Creating an Input Language with the lex and yacc Commands

For a program to receive input, either interactively or in a batch environment, you must provide another program or a routine to receive the input. Complicated input requires additional code to break the input into pieces that mean something to the program. You can use the **lex** and **yacc** commands to develop this type of input program.

lex	Generates a lexical analyzer program that analyzes input and breaks it into tokens, such as numbers, letters, or operators. The tokens are defined by grammar rules set up in the **lex** specification file.
yacc	Generates a parser program that analyzes input using the tokens identified by the lexical analyzer (generated by the **lex** command and stored in the **lex** specification file) and performs specified actions, such as flagging improper syntax.

Writing a Lexical Analyzer Program with the lex Command

The **lex** command helps write a C language program that can receive and translate character-stream input into program actions. To use the **lex** command, you must supply or write a specification file that contains:

Extended regular expressions	Character patterns that the generated lexical analyzer recognizes.
Action statements	C language program fragments that define how the generated lexical analyzer reacts to extended regular expressions it recognizes.

The format and logic allowed in this file are discussed in the **lex** Specification File section of the **lex** command.

How the lex Command Operates

The **lex** command generates a C language program that can analyze an input stream using information in the specification file. The **lex** command then stores the output program in a **lex.yy.c** file. If the output program recognizes a simple, one-word input structure, you can compile the **lex.yy.c** output file with the following command to get an executable lexical analyzer:

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

However, if the lexical analyzer must recognize more complex syntax, you can create a parser program to use with the output file to ensure proper handling of any input.

You can move a **lex.yy.c** output file to another system if it has C compiler that supports the **lex** library functions.

The compiled lexical analyzer performs the following functions:

- Reads an input stream of characters.
- Copies the input stream to an output stream.
- Breaks the input stream into smaller strings that match the extended regular expressions in the **lex** specification file.
- Executes an action for each extended regular expression that it recognizes. These actions are C language program fragments in the **lex** specification file. Each action fragment can call actions or subroutines outside of itself.

How the Lexical Analyzer Works

The lexical analyzer that the **lex** command generates uses an analysis method called a *deterministic finite-state automaton*. This method provides for a limited number of conditions that the lexical analyzer can exist in, along with the rules that determine what state the lexical analyzer is in.

The automaton allows the generated lexical analyzer to look ahead more than one or two characters in an input stream. For example, suppose you define two rules in the **lex** specification file: one looks for the string ab and the other looks for the string abcdefg. If the lexical analyzer receives an input string of abcdefh, it reads characters to the end of input string before determining that it does not match the string abcdefg. The lexical analyzer then returns to the rule that looks for the string ab, decides that it matches part of the input, and begins trying to find another match using the remaining input cdefh.

Extended Regular Expressions in the lex Command

Specifying extended regular expressions in a **lex** specification file is similar to methods used in the **sed** or **ed** commands. An extended regular expression specifies a set of strings to be matched. The expression contains both text characters and operator characters. Text characters match the corresponding characters in the strings being compared. Operator characters specify repetitions, choices, and other features.

Numbers and letters of the alphabet are considered text characters. For example, the extended regular expression integer matches the string integer, and the expression a57D looks for the string a57D.

Operators

The following list describes how operators are used to specify extended regular expressions:

Expression	**Use**
Character	Matches the character *Character.* **Example:** a matches the literal character a; b matches the literal character b, and c matches the literal character c.
"*String*"	Matches the string enclosed within quotes, even if the string includes an operator. **Example:** to prevent the **lex** command from interpreting $ (dollar sign) as an operator, enclose the symbol in quotes.
\Character or \Digits	Escape character. When preceding a character class operator used in a string, the \ character indicates that the operator symbol represents a literal character rather than an operator. Valid escape sequences include: **\a** Alert **\b** Backspace **\f** Form-feed **\n** New-line character (Do not use the actual new-line character in an expression.) **\r** Return **\t** Tab **\v** Verticle tab \\\\ Backslash \Digits The character whose encoding is represented by the one-, two-, or three-digit octal integer specified by the *Digits* string. **\x***Digits* The character whose encoding is represented by the sequence of hexadecimal characters specified by the *Digits* string.

When the \ character precedes a character that is not in the preceding list of escape sequences, the **lex** command interprets the character literally.

Example: \c is interpreted as the c character unchanged, and [\^abc] represents the class of characters that includes the characters ^abc.

Note: Never use \0 or \x0 in lex rules.

[*List*]

Matches any one character in the enclosed range ([*x*-*y*]) or the enclosed list ([*xyz*]) based on the locale in which the **lex** command is invoked. All operator symbols, with the exception of the following, lose their special meaning within a bracket expression: - (dash), ^ (carat), and \ (backslash).

Example: [abc-f] matches a, b, c, d, e, or f in the En_US locale.

[:*Class*:]

Matches any of the characters belonging to the character class specified between the [::] delimiters as defined in the LC_TYPE category in the current locale. The following character class names are supported in all locales:

alnum cntrl lower space

alpha digit print upper

blank graph punct xdigit

The **lex** command also recognizes user-defined character class names. The [::] operator is valid only in a [] expression.

Example: [[:alpha:]] matches any character in the **alpha** character class in the current locale, but [:alpha:] matches only the characters :,a,l,p, and h.

[.*CollatingSymbol*.]

Matches the collating symbol specified within the [..] delimiters as a single character. The [..] operator is valid only in a [ ] expression. The collating symbol must be a valid collating symbol for the current locale.

Example: [[.ch.]] matches c and h together while [ch] matches c or h.

[=*CollatingElement*=]

Matches the collating element specified within the [==] delimiters and all collating elements belonging to its equivalence class. The [==] operator is valid only in a [] expression.

Example: If w and v belong to the same equivalence class, [[=w=]] is the same as [wv] and matches w or v. If w does not belong to an equivalence class, then [[=w=]] matches w only.

[^*Character*]

Matches any character except the one following the ^ (caret) symbol.The resultant character class consists solely of single-byte characters. The character following the ^ symbol can be a multibyte character, however for this operator to match multibyte characters, you must set **%h** and **%m** to greater than zero in the definitions section.

Example: [^c] matches any character except c.

CollatingElement-CollatingElement

In a character class, indicates a range of characters within the collating sequence defined for the current locale. Ranges must be in ascending order. The ending range point must collate equal to or higher than the starting range point. Because the range is based on the collating sequence of the current locale, a given range may match different characters, depending on the locale in which the **lex** command was invoked.

Expression?	Matches either zero or one occurrence of the expression immediately preceding the ? operator. **Example:** ab?c matches either ac or abc.
.	Matches any character except the new-line character. In order for . to match multi-byte characters, **%z** must be set to greater than 0 in the definitions section of the **lex** specification file. If **%z** is not set, . matches single-byte characters only.
*Expression**	Matches zero or more occurrences of the expression immediately preceding the * operator. For example, a* is any number of consecutive a characters, including zero. The usefulness of matching zero occurrences is more obvious in complicated expressions. **Example:** The expression, [A-Za-z][A-Za-z0-9]* indicates all alphanumeric strings with a leading alphabetic character, including strings that are only one alphabetic character. You can use this expression for recognizing identifiers in computer languages.
Expression+	Matches one or more occurrences of the pattern immediately preceding the + operator. **Example:** a+ matches one or more instances of a. Also, [a-z]+ matches all strings of lowercase letters.
Expression|*Expression*	Indicates a match for the expression that precedes or follows the | (pipe) operator. **Example:** ab|cd matches either ab or cd.
(*Expression*)	Matches the expression in the parentheses. The () (parentheses) operator is used for grouping and causes the expression within parentheses to be read into the **yytext** array. A group in parentheses can be used in place of any single character in any other pattern. **Example:** (ab|cd+)?(ef)* matches such strings as abefef, efefef, cdef, or cddd; but not abc, abcd, or abcdef.
^*Expression*	Indicates a match only when the ^ (carat) operator is at the beginning of the line and the ^ is the first character in an expression. **Example:** ^h matches an h at the beginning of a line.
Expression$	Indicates a match only when the $ (dollar sign) is at the end of the line and the $ is the last character in an expression. **Example:** The description of *Expression*/*Expression*.
Expression1/*Expression2*	Indicates a match only if *Expression2* immediately follows *Expression1*. The / (slash) operator reads only the first expression into the **yytext** array. **Example:** ab/cd matches the string ab, but only if followed by cd, and then reads ab into the **yytext** array. **Note:** Only one / trailing context operator can be used in a single extended regular expression. The ^ (carat) and $ (dollar sign) operators cannot be used in the same expression with the / operator as they indicate special cases of trailing context.
{*DefinedName*}	Matches the name as you defined it in the definitions section. **Example:** If you defined D to be numerical digits, {D} matches all numerical digits.

{*Number1*,*Number2*}	Matches *Number1* to *Number2* occurrences of the pattern immediately preceding it. The expressions {*Number*} and {*Number*,} are also allowed and match exactly *Number* occurrences of the pattern preceding the expression. **Example:** xyz{2,4} matches either xyzxyz, xyzxyzxyz, or xyzxyzxyzxyz. This differs from the +, * and ? operators in that these operators match only the character immediately preceding them. To match only the character preceding the interval expression, use the grouping operator. For example, xy(z{2,4}) matches xyzz, xyzzz or xyzzzz.
<*StartCondition*>	Executes the associated action only if the lexical analyzer is in the indicated start condition. **Example:** If being at the beginning of a line is start condition ONE, then the ^ (caret) operator equals the expression, <ONE>.

To use the operator characters as text characters, use one of the escape sequences: " " (double quotation marks) or \ (backslash). The " " operator indicates what is enclosed is text. Thus, the following example matches the string xyz++:

```
xyz"++"
```

Note that a part of a string may be quoted. Quoting an ordinary text character has no effect. For example, the following expression is equivalent to the previous example:

```
"xyz++"
```

Quoting all characters that are not letters or numbers ensures that text is interpreted as text.

Another way to turn an operator character into a text character is to put a \ (backslash) character before the operator character. For example, the following expression is equivalent to the preceding examples:

```
xyz\+\+
```

lex Actions

When the lexical analyzer matches one of the extended regular expressions in the rules section of the specification file, it executes the *action* that corresponds to the extended regular expression. Without sufficient rules to match all strings in the input stream, the lexical analyzer copies the input to standard output. Therefore, do not create a rule that only copies the input to the output. The default output can help find gaps in the rules.

When using the **lex** command to process input for a parser that the **yacc** command produces, provide rules to match all input strings. Those rules must generate output that the **yacc** command can interpret.

Null Action

To ignore the input associated with an extended regular expression, use a ; (C language null statement) as an action. The following example ignores the three spacing characters (blank, tab, and new-line):

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

Same As Next Action

To avoid repeatedly writing the same action, use the **|** (pipe symbol). This character indicates that the action for this rule is the same as the action for the next rule. For instance, the previous example to ignore blank, tab, and new-line characters can also be written as follows:

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

The quotation marks around \n and \t are not required.

Printing a Matched String

To find out what text matched an expression in the rules section of the specification file, you can include a C language **printf** subroutine call as one of the actions for that expression. When the lexical analyzer finds a match in the input stream, the program puts the matched string into the external character (**char**) and wide character (**wchar_t**) arrays, called **yytext** and **yywtext,** respectively. For example, you can use the following rule to print the matched string:

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

The C language **printf** subroutine accepts a format argument and data to be printed. In this example the arguments to the **printf** subroutine have the following meanings:

%s	A symbol that converts the data to type string before printing
%S	A symbol that converts the data to wide character string (**wchar_t**) before printing
yytext	The name of the array containing the data to be printed
yywtext	The name of the array containing the multibyte type (**wchar_t**) data to be printed.

The **lex** command defines **ECHO**; as a special action to print out the contents of **yytext**. For example, the following two rules are equilvalent:

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

You can change the representation of **yytext** by using either **%array** or **%pointer** in the definitions section of the **lex** specification file.

%array	Defines **yytext** as a null-terminated character array. This is the default action.
%pointer	Defines **yytext** as a pointer to a null-terminated character string.

Finding the Length of a Matched String

To find the number of characters that the lexical analyzer matched for a particular extended regular expression, use the **yyleng** or the **yywleng** external variables.

yyleng	Tracks the number of bytes that are matched.
yywleng	Tracks the number of wide characters in the matched string. Multibyte characters have a length greater than 1.

To count both the number of words and the number of characters in words in the input, use the following action:

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

This action totals the number of characters in the words matched and puts that number in chars.

The following expression finds the last character in the string matched:

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

Matching Strings within Strings

The **lex** command partitions the input stream and does not search for all possible matches of each expression. Each character is accounted for only once. To override this choice and search for items that may overlap or include each other, use the**REJECT** directive. For example, to count all instance of she and he, including the instances of he that are included in she, use the following action:

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

After counting the occurrences of she, the **lex** command rejects the input string and then counts the occurrences of he. Because he does not include she, a **REJECT** action is not necessary on he.

Getting More Input

Normally, the next string from the input stream overwrites the current entry in the **yytext** array. If you use the **yymore** subroutine, the next string from the input stream is added to the end of the current entry in the **yytext** array.

For example, the following lexical analyser looks for strings:

```
%s instring
%%
<INITIAL>\"    {  /* start of string */
        BEGIN instring;
        yymore();
       }
<instring>\"   {  /* end of string */
        printf("matched %s\n", yytext);
        BEGIN INITIAL;
       }
<instring>.    {
        yymore();
       }
<instring>\n   {
        printf("Error, new line in string\n");
        BEGIN INITIAL;
       }
```

Even though a string may be recognized by matching several rules, repeated calls to the **yymore** subroutine ensure that the **yytext** array will contain the entire string.

Putting Characters Back

To return characters to the input stream, use the call:

```
yyless(n)
```

where n is the number of characters of the current string to keep. Characters in the string beyond this number are returned to the input stream. The **yyless** subroutine provides the same type of look-ahead that the / (slash) operator uses, but it allows more control over its usage.

Use the **yyless** subroutine to process text more than once. For example, when parsing a C language program, an expression such asx=-a is difficult to understand. Does it mean x *is equal to minus* a, or is it an older representation of x -= a which means *decrease* x *by the value of* a? To treat this expression as x *is equal to minus* a, but print a warning message, use a rule such as:

```
=-[a-zA-Z]     {
            printf("Operator (=-) ambiguous\n");
            yyless(yyleng-1);
            ... action for = ...
            }
```

Input/Output Subroutines

The **lex** program allows a program to use the following input/output (I/O) subroutines:

input()	Returns the next input character.
output(c)	Writes the character c on the output.
unput(c)	Pushes the character c back onto the input stream to be read later by the **input** subroutine.
winput()	Returns the next multibyte input character.
woutput(C)	Writes the multibyte character C back onto the output stream.
wunput(C)	Pushes the multibyte character C back onto the input stream to be read by the **winput** subroutine.

lex provides these subroutines as macro definitions. The subroutines are coded in the **lex.yy.c** file. You can override them and provide other versions.

The **winput**, **wunput**, and **woutput** macros are defined to use the **yywinput**, **yywunput**, and **yywoutput** subroutines. For compatibility, the **yy** subroutines subsequently use the **input**, **unput**, and **output** subroutine to read, replace, and write the necessary number of bytes in a complete multibyte character.

These subroutines define the relationship between external files and internal characters. If you change the subroutines, change them all in the same way. They should follow these rules:

- All subroutines must use the same character set.
- The **input** subroutine must return a value of 0 to indicate end of file.
- Do not change the relationship of the **unput** subroutine to the **input** subroutine or the look-ahead functions will not work.

The **lex.yy.c** file allows the lexical analyzer to back up a maximum of 200 characters.

To read a file containing nulls, create a different version of the **input** subroutine. In the normal version of the **input** subroutine, the returned value of 0 (from the null characters) indicates the end of file and ends the input.

Character Set

The lexical analyzers that the **lex** command generates process character I/O through the **input, output,** and **unput** subroutines. Therefore, to return values in the **yytext** subroutine, the **lex** command uses the character representation that these subroutines use. Internally however, the **lex** command represents each character with a small integer. When using the standard library, this integer is the value of the bit pattern the computer uses to represent the character. Normally, the letter 'a' is represented in the same form as the character constant 'a'. If you change this interpretation with different I/O subroutines, put a translation table in the definitions section of the specification file. The translation table begins and ends with lines that contain only the entries:

```
%T
```

The translation table contains additional lines that indicate the value associated with each character. For example:

```
%T
{integer}       {character string}
{integer}       {character string}
{integer}       {character string}
%T
```

End-of-File Processing

When the lexical analyzer reaches the end of a file, it calls the **yywrap** library subroutine.

yywrap	Returns a value of 1 to indicate to the lexical analyzer that it should continue with normal wrap-up at the end of input

However, if the lexical analyzer receives input from more than one source, change the **yywrap** subroutine. The new function must get the new input and return a value of 0 to the lexical analyzer. A return value of 0 indicates the program should continue processing.

You can also include code to print summary reports and tables when the lexical analyzer ends in a new version of the **yywrap** subroutine. The **yywrap** subroutine is the only way to force the **yylex** subroutine to recognize the end of input.

Passing Code to the Generated lex Program

The **lex** command passes C code, unchanged, to the lexical analyzer in the following circumstances:

- Lines beginning with a blank or tab in the definitions section, or at the start of the rules section before the first rule, are copied into the lexical analyzer. If the entry is in the definitions section, it is copied to the external declaration area of the **lex.yy.c** file. If the entry is at the start of the rules section, the entry is copied to the local declaration area of the **yylex** subroutine in the **lex.yy.c** file.
- Lines that lie between delimiter lines containing only %{ (percent sign, left brace) and %} (percent sign, right brace) either in the definitions section or at the start of the rules section are copied into the lexical analyzer in the same way as lines beginning with a blank or tab.
- Any lines occurring after the second %% (percent sign, percent sign) delimiter are copied to the lexical analyzer without format restrictions.

Defining lex Substitution Strings

You can define string macros that the **lex** program expands when it generates the lexical analyzer. Define them before the first %% delimiter in the **lex** specification file. Any line in this section that begins in column 1 and that does not lie between%{ and %} defines a **lex** substitution string. Substitution string definitions have the general format:

```
name                translation
```

where name and translation are separated by at least one blank or tab, and the specified name begins with a letter. When the **lex** program finds the string defined by name enclosed in {} (braces) in the rules part of the specification file, it changes that name to the string defined in translation and deletes the braces.

For example, to define the names D and E, put the following definitions before the first %% delimiter in the specification file:

```
D        [0-9]
E        [DEde][-+]{D}+
```

Then, use these names in the rules section of the specification file to make the rules shorter:

```
{D}+                    printf("integer");
```

```
{D}+"."{D}*({E})?              |
{D}*"."{D}+({E})?              |
{D}+{E}                   printf("real");
```

You can also include the following items in the definitions section:

- Character set table
- List of start conditions
- Changes to size of arrays to accommodate larger source programs

lex Start Conditions

A rule may be associated with any start condition. However, the **lex** program recognizes the rule only when in that associated start condition. You can change the current start condition at any time.

Define start conditions in the *definitions* section of the specification file by using a line in the following form:

```
%Start  name1 name2
```

where name1 and name2 define names that represent conditions. There is no limit to the number of conditions, and they can appear in any order. You can also shorten the word Start to s or S.

When using a start condition in the rules section of the specification file, enclose the name of the start condition in <> (less than, greater than) symbols at the beginning of the rule. The following example defines a rule, expression, that the **lex**program recognizes only when the **lex** program is in start condition name1:

```
<name1>expression
```

To put the **lex** program in a particular start condition, execute the action statement in the action part of a rule; for instance, BEGIN in the following line:

```
BEGIN name1;
```

This statement changes the start condition to name1.

To resume the normal state, enter:

```
BEGIN 0;
```

or

```
BEGIN INITIAL;
```

where INITIAL is defined to be 0 by the **lex** program. BEGIN 0; resets the **lex** program to its initial condition.

The **lex** program also supports exclusive start conditions specified with %**x** (percent sign, lowercase x) or %**X** (percent sign, uppercase X) operator followed by a list of exclusive start names in the same format as regular start conditions. Exclusive start conditions differ from regular start conditions in that rules that do not begin with a start condition are not active when the lexical analyzer is in an exclusive start state. For example:

```
%s     one
%x     two
%%
abc    {printf("matched ");ECHO;BEGIN one;}
<one>def        printf("matched ");ECHO;BEGIN two;}
```

```
<two>ghi        {printf("matched ");ECHO;BEGIN INITIAL;}
```

In start state one in the preceding example, both abc and def can be matched. In start state two, only ghi can be matched.

Compiling the Lexical Analyzer

Compiling a **lex** program is a two-step process:

1. Use the **lex** program to change the specification file into a C language program. The resulting program is in the **lex.yy.c** file.
2. Use the **cc** command with the **-ll** flag to compile and link the program with a library of **lex** subroutines. The resulting executable program is in the **a.out** file.

For example, if the **lex** specification file is called **lextest**, enter the following commands:

```
lex lextest
cc lex.yy.c -ll
```

lex Library

The **lex** library contains the following subroutines:

main()	Invokes the lexical analyser by calling the **yylex** subroutine.
yywrap()	Returns the value 1 when the end of input occurs.
yymore()	Appends the next matched string to the current value of the **yytext** array rather than replacing the contents of the **yytext** array.
yyless(int *n***)**	Retains *n* initial characters in the **yytext** array and returns the remaining characters to the input stream.
yyreject()	Allows the lexical analyser to match multiple rules for the same input string. (**yyreject** is called when the special action **REJECT** is used.)

Some of the **lex** subroutines can be substituted by user-supplied routines. For example, **lex** supports user-supplied versions of the **main** and **yywrap** subroutines. The library versions of these routines, provided as a base, are as follows:

main

```
#include <stdio.h>
#include <locale.h>
main() {
   setlocale(LC_ALL, "");
   yylex();
   exit(0);
}
```

yywrap

```
yywrap() {
     return(1);
}
```

yymore, **yyless**, and **yyreject** subroutines are available only through the **lex** library; however, these subroutines are required only when used in **lex** actions.