LEX

INTRODUCTION

Lex source is a table of regular expressions and corresponding program fragments. The table is translated to a program that reads an input stream, copying it to an output stream and partitioning the input into strings that 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 look ahead 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 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.

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.

Several tools have been built for constructing lexical analyzers from special purpose notations based on regular expression. Lex is widely used tool to specify lexical analyzers for a variety of languages. We refer to the tool as the Lex compiler and to its input specification as the Lex language.

Lex is generally used in the manner depicted in fig. 9.1. First a specification of a lexical analyzer is prepared by creating a program Lex-l in the Lex language. Lex-l is run through the Lex compiler to produce a C program Lex.YY.C. The program Lex.YY.C consists of a tabular representation of a transition diagram constructed from the regular

expressions of lex.l, together with a standard routine that uses the table to recognize LEXEMER. The lexical analyses phase reads the characters in the source program and groups them into a stream of tokens in which each token represents a logically cohesive sequence of characters, such as an identifier, a keyword (if, while, etc.) a punctuation character or a multi-character operator like : = . The character sequence forming a token is called the lexeme for the token. The actions associated with regular expressions in lex - a are pieces of C code and are carried over directly to lex. YY.C. Finally, lex .YY.C is run through the C compiler to produce an object program a.out.

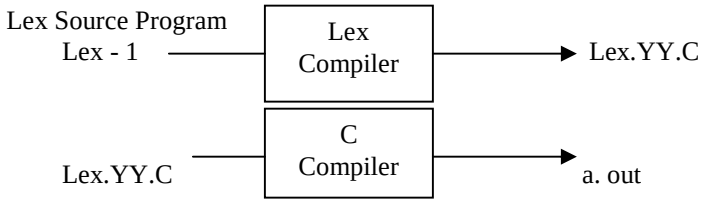

Fig. 9.1 Creating a Lexical Analyzer with Lex

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 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 gas; a way of dealing with this will be described later.

Lex Regular Expressions

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 A123Ba looks for the string A123Ba.

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". 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 automation generated for this source will scan for 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.

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.

Operators: The operator characters are:

```
" \ [ ] ^ - ? . * + | ( ) $ / { } % < >”
```

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.

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 [] must be quoted. Several normal C escapes with \ are recognized: \n is new line, \t is tab, and \b is backspace. To enter \ itself, use \\. Since new line is illegal in an expression, \n must be used; it is not required to escape tab and backspace. Every character but blank, tab, new line 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 ^. 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 ^ 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

[^abc]

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

[^a-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 new line. 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 no character; 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 ^ and $. If the first character of an expression is ^, 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 ^, 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. 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 ^ operator would be equivalent to

<ONE>

Start conditions are explained more fully latter.

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.

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 look ahead 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("Op (=-) 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("Op (=-) 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

=-/[^ \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.

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

```
'[^'\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.

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 beginning 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.

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.

9.5 Examples

1. Write a Lex source program to copy an input file while adding 3 to every positive number divisible by 7.

```
%%
      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".

2. Write a Lex program that 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.

THE LOOK AHEAD OPERATOR

Lexical analyzers for certain programming language constructs need to look ahead beyond the end of a lexeme before they can determine a token with certainty.

Example: Fortran **DO** Statement

```
DO  5  I =  1.25
DO  5  I =  1,25
```

In Fortran, blanks are not significant outside of comments and Hollerith strings;

Hence removing all the blanks in the above statements appears to the lexical analyzer as:

DO5I = 1.25
DO5I = 1,25

In the first statement, we cannot tell until we see the decimal point that the string DO is part of the identifier DO5I. In the second statement, DO is a keyword by itself.

In lex, a pattern of the form r1/r2 can be written. Where

r1 & r2 - regular expressions and / is the operator.

It means a string in r1 is followed by the string r2 with a ' / ' between the two strings indicated the right context for a match. It is used only to restrict a match not to be part of the match.

Example: A lex specification that recognizes the keyword DO in the context above is

DO/({letter} ; {digit }) * = ({ letter} | {digit}) * ,

With this specification, the lexical analyzer will look ahead in its input buffer for a sequence of letters and digits followed by an equal sign followed by letters and digit followed by a comma to be sure that it did not have an assignment statement. Then only the characters D and O, preceding the look ahead operator / would be part of the lexeme that was matched. After a successful match, yytext points to the D and yyleng = 2. Note that this simple look ahead pattern allows DO to be recognized when followed by garbage, like Z4 = 6Q, but it will never recognize DO that is part of an identifier.

Example: **Fortran IF Statement**

The look ahead operator can be used to cope with another difficult lexical analysis problem in Fortran *i.e.,* distinguishing keywords from identifiers. IF (I, J) = 3

It is a perfect Fortran assignment statement, not a logical if statement. One way to specify the keyword IF using Lex is to define its possible right context using the look ahead operator.

