LAB NO: 5 **Date:**

INTRODUCTION TO FLEX

Objectives:

- To implement programs using a Lexical Analyzer tool called FLEX.
- To apply regular expressions in pattern matching under FLEX.

Prerequisites:

- Knowledge of the C programing language.
- Knowledge of basic level regular expressions

I. INTRODUCTION

FLEX (Fast LEXical analyzer generator) is a tool for generating tokens. Instead of writing a lexical analyzer from scratch, you only need to identify the vocabulary of a certain language, write a specification of patterns using regular expressions (e.g. DIGIT [0-9]), and FLEX will construct a lexical analyzer for you. FLEX is generally used in the manner depicted in Fig. 5.1

Firstly, FLEX reads a specification of a scanner either from an input file *.flex, or from standard input, and it generates as output a C source file lex.yy.c. Then, lex.yy.c is compiled and linked with the flex library (using -lfl) to produce an executable a.out. Finally, a.out analyzes its input stream and transforms it into a sequence of tokens.

- *.l is in the form of pairs of regular expressions and C code.
- lex.yy.c defines a routine yylex() that uses the specification to recognize tokens.
- a.out is actually the scanner.

The various steps involved in generating Lexical Analyzer using Flex is shown in Fig. 5.1

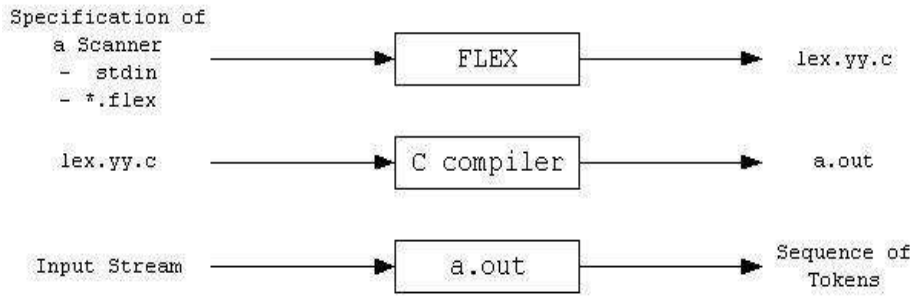

Fig. 5.1 Steps involved in generating Lexical Analyzer using Flex

Regular Expressions and Scanning

Scanners generally work by looking for patterns of characters in the input. For example, in a C program, an integer constant is a string of one or more digits, a variable name is a letter or an underscore followed by zero or more letters, underscores or digits, and the various operators are single characters or pairs of characters. A straightforward way to describe these patterns is regular expressions, often shortened to regex or regexp. A flex program basically consists of a list of regexps with instructions about what to do when the input matches any of them, known as actions. A flex-generated scanner reads through its input, matching the input against all of the regexps and doing the appropriate action on each match. Flex translates all of the regexps into an efficient internal form that lets it match the input against all the patterns simultaneously.

The general format of Flex source program is:

The structure of Flex program has three sections as follows:

%{ definitions%}

%%

rules

%%

user subroutines

Definition section: Declaration of variables and constants can be done in this section. This section introduces any initial C program code we want to get copied into the final program. This is especially important if, for example, we have header files that must be included for code later in the file to work. We surround the C code with the special delimiters "%{"and "%}." Lex copies the material between "%{" and "%}" directly to the generated C file, so we may write any valid C code here. The %% marks the end of this section.

Rule section: Each rule is made up of two parts: a pattern and an action, separated by whitespace. The lexer that lex generates will execute the action when it recognizes the pattern. These patterns are UNIX style regular expressions. Each pattern is at the beginning of a line (since flex considers any line that starts with whitespace to be code to be copied into the generated C program.), followed by the C code to execute when the pattern matches. The C code can be one statement or possibly a multiline block in braces, { }. If more than one rule matches the input, the longer match is taken. If two matches are the same length, the earlier one in the list is taken.

User Subroutines section: This is the final section which consists of any legal C code. This section has functions namely main( ) and yywrap( ).

- The function yylex( ) is defined in lex.yy.c file and is called from main( ). Unless the actions contain explicit return statements, yylex() won't return until it has processed the entire input. The function yywrap( ) is called when EOF is encountered. If this function returns 1, the parsing stops. If the function returns 0, then the scanner continues scanning.

