5. lex

An Overview of lex Programming

lex is a software tool that lets you solve a wide class of problems drawn from text processing, code enciphering, compiler writing, and other areas. In text processing, you may check the spelling of words for errors; in code enciphering, you may translate certain patterns of characters into others; and in compiler writing, you may determine what the tokens (smallest meaningful sequences of characters) are in the program to be compiled. The problem common to all these tasks is recognizing different strings of characters that satisfy certain characteristics. In the compiler writing case, creating the ability to solve the problem requires implementing the compiler's lexical analyzer. Hence the name **lex**.

It is not essential to use **lex** to handle problems of this kind. You could write programs in a standard language like C to handle them, too. In fact, what **lex** does is produce such C programs. (**lex** is therefore called a program generator.) What **lex** offers you, once you acquire a facility with it, is typically a faster, easier way to create programs that perform these tasks. Its weakness is that it often produces C programs that are longer than necessary for the task at hand and that execute more slowly than they otherwise might. In many applications this is a minor consideration, and the advantages of using **lex** considerably outweigh it.

To understand what **lex** does, see the diagram in Figure 5-1. We begin with the **lex** source (often called the **lex** specification) that you, the programmer, write to solve the problem at hand. This **lex** source consists of a list of rules specifying sequences of characters (expressions) to be searched for in an input text, and the actions to take when an expression is found. The source is read by the **lex** program generator. The output of the program generator is a C program that, in turn, must be compiled by a host language C compiler to generate the executable object program that does the lexical analysis. Note that this procedure is not typically automatic—user intervention is required. Finally, the lexical analyzer program produced by this process takes as input any source file and produces the desired output, such as altered text or a list of tokens.

You can also use **lex** to collect statistical data on features of the input, such as character count, word length, number of occurrences of a word, and so forth. In later sections of this chapter, we will see:

- how to write **lex** source to do some of these tasks
- how to translate **lex** source
- how to compile, link, and execute the lexical analyzer in C
- how to run the lexical analyzer program

We will then be on our way to appreciating the power that **lex** provides.

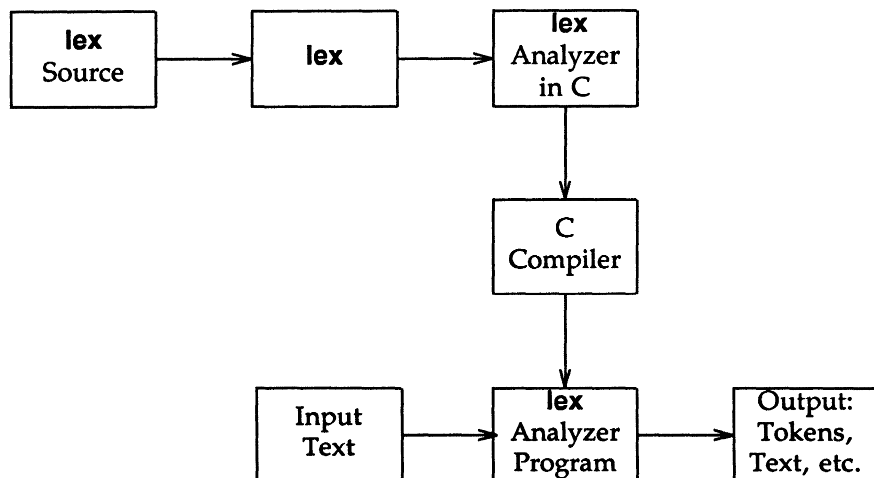

Figure 5-1. Creation and Use of a Lexical Analyzer with **lex**

Writing lex Programs

A **lex** specification consists of at most three sections: definitions, rules, and user subroutines. The rules section is mandatory. Sections for definitions and user subroutines are optional, but if present, must appear in the indicated order.

The Fundamentals of lex Rules

The mandatory rules section opens with the delimiter **%%.** If a subroutines section follows, another **%%** delimiter ends the rules section. If there is no second delimiter, the rules section is presumed to continue to the end of the program.

