Lex – A Lexical Analyzer Generator

M. E. Lesk and E. Schmidt

ABSTRACT

Lex helps write programs whose control flow is directed by instances of regular expressions in the input stream. It is well suited for editor-script type transformations and for segmenting input in preparation for a parsing routine.

Lex source is a table of regular expressions and corresponding program fragments. The table is translated to a program which reads an input stream, copying it to an output stream and partitioning the input into strings which match the given expressions. As each such string is recognized the corresponding program fragment is executed. The recognition of the expressions is performed by a deterministic finite automaton generated by Lex. The program fragments written by the user are executed in the order in which the corresponding regular expressions occur in the input stream.

The lexical analysis programs written with Lex accept ambiguous specifications and choose the longest match possible at each input point. If necessary, substantial lookahead is performed on the input, but the input stream will be backed up to the end of the current partition, so that the user has general freedom to manipulate it.

Lex can generate analyzers in either C or Ratfor, a language which can be translated automatically to portable Fortran. It is available on the PDP-11 UNIX, Honeywell GCOS, and IBM OS systems. This manual, however, will only discuss generating analyzers in C on the UNIX system, which is the only supported form of Lex under UNIX Version 7. Lex is designed to simplify interfacing with Yacc, for those with access to this compiler-compiler system.

July 21, 1975

Table of Contents

1. Introduction.

Lex is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem oriented specification for character string matching, and produces a program in a general purpose language which recognizes regular expressions. The regular expressions are specified by the user in the source specifications given to Lex. The Lex written code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. At the boundaries between strings program sections provided by the user are executed. The Lex source file associates the regular expressions and the program fragments. As each expression appears in the input to the program written by Lex, the corresponding fragment is executed.

The user supplies the additional code beyond expression matching needed to complete his tasks, possibly including code written by other generators. The program that recognizes the expressions is generated in the general purpose programming language employed for the user's program fragments. Thus, a high level expression language is provided to write the string expressions to be matched while the user's freedom to write actions is unimpaired. This avoids forcing the user who wishes to use a string manipulation language for input analysis to write processing programs in the same and often inappropriate string handling language.

Lex is not a complete language, but rather a generator representing a new language feature which can be added to different programming languages, called ‘‘host languages.’’ Just as general purpose languages can produce code to run on different computer hardware, Lex can write code in different host languages. The host language is used for the output code generated by Lex and also for the program fragments added by the user. Compatible run-time libraries for the different host languages are also provided. This makes Lex adaptable to different environments and different users. Each application may be directed to the combination of hardware and host language appropriate to the task, the user's background, and the properties of local implementations. At present, the only supported host language is C, although Fortran (in the form of Ratfor [2] has been available in the past. Lex itself exists on UNIX, GCOS, and OS/370; but the code generated by Lex may be taken anywhere the appropriate compilers exist.

Lex turns the user's expressions and actions (called ***source*** in this memo) into the host general-purpose language; the generated program is named ***yylex***. The ***yylex*** program will recognize expressions in a stream (called ***input*** in this memo) and perform the specified actions for each expression as it is detected. See Figure 1.

```
Source →  | Lex   |  → yylex

Input →   | yylex | → Output
```

An overview of Lex

Figure 1

For a trivial example, consider a program to delete from the input all blanks or tabs at the ends of lines.

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

is all that is required. The program contains a %% delimiter to mark the beginning of the rules, and one rule. This rule contains a regular expression which matches one or more instances of the characters blank or tab (written \t for visibility, in accordance with the C language convention) just prior to the end of a line. The brackets indicate the character class made of blank and tab; the + indicates ‘‘one or more ...’’; and the $ indicates ‘‘end of line,’’ as in QED. No action is specified, so the program generated by Lex (yylex) will ignore these characters. Everything else will be copied. To change any remaining string of blanks or tabs to a single blank, add another rule:

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

The finite automaton generated for this source will scan for both rules at once, observing at the termination of the string of blanks or tabs whether or not there is a newline character, and executing the desired rule action. The first rule matches all strings of blanks or tabs at the end of lines, and the second rule all remaining strings of blanks or tabs.

Lex can be used alone for simple transformations, or for analysis and statistics gathering on a lexical level. Lex can also be used with a parser generator to perform the lexical analysis phase; it is particularly easy to interface Lex and Yacc [3]. Lex programs recognize only regular expressions; Yacc writes parsers that accept a large class of context free grammars, but require a lower level analyzer to recognize input tokens. Thus, a combination of Lex and Yacc is often appropriate. When used as a preprocessor for a later parser generator, Lex is used to partition the input stream, and the parser generator assigns structure to the resulting pieces. The flow of control in such a case (which might be the first half of a compiler, for example) is shown in Figure 2. Additional programs, written by other generators or by hand, can be added easily to programs written by Lex.

```
          lexical      grammar
           rules        rules
             ↓            ↓
          | Lex |     | Yacc |
             ↓            ↓
Input →  | yylex |  → | yyparse | → Parsed input
```

Lex with Yacc

Figure 2

Yacc users will realize that the name ***yylex*** is what Yacc expects its lexical analyzer to be named, so that the use of this name by Lex simplifies interfacing.

