NAME

lex – generate programs for lexical tasks

SYNOPSIS

lex [**–cntv**] [**–e** | **–w**] [**–V** **–Q** [**y** | **n**]] [*file*] . . .

DESCRIPTION

The **lex** utility generates C programs to be used in lexical processing of character input, and that can be used as an interface to **yacc**. The C programs are generated from **lex** source code and conform to the ISO C standard. Usually, the **lex** utility writes the program it generates to the file **lex.yy.c**; the state of this file is unspecified if **lex** exits with a non-zero exit status. See **EXTENDED DESCRIPTION** for a complete description of the **lex** input language.

OPTIONS

The following options are supported:

–c Indicate C-language action (default option).

–e Generate a program that can handle EUC characters (cannot be used with the **–w** option).
yytext[ ] is of type **unsigned char[ ]**.

–n Suppress the summary of statistics usually written with the **–v** option. If no table sizes are specified in the **lex** source code and the **–v** option is not specified, then **–n** is implied.

–t Write the resulting program to standard output instead of **lex.yy.c**.

–v Write a summary of **lex** statistics to the standard error. (See the discussion of **lex** table sizes under the heading **Definitions in lex**.) If table sizes are specified in the **lex** source code, and if the **–n** option is not specified, the **–v** option may be enabled.

–w Generate a program that can handle EUC characters (cannot be used with the **–e** option).
Unlike the **–e** option, **yytext[ ]** is of type **wchar_t[ ]**.

–V Print out version information on standard error.

–Q[y|n] Print out version information to output file **lex.yy.c** by using **–Qy**. The **–Qn** option does not print out version information and is the default.

OPERANDS

The following operand is supported:

file A pathname of an input file. If more than one such *file* is specified, all files will be concatenated to produce a single **lex** program. If no *file* operands are specified, or if a *file* operand is –, the standard input will be used.

OUTPUT

Stdout

If the **–t** option is specified, the text file of C source code output of **lex** will be written to standard output.

Stderr

If the –**t** option is specified informational, error and warning messages concerning the contents of **lex** source code input will be written to the standard error.

If the –**t** option is not specified:

1. Informational error and warning messages concerning the contents of **lex** source code input will be written to either the standard output or standard error.
2. If the –**v** option is specified and the –**n** option is not specified, **lex** statistics will also be written to standard error. These statistics may also be generated if table sizes are specified with a **%** operator in the **Definitions in lex** section (see **EXTENDED DESCRIPTION**), as long as the –**n** option is not specified.

Output Files

A text file containing C source code will be written to **lex.yy.c**, or to the standard output if the –**t** option is present.

EXTENDED DESCRIPTION

Each input file contains **lex** source code, which is a table of regular expressions with corresponding actions in the form of C program fragments.

When **lex.yy.c** is compiled and linked with the **lex** library (using the –**l l** operand with **c89** or **cc**), the resulting program reads character input from the standard input and partitions it into strings that match the given expressions.

When an expression is matched, these actions will occur:

- The input string that was matched is left in *yytext* as a null-terminated string; *yytext* is either an external character array or a pointer to a character string. As explained in **Definitions in lex**, the type can be explicitly selected using the **%array** or **%pointer** declarations, but the default is **%array**.
- The external **int** *yyleng* is set to the length of the matching string.
- The expression's corresponding program fragment, or action, is executed.

During pattern matching, **lex** searches the set of patterns for the single longest possible match. Among rules that match the same number of characters, the rule given first will be chosen.

The general format of **lex** source is:

Definitions
%%
Rules
%%
User Subroutines

The first **%%** is required to mark the beginning of the rules (regular expressions and actions); the second **%%** is required only if user subroutines follow.

Any line in the **Definitions in lex** section beginning with a blank character will be assumed to be a C program fragment and will be copied to the external definition area of the **lex.yy.c** file. Similarly, anything in the **Definitions in lex** section included between delimiter lines containing only **%{** and **%}** will also be copied unchanged to the external definition area of the **lex.yy.c** file.

Any such input (beginning with a blank character or within **%{** and **%}** delimiter lines) appearing at the beginning of the *Rules* section before any rules are specified will be written to **lex.yy.c** after the declarations of variables for the **yylex** function and before the first line of code in **yylex**. Thus, user variables local to **yylex** can be declared here, as well as application code to execute upon entry to **yylex**.

