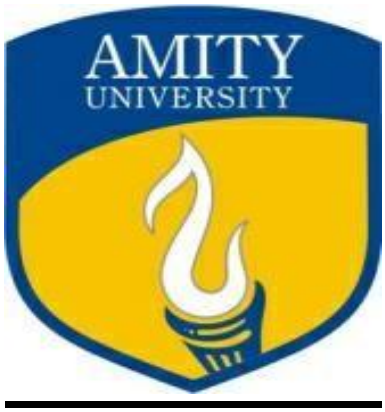
AMITY UNIVERSITY
UTTAR PRADESH

COMPILER CONSTRUCTION[CSE304]

PRACTICAL FILE



AMITY SCHOOL OF ENGINEERING AND TECHNOLOGY AMITY UNIVERSITY CAMPUS, SECTOR-125, NOIDA-201303

Submitted By-
Anchal kumari
A12405218083 (6CSE-3X)

Submitted To-
Dr. Rishi Dutt Sharma

INDEX

SNO	PROGRAM NAME	PERFORM DATE	SUBMISSION DATE	FACULTY SIGNATURE
1	Write a c program to convert infix to post fix notation.	05-01-2021	12/01/2021	

2	Write a program to count the no of tokens in a String.
	i. y=a+b
	ii. y=(a+b);
	iii. int y=(a+b/c * d),
	iv. printf("%d%d%d");
		12/01/2021	19/01/2021	

3	Write a program to find out first and follow of A, where A is grammer or string. i. abc ii. 8cd iii. Zy iv. S->abc | 8cd | zy v. S->ABC | 8cd | zy A-> str B-> f C-> d	19/01/2021	26/01/2021	
4	WAP to find first(S) where S->ABC, A-> a/b/ e, B-> c/ d/ e, C-> xyz/ e Where Symbol "e" is epsilon(null string)	26/01/2021	02/02/2021	
5	Write a program to find out follow of A, where A is grammar. S->a S-> aSb S-> aSbScS S-> aA S-> aAb S-> aAbA/ bBcS	02/02/2021	09/02/2021	

6	Write a program to find out follow of A, where A is grammar containing epsilon. S->ABC A->DEF B-> % C-> % D-> % E-> % F-> % follow of given grammer.	09/02/2021	16/02/2021	
7	Write a program which accepts a regular expression from the user and generates a regular grammar which is equivalent to the R.E. entered by user. The grammar will be printed to a text file, with only one production rule in each line. Also, make sure that all production rules are displayed in compact forms e.g. the production rules: S--> aB, S--> cd S- -> PQ Should be written as S--> aB | cd | PQ And not as three different production rules. Also, there should not be any repetition of production rules.	16/02/2021	23/02/2021	
8	Consider the following grammar: S --> ABC A--> abA | ab B--> b | BC C--> c | cC Following any suitable parsing technique(prefer topdown), design a parser which accepts a string and tells whether the string is accepted by above grammar or not	23/02/2021	02/03/2021	
9	Write a program which accepts a regular grammar with no leftrecursion, and	02/03/2021		

	no nullproduction rules, and then it accepts a string and reports whether the string is accepted by the grammar or not.		09/03/2021	
10	Design a parser which accepts a mathematical expression (containing integers only). If the expression is valid, then evaluate the expression else report that the expression is invalid. [Note: Design first the Grammar and then implement using ShiftReduce parsing technique. Your program should generate an output file clearly showing each step of parsing/evaluation of the intermediate subexpressions.	09/03/2021	16/03/2021	
11	OPEN ENDED EXPERIMENT	20/03/2021	30/03/2021	

EXPERIMENT-1

AIM: Write a c program to convert from infix to post fix notation.

THEORY: We write expression in infix notation, e.g. a - b + c, where operators are used in-between operands. It is easy for us humans to read, write, and speak in infix notation but the same does not go well with computing devices. An algorithm to process infix notation could be difficult and costly in terms of time and space consumption.

This notation style is known as Reversed Polish Notation. In this notation style, the operator is postfixed to the operands i.e., the operator is written after the operands. For example, ab+. This is equivalent to its infix notation a + b.

- Postfix notation is the useful form of intermediate code if the given language is expressions.
- Postfix notation is also called as 'suffix notation' and 'reverse polish'.
- Postfix notation is a linear representation of a syntax tree.
- In the postfix notation, any expression can be written unambiguously without parentheses.
- The ordinary (infix) way of writing the sum of x and y is with operator in the middle: x * y. But in the postfix notation, we place the operator at the right end as xy *.
- In postfix notation, the operator follows the operand.

CODE:

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

char stack[100];
int top = -1;

void push(char x)
{
   stack[++top] = x;
}

char pop()
{
   if(top == -1)
      return -1;
   else
      return stack[top--];
}
```

```
int priority(char x)
{
   if(x == '(')
      return 0;
   if(x == '+' || x == '-')
      return 1;
   if(x == '*' || x == '/')
      return 2;
   return 0;
}

int main()
{
   char exp[100];
   char *e, x;
   printf("Enter the expression : ");
   scanf("%s",exp);
   printf("\n");
   e = exp;

   while(*e != '\0')
   {
      if(isalnum(*e))
         printf("%c ",*e);
      else if(*e == '(')
         push(*e);
      else if(*e == ')')
      {
         while((x = pop()) != '(')
            printf("%c ", x);
      }
      else
      {
         while(priority(stack[top]) >= priority(*e))
            printf("%c ",pop());
         push(*e);
      }
      e++;
   }

   while(top != -1)
   {
      printf("%c ",pop());
   }return 0;
}
```

OUTPUT

1. **(0+1)*+0*1***

```
Enter the expression :  (0+1)*+0*1*

0 1 + * 0 1 * * +

...Program finished with exit code 0
Press ENTER to exit console.
```

2. **((ab*c+(def)+a*d+e)+c) ((a+b)*(c+d)*+ab*c*d**

```
Enter the expression : ((ab*c+(def)+a*d+e)+c)((a+b)*(c+d)*+ab*c*d

a b c * d e f + a d * + e + c + a b + c d + * * a b c * d * + (
Process returned 0 (0x0)   execution time : 73.181 s
Press any key to continue.
```

EXPERIMENT-2

AIM: Write a program to count the no of tokens in a String.

THEORY: Tokens in C is the most important element to be used in creating a program in C. We can define the token as the smallest individual element in C. For `example, we cannot create a sentence without using words; similarly, we cannot create a program in C without using tokens in C. Therefore, we can say that tokens in C is the building block or the basic component for creating a program in C language.

Tokens in C language can be divided into the following categories:

- Keywords in C
- Identifiers in C
- Strings in C
- Operators in C
- Constant in C
- Special Characters in C

CODE:

```
#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <stdbool.h>

bool isValidDelimiter(char ch)

{

   if (ch == ' ' || ch == '+' || ch == '-' || ch == '*' ||

      ch == '/' || ch == ',' || ch == ';' || ch == '>' ||

      ch == '<' || ch == '=' || ch == '(' || ch == ')' ||

      ch == '[' || ch == ']' || ch == '{' || ch == '}')

      return (true);

   return (false);

}


bool isValidSeparator(char ch)

{
```

```
    if (ch == ',' || ch == ';' || ch == '(' || ch == ')' ||
        ch == '[' || ch == ']' || ch == '{' || ch == '}')
        return (true);
    return (false);
}

bool isValidOperator(char ch)
{
    if (ch == '+' || ch == '-' || ch == '*' ||
    ch == '/' || ch == '>' || ch == '<' ||
    ch == '=')
    return (true);
    return (false);
}

bool isvalidIdentifier(char* str)
{
    if (str[0] == '0' || str[0] == '1' || str[0] == '2' ||
    str[0] == '3' || str[0] == '4' || str[0] == '5' ||
    str[0] == '6' || str[0] == '7' || str[0] == '8' ||
    str[0] == '9' || isValidDelimiter(str[0]) == true)
    return (false);
    return (true);
}

bool isValidKeyword(char* str)
{
    if (!strcmp(str, "if") || !strcmp(str, "else") || !strcmp(str, "while") || !strcmp(str, "do") ||
!strcmp(str, "break") || !strcmp(str, "continue") || !strcmp(str, "int")
    || !strcmp(str, "double") || !strcmp(str, "float") || !strcmp(str, "return") || !strcmp(str,   "char")
|| !strcmp(str, "case") || !strcmp(str, "char")
```

