AMITY SCHOOL OF ENGINEERING AND TECHNOLOGY

COMPILER CONSTRUCTION LAB FILE

CSE 304

BACHELOR OF TECHNOLOGY

COMPUTER SCIENCE AND ENGINEERING

Submitted by:

Devansh Srivastav - 6CSE1Y - A2305218053

Submitted to: Dr. Prabhishek Singh

AMITY UNIVERSITY UTTAR PRADESH

TABLE OF CONTENTS

13.	WAP To Find Left Recursion and Remove it.
14.	WAP To Find Left Factoring and Remove it.
15.	WAP to calculate First and Follow sets of given grammar.
16.	Write a program for Recursive Descent Calculator.
17.	WAP to generate LL(1) parsing table.
18.	WAP that recognizes different types of English words.
19.	Consider the following grammar: S --> ABC A--> abA | ab B--> b | bC C--> c | cC Following any suitable parsing technique (prefer top-down), WAP to design a parser which accepts a string and tells whether the string is accepted by above grammar or not.
20.	WAP to check whether given grammar is operator grammar or not.
21.	WAP which accepts a regular grammar with no left-recursion, and no null-production rules, and then it accepts a string and checks whether a string belongs to the grammar or not.
22.	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.
23.	WAP to check whether the given production is a reduced production or not in LR(0) canonical item.
24.	WAP to find out all the LR(0) canonical items for a given grammar.
25.	WAP to find out all the LR(1) canonical set of items for a given grammar.

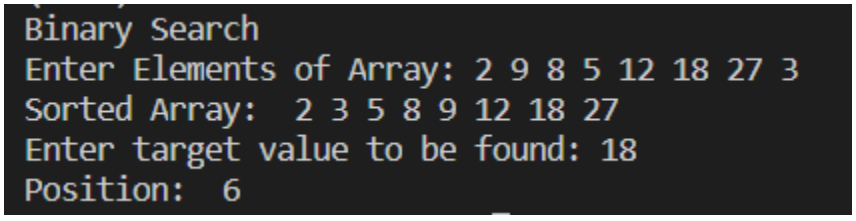
Experiment 1

Aim:
WAP on binary search.

Code:

```
def binary_search(arr, low, high, x):
    if high < low:
        return 'Not Found'
    mid = (low + high) // 2
    if x == arr[mid]:
        return mid
    elif x < arr[mid]:
        return binary_search(arr, low, mid-1, x)
    else:
        return binary_search(arr, mid+1, high, x)

def main():
    print('Binary Search')
    arr = list(map(int, input('Enter Elements of Array: ').strip().split()))
    arr.sort()
    print('Sorted Array: ', *arr)
    target = int(input('Enter target value to be found: '))
    print('Position: ', binary_search(arr, 0, len(arr) - 1, target))

if __name__ == '__main__':
    main()
```

Output:



Experiment 2

Aim:
WAP on quick sort.

Code:

```
import random

def partition(arr, l, r):
    x = arr[l]
    j = l
    k = l
    for i in range(l+1, r+1):
        if arr[i] <= x:
            j += 1
            arr[i], arr[j] = arr[j], arr[i]
            if arr[j] < x and arr[k] != arr[j]:
                arr[j], arr[k] = arr[k], arr[j]
                k += 1
    if k > l+1:
        arr[l], arr[k-1] = arr[k-1], arr[l]
    return j, k

def quick_sort(arr, l, r):
    if l > r:
        return
    k = random.randint(l, r)
    arr[l], arr[k] = arr[k], arr[l]
    m1, m2 = partition(arr, l, r)
    quick_sort(arr, l, m1-1)
    quick_sort(arr, m2+1, r)

def main():
    print('Quick Sort')
    arr = list(map(int, input('Enter Elements of Array: ').strip().split()))
    quick_sort(arr, 0, len(arr) - 1)
    print('Sorted Array: ', *arr)

if __name__ == '__main__':
    main()
```

Output:

```
Quick Sort
Enter Elements of Array: 5 3 2 19 88 1 15
Sorted Array:  1 2 3 5 15 19 88
```

Experiment 3

Aim:
WAP on merge sort.