The action taken by **lex** when encountering any input beginning with a blank character or within **%{** and **%}** delimiter lines appearing in the *Rules* section but coming after one or more rules is undefined. The presence of such input may result in an erroneous definition of the **yylex** function.

Definitions in lex

Definitions in lex appear before the first **%%** delimiter. Any line in this section not contained between **%{** and **%}** lines and not beginning with a blank character is assumed to define a **lex** substitution string. The format of these lines is:

name substitute

If a *name* does not meet the requirements for identifiers in the ISO C standard, the result is undefined. The string *substitute* will replace the string *{ name }* when it is used in a rule. The *name* string is recognized in this context only when the braces are provided and when it does not appear within a bracket expression or within double-quotes.

In the **Definitions in lex** section, any line beginning with a **%** (percent sign) character and followed by an alphanumeric word beginning with either **s** or **S** defines a set of start conditions. Any line beginning with a **%** followed by a word beginning with either **x** or **X** defines a set of exclusive start conditions. When the generated scanner is in a **%s** state, patterns with no state specified will be also active; in a **%x** state, such patterns will not be active. The rest of the line, after the first word, is considered to be one or more blank-character-separated names of start conditions. Start condition names are constructed in the same way as definition names. Start conditions can be used to restrict the matching of regular expressions to one or more states as described in **Regular expressions in lex**.

Implementations accept either of the following two mutually exclusive declarations in the **Definitions in lex** section:

%array Declare the type of *yytext* to be a null-terminated character array.

%pointer Declare the type of *yytext* to be a pointer to a null-terminated character string.

The default type of *yytext* is **char[]**. If an application refers to *yytext* outside of the scanner source file (that is, via an **extern**), the application will include the appropriate **%array** or **%pointer** declaration in the scanner source file.

lex will accept declarations in the **Definitions in lex** section for setting certain internal table sizes. The declarations are shown in the following table.

Table Size Declaration in lex

Declaration	Description	Default
%p *n*	Number of positions	2500
%n *n*	Number of states	500
%a *n*	Number of transitions	2000
%e *n*	Number of parse tree nodes	1000
%k *n*	Number of packed character classes	10000
%o *n*	Size of the output array	3000

Programs generated by **lex** need either the −**e** or −**w** option to handle input that contains EUC characters from supplementary codesets. If neither of these options is specified, **yytext** is of the type **char[ ]**, and the generated program can handle only ASCII characters.

When the −**e** option is used, **yytext** is of the type **unsigned char[ ]** and **yyleng** gives the total number of *bytes* in the matched string. With this option, the macros **input( )**, **unput(***c*), and **output(***c*) should do a byte-based I/O in the same way as with the regular ASCII **lex**. Two more variables are available with the −**e** option, **yywtext** and **yywleng**, which behave the same as **yytext** and **yyleng** would under the −**w** option.

When the −**w** option is used, **yytext** is of the type **wchar_t[ ]** and **yyleng** gives the total number of *characters* in the matched string. If you supply your own **input( )**, **unput(***c***)**, or **output(***c***)** macros with this option, they must return or accept EUC characters in the form of wide character (**wchar_t**). This allows a different interface between your program and the lex internals, to expedite some programs.

When either the −**e** or −**w** option is used, the generated C program must be linked with the wide character library **libw.a** using the −**lw** linker flag.

Rules in lex

The **Rules in lex** source files are a table in which the left column contains regular expressions and the right column contains actions (C program fragments) to be executed when the expressions are recognized.

ERE action
ERE action
. . .

The extended regular expression (*ERE*) portion of a row will be separated from *action* by one or more blank characters. A regular expression containing blank characters is recognized under one of the following conditions:

- The entire expression appears within double-quotes.
- The blank characters appear within double-quotes or square brackets.
- Each blank character is preceded by a backslash character.

User Subroutines in lex

Anything in the user subroutines section will be copied to **lex.yy.c** following **yylex**.

