DECUS C LANGUAGE SYSTEM

LEX

A lexical Analyser Generator

by

Charles H. Forsyth

University of Waterloo
Waterloo, Ontario, N2L 3G1
Canada

Revised by

Robert B. Denny & Martin Minow

Adapted for the Macintosh
Maarten Meijer, 90/06/19,
Using **THINK C 4.0** by Symantec Corp.

LEX transforms a regular-expression grammar and associated action routines into a C function and set of tables, yielding a table-driven lexical analyser which manages to be compact and rapid.

These notes are formatted for A4, reformat if using a different paper size.

DECUS Structured Languages SIG
Version of 30-Oct-82

Table of Contents

1.0 Introduction

A computer program often has an input stream which is composed of small elements, such as a stream of characters, and which it would like to convert to larger elements in order to process the data conveniently. A *compiler* is a common example of such a program: it reads a stream of characters forming a program, and would like to turn this sequence of characters into a sequence of larger items, namely identifiers, numbers, and operators, for parsing. In a compiler, the procedures which do this are collectively called the *lexical analyser*, or *scanner*; this terminology will be used more generally here.
It may happen that the speed with which this transformation is done will noticeably affect the speed at which the rest of the program operates. It is certainly true that although such code is rarely difficult to write, writing and debugging it is both tedious, and time-consuming, and one typically would rather spend the time on the hard parts of the program. It is also true that while certain transformations are easily thought of, they are often hard to express succinctly in the usual general-purpose programming languages (eg, the description of a floating-point number).
LEX is a program which tries to give a programmer a good deal of help in this, by writing large parts of the lexical analyser automatically, based on a description supplied by the programmer of the items to be recognised (which are known as tokens), as patterns of the more basic elements of the input stream. The **LEX** description is very much a special-purpose language for writing lexical analysers, and **LEX** is then simply a translator for this language. The **LEX** language is easy to write, and the resulting processor is compact and fast running.
The purpose of a **LEX** program is to read an input stream, and recognise tokens. As the lexical analyser will usually exist as a subroutine in a larger set of programs, it will return a "token number", which indicates which token was found, and possibly a "token value", which provides more detailed information about the token (eg, a copy of the token itself, or an index into a symbol table). This need not be the only possibility; a **LEX** program is often a good description of the structure of the whole computation, and in such a case, the lexical analyser might choose to call other routines to perform the necessary actions whenever a particular token is recognised, without reference to its own caller.

2.0 The Lex Language

LEX transforms a regular-expression grammar into a deterministic finite-state automaton that recognizes that grammar. Each rule of the grammar is associated with an action which is to be performed when that rule successfully matches some part of the input data.
Because of the nature of regular expression grammars, certain language constructions cannot be recognized by **LEX** programs.
Specifically, expressions with balanced parentheses cannot be recognized. This means that **LEX** cannot be used to recognize all Fortran keywords as some (`DO`, `IF`, and `FORMAT`, for example) require elaborate recognition to distinguish between ambiguous constructions.

2.1 Elementary Things

Strings, *characters*, sets of characters called *character classes*, and operators to form these into *patterns*, are the fundamental elements of the **LEX** language.
A *string* is a sequence of *characters*, not including newline, enclosed in quotes, or apostrophes. Within a string, the following *escape sequences* (which are those of the C language) allow any 8-bit character to be represented, including the escape character, quotes, and newlines:

```
\n          NL    (012)
\r          CR    (015)
\b          BS    (010)
\t          TAB   (011)
\"          "
\'          '
\c          c
```

```
\\              \
\NNN            (NNN)
```

where NNN is a number in octal, and c is any printable character. A string may be continued across a line by writing the escape character before the newline.
Outside a string, a sequence of upper-case letters stands for a sequence of the equivalent lower-case letters, while a sequence of lower-case letters is taken as the name of a **LEX** expression, and handled specially, as described below. These conventions make the use of named expressions and the description of lower-case keywords (the usual case on Unix) fairly convenient. Keywords in case-independent languages, such as Fortran, require additional effort to match, as will be noted.

An Ascii character other than one of

```
() {} [ * | = ; % / \ ' " -
```

may be used in LEX to stand for itself.

A sequence of characters enclosed by brackets (`'['` and `']'`) forms a character class, which stands for all those characters within the brackets. If a circumflex (`'^'`) follows the opening bracket, then the class will instead stand for all those characters but those inside the brackets. The escapes used in strings may be used in character classes as well.
Within a character class, the construction `"A-B"` (where `"A"` and `"B"` are arbitrary characters) stands for the range of characters between `"A"` and `"B"` inclusive.
For example,

`"ABC"`	matches `"abc"`
`"[ABC]"`	matches `"A"` or `"B"` or `"C"`
`"[A-Za-z0-9]"`	matches all letters and digits

Case-independent keyword recognition may be described by using auxiliary definitions to define expressions that match either case. For example,

```
a = [Aa];
b = [Bb];
...
z = [Zz];
%%
d o Matches "DO", "do", "Do", or "dO"
```

2.2 Putting Things Together

Several operators are provided to allow construction of a type of pattern called a regular expression. Such expressions can be implemented as finite-state automata (without memory or stacks). A reference to an "occurrence" of a regular expression is generally taken to mean an occurrence of any string matched by that regular expression. The operators are presented in order of decreasing priority. In all cases, operators work on either strings or character classes, or on other regular expressions.
Any string or character class forms a regular expression which matches whatever the string or character class stands for (as described above).
The operator `'*'` applied following a regular expression forms a new regular expression which matches an arbitrary number (ie, zero or more) of adjacent occurrences of the first regular expression. The operation is often referred to as (Kleene) *closure*.
The operation of *concatenation* of two regular expressions is expressed simply by writing the regular expressions adjacent to each other. The resulting regular expression matches any occurrence of the first regular expression followed directly by an occurrence of the second regular expression.
The operator `'|'`, *alternation*, written between two regular expressions forms a regular expression which matches

an occurrence of the first regular expression or an occurrence of the second regular expression.

