Introduction to lex, the Lexical Analyzer

Many computer applications involve reading text strings. This is especially true for man-machine communication.

For some forms of textual input, a programmer can design a program by hand to process it. However, it is much easier to implement such programs when you use a software tool that will automatically construct a program to process the data. The COHERENT command **lex** is such a tool.

lex accepts expressions that describe the text input, and generates a program to process it. In computer-ese, **lex** is a "lexical scanner program generator".

This document tells you how to use **lex**. It presents many simple examples to illustrate how to use its features and how to use the generated program with other tools provided with COHERENT, notably the parser generator **yacc**.

Readers of this document are presumed to be familiar with the C programming language and the use of the COHERENT system. Related documents include *Using the COHERENT System* and the tutorial to yacc, the COHERENT parser generator.

How To Use lex

lex generates lexical scanners for compilers, to do statistical analysis of text, and to generate filters for many diverse tasks. This section gives examples of how to use **lex**. Later sections discuss the concepts used in these examples in detail.

Translating Strings

The first example tells **lex** to match an input string and replace it with a different string — in this case, replace the misspelled word "removeable" with the correctly spelled "removable". The program outputs unchanged all strings that it does not recognize. Enter the following program into the file **rmv.lex**.

```
%%
removeable     printf ("removable");
```

This creates the **lex** specification. Use the following command line to pass this specification through **lex**:

```
lex rmv.lex
```

This produces a C program named **lex.yy.c**, which you can compile by typing:

```
cc lex.yy.c -ll -o rmv
```

The executable program **rmv** is now ready to use. To illustrate its use, type:

```
rmv
Is this file removeable?
<ctrl-D>
```

rmv replies:

```
Is this file removable?
```

Note that the generated program reads from standard input and writes to standard output.

Remove Blanks From Input

The next example deletes all blanks and tabs from the input. Type the following **lex** program into file **nosp.lex**:

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

Generate and compile the program with the following commands:

```
lex nosp.lex
cc lex.yy.c -ll -o nosp
```

To invoke the program, type **nosp**. Now, test it by typing the following:

```
This may be hard to read after processing.
<ctrl-D>
```

nosp outputs:

```
Thismaybehardtoreadafterprocessing.
```

Trimming Blanks

The previous example can be rewritten to remove strings of blanks or tabs and replace them with one space. Type the following into file **onesp.lex**:

```
%%
[ \t]+  printf (" ");
```

Generate and compile this with the following commands:

```
lex onesp.lex
cc lex.yy.c -ll -o onesp
```

Invoke your program with the command **onesp**. Now, type the following text to test the program; be sure to separate each word by two spaces:

```
This  should  be  easier  to  read.
<ctrl-D>
```

onesp prints the following:

```
This should be easier to read.
```

lex Specification Form

This section discusses the form of the **lex** specification.

Simple Form

The examples shown above use the simplest form of a **lex** program. Consider the text of the example **rmv.lex**:

```
%%
removeable     printf ("removable");
```

The symbol

```
%%
```

divides sections of the **lex** specification. Not all specifications need to be present, but at least one **%%** must appear in a **lex** program.

This symbol separates **lex** *definitions* from *rules*. With nothing before the **%%**, there are no definitions. Rules follow the **%%**. No definitions are needed in the simplest of **lex** specifications.

Rules in lex

The format of a **lex** rule is simple. Every rule has two parts. Refer to the program **rmv**:

```
removeable     printf ("removable");
```

The first part begins at the beginning of the line and ends with a space or tab. In the example rule, the first part is

```
removeable
```

This part is called the *pattern*.

The second part follows the space or tab, and is called the *action*. The action in this example is:

```
printf ("removable");
```

When the pattern specified by the rule is found in the input, the corresponding action is performed. Thus, this rule detects every appearance of *removeable* and outputs the correct spelling.

A **lex** program tries each rule's pattern in turn, and performs the associated action if and only if the pattern matches. Actions often modify the input that matched the pattern; they may also do nothing for certain patterns. To illustrate this, type the following specification into file **erase.lex**:

```
%%
erase   ;
```

Then compile the generated program with the following commands:

```
lex erase.lex
cc lex.yy.c -ll -o erase
```

This program copies all its input to its output, except for any appearance of the string **erase**. Invoke the program by typing **erase**, and then test it by typing:

```
Have you erased the blackboard?
<ctrl-D>
```

erase then prints:

```
Have you d the blackboard?
```

If the input contains patterns that do not match any of the patterns in the suite of rules you typed into **lex**, they are simply output unchanged. Usually, you will want to write a rule to cover every case.

