DELHI TECHNICAL CAMPUS
GREATER NOIDA
*Affiliated to GGSIPU and approved by AICTE & COA*

# PRACTICAL FILE

# SESSION: 2023-24

## Operating Systems Lab (CIC-353)

## III Year, V Sem

**Submitted to:**

**Name:** Dr Seema Verma
**Designation:** Professor

**Submitted by:**

**Name:** Adityan Verma
**Enrollment No.** 07818002721

**Department of Computer Science and Engineering**
**Delhi Technical Campus, Greater Noida**

# INDEX

| S.NO. | PROGRAM NAME | REMARKS | DATE OF EXPERIMENT | DATE OF SUBMISSION | SIGN. |
|---|---|---|---|---|---|
| 1 | About **LEX/YACC** of compiler writing | | | | |
| 2 | Write a program to check whether a **string** belong to the grammar or not. | | | | |
| 3 | Write a program to check whether a string include **keyword** or not. | | | | |
| 4 | Write a program to remove **Left Recursion** from a grammar | | | | |
| 5 | Write a program to perform **Left Factoring** on a grammar. | | | | |
| 6 | Write a program to show all the **operations of a stack**. | | | | |
| 7 | Write a program to find out the **leading or FOLLOW** of the non-terminal in a grammar. | | | | |
| 8 | Write a program to Implement **Shift Reduce parsing** for a String. | | | | |
| 9 | Write a program to find out the **FIRST** of the Non-terminals in a grammar. | | | | |
| 10 | Write a program to check whether a grammar is **operator precedence**. | | | | |

# Experiment-1

**AIM:** About **LEX/YACC** of compiler writing.

## Lexical Analysis

It is the first step of compiler design, it takes the input as a stream of characters and gives the output as tokens also known as tokenization. The tokens can be classified into identifiers, Sperators, Keywords, Operators, Constant and Special Characters.

It has three phases:

1. **Tokenization:** It takes the stream of characters and converts it into tokens.
2. **Error Messages:** It gives errors related to lexical analysis such as exceeding length, unmatched string, etc.
3. **Eliminate Comments:** Eliminates all the spaces, blank spaces, new lines, and indentations.

## 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 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()
```

## YACC

- YACC stands for Yet Another Compiler Compiler.
- YACC provides a tool to produce a parser for a given grammar.
- YACC is a program designed to compile a LALR (1) grammar.
- It is used to produce the source code of the syntactic analyzer of the language produced by LALR (1) grammar.
- The input of YACC is the rule or grammar and the output is a C program.

These are some points about YACC:

**Input: A CFG- file.y**

**Output: A parser y.tab.c (yacc)**

- The output file "file.output" contains the parsing tables.
- The file "file.tab.h" contains declarations.
- The parser called the yyparse ().
- Parser expects to use a function called yylex () to get tokens.

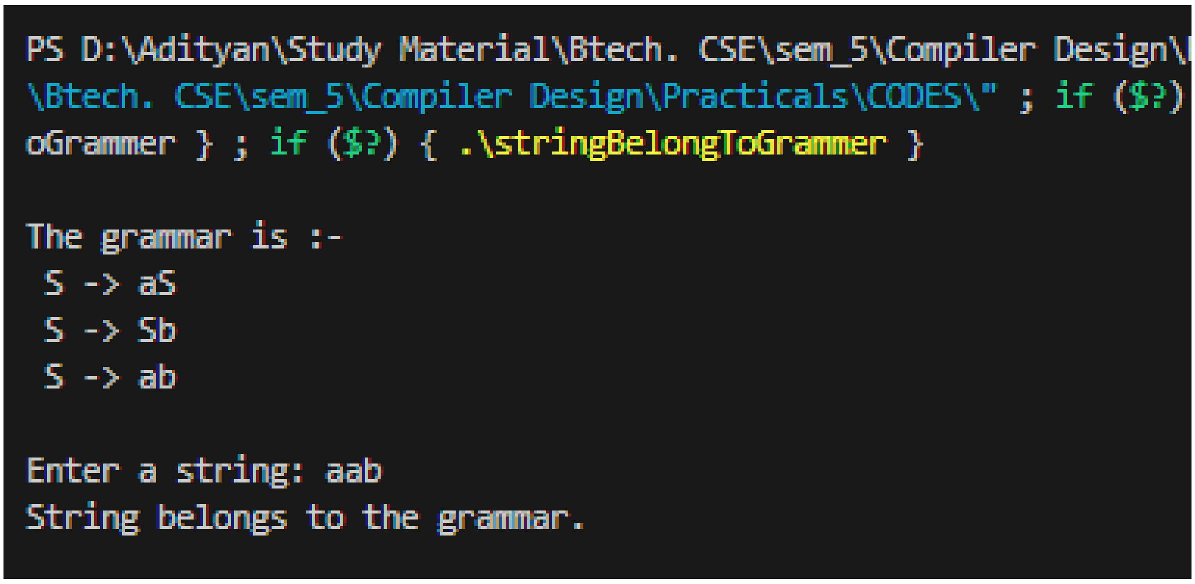
# Experiment-2

**AIM:** Write a program to check whether a **string** belong to the grammar or not.

**Program:**

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

int isBelongToGrammar(char *str) {
    int len = strlen(str);
    int i = 0;

    while (i < len) {
        if (str[i] == 'a') {
            i++;
        } else if (str[i] == 'b') {
            i++;
            while (i < len && str[i] == 'b') {
                i++;
            }
        } else {
            return 0;
        }
    }

    // Check if the string ends in 'ab'
    if (i == len || (i == len - 1 && str[i] == 'a')) {
        return 1;
    }
    return 0;
}

int main() {
    char input[100];

    printf("\nThe grammar is :- \n S -> aS \n S -> Sb \n S -> ab\n\n");

    printf("Enter a string: ");
    scanf("%s", input);

    if (isBelongToGrammar(input)) {
        printf("String belongs to the grammar.\n\n");
    } else {
        printf("String does not belong to the grammar.\n\n");
    }
    return 0;
}
```

**Output:**



```
PS D:\Adityan\Study Material\Btech. CSE\sem_5\Compiler Design\
\Btech. CSE\sem_5\Compiler Design\Practicals\CODES\" ; if ($?)
oGrammer } ; if ($?) { .\stringBelongToGrammer }

The grammar is :-
 S -> aS
 S -> Sb
 S -> ab

Enter a string: aab
String belongs to the grammar.
```

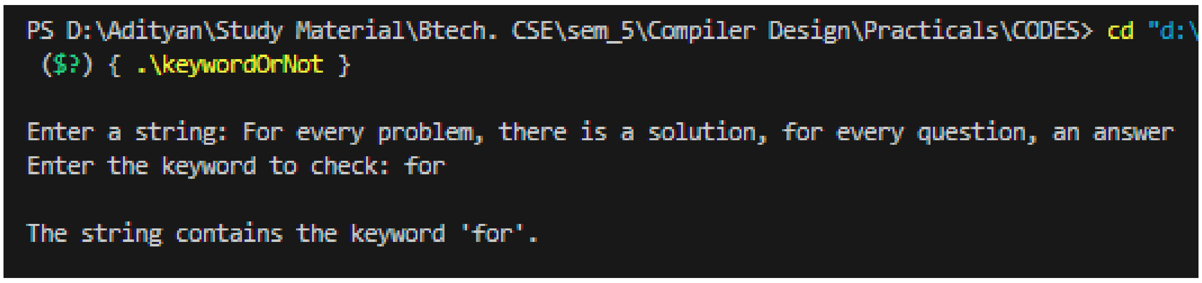
# Experiment-3

**AIM:** Write a program to check whether a string include **keyword** or not.

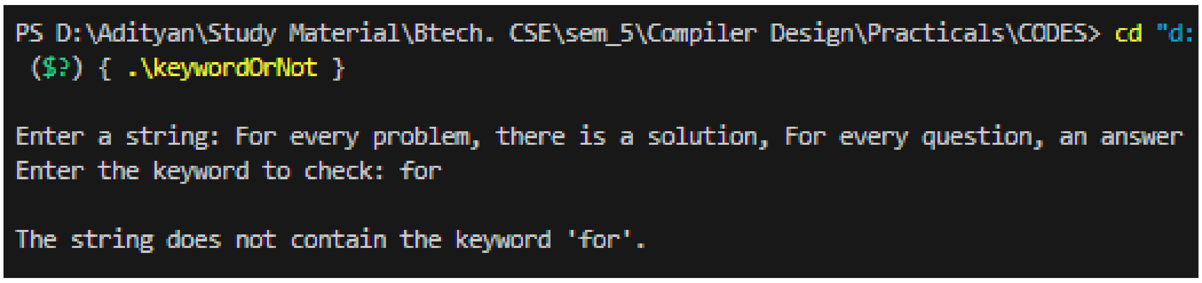
**Program:**

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

int containsKeyword(const char *str, const char *keyword) {
    if (strstr(str, keyword) != NULL) {
        return 1; // Keyword found in the string
    }
    return 0; // Keyword not found in the string
}

int main() {
    char input[500];
    char keyword[20];

    printf("\nEnter a string: ");
    fgets(input, sizeof(input), stdin);
    input[strcspn(input, "\n")] = '\0'; // Remove newline from fgets input

    printf("Enter the keyword to check: ");
    fgets(keyword, sizeof(keyword), stdin);
    keyword[strcspn(keyword, "\n")] = '\0'; // Remove newline from fgets input

    if (containsKeyword(input, keyword)) {
        printf("\nThe string contains the keyword '%s'.\n", keyword);
    } else {
        printf("\nThe string does not contain the keyword '%s'.\n", keyword);
    }

    printf("\n");

    return 0;
}
```

**Output:**



```
PS D:\Adityan\Study Material\Btech. CSE\sem_5\Compiler Design\Practicals\CODES> cd "d:\
 ($?) { .\keywordOrNot }

Enter a string: For every problem, there is a solution, for every question, an answer
Enter the keyword to check: for

The string contains the keyword 'for'.
```



```
PS D:\Adityan\Study Material\Btech. CSE\sem_5\Compiler Design\Practicals\CODES> cd "d:
 ($?) { .\keywordOrNot }

Enter a string: For every problem, there is a solution, For every question, an answer
Enter the keyword to check: for

The string does not contain the keyword 'for'.
```

# Experiment-4

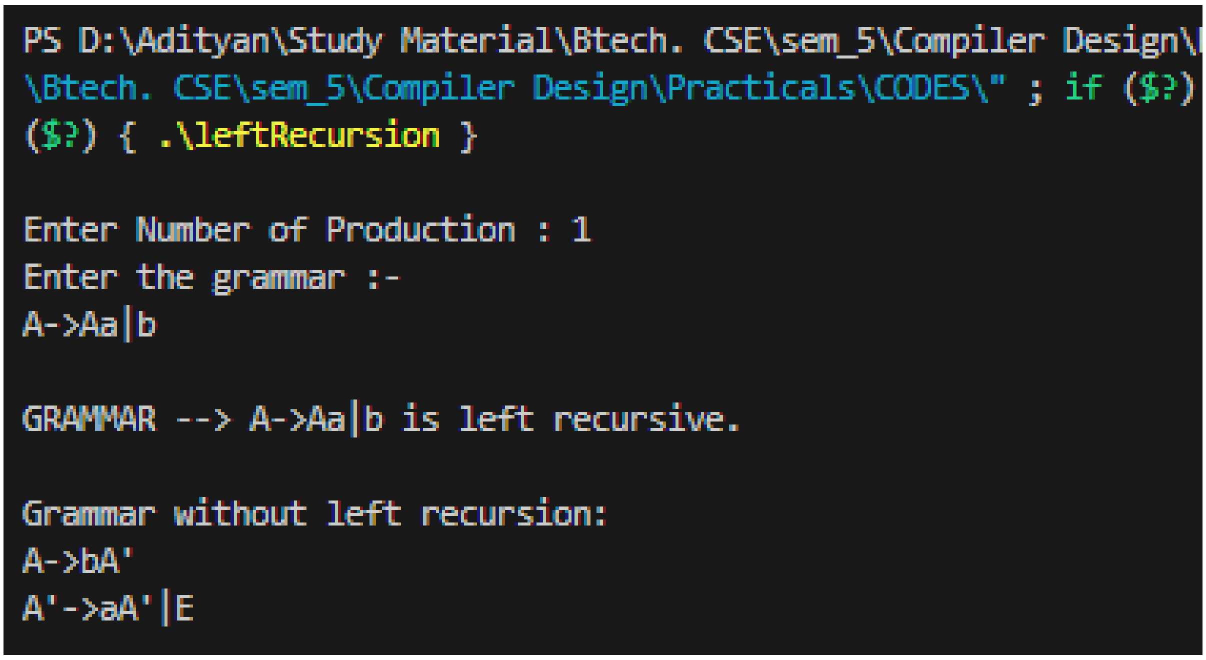
**AIM:** Write a program to remove **Left Recursion** from a grammar.

**Program:**

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

#define SIZE 10

int main()
{
   char non_terminal;
   char beta, alpha;
   int num;
   char production[10][SIZE];
   int index = 3;
   printf("\nEnter Number of Production : ");
   scanf("%d", &num);
   printf("Enter the grammar :-\n");
   for (int i = 0; i < num; i++)
   {
      scanf("%s", production[i]);
   }
   for (int i = 0; i < num; i++)
   {
      printf("\nGRAMMAR --> %s", production[i]);
      non_terminal = production[i][0];
      if (non_terminal == production[i][index])
      {
         alpha = production[i][index + 1];
         printf(" is left recursive.\n\n");
         while (production[i][index] != 0 && production[i][index] != '|')
            index++;
         if (production[i][index] != 0)
         {
            beta = production[i][index + 1];
            printf("Grammar without left recursion:\n");
            printf("%c->%c%c\'", non_terminal, beta, non_terminal);
            printf("\n%c\'->%c%c\'|E\n", non_terminal, alpha, non_terminal);
         }
         else
            printf(" can't be reduced\n");
      }
```

```
        else
            printf(" is not left recursive.\n");
        index = 3;
    }

    printf("\n");
    return 0;
}
```

**Output:**



```
PS D:\Adityan\Study Material\Btech. CSE\sem_5\Compiler Design\
\Btech. CSE\sem_5\Compiler Design\Practicals\CODES\" ; if ($?)
($?) { .\leftRecursion }

Enter Number of Production : 1
Enter the grammar :-
A->Aa|b

GRAMMAR --> A->Aa|b is left recursive.

Grammar without left recursion:
A->bA'
A'->aA'|E
```

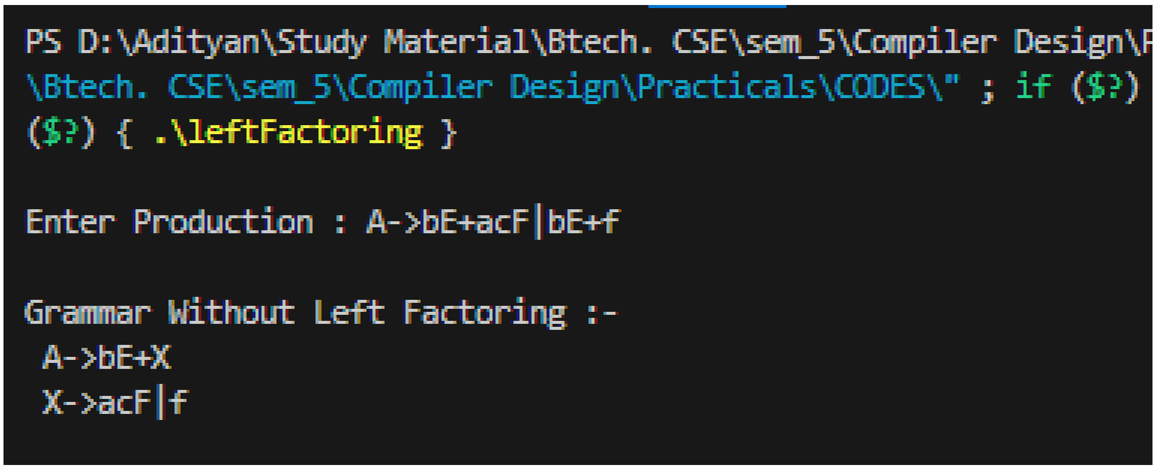
# Experiment-5

**AIM:** Write a program to perform **Left Factoring** on a grammar

**Program:**

```
#include <stdio.h>
#include <string.h>
int main()
{
    char gram[20], part1[20], part2[20], modifiedGram[20], newGram[20], tempGram[20];
    int i, j = 0, k = 0, l = 0, pos;
    printf("\nEnter Production : A->");
    gets(gram);
    for (i = 0; gram[i] != '|'; i++, j++)
        part1[j] = gram[i];
    part1[j] = '\0';
    for (j = ++i, i = 0; gram[j] != '\0'; j++, i++)
        part2[i] = gram[j];
    part2[i] = '\0';
    for (i = 0; i < strlen(part1) || i < strlen(part2); i++) {
        if (part1[i] == part2[i]) {
            modifiedGram[k] = part1[i];
            k++;
            pos = i + 1;
        }
    }
    for (i = pos, j = 0; part1[i] != '\0'; i++, j++) {
        newGram[j] = part1[i];
    }
    newGram[j++] = '|';
    for (i = pos; part2[i] != '\0'; i++, j++) {
        newGram[j] = part2[i];
    }
    modifiedGram[k] = 'X';
    modifiedGram[++k] = '\0';
    newGram[j] = '\0';
    printf("\nGrammar Without Left Factoring :-\n");
    printf(" A->%s", modifiedGram);
    printf("\n X->%s\n", newGram);
    printf("\n");
    return 0;
}
```