Sample Flex program

```
%{
int chars = 0; int
words = 0; int
lines = 0;
%}


%%
[a-zA-Z]+ { words++; chars += strlen(yytext); }
\n { chars++; lines++; }

. { chars++; }
%%
main(int argc, char **argv)
{
yylex();
 printf("%d%d%d\n", lines, words, chars); }

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

In this program, the definition section contains the declaration for character, word and line counts. The rule section consists of only three patterns. The first one, [a-zA-Z]+, matches a word. The characters in brackets, known as a character class, match any single upper- or lowercase letter, and the + sign means to match one or more of the preceding thing, which here means a string of letters or a word. The action code updates the number of words and characters seen. In any flex action, the variable yytext is set to point to the input text that the pattern just matched. The second pattern, \n, just matches a new line. The action updates the number of lines and characters. The final pattern is a dot, which is regex that matches any character. The action updates the number of characters. The end of the rules section is delimited by another %%.

Handling ambiguous patterns

Most flex programs are quite ambiguous, with multiple patterns that can match the same input. Flex resolves the ambiguity with two simple rules:

- Match the longest possible string every time the scanner matches input.
- In the case of a tie, use the pattern that appears first in the program.

These turn out to do the right thing in the vast majority of cases. Consider this snippet from a scanner for C source code: "+" { return ADD; }
"=" { return ASSIGN; }
"+=" { return ASSIGNADD; }
"if" { return KEYWORDIF; }
"else" { return KEYWORDELSE; }
[a-zA-Z_][a-zA-Z0-9_]* { return IDENTIFIER; }

Table 5.1 Variables and functions available by default in Flex

yytext	When the lexical analyzer matches or recognizes the token from the input, then the lexeme is stored in a null terminated string called *yytext*. It is an array of pointer to char where *lex* places the current token's lexeme. The string is automatically null terminated.
yyleng	Stores the length or number of characters in the input string. The value of yyleng is same as strlen( ) functions. In other words it is a integer that holds strlen(yytext).
yyval	This variables returns the value associated with token.
yyin	Points to the input file.
yyout	Points to the output file.
yylex( )	The function that starts the analysis process. It is automatically generated by Lex.
yywrap ( )	This function is called when EOF is encountered. If this function returns 1, the parsing stops. If the function returns 0, then the scanner continues scanning.

For the first three patterns, the string += is matched as one token, since += is longer than +. For the last three patterns, as long as the patterns for keywords precede the pattern that matches an identifier, the scanner will match keywords correctly. A list of various variables and functions available in Flex are given in Table 5.1

Table 5.2 Regular Definitions in FLEX

Reg Exp	**Description**
x	Matches the character x.
[xyz]	Any characters amongst x, y or z. You may use a dash for character intervals: [a-z] denotes any letter from a through z. You may use a leading hat to negate the class: [0-9] stands for any character which is not a decimal digit, including new-line.
"string"	"..." Anything within the quotation marks is treated literally
<<EOF>>	Match the end-of-file.
[a,b,c]	matches a, b or c
[a-f]	matches either a,b,c,d,e, or f in the range a to f
[0-9]	matches any digit
X+	matches one or more occurrences of X
X*	matches zero or more occurrences of X
[0-9]+	matches any integer
( )	grouping an expression into a single unit
|	alternation (or)
(a | b | c) *	equivalent to [a-c]*

X?	X is optional (zero or one occurrence)
[A-Za-z]	matches any alphabetical character
.	matches any character except newline
\ .	matches the **.** character
\n	matches the newline character.
\t	matches the tab character
[^a-d]	matches any character other than a,b,c and d.

The basic operators to make more complex regular expressions are, with r and s being two regular expressions:

- **(r)** : Match an r; parentheses are used to override precedence.
- **rs** : Match the regular expression r followed by the regular expression s. This is called concatenation.
- **r|s** : Match either an r or an s. This is called alternation.
- **{abbreviation}**: Match the expansion of the abbreviation definition. Instead of writing regular expression.

Example for abbreviation:

```
%%
[a-zA-Z_][a-zA-Z0-9_]*   return IDENTIFIER;
%%
```

We may write id [a-zA-Z_][a-zA-Z0-9_]*

```
%%
{id}   return IDENTIFIER;
%%
```

- **r/s:** Match an r but only if it is followed by an s. The text matched by s is included when determining whether this rule is the longest match, but is then returned to the input before the action is executed. So the action only sees the text matched by r. This type of pattern is called trailing context.
- **^r :** Match an r, but only at the beginning of a line (i.e., which just starting to scan, or right after a newline has been scanned).
- **r$:** Match an r, but only at the end of a line (i.e., just before a newline).

1. SOLVED EXERCISE:

Write a Flex program to recognize identifiers.

Program:

```
{% #include<stdio.h>
%}
%%
[a-zA-Z_][a-zA-Z0-9_]*   printf(" Identifier");
%%
int yywrap() { return 1;}
int main()  {
char stat[20];
printf("Enter the valid C statement");
scanf("%s,stat);
}
```

Installing FLEX:

Steps to download, compile, and install are as follows. Note: Replace 2.5.33 with your version number:

Downloading Flex (The fast lexical analyzer):

- **Run the command below,**

  wget http://prdownloads.sourceforge.net/flex/flex 2.5.33.tar.gz?download
- **Extracting files from the downloaded package:**

  tar -xvzf flex-2.5.33.tar.gz
- **Now, enter the directory where the package is extracted.**

  cd flex-2.5.33

- **Configuring flex before installation:**

  If you haven't installed m4 yet then please do so. Click here to read about the installation instructions for m4. Run the commands below to include m4 in your PATH variable.

```
PATH=$PATH:/usr/local/m4/bin/
```

NOTE: Replace '/usr/local/m4/bin' with the location of m4 binary. Now, configure the source code before installation.

```
./configure --prefix=/usr/local/flex
```

Replace "/usr/local/flex" above with the directory path where you want to copy the files and folders.

Note: check for any error message.

Compiling flex:

```
make
```

Note: check for any error message.

Installing flex:

As root (for privileges on destination directory), run the following.

With sudo,

```
sudo make install
```

Without sudo,

```
make install
```

Note: check for any error messages.

Flex has been successfully installed.

Steps to execute:

a. Type Flex program and save it using .l extension.
b. Compile the flex code using
   $ flex filename.l
c. Compile the generated C file using
   $ gcc lex.yy.c - o output
d. This gives an executable output
e. Run the executable using **$ ./output**

II. LAB EXERCISES

Write a FLEX program to

1. Count the number of vowels and consonants in the given input.
2. Count the number of words, characters, blanks and lines in a given text.
3. Find the number of positive integer, negative integer, positive floating positive number and negative floating point number
4. Given a input C file, replace all scanf with READ and printf with WRITE statements also find the number of scanf and printf in the file.
5. That changes a number from decimal to hexadecimal notation.
6. Convert uppercase characters to lowercase characters of C file excluding the characters present in the comment.

III. ADDITONAL EXERCISES:

1. Generate tokens for a simple C program. (Tokens to be considered are: Keywords, Identifiers, Special Symbols, arithmetic operators and logical operators)
2. Write a FLEX program to identify verb, noun and pronoun.