Regular Expressions in lex

The **lex** utility supports the set of Extended Regular Expressions (ERE's) described on **regex**(5) with the following additions and exceptions to the syntax:

... Any string enclosed in double-quotes will represent the characters within the double-quotes as themselves, except that backslash escapes (which appear in the following table) are recognized. Any backslash-escape sequence is terminated by the closing quote. For example, " \ 01""1" represents a single string: the octal value 1 followed by the character 1.

<*state*>*r*

<*state1*, *state2*, . . . >*r*
The regular expression *r* will be matched only when the program is in one of the start conditions indicated by *state*, *state1*, and so forth; for more information see **Actions in lex** (As an exception to the typographical conventions of the rest of this document, in this case <*state*> does not represent a metavariable, but the literal angle-bracket characters surrounding a symbol.) The start condition is recognized as such only at the beginning of a regular expression.

r / *x* The regular expression *r* will be matched only if it is followed by an occurrence of regular expression *x*. The token returned in *yytext* will only match *r*. If the trailing portion of *r* matches the beginning of *x*, the result is unspecified. The *r* expression cannot include further trailing context or the **$** (match-end-of-line) operator; *x* cannot include the ^ (match-beginning-of-line) operator, nor trailing context, nor the **$** operator. That is, only one occurrence of trailing context is allowed in a **lex** regular expression, and the ^ operator only can be used at the beginning of such an expression.

{*name*} When *name* is one of the substitution symbols from the *Definitions* section, the string, including the enclosing braces, will be replaced by the *substitute* value. The *substitute* value will be treated in the extended regular expression as if it were enclosed in parentheses. No substitution will occur if {*name*} occurs within a bracket expression or within double-quotes.

Within an ERE, a backslash character (\\, \ **a**, \ **b**, \ **f**, \ **n**, \ **r**, \ **t**, \ **v**) is considered to begin an escape sequence. In addition, the escape sequences in the following table will be recognized.

A literal newline character cannot occur within an ERE; the escape sequence **\ n** can be used to represent a newline character. A newline character cannot be matched by a period operator.

Escape Sequences in lex

Escape Sequence	**Description**	Meaning
\ *digits*	A backslash character followed by the longest sequence of one, two or three octal-digit characters (01234567). If all of the digits are 0, (that is, representation of the NUL character), the behavior is undefined.	The character whose encoding is represented by the one-, two- or three-digit octal integer. Multi-byte characters require multiple, concatenated escape sequences of this type, including the leading \ for each byte.
\ **x***digits*	A backslash character followed by the longest sequence of hexadecimal-digit characters (01234567abcdefABCDEF). If all of the digits are 0, (that is, representation of the NUL character), the behavior is undefined.	The character whose encoding is represented by the hexadecimal integer.
\ *c*	A backslash character followed by any character not described in this table. (\ \, \ a, \ b, \ f, \ n, \ r, \ t, \ v).	The character *c*, unchanged.

The order of precedence given to extended regular expressions for **lex** is as shown in the following table, from high to low.

Note: The escaped characters entry is not meant to imply that these are operators, but they are included in the table to show their relationships to the true operators. The start condition, trailing context and anchoring notations have been omitted from the table because of the placement restrictions described in this section; they can only appear at the beginning or ending of an ERE.

ERE Precedence in lex

collation-related bracket symbols	**[= =] [: :] [. .]**
escaped characters	**\\<***special character***>**
bracket expression	**[ ]**
quoting	**"..."**
grouping	**( )**
definition	{*name*}
single-character RE duplication	∗ **+ ?**
concatenation	
interval expression	{*m*,*n*}
alternation	**|**

The ERE anchoring operators (ˆ and $) do not appear in the table. With **lex** regular expressions, these operators are restricted in their use: the ˆ operator can only be used at the beginning of an entire regular expression, and the **$** operator only at the end. The operators apply to the entire regular expression. Thus, for example, the pattern (ˆabc)|(def$) is undefined; it can instead be written as two separate rules, one with the regular expression **ˆabc** and one with **def$**, which share a common action via the special **|** action (see below). If the pattern were written **ˆabc|def$**, it would match either of **abc** or **def** on a line by itself.