Statements in lex

As noted earlier, **lex** is a program generator. It reads the specifications that you prepare for it, and writes a C program that is used with the **lex** library. Many of the actions in the rules you specify, such as

```
printf ("removable");
```

are themselves C statements. These statements are included in the resulting program, along with other statements that **lex** provides so the program can run.

You can include other statements, should the program need them, by placing them in appropriate places. The following program, called **count.lex**, shows how this is done. It counts the number of *tokens*, or strings of non-blank characters. Type the following into the file **count.lex** exactly as printed:

```
        int count;
%%
[^ \t\n]+      count++;
[ \t\n]+       ;
%%
yywrap ()
{
        printf ("Number of tokens:%d\n", count);
        return (1);
}
```

Statements other than rule actions appear in two places in the program. The first such statement is in the definition section, which precedes the rule section delimiter **%%**:

```
        int count;
```

This C statement declares the variable **count** to be an integer variable. Notice that it is preceded by a tab; a tab or a space indicates to **lex** that an input line is not a rule.

The second kind of non-rule statement follows the second **%%**, which marks the end of the rules section. **lex** regards anything that follows the second delimiter as being source statements.

The above example includes a function named **yywrap**. **lex** programs always call this function at the end of processing. The above program fills this function with code that prints the number of tokens in the text. If you do not include a routine named **yywrap**, **lex** will use a standard one from its library.

Compile the program by typing the following commands:

```
lex count.lex
cc lex.yy.c -ll -o count
```

Run the program by typing:

```
count <count.lex
```

This counts the tokens in the **count.lex** file itself. **count** will print the following:

```
Number of tokens:21
```

Groups of Statements

In previous examples, the C statement in the action part of the rule is a single statement. In many **lex** applications, however, you will need to use more than one statement per rule.

To do so, enclose the statements in the braces **{ }**. The following example illustrates grouping. This **lex** specification generates a program to add numbers found in the input and print the total whenever it reads an asterisk '*'. Type the following program into **nsum.lex**:

```
         int number, sum;
%%
[0-9]+  {
        sscanf (yytext, "%d", &number);
        sum += number;
        printf ("%s", yytext);
        }
"*"     {
        printf ("%s", yytext);
        printf ("%d", sum);
        sum = 0;
        }
```

Compile the program by typing:

```
lex nsum.lex
cc lex.yy.c -ll -o nsum
```

To run the generated **nsum** program, enter a sample data file by typing:

```
cat >numbers
one two three
1 2 3 4 * 1 2 3 5 *
*
done
<ctrl-D>
```

Run the program by typing:

```
nsum <numbers
```

nsum will print:

```
one two three
1 2 3 4 *10 1 2 3 5 *11
*0
done
```

The statements that follow the definitions

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

and

```
*
```

are enclosed in braces, because each action triggers several statements. Consider the first of these:

```
[0-9]+  {
        sscanf (yytext, "%d", &number);
        sum += number;
        printf ("%s", yytext);
        }
```

The pattern looks for strings of digits. **sscanf** converts each such string into a number and saves it in the variable **number**. Now, consider the second rule:

```
"*"     {
        printf ("%s", yytext);
        printf ("%d", sum);
        sum = 0;
        }
```

This specifies that upon detection of * in the input, the program is to print the sum of the numbers and then reset the counter to zero. In both of these rules, the statement

```
printf ("%s", yytext);
```

prints the number or * so that the output shows the input as well as the total. **lex** defines the variable **yytext**. It always contains the string that matches the pattern.

If the input is neither a number nor an asterisk, no pattern specifically matches it. Therefore, the program echoes it unchanged to the standard output.

Using the Same Action

To make it easier for you to write actions, **lex** allows you to *abbreviate* rules; that is, you have to write only once any action that is performed upon detection of several patterns. To abbreviate rules represented symbolically by

```
p1      action1
p2      action1
```

use the vertical bar operator:

```
p1      |
p2      action1
```

The vertical bar means "use the action from the first rule that declares an action." An example is given in the section on macro abbreviations, below.

Patterns

The first part of each rule in the **lex** rules section is a pattern that may match parts of the input. This section describes how to construct these patterns, sometimes called *regular expressions*. If you are familiar with **ed** and how its patterns work, this will be familiar to you.

Simple Patterns

The simplest kind of pattern is a string of characters that matches itself. A previous section presented an illustration of this:

```
%%
removeable     printf ("removable");
```

The regular expression "removeable" matches all occurrences of *removeable* that appear in the input text.

Certain characters have special meaning to **lex** patterns. To match a special character literally, you must *quote* it. For example, * has special meaning. To match the asterisk literally (that is to match any '*'s that appear in the input), surround it with quotation marks:

```
"*"
```

Another way to quote characters is to precede it with the backslash character '****'.

```
\*
```

The following characters each have special meaning and must be quoted to be matched as text characters:

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

However, within **"**, the **** still has its meaning, so to match the string ***** use the regular expression:

```
"\\*"
```

Also, to match a quote character, use:

```
\"
```

Classes of Characters

The power of patterns comes from special characters that match more than one character. The following examines each special character in detail.

The *period* or *dot* matches any character except newline. The following regular expression matches any pair of characters that begins with **J**:

```
J.
```

The following example prints in square brackets any sequence of five characters that precedes a blank. Type the following into the file **five.lex**:

```
%%
....." "       printf ("[%s]", yytext);
```

Compile the program with the following commands:

```
lex five.lex
cc lex.yy.c -ll -o five
```

Now, type the following to create a test file for **five**:

```
cat > work
how well does this work?
no match
<ctrl-D>
```

Now, test **five** by typing:

```
five < work
```

The result is:

```
how[ well ]does[ this ]work?
no match
```

The second line of the input does not have any matches. Because the **dot** pattern character does not match the end-of-line character, all five characters that precede the blank must be on the same line.

Another way to match many characters, but selectively, is with the *character class* operation. Enclose in square brackets the set of characters to be matched. Any of the characters listed there will match one character of the input. For example,

```
[0123456789]
```

matches any decimal digit in the input. Characters may be in any order within the brackets. Thus

```
[0246813579]
```

is equivalent to the example above.

To simplify specifying for character classes, you can specify ranges of characters. The beginning and end of the range is separated by a hyphen. To match all decimal digits as above, use:

```
[0-9]
```

To match all alphabetic characters, type:

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

The special character **^**, when used after the opening bracket ‘**[**’, tells **lex** to match any character *except* those enclosed. The following example finds all strings that consist of two digits followed by a third character that is neither an alphabetic character nor a period, and prints them enclosed by **{** and **}**. Type the following into file **twodig.lex**:

```
%%
[0-9][0-9][^\.a-zA-Z]  printf ("{%s}", yytext);
```

Process and compile the program by typing the following commands:

```
lex twodig.lex
cc lex.yy.c -ll -o twodig
```

Invoke the program by typing **twodig**, and test it by entering the following text:

```
12. 12 12a 1 12 b
<ctrl-D>
```

twodig prints the following in reply:

```
12. {12 }12a 1 {12 }b
```

Repetition

In the patterns shown so far, each character matches only one character at a time. However, many interesting input patterns involve repetition of characters. The above program **twodig.lex** used such repetition, albeit in a primitive way.

To match one or more instances of a character, follow it with the pattern operator **+**. Consider the summation example in **nsum.lex**, shown earlier, which recognized strings of input numbers and added them to a total:

```
[0-9]+  {
        sscanf (yytext, "%d", &number);
        sum += number;
        printf ("%s", yytext);
        }
```

The pattern

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

matches a string of one or more digits.

The operator * will match *zero* or more characters of a specified type. The following example deletes all characters between square brackets. Type it into file **star.lex**:

```
%%
\[.*\]  printf ("[]");
```

Type the following commands to generate and compile the program:

```
lex star.lex
cc lex.yy.c -ll -o star
```

Type the following to create a test file:

```
cat > disappear
[This should disappear]
[what happens with two] of them [on a line?]
<ctrl-D>
```

Now, use the test file with **star**:

```
star < disappear
```

The output is:

```
[]
[]
```

In looking at the example's input, you might have expected the output to be:

```
[]
[] of them []
```

lex does not produce the latter output because it generates recognizers that find the longest match if several matches are possible. Therefore, **star** matched the first **[**, then all characters up to and including the second **]**. When you write a pattern that matches many characters, you should bear this possibility in mind.