```
    || !strcmp(str, "sizeof") || !strcmp(str, "long") || !strcmp(str, "short") || !strcmp(str, "typedef")
|| !strcmp(str, "switch") || !strcmp(str, "unsigned")
    || !strcmp(str, "void") || !strcmp(str, "static") || !strcmp(str, "struct") || !strcmp(str, "goto"))
    return (true);
    return (false);
}

bool isValidInteger(char* str)
{
    int i, len = strlen(str);
    if (len == 0)
    return (false);
    for (i = 0; i < len; i++)
    {
        if (str[i] != '0' && str[i] != '1' && str[i] != '2'&& str[i] != '3' && str[i] != '4' && str[i] !=
'5'
        && str[i] != '6' && str[i] != '7' && str[i] != '8' && str[i] != '9' || (str[i] == '-' && i > 0))
        return (false);
    }
    return (true);
}

bool isRealNumber(char* str)
{
    int i, len = strlen(str);
    bool hasDecimal = false;
    if (len == 0)
    return (false);
    for (i = 0; i < len; i++)
    {
```

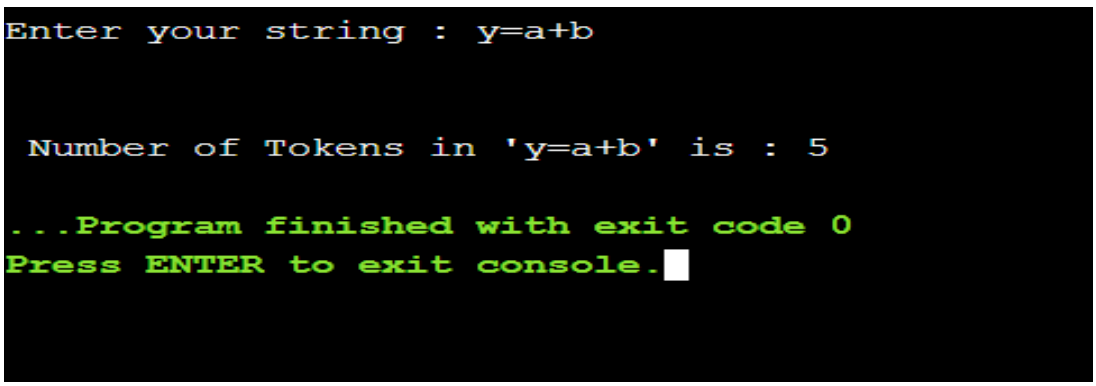
```
        if (str[i] != '0' && str[i] != '1' && str[i] != '2' && str[i] != '3' && str[i] != '4' && str[i]!=
'5' && str[i] != '6' && str[i] != '7' && str[i] != '8'
        && str[i] != '9' && str[i] != '.' || (str[i] == '-' && i > 0))
            return (false);
        if (str[i] == '.')
            hasDecimal = true;
    }
   return (hasDecimal);
}

char* subString(char* str, int left, int right)
{
    int i;
    char* subStr = (char*)malloc( sizeof(char) * (right - left + 2));
    for (i = left; i <= right; i++)
        subStr[i - left] = str[i];
    subStr[right - left + 1] = '\0';
    return (subStr);
}

void findTokens(char* str)
{
    int left = 0, right = 0,count=0;
    int length = strlen(str);
    while (right <= length && left <= right)
    {
        if (isValidDelimiter(str[right]) == false)
        right++;
        if (isValidDelimiter(str[right]) == true && left == right)
        {
```

```
        if (isValidSeparator(str[right]) == true)
        {
            count++;
        }
        if (isValidOperator(str[right]) == true)
        {
            if((str[right]==str[right+1])&&(str[right]!=str[right+2]))
                count--;
            count++;
        }
        right++;
        left = right;
    } else if (isValidDelimiter(str[right]) == true && left != right || (right == length && left
!=right))
    {
        char* subStr = subString(str, left, right - 1);
        if (isValidKeyword(subStr) == true)
        {
            count++;
        }
        else if (isValidInteger(subStr) == true)
        {
            count++;
        }
        else if (isRealNumber(subStr) == true)
        {
            count++;
        }
        else if (isvalidIdentifier(subStr) == true&& isValidDelimiter(str[right - 1]) == false)
        {
```

```
            count++;
        }
        else if (isvalidIdentifier(subStr) == false && isValidDelimiter(str[right - 1]) == false)
        {
            count++;
        }
        left = right;
      }
  }
  printf("\n Number of Tokens in '%s' is : %d",str,count);
  return;
}

int main()
{
   char str[100];
   printf("Enter your string : ");
  gets(str);
   printf("\n");
   findTokens(str);
   return 0;
}
```

OUTPUT

i. y=a+b



ii. y=(a+b);

```
Enter your string : y=a+b;


 Number of Tokens in 'y=a+b;' is : 6

...Program finished with exit code 0
Press ENTER to exit console.
```

iii. int y=(a+b/c * d),

```
Enter your string : int y=(a+b/c*d),


 Number of Tokens in 'int y=(a+b/c*d),' is : 13

...Program finished with exit code 0
Press ENTER to exit console.
```

iv. printf("%d%d%d");

```
Enter your string : printf("%d%d%d");


 Number of Tokens in 'printf("%d%d%d");' is : 5

...Program finished with exit code 0
Press ENTER to exit console.
```

v. for (i=1; i<10; i++);

```
Enter your string : for(i=1; i<10; i++);


 Number of Tokens in 'for(i=1; i<10; i++);' is : 14

...Program finished with exit code 0
Press ENTER to exit console.
```

vi. a==b++---+==j;

```
Enter your string : a==b++--+==j;


 Number of Tokens in 'a==b++--+==j;' is : 9

...Program finished with exit code 0
Press ENTER to exit console.
```

vii. in t == { "%d", x ++y (z-a))}

```
Enter your string : int t=={"%d",x++y(z-a))}


 Number of Tokens in 'int t=={"%d",x++y(z-a))}' is : 16

...Program finished with exit code 0
Press ENTER to exit console.
```

EXPERIMENT-3

AIM: Write a program to find out first and follow of A, where A is grammer or string.

THEORY: An important part of parser table construction is to create first and follow sets. These sets can provide the actual position of any terminal in the derivation. This is done to create the parsing table where the decision of replacing T[A, t] = α with some production rule.

First Set-This set is created to know what terminal symbol is derived in the first position by a non-terminal.

Follow Set-Likewise, we calculate what terminal symbol immediately follows a non-terminal α in production rules. We do not consider what the non-terminal can generate but instead, we see what would be the next terminal symbol that follows the productions of a non-terminal.

Rules For Calculating First Function-

Rule-01:

For a production rule $X \rightarrow \in$,

First(X) = { ∈ }

Rule-02:

For any terminal symbol 'a',

First(a) = { a }

Rule-03:

For a production rule $X \rightarrow Y_1Y_2Y_3$,

Calculating First(X)

If $\in \notin$ First(Y_1), then First(X) = First(Y_1)

If $\in \in$ First(Y_1), then First(X) = { First(Y_1) – ∈ } ∪ First(Y_2Y_3)

Calculating First(Y_2Y_3)

- If $\in \notin$ First(Y_2), then First(Y_2Y_3) = First(Y_2)
- If $\in \in$ First(Y_2), then First(Y_2Y_3) = { First(Y_2) – ∈ } ∪ First(Y_3)

Similarly, we can make expansion for any production rule $X \rightarrow Y_1Y_2Y_3.....Y_n$.

Rules For Calculating Follow Function-

Rule-01:

For the start symbol S, place $ in Follow(S).

Rule-02:

For any production rule A → αB,

Follow(B) = Follow(A)

Rule-03:

For any production rule A → αBβ,

If ∈ ∉ First(β), then Follow(B) = First(β)

If ∈ ∈ First(β), then Follow(B) = { First(β) – ∈ } ∪ Follow(A)

Important Notes-

Note-01:

∈ may appear in the first function of a non-terminal.

∈ will never appear in the follow function of a non-terminal.

Note-02:

Before calculating the first and follow functions, eliminate **Left Recursion** from the grammar, if present.

Note-03:

We calculate the follow function of a non-terminal by looking where it is present on the RHS of a production rule.

C Program To Find First of a Given Grammar