Code:

```
def merge(X, Y):
    R = [] * (len(X) + len(Y))
    while len(X) > 0 and len(Y) > 0:
        x = X[0]
        y = Y[0]
        if x <= y:
            R.insert(len(R), x)
            X.remove(x)
        else:
            R.insert(len(R), y)
            Y.remove(y)
    while len(X) > 0:
        R.insert(len(R), X[0])
        X.remove(X[0])
    while len(Y) > 0:
        R.insert(len(R), Y[0])
        Y.remove(Y[0])
    return R

def merge_sort(arr):
    n = len(arr)
    if n == 1:
        return arr
    m = n // 2
    X = merge_sort(arr[:m])
    Y = merge_sort(arr[m:])
    A = merge(X, Y)
    return A

def main():
    print('Merge Sort')
    arr = list(map(int, input('Enter Elements of Array: ').strip().split()))
    arr = merge_sort(arr)
    print('Sorted Array: ', *arr)
```

```
if __name__ == '__main__':
    main()
```

Output:

```
Merge Sort
Enter Elements of Array: 22 39 12 58 2
Sorted Array:  2 12 22 39 58
```

Experiment 4

Aim:
WAP to design a Lexical analyzer for identifying different types of token used in C language.

Code:

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

int isDelimiter(char ch)
{
   if (ch == ' ' || ch == '+' || ch == '-' || ch == '*' ||
      ch == '/' || ch == ',' || ch == ';' || ch == '>' ||
      ch == '<' || ch == '=' || ch == '(' || ch == ')' ||
      ch == '[' || ch == ']' || ch == '{' || ch == '}')
      return 1;
   return 0;
}

int isOperator(char ch)
{
   if (ch == '+' || ch == '-' || ch == '*' ||
      ch == '/' || ch == '>' || ch == '<' || ch == ';' ||
      ch == '=' || ch == '{' || ch == '}' || ch == '(' || ch == ')')
      return 1;
   return 0;
}

int validIdentifier(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' || isDelimiter(str[0]) == 1)
      return 0;
   return 1;
}

int isKeyword(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 1;
    return 0;
}

int isInteger(char *str)
{
    int i, len = strlen(str);

    if (len == 0)
        return 0;
    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 0;
    }
    return 1;
}

int isRealNumber(char *str)
{
    int i, len = strlen(str);
    int hasDecimal = 0;

    if (len == 0)
        return 0;
    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 0;
        if (str[i] == '.')
            hasDecimal = 1;
    }
```

```
    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 parse(char *str)
{
    int left = 0, right = 0;
    int len = strlen(str);
    while (right <= len && left <= right)
    {
        if (isDelimiter(str[right]) == 0)
            right++;
        if (isDelimiter(str[right]) == 1 && left == right)
        {
            if (isOperator(str[right]) == 1)
            {
                printf("'%c' is an OPERATOR\n", str[right]);
            }
            right++;
            left = right;
        }
        else if (isDelimiter(str[right]) == 1 && left != right || (right == len && left != right))
        {
            char *subStr = subString(str, left, right - 1);
            if (isKeyword(subStr) == 1)
            {
                printf("'%s' is a KEYWORD\n", subStr);
            }
            else if (isInteger(subStr) == 1)
            {
                printf("'%s' is a LITERAL (Integer)\n", subStr);
            }
            else if (isRealNumber(subStr) == 1)
            {
```

```
            printf("'%s' is a LITERAL (Real Number)\n", subStr);
        }
        else if (validIdentifier(subStr) == 1 && isDelimiter(str[right - 1]) == 0)
        {
            printf("'%s' is a valid IDENTIFIER\n", subStr);
        }
        else if (validIdentifier(subStr) == 0 && isDelimiter(str[right - 1]) == 0)
        {
            printf("'%s' is NOT a valid IDENTIFIER\n", subStr);
        }
        left = right;
      }
   }
   return;
}

int main()
{
   char str[50];
   printf("Enter a statement: ");
   gets(str);
   printf("\n");
   parse(str);
   return (0);
}
```

Output:

```
Enter a statement: if(a>b){printf("Hello");}

'if' is a KEYWORD
'(' is an OPERATOR
'a' is a valid IDENTIFIER
'>' is an OPERATOR
'b' is a valid IDENTIFIER
')' is an OPERATOR
'{' is an OPERATOR
'printf' is a valid IDENTIFIER
'(' is an OPERATOR
'"Hello"' is a valid IDENTIFIER
')' is an OPERATOR
';' is an OPERATOR
'}' is an OPERATOR
```

Experiment 5

Aim:
WAP to count the number of tokens in the given code.

Code:

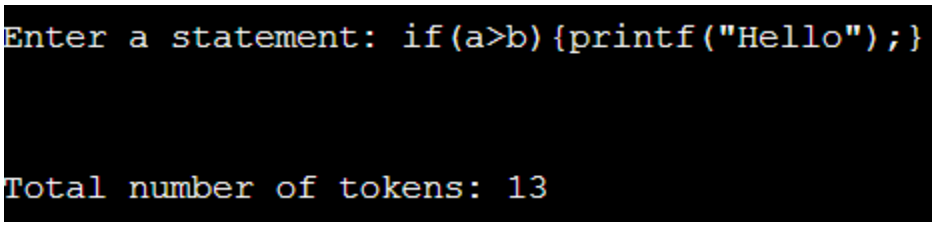
```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int isDelimiter(char ch)
{
    if (ch == ' ' || ch == '+' || ch == '-' || ch == '*' ||
        ch == '/' || ch == ',' || ch == ';' || ch == '>' ||
        ch == '<' || ch == '=' || ch == '(' || ch == ')' ||
        ch == '[' || ch == ']' || ch == '{' || ch == '}')
        return 1;
    return 0;
}

int isOperator(char ch)
{
    if (ch == '+' || ch == '-' || ch == '*' ||
        ch == '/' || ch == '>' || ch == '<' || ch == ';' ||
        ch == '=' || ch == '{' || ch == '}' || ch == '(' || ch == ')')
        return 1;
    return 0;
}

int validIdentifier(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' || isDelimiter(str[0]) == 1)
        return 0;
    return 1;
}

int isKeyword(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 1;
    return 0;
}

int isInteger(char *str)
{
    int i, len = strlen(str);

    if (len == 0)
        return 0;
    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 0;
    }
    return 1;
}

int isRealNumber(char *str)
{
    int i, len = strlen(str);
    int hasDecimal = 0;

    if (len == 0)
        return 0;
    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 0;
        if (str[i] == '.')
            hasDecimal = 1;
    }
    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 parse(char *str)
{
    int left = 0, right = 0;
    int len = strlen(str);
    int total_count = 0;
    while (right <= len && left <= right)
    {
        if (isDelimiter(str[right]) == 0)
            right++;
        if (isDelimiter(str[right]) == 1 && left == right)
        {
            if (isOperator(str[right]) == 1)
            {
                total_count += 1;
            }
            right++;
            left = right;
        }
        else if (isDelimiter(str[right]) == 1 && left != right || (right == len && left != right))
        {
            char *subStr = subString(str, left, right - 1);
            if (isKeyword(subStr) == 1)
            {
                total_count += 1;
            }
            else if (isInteger(subStr) == 1)
            {
                total_count += 1;
            }
            else if (isRealNumber(subStr) == 1)
            {
```

```
            total_count += 1;
        }
        else if (validIdentifier(subStr) == 1 && isDelimiter(str[right - 1]) == 0)
        {
            total_count += 1;
        }
        else if (validIdentifier(subStr) == 0 && isDelimiter(str[right - 1]) == 0)
        {
            total_count += 1;
        }
        left = right;
      }
   }
   printf("\nTotal number of tokens: %d", total_count);
   return;
}

int main()
{
   char str[50];
   printf("Enter a statement: ");
   gets(str);
   printf("\n");
   parse(str);
   return (0);
}
```

Output:



Experiment 6

Aim:
WAP to identify whether a given line is a comment or not.

Code:

```
import re

def main():
    statement = input('Enter statement:')
    regex1 = re.search("//(.*)", statement)
    regex2 = re.search("/\\*(.*?)\\*/", statement)
    if regex1 or regex2:
        print("It is a comment!")
    else:
        print("It is not a comment!")

if __name__ == '__main__':
    main()
```

Output:

```
Enter statement://Hello
It is a comment!
Enter statement:/*Hello*/
It is a comment!
Enter statement:Hello
It is not a comment!
```

Experiment 7

Aim:
WAP to create a symbol table for following input strings.
(a) int i , j;
(b) int k = int i + int j;

Code:

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

int isDelimiter(char ch)
{
   if (ch == ' ' || ch == '+' || ch == '-' || ch == '*' ||
      ch == '/' || ch == ',' || ch == ';' || ch == '>' ||
      ch == '<' || ch == '=' || ch == '(' || ch == ')' ||
      ch == '[' || ch == ']' || ch == '{' || ch == '}')
      return 1;
   return 0;
}

int validIdentifier(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' || isDelimiter(str[0]) == 1)
      return 0;
   return 1;
}

int isKeyword(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 1;
    return 0;
}

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 parse(char *str)
{
    int left = 0, right = 0;
    int len = strlen(str);
    int count = 0;
    while (right <= len && left <= right)
    {
        if (isDelimiter(str[right]) == 0)
            right++;
        if (isDelimiter(str[right]) == 1 && left == right)
        {
            right++;
            left = right;
        }
        else if (isDelimiter(str[right]) == 1 && left != right || (right == len && left != right))
        {
            char *subStr = subString(str, left, right - 1);
            if (isKeyword(subStr) == 0 && validIdentifier(subStr) == 1 && isDelimiter(str[right - 1]) ==
0)
            {
                count++;
                printf("%s               id%d\n", subStr,count);
            }
            left = right;
        }
    }
    return;
```

```
}

int main()
{
   char str[50];
   printf("Enter a statement: ");
   gets(str);
   printf("\n");
   printf("Symbol Table\n");
   parse(str);
   return (0);
}
```

Output:

```
Enter a statement: int i,j;

Symbol Table
i                id1
j                id2
```

```
Enter a statement: int k=int i+int j;

Symbol Table
k                id1
i                id2
j                id3
```

Experiment 8

Aim:

WAP to create lexeme and token table for:

(a) print(“Hello”);

Code:

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

int isDelimiter(char ch)
{
   if (ch == ' ' || ch == '+' || ch == '-' || ch == '*' ||
      ch == '/' || ch == ',' || ch == ';' || ch == '>' ||
      ch == '<' || ch == '=' || ch == '(' || ch == ')' ||
      ch == '[' || ch == ']' || ch == '{' || ch == '}')
      return 1;
   return 0;
}

int isOperator(char ch)
{
   if (ch == '+' || ch == '-' || ch == '*' ||
      ch == '/' || ch == '>' || ch == '<' || ch == ';' ||
      ch == '=' || ch == '{' || ch == '}' || ch == '(' || ch == ')')
      return 1;
   return 0;
}

int validIdentifier(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' || isDelimiter(str[0]) == 1)
      return 0;
   return 1;
}

int isKeyword(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 1;
    return 0;
}

int isInteger(char *str)
{
    int i, len = strlen(str);

    if (len == 0)
        return 0;
    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 0;
    }
    return 1;
}

int isRealNumber(char *str)
{
    int i, len = strlen(str);
    int hasDecimal = 0;

    if (len == 0)
        return 0;
    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 0;
        if (str[i] == '.')
            hasDecimal = 1;
    }
```

```
    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 parse(char *str)
{
    int left = 0, right = 0;
    int len = strlen(str);
    while (right <= len && left <= right)
    {
        if (isDelimiter(str[right]) == 0)
            right++;
        if (isDelimiter(str[right]) == 1 && left == right)
        {
            if (isOperator(str[right]) == 1)
            {
                printf("%c               OPERATOR\n", str[right]);
            }
            right++;
            left = right;
        }
        else if (isDelimiter(str[right]) == 1 && left != right || (right == len && left != right))
        {
            char *subStr = subString(str, left, right - 1);
            if (isKeyword(subStr) == 1)
            {
                printf("%s               KEYWORD\n", subStr);
            }
            else if (isInteger(subStr) == 1)
            {
                printf("%s               LITERAL (Integer)\n", subStr);
            }
            else if (isRealNumber(subStr) == 1)
            {
```

