**Practical 1(A)**

**Aim:**

**1(A): Implement Lexical Analyzer in Lex Tool for the input file given below:**

Input C file :-

#include<stdio.h>

#include<conio.h>

void main()

{

int a,b,c;

c=a+b;

printf("Sum:%d",c);

}

**Expected Output:**

#include<stdio.h> is a preprocessor directive

#include<conio.h> is a preprocessor directive

Void is a keyword

Main() is a function

Int is a keyword

a is an identifier.

b is an identifier.

c is an identifier.

= assignment operator

“ “ is a string

Printf is a keyword

, is a punctuation mark

; is a punctuation mark

**Theory:**

**Structure of Lex Programs**

Lex program will be in following form

declarations

%%

translation rules

%%

auxiliary functions

***Declarations***This section includes declaration of variables, constants and regular definitions.

***Translation rules***It contains regular expressions and code segments.

Form : Pattern {Action}

Pattern is a regular expression or regular definition.

Action refers to segments of code.

***Auxiliary functions:*** this section holds additional functions which are used in actions. These functions are compiled separately and loaded with lexical analyzer.

Lexical analyzer produced by lex starts its process by reading one character at a time until a valid match for a pattern is found.

Once a match is found, the associated action takes place to produce token.

The token is then given to parser for further processing.

**Conflict Resolution in Lex**

Conflict arises when several prefixes of input matches one or more patterns. This can be resolved by the following:

* Always prefer a longer prefix than a shorter prefix.
* If two or more patterns are matched for the longest prefix, then the first pattern listed in lex program is preferred.

**Lookahead Operator**

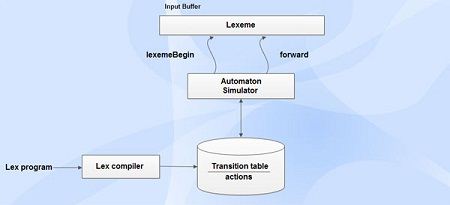
* Lookahead operator is the additional operator that is read by lex in order to distinguish additional pattern for a token.
* Lexical analyzer is used to read one character ahead of valid lexeme and then retracts to produce token.
* At times, it is needed to have certain characters at the end of input to match with a pattern. In such cases, slash (/) is used to indicate end of part of pattern that matches the lexeme.

**Design of Lexical Analyzer**

* Lexical analyzer can either be generated by NFA or by DFA.
* DFA is preferable in the implementation of lex.

**Structure of Generated Analyzer**

Architecture of lexical analyzer generated by lex is given in Fig.

[](http://ecomputernotes.com/images/Lex-program-used-by-finite-automaton-simulator.jpg)

Lexical analyzer program includes:

* Program to simulate automata
* Components created from lex program by lex itself which are listed as follows:
  + A transition table for automaton.
  + Functions that are passed directly through lex to the output.
  + Actions from input program (fragments of code) which are invoked by automaton simulator when needed.

**Source Code :**

**Program Code :**

**C Code :**

#include<stdio.h>;

#include<conio.h>;

void main()

{

int a,b,c;

c=a+b;

printf("Sum:%d",c);

}

**Lex Code :**

%{

#include<stdio.h>

%}

%%

#.\* {printf("\n\t%s is a preprocessor directive.",yytext);}

void|int|float|char|double|while|for|if|else|printf {printf("\n\t%s is a keyword.",yytext);}

main.\* {printf("\n\t%s is a function.",yytext);}

[a-zA-Z0-9]\* {printf("\n\t%s is an identifier.",yytext);}

\".\*\" {printf("\n\t%s is a string",yytext);}

["{}()"]

"=" {printf("\n\t%s is a assignment operator.",yytext);}

\+ {printf("\n\t%s is a arithmetic operator.",yytext);}

\; {printf("\n\t%s is a punctuation mark.",yytext);}

\, {printf("\n\t%s is a punctuation mark.",yytext);}

%%

int main(void)

{

yyin=fopen("Practical-1a.c","r");

yylex();

}

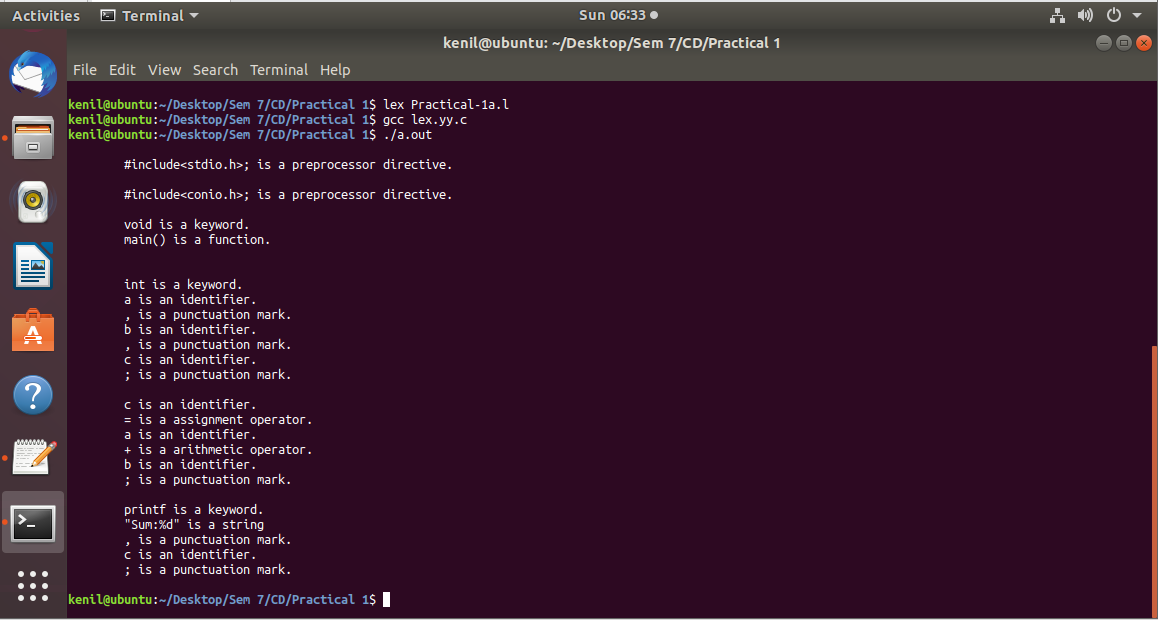
int yywrap()

{

return(1);

}

**OUTPUT :**

****