To change the program to match the first **]**, rewrite it as follows:

```
%%
\[[^\]]*\]     printf ("[]");
```

The regular expression now matches a string of all characters except a **]**, when that string is enclosed in square brackets.

The '**?**' character signals that the previous character or regular expression is optional. In other words, '**?**' signals zero or one instance of a character or regular expression.

To see how this would be used in a program, consider a text processor that regards a word as being a strings of alphabetic characters that may or may not be followed by a period. The following example does this, and encloses the recognized words in parentheses. Enter it into file **word.lex**:

```
%%
[a-zA-Z]+\.?   printf ("(%s)", yytext);
```

Generate and compile the program with the following commands:

```
lex word.lex
cc lex.yy.c -ll -o word
```

Create a test file:

```
cat > words
These are words.
Question mark not included?
<ctrl-D>
```

And test **word** with the following command:

```
word < words
```

The result is

```
(These) (are) (words.)
(Question) (mark) (not) (included)?
```

The question mark, like the ***** and **+** operators, can also follow another specification of a pattern. If you wanted to end a sentence with a character other than a period, the following code will do the job for you:

```
[a-zA-Z]+[.?!,]?
```

The characters

```
.?!,
```

are optional.

The '+' and '*' operators may match many characters. If you wish to match a specific number of characters or patterns, follow the patterns with the repetition within braces **{** and **}**. For example

```
[0-9]{3}
```

matches a string of exactly three characters. With this information, you should be able to rewrite the pattern part of **twodig.lex**, described above.

You can also specify a range of counts. To match from seven to nine occurrences of lower-case alphabetic characters, use:

```
[a-z]{7,9}
```

Choices and Grouping

To indicate alternate choices of characters or regular expressions, separate them in the regular expression with a vertical bar operator **|**. For example, if you wish to match either three decimal digits or the character **a**, use:

```
[0-9]{3}|a
```

Parentheses help to group the parts of the pattern that are separated by the vertical bar:

```
(abc)|(def)
```

This pattern will match either the string **abc** or the string **def**.

Matching Non-Graphic Characters

Non-special, graphic characters in patterns match themselves. Most non-graphic characters, such as space, tab, and control characters, cannot be matched directly. **lex** provides special sequences to match control characters. The following example removes tabs and blanks from the beginning and end of input lines. Type it into file **deblank.lex**:

```
%%
[ \t]+\n       printf ("\n");
\n[ \t]+       printf ("\n");
```

Generate and compile the program with the following commands:

```
lex deblank.lex
cc lex.yy.c -ll -o deblank
```

Type the following to create a test file:

```
cat > sportab
begins with no space or tab
       begins with tab
   begins with three spaces
<ctrl-D>
```

Type the following to test **deblank**:

```
deblank < sportab
```

The result is:

```
begins with no space or tab
begins with tab
begins with three spaces
```

The special regular expression **\t** represents *tab*, and **\n** represents *newline*.

To match the backspace character, use **\b**. Form feed is matched by **\f**. To match an arbitrary character with a known octal value, use three octal digits after the backslash; for example,

```
\007
```

More Patterns

This section discusses more advanced capabilities of patterns.

Line Context

Like **ed**, **lex** patterns can include characters that represent the beginning and end of line. To match a line that consists of exactly five alphabetic characters, type:

```
^[a-zA-Z]{5}$
```

The character **^** matches the beginning of the line, and **$** matches the end of the line.

Context Matching

A slash (virgule) '**/**' shows that a following context is necessary to match a string. For example, the following program matches the string **match** only if it is immediately followed by the string **ing**. Type it into file **match.lex**:

```
%%
match/ing      printf ("{%s}", yytext);
```

To compile the program, type the following commands:

```
lex match.lex
cc lex.yy.c -ll -o match
```

Type the following to create a test file:

```
cat > matchtit
Will this match?
This is a matching test.
<ctrl-D>
```

And run it against **match** by typing:

```
match < matchtit
```

The result is:

```
Will this match?
This is a {match}ing test.
```

Notice that the string before the slash is matched. The program does not match the part that follows the slash, even though the string must be there for the first part to be matched. Thus, the regular expression that follows the slash may also be matched on its own. To see how this works, type the following into the file **match2.lex**:

```
%%
match/ing      printf ("{%s}", yytext);
ing            printf ("ed");
```

