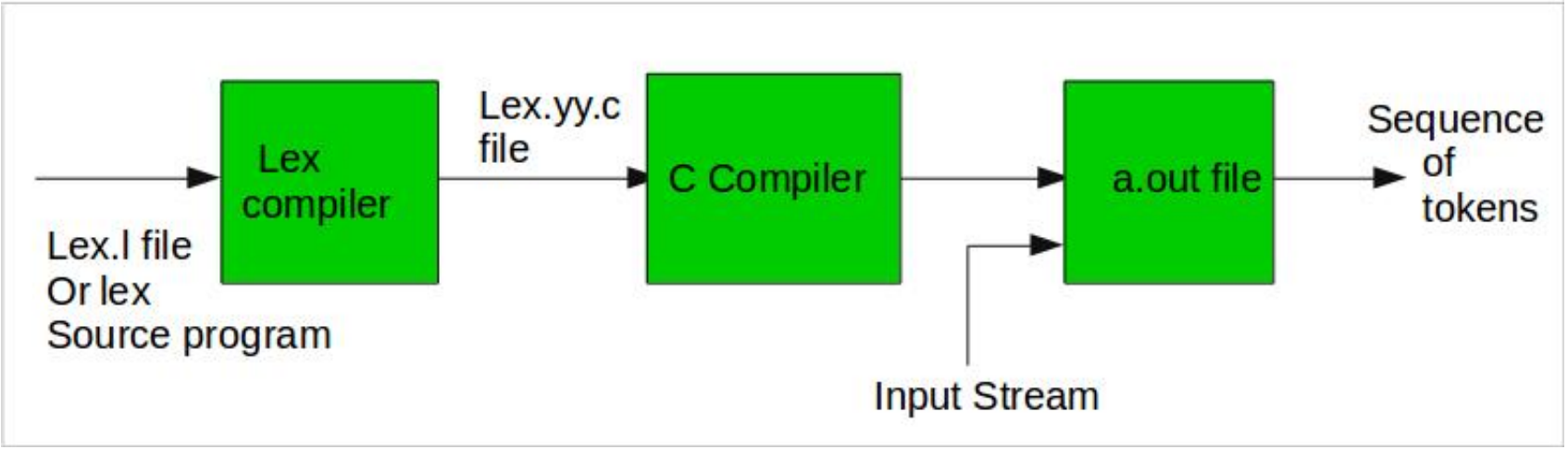
Experiment 7

Aim: Implementation of Lexical Analyzer using Lex Tool.
The lexical analyzer must do the following.

- Open a file containing a C program and read from it.
- Recognize the pre-processor directives, punctuators, strings and comments in the C code.
- Accept the usual keywords in a C program.
- Recognize identifiers and integer numbers.
- Recognize arithmetic, assignment and relational operators in C

Using flex tool



1. An input file describes the lexical analyzer to be generated named lex.l is written in lex language. The lex compiler transforms lex.l to C program, in a file that is always named lex.yy.c.
2. The C complier compile lex.yy.c file into an executable file called a.out.
3. The output file a.out take a stream of input characters and produce a stream of tokens.

Lex Program Structure: The three sections

```
declarations
%%
translation rules
%%
auxiliary procedures
```

1. The declarations section includes declarations of variables, manifest constants (A manifest constant is an identifier that is declared to represent a constant. e.g. # define PIE 3.14), and regular definitions.

Lex Program Structure: The three sections

2. The translation rules of a Lex program are statements of the form :
   p1 action 1
   p2 action 2
   p3 action 3
   ... ...
   ... ...
   where each p is a regular expression and each action is a program fragment describing what action the lexical analyzer should take when a pattern p matches a lexeme. In Lex the actions are written in C.
3. The third section holds whatever auxiliary procedures are needed by the actions.

First section - declarations

```
preprocessor #.*
identifier [_a-zA-Z][_a-zA-Z0-9]*
keyword int|float|char|double|while\(.*\)|for\(.*\)
|do|if\(.*\)|break|continue|void|switch\(.*\)
|case|long|struct|const|typedef|return|else|goto
comment "/*".*"*/"
asmnt_op "="|"+="|"-="|"/="|"*="
relop "=="|"!="|"<"|"<="|">"|">="
arith_op [+\-*/%]
integer [-+]?([0-9]*)
string \".*\"
punct [;\.,]
%%
```

Second Section - translation rules

```
{string} {printf("\n String:\t%s ",yytext);}
{keyword} {printf("\n Keyword:\t%s",yytext);}
{preprocessor} {printf("\n Preprocessor directive:\t%s",yyt
{comment} {printf("\n Comment:\t %s",yytext);}
{identifier}\( {printf("\n Function:\t%s",yytext);}
\{  {printf("\n begin block");}
\}  {printf("\n end block ");}
{relop} {printf("\nRelational Operator:\t%s ",yytext);}
{asmnt_op} {printf("\n Assignment Operator:\t%s ",yytext);}
{arith_op} {printf("\n Arithmetic Operator:\t%s ",yytext);}
{identifier}: {printf("\n Label:\t%s ",yytext);}
{identifier} {printf("\n Variable:\t%s ",yytext);}
{integer} {printf("\n Integer Number:\t%s ",yytext);}
\) {printf("\n End of function\t%s ",yytext);}
{punct} {printf("\n Punctuation:\t%s ",yytext);}
%%
```

Third Section - auxiliary procedures

```
int main(int argc, char **argv)
{
  FILE *file;
  file=fopen("input.c","r");
  if(!file)
  {
    printf("File open error");
    exit(0);
  }
  yyin=file;
  yylex();
  printf("\n");
  return(0);
}
int yywrap()
{
  return 1;
}
```