```
#include<stdio.h>

#include<ctype.h>

void Find_First(char[], char);

void Array_Manipulation(char[], char);


int limit;

char production[25][25];
```

```
int main()
{
    char option;
    char ch;
    char array[25];
    int count;
    printf("\nEnter Total Number of Productions:\t");
    scanf("%d", &limit);
    for(count = 0; count < limit; count++)
    {
        printf("\nValue of Production Number [%d]:\t", count + 1);
        scanf("%s", production[count]);
    }
    do
    {
        printf("\nEnter a Value to Find First:\t");
        scanf(" %c", &ch);
        Find_First(array, ch);
        printf("\nFirst Value of %c:\t{ ", ch);
        for(count = 0; array[count] != '\0'; count++)
        {
            printf(" %c ", array[count]);
        }
        printf("}\n");
        printf("To Continue, Press Y:\t");
        scanf(" %c", &option);
    }while(option == 'y' || option == 'Y');
    return 0;
}
```

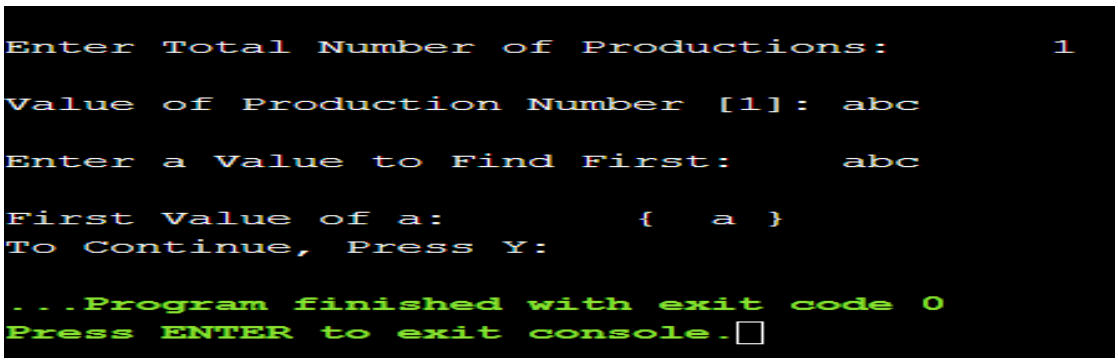
```
void Find_First(char* array, char ch)
{
    int count, j, k;
    char temporary_result[20];
    int x;
    temporary_result[0] = '\0';
    array[0] = '\0';
    if(!(isupper(ch)))
    {
        Array_Manipulation(array, ch);
        return ;
    }
    for(count = 0; count < limit; count++)
    {
        if(production[count][0] == ch)
        {
            if(production[count][2] == '$')
            {
                Array_Manipulation(array, '$');
            }
            else
            {
                j = 2;
                while(production[count][j] != '\0')
                {
                    x = 0;
                    Find_First(temporary_result, production[count][j]);
                    for(k = 0; temporary_result[k] != '\0'; k++)
                    {
                        Array_Manipulation(array,temporary_result[k]);
```

```
                    }
                    for(k = 0; temporary_result[k] != '\0'; k++)
                    {
                        if(temporary_result[k] == '$')
                        {
                            x = 1;
                            break;
                        }
                    }
                    if(!x)
                    {
                        break;
                    }
                    j++;
                }
            }
        }
    }
    return;
}

void Array_Manipulation(char array[], char value)
{
    int temp;
    for(temp = 0; array[temp] != '\0'; temp++)
    {
        if(array[temp] == value)
        {
            return;
        }
```

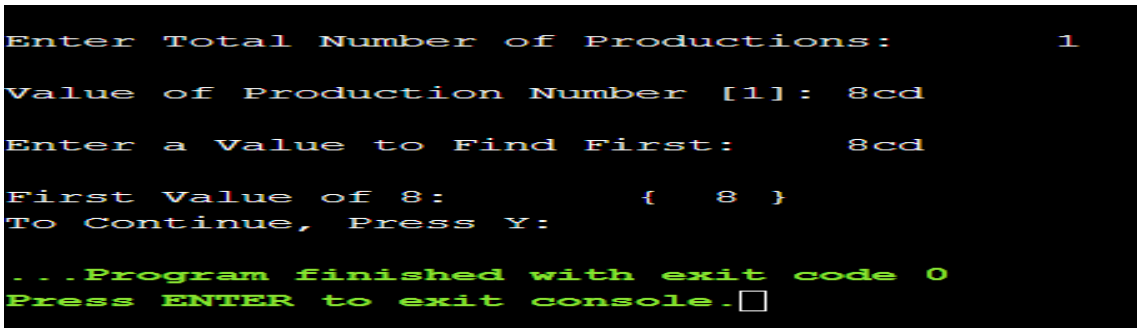
```
    }
    array[temp] = value;
    array[temp + 1] = '\0';
}
```

OUTPUT

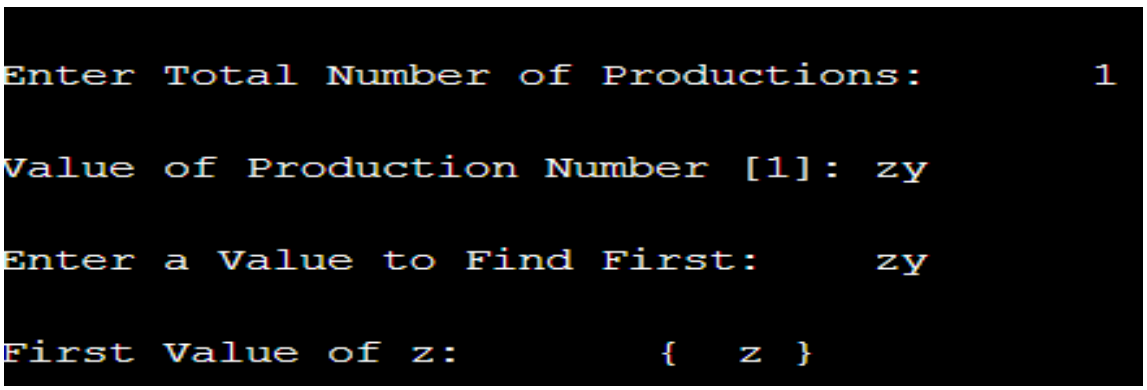
i. abc



ii. 8cd



iii. zy



iv. S->abc | 8cd | zy

```
Enter Total Number of Productions:      3

Value of Production Number [1]: S=abc

Value of Production Number [2]: S=8cd

Value of Production Number [3]: S=zy

Enter a Value to Find First:    S=abc|8cd|zy

First Value of S:       {  a  8  z }
To Continue, Press Y:
```

v. S->ABC | 8cd | zy
A-> str
B-> f
C-> d

```
Enter Total Number of Productions:      6

Value of Production Number [1]: S=ABC

Value of Production Number [2]: S=8cd

Value of Production Number [3]: S=zy

Value of Production Number [4]: A=str

Value of Production Number [5]: B=f

Value of Production Number [6]: C=d

Enter a Value to Find First:    S=ABC|8cd|zy

First Value of S:       {  s  8  z }
To Continue, Press Y:

...Program finished with exit code 0
Press ENTER to exit console.
```

C Program to Find Follow of a Grammar

```
#include<stdio.h>

#include<ctype.h>

#include<string.h>


int limit, x = 0;

char production[10][10], array[10];


void find_first(char ch);

void find_follow(char ch);

void Array_Manipulation(char ch);
```

```
int main()
{
    int count;
    char option, ch;
    printf("\nEnter Total Number of Productions:\t");
    scanf("%d", &limit);
    for(count = 0; count < limit; count++)
    {
        printf("\nValue of Production Number [%d]:\t", count + 1);
        scanf("%s", production[count]);
    }
    do
    {
        x = 0;
        printf("\nEnter production Value to Find Follow:\t");
        scanf(" %c", &ch);
        find_follow(ch);
        printf("\nFollow Value of %c:\t{ ", ch);
        for(count = 0; count < x; count++)
        {
            printf("%c ", array[count]);
        }
        printf("}\n");
        printf("To Continue, Press Y:\t");
        scanf(" %c", &option);
    }while(option == 'y' || option == 'Y');
    return 0;
}

void find_follow(char ch)
```

```
{
    int i, j;
    int length = strlen(production[i]);
    if(production[0][0] == ch)
    {
        Array_Manipulation('$');
    }
    for(i = 0; i < limit; i++)
    {
        for(j = 2; j < length; j++)
        {
            if(production[i][j] == ch)
            {
                if(production[i][j + 1] != '\0')
                {
                    find_first(production[i][j + 1]);
                }
                if(production[i][j + 1] == '\0' && ch != production[i][0])
                {
                    find_follow(production[i][0]);
                }
            }
        }
    }
}

void find_first(char ch)
{
    int i, k;
    if(!(isupper(ch)))
```

```
    {
        Array_Manipulation(ch);
    }
    for(k = 0; k < limit; k++)
    {
        if(production[k][0] == ch)
        {
            if(production[k][2] == '$')
            {
                find_follow(production[i][0]);
            }
            else if(islower(production[k][2]))
            {
                Array_Manipulation(production[k][2]);
            }
            else
            {
                find_first(production[k][2]);
            }
        }
    }
}

void Array_Manipulation(char ch)
{
    int count;
    for(count = 0; count <= x; count++)
    {
        if(array[count] == ch)
        {
```