Lex generates a deterministic finite automaton from the regular expressions in the source [4]. The automaton is interpreted, rather than compiled, in order to save space. The result is still a fast analyzer. In particular, the time taken by a Lex program to recognize and partition an input stream is proportional to the length of the input. The number of Lex rules or the complexity of the rules is not important in determining speed, unless rules which include forward context require a significant amount of rescanning. What does increase with the number and complexity of rules is the size of the finite automaton, and therefore the size of the program generated by Lex.

In the program written by Lex, the user's fragments (representing the ***actions*** to be performed as each regular expression is found) are gathered as cases of a switch. The automaton interpreter directs the control flow. Opportunity is provided for the user to insert either declarations or additional statements in the routine containing the actions, or to add subroutines outside this action routine.

Lex is not limited to source which can be interpreted on the basis of one character lookahead. For example, if there are two rules, one looking for ***ab*** and another for ***abcdefg***, and the input stream is ***abcdefh***, Lex will recognize ***ab*** and leave the input pointer just before ***cd. . .*** Such backup is more costly than the processing of simpler languages.

2. Lex Source.

The general format of Lex source is:

```
{definitions}
%%
{rules}
%%
{user subroutines}
```

where the definitions and the user subroutines are often omitted. The second ***%%*** is optional, but the first is required to mark the beginning of the rules. The absolute minimum Lex program is thus

```
%%
```

(no definitions, no rules) which translates into a program which copies the input to the output unchanged.

In the outline of Lex programs shown above, the ***rules*** represent the user's control decisions; they are a table, in which the left column contains ***regular expressions*** (see section 3) and the right column contains ***actions,*** program fragments to be executed when the expressions are recognized. Thus an individual rule might appear

```
integer     printf("found keyword INT");
```

to look for the string ***integer*** in the input stream and print the message ‘‘found keyword INT’’ whenever it appears. In this example the host procedural language is C and the C library function ***printf*** is used to print the string. The end of the expression is indicated by the first blank or tab character. If the action is merely a single C expression, it can just be given on the right side of the line; if it is compound, or takes more than a line, it should be enclosed in braces. As a slightly more useful example, suppose it is desired to change a number of words from British to American spelling. Lex rules such as

```
colour        printf("color");
mechanise     printf("mechanize");
petrol        printf("gas");
```

would be a start. These rules are not quite enough, since the word ***petroleum*** would become ***gaseum***; a way of dealing with this will be described later.

3. Lex Regular Expressions.

The definitions of regular expressions are very similar to those in QED [5]. A regular expression specifies a set of strings to be matched. It contains text characters (which match the corresponding characters in the strings being compared) and operator characters (which specify repetitions, choices, and other features). The letters of the alphabet and the digits are always text characters; thus the regular expression

```
integer
```

matches the string ***integer*** wherever it appears and the expression

```
a57D
```

looks for the string *a57D.*

Operators. The operator characters are

" \ [ ] ˆ – ? . ∗ + | ( ) $ / { } % < >

and if they are to be used as text characters, an escape should be used. The quotation mark operator (") indicates that whatever is contained between a pair of quotes is to be taken as text characters. Thus

xyz"++"

matches the string *xyz++* when it appears. Note that a part of a string may be quoted. It is harmless but unnecessary to quote an ordinary text character; the expression

"xyz++"

is the same as the one above. Thus by quoting every non-alphanumeric character being used as a text character, the user can avoid remembering the list above of current operator characters, and is safe should further extensions to Lex lengthen the list.

An operator character may also be turned into a text character by preceding it with \ as in

xyz\+\+

which is another, less readable, equivalent of the above expressions. Another use of the quoting mechanism is to get a blank into an expression; normally, as explained above, blanks or tabs end a rule. Any blank character not contained within [ ] (see below) must be quoted. Several normal C escapes with \ are recognized: \n is newline, \t is tab, and \b is backspace. To enter \ itself, use \\. Since newline is illegal in an expression, \n must be used; it is not required to escape tab and backspace. Every character but blank, tab, newline and the list above is always a text character.

Character classes. Classes of characters can be specified using the operator pair [ ]. The construction *[abc]* matches a single character, which may be *a* , *b* , or *c* . Within square brackets, most operator meanings are ignored. Only three characters are special: these are \ – and ˆ. The – character indicates ranges. For example,

[a–z0–9<>_]

indicates the character class containing all the lower case letters, the digits, the angle brackets, and underline. Ranges may be given in either order. Using – between any pair of characters which are not both upper case letters, both lower case letters, or both digits is implementation dependent and will get a warning message. (E.g., [0–z] in ASCII is many more characters than it is in EBCDIC). If it is desired to include the character – in a character class, it should be first or last; thus

[–+0–9]

matches all the digits and the two signs.