Each rule consists of a specification of the pattern sought and the action(s) to take on finding it. (Note the dual meaning of the term specification—it may mean either the entire **lex** source itself or, within it, a representation of a particular pattern to be recognized.) Whenever the input consists of patterns not sought, **lex** writes out the input exactly as it finds it. So, the simplest **lex** program is just the beginning rules delimiter, **%%**. It writes out the entire input to the output with no changes at all. Typically, the rules are more elaborate than that.

Specifications

You specify the patterns you are interested in with a notation called regular expressions. A regular expression is formed by stringing together characters with or without operators. The simplest regular expressions are strings of text characters with no operators at all, such as:

apple
orange
pluto

These three regular expressions match any occurrences of those character strings in an input text. If you want to have your lexical analyzer **a.out** remove every occurrence of **orange**, from the input text, you could specify the rule:

```
orange;
```

Because you did not specify an action on the right (before the semicolon), **lex** does nothing but print out the original input text with every occurrence of this regular expression removed, that is, without any occurrence of the string **orange** at all.

Unlike **orange** above, most of the expressions that we want to search for cannot be specified so easily. The expression itself might simply be too long. More commonly, the class of desired expressions is too large; it may, in fact, be infinite.

Thanks to the use of operators, we can form regular expressions signifying any expression of a certain class. The + operator, for instance, means one or more occurrences of the preceding expression, the **?** means 0 or 1 occurrence(s) of the preceding expression (this is equivalent, of course, to saying that the preceding expression is optional), and * means 0 or more occurrences of the preceding expression. (It may seem odd to speak of 0 occurrences of an expression and to

need an operator to capture the idea, but it is often helpful. We will see an example in a moment.) So **m+** is a regular expression matching any string of **m**s such as each of the following:

```
mmm
m
mmmmm
mm
```

and **7*** is a regular expression matching any string of zero or more 7s:

```
77
77777

777
```


The string of blanks on the third line matches simply because it has no 7s in it at all.

Brackets, [ ], signify any one character from the string of characters specified between the brackets. Thus, **[dgka]** matches a single **d**, **g**, **k**, or **a**. Note that commas are not included within the brackets. Any comma here would be taken as a character to be recognized in the input text. Ranges within a standard alphabetic or numeric order are indicated with a hyphen, **–**. The sequence **[a-z]**, for instance, represents any lowercase letter. Somewhat more interestingly, the expression:

```
[A-Za-z0-9*&#]
```

is a regular expression that matches any letter (whether upper- or lowercase), any digit, an asterisk, an ampersand, or a sharp character. Given the input text:

```
$$$$?? ????!!!*$$ $$$$$$&+====r~~# ((
```

the lexical analyzer with the previous specification in one of its rules will recognize the *, &, r, and #, perform on each recognition whatever action the rule specifies (we have not indicated an action here), and print out the rest of the text as it stands.

The operators become especially powerful in combination. For example, the regular expression to recognize an identifier in many programming languages is:

```
[a-zA-Z][0-9a-zA-Z]*
```

An identifier in these languages is defined to be a letter followed by zero or more letters or digits, and that is just what the regular expression says. The first pair of brackets matches any letter. The second, if it were not followed by a *, would match any digit or letter. The two pairs of brackets with their enclosed characters would then match any letter followed by a digit or a letter. But with the asterisk,

*, the example matches any letter followed by any number of letters or digits. In particular, it would recognize the following as identifiers:

e
pay
distance
pH
EngineNo99
R2D2

Note that it would not recognize the following as identifiers:

not_idenTIFER
5times
$hello

because **not_idenTIFER** has an embedded underscore; **5times** starts with a digit, not a letter; and **$hello** starts with a special character. Of course, you may want to write the specifications for these three examples as an exercise.

A potential problem with operator characters is how we can refer to them as characters to look for in our search pattern. The last example, for instance, will not recognize text with an * in it. The **lex** program solves the problem in one of two ways: a character enclosed in quotation marks or a character preceded by a \ is taken literally, that is, as part of the text to be searched for. To use the backslash method to recognize, say, an * followed by any number of digits, we can use the pattern:

```
\*[1-9]*
```

To recognize a \ itself, we need two backslashes: \\.

