**Practical - 2**

**Aim:**

**(a) Introduction to FLEX.**

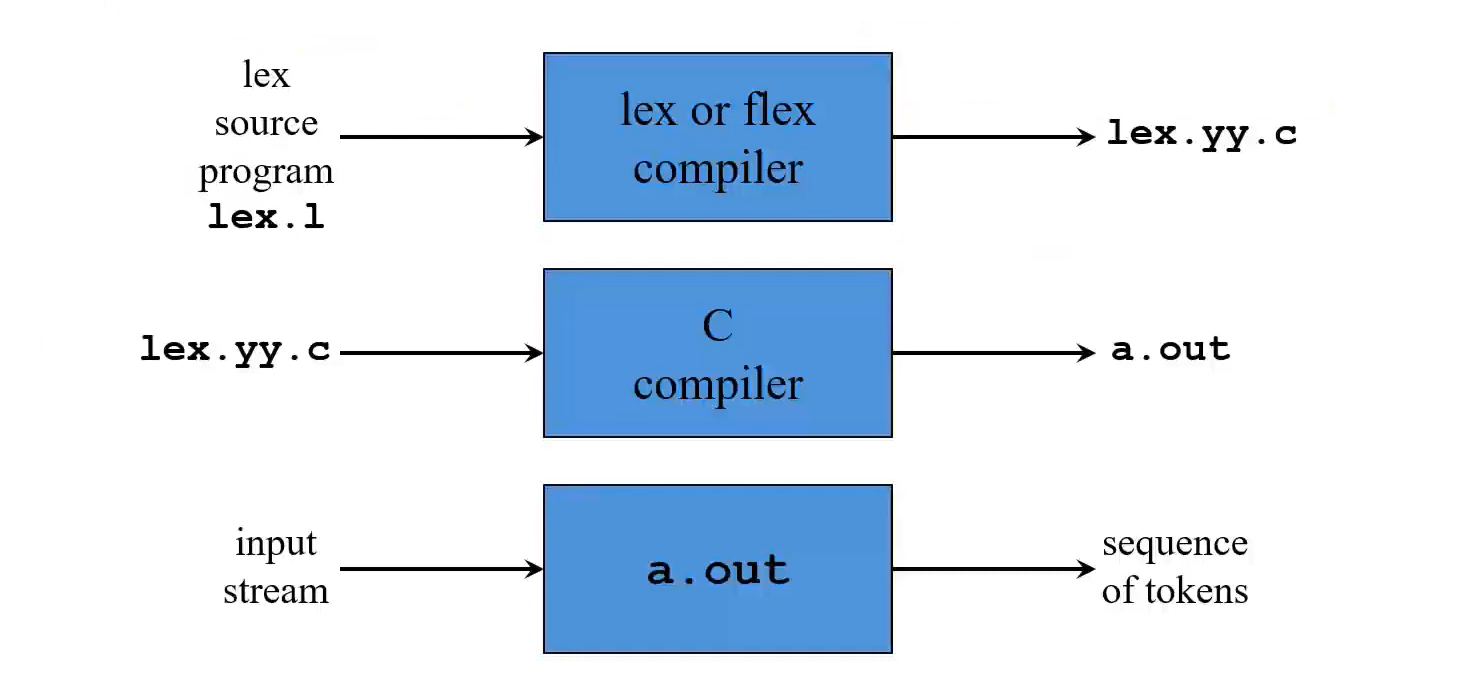
**(b) Write a FLEX program to print name when an ENTER key is pressed.**

**(c) Write a FLEX program to check whether the given word starts with vowel or consonant.**

**Solution (a):**

FLEX is the newer opensource cousin of LEX which was a previous generation scanner generator or Lexical analyser.

FLEX is used to Systematically translate regular definitions into C source code for efficient scanning. And the generated code is easier to integrate into C applications.



**First some simple examples to get the flavor of how one uses flex.**

The following flex input specifies a scanner which whenever it encounters the string "username" will replace it with the user's login name:

%%

username printf( "%s", getlogin() );

By default, any text not matched by a flex scanner is copied to the

output, so the net effect of this scanner is to copy its input file to its output with each occurrence of "username" expanded. In this input, there is just one rule. "username" is the pattern and the "printf" is the action. The "%%" marks the beginning of the rules.