In character classes, the ˆ operator must appear as the first character after the left bracket; it indicates that the resulting string is to be complemented with respect to the computer character set. Thus

[ˆabc]

matches all characters except a, b, or c, including all special or control characters; or

[ˆa–zA–Z]

is any character which is not a letter. The \ character provides the usual escapes within character class brackets.

Arbitrary character. To match almost any character, the operator character

.

is the class of all characters except newline. Escaping into octal is possible although non-portable:

[\40–\176]

matches all printable characters in the ASCII character set, from octal 40 (blank) to octal 176 (tilde).

Optional expressions. The operator ***?*** indicates an optional element of an expression. Thus

ab?c

matches either ***ac*** or ***abc***.

Repeated expressions. Repetitions of classes are indicated by the operators ∗ and + .

***a*∗**

is any number of consecutive ***a*** characters, including zero; while

a+

is one or more instances of ***a.*** For example,

[a–z]+

is all strings of lower case letters. And

[A–Za–z][A–Za–z0–9]∗

indicates all alphanumeric strings with a leading alphabetic character. This is a typical expression for recognizing identifiers in computer languages.

Alternation and Grouping. The operator | indicates alternation:

(ab | cd)

matches either ***ab*** or ***cd.*** Note that parentheses are used for grouping, although they are not necessary on the outside level;

ab | cd

would have sufficed. Parentheses can be used for more complex expressions:

(ab | cd+)?(ef)∗

matches such strings as ***abefef***, ***efefef***, ***cdef***, or ***cddd*** ; but not ***abc***, ***abcd***, or ***abcdef***.

Context sensitivity. Lex will recognize a small amount of surrounding context. The two simplest operators for this are ˆ and ***$***. If the first character of an expression is ˆ, the expression will only be matched at the beginning of a line (after a newline character, or at the beginning of the input stream). This can never conflict with the other meaning of ˆ, complementation of character classes, since that only applies within the [ ] operators. If the very last character is ***$***, the expression will only be matched at the end of a line (when immediately followed by newline). The latter operator is a special case of the / operator character, which indicates trailing context. The expression

ab/cd

matches the string ***ab***, but only if followed by ***cd.*** Thus

ab$

is the same as

ab/\n

Left context is handled in Lex by ***start conditions*** as explained in section 10. If a rule is only to be executed when the Lex automaton interpreter is in start condition ***x***, the rule should be prefixed by

<x>

using the angle bracket operator characters. If we considered ‘‘being at the beginning of a line’’ to be start condition ***ONE***, then the ˆ operator would be equivalent to

<ONE>

Start conditions are explained more fully later.

Repetitions and Definitions. The operators {} specify either repetitions (if they enclose numbers) or definition expansion (if they enclose a name). For example

{digit}

looks for a predefined string named ***digit*** and inserts it at that point in the expression. The definitions are given in the first part of the Lex input, before the rules. In contrast,

a{1,5}

looks for 1 to 5 occurrences of ***a*** .

Finally, initial ***%*** is special, being the separator for Lex source segments.

4. Lex Actions.

When an expression written as above is matched, Lex executes the corresponding action. This section describes some features of Lex which aid in writing actions. Note that there is a default action, which consists of copying the input to the output. This is performed on all strings not otherwise matched. Thus the Lex user who wishes to absorb the entire input, without producing any output, must provide rules to match everything. When Lex is being used with Yacc, this is the normal situation. One may consider that actions are what is done instead of copying the input to the output; thus, in general, a rule which merely copies can be omitted. Also, a character combination which is omitted from the rules and which appears as input is likely to be printed on the output, thus calling attention to the gap in the rules.

One of the simplest things that can be done is to ignore the input. Specifying a C null statement, ***;*** as an action causes this result. A frequent rule is

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

which causes the three spacing characters (blank, tab, and newline) to be ignored.

Another easy way to avoid writing actions is the action character |, which indicates that the action for this rule is the action for the next rule. The previous example could also have been written

```
" "
"\t"
"\n"
```

with the same result, although in different style. The quotes around \n and \t are not required.

In more complex actions, the user will often want to know the actual text that matched some expression like *[a–z]+*. Lex leaves this text in an external character array named *yytext.* Thus, to print the name found, a rule like

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

will print the string in *yytext.* The C function *printf* accepts a format argument and data to be printed; in this case, the format is ‘‘print string’’ (% indicating data conversion, and *s* indicating string type), and the data are the characters in *yytext.* So this just places the matched string on the output. This action is so common that it may be written as ECHO:

```
[a–z]+      ECHO;
```

is the same as the above. Since the default action is just to print the characters found, one might ask why give a rule, like this one, which merely specifies the default action? Such rules are often required to avoid matching some other rule which is not desired. For example, if there is a rule which matches *read* it will normally match the instances of *read* contained in *bread* or *readjust*; to avoid this, a rule of the form *[a–z]+* is needed. This is explained further below.