Any regular expression may be enclosed in parentheses to cause the priority of operators to be overridden in the usual manner.
A few examples should help to make all of this clear:

`"[0-9]*"`	matches a (possibly empty) sequence of digits.
`"[A-Za-z_$][A-Za-z0-9_$]*"`	matches a C identifier.
`"([A-Za-z_$]|[0-9])*"`	matches a C identifier, or a sequence of digits, or a sequence of letters and digits intermixed, or nothing.

2.3 The General Form of Lex Programs

A **LEX** source input file consists of three sections: a section containing auxiliary definitions, a section containing translations, and a section containing programs. Throughout a **LEX** program, spaces, tabs, and newlines may be used freely, and PL/1-style comments:

```
/* ... anything but '*/' ... */
```

may be used, and are treated as a space.

The auxiliary definition section must be present, separated from following sections by the two-character sequence `'%%'`, but may be empty. This section allows definition of named regular expressions, which provide the useful ability to use names of regular expressions in the translation section, in place of common sub-expressions, or to make that section more readable.
The translation section follows the `'%%'` sequence, and contains regular expressions paired with actions which describe what the lexical analyser should do when it discovers an occurrence of a given regular expression in its input stream.
The program section may be omitted; if it is present it must be separated from the translation section by the `'%%'` sequence. If present, it may contain anything in general, as it is simply tacked on to the end of the **LEX** output file. The style of this layout will be familiar to users of **Yacc**. As **LEX** is often used with that processor, it seemed reasonable to keep to a similar format.

2.4 Auxiliary Definitions

Given the set of regular expressions forming a complete syntax, there are often common sub-expressions. **LEX** allows these to be named, defined but once, and referred to by name in any subsequent regular expression. Note that definition must precede use. A definition has the form:

```
expression_name = regular_expression ;
```

where a name is composed of a lower-case letter followed by a sequence string of letters and digits, and where an underscore is considered a letter. For example,

```
digit =      [0-9];
letter       =      [a-zA-Z];
name         =      letter(letter|digit)*;
```

The semicolon is needed to resolve some ambiguities in the **LEX** syntax.

Three auxiliary definitions have special meaning to **LEX**: "`break`", "`illegal`", and "`ignore.`" They are all defined as character classes ("`break = [,.?]`", for example) and are used as follows:

`break`	An input token will always terminate if a member of the "break" class is scanned.

`illegal`	The `"illegal"` class allows simplification of error detection, as will be described in a later section. If this class is defined, and the lexical analyser stops at a character that "cannot" occur in its present context, the analyser will output a suitable error message and ignore the offender.
`ignore`	This class defines a set of characters that are ignored by the analyser's input routine.

2.5 Translations

One would like to provide a description of the action to be taken when a particular sequence of input characters has been matched by a given regular expression. The kind of action taken might vary considerably, depending upon the application. In a compiler, typical actions are: enter an identifer into a symbol table, read and store a string, or return a particular token to the parser. In text processing, one might wish to reproduce most of the input stream on an output stream unchanged, making substitutions when a particular sequence of characters is found. In general, it is hard to predict what form the action might take, and so, in **LEX** the nature of the action is left to the user, by allowing specification, for each regular expression of interest, C-language code to be executed when a string matching that expression is discovered by the driving program of the lexical analyser. An action, together with its regular expression, is called a translation, and has the format:

```
regular_expression { action }
```

All of this may be spread across several lines. The action may be empty, but the braces must appear.
Earlier, it was argued that most general-purpose languages are inappropriate for writing lexical analysers, and it is important to see that the subsequent use of such a language to form the actions is not a contradiction. Most languages are fairly good at expressing the actions described above (symbol table manipulation, writing character strings, and such). Leaving this part of the lexical analyser to those languages therefore not only makes sense, but also ensures that decisions by the writer of the lexical analyser generator will not unduly cramp the user's style. However, general-purpose languages do not as a rule provide inexpensive pattern matching facilities, or input description formats, appropriate for describing or structuring a lexical analyser.
Allowing a user to provide his own code is not really enough, as he will need some help from **LEX** to obtain a copy of, or a pointer to, the current token, if nothing else. **LEX** provides a library of C functions which may be called to obtain controlled access to some of the data structures used by the driving programs of the lexical analyser. These are described in a later section.

2.5.1 Numbers and Values

Typically, a lexical analyser will return a value to its caller indicating which token has been found. Within an action, this is done by writing a C return statement, which returns the appropriate value:

```
BEGIN        {
                   return(T_BEGIN);
                   }
name               {
                   lookup(token(NULL));
                   return(T_NAME);
                   }
```

```
"/"                     {
                        return('/');
                        }
```

Note that function `lookup()` is provided by the user program.