```
            return;
        }
    }
    array[x++] = ch;
}
```

OUTPUT

i. abc

```
Enter Total Number of Productions:      1

Value of Production Number [1]: abc

Enter production Value to Find Follow:  abc

Follow Value of a:      { $ }
To Continue, Press Y:

...Program finished with exit code 0
Press ENTER to exit console.
```

ii. 8cd

```
Enter Total Number of Productions:      1

Value of Production Number [1]: 8cd

Enter production Value to Find Follow:  8cd

Follow Value of 8:      { $ }
To Continue, Press Y:

...Program finished with exit code 0
Press ENTER to exit console.
```

iii. zy

```
Enter Total Number of Productions:      1

Value of Production Number [1]: zy

Enter production Value to Find Follow:  zy

Follow Value of z:      { $ }
To Continue, Press Y:
Enter production Value to Find Follow:
```

iv. S->abc | 8cd | zy

```
Enter Total Number of Productions:       3

Value of Production Number [1]: S=abc

Value of Production Number [2]: S=8cd

Value of Production Number [3]: S=zy

Enter production Value to Find Follow:   S=abc|8cd|zy

Follow Value of S:       { $ }
To Continue, Press Y:
```

vi. S->ABC | 8cd | zy
A-> str
B-> f
C-> d

```
Enter Total Number of Productions:       6

Value of Production Number [1]: S=ABC

Value of Production Number [2]: S=8cd

Value of Production Number [3]: S=zy

Value of Production Number [4]: A=str

Value of Production Number [5]: B=f

Value of Production Number [6]: C=d

Enter production Value to Find Follow:   S=ABC|8cd|zy

Follow Value of S:       { $ }
To Continue, Press Y:
```

EXPERIMENT-4

AIM: Write a program to find out first of A, where A is grammar containing epsilon.

THEORY: An important part of parser table construction is to create first and follow sets. These sets can provide the actual position of any terminal in the derivation. This is done to create the parsing table where the decision of replacing T[A, t] = α with some production rule.

First Set-This set is created to know what terminal symbol is derived in the first position by a non-terminal.

CODE-

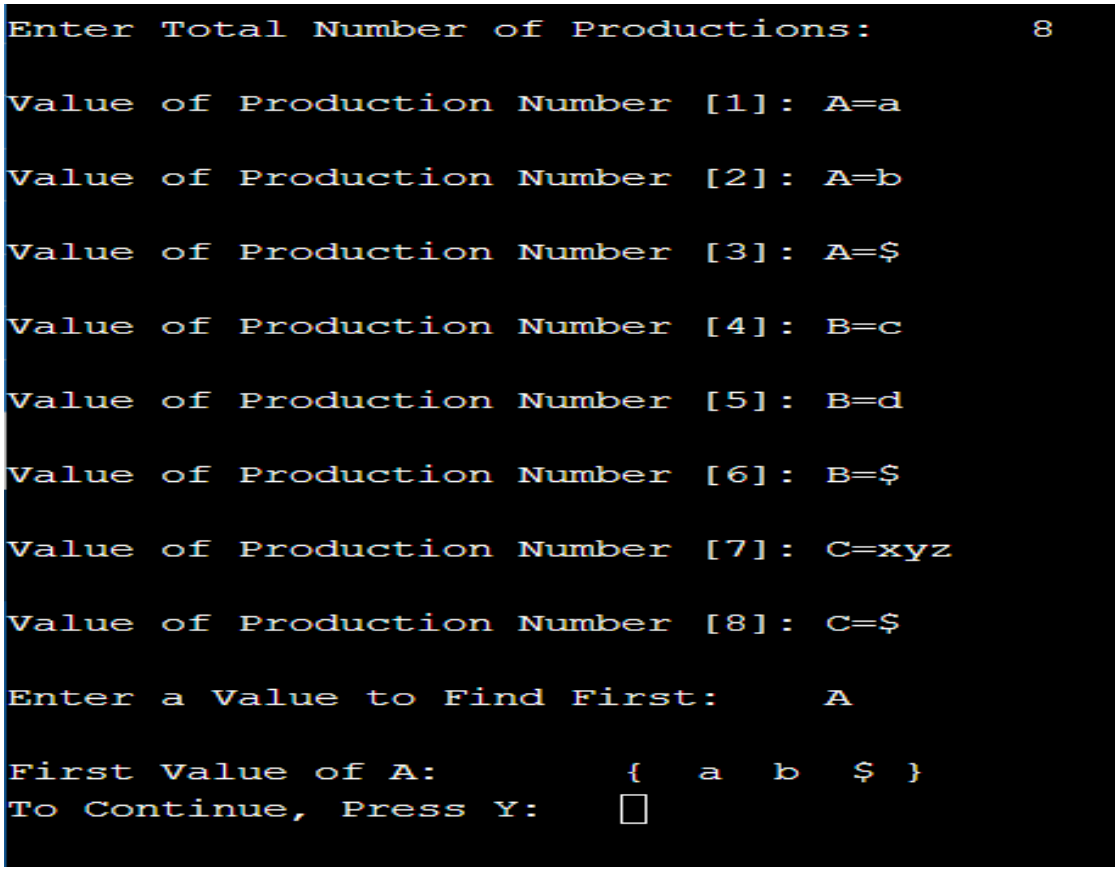
```
#include<stdio.h>
#include<ctype.h>

void Find_First(char[], char);
void Array_Manipulation(char[], char);

int limit;
char production[25][25];

int main()
{
    char option;
    char ch;
    char array[25];
    int count;
    printf("\nEnter Total Number of Productions:\t");
    scanf("%d", &limit);
    for(count = 0; count < limit; count++)
    {
        printf("\nValue of Production Number [%d]:\t", count + 1);
        scanf("%s", production[count]);
```

```
    }
    do
    {
        printf("\nEnter a Value to Find First:\t");
        scanf(" %c", &ch);
        Find_First(array, ch);
        printf("\nFirst Value of %c:\t{ ", ch);
        for(count = 0; array[count] != '\0'; count++)
        {
            printf(" %c ", array[count]);
        }
        printf("}\n");
        printf("To Continue, Press Y:\t");
        scanf(" %c", &option);
    }while(option == 'y' || option == 'Y');
    return 0;
}

void Find_First(char* array, char ch)
{
    int count, j, k;
    char temporary_result[20];
    int x;
    temporary_result[0] = '\0';
    array[0] = '\0';
    if(!(isupper(ch)))
    {
        Array_Manipulation(array, ch);
        return ;
    }
```

```
for(count = 0; count < limit; count++)
{
    if(production[count][0] == ch)
    {
        if(production[count][2] == '$')
        {
            Array_Manipulation(array, '$');
        }
        else
        {
            j = 2;
            while(production[count][j] != '\0')
            {
                x = 0;
                Find_First(temporary_result, production[count][j]);
                for(k = 0; temporary_result[k] != '\0'; k++)
                {
                    Array_Manipulation(array,temporary_result[k]);
                }
                for(k = 0; temporary_result[k] != '\0'; k++)
                {
                    if(temporary_result[k] == '$')
                    {
                        x = 1;
                        break;
                    }
                }
                if(!x)
                {
                    break;
```

