Experiment 1

1.1: AIM: Write a lex program whose output is same as input.

DESCRIPTION:

Lex is a program that generates lexical analyzer. It is used with YACC parser generator.The lexical analyzer is a program that transforms an input stream into a sequence of tokens.It reads the input stream and produces the source code as output through implementing the lexical analyzer in the C program.

A Lex program is separated into three sections by %% delimiters. The formal of Lex source is as follows:

{ definitions }

%%

{ rules }

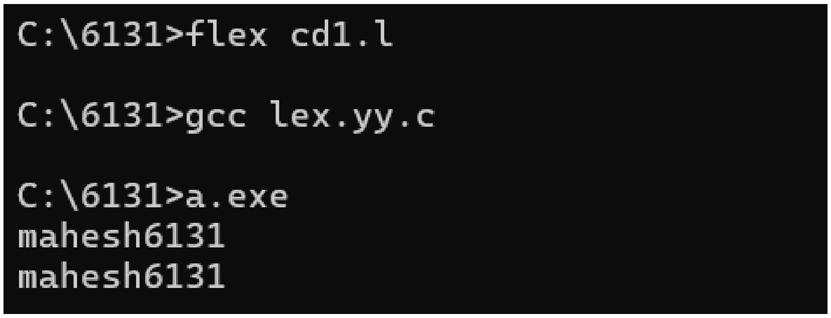
%%

{ user subroutines }

CODE:

```
%%
. ECHO;
%%
int yywrap(void) {
return 1;
}
int main(void) {
yylex();return 0;

}
```

OUTPUT:



```
C:\6131>flex cd1.l

C:\6131>gcc lex.yy.c

C:\6131>a.exe
mahesh6131
mahesh6131
```

1.2 **AIM:** Write a lex program which removes white spaces from its input file.

DESCRIPTION: Lex is a program that generates lexical analyzer. It is used with YACC parser generator. The lexical analyzer is a program that transforms an input stream into a sequence of tokens. It reads the input stream and produces the source code as output through implementing the lexical analyzer in the C program.

The function of Lex is as follows:

1. Firstly lexical analyzer creates a program lex.1 in the Lex language. Then Lex compiler runs the lex.1 program and produces a C program lex.yy.c.

2. Finally C compiler runs the lex.yy.c program and produces an object program a.out.

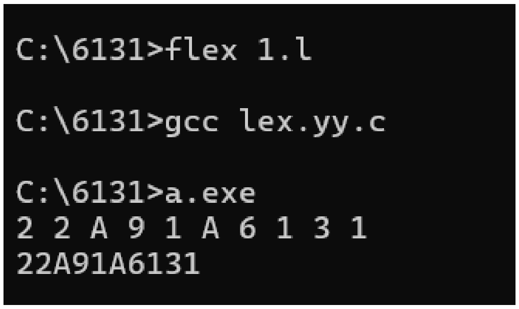
3. a.out is lexical analyzer that transforms an input stream into a sequence of tokens.

Definitions include declarations of constant, variable and regular definitions.Rules define the statement of form p1 {action1} p2 {action2}. .. pn {action}.Where pi describes the regular expression and action1 describes the actions what action the lexical analyzer should take when pattern pi matches a lexeme.User subroutines are auxiliary procedures needed by the actions. The subroutine can be loaded with the lexical analyzer and compiled separately.

CODE:

```
%%
[ ] {};
. ECHO;
%%
int yywrap(void) {
   return 1;}
int main(void) {
yylex();return 0;
}
```

OUTPUT:



```
C:\6131>flex 1.l

C:\6131>gcc lex.yy.c

C:\6131>a.exe
2 2 A 9 1 A 6 1 3 1
22A91A6131
```

Experiment 2

2.1 AIM: Write a lex program to identify the patterns in the input file.

DESCRIPTION:

%{ and %} sections: These sections are optional and can be used to declare variables or include header files needed for your program.

%% delimiters: These delimit the beginning and end of the pattern rules section.