In many cases, other information must be supplied to its caller by the scanner. When an identifier is recognised, for example, both a pointer to a symbol-table entry, and the token number `T_NAME` must be returned, yet the C return statement can return but a single value. **Yacc** has a similar problem, and so its lexical analyser sets an external word `'yylval'` to the token value, while the token number is returned by the scanner. LEX uses the external `'yylval'` (to be compatible), but, to make **LEX** programs more readable when used alone, the name `'lexval'` is set by a `#define` statement to `'yylval'`. For example,

```
name                {
                    lexval = lookup(token(NULL));
                    return(T_NAME);
                    }
```

Certain token numbers are treated specially; these are automatically defined as manifests (see section 3.2) by LEX, and all begin with the sequence 'LEX...' so as not to clash with the user's own names. There are two such tokens defined at present:

`LEXSKIP`	When returned by a user's action routine, `LEXSKIP` causes the lexical analyser to ignore the current token (ie, it does not inform the parser of its presence), and to look instead for a new token. This may be used when a comment sequence has been discovered, and discarded. It is also useful when the action routine completes processing of the token. See the discussion of the `comment()` library function for an example of its usage.
`LEXERR`	This is returned by the lexical analyser (function `yylex()`) when an unrecognizable inputsequence has been detected. By default, `LEXERR` is 256. This the same as the yacc error value.

To summarise, the token number is set by the action with a return statement, and the token value (ie, auxiliary information) is set by assigning this value to the external integer `'lexval'`.

2.6 Declaration Sections

Declarations in the language of the actions may be included in both the auxiliary definition section and in the translation section. In the former case, these declarations will be external to the lexical analyser, and in the latter case, they will be local to the lexical analyser (ie, `static`, or automatic storage). Declaration sections consist of a sequence of declarations surrounded by the special bracketing sequences `'%{'` and `'%}'` (as in **Yacc**). The characters within these brackets are copied unchanged into the appropriate spots in the lexical analyser program that **LEX** writes. The examples in appendix A suggest how these might be used.

3.0 Using Lex from C

The present version of **LEX** is intended for use with C; and it is this usage which will be described here.

3.1 The Function yylex()

The structure of **LEX** programs is influenced by what **Yacc** requires of its lexical analyser.
To begin with, the lexical analyser must be named `'yylex'`, has no parameters, and is expected to return a token number, where that number is determined by **Yacc**. The token number for an Ascii character is its Ascii value (ie, its value as a C character constant). Named tokens, defined in yacc `'%token'` statements, have a number above 256, with the particular number accessible through a **Yacc**-produced `#define` of the given token name as its number. **Yacc** also allows `'yylex'` to pass a value to the **Yacc** action routines, by assigning that value to the external `'yylval'`.
LEX thus provides a lexical analyser function named `'yylex'`, which interprets tables constructed by the **LEX**

program returning the token number returned by the actions it performs. Values assigned to `lexval` are available in `'yylval'`, so that use with **Yacc** is straightforward.
A value of zero is returned by `'yylex'` at end-of-file, and in the absence of a return statement in an action, a non-zero value is returned. If computation is performed entirely by the lexical analyser, then a suitable main program would be

```
main()
       {
       while (yylex()) ;
       }
```

3.2 Serial Re-Use of yylex()

The `yylex()` function contains several variables which are statically initialized at compile time. Once `yylex()` sees an `EOF` (-1) input character, it will continue to return `NULL`. If `yylex()` is to be used inside a loop which processes multiple files, it must be re-initialized at the beginning of each new file with a call to the LEX library routine `llinit()`. For example (slightly extending the previous example):

```
main()
       {
       getfilelist();
       for(file = first; file != last; file = next)
              {
              llinit();
              while (yylex());
              }
       printf("All files done\n");
       }
```

The call to `llinit()` is unnecessary if `yylex()` is to process only one file, or is kept from seeing an `EOF` input character.

3.3 The Lex Table File

In the absence of instructions to the contrary (see below), **LEX** reads a given **LEX** language file, (from the standard input, if an input file has not been specified) and produces a C program file '**lextab.c**' which largely consists of tables which are then interpreted by '`yylex()`' (which is in the **LEX** library). The actions supplied by the user in each translation are combined with a switch statement into a single function, which is called by the table interpreter when a particular token is found. The contents of the program section of the **LEX** file are added at the end of the output file (l**extab.c** by default). Normally, **LEX** also inserts the lines

```
#include    <stdio.h>
#include    <lex.h>
```

at the top of the file; this causes declarations required by the standard I/O library and by **LEX** to be included when the C program is compiled.

3.4 Analyzers Which Don't Use "Standard I/O"

With the current release, **LEX** supports the generation of analyzers which may be incorporated into programs which do not use the "**standard I/O**" library. By setting the "-s" switch, as shown below, the generation of the `"#include <stdio.h>"` line is supressed. All references to standard I/O specific files and stdio.h have been removed from the LEX library (described in a later section), with the exception of `lexgetc()`, `lexerror()`, `mapch()` and `lexecho()`, which are standard I/O dependent.
The declaration of `yylex()`'s input file iov pointer `"lexin"` now resides in LEXGET.C (`lexgetc()`). The code which defaults lexin to `stdin` has been moved from `yylex()` to the table file. `yylex()` now calls the routine `llstin()`, which is generated into the table file. There are no longer any hardwired references to the variable `"lextab"`, the default table name. Instead, **LEX** generates a call to `lexswitch()` in `llstin()`, which initializes `yylex()` to use the table whose name was given in a "-t" or "-e" option in **LEX**'s command

line. If neither was given, the default name `"lextab"` is used. Once the initial table has been set up, further automatic calls to `lexswitch()` are supressed, allowing the user to manually switch tables as before.

In addition, If the "-s" switch is not given (i.e., normal use with standard I/O), `llstin()` defaults `lexin` to `stdin`. If "-s" is given, `llstin()` is generated to do the `lexswitch()` mentioned above only. In any case, `yylex()` contains no references to the standard I/O system.

What all of this means is that under normal operation, you won't notice any change in **LEX**'s characteristics. In addition, you may use the "-e" ("easy") switch, which will generate a C output file and **LEX** tables which (conveniently) have the same name as the input file, and everything will get set up automagically. If you specify the "-s" switch, the table file will contain no references to the standard I/O package, and you may use any of the `lexlib` routines except `lexgetc()`, `lexerror()`, `mapch()` or `lexecho()`.

Don't forget that you must supply your own startup routine `"$$main"` if you do not want the standard I/O library. With a bit of care in this regard, it will be possible to link your program with the C library without dragging in any I/O modules. This prevents your having to build another library in order to access non-I/O library functions. Just make the reference to the C library the last one given to the linker or taskbuilder so that only those routines which have not already been found are pulled from CLIB.

NOTE

Programs that use LEX-generated analyzers and do not use the standard I/O package must supply their own `lexgetc()` and `lexerror()` routines. Failure to do so will result in undefined globals.

3.5 Operating LEX

Please note that all following information has been made redundant on the Macintosh by the frontend program **MacLex APPL** that writes out a ‘TEXT’ file called **Lex_args**. A short summary:

Startup MacLex APPL, this creates the options file **Lex_args** and launches **LEX**. This allows the **LEX** program to be run again with the same arguments. Examine the file **Lex_args** after some runs with different arguments to examine the details. Lex tables can be saved in resources of type 'LTAB', a general setup 'LEXT' is also created. By typing in an existing name, resources can be merged (very handy with **THINK C**), all previous versions are removed. The ID's are generated uniquely (I hope) from the `lextab` table name provided..

The minimizer doesn't work yet, but possible optimizations are shown in the info *`name`*`.lex.out` file, as are the states of the NFA's and DFA's. By the following defines all state numbers are printed to `stdout`:

```
#define     _lmovi      _lmovi_debug
#define     _lmovb      _lmovb_debug
```

Default extensions can be changed by changing the 'defaults' 'STR#' resource in **MacLex APPL**.

Please note that LEX and MacLex APPL must be in the same folder!! Do not change the name of the LEX program!! You must recompile the library yourself.

FUll documentation will follow a.s.a.p.,

Have fun,

Maarten Meijer.

PS:

The rest of this section can be ignored.

LEX normally reads the grammar from the standard input, writing the C program to the file 'lextab.c'. It may be further controlled by using the following flags upon invocation:

-i *filename*	The grammar is read from *'filename'*.
-o *filename*	The analyser is written to *'filename'*.
-t *tablename*	The default finite-state automaton is named `lextab` (and it is, by default, written to file 'lextab.c'). The -t switch causes the internal tables to be named *'tablename'* and, if the -o switch is not given, written to file *'tablename*.c'. This is necessary if the processor-switching capabilities described in a later section are to be used.

-e *name* "Easy" command line. "-e *name*" is equivalent to typing

-i *name*.LXI -o *name*.C -t *name*

Do not include device names or file extensions on the "easy" command line.

-v *[filename]* Internal state information is written to *'filename'*. If not present, state information is written to file 'lex.out.'

-d Enable various debugging printouts.

-s Generate analyzer without references to standard I/O

The command line for compilation of the table file should contain no surprises:

```
cc -c -O lextab.c (on Unix)
xcc lextab -a            (on Dec operating systems)
```

but when one is producing the running program, one must be careful to include the necessary libraries. On Unix, the proper sequence is:

```
cc userprog.o lextab.o -ll -lS
```

The '-ll' causes the LEX library (described below) to be searched, and the '-lS' causes the Standard I/O library to be used; both libraries are required. If Yacc is used as well, the library '-ly' should be included before the '-ll'. The actual order and content of the rest of the command line is determined by the user's own requirements.
If using the Decus C compiler, the lexical analyser built by LEX is linked with c:lexlib.
The complete process (assuming the Decus compiler running on RSTS/E in RT11 mode) is thus:

```
mcr lex -i grammar.lxi -o grammar.c ! Build analyser
cc grammar ! Compile the
as grammar ! grammar table
link out=in,grammar,c:lexlib,c:suport,c:clib/b:2000
```

4.0 The Lex Library

All programs using grammars generated by **LEX** must be linked together with the **LEX** library. On Unix, this is '/lib/libl.a' (or '-ll' on the cc command line) and on DEC operating systems, C:LEXLIB (LB:[1,1]LEX.OLB for RSX). It contains routines which are either essential or merely useful to users of **LEX**. The essential routines include a routine to obtain a copy of the current token, and a routine to switch to a different set of scanning tables. Routines of the second, useful, class perform functions which might well be written by the user himself, but are there to save him the bother; including a routine to process various forms of comments and a routine to transform numbers written in arbitrary bases. Both sets of routines are expected to grow as **LEX** sees use.
Those functions which produce diagnostics do so by calling `lexerror()`, which is called as
`lexerror(string, arg1, ..., argN)` and is expected to write its arguments (likely using the "remote format" facility of the `fprintf()` function), followed by a newline, on some output stream. A `lexerror()` function is included in the LEX library, but a user is free to include his own. The routine in the LEX library is standard I/O specific.