Sometimes it is more convenient to know the end of what has been found; hence Lex also provides a count *yyleng* of the number of characters matched. To count both the number of words and the number of characters in words in the input, the user might write

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

which accumulates in *chars* the number of characters in the words recognized. The last character in the string matched can be accessed by

```
yytext[yyleng–1]
```

Occasionally, a Lex action may decide that a rule has not recognized the correct span of characters. Two routines are provided to aid with this situation. First, *yymore()* can be called to indicate that the next input expression recognized is to be tacked on to the end of this input. Normally, the next input string would overwrite the current entry in *yytext.* Second, *yyless (n)* may be called to indicate that not all the characters matched by the currently successful expression are wanted right now. The argument *n* indicates the number of characters in *yytext* to be retained. Further characters previously matched are returned to the input. This provides the same sort of lookahead offered by the / operator, but in a different form.

Example: Consider a language which defines a string as a set of characters between quotation (") marks, and provides that to include a " in a string it must be preceded by a \. The regular expression which matches that is somewhat confusing, so that it might be preferable to write

```
\"[^"]*      {
             if (yytext[yyleng–1] == ´\\´)
                  yymore();
             else
                  ... normal user processing
             }
```

which will, when faced with a string such as *"abc\"def"* first match the five characters *"abc*; then the call to *yymore()* will cause the next part of the string, *"def*, to be tacked on the end. Note that the final quote terminating the string should be picked up in the code labeled ‘‘normal processing’’.

The function *yyless()* might be used to reprocess text in various circumstances. Consider the C problem of distinguishing the ambiguity of ‘‘=–a’’. Suppose it is desired to treat this as ‘‘=– a’’ but print a message. A rule might be

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

which prints a message, returns the letter after the operator to the input stream, and treats the operator as ‘‘=–’’. Alternatively it might be desired to treat this as ‘‘= –a’’. To do this, just return the minus sign as well as the letter to the input:

```
=–[a–zA–Z]    {
                printf("Operator (=–) ambiguous\n");
                yyless(yyleng–2);
                ... action for = ...
                }
```

will perform the other interpretation. Note that the expressions for the two cases might more easily be written

```
=–/[A–Za–z]
```

in the first case and

```
=/–[A–Za–z]
```

in the second; no backup would be required in the rule action. It is not necessary to recognize the whole identifier to observe the ambiguity. The possibility of ‘‘=–3’’, however, makes

```
=–/[ˆ \t\n]
```

a still better rule.

In addition to these routines, Lex also permits access to the I/O routines it uses. They are:

1) *input()* which returns the next input character;

2) *output(c)* which writes the character *c* on the output; and

3) *unput(c)* pushes the character *c* back onto the input stream to be read later by *input().*

By default these routines are provided as macro definitions, but the user can override them and supply private versions. These routines define the relationship between external files and internal characters, and must all be retained or modified consistently. They may be redefined, to cause input or output to be transmitted to or from strange places, including other programs or internal memory; but the character set used must be consistent in all routines; a value of zero returned by *input* must mean end of file; and the relationship between *unput* and *input* must be retained or the Lex lookahead will not work. Lex does not look ahead at all if it does not have to, but every rule ending in *+ ∗ ?* or *$* or containing */* implies lookahead. Lookahead is also necessary to match an expression that is a prefix of another expression. See below for a discussion of the character set used by Lex. The standard Lex library imposes a 100 character limit on backup.

Another Lex library routine that the user will sometimes want to redefine is ***yywrap()*** which is called whenever Lex reaches an end-of-file. If ***yywrap*** returns a 1, Lex continues with the normal wrapup on end of input. Sometimes, however, it is convenient to arrange for more input to arrive from a new source. In this case, the user should provide a ***yywrap*** which arranges for new input and returns 0. This instructs Lex to continue processing. The default ***yywrap*** always returns 1.

This routine is also a convenient place to print tables, summaries, etc. at the end of a program. Note that it is not possible to write a normal rule which recognizes end-of-file; the only access to this condition is through ***yywrap.*** In fact, unless a private version of ***input()*** is supplied a file containing nulls cannot be handled, since a value of 0 returned by ***input*** is taken to be end-of-file.

5. Ambiguous Source Rules.

Lex can handle ambiguous specifications. When more than one expression can match the current input, Lex chooses as follows:

1) The longest match is preferred.

2) Among rules which matched the same number of characters, the rule given first is preferred.

Thus, suppose the rules

```
integer    keyword action ...;
[a–z]+     identifier action ...;
```

to be given in that order. If the input is ***integers***, it is taken as an identifier, because ***[a–z]+*** matches 8 characters while ***integer*** matches only 7. If the input is ***integer***, both rules match 7 characters, and the keyword rule is selected because it was given first. Anything shorter (e.g. ***int***) will not match the expression ***integer*** and so the identifier interpretation is used.