```
                    }
                    j++;
                }
            }
        }
    }
    return;
}
void Array_Manipulation(char array[], char value)
{
    int temp;
    for(temp = 0; array[temp] != '\0'; temp++)
    {
        if(array[temp] == value)
        {
            return;
        }
    }
    array[temp] = value;
    array[temp + 1] = '\0';
}
```

OUTPUT-


Enter Total Number of Productions: 8
Value of Production Number [1]: A=a
Value of Production Number [2]: A=b
Value of Production Number [3]: A=$
Value of Production Number [4]: B=c
Value of Production Number [5]: B=d
Value of Production Number [6]: B=$
Value of Production Number [7]: C=xyz
Value of Production Number [8]: C=$
Enter a Value to Find First: A
First Value of A: { a b $ }
To Continue, Press Y:

EXPERIMENT-5

AIM: Write a program to find out follow of A, where A is grammar.

THEORY: An important part of parser table construction is to create first and follow sets. These sets can provide the actual position of any terminal in the derivation. This is done to create the parsing table where the decision of replacing T[A, t] = α with some production rule.

Follow Set-Likewise, we calculate what terminal symbol immediately follows a non-terminal α in production rules. We do not consider what the non-terminal can generate but instead, we see what would be the next terminal symbol that follows the productions of a non-terminal.

C Program To Find Follow of a Given Grammar

```
#include<stdio.h>
#include<string.h>
int n,m=0,p,i=0,j=0;
char a[10][10],f[10];
void follow(char c);
void first(char c);
int main()
{
 int i,z;
 char c,ch;
 printf("Enter the no.of productions:");
 scanf("%d",&n);
 printf("Enter the productions(epsilon=$):\n");
 for(i=0;i<n;i++)
  scanf("%s%c",a[i],&ch);

 do
 {
  m=0;
  printf("Enter the element whose FOLLOW is to be found:");
```

```
 scanf("%c",&c);
 follow(c);
 printf("FOLLOW(%c) = { ",c);
 for(i=0;i<m;i++)
  printf("%c ",f[i]);
 printf(" }\n");
 printf("Do you want to continue(0/1)?");
 scanf("%d%c",&z,&ch);
 }
 while(z==1);
}
void follow(char c)
{

 if(a[0][0]==c)f[m++]='$';
 for(i=0;i<n;i++)
 {
  for(j=2;j<strlen(a[i]);j++)
  {
   if(a[i][j]==c)
   {
    if(a[i][j+1]!='\0')first(a[i][j+1]);

    if(a[i][j+1]=='\0'&&c!=a[i][0])
     follow(a[i][0]);

   }
  }
 }
```

```
}
void first(char c)
{
    int k;
            if(!(isupper(c)))f[m++]=c;
            for(k=0;k<n;k++)
            {
            if(a[k][0]==c)
            {
            if(a[k][2]=='$') follow(a[i][0]);
            else if(islower(a[k][2]))f[m++]=a[k][2];
            else first(a[k][2]);
            }
            }
}
```

OUTPUT:

```
Enter the no.of productions:7
Enter the productions(epsilon=$):
S=a
S=aSb
S=aSbScS
S=aA
S=aAb
S=aAbA
S=bBcS
Enter the element whose FOLLOW is to be found:S
FOLLOW(S) = { $ b b c  }
Do you want to continue(0/1)?0


...Program finished with exit code 0
Press ENTER to exit console.
```

EXPERIMENT-6

AIM: Write a program to find out follow of A, where A is grammar containing epsilon.

THEORY: An important part of parser table construction is to create first and follow sets. These sets can provide the actual position of any terminal in the derivation. This is done to create the parsing table where the decision of replacing T[A, t] = α with some production rule.

Follow Set-Likewise, we calculate what terminal symbol immediately follows a non-terminal α in production rules. We do not consider what the non-terminal can generate but instead, we see what would be the next terminal symbol that follows the productions of a non-terminal.

C Program To Find Follow of a Given Grammar

```
#include<stdio.h>
#include<string.h>
int n,m=0,p,i=0,j=0;
char a[10][10],f[10];
void follow(char c);
void first(char c);
int main()
{
 int i,z;
 char c,ch;
 printf("Enter the no.of productions:");
 scanf("%d",&n);
 printf("Enter the productions(epsilon=$):\n");
 for(i=0;i<n;i++)
  scanf("%s%c",a[i],&ch);

 do
 {
  m=0;
```

```
 printf("Enter the element whose FOLLOW is to be found:");

 scanf("%c",&c);
 follow(c);
 printf("FOLLOW(%c) = { ",c);
 for(i=0;i<m;i++)
  printf("%c ",f[i]);
 printf(" }\n");
 printf("Do you want to continue(0/1)?");
 scanf("%d%c",&z,&ch);
 }
 while(z==1);
}
void follow(char c)
{

 if(a[0][0]==c)f[m++]='$';
 for(i=0;i<n;i++)
 {
  for(j=2;j<strlen(a[i]);j++)
  {
   if(a[i][j]==c)
   {
    if(a[i][j+1]!='\0')first(a[i][j+1]);

    if(a[i][j+1]=='\0'&&c!=a[i][0])
     follow(a[i][0]);  }
  }
 }
}
```

```
void first(char c)
{
    int k;
            if(!(isupper(c)))f[m++]=c;
            for(k=0;k<n;k++)
            {
            if(a[k][0]==c)
            {
            if(a[k][2]=='$') follow(a[i][0]);
            else if(islower(a[k][2]))f[m++]=a[k][2];
            else first(a[k][2]);
            }
            }
}
```

OUTPUT:

```
Enter the no.of productions:7
Enter the productions(epsilon=$):
S=ABC
A=DEF
B=$
C=$
D=$
E=$
F=$
Enter the element whose FOLLOW is to be found:S
FOLLOW(S) = { $  }
Do you want to continue(0/1)?1
Enter the element whose FOLLOW is to be found:A
FOLLOW(A) = { $  }
```

EXPERIMENT-7

AIM: Write a program which accepts a regular expression from the user and generates a regular grammar which is equivalent to the R.E. entered by user. The grammar will be printed to a text file, with only one production rule in each line. Also, make sure that all production rules are displayed in compact forms e.g. the production rules: S--> aB, S--> cd S- -> PQ Should be written as S--> aB | cd | PQ And not as three different production rules. Also, there should not be any repetition of production rules.

THEORY: Regular expressions have the capability to express finite languages by defining a pattern for finite strings of symbols. The grammar defined by regular expressions is known as regular grammar. The language defined by regular grammar is known as regular language. Regular expression is an important notation for specifying patterns. Each pattern matches a set of strings, so regular expressions serve as names for a set of strings. Programming language tokens can be described by regular languages. The specification of regular expressions is an example of a recursive definition. Regular languages are easy to understand and have efficient implementation.

There are a number of algebraic laws that are obeyed by regular expressions, which can be used to manipulate regular expressions into equivalent forms.

CODE:

```
#include <bits/stdc++.h>
using namespace std;

int main() {

    string str[50];
    string s[50];
    int n,i,j,k,index;
    int a [26]={0};
```

```
cout<<"how many production rule you want to enter? "<<endl;
cin>>n;

cout<<"Enter production :"<<endl;
 k=-1;

for(i=0;i<n;i++)
{
   if(i==0) getline(cin,str[i]);

   cout<<"["<<i+1<<"] : ";
   getline(cin,str[i]);

   if(int(str[i][0])<65 || int(str[i][0])>91)
   {
      cout<<"variables cannot be smallcase "<<endl;
      return 0;

   }

   a[int(str[i][0]) -65]++;
   if(a[int(str[i][0]) -65]>1)
   {
      j=0;
     for(j=0;j<=k;j++)
      {
         if(s[j][0]== str[i][0])
         {
            index=j;

            break;
         }
      }
```

```
            if(s[index].find(str[i].substr(2))== string::npos)
            {
               string st = str[i].substr(2);
               s[index] += '/' + st;
            }

         }
         else
         {
            ++k;
            s[k] = str[i];

         }


   }
   cout<<"Input you entered"<<endl;
    for(i=0;i<n;i++)
   {
      cout<<"["<<i+1<<"] ";
      cout<<str[i]<<endl;
   }
   cout<<"Output: "<<endl;
   n=k;
    for(k=0;k<=n;k++)
   {
      cout<<"["<<k+1<<"] ";
      cout<<s[k]<<endl;
   }
return 0;
}
```

OUTPUT:

```
how many production rule you want to enter?
3
Enter production :
[1] : S=aB
[2] : S=cd
[3] : S=PQ
Input you entered
[1] S=aB
[2] S=cd
[3] S=PQ
Output:
[1] S=aB/cd/PQ


...Program finished with exit code 0
Press ENTER to exit console.
```

EXPERIMENT-8

AIM: Consider the following grammar: S --> ABC A--> abA | ab B--> b | BC C--> c | cC Following any suitable parsing technique(prefer topdown), design a parser which accepts a string and tells whether the string is accepted by above grammar or not.

THEORY:

Parser is that phase of compiler which takes token string as input and with the help of existing grammar, converts it into the corresponding parse tree. Parser is also known as Syntax Analyzer.

Types of Parser:

Parser is mainly classified into 2 categories: Top-down Parser, and Bottom-up Parser. These are explained as following below.

1. Top-down Parser:

Top-down parser is the parser which generates parse for the given input string with the help of grammar productions by expanding the non-terminals i.e. it starts from the start symbol and ends on the terminals. It uses left most derivation.

Further Top-down parser is classified into 2 types: Recursive descent parser, and Non-recursive descent parser.

(i). Recursive descent parser:

It is also known as Brute force parser or the with backtracking parser. It basically generates the parse tree by using brute force and backtracking.

(ii). Non-recursive descent parser:

It is also known as LL(1) parser or predictive parser or without backtracking parser or dynamic parser. It uses parsing table to generate the parse tree instead of backtracking.

2. Bottom-up Parser:

Bottom-up Parser is the parser which generates the parse tree for the given input string with the help of grammar productions by compressing the non-terminals i.e. it starts from non-terminals and ends on the start symbol. It uses reverse of the right most derivation.

Further Bottom-up parser is classified into 2 types: LR parser, and Operator precedence parser.

(i). LR parser:

LR parser is the bottom-up parser which generates the parse tree for the given string by using unambiguous grammar. It follow reverse of right most derivation.

LR parser is of 4 types:

(a). LR(0)

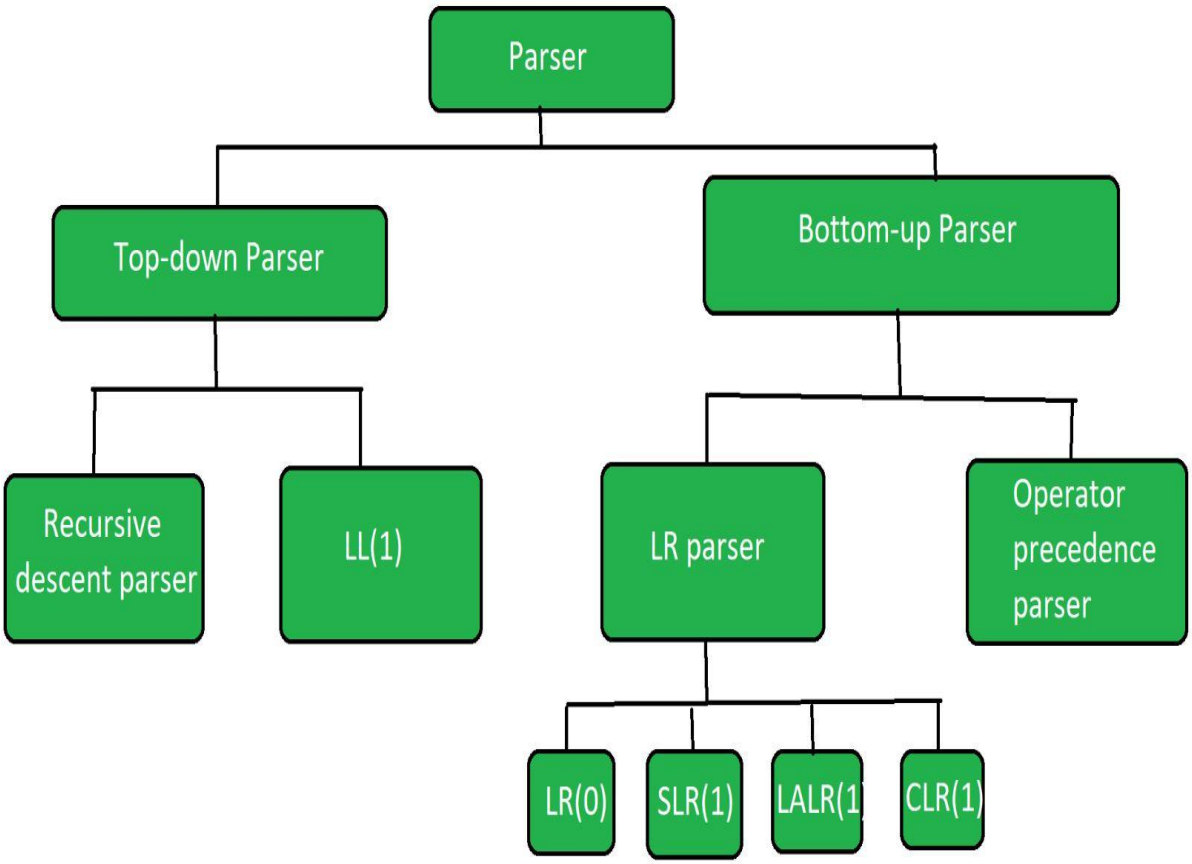
(b). SLR(1)

(c). LALR(1)

(d). CLR(1)

(ii). Operator precedence parser:

It generates the parse tree form given grammar and string but the only condition is two consecutive non-terminals and epsilon never appear in the right-hand side of any production.



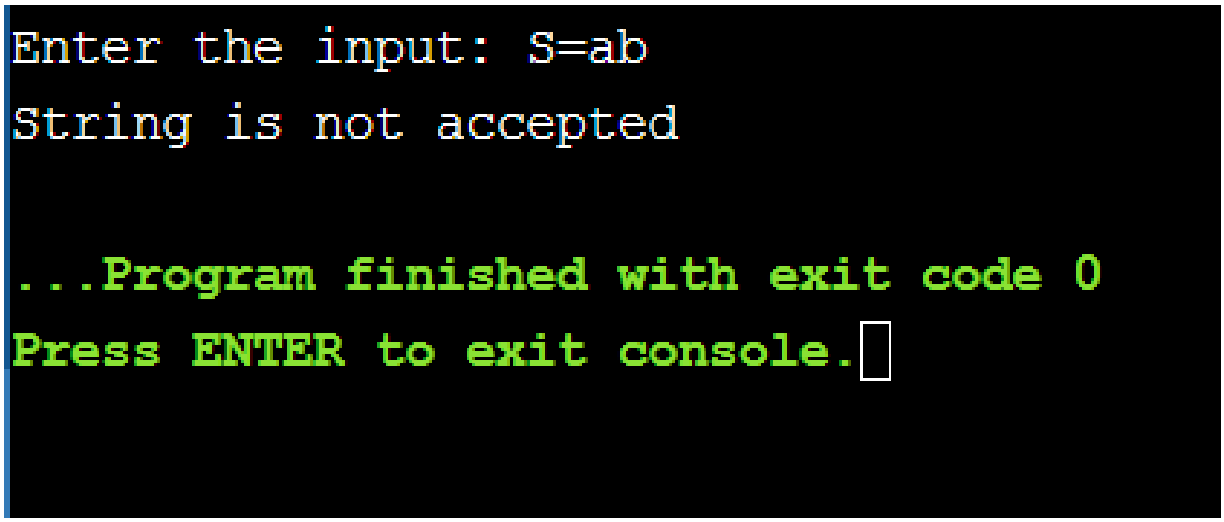
CODE:

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

void S();
void A();
void B();
void C();

char input[100];
int i=0, error=0, bcount=0, ccount=0;

int main()
{
   printf("Enter the input: ");
   scanf("%s", &input);
   S();
   if(error==0)
   {
      printf("String is accepted");
   }
   else
   {
      printf("String is not accepted");
   }
}

void S()
{
   A();
   B();
   C();
}
```

```
void A()
{
   error=1;
   while(input[i]=='a' && input[i+1]=='b')
   {
      i+=2;
      error=0;
   }
}

void B()
{
   error=1;
   while(input[i]=='b')
   {
      i++;
      error=0;
      bcount++;
   }
}

void C()
{
   error=1;
   while(input[i]=='c')
   {
      i++;
      error=0;
      ccount++;
   }
   if(input[i]=='\0' && ccount>=bcount)
   {
```

```
        error=0;
    }
    else
    {
        error=1;
    }
}
```

OUTPUT:



EXPERIMENT-9

AIM: Write a program which accepts a regular grammar with no left recursion, and no null production rules, and then it accepts a string and reports whether the string is accepted by the grammar or not.