NOTE
The VAX/VMS native C library does not support remote formats. The Lexerror function in the LEX library conditionally compiles to support a call to `lexerror()` with only an error message string. Remote

formats are supported under Decus C. Learn to use them, they are very nice!

4.0.1 Comment -- skip over a comment

```
comment(delim)
char delim[];
```

`Comment()` may be called by a translation when the sequence of characters which mark the start of a comment in the given syntax has been recognised by **LEX**. It takes a string which gives the sequence of characters which mark the end of a comment, and skips over characters in the input stream until this sequence is found. Newlines found while skipping characters cause the external `'yyline'` to be incremented; an unexpected end-of-file produces a suitable diagnostic. Thus, `'comment("*/")'` matches C-style comments, and `'comment("\n")'` matches as-style comments. There are other methods of handling comments in **LEX**; the `comment()` function is usually the best with regard to both space and time.

4.0.2 Gettoken -- obtain a copy of token

```
gettoken(buf, sizeof(buf))
char buf[];
```

`Gettoken()` takes the address of a character buffer, and its size in bytes, and copies the token most recently matched by **LEX** into the buffer. A null byte is added to mark the end of the token in the buffer, but, as null bytes are legitimate characters to **LEX**, the true length of the token is returned by `gettoken()`.
For example, the following function calls `lexlength()` to obtain the length of a token. It then calls the storage allocator to allocate sufficient storage for the token and copies the token into the allocated area.

```
char *
save()
/*
*      Save current token, return a pointer to it
*/
       {
       register char *tbuffer;
       register int len;
       register char *tend;
       extern char *token();
       extern char *copy();

       len = lexlength() + 1;
       if (tbuffer = malloc(len)) == NULL)
             error("No room for token");
       gettoken(tbuffer, len);
       return(tbuffer);
       }
```

4.0.3 Integ -- long integer, any base

```
long
integ(nptr, base)
char *nptr;
```

`Integ()` converts the Ascii string at '`nptr'` into a long integer, which it returns. Conversion stops at the first non-digit, where the digits are taken from the class `"[0-9a-zA-Z]"` as limited by the given 'base'. `Integ()` does not understand signs, nor are blanks or tabs allowed in the string.

4.0.4 Lexchar -- steal character

```
lexchar()
```

`Lexchar()` returns the next character from the **LEX** input stream. (This means that **LEX** will no longer see it.) **LEX** uses a *look-ahead buffer* to handle complex languages, and this function takes this into account.

4.0.5 Lexecho -- write token to a file (STDIO ONLY)

```
lexecho(fp);
FILE *fp;
```

`Lexecho()` may be called by a **LEX** action routine to write the current token to a specified file.

NOTE

Programs using analyzers built with **LEX**'s "-s" switch must supply their own lexecho() function if needed.

4.0.6 Lexgetc -- supply characters to yylex (STDIO ONLY)

```
lexgetc()
```

`Lexgetc()` is called by the lexical analyser to obtain characters from its input stream. The version in the library is dependent on the standard I/O package, and is:

```
FILE *lexin; /* Declare iov address locally */
lexgetc()
{
return(getc(lexin));
}
```

If `lexin` is `NULL` when `yylex()` is entered, it will be assigned to `stdin`. This is done by `yylex()` calling the function `llstin()`, which is generated in the table file. Unless the "-s" switch is given to **LEX**, the `llstin()` function assigns `lexin` to `stdin` if `lexin` is `NULL`. If the "-s" switch was given, the `llstin()` routine is a no-op. The user may provide his own version of `lexgetc()` to pre-process the data to the lexical analyser. An example of this is shown in the appendix.

NOTE

Programs using analyzers built with **LEX**'s "-s" switch must supply their own `lexgetc()` function, and `"lexin"` has no meaning in this context.

4.0.7 Lexlength -- return length of a token

```
lexlength();
```

`Lexlength()` may be called by a **LEX** action routine to obtain the length of the current token in bytes. An example of this is shown in the description of `gettoken()`.

4.0.8 Lexpeek -- examine character

```
lexpeek()
```

`Lexpeek()` performs a function similar to that of `Lexchar()`, but does not have the side-effect of removing the character from **LEX**'s view.

4.0.9 Lexswitch -- switch scanning tables

```
struct lextab *
lexswitch(newtb)
struct lextab *newtb;
```

`Lexswitch()` is called to cause **LEX** to use a different scanning table; it returns a pointer to the one previously in use. This facility is useful if certain objects of the language (eg, strings in C) have a fairly complicated structure of their own which cannot be handled within the translation section of the **LEX** description of the larger language.

4.0.10 Llinit -- Reinitialize yylex()

```
llinit()
```

`Llinit()` is a function which resets the state of `yylex()` to it's cold-start condition. Several of `yylex()`'s variables are initialized at compile time, and must be reinitialized if it is to be serially re-used. An example of this is where `yylex()` is repeatedly called inside a loop which processes multiple input files. Each time a new file is started, `llinit()` must be called before the first call to `yylex()` for the new file.

4.0.11 Mapch -- Handle C escapes within strings (STDIO ONLY)

```
int mapch(delim, esc)
char delim;
char esc;
```

`Mapch()` is a function which handles C "escape" characters such as `"\n"` and `"\nnn"`. It will scan off the entire escape sequence and return the equivalent ASCII code as an integer. It is meant for use with **YACC** while scanning quoted strings and character constants.
If it encounters `EOF` while scanning, it calls `lexerror()` to print an error message warning of "Unterminated string". If a *normal character* is read, it returns the ASCII value. If "delim" (usually " or ') is read, it returns `EOF`. If a *newline* (ASCII linefeed) is read, it increments the global `"yyline"` and calls itself recursively for the next line of input. It may use the `ungetc()` function to back up in the input stream.