Unlike the general ERE rules, embedded anchoring is not allowed by most historical **lex** implementations. An example of embedded anchoring would be for patterns such as (ˆ)foo($) to match **foo** when it exists as a complete word. This functionality can be obtained using existing **lex** features:

```
ˆfoo/[ \n]      |
" foo"/[ \n]    /* found foo as a separate word */
```

Note also that **$** is a form of trailing context (it is equivalent to **/\n**) and as such cannot be used with regular expressions containing another instance of the operator (see the preceding discussion of trailing context).

The additional regular expressions trailing-context operator **/** can be used as an ordinary character if presented within double-quotes, **"/"**; preceded by a backslash, **\/**; or within a bracket expression, **[/]**. The start-condition **<** and **>** operators are special only in a start condition at the beginning of a regular expression; elsewhere in the regular expression they are treated as ordinary characters.

The following examples clarify the differences between **lex** regular expressions and regular expressions appearing elsewhere in this document. For regular expressions of the form *r*/*x*, the string matching *r* is always returned; confusion may arise when the beginning of *x* matches the trailing portion of *r*. For example, given the regular expression a∗b/cc and the input **aaabcc**, *yytext* would contain the string **aaab** on this match. But given the regular expression x∗/xy and the input **xxxy**, the token **xxx**, not **xx**, is returned by some implementations because **xxx** matches x∗.

In the rule ab*/bc, the b* at the end of *r* will extend *r*'s match into the beginning of the trailing context, so the result is unspecified. If this rule were ab/bc, however, the rule matches the text **ab** when it is followed by the text **bc**. In this latter case, the matching of *r* cannot extend into the beginning of *x*, so the result is specified.

Actions in lex

The action to be taken when an *ERE* is matched can be a C program fragment or the special actions described below; the program fragment can contain one or more C statements, and can also include special actions. The empty C statement **;** is a valid action; any string in the **lex.yy.c** input that matches the pattern portion of such a rule is effectively ignored or skipped. However, the absence of an action is not valid, and the action **lex** takes in such a condition is undefined.

The specification for an action, including C statements and special actions, can extend across several lines if enclosed in braces:

ERE <one or more blanks> { **program statement**
program statement }

The default action when a string in the input to a **lex.yy.c** program is not matched by any expression is to copy the string to the output. Because the default behavior of a program generated by **lex** is to read the input and copy it to the output, a minimal **lex** source program that has just **%%** generates a C program that simply copies the input to the output unchanged.

Four special actions are available:

| ECHO; REJECT; BEGIN

| The action **|** means that the action for the next rule is the action for this rule. Unlike the other three actions, **|** cannot be enclosed in braces or be semicolon-terminated; it must be specified alone, with no other actions.

ECHO; Write the contents of the string *yytext* on the output.

REJECT; Usually only a single expression is matched by a given string in the input. **REJECT** means "continue to the next expression that matches the current input," and causes whatever rule was the second choice after the current rule to be executed for the same input. Thus, multiple rules can be matched and executed for one input string or overlapping input strings. For example, given the regular expressions **xyz** and **xy** and the input **xyz**, usually only the regular expression **xyz** would match. The next attempted match would start after z. If the last action in the **xyz** rule is **REJECT** , both this rule and the **xy** rule would be executed. The **REJECT** action may be implemented in such a fashion that flow of control does not continue after it, as if it were equivalent to a **goto** to another part of **yylex**. The use of **REJECT** may result in somewhat larger and slower scanners.

BEGIN The action:

BEGIN newstate;

switches the state (start condition) to *newstate*. If the string *newstate* has not been declared previously as a start condition in the **Definitions in lex** section, the results are unspecified. The initial state is indicated by the digit **0** or the token **INITIAL**.

The functions or macros described below are accessible to user code included in the **lex** input. It is unspecified whether they appear in the C code output of **lex**, or are accessible only through the **–l l** operand to **c89** or **cc** (the **lex** library).

int yylex(void)
: Performs lexical analysis on the input; this is the primary function generated by the **lex** utility. The function returns zero when the end of input is reached; otherwise it returns non-zero values (tokens) determined by the actions that are selected.

int yymore(void)
: When called, indicates that when the next input string is recognized, it is to be appended to the current value of *yytext* rather than replacing it; the value in *yyleng* is adjusted accordingly.

int yyless(int ***n*****)**
: Retains *n* initial characters in *yytext*, NUL-terminated, and treats the remaining characters as if they had not been read; the value in *yyleng* is adjusted accordingly.

int input(void)
: Returns the next character from the input, or zero on end-of-file. It obtains input from the stream pointer *yyin*, although possibly via an intermediate buffer. Thus, once scanning has begun, the effect of altering the value of *yyin* is undefined. The character read is removed from the input stream of the scanner without any processing by the scanner.

int unput(int ***c*****)**
: Returns the character *c* to the input; *yytext* and *yyleng* are undefined until the next expression is matched. The result of using *unput* for more characters than have been input is unspecified.