To compile the program, type the following commands:

```
lex match2.lex
cc lex.yy.c -ll -o match2
```

Once again, create a test file:

```
cat > matching
Will this match?
This is a matching test.
You must now sing for your supper.
<ctrl-D>
```

And run it:

```
match2 < matching
```

The result is:

```
Will this match?
This is a {match}ed test.
You must now sed for your supper.
```

The context-string that follows the **/** may be a regular expression. The following example matches the whole-number portion of a decimal fraction. Type it into the file **wholept.lex**:

```
%%
"-"?[0-9]+/"."[0-9]+   printf ("(%s)", yytext);
```

To compile the program, type the following commands:

```
lex wholept.lex
cc lex.yy.c -ll -o wholept
```

Invoke the program by typing **wholept**; then type the following to test it:

```
        123 12345 1234.567
1       <ctrl-D>
```

The result will be:

```
123 12345 (1234).567
```

As you can see, the part of the regular expression

```
"-"?
```

matches an optional leading minus sign. Then

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

matches a string of at least one decimal digit. Then, the following context must match the regular expression

```
"."[0-9]+
```

which matches the fractional part of the number. When it finds a number that matches, it prints the number's whole part enclosed in parentheses.

Macro Abbreviations

lex also provides a macro facility that can substantially simplify the writing of complex regular expressions.

A *macro* is a named body of text. A macro processor simply replaces the name of the macro with the text of the macro.

To illustrate, type following example into file **float.lex**. It recognizes integer and floating point constants according to the C format:

```
d [0-9]+
e [Ee][+-]?[0-9]+
%%
{d}\.          |
{d}\.{d}       |
\.{d}          |
{d}\.{e}       |
\.{d}{e}       |
{d}\.{d}{e}    |
{d}{e}         printf ("F:[%s]", yytext);
```

lex replaces the macro name **e** with the code that matches a string of digits at least one digit long. It replaces the macro name **d** with code that matches the number's exponent. These two are invoked in the manner of

```
{d}
```

within a pattern. To compile the program, type the following commands:

```
lex float.lex
cc lex.yy.c -ll -o float
```

Type the following to create a test file:

```
cat > flonumb
1 1. 1.2 1.e4 1e4
.1e4 e4 .1 . 0 1.2e3
<ctrl-D>
```

And test it by typing:

```
float < flonumb
```

The result is:

```
1 F:[1.] F:[1.2] F:[1.e4] F:[1e4]
F:[.1e4] e4 F:[.1] . 0 F:[1.2e3]
```

Context: Start Rules

Many tasks in lexical processing require the program to be aware of a token's context. **lex** lets you make processing conditional upon previously processed input. This is done by using **start conditions**.

Start conditions are named in the definitions section as follows:

```
%S name1 name2
```

where **name1** and **name2** are names of start conditions. These start conditions are then used by prefixing a pattern with the start condition's name enclosed in angle brackets. For example:

```
<name1>
```

For example, you can use one start condition to control the scanning of comments in a Pascal-like language. The start condition is set by the **lex** statement **BEGIN** when the beginning bracket of the comment is found. The comment is scanned for strings that begin with **$** to signal compiler operation. To see how this works, type the following into the file **comment.lex**:

```
%S CMNT
%%
<CMNT>\$[ler]     printf ("Option is %s.\n", yytext);
<CMNT>[^\}]       ;
<CMNT>\}          BEGIN 0;
\{                BEGIN CMNT;
```

To compile, use the following commands:

```
lex comment.lex
cc lex.yy.c -ll -o comment
```

Once again, create a test file:

```
cat > option
{This is a comment}
{This comment has options $l $e $r}
program
information
<ctrl-D>
```

And run it by typing:

```
comment < option
```

The result is:

```
Option is $l.
Option is $e.
Option is $r.

program
information
```

The context start condition is named following **BEGIN** in the action part of the rule. To return to the normal condition, use **0** as the context name.

Separate Contexts

If you wish to perform context-dependent processing that is more complex than that shown in the example above, you will find it convenient to use separate contexts.

The names of the contexts are defined in the definitions sections, after the definitions of any start conditions: For example:

```
%C name name ...
```

The **lex** function **yyswitch** switches to a new context.