> NOTE
> This routine is very application-specific for use by **LEX** and **YACC** when they are working together. You should read the code in MAPCH.C before using this function.

4.0.12 Token -- get pointer to token

```
char *
token(end_pointer)
char **end_pointer;
```

`Token()` locates the first byte of the current token and returns its address. It takes an argument which is either `NULL` or a pointer to a character pointer; if the latter, that pointer is set to point to the byte after the last byte of the current token. `Token()` is slightly faster, and more convenient than `gettoken()` for those cases where the token is only one or two bytes long.

5.0 Error Detection and Recovery

If a character is detected in the input stream which cannot be added to the last-matched string, and which cannot start a string, then that character is considered illegal by **LEX**. **LEX** may be instructed to return a special 'error' token, or to write a diagnostic with `lexerror()`. At present, the former is the default action.
The token `LEXERR` is a special value which is recognised by **Yacc**, and causes it to start its own error recovery. It is defined by the header file `lex.h` for use by other programs.
Often, it makes more sense to simply type a suitable diagnostic, and continue by ignoring the offending character. It is fairly easy to cause **LEX** to do this, by including the auxiliary definition:

```
illegal      =      [\0-\377];
```

which defines a character class `"illegal"` which is handled specially by **LEX**. If the character that is causing the trouble is a member of that character class (and in the example, all characters are), then **LEX** will write a diagnostic, and ignore it; otherwise, it will return the special token `LEXERR`
More comprehensive techniques may be added as they become apparent.

6.0 Ambiguity and Look-ahead

Many computer languages have ambiguous grammars in that an input token may represent more than one logical entity. This section discusses the way in which grammars built by **LEX** resolve ambiguous input, as well as a way for the grammar to assign unique meaning to a token by looking ahead in the input stream.

6.1 Resolving Ambiguities

A **LEX** program may be ambiguous, in the sense that a particular input string or strings might be matched by the regular expression of more than one translation. Consider,

```
[a-z]        { putchar(*token(NULL)); }
aaa*               { printf("abc"); }
```

in which the string 'aa' is matched by both regular expressions (twice by the first, and once by the second). Also, the string `'aaaaaa'` may be matched in many different ways. **LEX** has to decide somehow which actions should be performed. (Alternatively, it could produce a diagnostic, and give up. As it happens, LEX never does this.) Consider a second example,

```
letter       =     [a-z];
%%
A(letter)*  { return(1); }
AB(letter)* { return(2); }
```

which attempts to distinguish sequences of letters that begin with `'a'` from similar sequences that begin with `'ab'`. These two examples illustrate two different kinds of ambiguity, and the following indicates how **LEX** resolves these.
In the first example, it seems likely that the intent was to have both `'aa'` and `'aaaaaa'` perform the second action, while all single letters `'a'` cause the first action to be performed. **LEX** does this by ensuring that the longest possible part of the input stream will be used to determine the match. Thus,

```
<                  { return(LESS); }
<=                 { return(LESSEQ); }
```

or

```
digit(digit)*      { return(NUMBER); }
letter(letter|digit)*
                   { return(NAME); }
```

would work as one might expect.
In the second example, the longest-string need not work. On the string `"abb9"`, either action could apply, and so another rule must be followed. This states that if, after the longest-string rule has been applied, there remains an ambiguity, then the action which appears first in the **LEX** program file is to be performed. As the second example is written, the second action will never be performed. It would have been written as:

```
letter       =     [a-z];
%%
AB(letter)* { return(1); }
A(letter)*  { return(2); }
```

The two rules together completely determine a string.
At present, **LEX** produces no diagnostic in either case; it merely applies the rules and proceeds. In the case where priority is given to the first-appearing rule, it might be a good idea to produce a diagnostic.

6.2 Look-ahead

Some facility for looking ahead in the input stream is sometimes required. (This facility might also be regarded as a way for the programmer to more closely control **LEX's** ambiguity resolution process.) For example, in C, a name followed by `"("` is to be contextually declared as an external function if it is otherwise undefined.
In Pascal, look-ahead is required to determine that

```
123..1234
```

is an integer `123`, followed by the subrange symbol `".."`, followed by the integer `1234,` and not simply two real numbers run together.
In both of these cases, the desire is to look ahead in the input stream far enough to be able to make a decision, but without losing tokens in the process.
A special form of regular expression is used to indicate look-ahead:

```
re1 '/' re2 '{' action '}'
```

where `'re1'` and `'re2'` are regular expressions. The slash is treated as concatenation for the purposes of matching incoming characters; thus both `'re1'` and `'re2'` must match adjacently for the action to be performed. `'Re1'` indicates that part of the input string which is the token to be returned, while `'re2'` indicates the context. The characters matched by `'re2'` will be re-read at the next call to `yylex()`, and broken into tokens.
Note that you cannot write:

```
name = re1 / re2;
```

The look-ahead operator must be part of the rule. It is not valid in definitions.
In the first example, the look-ahead operator would be used as:

```
name / "("  {
                    if (name undefined)
                    declare name a global function;
                    }
name                {
                    /* usual processing for identifiers */
                    }
```

In the second example, the range construction would be parsed as follows:

```
digit =      [0-9];
int          =     digit(digit)*;
%%
int / ".." int { /* Start of a range */
".." int { /* End of a range */
```

Note that right-context is not sufficient to handle certain types of ambiguity, as is found in several places in the **Fortran** language. For example,

```
do i = 1                Is an assignment statement
do i = 1,  4  Is a DO statement
```

It is not sufficient to use right-context scanning to look for the comma, as it may occur within a parenthesized sub-expression:

```
do i = j(k,l)           Is an assignment statement
```

In **Fortran**, similar problems exist for `IF` and `FORMAT` statements, as well as counted (Hollerith) string constants. All of these require a more powerful grammar than is possible with **LEX** regular-expressions.