**Here's another simple example:**

int num\_lines = 0, num\_chars = 0;

%%

\n ++num\_lines; ++num\_chars;

. ++num\_chars;

%%

void main() {

{

yylex();

printf( "# of lines = %d, # of chars = %d\n",

num\_lines, num\_chars );

}

}

This scanner counts the number of characters and the number of lines in

its input (it produces no output other than the final report on the

counts). The first line declares two globals, "num\_lines" and

"num\_chars", which are accessible both inside yylex() and in the main()

routine declared after the second "%%". There are two rules, one which

matches a newline ("\n") and increments both the line count and the

character count, and one which matches any character other than a newline

(indicated by the "." regular expression).

**A somewhat more complicated example:**

/\* scanner for a toy Pascal-like language \*/

%{

#include <math.h>

%}

DIGIT [0-9]

ID [a-z][a-z0-9]\*

%%

{DIGIT}+ {

printf( "An integer: %s (%d)\n", yytext,

atoi( yytext ) );

}

{DIGIT}+"."{DIGIT}\* {

printf( "A float: %s (%g)\n", yytext,

atof( yytext ) );

}

if|then|begin|end|procedure|function {

printf( "A keyword: %s\n", yytext );

}

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

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

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

[ \t\n]+ /\* eat up whitespace \*/

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

%%

int main( int argc, char \*\*argv )

{

++argv, --argc; /\* skip over program name \*/

if ( argc > 0 )

yyin = fopen( argv[0], "r" );

else

yyin = stdin;

yylex();

}

This is the beginnings of a simple scanner for a language like Pascal. It identifies different types of "tokens" and reports on what it has seen.

**Structure of a Flex program**

%{

Definition Section

%}

%%

Rules Section

%%

User Code

**Definition Section**

The "definitions section" contains declarations of simple "name" definitions to simplify the scanner specification, and declarations of "start conditions", which are explained in a later section.

Name definitions have the form:

name definition

The 'name' is a word beginning with a letter or an underscore ('\_')

followed by zero or more letters, digits, '\_', or '-' (dash). The

definition is taken to begin at the first non-whitespace character

following the name and continuing to the end of the line. The

definition can subsequently be referred to using '{name}', which will

expand to '(definition)'. For example,

DIGIT [0-9]

ID [a-z][a-z0-9]\*

**Rules Section**

The "rules" section of the 'flex' input contains a series of rules of

the form:

pattern action

where the pattern must be unindented and the action must begin on the

same line.

In the rules section, any indented or %{ %} enclosed text appearing

before the first rule may be used to declare variables which are local

to the scanning routine and (after the declarations) code which is to be

executed whenever the scanning routine is entered. Other indented or %{

%} text in the rule section is still copied to the output, but its

meaning is not well-defined and it may well cause compile-time errors.

**User Defined Section(Auxiliary Section)**

The user code section is simply copied to 'lex.yy.c'. It is

used for companion routines which call or are called by the scanner.

The presence of this section is optional; if it is missing, the second

'%%' in the input file may be skipped, too.

**(2b) Write a FLEX program to print name when an ENTER key is pressed.**

**Code:**

%%

[\n] printf("Hello Utsav\n");

%%

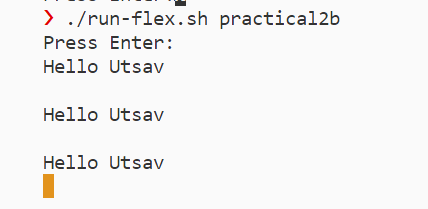
void main() {

printf("Press Enter:");

yylex();

}

**Output:**

****

**(2c): Write a FLEX program to check whether the given word starts with vowel or consonant.**

**Code:**

%%

^[a|e|i|o|u|A|E|I|O|U][a-zA-Z]\* {ECHO; printf(": is vowel\n");}

.\* {ECHO; printf(": is consonant\n");}

%%

void main() {

printf("Enter a word or a letter: ");

yylex();

}

**Output:**

**Graphical user interface, text, application

Description automatically generated**