The body of the context's rules is preceded in the rules section by:

```
%C name
```

To see how this works, type the following into file **pre.lex**. It is part of a program that recognizes the preprocessor statements in a C program:

```
%C PRE
%%
^#        yyswitch (PRE);
[^#\n]+         printf ("[%s]", yytext);
%C PRE
include.+       |
define.+        {
                printf("{%s}", yytext);
                yyswitch(0);
                }
.+              {
                printf ("{??%s}", yytext);
                yyswitch (0);
                }
```

A **#** in column 1 signals the beginning of a preprocessor statement. Upon recognizing this condition, this program uses **yyswitch** to activate the context **PRE**.

Within this separate context, individual rules recognize different preprocessor statements; this example includes only two. Each of the rules prints the preprocessor line enclosed in braces **{ }**. In addition, the rules switch back to the original (and unnamed) context by the statement

```
yyswitch (0);
```

To compile and test this program, use the following commands:

```
lex pre.lex
cc lex.yy.c -ll -o pre
pre <lex.yy.c | more
```

This example uses the function **yyswitch** to return to the original context at the end of each rule in the secondary context. Some applications require a return to the context that was previously in force. To assist in this, **yyswitch** returns the value of the previous context.

To modify the example to switch to the previous context, add a statement to the definitions section to declare a variable to hold the previous context:

```
int prev;
```

Then, when switching, save the current context:

```
prev = yyswitch (NEW);
```

To switch back, use:

```
yyswitch (prev);
```

To summarize, you can specify a match at the beginning and end of input lines. You may need a following context for a match. Macros provide a means of abbreviating elements of patterns. **lex** can qualify some patterns based on a start context, or process entirely separate contexts.

More About Writing Actions

This section discusses predefined **lex** actions and how to use them. It also presents other **lex** routines that are useful in writing actions.

ECHO

Many **lex** actions simply output the matched pattern:

```
[0-9]+  printf ("%s", yytext);
```

This form has been used in the examples because many examples also output additional material, such as enclosing braces, to illustrate the matched token.

lex provides a simpler way to echo the exact token matched:

```
[0-9]+          ECHO;
```

The following example echoes all strings of digits twice, and everything else once. Type it into file **double.lex**:

```
%%
[0-9]+          {ECHO; ECHO;}
[^0-9]+         ECHO;
```

To compile the program, use the commands:

```
lex double.lex
cc lex.yy.c -ll -o double
```

To invoke the program, type **double**; and to test it, type the following text:

```
abcdef 123 678 as45 67gh
<ctrl-D>
```

double will reply:

```
abcdef 123123 678678 as4545 6767gh
```

Processing Overlapping Strings

The **lex** processing illustrated to this point has been restricted to programs whose rules recognize distinct strings. That is, once any character of a string is matched by a regular expression, it cannot be matched by another.

Some applications require that strings be matched by more than one rule; such multiply-matched strings are called *overlapping strings*. The **lex** action word **REJECT** provides this capability. When **REJECT** appears in a rule, other rules can also match the string. Remember, however, that **lex** programs give precedence to the longest string that matches a regular expression.

The following example determines the number of letter pairs, or *digrams*, in its input. The input is presumed to be lower-case letters. Enter the following into **digram.lex**:

```
        int digram [128] [128];
%%
[a-z][a-z]     {
               digram [yytext [0]] [yytext [1]]++;
               REJECT;
               }
.              ;
\n             ;
%%
yywrap ()
{
    int i1, i2;
    for (i1 = 'a'; i1 <= 'z'; i1++)
        for (i2 = 'a'; i2 <= 'z'; i2++)
            if (digram [i1] [i2] != 0)
               printf ("%d\t%c%c\n",
                   digram [i1] [i2], i1, i2);
}
```

To compile the program, type the commands:

```
lex digram.lex
cc lex.yy.c -ll -o digram
```

To invoke the program, type **digram**; and test it with the following text:

```
this is a test of digrams.
<ctrl-D>
```

The result will be:

```
1       am
1       di
1       es
1       gr
1       hi
1       ig
2       is
1       ms
1       of
1       ra
1       st
1       te
1       th
```

yylex

lex places the actions you provide for the rules in your **lex** program into a C routine named **yylex**.