7.0 Multiple Scanning Tables; Processor Switching

Even a fairly simple syntax may be difficult, or impossible, to describe and process with a single set of translations. An example of this may be found in C, where strings, which are part of the language, have quite a different structure, and in order to process them, either a function must be called which reads and parses the input stream for itself, or some mechanism within **LEX** must be invoked to cause a (usually massive) change of state.
LEX does provide such a facility, which is known, after AED, as 'processor switching'. `Yylex()` locates its tables through a pointer; if one simply changes the pointer to point at a new set of tables, one will have effected the required change of state. The **LEX** library function `lexswitch()`, which is described elsewhere in this guide, arranges to do this; it also returns the old value of the pointer so that it may be restored by a later call to `Lexswitch()`. Thus, scanning environments may be stacked, or not, as the user requires.

7.1 Creation of a Processor

It should be clear that if all the tables produced by **LEX** from a user's translation file have the same name, someone (the loader) is bound to object. Some method must be provided to change the name of the table.
This is done by an option flag to the **LEX** command:

-t *name*

will cause the scanning table to be declared as

```
struct lextab name;
```

so that it may be passed to LEXswitch:

```
lexswitch(&name);
```

LEX also writes the program file to the file "*name*.c" rather than to "lextab.c."

NOTE
If you use the "easy" command line ("-e *name*") when running **LEX**, the output file and table names will correspond nicely. Re-read the section on operating **LEX** for more details.

8.0 Conclusion

LEX seems to handle most lexical analysis tasks easily. Indeed, **LEX** may be more generally used to write commands of a text-processing nature; an example of such usage may be found in an appendix. **LEX** programs are far easier to write than the equivalent C programs, and generally consume less space (although there is an initial overhead for the more general table-interpreter program). The encoding suggested in [4] achieves a reasonable compromise between table size, and scanning speed. Certainly lexical analysers are less tedious and time-consuming to write.
It is expected that most change in the future will be through additions to the **LEX** library. The **LEX** language may change slightly to accomodate common kinds of

processing (eg, break characters), or to extend its range of application. Neither kind of change should affect existing LEX programs.
LEX produces tables and programs for the C language. The tables are in a very simple (and stylised) format, and when **LEX** copies the action routines or the program section, the code might as well be **Fortran** for all it cares. One could write Unix filters to translate the very simple C format tables into other languages, allowing **LEX** to be used with a larger number of languages, with little extra development cost. This seems a likely future addition.
Because of the look-ahead necessary to implement the "longest string match" rule, **LEX** is unsuitable for interactive programs whose overall structure is:

```
for (;;) {
      prompt_user();
      get_input();
      process();
```

```
    prnt_output();
          }
```

If these are rewritten as **LEX**-generated grammars, the user will be confused by the fact the second input datum must be entered before the first is processed. It is possible to solve this dilemna by rewriting function `lexgetc()` to return an "end-of-line" character until processing is complete for that line. An example is shown in the Appendix.

9.0 Acknowledgements

LEX is based on a processor of the same name at Bell Laboratories, which also runs under Unix [3], and, more distantly, on AED-0 [1]. This version of **LEX** was based on the description and suggestions of [4], although the implementation differs significantly in a number of ways.

10.0 References

1. Johnson, W.L., et. al., *"Automatic generation of efficient lexical analysers using finite state techniques"*, CACM Vol. 11, No. 12, pp. 805-813, 1968.

2. Johnson, S.C., *"Yacc -- Yet Another Compiler-Compiler"*, CSTR-32, Bell Telephone Laboratories, Murray Hill, New Jersey, 1974.

3. Lesk, M.E., *"Lex - a lexical analyser generator"*, CSTR-39, Bell Telephone Laboratories, Murray Hill, New Jersey, 1975.

4. Aho, A.V., Ullman, J.D., *Principles of Compiler Design*, Addison-Wesley, Don Mills, Ontario, 1977.

LEX SOURCE GRAMMAR

The following is a grammar of LEX programs which generally follows Bacus-Naur conventions. In the rules, "||" stands for alternation (choose one or the other). Other graphic text stands for itself. Several grammar elements have special meaning:

<anything>	Any text not including the following grammar element (either a literal or end-of-file).
<nothing>	Nothing -- used for optional rule elements.
<name>	A variable name.
<char_class>	A character class specifier.
<string>	A string (text inclosed in "").
<EOF>	The end of the input file.