The principle of preferring the longest match makes rules containing expressions like **.**∗ dangerous. For example,

′.∗′

might seem a good way of recognizing a string in single quotes. But it is an invitation for the program to read far ahead, looking for a distant single quote. Presented with the input

′first′ quoted string here, ′second′ here

the above expression will match

′first′ quoted string here, ′second′

which is probably not what was wanted. A better rule is of the form

′[ˆ′\n]∗′

which, on the above input, will stop after ***′first′***. The consequences of errors like this are mitigated by the fact that the **.** operator will not match newline. Thus expressions like **.**∗ stop on the current line. Don't try to defeat this with expressions like ***[.\n]+*** or equivalents; the Lex generated program will try to read the entire input file, causing internal buffer overflows.

Note that Lex is normally partitioning the input stream, not searching for all possible matches of each expression. This means that each character is accounted for once and only once. For example, suppose it is desired to count occurrences of both ***she*** and ***he*** in an input text. Some Lex rules to do this might be

```
she     s++;
he      h++;
\n      |
.       ;
```

where the last two rules ignore everything besides ***he*** and ***she***. Remember that . does not include newline. Since ***she*** includes ***he***, Lex will normally ***not*** recognize the instances of ***he*** included in ***she***, since once it has passed a ***she*** those characters are gone.

Sometimes the user would like to override this choice. The action REJECT means ‘‘go do the next alternative.’’ It causes whatever rule was second choice after the current rule to be executed. The position of the input pointer is adjusted accordingly. Suppose the user really wants to count the included instances of ***he***:

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

these rules are one way of changing the previous example to do just that. After counting each expression, it is rejected; whenever appropriate, the other expression will then be counted. In this example, of course, the user could note that ***she*** includes ***he*** but not vice versa, and omit the REJECT action on ***he***; in other cases, however, it would not be possible a priori to tell which input characters were in both classes.

Consider the two rules

```
a[bc]+     { ... ; REJECT;}
a[cd]+     { ... ; REJECT;}
```

If the input is ***ab***, only the first rule matches, and on ***ad*** only the second matches. The input string ***accb*** matches the first rule for four characters and then the second rule for three characters. In contrast, the input ***accd*** agrees with the second rule for four characters and then the first rule for three.

In general, REJECT is useful whenever the purpose of Lex is not to partition the input stream but to detect all examples of some items in the input, and the instances of these items may overlap or include each other. Suppose a digram table of the input is desired; normally the digrams overlap, that is the word ***the*** is considered to contain both ***th*** and ***he***. Assuming a two-dimensional array named ***digram*** to be incremented, the appropriate source is

```
%%
[a–z][a–z]     {digram[yytext[0]][yytext[1]]++; REJECT;}
\n             ;
```

where the REJECT is necessary to pick up a letter pair beginning at every character, rather than at every other character.

6. Lex Source Definitions.

Remember the format of the Lex source:

```
{definitions}
%%
```

```
{rules}
%%
{user routines}
```

So far only the rules have been described. The user needs additional options, though, to define variables for use in his program and for use by Lex. These can go either in the definitions section or in the rules section.

Remember that Lex is turning the rules into a program. Any source not intercepted by Lex is copied into the generated program. There are three classes of such things.

1) Any line which is not part of a Lex rule or action which begins with a blank or tab is copied into the Lex generated program. Such source input prior to the first %% delimiter will be external to any function in the code; if it appears immediately after the first %%, it appears in an appropriate place for declarations in the function written by Lex which contains the actions. This material must look like program fragments, and should precede the first Lex rule.

   As a side effect of the above, lines which begin with a blank or tab, and which contain a comment, are passed through to the generated program. This can be used to include comments in either the Lex source or the generated code. The comments should follow the host language convention.

2) Anything included between lines containing only *%{* and *%}* is copied out as above. The delimiters are discarded. This format permits entering text like preprocessor statements that must begin in column 1, or copying lines that do not look like programs.

3) Anything after the third %% delimiter, regardless of formats, etc., is copied out after the Lex output.

Definitions intended for Lex are given before the first %% delimiter. Any line in this section not contained between %{ and %}, and begining in column 1, is assumed to define Lex substitution strings. The format of such lines is

```
name translation
```

and it causes the string given as a translation to be associated with the name. The name and translation must be separated by at least one blank or tab, and the name must begin with a letter. The translation can then be called out by the {name} syntax in a rule. Using {D} for the digits and {E} for an exponent field, for example, might abbreviate rules to recognize numbers:

```
D                   [0–9]
E                   [DEde][–+]?{D}+
%%
{D}+                printf("integer");
{D}+"."{D}∗({E})?   |
{D}∗"."{D}+({E})?   |
{D}+{E}
```

Note the first two rules for real numbers; both require a decimal point and contain an optional exponent field, but the first requires at least one digit before the decimal point and the second requires at least one digit after the decimal point. To correctly handle the problem posed by a Fortran expression such as *35.EQ.I*, which does not contain a real number, a context-sensitive rule such as

```
[0−9]+/"."EQ      printf("integer");
```

could be used in addition to the normal rule for integers.

The definitions section may also contain other commands, including the selection of a host language, a character set table, a list of start conditions, or adjustments to the default size of arrays within Lex itself for larger source programs. These possibilities are discussed below under ‘‘Summary of Source Format,’’ section 12.