THEORY:

Problem with Left Recursion:
If a left recursion is present in any grammar then, during parsing in the the syntax analysis part of compilation there is a chance that the grammar will create infinite loop. This is because at every time of production of grammar S will produce another S without checking any condition.

Algorithm to Remove Left Recursion with an example:
Suppose we have a grammar which contains left recursion:
S-->S a / S b / c / d

1. Check if the given grammar contains left recursion, if present then separate the production and start working on it.
   In our example,

   S-->S a/ S b /c / d
2. Introduce a new nonterminal and write it at the last of every terminal. We produce a new nonterminal S'and write new production as,
   S-->cS' / dS'
3. Write newly produced nonterminal in LHS and in RHS it can either produce or it can produce new production in which the terminals or non terminals which followed the previous LHS will be replaced by new nonterminal at last.
   S'-->? / aS' / bS'

   So after conversion the new equivalent production is

   S-->cS' / dS'

   S'-->? / aS' / bS'

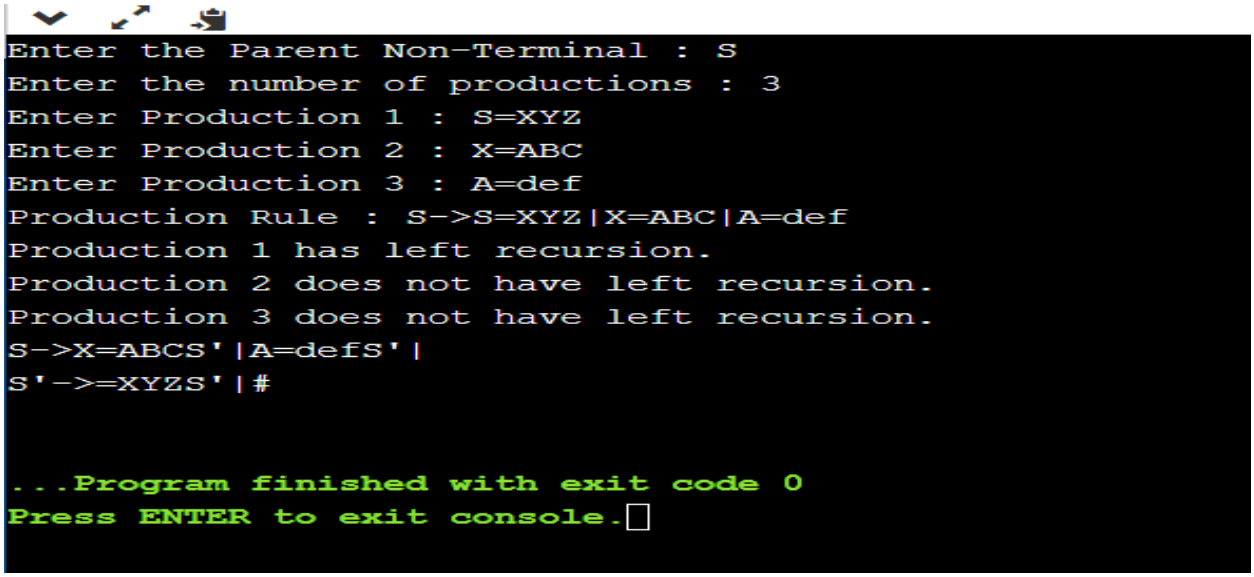
CODE:

```
#include<iostream>
#include<string>
using namespace std;
int main()
{  string ip,op1,op2,temp;
   int sizes[10] = {};
```

```
char c;
int n,j,l;
cout<<"Enter the Parent Non-Terminal : ";
cin>>c;
ip.push_back(c);
op1 += ip + "\'->";
ip += "->";
op2+=ip;
cout<<"Enter the number of productions : ";
cin>>n;
for(int i=0;i<n;i++)
{   cout<<"Enter Production "<<i+1<<" : ";
    cin>>temp;
    sizes[i] = temp.size();
    ip+=temp;
    if(i!=n-1)
        ip += "|";
}
cout<<"Production Rule : "<<ip<<endl;
for(int i=0,k=3;i<n;i++)
{
    if(ip[0] == ip[k])
    {
        cout<<"Production "<<i+1<<" has left recursion."<<endl;
        if(ip[k] != '#')
        {
            for(l=k+1;l<k+sizes[i];l++)
                op1.push_back(ip[l]);
            k=l+1;
            op1.push_back(ip[0]);
            op1 += "\'|";
        }
    }
```

```
        else
        {
            cout<<"Production "<<i+1<<" does not have left recursion."<<endl;
            if(ip[k] != '#')
            {
                for(j=k;j<k+sizes[i];j++)
                    op2.push_back(ip[j]);
                k=j+1;
                op2.push_back(ip[0]);
                op2 += "\'|";
            }
            else
            {
                op2.push_back(ip[0]);
                op2 += "\'";
            }}}
    op1 += "#";
    cout<<op2<<endl;
    cout<<op1<<endl;
    return 0;
}
```

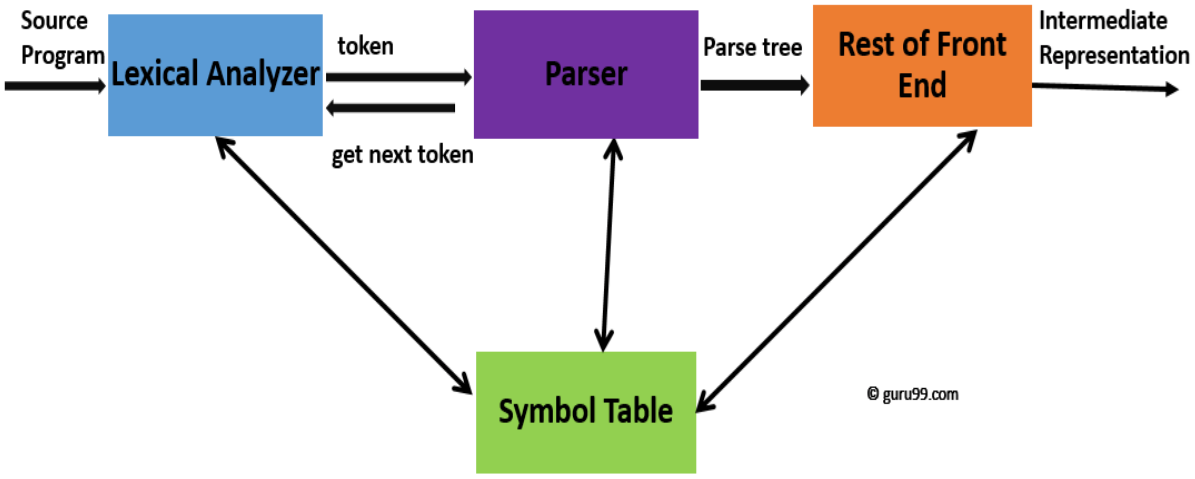
OUTPUT:



EXPERIMENT-10

AIM: Design a parser which accepts a mathematical expression (containing integers only). If the expression is valid, then evaluate the expression else report that the expression is invalid.