If you add variable declarations in the definitions section before the first **%%**, **yylex** can access them, as in the example **digram.lex**, shown above. You can also declare variables that are local to **yylex**, if you place the declarations after the rules section delimiter and before the first rule. A tab or space must precede the declaration, where the **%** symbols are at the beginning of the line. See the example **yacclex.lex**, below.

The following program is a different version of **digram.lex**, called **digram2.lex**; it uses such a declaration.

```
        int digram [128] [128];
%%
        int t0, t1;
[a-z][a-z]      {
                t0 = yytext [0];
                t1 = yytext [1];
                digram [t0] [t1]++;
                REJECT;
                }
%%
yywrap ()
{
    int i1, i2;
    for (i1 = 'a'; i1 <= 'z'; i1++)
        for (i2 = 'a'; i2 <= 'z'; i2++)
            if (digram [i1] [i2] != 0)
                printf ("%d\t%c%c\n",
                    digram [i1] [i2], i1, i2);
}
```

Header Section

You can insert additional code at the beginning of the generated program by including such code in the definitions section. An earlier example, **count.lex**, demonstrated how to do this:

```
        int count;
%%
[^ \t\n]+       count++;
[ \t\n]+        ;
%%
yywrap ()
{
    printf ("Number of tokens:%d \n ", count);
    return (1);
}
```

A tab or space character must precede the code you include.

If you wish to insert **include** or any other C preprocessor statement at the beginning of the program, however, a different technique must be used. This stems from the fact that the preprocessor statements must begin at the beginning of the line, and the blank or tab precludes this.

The alternative method to add code to the beginning is as follows:

```
%{
... code ...
%}
```

where the **%** symbols are at the beginning of the line.

Additional Routines

If your version of **yywrap** or any of the rules that you write need other routines, you can include code for them after a second **%%**. (This was where **yywrap** was shown in **digram.lex**.) If you wish to provide your own version of **input** or **output**, you must define it there.

Using lex With yacc

Although **lex** can handle many applications by itself, it is often used with the parser-generator **yacc**. For example, programming-language compilers often have parts generated by both **lex** and **yacc**.

Like **lex**, **yacc** is a program generator. Its programs can recognize input that is structured according to a grammar fed to the **yacc** program generator. In most instances, **yacc**-generated programs require *tokens* as input, instead of individual characters. In the context of a programming language, a token is a variable name or a special character (such as an operator). **lex** is often used with **yacc** because **lex** is especially well suited for partitioning text input into tokens.

A **yacc**-generated program expects a token number as input from the routine **yylex**. **yacc** assigns a unique number, or constant definition, to each unique type of token, and expects **yylex** to return these numbers as input.

For your **lex** program to access these predefined constant definitions for token types, you must include the generated **lex** source in the **yacc** specification.

The following examples process very simple input, to illustrate how to assemble **lex**- and **yacc**-generated programs. To begin, type the following into the file **yacclex.yy**:

```
%token beginning midtok ending
%start simplistic
%%
simplistic :   beginning middle ending
                       {printf ("recognized"); };
middle  :      midtok;
middle  :      middle midtok;
%%
```

When **yacc** processes this program, it produces the file **y.tab.h** that contains the token-name definitions. The following **lex** source reads **y.tab.h** to learn of the constant definitions that **yacc** generated; type it into file **yacclex.lex**:

```
%{
#include "y.tab.h"
%}
%%
"("     return (beginning);
")"     return (ending);
[a-zA-Z]       return (midtok);
```

The symbolic definition of the token names are **beginning**, **ending** and **midtok**.

To compile the programs, type the following commands:

```
yacc yacclex.yy
lex yacclex.lex
cc y.tab.c lex.yy.c -ly -ll -o yacclex
```

Type **yacclex** to invoke the new program; and test by typing the following:

```
(abcdef)
<ctrl-D>
```

The result will be:

```
recognized
```

Summary

lex is a utility that generates lexical analyzers according to a set of specifications that you write. *Lexical analysis* means to read a mass of text, recognize strings within that mass, and react appropriately when each type of string is discovered. With **lex**, you can write programs to perform complex analysis of text simply by describing what analysis you want to perform, without worrying about the messy details of how that analysis is actually performed; thus, **lex** is a fine example of what is nowadays called a “fourth-generation language”.

lex is especially well suited to work with the parser-generator **yacc**. By using them together, you can efficiently build command processors and even entire computer languages.