This grammar was abstracted from the **Yacc** grammar used to describe **LEX**.

```
program         ::=   aux_section trans_section
aux_section     ::=   auxiliaries %%
                ||         %%
auxiliaries     ::=   auxiliaries aux_def
                ||         aux_def
aux_def         ::=   name_def = reg_exp ;
                ||         %{ <anything> %}
name_def        ::=   <name>
reg_exp         ::=   <char_class>
                ||         <string>
                ||         <name>
                ||         reg_exp *
                ||         reg_exp | reg_exp
                ||         reg_exp reg_exp
                ||         ( reg_exp )
trans_section   ::=   translations
                ||         <nothing>
translations    ::=   translations translation
                ||         translation
translation     ::=   pattern action
                ||         %{ <anything> %}
                ||         %% <anything> <EOF>
pattern         ::=   reg_exp / reg_exp
                ||         reg_exp
```

SOME SMALL EXAMPLES

The following example illustrates the use of the look-ahead operator, and various other of the nuances of using LEX.

B.1 A Complete Command

The C programming language has had two different ways of writing its assignment operators. The original method was to write a binary operator immediately following the ordinary assignment operator, forming a compound operator. Thus 'a =+ b' caused the value of 'a+b' to be assigned to 'a'. Similarly,

```
=- =/ =% =* =<< =>> =| =& =^
```

were written for the assignment operators corresponding to subtraction, division, modulus, multiplication, left shift, right shift, logical OR, logical AND, and exclusive OR. In the current version of the language, the binary operator is written to the left of the assignment operator, to remove potential ambiguity.
The LEX program "ctoc" is a filter which converts programs written in the older style into programs written in the newer style. It uses the look-ahead operator, and the various dis-ambiguating rules to ensure that sequences like `a==-1` `a=++b` remain unchanged.

```
/*
*      ctoc.lxi -- Convert old C operators to new C form
*
*      Adapted from example in C. Forsythe's LEX manual.
*
*      NOTE:
*      Forsythe's program put an entire comment into the token
*      buffer. Either define a huge token buffer for my typical
*      monster comments, or filter text within comments as if
*      it were real C code. This is what I did. So =+ inside
*      a comment will get changed to +=, etc. Note tnat you
*      may use the commen() function in LEXLIB if you want the
*      comments eaten. I wanted 'em in the output.
*      by
*      Bob Denny
*      31-Feb-81
*/
%{
char tbuf[80]; /* Token buffer */
main()
      {
      while (yylex())
      ;
      }
%}
any          =     [\0-\177];
nesc         =     [^\\];
nescquote    =     [^\\"];
nescapost    =     [^\\'];
schar =      "\\" any | nescquote;
cchar =      "\\" any | nescapost;
string       =     '"' schar* '"';
charcon      =     "'" cchar* "'";
%%
"=" ( << | >> | "*" | + | - | "/" | "%" | "&" | "|" | "^" )
      {
      gettoken(tbuf, sizeof tbuf);
      printf("%s=",tbuf+1);
      }
/*
```

```
*      The following will overflow the token buffer on any but a
*     small comment:
*/
/*********
"/*" ([^*] | "*"[^/])* "*/"
{
lexecho(stdout);
}
**********/
[<=>!]"=" | "="[<>]
                  {
                  lexecho(stdout);
                  }
"=" / ( ++ | -- )
                  {
                  lexecho(stdout);
                  }
charcon
                  {
                  lexecho(stdout);
                  }
string
                  {
                  lexecho(stdout);
                  }
[\0-\377]
                  {
                  lexecho(stdout);
                  }
```

Assuming the Decus compiler running on RSTS/E in RT11 mode, the above program would be built and executed as follows:

```
mcr lex -i ctoc.lxi -o ctoc.c
cc ctoc/v
as ctoc/d
link ctoc=ctoc,c:lexlib,c:suport,c:clib/b:2000
mcr ctoc <old.c >new.c
```

B.2 Interactive Lexical Analysis

The following program reads words from the terminal, counting each as they are entered. The interaction with the operator is "natural" in the sense that processing for one line is complete before the next line is input. To implement this program, it was necessary to include a special version of `lexgetc()` which returns <NULL> if the current line has been completely transmitted to the parser. Because the parser must still have some look-ahead context, it will return the "end-of-line" token twice at the beginning of processing. This required some additional tests in the main program.

```
/*
*       Count words -- interactively
*/
white =       [\n\t ];     /* End of a word */
eol           =       [\0]; /* End of input line */
any           =       [!-~];       /* All printing char's */
illegal       =       [\0-\377];  /* Skip over junk */
%{
char line[133];
char *linep = &line;
int iseof = 0;
int wordct = 0;
#define T_EOL 1
main()
      {
      register int i;
      while ((i = yylex()) != 0) {
/*
*     If the "end-of-line" token is
*     returned AND we're really at
*     the end of a line, read the
*     next line. Note that T_EOL is
*     returned twice when the program
*     starts because of the nature of
*     the look-ahead algorithms.
*/
            if (i == T_EOL && !is_eof&& *linep == 0) {
                  printf("* ");
                  fflush(stdout);
                  getline();
                  }
            }
      printf("%d words\n", wordct);
      }
%}
%%
any(any)*           {
/*
*     Write each word on a
*     seperate output line.
*/
                  lexecho(stdout);
                  printf("\n");
                  wordct++;
                  return(LEXSKIP);
                  }
eol               {
                  return(T_EOL);
                  }
white(white)* {
      return(LEXSKIP);
```

```
          }
%%
```

```
getline()
/*
*     Read a line for lexgetc()
*/
      {
      is_eof = (fgets(line, sizeof line, stdin) == NULL);
      linep = &line;
      }

lexgetc()
/*
*     Homemade lexgetc -- return zero while at the
*     end of an input line or EOF at end of file. If
*     more on this line, return the next byte.
*/
      {
      return( (is_eof) ? EOF
      : (*linep == 0) ? 0
      : *linep++);
      }
```