The following functions appear only in the **lex** library accessible through the **–l l** operand; they can therefore be redefined by a portable application:

int yywrap(void)
: Called by **yylex** at end-of-file; the default **yywrap** always will return 1. If the application requires **yylex** to continue processing with another source of input, then the application can include a function **yywrap**, which associates another file with the external variable **FILE** **yyin* and will return a value of zero.

int main(int ***argc*****, char** ****argv*****[ ])**
: Calls **yylex** to perform lexical analysis, then exits. The user code can contain **main** to perform application-specific operations, calling **yylex** as applicable.

The reason for breaking these functions into two lists is that only those functions in **libl.a** can be reliably redefined by a portable application.

Except for **input**, **unput** and **main**, all external and static names generated by **lex** begin with the prefix **yy** or **YY**.

USAGE

Portable applications are warned that in the **Rules in lex** section, an *ERE* without an action is not acceptable, but need not be detected as erroneous by **lex**. This may result in compilation or run-time errors.

The purpose of **input** is to take characters off the input stream and discard them as far as the lexical analysis is concerned. A common use is to discard the body of a comment once the beginning of a comment is recognized.

The **lex** utility is not fully internationalized in its treatment of regular expressions in the **lex** source code or generated lexical analyzer. It would seem desirable to have the lexical analyzer interpret the regular expressions given in the **lex** source according to the environment specified when the lexical analyzer is executed, but this is not possible with the current **lex** technology. Furthermore, the very nature of the lexical analyzers produced by **lex** must be closely tied to the lexical requirements of the input language being described, which will frequently be locale-specific anyway. (For example, writing an analyzer that is used for French text will not automatically be useful for processing other languages.)

EXAMPLES

The following is an example of a **lex** program that implements a rudimentary scanner for a Pascal-like syntax:

```
%{
/* need this for the call to atof() below */
#include <math.h>
/* need this for printf(), fopen() and stdin below */
#include <stdio.h>
%}

DIGIT    [0-9]
ID       [a-z][a-z0-9]*

%%

{DIGIT}+                    {
                            printf("An integer: %s (%d)\n", yytext,
                            atoi(yytext));
                            }

{DIGIT}+"."{DIGIT}*         {
                            printf("A float: %s (%g)\n", yytext,
                            atof(yytext));
                            }

if|then|begin|end|procedure|function     {
                            printf("A keyword: %s\n", yytext);
                            }

{ID}                        printf("An identifier: %s\n", yytext);

"+"|"-"|"*"|"/"             printf("An operator: %s\n", yytext);

"{"[^}\n]*"}"               /* eat up one-line comments */

[ \t\n]+                    /* eat up white space */

.                           printf("Unrecognized character: %s\n", yytext);

%%

int main(int argc, char *argv[ ])
{
                            ++argv, --argc;  /* skip over program name */
                            if (argc > 0)
                            yyin = fopen(argv[0], "r");
                            else
                            yyin = stdin;

                            yylex();
}
```

ENVIRONMENT

See **environ**(5) for descriptions of the following environment variables that affect the execution of **lex**: **LC_COLLATE**, **LC_CTYPE**, **LC_MESSAGES**, and **NLSPATH**.

EXIT STATUS

The following exit values are returned:

0 Successful completion.

>0 An error occurred.

SEE ALSO

yacc(1), **environ**(5), **regex**(5)

NOTES

If routines such as **yyback( )**, **yywrap( ),** and **yylock( )** in **.l** (ell) files are to be extern C functions, the command line to compile a C++ program must define the **__EXTERN_C__** macro, for example:

CC −D__EXTERN_C__ . . . *file*