ll. Por lo tanto un conjunto de comandos apropiados es

```
lex source
cc lex.yy.c –ll
```

El programa resultante se pone en el fichero usual a.out para ser ejecutado posteriormente. Para usar el lex con el yacc ver la sección " lex y Yacc" de este capítulo y en el capitulo 9, " Yacc: el Compilador-Compilador", Aunque las rutinas I/O por defecto del lex usan la librería estándar C, el autómata del lex no lo hace. Si se especifican las versiones privadas de input, output, y unput, la librería se puede evitar.

8.5. Especificación de clases de caracteres.

Las clases de caracteres se pueden especificar usando corchetes: **[y]**. La construcción

```
[ abc ]
```

coincide con cualquier carácter, que pueda se una a, b, o c. Dentro de los corchetes, la mayoría de los significados de los operadores se ignoran. Sólo tres caracteres son especiales: éstos son la barra invertida **(\ )**, el guión **(-)**, y el signo de intercalación **(^)**. El carácter guión indica rangos, por ejemplo

```
[ a-z0-9<>_ ]
```

indica la clase de carácter que contiene todos los caracteres en minúsculas, los dígitos, los ángulos y el subrayado. Los rangos se pueden especificar en cualquier orden. Usando el guión entre cualquier par de caracteres que ambos no sean letras mayúsculas, letras minúsculas, o dígitos, depende de la implementación y produce un mensaje de aviso. Si se desea incluir el guión en una clase de caracteres, éste deberá ser el primero o el último; por lo tanto

```
[ -+0-9 ]
```

coincide con todos los dígitos y los signos más y menos.

En las clases de caracteres, el operador **(^)** debe aparecer como el primer carácter después del corchete izquierdo; esto indica que el literal resultante va a ser complementado con respecto al conjunto de caracteres del ordenador. Por lo tanto

```
[ ^abc ]
```

coincide con todos los caracteres excepto a, b, o c, incluyendo todos los caracteres especiales o de control; o

```
[ ^a-zA-Z ]
```

es cualquier carácter que no sea una letra. El carácter barra invertida (\ ) proporciona un mecanismo de escape dentro de los corchete de clases de caracteres, de forma que éstos se pueden introducir literalmente precediéndolos con este carácter.

8.6. Especificación de un carácter arbitrario

Para hacer coincidir casi con cualquier carácter, el punto **(.)** designa la clase de todos los caracteres excepto un carácter newline. Hacer escape en octal es posible, aunque esto no es portable. Por ejemplo

```
[ \ 40 - \ 176 ]
```

coincide con todos los caracteres imprimibles del conjunto de caracteres ASCII, desde el octal 40 (espacio en blanco) hasta el octal 176 (la tilde).

8.7. Especificar expresiones opcionales.

El operador signo de interrogación **(?)** indica un elemento opcional de una expresión. Por lo tanto

```
ab?c
```

coincide o con ac o con abc. Nótese que aquí el significado del signo de interrogación difiere de su significado en la shell.

8.8. Especificación de expresiones repetidas.

Las repeticiones de clases se indican con los operadores asterisco **(*)** y el signo más **(+)**. Por ejemplo

a*

coincide con cualquier número de caracteres consecutivos, incluyendo cero; mientras que a+ coincide con una o más apariciones de a. Por ejemplo,

[a-z]+

coincide con todos los literales de letras minúsculas, y

[A-Za-z] [A-Za-z0-9]*

coincide con todos los literales alfanuméricos con un carácter alfabético al principio; ésta es una expresión típica para reconocer identificadores en lenguajes informáticos.

8.9. Especificación de alternación y de agrupamiento.

El operador barra vertical **(|)** indica alternación. Por ejemplo

(ab|cd)

coincide con ab o con cd. Nótese que los paréntesis se usan para agrupar, aunque éstos no son necesarios en el nivel exterior. Por ejemplo

ab | cd

hubiese sido suficiente en el ejemplo anterior. Los paréntesis se deberán usar para expresiones más complejas, tales como

(ab | cd+)?(ef)*

la cual coincide con tales literales como abefef, efefef, cdef, cddd, pero no abc, abcd, o abcdef.

8.10. Especificación de sensitividad de contexto

El lex reconoce una pequeña cantidad del contexto que le rodea. Los dos operadores más simples para éstos son el signo de intercalación **(^)** y el signo de dólar **($)**. Si el primer carácter de una expresión es un signo ^, entonces la expresión sólo coincide al principio de la línea (después de un carácter newline, o al principio del input). Esto nunca se puede confundir con el otro significado del signo ^, complementación de las clases de caracteres, puesto que la complementación sólo se aplica dentro de corchetes. Si el primer carácter es el signo de dólar, la expresión sólo coincide al final de una línea (cuando va seguido inmediatamente de un carácter newline). Este último operador es un caso especial del operador barra **(/)** , el cual indica contexto al final.