Actions

Once **lex** recognizes a string matching the regular expression at the start of a rule, it looks to the right of the rule for the action to be performed. Kinds of actions include recording the token type found and its value, if any; replacing one token with another; and counting the number of instances of a token or token type. What you want to do is write these actions as program fragments in the host language C. An action may consist of as many statements as are needed for the job at hand. You may want to print out a message noting that the text has been found or a message transforming the text in some way. Thus, to recognize the expression Amelia Earhart and to note such recognition, the rule:

```
"Amelia Earhart"   printf("found Amelia");
```

would do. And to replace in a text lengthy medical terms with their equivalent

acronyms, a rule such as:

```
Electroencephalogram    printf("EEG");
```

would be called for. To count the lines in a text, we need to recognize end-of-lines and increment a linecounter. **lex** uses the standard escape sequences from C like **\n** for end-of-line. To count lines, we might have:

```
\n    lineno++;
```

where **lineno**, like other C variables, is declared in the definitions section that we discuss later.

5

lex stores every character string that it recognizes in a character array called **yytext[]**. You can print or manipulate the contents of this array as you want. Sometimes your action may consist of two or more C statements and you must (or for style and clarity, you choose to) write it on several lines. To inform **lex** that the action is for one rule only, simply enclose the C code in braces. For example, to count the total number of all digit strings in an input text, print the running total of the number of digit strings (not their sum, here) and print out each one as soon as it is found, your **lex** code might be:

```
+?[1-9]+              { digstrngcount++;
                        printf("%d",digstrngcount);
                        printf("%s", yytext);    }
```

This specification matches digit strings whether they are preceded by a plus sign or not, because the **?** indicates that the preceding plus sign is optional. In addition, it will catch negative digit strings because that portion following the minus sign, **–**, will match the specification. The next section explains how to distinguish negative from positive integers.

Advanced lex Usage

lex provides a suite of features that lets you process input text riddled with complicated patterns. These include rules that decide what specification is relevant, when more than one seems so at first; functions that transform one matching pattern into another; and the use of definitions and subroutines. Before considering these features, you may want to affirm your understanding thus far by examining an example drawing together several of the points already covered.

```
%%
-[0-9]+         printf("negative integer");
+?[0-9]+        printf("positive integer");
-0.[0-9]+       printf("negative fraction, no whole number part");
rail[ ]+road    printf("railroad is one word");
crook           printf("Here's a crook");
function        subprogcount++;
G[a-zA-Z]*     { printf("may have a G word here: ", yytext);
                  Gstringcount++; }
```

5

The first three rules recognize negative integers, positive integers, and negative fractions between 0 and –1. The use of the terminating + in each specification ensures that one or more digits compose the number in question. Each of the next three rules recognizes a specific pattern. The specification for **railroad** matches cases where one or more blanks intervene between the two syllables of the word. In the cases of **railroad** and **crook**, you may have simply printed a synonym rather than the messages stated. The rule recognizing a **function** simply increments a counter. The last rule illustrates several points:

- The braces specify an action sequence extending over several lines.
- Its action uses the **lex** array **yytext**[], which stores the recognized character string.
- Its specification uses the * to indicate that zero or more letters may follow the **G**.

Some Special Features

Besides storing the recognized character string in **yytext**[], **lex** automatically counts the number of characters in a match and stores it in the variable **yyleng**. You may use this variable to refer to any specific character just placed in the array **yytext**[]. Remember that C numbers locations in an array starting with 0, so to print out the third digit (if there is one) of an integer just recognized, you might write:

```
[1-9]+          {if  (yyleng > 2)
                  printf("%c", yytext[2]); }
```

lex follows a number of high-level rules to resolve ambiguities that may arise from the set of rules that you write. *Prima facie,* any reserved word, for instance, could match two rules. In the lexical analyzer example developed later in the section on **lex** and **yacc,** the reserved word **end** could match the second rule as well as the seventh, the one for identifiers.

NOTE

lex follows the rule that where there is a match with two or more rules in a specification, the first rule is the one whose action will be executed.

By placing the rule for **end** and the other reserved words before the rule for identifiers, we ensure that our reserved words will be duly recognized.

