**Practical: 1**

**Aim:** Implementation of Finite Automata and String Validation.

**Code:**

#include <stdio.h>

#include <string.h>

int main() {

int length;

printf("Enter length of the string: ");

scanf("%d", &length);

char str[length + 1]; // +1 to accommodate the null terminator

printf("Enter a string to be checked: ");

scanf("%s", str);

int flag = 0;

// Check if the string contains '0' or '1'

for (int i = 0; str[i] != '\0'; i++) {

if (str[i] == '0' || str[i] == '1') {

flag = 1;

break;

}

}

// Check the last two characters if the flag is set

if (flag == 1) {

if (str[length - 1] == '1' && str[length - 2] == '0') {

printf("String is accepted\n");

} else {

printf("String is not accepted\n");

}

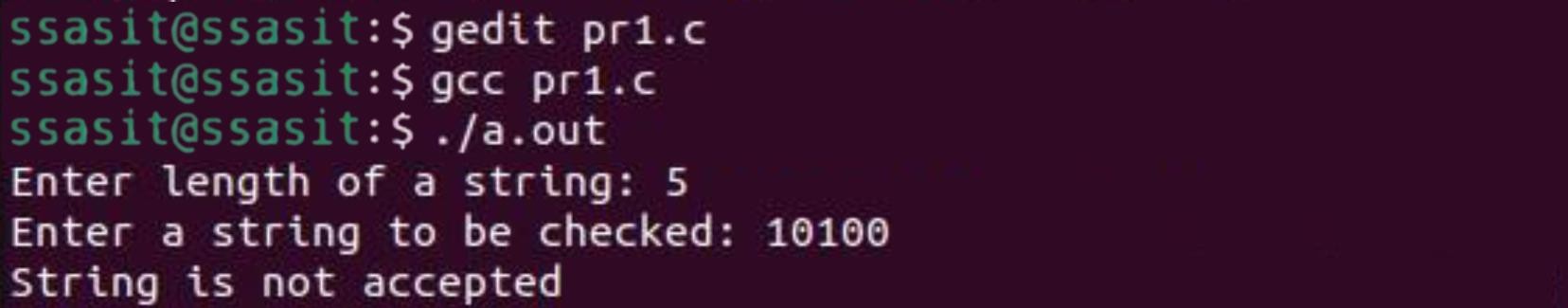
} else {

printf("String is not accepted\n");

}

return 0;

}

**Output :**

**Practical: 2**

**Aim:** Introduction to Lex & Flex Tool.

* **Lex:**

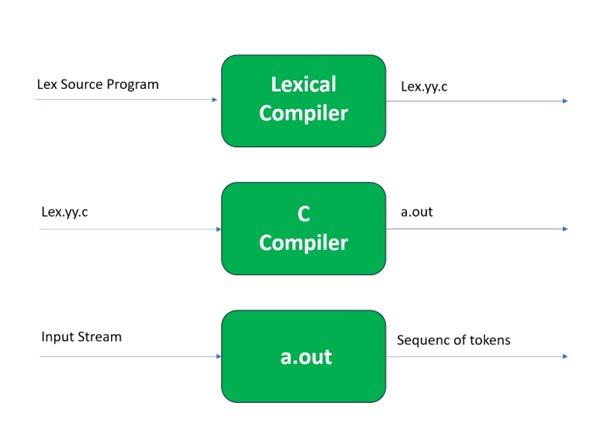
Lex is a tool or a computer program that generates Lexical Analyzers (converts the stream of characters into tokens). The Lex tool itself is a compiler. The Lex compiler takes the input and transforms that input into input patterns. It is commonly used with YACC (Yet Another Compiler Compiler). It was written by Mike Lesk and Eric Schmidt.

**Function of Lex:**

1. In the first step the source code which is in the Lex language having the file name ‘File.l’ gives as input to the Lex Compiler commonly known as Lex to get the output as lex.yy.c.

2. After that, the output lex.yy.c will be used as input to the C compiler which gives the output in the form of an ‘a.out’ file, and finally, the output file a.out will take the stream of character and generates tokens as output.

lex.yy.c: It is a C program.  
File.l: It is a Lex source program  
a.out: It is a Lexical analyzer



**Lex File Format:**

A Lex program consists of three parts and is separated by %% delimiters:-

Declarations  
%%  
Translation rules  
%%  
Auxiliary procedures

**Declarations:**The declarations include declarations of variables.

**Transition rules:** These rules consist of Pattern and Action.

**Auxiliary procedures:** The Auxilary section holds auxiliary functions used in the actions.

For example:

**declaration**  
number[0-9]  
%%  
**translation**  
if {return (IF);}  
%%  
**auxiliary function**  
int numberSum()

# **Flex:**

**FLEX (fast lexical analyzer generator)** is a tool/computer program for generating lexical analyzers

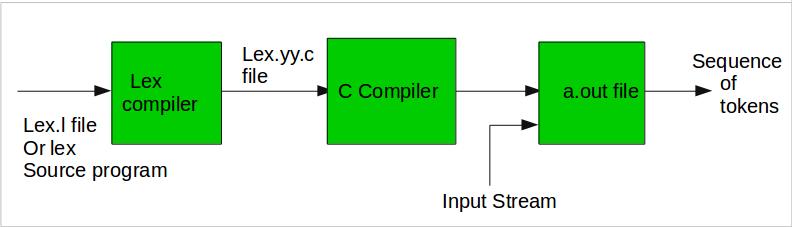
written by Vern Paxson in C around 1987. Flex and Bison both are more flexible than Lex and Yacc and produces faster code.   
Bison produces parser from the input file provided by the user. The function **yylex()** is automatically generated by the flex when it is provided with a **.l file** and this yylex() function is expected by parser to call to retrieve tokens from current/this token stream.

**Installing Flex on Ubuntu:**

sudo apt-get update

sudo apt-get install flex

Given image describes how the Flex is used:



**Step 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.

**Step 2:** The C compiler compile lex.yy.c file into an executable file called a.out.

**Step 3:** The output file a.out take a stream of input characters and produce a stream of tokens.

**Program Structure:**

**In the input file, there are 3 sections:**

**1. Definition Section:** The definition section contains the declaration of variables, regular definitions, manifest constants. In the definition section, text is enclosed in “%{ %}” brackets. Anything written in this brackets is copied directly to the file lex.yy.c

**Syntax:**

%{

// Definitions

%}

**2. Rules Section:** The rules section contains a series of rules in the form: *pattern action* and pattern must be unintended and action begin on the same line in {} brackets. The rule section is enclosed in **“%% %%”**.

**Syntax:**

%%

pattern action

%%

**3. User Code Section:** This section contains C statements and additional functions. We can also compile these functions separately and load with the lexical analyzer.

Basic Program Structure:

%{

// Definitions

%}

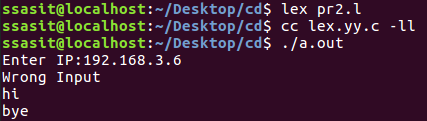
%%

Rules

%%

User code section

**Output:**



**Practical: 3**

**Aim:** Implementation following Programs Using Lex.

a. Generate Histogram of words

b.Check Cypher

c. Extract single and multiline comments from C Program

1. **Generate Histogram of words**

%{

#include<stdio.h>

#include<string.h>

int i = 0;

%}

/\* Rules Section\*/

%%

([a-zA-Z0-9])\* {i++;} /\* Rule for counting

number of words\*/

"\n" {printf("%d\n", i); i = 0;}

%%

int yywrap(void){

int main()

{

// The function that starts the analysis

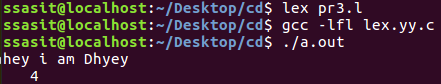
yylex();

return 0;

}

}

**Output:**



**b.Check Cypher**

%{

#include <stdio.h>

#include <stdlib.h>

void encrypt(char \*text, int shift);

%}

%option noyywrap

%%

[a-zA-Z]+ {

encrypt(yytext, 3); // Shift by 3 positions for Caesar cipher

}

.|\n {

putchar(yytext[0]);

}

%%

void encrypt(char \*text, int shift) {

while (\*text) {

char c = \*text;

if (c >= 'a' && c <= 'z') {

c = 'a' + (c - 'a' + shift) % 26;

} else if (c >= 'A' && c <= 'Z') {

c = 'A' + (c - 'A' + shift) % 26;

}

putchar(c);

text++;

}

}

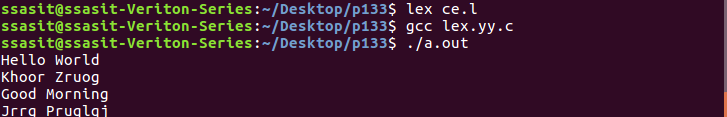
int main() {

yylex();

return 0;

}

**Output:**

****

**c. Extract single and multiline comments from C Program**

%{

#include<stdio.h>

int sl=0,ml=0,c;

%}

%%

[/]{1}[/]{1}[a-zA-Z0-9\_ ]\* {sl++;} printf("Single line comment %d",sl);

[/]{1}[\*]{1}[a-zA-Z0-9\_ ]\*[\*]{1}[/]{1} {ml++;} printf("Multipleline comment%d",ml);

%%

int yywrap(void){return 1;}

int main()

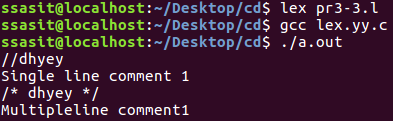
{

yylex();

return 0;

}

**Output:**



**Practical-4**

**Aim:** Implement following Programs Using Lex.

a. Convert Roman to Decimal

b. Check weather given statement is compound or simple

c. Extract html tags from .html file

**a. Convert Roman to Decimal**

#include<stdio.h>

int total=0;

%}

WS [ \t]+

%%

I total += 1;

IV total += 4;

V total += 5;

IX total += 9;

X total += 10;

XL total += 40;

L total += 50;

XC total += 90;

C total += 100;

CD total += 400;

D total += 500;

CM total += 900;

M total += 1000;

{WS} |

\n return total;

%%

int main (void)

{

int first;

printf("Enter Roman Number: ");

first = yylex ();

printf("Decimal Number is: %d\n", first);

return 0;

}

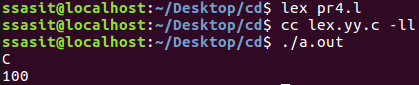
int yywrap(void)

{

return 1;

}

**Output:**



**b. Check weather given statement is compound or simple**

%{

#include<stdio.h>

int flag=0;

%}

%%

and |

or |

but |

because |

if |

then |

nevertheless { flag=1; }

. ;

\n { return 0; }

%%

int main()

{

printf("Enter the sentence:\n");

yylex();

if(flag==0)

printf("Simple sentence\n");

else

printf("compound sentence\n");

}

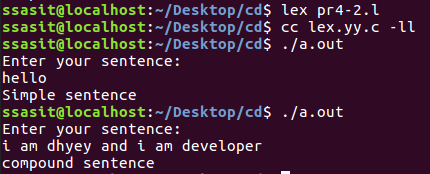
int yywrap( )

{

return 1;

}

**Output:**



**c. Extract html tags from .html file**

/\* Declaration section\*/

%{

%}

%%

"<"[^>]\*> {printf("%s\n", yytext); } /\* if anything enclosed in

these < > occur print text\*/

. ; // else do nothing

%%

int yywrap(){}

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

{

// Open tags.txt in read mode

extern FILE \*yyin = fopen("ex.txt","r");

// The function that starts the analysis

yylex();

return 0;

}

**Output:**



**Practical-5**

**Aim:** Implementation of Recursive Descent Parser without backtracking.

Input: The string to be parsed.

Output: Whether string parsed successfully or not.

Explanation: Students have to implement the recursive procedure for

RDP for a typical grammar. The production no. is displayed as it is used to derive the string.

**Code:**

#include <stdio.h>

char str[20];

int p = 0; // Pointer to traverse the string

char l; // Current character being parsed

int flag = 1; // Flag to determine if the string is accepted (1) or not (0)

void getString() {

printf("The Production rules are as follows:\n");

printf("E -> i E'\n");

printf("E' -> + i E | ε\n");

printf("Enter a String: ");

scanf("%[^\n]%\*c", str);

}

void error() {

flag = 0;

}

void match(char t) {

if (l == t) {

p++;

l = str[p];

} else {

error();

}

}

void E(); // Forward declaration

void Edesh() {

if (l == '+') {

match('+');

match('i');

E();

} else if (l != '$') {

error();

}

}

void E() {

l = str[p];

if (l == 'i') {

match('i');

Edesh();

} else {

error();

}

}

int main() {

getString();

E();

if (l == '$' && flag == 1) {

printf("ACCEPTED\n");

} else {

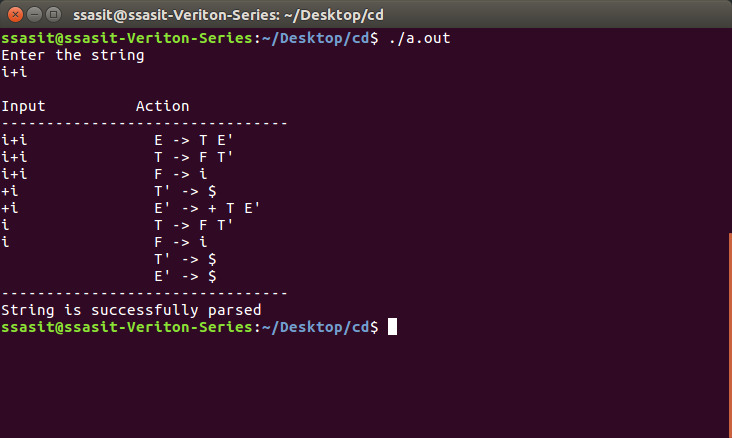
printf("NOT ACCEPTED\n");

}

return 0;

}

**Output:**



**Practical-6**

**Aim:** Find the “First” set. Input: The string consists of grammar symbols.

Output: The First set for a given string.

**Code:**

#include <stdio.h>

#include <string.h>

char first[100];

void appendToFirst(const char \*str) {

strncat(first, str, strlen(str));

}

void FiS() {

appendToFirst("{a, d}");

}

void FiA() {

appendToFirst("{a, d}");

}

void FiB() {

appendToFirst("{b, f}");

}

void FiC() {

appendToFirst("{g}");

}

int main() {

char r;

printf("Enter your Non-Terminal: ");

scanf(" %c", &r);

switch (r) {

case 'S':

FiS();

break;

case 'A':

FiA();

break;

case 'B':

FiB();

break;

case 'C':

FiC();

break;

default:

printf("INVALID NON-TERMINAL\n");

return 1;

}

printf("First of '%c' is: ", r);

puts(first);

return 0;

}

**Output:**