La expresión

ab/cd

coincide con el literal ab, pero sólo si va seguido de cd. Por lo tanto

ab$

es lo mismo que

ab/\n

El contexto de la izquierda se maneja en el lex especificando las condiciones start según se explica en la sección "Especificación de sensibilidad de contexto izquierdo ". Si una orden sólo se va a ejecutar cuando el interprete del autómata del lex está en la condición x start, la orden se deberá incluir entre corchetes de ángulos:

<x>

Si consideramos que estamos al comienzo de una línea que es el comienzo de la condición ONE, entonces el operador (^) será equivalente a

<ONE>

Las condiciones start se explican con detalles más tarde.

8.11. Especificación de repetición de expresiones.

Las llaves **({ y })** especifican o bien repeticiones (si éstas incluyen números) o definición de expansión (si incluyen un nombre). Por ejemplo

{dígito}

busca un literal predefinido llamado dígito y lo inserta en la expresión, en ese punto.

8.12. Especificar definiciones.

Las definiciones se dan en la primera parte del input del lex, antes de las órdenes. En contraste,

a{1,5}

busca de una a cinco apariciones del carácter "a".

Finalmente, un signo de tanto por ciento inicial (%) es especial puesto que es el separador para los segmentos fuente del lex.

8.13. Especificación de acciones.

Cuando una expresión coincide con un modelo de texto en el input el lex ejecuta la acción correspondiente. Esta sección describe algunas características del lex, las cuales ayudan a escribir acciones. Nótese que hay una acción por defecto, la cual consiste en copiar el input en el output. Esto se lleva a cabo en todos los literales que de otro modo no coincidirían. Por lo tanto el usuario del lex que desee absorber el input completo, sin producir ningún output, debe proporcionar órdenes para hacer que coincida todo. Cuando se está usando el lex con el yacc, ésta es la situación normal. Se puede tener en cuenta qué acciones son las que se hacen en vez de copiar el input en el output; por lo tanto, en general, una orden que simplemente copia se puede omitir.

Una de las cosas más simples que se pueden hacer es ignorar el input. Especificar una sentencia nula de C; como una acción produce este resultado. La orden frecuente es

```
[ \ t \ n]   ;
```

la cual hace que se ignoren tres caracteres de espaciado (espacio en blanco, tabulador, y newline).

Otra forma fácil de evitar el escribir acciones es usar el carácter de repetición de acción, | , el cual indica que la acción de esta orden es la acción para la orden siguiente. El ejemplo previo también se podía haber escrito:

```
" "      |
"\ t"    |
"\ n"    ;
```

con el mismo resultado, aunque en un estilo diferente. Las comillas alrededor de \ n y \ t no son necesarias.

En acciones más complejas, a menudo se quiere conocer el texto actual que coincida con algunas expresiones como:

```
[ a-z ] +
```

El lex deja este texto en una matriz de caracteres externos llamada **yytext**. Por lo tanto, para imprimir el nombre localizado, una orden como

```
[ a-z ] +        printf ("%s" , yytext);
```

imprime el literal de yytext. La función C printf acepta un argumento de formato y datos para imprimir; en este caso , el formato es print literal donde el signo de tanto por ciento (%) indica conversión de datos, y la s indica el tipo de literal, y los datos son los caracteres de yytext. Por lo tanto esto simplemente coloca el literal que ha coincidido en el output. Esta acción es tan común que se puede escribir como ECHO. Por ejemplo

```
[ a-z ] +        ECHO;
```

es lo mismo que el ejemplo anterior. Puesto que la acción por defecto es simplemente imprimir los caracteres que se han encontrado, uno se puede preguntar ¿Porqué especificar una orden, como ésta, la cual simplemente especifica la acción por defecto? Tales órdenes se requieren a menudo para evitar la coincidencia con algunas otras órdenes que no se desean. Por ejemplo, si hay una orden que coincide con "read", ésta normalmente coincidirá con las apariciones de "read" contenidas en "bread" o en "readjust"; para evitar esto, una orden de la forma

```
[ a-z ] +
```

es necesaria. Esto se explica más ampliamente a continuación.

A veces es más conveniente conocer el final de lo que se ha encontrado; aquí el lex también proporciona un total del número de caracteres que coinciden en la variable **yyleng**. Para contar el número de palabras y el número de caracteres en las palabras del input, será necesario escribir

```
[ a-zA-Z ] +    {words++ ; chars += yyleng;}
```

lo cual acumula en las variables chars el número de caracteres que hay en las palabras reconocidas. Al último carácter del literal que ha coincidido se puede acceder por medio de

```
yytext[ yyleng - 1]
```

Ocasionalmente, una acción lex puede decidir que una orden no ha reconocido el alcance correcto de los caracteres. Para ayudar a resolver esta situación hay dos rutinas. La primera, **yymore( )** se puede llamar para indicar que la siguiente expresión de input reconocida se va a añadir al final de este input. Normalmente, el siguiente literal de input escribirá encima de la entrada actual en yytext. Segundo, **yyless(n)** se puede llamar para indicar que no todos los caracteres que coinciden con la expresión actual se quieren en ese momento. El argumento n indica el número de caracteres de yytext que se van a retener. Otros caracteres que han coincidido previamente se devuelven al input. Esto proporciona el mismo tipo de anticipación ofrecido por el operador barra (/), pero en una forma diferente.

Por ejemplo, considere un lenguaje que define un literal como un conjunto de caracteres entre comillas (‘), y proporciona ése para incluir una marca comilla en un literal, tiene que ir precedido de un signo barra invertida (\). La expresión que coincide con esto es bastante confusa, de forma que puede ser preferible escribir

```
\ ” [ ^” ]*        {
                    if (yytext [yyleng - 1] == ‘\\’)
                          yymore();
                    else
                          .... proceso del usuario normal
                   }
```

lo cual, cuando se compara con un literal tal como

“abc\”def”

coincidirá primero con los cinco caracteres

“abc\

y después la llamada a yymore() hará que se añada al final la siguiente parte del literal,

“def

Téngase en cuenta que el signo de comillas final que se encuentra al final del literal, se cogerá en el código marcado como proceso normal.

La función yyless( ) se deberá usar para reprocesar texto en diversas circunstancias. Considere el problema en la sintaxis C antigua de distinguir la ambigüedad de =-a. Suponga que es deseable tratar esto como =-a e imprimir un mensaje. Una orden podría ser

```
=-[ a-zA-Z ]    {
                  printf (“Operator (=-) ambiguous\n”);
                  yyless (yyleng –1 );
                  ... acción para =- ...
                }
```

lo cual imprime un mensaje, devuelve las letras después del operador al input, y trata el operador como =- .

Alternativamente podría ser deseable tratar esto como =-a. Para hacer esto, simplemente se devuelve el signo menos así como la letra al input. Lo siguiente lleva a cavo la interpretación:

```
=- [ a-zA-Z ]   {
                  printf (“Operator (=-) ambiguious\n”);
                  yyless (yyleng – 2);
                  ... acción para =...
                }
```

Téngase en cuenta que la expresión para los dos casos se podría escribir más fácilmente con

=- / [A-Za-z]

en el primer caso, y

= / - [A-Za-z]

en el segundo: en la orden de acción no se requerirá ningún backup. No es necesario reconocer el identificador completo para observar la ambigüedad. La posibilidad de =-3, de todos modos hace de

=-[^\ t\ n]

una orden aún mejor.

Además de estas rutinas, el lex también permite acceso a las rutinas de I/O que usa. Estas incluyen:

1. **input ( )** el cual devuelve el siguiente carácter de input;

2. **output ( )** el cual escribe el carácter c en el output; y

3. **unput (c)** el cual mete el carácter c de nuevo dentro del input que se va a leer después por input ( ).

Por defecto, estas rutinas se proporcionan como definiciones macro, pero el usuario puede contrarrestarlas y proporcionar versiones privadas. Estas rutinas definen la relación entre los ficheros externos y los caracteres internos, y todas se deben retener o modificar consistentemente. Estas se pueden redefinir, para hacer que el input o el output sea transmitido a o de lugares extraños, incluyendo otros programas o memoria internas pero el conjunto de caracteres que se usa debe ser consistente en todas las rutinas; un valor de cero devuelto por input tiene que significar final de fichero, y la relación entre unput o input debe ser retenido o no funcionará el “loockahead”. Lex no mira hacia delante si no es necesario, pero cada orden que contiene una barra (/) o que termina en uno de los siguientes caracteres implica esto:

+ * ? $

El “loockahead” es también necesario para que coincida una expresión que es un prefijo de otra expresión. Ver a continuación una explicación del conjunto de caracteres que usa el lex. La librería estándar del lex impone un límite de 100 caracteres.

Otra rutina de la librería del lex que a veces es necesaria redefinir es **yywrap ( )** la cual se llama siempre que lex llega a final de fichero. Si yywrap devuelve un 1, lex continúa con un wrapup normal al final del input. A veces, es conveniente hacer que llegue más input de una nueva fuente. En este caso, el usuario deberá proporcionar un yywrap que obtenga nuevo input que devuelva 0. Esto indica al lex que continúe el proceso; yywrap por defecto siempre devuelve 1.

Esta rutina es también un lugar conveniente para imprimir tabla resúmenes, etc. al final del programa. Nótese que si no es posible escribir una orden normal que reconozca el final del fichero; el único acceso a esta condición es a través de yywrap ( ). De hecho, a no ser que se proporcione una versión privada de input ( ) no se puede manejar un fichero que contenga datos nulos, puesto que el valor 0 que input ( ) devuelve se toma como el final de fichero.

8.14. Manejo de órdenes fuentes ambiguas

El lex puede manejar especificaciones ambiguas. Cuando más de una expresión puede coincidir con el input en curso, el lex selecciona de la forma siguiente:

- Se prefiere la coincidencia más larga.
- De entre las órdenes que coinciden en el mismo número de caracteres, se prefiere la primera orden especificada.

Por ejemplo: suponga que se especifican las órdenes siguientes:

```
integer keyword action ... ;
[ a-z ] + identifier action ... ;
```

Si el input es integers, se toma como un identificador, puesto que

```
[ a-z ] +
```

coincide con ocho caracteres mientras que

```
integer
```

sólo coincide con siete. Si el input es integer, ambas órdenes coinciden en siete caracteres, y se selecciona la orden keyword porque ésta ha sido especificada primero. Cualquier cosa más corta (por ejemplo, int) no coincide con la expresión integer, por lo tanto se usa la interpretación identifier.

El principio de la preferencia de la coincidencia más larga hace que ciertas construcciones sean peligrosas, tales como la siguiente:

```
.*
```

Por ejemplo

‘ .* ’

parece ser una buena forma de reconocer un literal que se encuentra entre comillas. Pero es una invitación a que el programa lea más allá, buscando un solo signo de comilla distante. Presentado el input

‘ first ’ quoted string here, ‘ second ‘ here

la expresión anterior coincide con

‘ first’ quoted string here, ‘ second’

lo que probablemente no es lo que se quería. Una orden mejor es de la forma

‘[^’\ n’] *’

lo cual, en el input anterior, se para después de ‘first’. Las consecuencias de errores como éste están mitigadas por el hecho de que el operador punto (**.**) no coincide con un newline. Por lo tanto, tales expresiones nunca coinciden con más de una línea. No intente evitar esto usando expresiones como

[.\ n] +

o sus equivalentes: el programa generado por el lex intentará leer el fichero de input completo, haciendo que el buffer interno se desborde.

Téngase en cuenta que normalmente el lex divide el input, y no busca todas las posibles coincidencias de cada expresión. Esto quiere decir que cada carácter sólo se cuenta una vez, y sólo una. Por ejemplo, suponga quw se desea contar las apariciones de “she” y “he” en un texto de input. Algunas órdenes lex para hacer esto podrían ser

```
she     s++;
he      h++;
\ n     |
.       ;
```

donde las últimas dos órdenes ignoran todo lo que no sea he y she. Recuerde que el punto (**.**) no incluye el carácter newline. Puesto que she incluye a he, el lex normalmente no reconocerá las apariciones de he incluidas en she, puesto que una vez que ha pasado un she estos caracteres se han ido para siempre.

A veces el usuario quiere cancelar esta selección. La acción **REJECT** quiere decir, “ ve y ejecuta la siguiente alternativa”. Esto hace que se ejecute cualquiera que fuese la segunda orden después de la orden en curso. La posición del puntero de input se ajusta adecuadamente. Suponga que el usuario quiere realmente contar las apariciones incluidas en “she”:

```
she     { s++; REJECT;}
he      { h++; REJECT;}
\ n     |
.       ;
```

Estas órdenes son una forma de cambiar el ejemplo anterior para hacer justamente eso. Después de contar cada expresión, ésta se desecha; siempre que sea apropiado, la otra expresión se contará. En este ejemplo, naturalmente, el usuario podría tener en cuenta que she incluye a he, pero no viceversa, y omitir la acción REJECT en he; en otros casos, no sería posible decir qué caracteres de input estaban en ambas clases.

Considere las dos órdenes

```
a [ bc ] +      { ... ; REJECT;}
a [ cd ] +      { ... ; REJECT;}
```

Si el input es ab, sólo coincide la primera orden, y en ad sólo coincide la segunda. La cadena de caracteres del input accb, cuatro caracteres coinciden con la primera orden, y después la segunda orden con tres caracteres. En contrate con esto, el input accd coincide con la segunda orden en cuatro caracteres y después la primera orden con tres.

En general, REJECT es muy útil cuando el propósito de lex no es dividir el input, sino detectar todos los ejemplares de algunos items del input, y las apariciones de estos items pueden solaparse o incluirse uno dentro de otro. Suponga que se desea una tabla diagrama del input; normalmente los diagramas se solapan, es decir, la palabra “the” se considera que contiene a th y a he. Asumiendo una matriz bidimensional llamada digram que se va a incrementar, el fuente apropiado es

```
%%
[ a-z ] [ a-z ]   {digram[yytext[0]] [yytext[1]] ++;
                    REJECT; }
.                 ;
\ n               ;
```

donde el REJECT es necesario para coger un par de letras que comienzan en cada carácter, en vez de en un carácter si y otro no.

Recuerde que REJECT no vuelve a explorar el input. En vez de esto recuerda los resultados de la exploración anterior. Esto quiere decir que si se encuentra una orden con un contexto, y se ejecuta REJECT, no debería haber usado unput para cambiar los caracteres que vienen del input. Esta es la única restricción de la habilidad de manipular el input que aún no ha sido manipulado.

8.15. Especificación de sensibilidad de contexto izquierdo

A veces es deseable aplicar varios grupos de órdenes léxicas en diferentes ocasiones en el input. Por ejemplo, un compilador preprocesador

puede distinguir sentencias del preprocesador y analizarlas de forma diferente a como hace con las sentencias ordinarias. Esto requiere sensitividad al contexto previo, y hay varias formas de tratar tales problemas. El operador (^), por ejemplo, es un operador de contexto previo, que reconoce inmediatamente contexto que precede por la izquierda del mismo modo que el signo de dólar ($) reconoce el contexto que va inmediatamente a la derecha. El contexto adyacente a la izquierda se podría extender, para producir un dispositivo similar al del contexto adyacente a la derecha, pero no es muy probable que sea de utilidad, puesto que a menudo el contexto de la derecha relevante apareció algún tiempo antes tal como al principio de una línea.
Esta sección describe tres formas de tratar con entornos diferentes:

1. El uso de flags, cuando de un entorno a otro sólo cambian unas pocas órdenes.

2. El uso de condiciones start con órdenes.

3. El uso de diversos analizadores léxicos funcionando juntos.

En cada caso, hay órdenes que reconocen la necesidad de cambiar el entorno en el cual se analiza el texto de input siguiente, y ponen varios parámetros para reflejar el cambio. Esto puede ser un flag probado explícitamente por el código de acción del usuario; tal flag es una forma más sencilla de tratar con el problema, puesto que el lex no está relacionado. Puede que sea más conveniente, hacer que el lex recuerde los flags como condiciones iniciales de las órdenes. Cualquier orden puede estar relacionada con una condición start. Sólo será reconocida cuando el lex esté en esa misma condición. La condición de start en curso se puede cambiar en cualquier momento. Finalmente, si los conjuntos de órdenes de los diferentes entornos son muy diferentes, se puede lograr una mayor claridad escribiendo varios analizadores léxicos distintos, y cambiar de uno a otro según se desee.

Considere el siguiente problema: copie el input en el output, cambiando la palabra “magic” a “primero” en cada línea que comience con la letra a, cambiando “magic” por “segundo” en cada línea que comience con la letra b, y cambiando “magic” por “tercero” en cada línea que comience con la letra c. Todas las demás palabras y todas las otras líneas no varían.

Estas órdenes son tan sencillas que la forma más fácil de hacer el trabajo, es con un flag:

```
        int flag;
%%
^a      {flag = 'a' ; ECHO;}
^b      {flag = 'b' ; ECHO;}
^c      {flag = 'c' ; ECHO;}
magic
        switch (flag)
       {
        case 'a': printf ( "first" ); break;
        case 'b': printf ( "second" ); break;
        case 'c': printf ( "third" ); break;
        default : ECHO; break ;
        }
        }
```

deberá ser adecuado.

Para tratar el mismo problema con las condiciones start, cada condición start debe ser introducida en el lex en la sección de definiciones con una línea que diga

%Start nombre1, nombre2

donde las condiciones se pueden especificar en cualquier orden. La palabra start se puede abreviar a s o S. Las condiciones se pueden referenciar al principio de una orden incluyéndolas entre ángulos. Por ejemplo

nombre1 expresión

es una orden que sólo se reconoce cuando el lex está en la condición start nombre1. Para introducir una condición start

BEGIN nombre1;

la cual cambia la condición start por nobre1, Para volver al estado inicial

BEGIN 0;

Restaura la condición del intérprete del autómata del lex. Una orden puede estar activa en varias condiciones start; por ejemplo:

< nombre1, nombre2, nombre3 >

es un prefijo legal. Cualquier orden que no comience con el operador prefijo <> está siempre activa.

El mismo ejemplo anterior se puede escribir:

```
%START AA BB CC
%%
^a      {ECHO; BEGIN AA;}
^b      {ECHO; BEGIN BB;}
^c      {ECHO; BEGIN CC;}
\n      {ECHO; BEGIN 0;}
<AA>  magic         printf (“primero”);
<BB>  magic         printf (“segundo”);
<CC>  magic         printf (“ tercero ”);
```

donde la lógica es exactamente la misma que en el método anterior de tratar el problema, pero el lex hace el trabajo en vea de hacerlo el código del usuario.

8.16. Especificación de definiciones fuente

Recuerde el formato de la fuente lex

```
{ definiciones }
%%
{ órdenes }
%%
{ rutinas del usuario }
```

Hasta ahora sólo se han descrito las órdenes. Se necesitarán opciones adicionales, para definir variables para usarlas en el programa y para que las use el lex. Estas pueden ir bien en la sección de definiciones o en la sección de órdenes.

1. Cualquier línea que no aparezca como una orden o acción de lex, la cual comienza con un espacio en blanco o un tabulador se copia en el programa generado. Tal input de fuente antes del primer delimitador %% será externo a cualquier función del código; si aparece inmediatamente después del primer %%, aparece en un lugar apropiado para las declaraciones en la función escrita por el lex el cual contiene las acciones. Este material tiene que buscar fragmentos de programas, y deberá ir delante de la primera orden de lex.

   Como efecto secundario de lo anterior, las líneas que comienzan con un espacio en blanco o un tabulador, y las cuales contienen un comentario, se pasan al programa generado. Esto se puede usar para incluir comentarios bien en la fuente lex o bien en el código generado. Los comentarios deberán seguir las convenciones del lenguaje C.

2. Cualquier cosa que vaya incluida entre líneas que sólo contienen % { y % } se copia como en el anterior. Los delimitadores se desechan. Este formato permite introducir texto como sentencias de preprocesador que deben comenzar en la columna 1, o copiar líneas que no parecen programas.

3. Cualquier cosa que haya después del tercer delimitador %% sin tener en cuenta los formatos, se copia después del output del lex.

Las definiciones pensadas para el lex se especifican antes del primer delimitador %%. Cualquier línea de esta sección que no esté incluida entre %{ y % }, y que comience en la columna 1, se asume que define los literales de sustitución del lex. El formato de tales líneas es

nombre interpretación

y hace que el literal especificado como una interpretación sea asociado con el nombre. El nombre y la interpretación tienen que estar separados por al menos un espacio en blanco o un tabulador, y el nombre tiene que comenzar con una letra. La interpretación se puede llamar entonces por la sintaxis nombre de una orden. Usando D para los dígitos, y E para un campo exponencial, por ejemplo, se puede abreviar las órdenes para que reconozcan números:

```
D                       [ 0-9 ]
E                       [ DEde][ - + ]? {D}+
%%
{D}+                    pritnf (“integer”);
{D}+”.”{D}*({E})?       |
{D}*”.”{D}+({E})?       |
{D}+{E}                 printf (“real”);
```

Téngase en cuenta que las dos primeras órdenes para los números reales; requieren un punto decimal y contienen un campo exponencial opcional, pero la primera requiere que al menos un dígito antes del punto decimal, y el segundo requiere al menos un dígito después del punto decimal. Para tratar correctamente el problema creado por una expresión FORTRAN tal como 25.EQ.I, la cual no contiene un número real, una orden sensitiva de contexto tal como

```
[ 0-9 ]+/”.”EQ          printf(“integer”);
```

se podría usar además de la orden normal para números enteros.

La sección de definiciones puede también contener otros comandos, incluyendo una tabla de caracteres, una lista de condiciones start, o ajustes al tamaño por defecto de las matrices dentro del propio lex para programas fuentes más largos. Estas posibilidades se explican en la sección “Formato Fuente”.

8.17. Lex y Yacc

Si se quiere usar el lex con el yacc, téngase en cuenta que el lex escribe un programa llamado yylex( ), el nombre requerido por el yacc para su analizador. Normalmente el programa main por defecto de la librería lex llama a esta rutina, pero si yacc está cargado, y se usa su programa main, yacc llamará al yylex( ). En este caso, cada orden lex deberá terminar con

```
return (token);
```

donde se devuelve el valor de token apropiado. Una forma fácil de obtener acceso a los nombres del yacc para los tokens es compilar el fichero de output del lex como parte del fichero de output del yacc poniendo la línea

```
#include "lex.yy.c"
```

en la última sección del input del yacc. Suponiendo que la gramática se llame "good" y las órdenes léxicas se llamen "better" , la secuencia de comandos XENIX puede ser:

```
yacc good
lex better
cc y.tab.c –ly –ll
```

La librería yacc (-ly) se deberá cargar antes de la librería lex para obtener un programa main que llame al reconocedor yacc. La generación de programas lex y yacc se puede hacer en cualquier orden.

Como problema trivial, considere copiar un fichero input mientras se añade 3 a cada número positivo divisible por 7. Podemos desarrollar un programa fuente lex que hace precisamente eso:

```
%%
        int k;
[ 0-9 ]+        {
                    k = atoi(yytext);
                    if (k%7 == 0)
                        printf("%d",k+3);
                    else
                        printf("%d", k);
                }
```

La orden [0-9]+ reconoce cadenas de dígitos; atoi( ) convierte los dígitos en binario y almacena el resultado en k. El operador (%) se usa para comprobar si k es divisible por 7; si lo es, se incrementa en 3 a medida que se va escribiendo. Se puede objetar que este programa alterará input tal como 49.63 o X7. Además incrementa el valor absoluto de todos los números negativos divisible por 7. Para evitar esto, añada simplemente unas pocas más de órdenes después de la orden activa, como:

```
%%
        int k;
-? [ 0 –9 ]+                {
                              k = atoi(yytest);
                              printf("%d",k%7 == 0? K+3 : k);
                            }
-? [ 0-9 ]+                 ECHO;
[A-Za-z] [A-Za-z0-9 ]+      ECHO;
```

Las cadenas de caracteres numéricos que contienen un punto decimal o precedidas de una letra serán cogidas por una de las dos órdenes, y no se cambiarán. Si el if-else ha sido cambiado por una expresión condicional C para ahorrar espacio; la forma a?b:c quiere decir: si a entonces b, en caso contrario c.

Como ejemplo de obtención de estadísticas, aquí hay un programa que hace histogramas de longitudes de palabras, donde una palabra está definida como una cadena de letras.

```
        int lengs[100];
%%
[ a-z ] +       lengs[yyleng]++;
.               |
\n              ;
%%
yywrap( )
{
int i;
printf("Length No. words\n");
for(i=0; i<100; i++)
        if (lengs[i] > 0)
                printf("%5d%10d\n",i,lengs[i]);
return (1);
}
```

Este programa acumula el histograma, pero no produce output. Al final del input imprime una tabla. La sentencia final return (1); indica que el lex va a llevar a cabo un wrapup. Si yywrap( ) devuelve un valor cero (false) implica que hay disponible más input y el programa va a continuar leyendo y procesando. Un yywrap( ) que nunca devuelve un valor verdadero produce un bucle infinito.

Como un ejemplo más grande, aquí hay algunas partes de un programa escrito para convertir FORTRAN de doble precisión en FORTRAN de precisión simple. Puesto que FORTRAN no distingue entre letras mayúsculas y letras minúsculas, esta rutina comienza definiendo un conjunto de clases que incluyen ambos tipos de letras de cada letra:

```
a               [aA]
b               [bB]
c               [cC]
.                 .
.                 .
.                 .
z               [zZ]
```

una clase adicional reconoce espacio en blanco:

```
W               [ \ t ]*
```

La primera orden cambia precisión doble a real, o DOBLE PRECISION A REAL

```
{d}{o}{u}{b}{l}{e}{W}{p}{r}{e}{c}{i}{s}{i}{o}{n}{
                        printf(yytext [0] == 'd' ? "real" : "REAL"
                        }
```

A lo largo de este programa se tienen cuidado de conservar el tipo de letra del programa original. El operador condicional se usa para seleccionar la forma apropiada de la palabra reservada. La siguiente orden copia indicaciones de la tarjeta de continuación para evitar confundirlos con las constantes:

```
^"    "[ ^ 0]    ECHO;
```

En la expresión regular, las comillas rodean los espacios en blanco. Esto se interpreta como el principio de línea, después cinco espacios en blancos, después nada excepto un espacio en blanco o un cero. Nótese los dos significados diferentes del signo de intercalación (^). A continuación van algunas órdenes para cambiar constantes de doble precisión por constantes de precisión flotante.

```
[ 0-9 ]+ {W}{d}{W}[+-]?{W}[ 0-9 ]+              |
[ 0-9 ]+ {W}"."{W}{d}{W}[+-]?{W}[ 0-9 ]+       |
"."{W}[ 0-9 ]+ {W}{d}{W}[+-]?{W}[ 0-9 ]+       {
                /* convert constants*/
                for (p=yytext; *p ¡= 0; p++)
                        {
                          if (*p == 'd' 11 *p == 'D')
                                *p += 'e' - 'd';
                                ECHO;
                        }
```

Después de que se ha reconocido la constante de punto flotante, el bucle for la explora para localizar la letra "d" o "D". El programa añade " 'e' – 'd' " lo cual lo convierte en la siguiente letra del alfabeto. La constante modificada, ahora de precisión simple, se escribe de nuevo. A continuación va una serie de nombres que tienen que volverse a escribir para suprimir su " d" inicial. Usando la matriz yytext la misma acción es suficiente para todos los nombres (aquí sólo se especifica un ejemplo de una lista relativamente larga).

```
{d}{s}{i}{n}          |
{d}{c}{o}{s}          |
{d}{s}{q}{r}{t}       |
{d}{a}{t}{a}{n}       |
...
{d}{f}{l}{o}{a}{t}    {printf("%s", yytext + 1);}
```

Otra lista de nombres debe cambiar la 'd' inicial por una 'a' inicial

```
{d}{l}{o}{g}          |
{d}{l}{o}{g}10        |
{d}{m}{i}{n}1         |
{d}{m}{a}{x}1         {
                          yytext[0] += 'a' – 'd';
                          ECHO;
                      }
```

Si se cambia esta interpretación proporcionando rutinas I/O que traducen los caracteres, es necesario decírselo al lex, dando una tabla de traducción. Esta tabla tiene que estar en la sección de definiciones, y tienen que estar incluidas por líneas que contenga sólo %T. La tabla contiene líneas de la forma

```
{entero}{cadena de caracteres}
```

```
%T
1           Aa
2           Bb
...
26          Zz
27          |n
28          +
29          –
30          0
31          1
...
39          9
%T
```

Esta tabla convierte las letras minúsculas y mayúsculas en los enteros del 1 al 26, el carácter newline en el número 27, los signos más (+) y menos(+) en los números 28 y 29, y los dígitos en los números 30 a 39. Nótese el escape para el caracteres newline. Si se proporciona una tabla, cada carácter que va a aparecer o bien en las órdenes o en cualquier input válido tiene que estar incluido en la tabla. Ningún carácter puede ser asignado al número 0, y ningún carácter puede ser asignado a un número más grande que el tamaño del conjunto de caracteres del hardware.

8.19. Formato Fuente

El formato general de un fichero fuente del lex es:

```
{definiciones}
%%
{órdenes}
%%
{subrutinas del usuario}
```

La sección de definiciones contiene una combinación de

1. Definiciones en el formato “nombre espacio traducción”.

2. Código incluido, en el formato “ espacio código ”.

3. Código incluido en el formato

```
%{
 código
%}
```

4. Condiciones start, especificadas en el formato

```
%S nombre1, nombre2, ...
```

5. Tablas del conjunto de caracteres, en el formato

```
%T
numero espacio cadena de caracteres
%T
```

6. Cambia los tamaños de las matrices internas, en el formato

```
%x    nnn
```

donde nnn es un número entero decimal que representa un tamaño de matriz y x selecciona el parámetro de la forma siguiente:

Letra	Parámetro
p	posiciones
n	estados
e	nudos de árbol
a	transiciones
k	Clases de caracteres compactados
o	tamaño de la matriz de output

Las líneas en la sección de órdenes tienen la forma:

expresión acción

donde la acción se puede continuar en las líneas siguientes usando llaves para delimitarlo.

Las expresiones regulares del lex usan los operadores siguientes:

x	El carácter " x "
"x"	Una "x", incluso si x es un operador.
\x	Una "x", incluso si x es un operador.
[xy]	El carácter x o y.
[x-z]	Los caracteres x, y o z.
[^ x]	Cualquier carácter salvo x.
.	Cualquier carácter salvo newline.
^ x	Una x al principio de la línea.
<y>x	Una x cuando el lex está en la condición start y.
x$	Una x al final de una línea.
x?	Una x opcional.
x*	0,1,2 .... apariciones de x.
x+	1,2,3 .... apariciones de x.
x|y	Una x o una y.
x/y	Una x, pero sólo si va seguida de una y.
{xx}	La traducción de xx de la sección de definiciones.
x{m,n}	m a través de n ocurrencias de x.