Pattern rules:

- [\t\n]+: Matches one or more whitespace characters (spaces, tabs, and newlines). You can choose to do nothing or print a message for these.
- [a-zA-Z][a-zA-Z0-9]*: Matches identifiers that start with a letter (uppercase or lowercase) and can be followed by any number of letters or digits. You can print or store the identifier.
- [0-9]+: Matches one or more digits, representing integers. You can print or store the integer.

CODE:

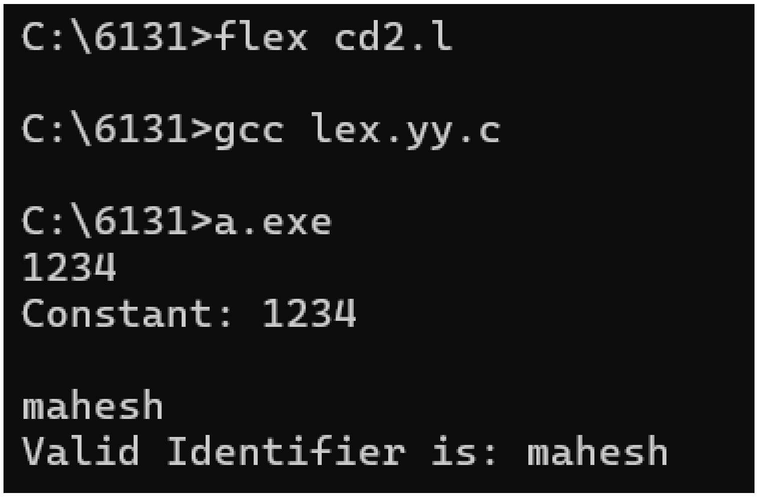
```
%{
#include<stdio.h>
%}
%%
["int""char""for""if""while""then""return""do"] {printf("keyword : %s\n");}
[*%+\-] {printf("Operator : %s ", yytext);}
[(){};] {printf("Special Character: %s\n", yytext);}
[0-9]+ {printf("Constant : %s\n", yytext);}
[a-zA-Z_][a-zA-Z0-9_]*  {printf("Valid Identifier is : %s\n", yytext);}
^[^a-zA-Z_] {printf("Invalid Indentifier \n");}
%%

int yywrap() {
   return 1; // Indicate end of input
}
```

```
int main(void) {
yylex();
    return 0;
}
```

input.txt

```
=======
12345
Hello
world
123abc456
!@#$%
This is a test file.
```

OUTPUT :



```
C:\6131>flex cd2.l

C:\6131>gcc lex.yy.c

C:\6131>a.exe
1234
Constant: 1234

mahesh
Valid Identifier is: mahesh
```

2.2 : AIM: Design a lexical analyzer for given language and the lexical analyzer should ignore redundant spaces, tabsand new lines.

DESCRIPTION:

Tools like Lex (often used with C) or JFlex (used with Java) can simplify the development of lexical analyzers by providing a framework for defining states, transitions, and token types. These tools generate code that handles the low-level details of character streams and state management.

Components:

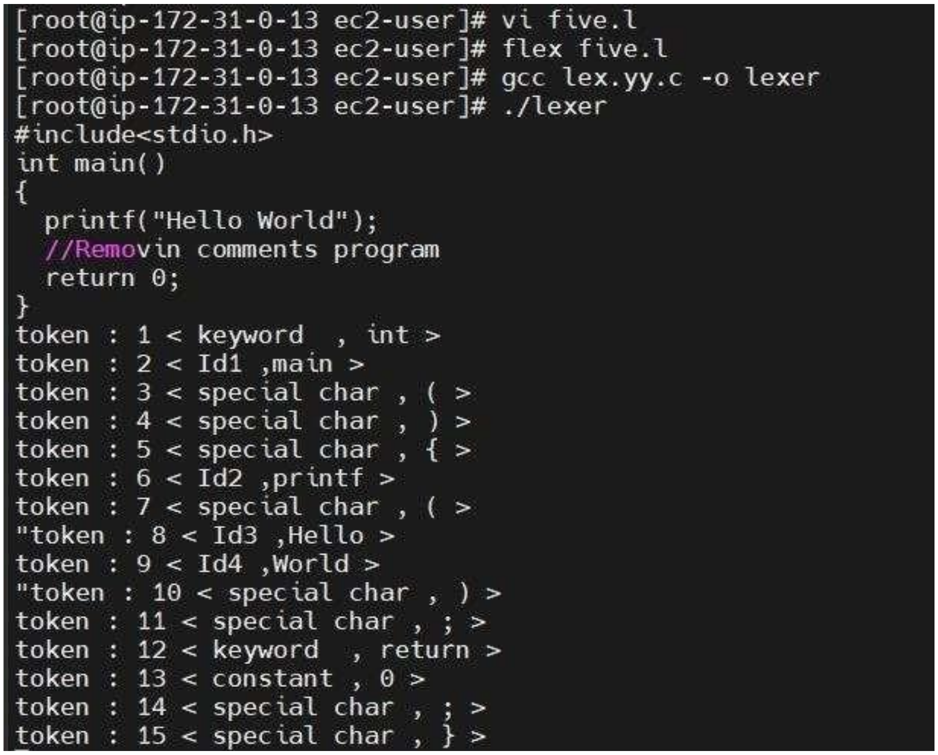
- Character Stream: The input for the lexical analyzer is a stream of characters from the source code file.
- Finite State Automata: The core component, FSA is a machine with a set of states and transitions between them. The transitions are triggered by the next character in the input stream.
- Transition Function: This function determines the next state to move to based on the current state and the next character in the input stream.
- Tokens: The output of the lexical analyzer is a sequence of tokens. Each token represents a specific category (type) like identifier, integer, keyword, operator, etc., along with the actual lexeme (the matched character sequence).

CODE:

```
%{
#include<stdio.h>
int i=0,id=0;
%}


%%
[#].*[<].*[>]\n {}
[ \t\n]+ {}
\/\/.*\n {}
\/\*(.*\n)*.*\*\/  {}
auto|break|case|char|const|continue|default|do|double|else|enum|extern|float|for|goto|if|int|long|register|return|short|signed|sizeof|static|struct|switch|typedef|union|unsigned|void|volatile|while  {printf("token : %d < keyword  , %s >\n",++i,yytext);}
[+\-\*\/%<>] {printf("token : %d < operator , %s >\n",++i,yytext);}
```

```
[();{}] {printf("token : %d < special char , %s >\n",++i,yytext);}

[0-9]+ {printf("token : %d < constant , %s >\n",++i,yytext);}

[a-zA-Z_][a-zA-Z0-9_]* {printf("token : %d <Id%d ,%s >\n",++i,++id,yytext);}

^[^a-zA-Z_] {printf("ERROR Invaild token %s \n",yytext);}

%%


int yywrap() {

    return 1; // Indicate end of input

}

int main(void) {

yylex();

    return 0;

}
```

OUTPUT :



```
[root@ip-172-31-0-13 ec2-user]# vi five.l
[root@ip-172-31-0-13 ec2-user]# flex five.l
[root@ip-172-31-0-13 ec2-user]# gcc lex.yy.c -o lexer
[root@ip-172-31-0-13 ec2-user]# ./lexer
#include<stdio.h>
int main()
{
  printf("Hello World");
  //Removin comments program
  return 0;
}
token : 1 < keyword  , int >
token : 2 < Id1 ,main >
token : 3 < special char , ( >
token : 4 < special char , ) >
token : 5 < special char , { >
token : 6 < Id2 ,printf >
token : 7 < special char , ( >
"token : 8 < Id3 ,Hello >
token : 9 < Id4 ,World >
"token : 10 < special char , ) >
token : 11 < special char , ; >
token : 12 < keyword  , return >
token : 13 < constant , 0 >
token : 14 < special char , ; >
token : 15 < special char , } >
```