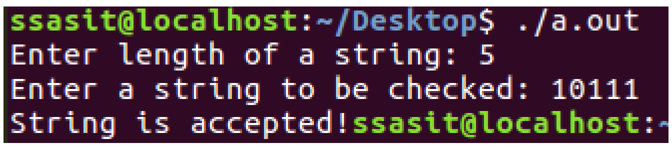
Practical-1

Aim: Implementation of Finite Automata and String Validation.

Code:

```
#include <stdio.h>
#include <string.h>

// States of the DFA
typedef enum {
    EVEN,
    ODD
} State;
 // Function to get the next state based on the current state and input character
State getNextState(State currentState, char inputChar) {
    switch(currentState) {
        case EVEN:
            return (inputChar == '1') ? ODD : EVEN;
        case ODD:
            return (inputChar == '1') ? EVEN : ODD;
        default:
            return EVEN;
    }
}

// Function to check if the string is accepted by the DFA
int isAccepted(char *inputString) {
    State currentState = EVEN;
    for (int i = 0; i < strlen(inputString); i++) {
        currentState = getNextState(currentState, inputString[i]);
    }
    return (currentState == EVEN);
}

int main() {
    char inputString[100];
    printf("Enter a binary string: ");
    scanf("%s", inputString);

    if (isAccepted(inputString)) {
        printf("The string is accepted by the DFA (even number of '1's).\n");
    } else {
        printf("The string is not accepted by the DFA (odd number of '1's).\n");
    }

    return 0;
}
```

Output:



```
ssasit@localhost:~/Desktop$ ./a.out
Enter length of a string: 5
Enter a string to be checked: 10111
String is accepted!ssasit@localhost:~
```

Practical-2

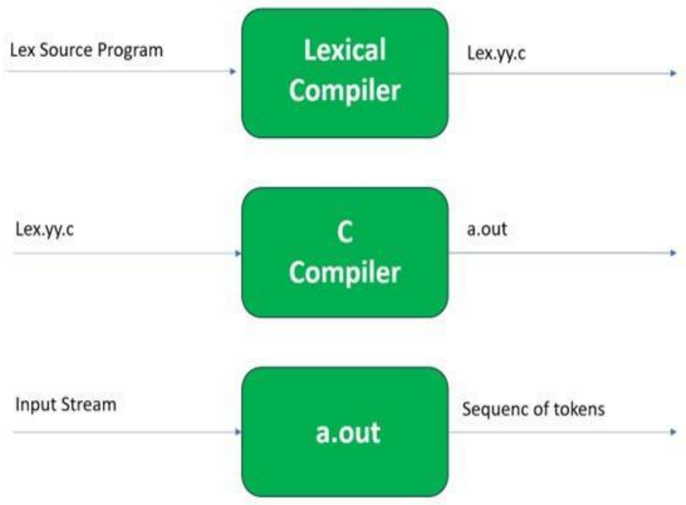
Aim: Introduction to Lex & Flex Tool.

➢ **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:

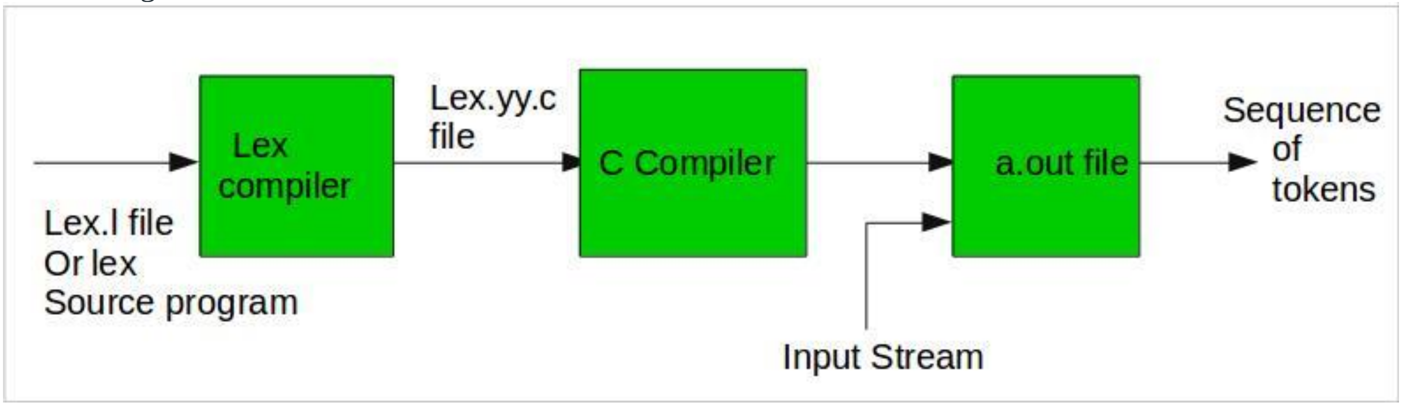
```
declaration
number[0-9]
%%
translation
if {return (IF);}
%%
auxiliary function
int numberSum()
```

- **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.

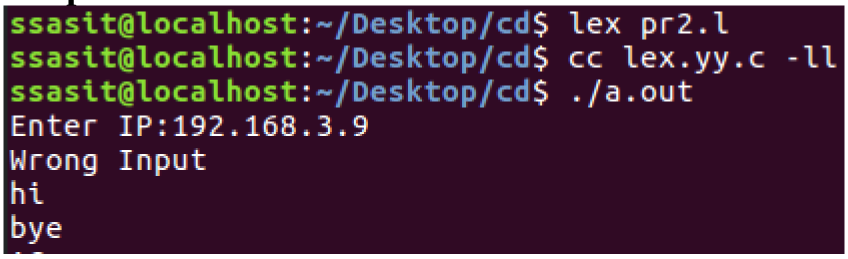
Basic Program Structure:

```
%{
// Definitions
%}


%%
Rules
%%


User code section
```

CODE:

```
%{
#include<stdio.h>
%}
%%
"hi" {printf("bye");}
.* {printf("Wrong Input");}
%%
void main()
{
printf("Enter IP:");
yylex();
}
```

Output:



```
ssasit@localhost:~/Desktop/cd$ lex pr2.l
ssasit@localhost:~/Desktop/cd$ cc lex.yy.c -ll
ssasit@localhost:~/Desktop/cd$ ./a.out
Enter IP:192.168.3.9
Wrong Input
hi
bye
```

Practical-3

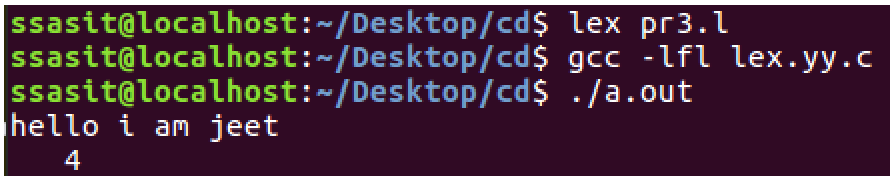
Aim: Implementation following Programs Using Lex.

a. Generate Histogram of words
b. Check Cypher
c. Extract single and multiline comments from C Program

a. 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:

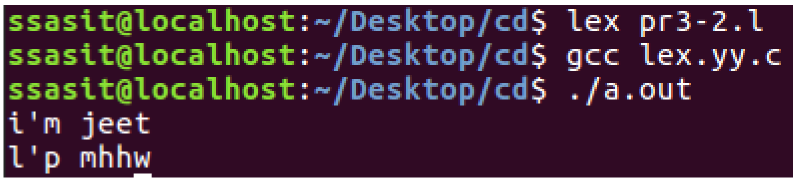


```
ssasit@localhost:~/Desktop/cd$ lex pr3.l
ssasit@localhost:~/Desktop/cd$ gcc -lfl lex.yy.c
ssasit@localhost:~/Desktop/cd$ ./a.out
hello i am jeet
    4
```

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:

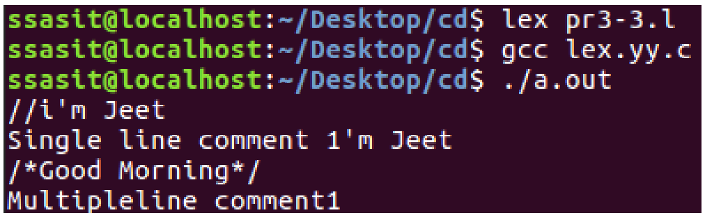


```
ssasit@localhost:~/Desktop/cd$ lex pr3-2.l
ssasit@localhost:~/Desktop/cd$ gcc lex.yy.c
ssasit@localhost:~/Desktop/cd$ ./a.out
i'm jeet
l'p mhhw
```

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:



```
ssasit@localhost:~/Desktop/cd$ lex pr3-3.l
ssasit@localhost:~/Desktop/cd$ gcc lex.yy.c
ssasit@localhost:~/Desktop/cd$ ./a.out
//i'm Jeet
Single line comment 1'm Jeet
/*Good Morning*/
Multipleline comment1
```

Practical-4

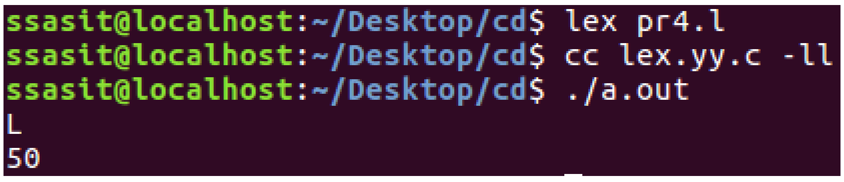
Aim: Implement following Programs Using Lex.

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:

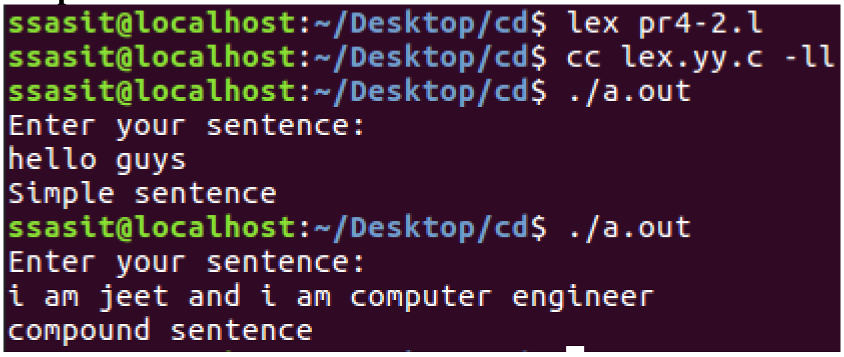


```
ssasit@localhost:~/Desktop/cd$ lex pr4.l
ssasit@localhost:~/Desktop/cd$ cc lex.yy.c -ll
ssasit@localhost:~/Desktop/cd$ ./a.out
L
50
```

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:



```
ssasit@localhost:~/Desktop/cd$ lex pr4-2.l
ssasit@localhost:~/Desktop/cd$ cc lex.yy.c -ll
ssasit@localhost:~/Desktop/cd$ ./a.out
Enter your sentence:
hello guys
Simple sentence
ssasit@localhost:~/Desktop/cd$ ./a.out
Enter your sentence:
i am jeet and i am computer engineer
compound sentence
```

c. Extract html tags from .html file

index.html:

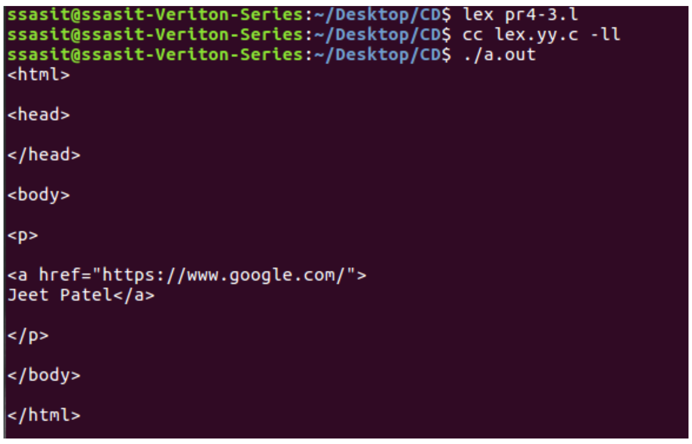
```
<html>
<head>
</head>
<body>
<p>
```

```
<a href="https://www.google.com/">Jeet Patel</a>
</p>
</body>
</html>
```

Code:

```
%{
#include<stdio.h>
%}
%%
"<"[^>]*> {printf("%s\n", yytext); }
.;
%%
int yywrap(){}
int main(int argc, char*argv[])
{
yyin = fopen("index.html","r");
yylex();
return 0;
}
```

Output:



```
ssasit@ssasit-Veriton-Series:~/Desktop/CD$ lex pr4-3.l
ssasit@ssasit-Veriton-Series:~/Desktop/CD$ cc lex.yy.c -ll
ssasit@ssasit-Veriton-Series:~/Desktop/CD$ ./a.out
<html>

<head>

</head>

<body>

<p>

<a href="https://www.google.com/">
Jeet Patel</a>

</p>

</body>

</html>
```

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. are displayed as they are used to derive the string.

CODE:

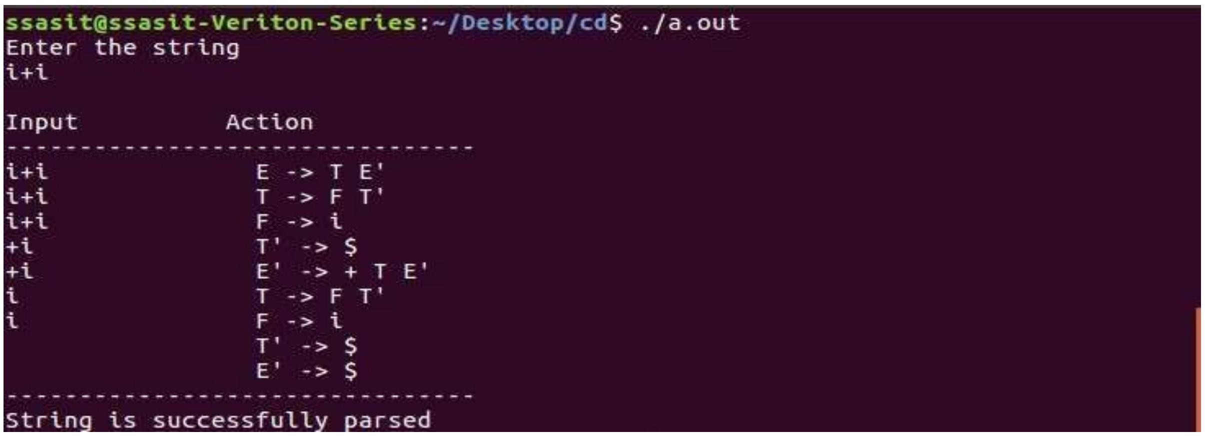
```
#include <stdio.h>

#include <stdlib.h>

// Global variables

char *input;

char currentChar;

// Function prototypes

void E();

void EPrime();

void T();

void TPrime();

void F();

void advance();

void error();

void advance() {

    currentChar = *input++;

 }

void error() {

        printf("Error: Invalid string.\n");

        exit(1);

  }

   // E -> T E'
```

```
    void E() {
    printf("E -> T E'\n");
    T();
    EPrime();
    }
    // E' -> + T E' | ε void EPrime() {
    if (currentChar == '+') {
              printf("E' -> + T E'\n");
              advance();
              T();
              EPrime();
    } else {
       printf("E' -> ε\n");
    }
}
// T -> F T'
void T() {
    printf("T -> F T'\n");
    F();
    TPrime();
}
// T' -> * F T' | ε
void TPrime() {
      if (currentChar == '*') {
              printf("T' -> * F T'\n");
              advance();
              F();
              TPrime();
      } else {
              printf("T' -> ε\n");
```

```
            }
}
// F -> ( E ) | i
void F() {
    if (currentChar == 'i') {
    printf("F -> i\n");

    advance();

    } else if (currentChar == '(') {

    printf("F -> ( E )\n");

    advance();

        E();

        if (currentChar == ')')
            { advance();

        } else {

            error();

        }
    } else {

        error();

    }
}
int main() {
    // Input string

    input = "i+i$";

    advance();

    E();

    if (currentChar == '$') {
        printf("String is successfully parsed\n");

    } else {

        error();
```

```
}
return 0;
}
```

OUTPUT:



```
ssasit@ssasit-Veriton-Series:~/Desktop/cd$ ./a.out
Enter the string
i+i

Input           Action
--------------------------------
i+i              E -> T E'
i+i              T -> F T'
i+i              F -> i
+i               T' -> $
+i               E' -> + T E'
i                T -> F T'
i                F -> i
                 T' -> $
                 E' -> $
--------------------------------
String is successfully parsed
```

Practical-6

Aim: Finding "First" set Input: The string consists of grammar symbols.

Output: The First set for a given string.

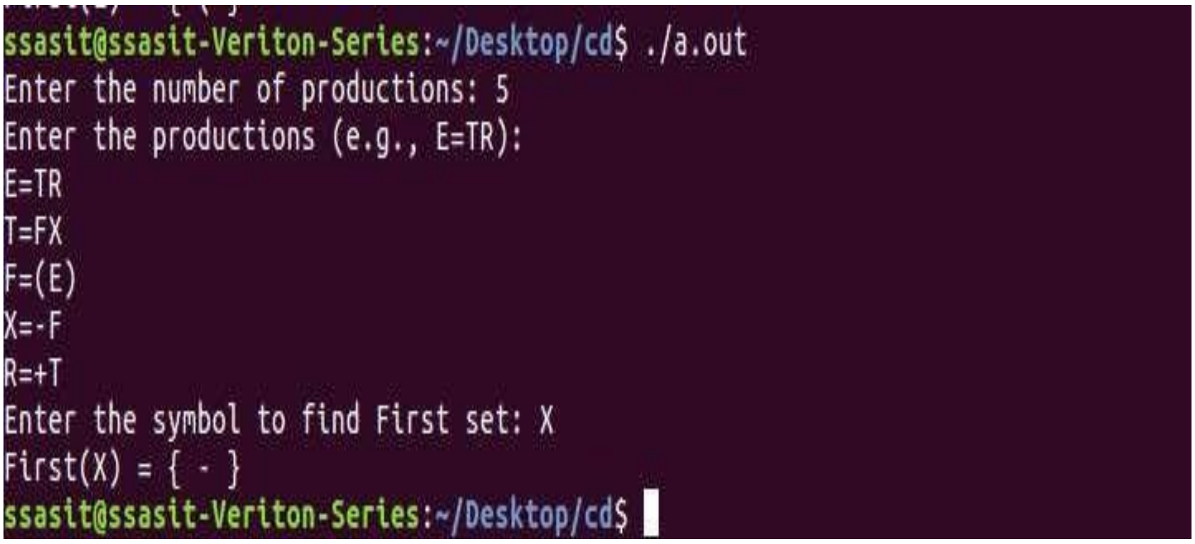
Explanation: The student has to assume a typical grammar. The program when run will ask for the string to be entered. The program will find the First set of the given string.

CODE:

```
#include<stdio.h>
#include<ctype.h>
#include<string.h>
#define MAX 10


   void findFirst(char, int, int);
   void addToResultSet(char);
   int numOfProductions;
   char productionSet[MAX][MAX];
   char firstSet[MAX];
  int main() {
   int i, choice;
   char c, ch;
   printf("Enter the number of productions: ");
   scanf("%d", &numOfProductions); printf("Enter the
   productions (e.g., E=TR):\n"); for (i = 0; i <
   numOfProductions; i++) {
  scanf("%s", productionSet[i]);
  }
  do
  {
  printf("Enter the symbol to find First set: ");
```

```
        scanf(" %c", &c);

        findFirst(c, 0, 0);
        printf("First(%c) = { ");

        for (i = 0; firstSet[i] != '\0'; i++) {
            printf("%c ", firstSet[i]);
        }
        printf("}\n");


        firstSet[0] = '\0';
        printf("Do you want to find another First set? (1 for Yes / 0 for No): "); scanf("%d",
        &choice);

    } while(choice == 1); return 0;

}

void findFirst(char c, int q1, int q2) { int j;

    if (!(isupper(c)))
        { addToResultSet(c);

    }
    for (j = 0; j < numOfProductions; j++) { if
        (productionSet[j][0] == c) {

            if (productionSet[j][2] == '$') {
                if (productionSet[q1][q2] == '\0')
                    { addToResultSet('$');

                } else if (productionSet[q1][q2] != '\0' && (q1 != 0 || q2 != 0))
                    { findFirst(productionSet[q1][q2], q1, (q2 + 1));

                } else {
                    addToResultSet('$');

                }
            } else if (!isupper(productionSet[j][2])) { addToResultSet(productionSet[j][2]);
            } else {

                findFirst(productionSet[j][2], j, 3);

            }

        }

    }

}
void addToResultSet(char c) { int i;

    for (i = 0; firstSet[i] != '\0'; i++) { if (firstSet[i]

        == c) {

            return;

        }

    }

    firstSet[i] = c; firstSet[i + 1] = '\0';

}
```

OUTPUT:



```
ssasit@ssasit-Veriton-Series:~/Desktop/cd$ ./a.out
Enter the number of productions: 5
Enter the productions (e.g., E=TR):
E=TR
T=FX
F=(E)
X=-F
R=+T
Enter the symbol to find First set: X
First(X) = { - }
ssasit@ssasit-Veriton-Series:~/Desktop/cd$
```