Logical if statement is

IF (condition) statement

From of logical - if - statement is

IF (condition) THEN
Then - block
ELSE
else - block
END IF

We note that every unlabeled Fortran statement begins with a letter and that every right parentheses used for subscripting or operand grouping must be followed by an operator symbol such as =, + or comma, another right parentheses or the end of the statement. A letter cannot follow such a right parentheses. In this situation, to confirm that IF is a keyword rather than an array name we scan forward looking for a right

parentheses followed by a letter before seeing a new time character. This pattern for the keyword IF can be written as

IF / \ (. * \ ) {letter}

The . (dot) stands for "any character but new line" and the back slashes in front of the parentheses till Lex to treat them literally, not as meta symbols for grouping in regular expressions.

The IF statements in Fortran can be solved by seeing IF (. to determine whether IF has been declared an array. We scan for the full pattern indicated above if it has been so declared. Such texts makes the automatic implementation of a lexical analyzer from a Lex specification harder, and they may even cost time in the long run, since frequent checks must be made by the program simulating a transition diagram to determine whether any such tests must be made.

Programs: The lex programs are written in a file with a dot extension. Example first.l. The program is executed in the following manner:

% lex < filename.l >

% cc lex.yy.c –o <execfilename> -ll

The lex translates the lex specification into a C source file called lex.yy.c which we compile with the lex library –ll. We then execute the resulting program to check that it works as we expect.

The exeution is as following:

1. If –o <execfilename> is not given run the program by using **./a.out**
2. If –o<execfilename> is given run the program by using **execfilename**.
3. Enter the data after execution.
4. Use **^d** to terminate the program and give result.

There are two programs written for the above requirement. The one is to give the text online after execution. When carriage return is pressed the yylex is executed and the printf statement is carried on displaying on the screen the total number of vowels and consonants.

This program is used to read the given text from a file. The file is opened in the read mode and the yylex function calls the base program to count the number of vowels and consonants. After it cover across the EOF, the C-program is executed to print the number of vowels and consonants.

1. Write a lex program to find the number of vowels and consonants.

```
%{
/*  to find vowels and consonents*/
   int vowels = 0;
   int consonents = 0;
%}
%%
[ \t\n]+
[aeiouAEIOU] vowels++;
[bcdfghjklmnpqrstvwxyzBCDFGHJKLMNPQRSTVWXYZ] consonents++;
```

```
.
%%
main()
{
   yylex();
   printf(" The number of vowels = %d\n", vowels);
   printf(" number of consonents = %d \n", consonents);
return(0);
}
```

The same program can be executed by giving alternative grammar. It is as follows: Here a file is opened which is given as a argument and reads to text and counts the number of vowels and consonants.

```
%{
   unsigned int vowelcount=0;
   unsigned int consocount=0;
%}
vowel [aeiouAEIOU]
consonant [bcdfghjklmnpqrstvwxyzBCDFGHJKLMNPQRSTVWXYZ]
eol \n

%%

{vowel} { vowelcount++;}
{consonant} { consocount++; }

%%
 main(int argc,char *argv[])
{
   if(argc > 1)
  {
    FILE *fp;
    fp=fopen(argv[1],"r");
    if(!fp)
    {
      fprintf(stderr,"could not open %s\n",argv[1]);
      exit(1);
    }
   yyin=fp;
  }
  yylex();
 printf(" vowelcount=%u  consonantcount=%u\n
",vowelcount,consocount);
return(0);
}
```

2. Write a Lex program to count the number of words, characters, blanks and lines in a given text.

```
%{
      unsigned int charcount=0;
      int wordcount=0;
      int linecount=0;
      int blankcount =0;
%}
word[^ \t\n]+
eol \n
%%
[ ] blankcount++;
{word} { wordcount++; charcount+=yyleng;}
{eol} {charcount++; linecount++;}
. { ECHO; charcount++;}
%%
main(argc, argv)
int argc;
char **argv;
{
     if(argc > 1)
     {
      FILE *file;
      file = fopen(argv[1],"r");
      if(!file)
      {
        fprintf(stderr, "could not open %s\n", argv[1]);
        exit(1);
      }
      yyin = file;
        yylex();
        printf("\nThe number of characters = %u\n", charcount);
        printf("The number of wordcount  = %u\n", wordcount);
        printf("The number of linecount  = %u\n", linecount);
        printf("The number of blankcount = %u\n", blankcount);
        return(0);
      }
      else
        printf(" Enter the file name along with the program \n");
}
```

3. Write a lex program to find the number of positive integer, negative integer, positive floating positive number and negative floating point number.

```
%{
      int posnum = 0;
      int negnum = 0;
      int posflo = 0;
```

```
      int negflo = 0;
%}
%%
[\n\t ];
   ([0-9]+) {posnum++;}
   -?([0-9]+) {negnum++; }
   ([0-9]*\.[0-9]+)   { posflo++; }
   -?([0-9]*\.[0-9]+) { negflo++; }
. ECHO;
%%
 main()
 {
      yylex();
       printf("Number of positive numbers = %d\n", posnum);
      printf("number of negative numbers = %d\n", negnum);
      printf("number of floting positive number = %d\n", posflo);
      printf("number of floating negative number = %d\n", negflo);
 }
```

4. Write a lex program to find the given c program has right number of brackets. Count the number of comments. Check for while loop.

```
%{
   /* find main, comments, {, (, ), } */
  int comments=0;
  int opbr=0;
  int clbr=0;
  int opfl=0;
  int clfl=0;
  int j=0;
  int k=0;
%}
%%
"main()" j=1;
"/*"[ \t].*[ \t]"*/" comments++;
"while("[0-9a-zA-Z]*")"[ \t]*\n"{"[ \t]*.*"}"  k=1;
^[ \t]*"{"[ \t]*\n
^[ \t]*"}"                                    k=1;
"(" opbr++;
")" clbr++;
"{"  opfl++;
 "}" clfl++;
[^ \t\n]+
. ECHO;
%%
main(argc, argv)
int argc;
char *argv[];
```

```
{
    if (argc > 1)
    {
      FILE *file;
      file = fopen(argv[1], "r");
      if (!file)
        {
           printf("error opeing a file \n");
           exit(1);
        }
      yyin = file;
     }
     yylex();
     if(opbr != clbr)
         printf("open brackets is not equal to close brackets\n");
     if(opfl != clfl)
         printf("open flower brackets is not equal to close flower
                brackets\n");
         printf(" the number of comments = %d\n",comments);
     if (!j)
         printf("there is no main function \n");
     if (k)
         printf("there is loop\n");
     else printf("there is no valid for loop\n");
     return(0);
}
```

5. Write a lex program to replace scanf with READ and printf with WRITE statement also find the number of scanf and printf.

```
%{
int pc=0,sc=0;
%}
%%
printf fprintf(yyout,"WRITE");pc++;
scanf fprintf(yyout,"READ");sc++;
. ECHO;
%%
main(int argc,char* argv[])
{
  if(argc!=3)
  {
   printf("\nUsage: %s <src> <dest>\n",argv[0]);
   return;
  }
  yyin=fopen(argv[1],"r");
  yyout=fopen(argv[2],"w");
  yylex();
  printf("\nno. of printfs:%d\nno. of scanfs:%d\n",pc,sc);
```

```
}
```

6. Write a lex program to find whether the given expression is valid.

```
%{
   #include <stdio.h>
   int valid=0,ctr=0,oc = 0;
%}
NUM [0-9]+
OP  [+*/-]
%%
{NUM}({OP}{NUM})+ {
                   valid = 1;
                   for(ctr = 0;yytext[ctr];ctr++)
                   {
                     switch(yytext[ctr])
                       {
                          case '+':
                          case '-':
                          case '*':
                          case '/': oc++;
                       }
                    }
                   }
{NUM}\n {printf("\nOnly a number.");}
\n  { if(valid) printf("valid \n operatorcount = %d",oc);
      else  printf("Invalid");
      valid = oc = 0;ctr=0;
    }
%%
main()
{
  yylex();
}

/*    Another solution for the same problem     */

%{
int oprc=0,digc=0,top=-1,flag=0;
char stack[20];
%}
digit [0-9]+
opr [+*/-]
%%
[ \n\t]+
['(']  {stack[++top]='(';}
[')']  {flag=1;
        if(stack[top]=='('&&(top!=-1))
            top--;
```

```
          else
            {
            printf("\n Invalid expression\n");
            exit(0);
            }
         }
{digit}  {digc++;}
{opr}/['('] { oprc++; printf("%s",yytext);}
{opr}/{digit} {oprc++; printf("%s",yytext);}
. {printf("Invalid "); exit(0);}
%%
main()
{
yylex();
if((digc==oprc+1||digc==oprc) && top==-1)
{
printf("VALID");
printf("\n oprc=%d\n digc=%d\n",oprc,digc);
}
else
printf("INVALID");
}
```

7.Write a lex program to find the given sentence is simple or compound.

```
%{
int flag=0;
%}
%%
(" "[aA][nN][dD]" ")|(" "[oO][rR]" ")|(" "[bB][uU][tT]" ") flag=1;
.  ;
%%
main()
{yylex();
if (flag==1)
      printf("COMPOUND SENTENCE \n");
else
      printf("SIMPLE SENTENCE \n");
}
```

8. Write a lex program to find the number of valid identifiers.

```
%{
int count=0;
%}
%%
(" int ")|(" float ")|(" double ")|(" char ")

{
```

```
int ch; ch = input();
for(;;)
   {
       if (ch==',') {count++;}
       else
        if(ch==';') {break;}
           ch = input();
   }
count++;
}
%%
main(int argc,char *argv[])
{
yyin=fopen(argv[1],"r");
yylex();
printf("the no of identifiers used is %d\n",count);
}
```

Exercises

1. How is Lexical analyzer created with Lex?
2. How does Lex behave in concert with aparser?
3. Why should the lex select the longest prefix match pattern?
4. Why should lexical analyzer need to llo ahead to determine a token?
5. Explain how the lexical analyzer will recognize IF statement in FOTRAN?