Another potential problem arises from cases where one pattern you are searching for is the prefix of another. For instance, the last two rules in the lexical analyzer example above are designed to recognize > and >= . If the text has the string >= at one point, you might worry that the lexical analyzer would stop as soon as it recognized the > character to execute the rule for > rather than read the next character and execute the rule for >=.

NOTE

lex follows the rule that it matches the longest character string possible and executes the rule for that.

Here it would recognize the >= and act accordingly. As a further example, the rule would enable you to distinguish + from ++ in a program in C.

Still another potential problem exists when the analyzer must read characters beyond the string you are seeking because you cannot be sure you've found it until you've read the additional characters. These cases reveal the importance of trailing context. The classic example here is the DO statement in FORTRAN. In the statement:

```
DO 50 k = 1 , 20, 1
```

we cannot be sure that the first 1 is the initial value of the index **k** until we read the first comma. Until then, we might have the assignment statement:

```
DO50k = 1
```

(Remember that FORTRAN ignores all blanks.) The way to handle this is to use the forward slash, **/** (not the backslash, \), which signifies that what follows is trailing context, something not to be stored in **yytext**[], because it is not part of the token itself. So the rule to recognize the FORTRAN DO statement could be:

```
30/[ ]*[0-9][ ]*[a-z A-Z0-9]+=[a-z A-Z0-9]+,    printf("found DO");
```

Different versions of FORTRAN have limits on the size of identifiers, here the index name. To simplify the example, the rule accepts an index name of any length.

lex uses the **$** as an operator to mark a special trailing context—the end of line. (It is therefore equivalent to **\n**.) An example would be a rule to ignore all blanks and tabs at the end of a line:

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

On the other hand, if you want to match a pattern only when it starts a line, **lex** offers you the circumflex, ^, as the operator. The formatter **nroff**, for example, demands that you never start a line with a blank, so you might want to check input to **nroff** with some such rule as:

```
^[ ]      printf("error: remove leading blank");
```

Finally, some of your action statements themselves may require your reading another character, putting one back to be read again a moment later, or writing a character on an output device. **lex** supplies three functions to handle these tasks—**input()**, **unput(c)**, and **output(c)**, respectively. One way to ignore all characters between two special characters, say between a pair of double quotation marks, would be to use **input()**, thus:

```
\"        while (input() != '"');
```

On finding the first double quotation mark, the generated **a.out** will simply continue reading all subsequent characters so long as none is a quotation mark, and not again look for a match until it finds a second double quotation mark.

To handle special I/O needs such as writing to several files, you may use standard I/O routines in C to rewrite the functions **input()**, **unput(c)**, and **output**. These and other programmer-defined functions should be placed in your subroutine section. Your new routines will then replace the standard ones. The standard **input()** is equivalent to **getchar()**, and the standard **output(c)** is equivalent to **putchar(c)**.

There are a number of **lex** routines that let you handle sequences of characters to be processed in more than one way. These include **yymore()**, **yyless**(n), and **REJECT**. Recall that the text matching a given specification is stored in the array **yytext**[]. In general, once the action is performed for the specification, the characters in **yytext**[] are overwritten with succeeding characters in the input stream to form the next match. The function **yymore()**, by contrast, ensures that the succeeding characters recognized are appended to those already in **yytext**[]. This lets you do one thing and then another, when one string of characters is

significant and a longer one including the first is significant as well. Consider a character string bound by **B**s and interspersed with one at an arbitrary location:

```
B...B...B
```

In a simple code-deciphering situation, you may want to count the number of characters between the first and second **B**'s and add it to the number of characters between the second and third **B**. (Only the last **B** is not to be counted.) The code to do this is:

```
B[^B]*          { if (flag = 0)
                      save = yyleng;
                      flag = 1;
                      yymore();
                 else     {
                      importantno = save + yyleng;
                      flag = 0; }
                   }
```

5

where **flag**, **save**, and **importantno** are declared (and at least **flag** initialized to 0) in the definitions section. The **flag** distinguishes the character sequence terminating just before the second **B** from that terminating just before the third.

The function **yyless**(*n*) lets you reset the end point of the string to be considered to the *n*th character in the original **yytext**[]. Suppose you are again in the code deciphering business and the gimmick here is to work with only half the characters in a sequence ending with a certain one, say upper- or lowercase **Z**. The code you want might be:

```
[a-yA-Y]+[Zz]        {  yyless(yyleng/2);
                     ... process first half of string... }
```

Finally, the function REJECT lets you more easily process strings of characters even when they overlap or contain one another as parts. REJECT does this by immediately jumping to the next rule and its specification without changing the contents of **yytext[]**. If you want to count the number of occurrences both of the regular expression **snapdragon** and of its subexpression **dragon** in an input text, the following will do:

```
snapdragon      {countflowers++; REJECT;}
dragon          countmonsters++;
```

As an example of one pattern overlapping another, the following counts the number of occurrences of the expressions **comedian** and **diana**, even where the input text has sequences such as **comediana..**:

```
comedian        {comiccount++; REJECT;}
diana           princesscount++;
```

Note that the actions here may be considerably more complicated than simply incrementing a counter. Always, the counters and other necessary variables are declared in the definitions section beginning the **lex** specification.

Definitions

The **lex** definitions section may contain any of several classes of items. The most critical are external definitions, **#include** statements, and abbreviations. Recall that for legal **lex** source this section is optional, but usually some of these items are necessary. External definitions have the form and function that they do in C. They declare that variables globally defined elsewhere (perhaps in another source file) will be accessed in your **lex**-generated **a.out**. Consider a declaration from an example to be developed later.

```
extern int tokval;
```

When you store an integer value in a variable declared in this way, it will be accessible in the routine, say a parser, that calls it. If, on the other hand, you want to define a local variable for use within the action sequence of one rule (as you might for the index variable for a loop), you can declare the variable at the start of the action itself right after the left brace, { .

The purpose of the **#include** statement is the same as in C: to include files of importance for your program. Some variable declarations and **lex** definitions might be needed in more than one **lex** source file. It is then advantageous to place them all in one file to be included in every file that needs them. One example occurs in using **lex** with **yacc**, which generates parsers that call a lexical analyzer. In this context, you should include the file **y.tab.h**, which may contain **#define**s for token names. Like the declarations, **#include** statements should come between %{ and }%, thus:

```
%{
#include "y.tab.h"
extern int tokval;
int lineno;
%}
```

In the definitions section, after the %} that ends your **#include**'s and declarations, you place your abbreviations for regular expressions to be used in the rules section. The abbreviation appears on the left of the line and, separated by one or more spaces, its definition or translation appears on the right. When you later use abbreviations in your rules, be sure to enclose them within braces.

5

NOTE

The purpose of abbreviations is to avoid needless repetition in writing your specifications and to provide clarity in reading them.

As an example, reconsider the **lex** source reviewed at the beginning of this section on advanced **lex** usage. The use of definitions simplifies our later reference to digits, letters, and blanks. This is especially true if the specifications appear several times:


```
D               [0-9]
L               [a-zA-Z]
B               [ ]
%%
-{D}+            printf("negative integer");
+?{D}+           printf("positive integer");
-0.{D}+          printf("negative fraction");
G{L}*            printf("may have a G word here");
rail{B}+road      printf("railroad is one word");
crook            printf("criminal");
 \"\./{B}+       printf(".\"");
   .                .
   .                .
```

The rule ensures that a period always precedes a quotation mark at the end of a sentence. It would change example". to example."

Subroutines

You may want to use subroutines in **lex** for much the same reason that you do so in other programming languages. Action code that is to be used for several rules can be written once and called when needed. As with definitions, this can simplify the writing and reading of programs. The function **put_in_tabl()**, to be discussed in the next section on **lex** and **yacc**, is a good candidate for a subroutine.

Another reason to place a routine in this section is to highlight some code of interest or to simplify the rules section, even if the code is to be used for one rule only. As an example, consider the following routine to ignore comments in a language like C where comments occur between the /* and the */ symbols.

```
"/*"                    skipcmnts();
.
.               /* rest of rules */
%%
skipcmnts()
{
        for(;;)
        {
           while (input() != '*');
           if (input() != '/') {
                unput(yytext[yyleng-1]);
           else return;
        }
 }
```


There are three points of interest in this example. First, the **unput(c)** function (putting back the last character read) is necessary to avoid missing the final / if the comment ends unusually with a **/ . In this case, eventually having read an *, the analyzer finds that the next character is not the terminal / and must read some more. Second, the expression **yytext**[**yyleng–1**] picks out that last character read. Third, this routine assumes that the comments are not nested. (This is indeed the case with the C language.) If, unlike C, they are nested in the source text, after **input**()ing the first */ ending the inner group of comments, the **a.out** will read the rest of the comments as if they were part of the input to be searched for patterns.

Other examples of subroutines would be programmer-defined versions of the I/O routines **input**(), **unput(c)**, and **output**(), discussed above. Subroutines such as these that may be exploited by many different programs would probably do best to be stored in their own individual file or library to be called as needed. The appropriate **#include** statements would then be necessary in the definitions section.

Using lex with yacc

If you work on a compiler project or develop a program to check the validity of an input language, you may want to use the operating system program tool **yacc**. **yacc** generates parsers, programs that analyze input to ensure that it is syntactically correct. (**yacc** is discussed in detail in Chapter 6 of this guide.) **lex** often forms a fruitful union with **yacc** in the compiler development context. Whether or not you plan to use **lex** with **yacc**, be sure to read this section because it covers information of interest to all **lex** programmers.

The lexical analyzer that **lex** generates (not the file that stores it) takes the name **yylex**(). This name is convenient because **yacc** calls its lexical analyzer by this

same name. To use **lex** to create the lexical analyzer for the parser of a compiler, you want to end each **lex** action with the statement **return** *token*, where *token* is a defined term whose value is an integer. The integer value of the token returned indicates to the parser what the lexical analyzer has found. The parser, whose file is called **y.tab.c** by **yacc**, then resumes control and makes another call to the lexical analyzer when it needs another token.

In a compiler, the different values of the token indicate what, if any, reserved word of the language has been found or whether an identifier, constant, arithmetic operand, or relational operator has been found. In the latter cases, the analyzer must also specify the exact value of the token: what the identifier is, whether the constant, say, is 9 or 888, whether the operand is + or * (multiply), and whether the relational operator is = or >. Consider the following portion of **lex** source for a lexical analyzer for some programming language perhaps slightly reminiscent of Ada:


```
begin                          return(BEGIN);
end                                return(END);
while                              return(WHILE);
if                                 return(IF);
package                            return(PACKAGE);
reverse                            return(REVERSE);
loop                               return(LOOP);
[a-zA-Z][a-zA-Z0-9]*             { tokval = put_in_tabl();
                                     return(IDENTIFIER); }
[0-9]+                           { tokval = put_in_tabl();
                                     return(INTEGER); }
\+                              { tokval = PLUS;
                                     return(ARITHOP); }
\-                             { tokval = MINUS;
                                     return(ARITHOP); }
>                                { tokval = GREATER;
                                      return(RELOP); }
>=                               { tokval = GREATEREQL;
                                     return(RELOP); }
```

Despite appearances, the tokens returned and the values assigned to **tokval** are indeed integers. Good programming style dictates that we use informative terms such as **BEGIN**, **END**, **WHILE**, and so forth to signify the integers the parser understands, rather than use the integers themselves. You establish the association by using **#define** statements in your parser calling routine in C.

For example:

```
#define BEGIN  1
#define END  2
   .
#define PLUS 7
   .
```

If the need to change the integer for some token type arises, you then change the **#define** statement in the parser rather than hunt through the entire program, changing every occurrence of the particular integer. In using **yacc** to generate your parser, it is helpful to insert the statement:

```
#include y.tab.h
```


into the definitions section of your **lex** source. The file **y.tab.h** provides **#define** statements that associate token names such as **BEGIN**, **END**, and so on with the integers of significance to the generated parser.

To indicate the reserved words in the example, the returned integer values suffice. For the other token types, the integer value of the token type is stored in the programmer-defined variable **tokval**. This variable, whose definition was an example in the definitions section, is globally defined so that the parser as well as the lexical analyzer can access it. **yacc** provides the variable **yylval** for the same purpose.

Note that the example shows two ways to assign a value to **tokval**. First, a function **put_in_tabl()** places the name and type of the identifier or constant in a symbol table so that the compiler can refer to it in this or a later stage of the compilation process. More to the present point, **put_in_tabl()** assigns a type value to **tokval** so that the parser can use the information immediately to determine the syntactic correctness of the input text. The function **put_in_tabl()** would be a routine that the compiler writer might place in the subroutines section discussed later. Second, in the last few actions of the example, **tokval** is assigned a specific integer indicating which operand or relational operator the analyzer recognized. If the variable PLUS, for instance, is associated with the integer 7 by means of the **#define** statement above, then when a + sign is recognized, the action assigns to **tokval** the value 7, which indicates the +. The analyzer indicates the general class of operator by the value it returns to the parser (in the example, the integer signified by ARITHOP or RELOP).

Running lex under the Operating System

As you review the following few steps, you might recall Figure 5-1 at the start of the chapter. To produce the lexical analyzer in C, run:

```
lex lex.l
```

where **lex.l** is the file containing your **lex** specification. The name **lex.l** is conventionally the favorite, but you may use whatever name you want. The output file that **lex** produces is automatically called **lex.yy.c**; this is the lexical analyzer program that you created with `lex`. You then compile and link this as you would any C program, making sure that you invoke the **lex** library with the **–ll** option:


```
cc lex.yy.c –ll
```

The **lex** library provides a default **main()** program that calls the lexical analyzer under the name **yylex()**, so you need not supply your own **main()**.

If you have the **lex** specification spread across several files, you can run **lex** with each of them individually, but be sure to rename or move each **lex.yy.c** file (with **mv**) before you run **lex** on the next one. Otherwise, each will overwrite the previous one. Once you have all the generated **.c** files, you can compile all of them, of course, in one command line.

With the executable **a.out** produced, you are ready to analyze any desired input text. Suppose that the text is stored under the filename **textin** (this name is also arbitrary). The lexical analyzer **a.out** by default takes input from your terminal. To have it take the file **textin** as input, simply use redirection:

```
a.out < textin
```

By default, output will appear on your terminal, but you can redirect this as well:

```
a.out < textin > textout
```

In running **lex** with **yacc**, either may be run first. The segment:

```
yacc –d grammar.y
lex lex.l
```

spawns a parser in the file **y.tab.c**. (The **–d** option creates the file **y.tab.h**, which contains the **#define** statements that associate the **yacc** assigned integer token values with the user-defined token names.) To compile and link the output files produced, run:

```
cc lex.yy.c y.tab.c –ly –ll
```

The **yacc** library is loaded (with the **–ly** option) before the **lex** library (with the **–ll** option) to ensure that the **main()** program supplied will call the **yacc** parser.

Several options are available with the **lex** command. If you use one or more of them, place them between the command name **lex** and the filename argument. If you care to see the C program, **lex.yy.c**, that **lex** generates on your terminal (the default output device), use the **–t** option:

lex –t lex.l

The **–v** option prints out for you a small set of statistics describing the so-called finite automata that **lex** produces with the C program **lex.yy.c**. (For a detailed account of finite automata and their importance for **lex**, see the Aho, Sethi, and Ullman text, *Compilers: Principles, Techniques, and Tools,* Addison-Wesley, 1986.)

lex uses a table (a two-dimensional array in C) to represent its finite automaton. The maximum number of states that the finite automaton requires is set by default to 500. If your **lex** source has many rules or the rules are very complex, this default value may be too small. You can enlarge the value by placing another entry in the definitions section of your **lex** source, as follows:

```
%n 700
```

This entry tells **lex** to make the table large enough to handle as many as 700 states. (The **–v** option will indicate how large a number you should choose.) If you have need to increase the maximum number of state transitions beyond 2000, the designated parameter is **a**, thus:

```
%a 2800
```

Finally, check the *Programmer's Reference Manual* page on **lex** for a list of all the options available with the **lex** command. In addition, review the paper by Lesk (the originator of **lex**) and Schmidt, "Lex—A Lexical Analyzer Generator," in volume 5 of the *UNIX Programmer's Manual,* Holt, Rinehart, and Winston, 1986. It is somewhat dated, but offers several interesting examples.

This tutorial has introduced you to **lex** programming. As with any programming language, the way to master it is to write programs.

5

5