7. Usage.

There are two steps in compiling a Lex source program. First, the Lex source must be turned into a generated program in the host general purpose language. Then this program must be compiled and loaded, usually with a library of Lex subroutines. The generated program is on a file named *lex.yy.c*. The I/O library is defined in terms of the C standard library [6].

The C programs generated by Lex are slightly different on OS/370, because the OS compiler is less powerful than the UNIX or GCOS compilers, and does less at compile time. C programs generated on GCOS and UNIX are the same.

UNIX. The library is accessed by the loader flag *−ll*. So an appropriate set of commands is

```
lex source cc lex.yy.c −ll
```

The resulting program is placed on the usual file *a.out* for later execution. To use Lex with Yacc see below. Although the default Lex I/O routines use the C standard library, the Lex automata themselves do not do so; if private versions of *input, output* and *unput* are given, the library can be avoided.

8. Lex and Yacc.

If you want to use Lex with Yacc, note that what Lex writes is a program named *yylex(),* the name required by Yacc for its analyzer. Normally, the default main program on the Lex library calls this routine, but if Yacc is loaded, and its main program is used, Yacc will call *yylex().* In this case each Lex rule should end with

```
return(token);
```

where the appropriate token value is returned. An easy way to get access to Yacc's names for tokens is to compile the Lex output file as part of the Yacc output file by placing the line

```
# include "lex.yy.c"
```

in the last section of Yacc input. Supposing the grammar to be named ‘‘good’’ and the lexical rules to be named ‘‘better’’ the UNIX command sequence can just be:

```
yacc good
lex better
cc y.tab.c −ly −ll
```

The Yacc library (–ly) should be loaded before the Lex library, to obtain a main program which invokes the Yacc parser. The generations of Lex and Yacc programs can be done in either order.

9. Examples.

As a trivial problem, consider copying an input file while adding 3 to every positive number divisible by 7. Here is a suitable Lex source program

```
%%
        int k;
[0–9]+  {
        k = atoi(yytext);
        if (k%7 == 0)
             printf("%d", k+3);
        else
             printf("%d",k);
        }
```

to do just that. The rule [0–9]+ recognizes strings of digits; ***atoi*** converts the digits to binary and stores the result in ***k.*** The operator % (remainder) is used to check whether ***k*** is divisible by 7; if it is, it is incremented by 3 as it is written out. It may be objected that this program will alter such input items as ***49.63*** or ***X7***. Furthermore, it increments the absolute value of all negative numbers divisible by 7. To avoid this, just add a few more rules after the active one, as here:

```
%%
                        int k;
–?[0–9]+                {
                        k = atoi(yytext);
                        printf("%d", k%7 == 0 ? k+3 : k);
                        }
–?[0–9.]+               ECHO;
[A-Za-z][A-Za-z0-9]+    ECHO;
```

Numerical strings containing a ‘‘.’’ or preceded by a letter will be picked up by one of the last two rules, and not changed. The ***if–else*** has been replaced by a C conditional expression to save space; the form ***a?b:c*** means ‘‘if ***a*** then ***b*** else ***c***’’.

For an example of statistics gathering, here is a program which histograms the lengths of words, where a word is defined as a string of letters.

```
            int lengs[100];
%%
[a–z]+      lengs[yyleng]++;
.           |
\n          ;
%%
yywrap()
{
int i;
printf("Length  No. words\n");
for(i=0; i<100; i++)
     if (lengs[i] > 0)
          printf("%5d%10d\n",i,lengs[i]);
return(1);
}
```

This program accumulates the histogram, while producing no output. At the end of the input it prints the table. The final statement ***return(1);*** indicates that Lex is to perform wrapup. If ***yywrap*** returns zero (false) it implies that further input is available and the program is to continue reading and processing. To provide a ***yywrap*** that never returns true causes an infinite loop.

As a larger example, here are some parts of a program written by N. L. Schryer to convert double precision Fortran to single precision Fortran. Because Fortran does not distinguish upper and lower case letters, this routine begins by defining a set of classes including both cases of each letter:

```
a    [aA]
b    [bB]
c    [cC]
...
z    [zZ]
```

An additional class recognizes white space:

```
W    [ \t]*
```

The first rule changes ‘‘double precision’’ to ‘‘real’’, or ‘‘DOUBLE PRECISION’’ to ‘‘REAL’’.

```
{d}{o}{u}{b}{l}{e}{W}{p}{r}{e}{c}{i}{s}{i}{o}{n} {
    printf(yytext[0]=='d'? "real" : "REAL");
    }
```

Care is taken throughout this program to preserve the case (upper or lower) of the original program. The conditional operator is used to select the proper form of the keyword. The next rule copies continuation card indications to avoid confusing them with constants:

```
^"     "[^ 0]    ECHO;
```

In the regular expression, the quotes surround the blanks. It is interpreted as ‘‘beginning of line, then five blanks, then anything but blank or zero.’’ Note the two different meanings of **^**. There follow some rules to change double precision constants to ordinary floating constants.

```
[0–9]+{W}{d}{W}[+–]?{W}[0–9]+    |
[0–9]+{W}"."{W}{d}{W}[+–]?{W}[0–9]+    |
"."{W}[0–9]+{W}{d}{W}[+–]?{W}[0–9]+    {
    /* convert constants */
    for(p=yytext; *p != 0; p++)
        {
        if (*p == 'd' || *p == 'D')
            *p=+ 'e'– 'd';
        ECHO;
        }
```

After the floating point constant is recognized, it is scanned by the ***for*** loop to find the letter ***d*** or ***D***. The program than adds ***'e'–'d'***, which converts it to the next letter of the alphabet. The modified constant, now single-precision, is written out again. There follow a series of names which must be respelled to remove their initial ***d***. By using the array ***yytext*** the same action suffices for all the names (only a sample of a rather long list is given here).

```
{d}{s}{i}{n}          |
{d}{c}{o}{s}          |
{d}{s}{q}{r}{t}       |
{d}{a}{t}{a}{n}       |
...
{d}{f}{l}{o}{a}{t}    printf("%s",yytext+1);
```

Another list of names must have initial ***d*** changed to initial ***a***:

```
{d}{l}{o}{g}      |
{d}{l}{o}{g}10    |
{d}{m}{i}{n}1     |
{d}{m}{a}{x}1     {
                  yytext[0] =+ ′a′ − ′d′;
                  ECHO;
                  }
```

And one routine must have initial ***d*** changed to initial ***r***:

```
{d}1{m}{a}{c}{h}    {yytext[0] =+ ′r′  − ′d′;
```

To avoid such names as ***dsinx*** being detected as instances of ***dsin***, some final rules pick up longer words as identifiers and copy some surviving characters:

```
[A−Za−z][A−Za−z0−9]∗    |
[0−9]+                  |
\n                      |
.                       ECHO;
```

Note that this program is not complete; it does not deal with the spacing problems in Fortran or with the use of keywords as identifiers.

10. Left Context Sensitivity.

Sometimes it is desirable to have several sets of lexical rules to be applied at different times in the input. For example, a compiler preprocessor might distinguish preprocessor statements and analyze them differently from ordinary statements. This requires sensitivity to prior context, and there are several ways of handling such problems. The ˆ operator, for example, is a prior context operator, recognizing immediately preceding left context just as ***$*** recognizes immediately following right context. Adjacent left context could be extended, to produce a facility similar to that for adjacent right context, but it is unlikely to be as useful, since often the relevant left context appeared some time earlier, such as at the beginning of a line.

This section describes three means of dealing with different environments: a simple use of flags, when only a few rules change from one environment to another, the use of ***start conditions*** on rules, and the possibility of making multiple lexical analyzers all run together. In each case, there are rules which recognize the need to change the environment in which the following input text is analyzed, and set some parameter to reflect the change. This may be a flag explicitly tested by the user's action code; such a flag is the simplest way of dealing with the problem, since Lex is not involved at all. It may be more convenient, however, to have Lex remember the flags as initial conditions on the rules. Any rule may be associated with a start condition. It will only be recognized when Lex is in that start condition. The current start condition may be changed at any time. Finally, if the sets of rules for the different environments are very dissimilar, clarity may be best achieved by writing several distinct lexical analyzers, and switching from one to another as desired.

Consider the following problem: copy the input to the output, changing the word ***magic*** to ***first*** on every line which began with the letter ***a***, changing ***magic*** to ***second*** on every line which began with the letter ***b***, and changing ***magic*** to ***third*** on every line which began with the letter ***c***. All other words and all other lines are left unchanged.

These rules are so simple that the easiest way to do this job is with a flag:

```
        int flag;
%%
^a      {flag = 'a'; ECHO;}
^b      {flag = 'b'; ECHO;}
^c      {flag = 'c'; ECHO;}
\n      {flag =  0 ; ECHO;}
magic   {
        switch (flag)
        {
        case 'a': printf("first"); break;
        case 'b': printf("second"); break;
        case 'c': printf("third"); break;
        default: ECHO; break;
        }
        }
```

should be adequate.

To handle the same problem with start conditions, each start condition must be introduced to Lex in the definitions section with a line reading

```
%Start    name1 name2 ...
```

where the conditions may be named in any order. The word ***Start*** may be abbreviated to ***s*** or ***S***. The conditions may be referenced at the head of a rule with the <> brackets:

```
<name1>expression
```

is a rule which is only recognized when Lex is in the start condition ***name1***. To enter a start condition, execute the action statement

```
BEGIN name1;
```

which changes the start condition to ***name1***. To resume the normal state,

```
BEGIN 0;
```

resets the initial condition of the Lex automaton interpreter. A rule may be active in several start conditions:

```
<name1,name2,name3>
```

is a legal prefix. Any rule not beginning with the <> prefix operator is always active.

The same example as before can be written:

```
%START AA BB CC
%%
^a          {ECHO; BEGIN AA;}
^b          {ECHO; BEGIN BB;}
^c          {ECHO; BEGIN CC;}
\n          {ECHO; BEGIN 0;}
<AA>magic   printf("first");
<BB>magic   printf("second");
<CC>magic   printf("third");
```

where the logic is exactly the same as in the previous method of handling the problem, but Lex does the work rather than the user's code.

11. Character Set.

The programs generated by Lex handle character I/O only through the routines ***input, output,*** and ***unput.*** Thus the character representation provided in these routines is accepted by Lex and employed to return values in ***yytext.*** For internal use a character is represented as a small integer which, if the standard library is used, has a value equal to the integer value of the bit pattern representing the character on the host computer. Normally, the letter ***a*** is represented as the same form as the character constant ***'a'*** . If this interpretation is changed, by providing I/O routines which translate the characters, Lex must be told about it, by giving a translation table. This table must be in the definitions section, and must be bracketed by lines containing only ''%T''. The table contains lines of the form

{integer} {character string}

which indicate the value associated with each character. Thus the next example

```
%T
 1      Aa
 2      Bb
...
26      Zz
27      \n
28      +
29      –
30      0
31      1
...
39      9
%T
```

Sample character table.

maps the lower and upper case letters together into the integers 1 through 26, newline into 27, + and – into 28 and 29, and the digits into 30 through 39. Note the escape for newline. If a table is supplied, every character that is to appear either in the rules or in any valid input must be included in the table. No character may be assigned the number 0, and no character may be assigned a bigger number than the size of the hardware character set.

12. Summary of Source Format.

The general form of a Lex source file is:

```
{definitions}
%%
{rules}
%%
{user subroutines}
```

The definitions section contains a combination of

1) Definitions, in the form ''name space translation''.

2) Included code, in the form ''space code''.

3) Included code, in the form

```
%{
code
%}
```

4) Start conditions, given in the form

%S name1 name2 ...

5) Character set tables, in the form

%T

number space character-string

...

%T

6) Changes to internal array sizes, in the form

%*x* ***nnn***

where ***nnn*** is a decimal integer representing an array size and ***x*** selects the parameter as follows:

Letter	Parameter
p	positions
n	states
e	tree nodes
a	transitions
k	packed character classes
o	output array size

Lines in the rules section have the form ‘‘expression action’’ where the action may be continued on succeeding lines by using braces to delimit it.

Regular expressions in Lex use the following operators:

x — the character "x"

"x" — an "x", even if x is an operator.

\x — an "x", even if x is an operator.

[xy] — the character x or y.

[x–z] — the characters x, y or z.

[ˆx] — any character but x.

. — any character but newline.

ˆx — an x at the beginning of a line.

<y>x — an x when Lex is in start condition y.

x$ — an x at the end of a line.

x? — an optional x.

x* — 0,1,2, ... instances of x.

x+ — 1,2,3, ... instances of x.

x|y — an x or a y.

(x) — an x.

x/y — an x but only if followed by y.

{xx} — the translation of xx from the definitions section.

x{m,n} — ***m*** through ***n*** occurrences of x

13. Caveats and Bugs.

There are pathological expressions which produce exponential growth of the tables when converted to deterministic machines; fortunately, they are rare.

REJECT does not rescan the input; instead it remembers the results of the previous scan. This means that if a rule with trailing context is found, and REJECT executed, the user must not have used ***unput*** to change the characters forthcoming from the input stream. This is the only restriction on the user’s ability to manipulate the not-yet-processed input.

14. Acknowledgments.

As should be obvious from the above, the outside of Lex is patterned on Yacc and the inside on Aho’s string matching routines. Therefore, both S. C. Johnson and A. V. Aho are really originators of much of Lex, as well as debuggers of it. Many thanks are due to both.

The code of the current version of Lex was designed, written, and debugged by Eric Schmidt.

15. References.

1. B. W. Kernighan and D. M. Ritchie, ***The C Programming Language,*** Prentice-Hall, N. J. (1978).

2. B. W. Kernighan, ***Ratfor: A Preprocessor for a Rational Fortran,*** Software – Practice and Experience, **5**, pp. 395-496 (1975).

3. S. C. Johnson, ***Yacc: Yet Another Compiler Compiler,*** Computing Science Technical Report No. 32, 1975, Bell Laboratories, Murray Hill, NJ 07974.

4. A. V. Aho and M. J. Corasick, ***Efficient String Matching: An Aid to Bibliographic Search,*** Comm. ACM **18,** 333-340 (1975).

5. B. W. Kernighan, D. M. Ritchie and K. L. Thompson, ***QED Text Editor,*** Computing Science Technical Report No. 5, 1972, Bell Laboratories, Murray Hill, NJ 07974.

6. D. M. Ritchie, private communication. See also M. E. Lesk, ***The Portable C Library,*** Computing Science Technical Report No. 31, Bell Laboratories, Murray Hill, NJ 07974.