THEORY:



CODE:

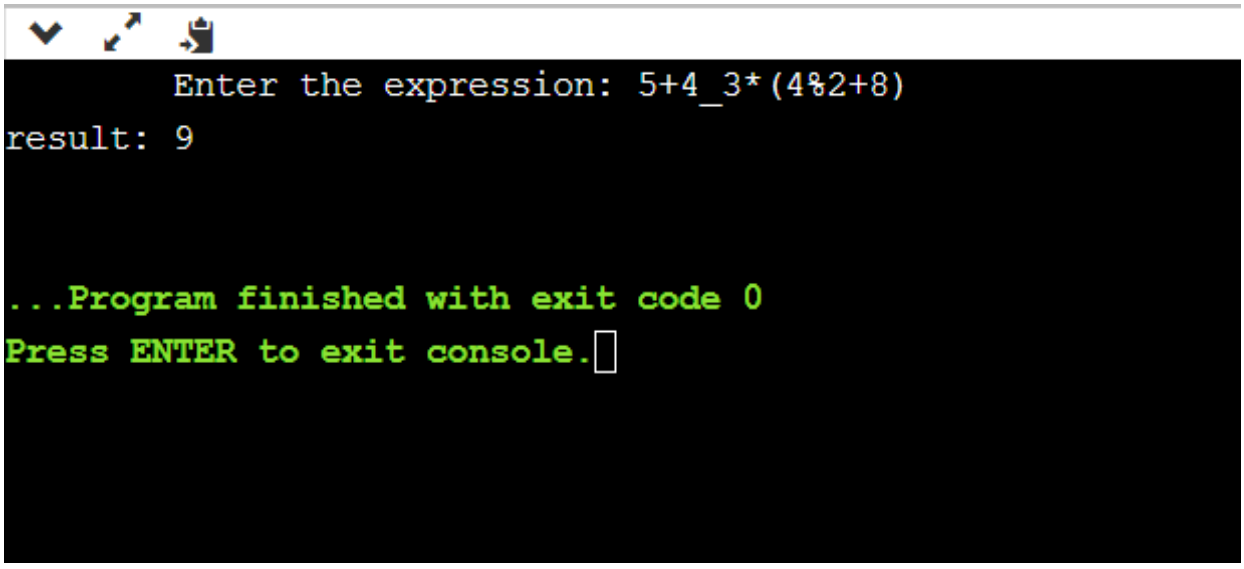
```
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
typedef int findint_t;
findint_t expr(void);

char token;
void error(const char *msg) {
  fputs(msg, stderr);
  exit(1);
}
void match(char expected) {
  if (token == expected) {
     token = getchar();
     return;
```

```
    }

    fprintf(stderr, "Expected %c, got %c", expected, token);
    exit(1);
}


findint_t factor(void) {
    findint_t value;

    if (token == '(') {
        match('(');
        value = expr();
        match(')');

    } else if (isdigit(token) || token == '+' || token == '-') {
        ungetc(token, stdin);
        scanf("%d", &value);
        token = getchar();
    } else {
        error("\nThis is Not Accepted");
    }

    return value;
}

findint_t term(void) {
    findint_t value = factor();

    while (token == '*' || token == '/') {
        switch(token) {
        case '*':
            match('*');
```

```
            value *= factor();
            break;

        case '/':
            match('/');
            value /= factor();
            break;

        default:
            error("bad term");
        }
    }

    return value;
}

findint_t expr() {
    findint_t value = term();

    if (token == '+' || token == '-') {
        switch(token) {
        case '+':
            match('+');
            value += term();
            break;

        case '-':
            match('-');
            value -= term();
            break;
        default:
            error("bad expression");
```

```
    }
  }

  return value;
}

int main(void) {

  printf("\tEnter the expression: ");
  token = getchar();
  findint_t result = expr();
  printf("result: %d\n", result);

  return 0;
}
```

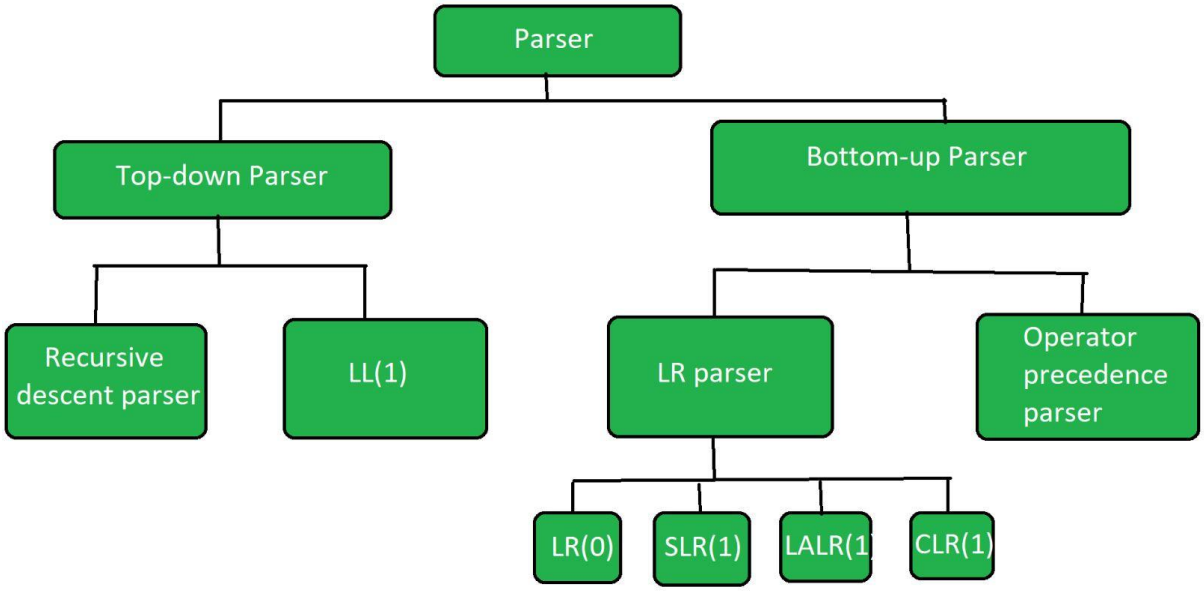
OUTPUT:



OPEN ENDED EXPERIMENT

Problem statement: Designing of various type of parser

Theory: **Parser** is that phase of compiler which takes token string as input and with the help of existing grammar, converts it into the corresponding parse tree. Parser is also known as Syntax Analyzer.



Types of Parser:
Parser is mainly classified into 2 categories: Top-down Parser, and Bottom-up Parser. These are explained as following below.

Top-down Parser:
Top-down parser is the parser which **generates parse for the given input string** with the help of grammar productions by expanding the non-terminals i.e. it starts from the start symbol and ends on the terminals. It uses left most derivation.
Further Top-down parser is classified into 2 types: Recursive descent parser, and Non-recursive descent parser.

Recursive descent parser:
It is also known as Brute force parser or the with backtracking parser. It basically generates the parse tree by using brute force and backtracking.

Non-recursive descent parser:
It is also known as LL(1) parser or predictive parser or without backtracking parser or dynamic parser. It uses parsing table to generate the parse tree instead of backtracking.

Bottom-up Parser:
Bottom-up Parser is the parser which generates the parse tree for the given input string with the help of grammar productions by compressing the non-terminals i.e. it starts from non-terminals and ends on the start symbol. It uses reverse of the right most derivation.
Further Bottom-up parser is classified into 2 types: LR parser, and Operator precedence parser.

LR parser:
LR parser is the bottom-up parser which generates the parse tree for the given string by using unambiguous grammar. It follow reverse of right most derivation.
LR parser is of 4 types:

(a). LR(0)
(b). SLR(1)
(c). LALR(1)
(d). CLR(1)

Operator precedence parser:
It generates the parse tree form given grammar and string but the only condition is two consecutive non-terminals and epsilon never appear in the right-hand side of any production.
Operator precedence grammar is kinds of shift reduce parsing method. It is applied to a small class of operator grammars.

A grammar is said to be operator precedence grammar if it has two properties:

- No R.H.S. of any production has a∈.
- No two non-terminals are adjacent.

Operator precedence can only established between the terminals of the grammar. It ignores the non-terminal.

a ⋗ b means that terminal "a" has the higher precedence than terminal "b".

a ⋖ b means that terminal "a" has the lower precedence than terminal "b".

a ≐ b means that the terminal "a" and "b" both have same precedence.

Precedence table:

	+	*	(	)	id	$
+	⋗	⋖	⋖	⋗	⋖	⋗
*	⋗	⋗	⋖	⋗	⋖	⋗
(	⋖	⋖	⋖	≐	⋖	X
)	⋗	⋗	X	⋗	X	⋗
id	⋗	⋗	X	⋗	X	⋗
$	⋖	⋖	⋖	X	⋖	X

Parsing Action

- Both end of the given input string, add the $ symbol.
- Now scan the input string from left right until the ⋗ is encountered.
- Scan towards left over all the equal precedence until the first left most ⋖ is encountered.
- Everything between left most ⋖ and right most ⋗ is a handle.
- $ on $ means parsing is successful.

CODE:

```
#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;

   printf("\nANCHAL\nA12405218083\n6CSE-3X");
   // Here using flag variable,
   // considering grammar is not operator grammar
   int i, n, j = 2, flag = 0;

   // taking number of productions from user
   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: