lex Lexical Generator Tutorial

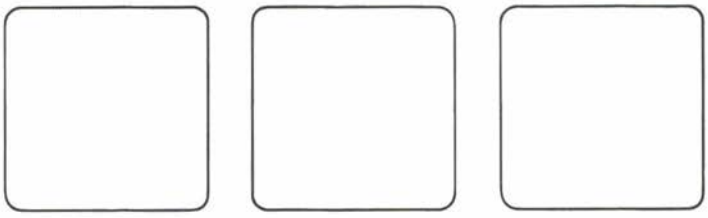

Table of Contents

1. Introduction

Many computer applications involve reading text strings. While machine readable information is often used to communicate between programs, man-machine interfaces usually involve strings of alphanumeric information.

For some forms of textual input, a programmer can easily design a program that will process such input. However, such programs can be implemented much more quickly by using a software tool that will automatically construct a program to process the data.

The COHERENT command **lex** is such a tool. **lex** is a tool that accepts expressions describing the input to the program, and produces a program that will process the input. **lex** is a lexical scanner program generator.

This document tells you how to use **lex**. Many simple examples are presented to illustrate how to use the various features of **lex**, and how to use the generated program in conjunction 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 *Introduction to the COHERENT System* and *yacc Parser Generator Tutorial*.

2. How to use lex

lex is used to generate lexical scanners for compilers, to do statistical analysis of text, and to generate COHERENT 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 shows how a **lex** program can match an input string and output a different string. Strings not recognized by the program will be output unchanged. Enter the following program into the file **rmv.lex**.

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

This creates the **lex** specification. Pass this specification through **lex**, and it will produce a program named **lex.yy.c**:

```
lex rmv.lex
```

This produces a C program, 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>
```

And **rmv** will reply

```
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 will delete all blanks and tabs in the input. Put the following **lex** program into **nosp.lex**:

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

Generate and compile the program:

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

Here is how to use the program:

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

The reply will be

```
Thismaybehardtoreadafterprocessing.
```

Trimming blanks

The previous example can be rewritten to remove strings of blanks or tabs and replace them with one space. Create **onesp.lex** containing:

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

Generate and compile this with

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

Now test the program by typing

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

Note that the words in this input are separated by two spaces. The output produced by **onesp** is

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

Summary

This section shows you the basics of using **lex**. All the steps necessary to write, generate, compile, and test simple **lex** programs are shown.

3. lex specification form

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

Simple form

The examples shown in the previous section use the simplest form of a **lex** program. The text of the example **rmv.lex** was

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

The symbol

```
%%
```

is used to divide sections of the **lex** specification. Not all specifications need to be present, but at least one %% will appear in a correct **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. There are two parts to every rule. Again referring to the **rmv** example:

```
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.

The generated program tries each rule's pattern in turn, and performs the associated action if and only if the pattern matches. Actions often output some modification of the input that matched the pattern. Actions may also do nothing for certain patterns. To illustrate this, create the **lex** specification in **erase.lex**

```
%%
erase    ;
```

Then compile the generated program with

```
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**. Type the commands

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

and the program will reply

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

If there are any patterns in the file that do not match any of the patterns in the rules, that pattern is simply output unchanged. Usually, you will want to write rules to cover all cases.

Statements in lex

As noted in the previous sections, **lex** is a program generator. It reads the specifications that you prepare for it, and produces a C program that is used in conjunction with the **lex** library. Many of the actions in the rules you specify, for example

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

are themselves C statements. These statements are included in the resulting program along with other statements provided by **lex** necessary for the program to operate.

Other statements can be included if the program needs them by placing them in appropriate places. The following example counts the number of *tokens*, or strings of non-blank characters. Enter the following program into the file **count.lex**

```
        int count;
%%
[^ \t\n]+       count++;
[ \t\n]+        ;
%%
yywrap ()
{
        printf ("Number of tokens:%d0, count);
        return (1);
}
```

There are two places that statements other than rule actions are placed. The first non-rule statement is in the definition section, which precedes the rule section delimiter %%:

```
        int count;
```

This C statement declares **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 %% delimiter marking the end of the rules. Anything following the second delimiter will be treated as source statements and placed at the end of the program.

This example has included a routine named **yywrap**. This is a routine that **lex** programs call at then end of processing. This is the routine that prints the count of the tokens in the input.

Compile the program by typing

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

Run the program by typing

```
count <count.lex
```

which will count the tokens in the **count.lex** file itself. The result will be:

```
Number of tokens:21
```

If you do not include a routine named **yywrap**, **lex** will use a standard one.

Groups of statements

In previous examples, the C statement as the action part of the rule is a single statement. In many **lex** applications, 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 will generate a program to total numbers found in the input and print the total whenever the asterisk character '*' is input. Enter 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;
        }
```

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>
```

This builds a sample data file. Run the program by typing

```
nsum <numbers
```

The reply will be

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

The statements following the

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

and

```
*
```

patterns are enclosed in braces, since each action triggers several statements. The first of these,

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

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

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

specifies that upon detection of * in the input, the total sum of the numbers is to be printed and the total is reset. 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.

The variable **yytext** is defined by **lex**, and aiways contains the string matched by the rule.

If the input is neither a number or an asterisk, it is echoed by default, since no rule will explicitly match it.

Using the same action

To make it easier to write actions, **lex** allows you to write an action performed by several rules only one time. To abbreviate rules represented symbolically by

```
p1      action1
p2      action1
```

abbreviate by using the vertical bar operator

```
p1      |
p2      action1
```

The vertical bar means "use the action from the rule that follows".

Summary

Each **lex** specification, or program, has a specific form. Each program must have at least a definitions and rules section, and may also have a program section. Each rule has a pattern part and an action part. Actions may be made up of one or more C statements.

4. Patterns

The first part of each rule in the **lex** rules section is a pattern that matches parts of the input. This section describes how these patterns, sometimes called *regular expressions*, are constructed. 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 match themselves. An illustration of this was presented in the previous section:

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

This regular expression will match all occurrences of *removeable* that appear in the input text.

Certain characters have special meaning to **lex** patterns. To match a special character, you must *quote* it. For example, * has special meaning. In order to match the asterisk as a text character, as in the **lex** program **nsum.lex** in the previous section, surround it by quotes:

```
"*"
```

Another way to quote characters is 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 period or dot matches any character except newline. The following regular expression matches any pair of characters beginning with **J**:

```
J.
```

The following example outputs in square brackets any sequence of five characters ending with a blank. Enter the following program into the file **five.lex**

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

Compile the program:

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

Test it with

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

The result is

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

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

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

```
[0123456789]
```

will match any single decimal digit of 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 '[' signifies that any character *except* those enclosed are to be matched. The following example finds all two digit numbers not followed by a period or alphabetic character and prints them surrounded by { and }. Enter the program **twodig.lex**

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

Process and compile by typing:

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

Run the program by typing:

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

The result will be

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

Repetition

Each character in patterns shown so far will match one character at a time. Many interesting input patterns involve repetitions of characters.

To match more than one instance of a character, follow it with the pattern operator +. The summation example in **nsum.lex** shown earlier recognized strings of input numbers and added them to a total. Here is the segment of the program that recognizes numbers:

```
[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 (enter it into file **star.lex**) deletes all characters between square brackets:

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

Then, type the following lines to generate and compile the program:

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

test the program

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

The brackets are preceded by a backslash since they have special meaning in regular expressions. The result from this example is:

```
[]
[]
```

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

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

The reason that the latter result is not produced is that **lex** generated recognizers find the longest match if several matches are possible. Therefore, the first **[** was matched, then all characters up to and including the second **]** were matched.

When writing a pattern that matches many characters, you should be aware of this possibility and account for it.

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 is used to signal that the previous character or regular expression is optional. In other words, '?' signals zero or one instance of a character or regular expression. A text processor might recognize words as strings of alphabetic characters optionally followed by a period. This example (enter into file **word.lex**) will do this and output recognized words enclosed in parentheses:

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

Generate and compile the program with

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

Test the program with

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

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 include other sentence terminators as the last character of a word, the pattern is written:

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

The characters

```
.?!,
```

are optional.

The plus and asterisk repetition operators may match many characters. If you want to match a specific number of characters or patterns, follow the patterns with the repetition within braces { and }:

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

matches a string of exactly three digits.

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

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

Choices and grouping

You can indicate alternate choices of characters or regular expressions by separating them in the regular expression with a vertical bar operator |. To match either three decimal digits or the character **a**, use

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

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

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

This pattern will match 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 (enter it into **deblank.lex**) removes tabs and blanks from the beginning and end of input lines:

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

Generate and compile the program

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

Test the program by typing

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

The result will be

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

The special regular expression **\t** represents *tab*, and **\n** represents *newline*. Notice that this is the same expression used in C strings, as in the **printf** statement.

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
```

Summary

This section discusses patterns for rules. Simple patterns match specified characters one by one. Character classes match any character at a given position. Repetition of patterns can be specified for matching.

5. More patterns

This section discusses more advanced capabilities of patterns.

Line context

Like **ed**, **lex** patterns can include characters representing beginning and end of line. To match a line containing exactly five alphabetic characters,

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

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

Context matching

The slash character '/' is used to show that a following context is necessary to match a string. For example, to match the string **match** only if it is immediately followed with the string **ing**, enter the following lex program into **match.lex**:

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

Generate and compile the program:

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

Test it with

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

The result will be

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

Notice that the string before the slash is matched. The part following the slash is not matched, even though the string must be there in order for the first part to be matched. Thus, the regular expression following the slash is susceptible to further matching. To illustrate, the following example is a variant of **match.lex** to be entered into **match2.lex**

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

Generate and compile with

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

Test the program with

```
match2
Will this match?
This is a matching test.
You must now sing for your supper.
<ctrl-D>
```

The result will be

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

The context following the / can be a general regular expression. To match the whole part of a number with decimal fractions, enter the following into **wholept.lex**

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

Generate and compile the program with:

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

Test the program with

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

The result will be

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

The part of the regular expression

```
"-"?
```

matches an optional leading minus sign. Then,

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

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

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

matches the fractional part of the number. For numbers that match, the whole part of the number is printed enclosed in parentheses.

Macro abbreviations

To assist you in writing regular expressions, **lex** provides a macro facility that can substantially simplify writing complex regular expressions.

A *macro* is a named body of text. The appearance of the name of the macro is replaced by the text of the macro.

To illustrate, the following example (enter into file **float.lex**) 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);
```

The macro **e** translates to a pattern that matches a string of digits at least one digit long. The macro **d** matches the exponent part. These two are invoked in the manner of

```
{d}
```

within a pattern. Generate and compile the program by typing

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

Now, run the program:

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

The result will be

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

Context—start rules

Many **lex** tasks require processing that depends upon context. **lex** provides the ability to condition processing upon previously processed input. This is done by **start conditions**.

Start conditions are named in the definitions section by

```
%S name1 name2
```

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

```
<name1>
```

An example of use of context conditions is to use one start condition for 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 beginning with **$** to signal compiler operation. Enter the following into **comment.lex**:

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

Compile:

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

Test the program with

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

and the result will be

```
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 the context-dependent processing is more complex than that shown in the example above, it will be more convenient to use separate contexts. **lex** provides the capability to define separate contexts.

The names of the contexts are defined in the definitions sections following any start conditions definitions:

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

The **lex** function **yyswitch** is used to switch to a new context.

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

```
%C name
```

As an illustration, the following example (enter it into **pre.lex**) is part of a program that will recognize 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 one signals the beginning of a preprocessor statement. Upon recognition of this condition, this program uses **yyswitch** to activate the context **PRE**.

Within this separate context, individual rules recognize different preprocessor statements. Only two are included in this example. Each of the rules prints the preprocessor line enclosed in braces { }. Additionally, the rules switch back to the original (and unnamed) context by the statement

```
yyswitch (0);
```

Compile and test this program:

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

This will process the generated C program.

This example uses **yyswitch** to return to the original context at the end of each rule in the secondary context. Some applications will 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 declaring 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);
```

Summary

Advanced matching elements are discussed in this section. You can specify a match at the beginning and end of input lines. You may require 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.

6. More about writing actions

This section discusses predefined **lex** actions and how to use them as well as other **lex** routines 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 (enter it into **double.lex**) because many examples 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 will echo all strings of digits twice, and everyting else once:

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

Generate and compile the program with

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

Test the program with:

```
double
abcdef 123 1234
<ctrl-D>
```

The reply will be:

```
abcdef 123123 12341234
```

Processing overlapping strings

lex processing illustrated to this point is 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 the matching of strings by more than one rule, or the matching of overlapping strings. The **lex** action word **REJECT** provides this capability. When the word **REJECT** appears in a rule, other rules have a chance to match the string. Keep in mind that **lex** programs will 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);
}
```

Generate and compile the program with

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

Given an input of

```
digram
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

The actions you provide for the rules in your **lex** program are placed in a C routine named **yylex**.

If you add variable declarations in the definitions section before the first %%, **yylex** can access them, as in the **digram.lex** example above. Declarations local to **yylex** can also be provided, if you place the declarations after the rules section delimiter and before the first rule. The declaration must have a space or tab preceding it. The following is a different version of **digram.lex** called **digram2.lex** using 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

Additional code can be inserted at the beginning of the generated program by including such code in the definitions section. An example of this presented earlier called **count.lex** demonstrated this:

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

The code to be included must be preceded by a blank or tab.

However, if it is necessary to insert **include** or other C preprocessor statements at the beginning of the program, 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 other routines are needed by your version of **yywrap** or any of the rules that you write, code for them is included after a second %%. This is where **yywrap** was shown in **digram.lex**. If you wish to provide your own version of **input** or **output**, it must be defined here.

Summary

This section discussed topics concerning actions, such as **ECHO**, which echoed the matched input unchanged. Overlapping strings can be matched if **REJECT** is used to "back up" the input. The actions you write are placed in the **yylex** function, and you can add declarations to the program that are either local to **yylex** or global to the program.

7. Using lex with yacc

Although many applications are handled by **lex** alone, it is often used in conjunction with **yacc**. Typical uses include programming language compilers which have parts generated by **lex** and **yacc**.

Like **lex**, **yacc** is a program generator. Its programs recognize input structured according to a grammar fed to the **yacc** program generator. Rather than input individual characters, **yacc** generated programs are more likely to input *tokens*. Tokens are, in the context of programming languages, variable names and special characters. **lex** is especially suited for partitioning text input into tokens.

A **yacc** generated program expects a token number as input from the routine **yylex**. Each unique token type is assigned a number by **yacc** and a symbolic constant is defined for each token. The **yacc** generated program expects **yylex** to return these numbers as the value.

To access these predefined constant definitions for token types with your generated **lex** program, include the generated **lex** source in the **yacc** specification.

To illustrate putting **lex** and **yacc** generated programs together, the following program processes very simple input. Create **yacclex.yy** to contain

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

When **yacc** generates the source, it produces a file **y.tab.h** that contains the token name definitions. The **lex** specification that is to interface to a **yacc** program (enter into **yacclex.lex**) refers to that:

```
%{
#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**.

Generate the programs and compile and link the result with:

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

To demonstrate the combined program, type:

```
yacclex
(abcdef)
```

The result will be

```
recognized
```

Index

User Reaction Report

To keep this manual and COHERENT free of bugs and facilitate future improvements, we would appreciate receiving your reactions. Please fill in the appropriate sections below and mail to us. Thank you.

Mark Williams Company
1430 W. Wrightwood Avenue
Chicago, IL 60614

Name: _____

Company: _____

Address: _____

_____

Phone: _____ Date: _____

Version and hardware used: _____

Did you find any errors in the manual? _____

_____

_____

Can you suggest any improvements to the manual? _____

_____

_____

Did you find any bugs in the software? _____

_____

_____

Can you suggest improvements or enhancements to the software?

_____

_____

_____

Additional comments: (Please use other side.)