**Output:**



```
PS D:\Adityan\Study Material\Btech. CSE\sem_5\Compiler Design\P
\Btech. CSE\sem_5\Compiler Design\Practicals\CODES\" ; if ($?)
($?) { .\leftFactoring }

Enter Production : A->bE+acF|bE+f

Grammar Without Left Factoring :-
 A->bE+X
 X->acF|f
```

# Experiment-6

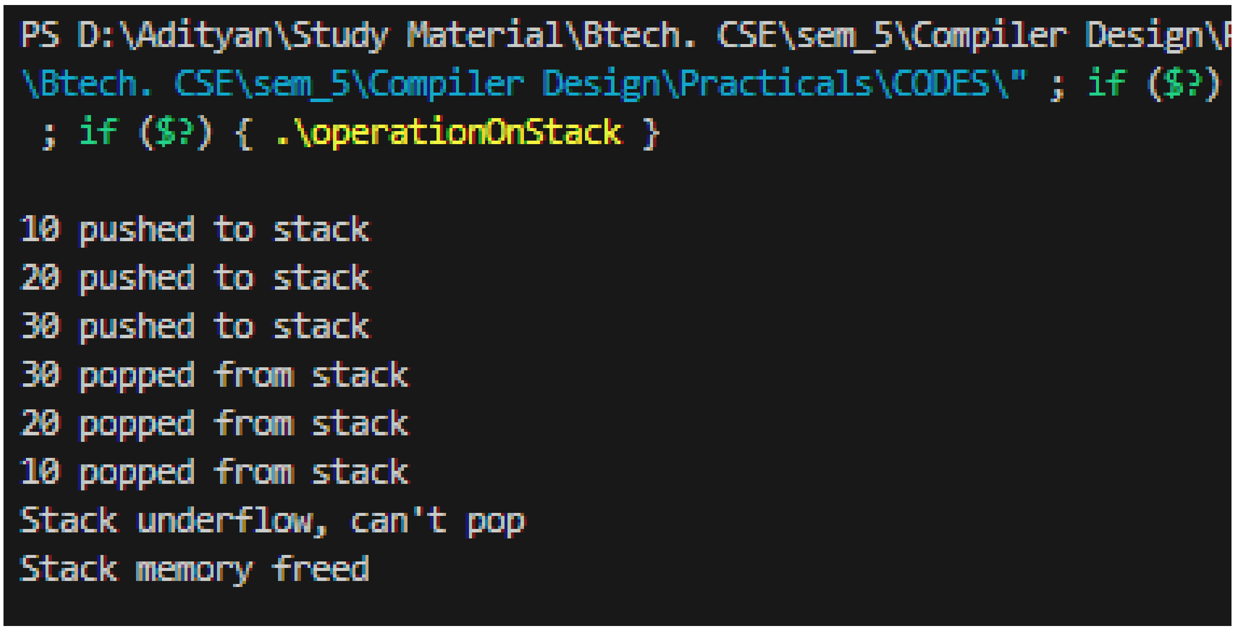
**AIM:** Write a program to show all the **operations of a stack.**

**Program:**

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#define STACK_CAPACITY 100

struct Stack {
    int top;
    unsigned capacity;
    int *array;
};

struct Stack *createStack(unsigned capacity) {
    struct Stack *stack = (struct Stack *)malloc(sizeof(struct Stack));
    if (stack == NULL) {
        printf("Memory allocation failed.\n");
        return NULL;
    }
    stack->capacity = capacity;
    stack->top = -1;
    stack->array = (int *)malloc(stack->capacity * sizeof(int));
    if (stack->array == NULL) {
        printf("Memory allocation failed.\n");
        free(stack);
        return NULL;
    }
    return stack;
}

bool isFull(struct Stack *stack) {
    return stack->top == stack->capacity - 1;
}

bool isEmpty(struct Stack *stack) {
    return stack->top == -1;
}
```

```
void push(struct Stack *stack, int item) {
    if (isFull(stack)) {
        printf("Stack overflow, can't push %d\n", item);
        return;
    }
    stack->array[++stack->top] = item;
    printf("%d pushed to stack\n", item);
}

int pop(struct Stack *stack) {
    if (isEmpty(stack)) {
        printf("Stack underflow, can't pop\n");
        return -1; // Returning an arbitrary value to indicate underflow
    }
    int item = stack->array[stack->top--];
    printf("%d popped from stack\n", item);
    return item;
}

int peek(struct Stack *stack) {
    if (isEmpty(stack)) {
        printf("Stack is empty, no top element\n");
        return -1; // Returning an arbitrary value to indicate empty stack
    }
    return stack->array[stack->top];
}

void freeStack(struct Stack *stack) {
    free(stack->array);
    free(stack);
    printf("Stack memory freed\n");
}

int main() {
    struct Stack *stack = createStack(STACK_CAPACITY);
    if (stack == NULL) {
        return -1; // Exiting due to memory allocation failure
    }
    printf("\n");

    push(stack, 10);
    push(stack, 20);
    push(stack, 30);

    pop(stack);
    pop(stack);
    pop(stack);
    pop(stack); // Trying to pop from an empty stack

    freeStack(stack);
    printf("\n");
    return 0;
}
```

**Output:**



```
PS D:\Adityan\Study Material\Btech. CSE\sem_5\Compiler Design\P
\Btech. CSE\sem_5\Compiler Design\Practicals\CODES\" ; if ($?)
 ; if ($?) { .\operationOnStack }

10 pushed to stack
20 pushed to stack
30 pushed to stack
30 popped from stack
20 popped from stack
10 popped from stack
Stack underflow, can't pop
Stack memory freed
```

# Experiment-7

**AIM:** Write a program to find out the leading or **FOLLOW** of the non-terminalsin a grammar.

**Program:**

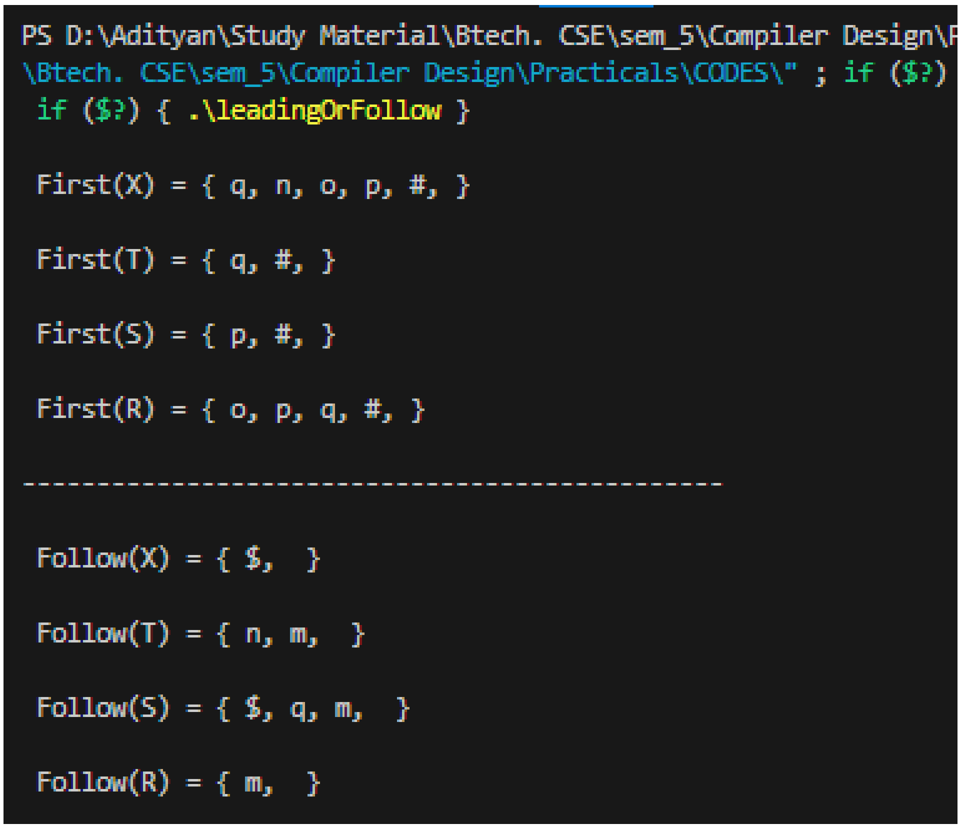
```
#include <ctype.h>
#include <stdio.h>
#include <string.h>

// Functions to calculate Follow
void followfirst(char, int, int);
void follow(char c);

// Function to calculate First
void findfirst(char, int, int);

int count, n = 0;

// Stores the final result
// of the First Sets
char calc_first[10][100];

// Stores the final result
// of the Follow Sets
char calc_follow[10][100];
int m = 0;

// Stores the production rules
char production[10][10];
char f[10], first[10];
int k;
char ck;
int e;

int main(int argc, char** argv)
```

```
{
        int jm = 0;
        int km = 0;
        int i, choice;
        char c, ch;
        count = 8;

        // The Input grammar
        strcpy(production[0], "X=TnS");
        strcpy(production[1], "X=Rm");
        strcpy(production[2], "T=q");
        strcpy(production[3], "T=#");
        strcpy(production[4], "S=p");
        strcpy(production[5], "S=#");
        strcpy(production[6], "R=om");
        strcpy(production[7], "R=ST");

        int kay;
        char done[count];
        int ptr = -1;

        // Initializing the calc_first array
        for (k = 0; k < count; k++) {
                for (kay = 0; kay < 100; kay++) {
                        calc_first[k][kay] = '!';
                }
        }
        int point1 = 0, point2, xxx;

        for (k = 0; k < count; k++) {
                c = production[k][0];
                point2 = 0;
                xxx = 0;

                // Checking if First of c has
                // already been calculated
                for (kay = 0; kay <= ptr; kay++)
        if (c == done[kay])
            xxx = 1;

    if (xxx == 1)
        continue;

    // Function call
    findfirst(c, 0, 0);
    ptr += 1;

    // Adding c to the calculated list
    done[ptr] = c;
    printf("\n First(%c) = { ", c);
    calc_first[point1][point2++] = c;

    // Printing the First Sets of the grammar
    for (i = 0 + jm; i < n; i++) {
        int lark = 0, chk = 0;

        for (lark = 0; lark < point2; lark++) {

            if (first[i] == calc_first[point1][lark]) {
                chk = 1;
                break;
            }
        }
        if (chk == 0) {
            printf("%c, ", first[i]);
            calc_first[point1][point2++] = first[i];
        }
    }
    printf("}\n");
    jm = n;
    point1++;
}
printf("\n");
printf("-----------------------------------------------"
        "\n\n");
char donee[count];
ptr = -1;

// Initializing the calc_follow array
for (k = 0; k < count; k++) {
        for (kay = 0; kay < 100; kay++) {
                calc_follow[k][kay] = '!';
        }
}
point1 = 0;
int land = 0;
for (e = 0; e < count; e++) {
        ck = production[e][0];
        point2 = 0;
        xxx = 0;

        // Checking if Follow of ck
        // has already been calculated
        for (kay = 0; kay <= ptr; kay++)
                if (ck == donee[kay])
                        xxx = 1;

        if (xxx == 1)
                continue;
        land += 1;

        // Function call
        follow(ck);
        ptr += 1;

        // Adding ck to the calculated list
        donee[ptr] = ck;
        printf(" Follow(%c) = { ", ck);
        calc_follow[point1][point2++] = ck;

        // Printing the Follow Sets of the grammar
```

```
                for (i = 0 + km; i < m; i++) {
                        int lark = 0, chk = 0;
                        for (lark = 0; lark < point2; lark++) {
                                if (f[i] == calc_follow[point1][lark]) {
                                        chk = 1;
                                        break;
                                }
                        }
                        if (chk == 0) {
                                printf("%c, ", f[i]);
                                calc_follow[point1][point2++] = f[i];
                        }
                }
                printf(" }\n\n");
                km = m;
                point1++;
        }
}

void follow(char c)
{
        int i, j;

        // Adding "$" to the follow
        // set of the start symbol
        if (production[0][0] == c) {
                f[m++] = '$';
        }
        for (i = 0; i < 10; i++) {
                for (j = 2; j < 10; j++) {
                        if (production[i][j] == c) {
                                if (production[i][j + 1] != '\0') {
                                        // Calculate the first of the next
                                        // Non-Terminal in the production
                                        followfirst(production[i][j + 1], i,
                                                                (j + 2));
                                }
                                if (production[i][j + 1] == '\0'
                                        && c != production[i][0]) {
                                        // Calculate the follow of the
                                        // Non-Terminal in the L.H.S. of the
                                        // production
                                        follow(production[i][0]);
                                }
                        }
                }
        }
}

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

        // The case where we
        // encounter a Terminal
        if (!(isupper(c))) {
                first[n++] = c;
        }
        for (j = 0; j < count; j++) {
                if (production[j][0] == c) {
                        if (production[j][2] == '#') {
                                if (production[q1][q2] == '\0')
                                        first[n++] = '#';
                                else if (production[q1][q2] != '\0'
                                                && (q1 != 0 || q2 != 0)) {
                                        // Recursion to calculate First of New
                                        // Non-Terminal we encounter after
                                        // epsilon
                                        findfirst(production[q1][q2], q1,
                                                        (q2 + 1));
                                }
                                else
                                        first[n++] = '#';
                        }
                        else if (!isupper(production[j][2])) {
                                first[n++] = production[j][2];
                        }
                        else {
                                // Recursion to calculate First of
                                // New Non-Terminal we encounter
                                // at the beginning
                                findfirst(production[j][2], j, 3);
                        }
                }
        }
}

void followfirst(char c, int c1, int c2)
{
        int k;

        // The case where we encounter
        // a Terminal
        if (!(isupper(c)))
                f[m++] = c;
        else {
                int i = 0, j = 1;
                for (i = 0; i < count; i++) {
                        if (calc_first[i][0] == c)
                                break;
                }

                // Including the First set of the
                // Non-Terminal in the Follow of
                // the original query
                while (calc_first[i][j] != '!') {
                        if (calc_first[i][j] != '#') {
                                f[m++] = calc_first[i][j];
                        }
                        else {
```

```
                                        if (production[c1][c2] == '\0') {
                                                // Case where we reach the
                                                // end of a production
                                                follow(production[c1][0]);
                                        }
                                        else {
                                                // Recursion to the next symbol
                                                // in case we encounter a "#"
                                                followfirst(production[c1][c2], c1,
                                                                        c2 + 1);
                                        }
                                }
                                j++;
                        }
                }
}
```

**<u>Output:</u>**



```
PS D:\Adityan\Study Material\Btech. CSE\sem_5\Compiler Design\P
\Btech. CSE\sem_5\Compiler Design\Practicals\CODES\" ; if ($?)
 if ($?) { .\leadingOrFollow }

 First(X) = { q, n, o, p, #, }

 First(T) = { q, #, }

 First(S) = { p, #, }

 First(R) = { o, p, q, #, }

-----------------------------------------------

 Follow(X) = { $,  }

 Follow(T) = { n, m,  }

 Follow(S) = { $, q, m,  }

 Follow(R) = { m,  }
```

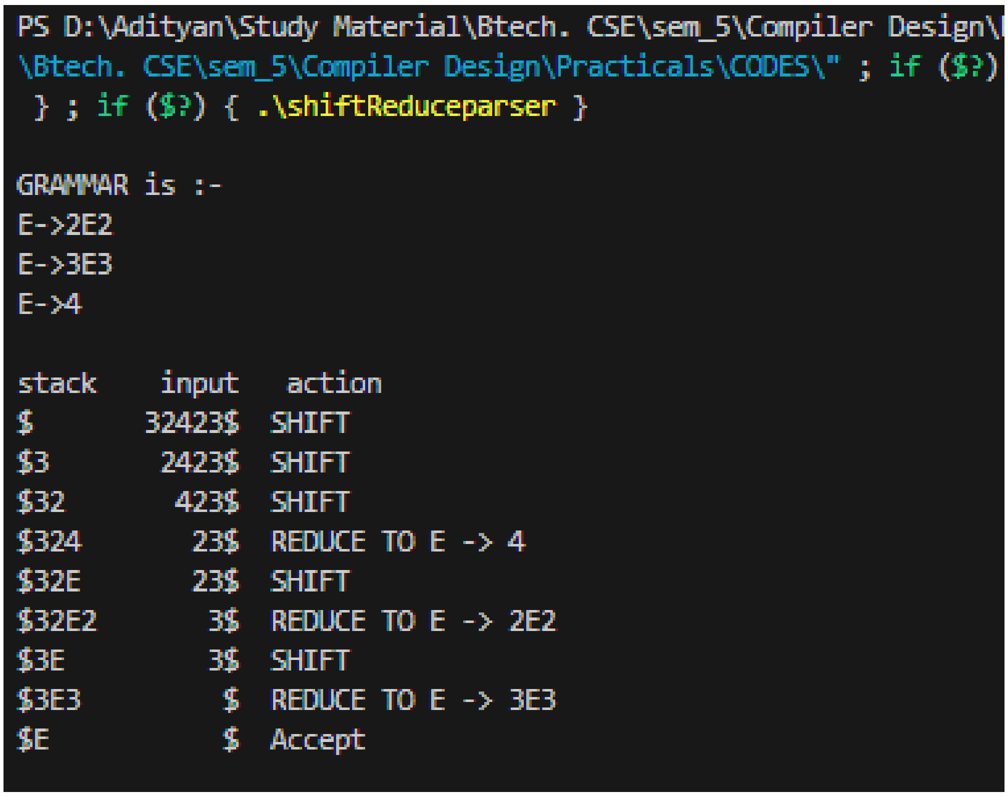
# Experiment-8

**AIM:** Write a program to Implement **Shift Reduce parsing** for a String.

**Program:**

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

int z = 0, i = 0, j = 0, c = 0;
char a[16], ac[20], stk[15], act[10];

void check() {
    strcpy(ac, "REDUCE TO E -> ");
    for (z = 0; z < c; z++) {
        if (stk[z] == '4') {
            printf("%s4", ac);
            stk[z] = 'E';
            stk[z + 1] = '\0';
            printf("\n$%s\t%s$\t", stk, a);
        }
    }

    for (z = 0; z < c - 2; z++) {
        if (stk[z] == '2' && stk[z + 1] == 'E' && stk[z + 2] == '2') {
            printf("%s2E2", ac);
            stk[z] = 'E';
            stk[z + 1] = '\0';
            stk[z + 2] = '\0';
            printf("\n$%s\t%s$\t", stk, a);
            i = i - 2;
        }
    }

    for (z = 0; z < c - 2; z++) {
        if (stk[z] == '3' && stk[z + 1] == 'E' && stk[z + 2] == '3') {
            printf("%s3E3", ac);
            stk[z] = 'E';
            stk[z + 1] = '\0';
            printf("\n$%s\t%s$\t", stk, a);
            i = i - 2;
        }
    }
}

int main() {
    printf("\nGRAMMAR is :-\nE->2E2 \nE->3E3 \nE->4\n");
    strcpy(a, "32423");
    c = strlen(a);
    strcpy(act, "SHIFT");
    printf("\nstack \t input \t action");
    printf("\n$\t%s$\t", a);

    for (i = 0; j < c; i++, j++) {
        printf("%s", act);
        stk[i] = a[j];
        stk[i + 1] = '\0';
        a[j] = ' ';
        printf("\n$%s\t%s$\t", stk, a);
        check();
    }

    check();

    if (stk[0] == 'E' && stk[1] == '\0') {
        printf("Accept\n");
    } else {
        printf("Reject\n");
    }

    printf("\n");
    return 0;
}
```

**Output:**



```
PS D:\Adityan\Study Material\Btech. CSE\sem_5\Compiler Design\
\Btech. CSE\sem_5\Compiler Design\Practicals\CODES\" ; if ($?)
 } ; if ($?) { .\shiftReduceparser }

GRAMMAR is :-
E->2E2
E->3E3
E->4

stack     input   action
$        32423$  SHIFT
$3        2423$  SHIFT
$32        423$  SHIFT
$324        23$  REDUCE TO E -> 4
$32E        23$  SHIFT
$32E2        3$  REDUCE TO E -> 2E2
$3E          3$  SHIFT
$3E3          $  REDUCE TO E -> 3E3
$E            $  Accept
```

# Experiment-9

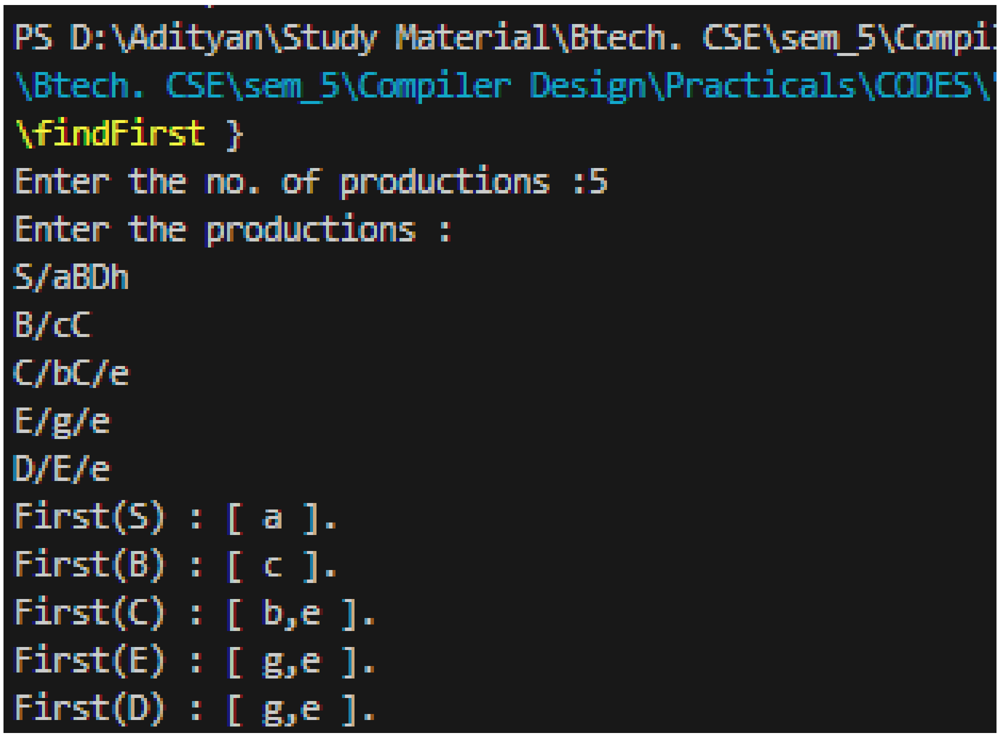
**AIM:** Write a program to find out the **FIRST** of the Non-terminals in a grammar.

**Program:**

```
#include <stdio.h>

char array[10][20], temp[10];
int c, n;

int fun2(int i, int j, int p[], int key); // Function prototype

void fun(int i, int p[])
{
    int j, k, key;
    for (j = 2; array[i][j] != '\0'; j++)
    {
        if (array[i][j - 1] == '/')
        {
            if (array[i][j] >= 'A' && array[i][j] <= 'Z')
            {
                key = 0;
                fun2(i, j, p, key);
            }
            else
            {
                key = 1;
                if (fun2(i, j, p, key))
                    temp[++c] = array[i][j];
                if (array[i][j] == '@' && p[0] != -1) // taking '@' as null symbol
                {
                    if (array[p[0]][p[1]] >= 'A' && array[p[0]][p[1]] <= 'Z')
                    {
                        key = 0;
                        fun2(p[0], p[1], p, key);
                    }
                    else if (array[p[0]][p[1]] != '/' && array[p[0]][p[1]] != '\0')
                    {
                        if (fun2(p[0], p[1], p, key))
                            temp[++c] = array[p[0]][p[1]];
                    }
                }
            }
        }
    }
}

int fun2(int i, int j, int p[], int key)
{
    int k;
    if (!key)
    {
        for (k = 0; k < n; k++)
            if (array[i][j] == array[k][0])
                break;
        p[0] = i;
        p[1] = j + 1;
        fun(k, p);
        return 0;
    }
    else
    {
        for (k = 0; k <= c; k++)
        {
            if (array[i][j] == temp[k])
                break;
        }
        if (k > c)
            return 1;
        else
```

```
            return 0;
    }
}

int main()
{
    int p[2], i, j;
    printf("Enter the no. of productions :");
    scanf("%d", &n);
    printf("Enter the productions :\n");
    for (i = 0; i < n; i++)
        scanf("%s", array[i]);
    for (i = 0; i < n; i++)
    {
        c = -1, p[0] = -1, p[1] = -1;
        fun(i, p);
        printf("First(%c) : [ ", array[i][0]);
        for (j = 0; j <= c; j++)
            printf("%c,", temp[j]);
        printf("\b ].\n");
    }

    return 0;
}
```

**Output:**



```
PS D:\Adityan\Study Material\Btech. CSE\sem_5\Compi
\Btech. CSE\sem_5\Compiler Design\Practicals\CODES\
\findFirst }
Enter the no. of productions :5
Enter the productions :
S/aBDh
B/cC
C/bC/e
E/g/e
D/E/e
First(S) : [ a ].
First(B) : [ c ].
First(C) : [ b,e ].
First(E) : [ g,e ].
First(D) : [ g,e ].
```

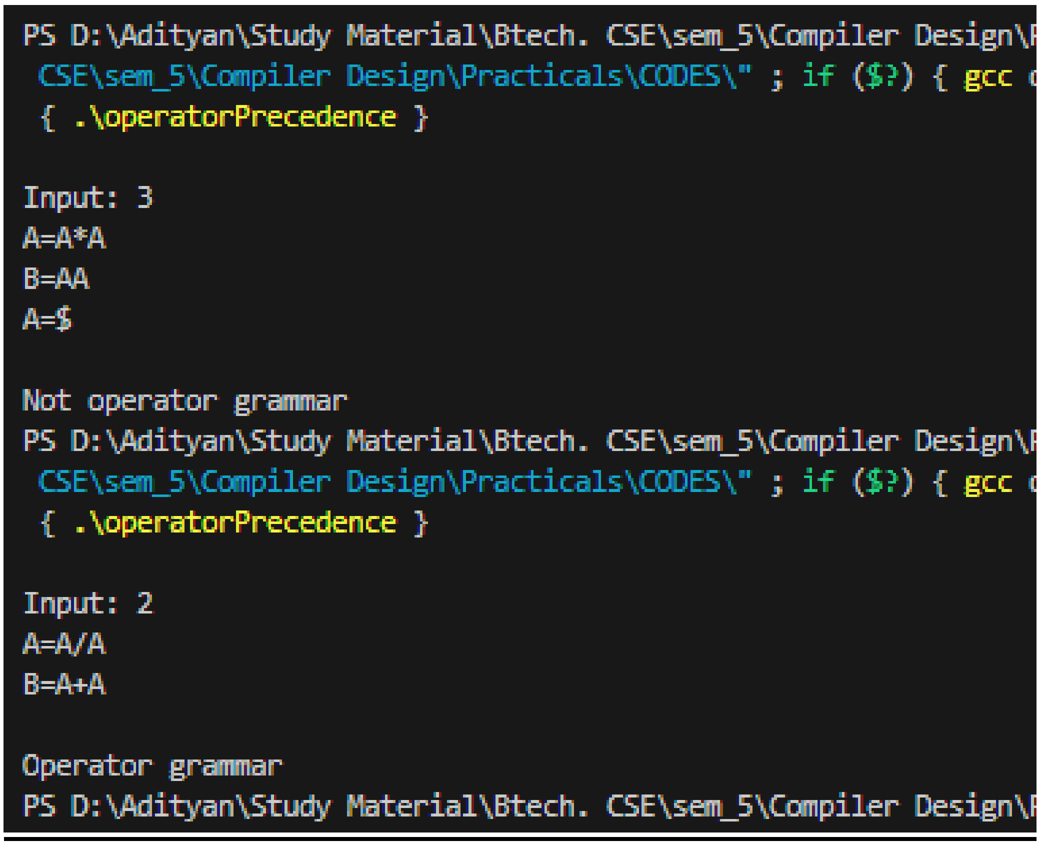
# Experiment-10

**AIM:** Write a program to check whether a grammar is **operator precedence**.

**Program:**

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

// function f to exit from the loop
// if given condition is not true
void f()
{
	printf("Not operator grammar");
	exit(0);
}

void main()
{
	char grm[20][20], c;

	// Here using flag variable,
	// considering grammar is not operator grammar
	int i, n, j = 2, flag = 0;

	printf("\nInput: ");
	scanf("%d", &n);
	for (i = 0; i < n; i++)
		scanf("%s", grm[i]);

	for (i = 0; i < n; i++) {
		c = grm[i][2];

		while (c != '\0') {

			if (grm[i][3] == '+' || grm[i][3] == '-'
                    || grm[i][3] == '*' || grm[i][3] == '/')

                    flag = 1;

                else {

                    flag = 0;
                    f();
                }

                if (c == '$') {
                    flag = 0;
                    f();
                }

                c = grm[i][++j];
            }
        }

        if (flag == 1)
            printf("Operator grammar");
    }
```

**Output:**



```
PS D:\Adityan\Study Material\Btech. CSE\sem_5\Compiler Design\P
 CSE\sem_5\Compiler Design\Practicals\CODES\" ; if ($?) { gcc o
 { .\operatorPrecedence }

Input: 3
A=A*A
B=AA
A=$

Not operator grammar
PS D:\Adityan\Study Material\Btech. CSE\sem_5\Compiler Design\P
 CSE\sem_5\Compiler Design\Practicals\CODES\" ; if ($?) { gcc o
 { .\operatorPrecedence }

Input: 2
A=A/A
B=A+A

Operator grammar
PS D:\Adityan\Study Material\Btech. CSE\sem_5\Compiler Design\P
```