```
            printf("%s              LITERAL (Real Number)\n", subStr);
        }
        else if (validIdentifier(subStr) == 1 && isDelimiter(str[right - 1]) == 0)
        {
            printf("%s         IDENTIFIER\n", subStr);
        }
        left = right;
      }
   }
   return;
}

int main()
{
   char str[50];
   printf("Enter a statement: ");
   gets(str);
   printf("\n");
   printf("Lexeme          Token\n");
   parse(str);
   return (0);
}
```

Output:

```
Enter a statement: printf("Hello");

Lexeme            Token
printf           IDENTIFIER
(                 OPERATOR
"Hello"           IDENTIFIER
)                 OPERATOR
;                 OPERATOR
```

Experiment 9

Aim:

WAP for following regular expressions:

a) (0 + 1)* + 0*1*
b) (ab*c + (def)+ + a*d+e)+
c) ((a + b)(c + d))+ + ab*c*d

Code:

```
import re

def main():
    exp = input('Enter expression for regex (0 + 1)* + 0*1*  :')
    if re.search('(0|1)*0*1*', exp):
        print('String matched')
    else:
        print('String not matched')
    exp = input('Enter expression for regex (ab*c + (def)+ + a*d+e)+  :')
    if re.search('(ab*c(def)+a+d+e)+', exp):
        print('String matched')
    else:
        print('String not matched')
    exp = input('Enter expression for regex ((a + b)(c + d))+ + ab*c*d  :')
    if re.search('((a|b)(c|d))+ab*c*d', exp):
        print('String matched')
    else:
        print('String not matched')

if __name__ == '__main__':
    main()
```

Output:

```
Enter expression for regex (0 + 1)* + 0*1*  :01
String matched
Enter expression for regex (ab*c + (def)+ + a*d+e)+  :abbcdefdefadde
String matched
Enter expression for regex ((a + b)(c + d))+ + ab*c*d  :bdacabbcccd
String matched
```

Experiment 10

Aim:
WAP which accepts a regular expression from the user and generates a regular grammar which is equivalent to the R.E. entered by the user. The grammar will be printed 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.

Code:

```
def main():
    grammar = []
    n = int(input('Enter number of productions:'))
    for i in range(n):
        production = input(f'Enter production {i+1}: ')
        grammar.append(production)
    grammar = list(set(grammar))
    grammars = dict()
    for production in grammar:
        if production[0] not in grammars:
            grammars[production[0]] = []
        grammars[production[0]].append(production[3:])
    print('Combined Productions:')
    for production in grammars:
        print(production,'->','|'.join(grammars[production]))

if __name__ == '__main__':
    main()
```

Output:

```
Enter number of productions:6
Enter production 1: S->ab
Enter production 2: A->ab
Enter production 3: S->ab
Enter production 4: A->ab
Enter production 5: S->Ce
Enter production 6: S->gh
Combined Productions:
S -> gh|Ce|ab
A -> ab
```

Experiment 11

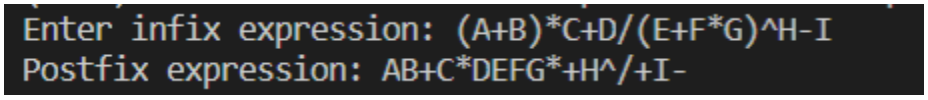
Aim:
WAP to convert infix expression to postfix.

Code:

```
stack = []

def push(c):
    stack.append(c)

def pop():
    return stack.pop()

def peek():
    return stack[-1]

def check_priority(c):
    if c == '^':
        return 3
    elif c == '/' or c == '*':
        return 2
    elif c == '+' or c == '-':
        return 1
    else:
        return 0

def main():
    expr = input('Enter infix expression: ')
    print('Postfix expression: ',end='')
    expr += ')'
    push('(')
    for i in range(len(expr)):
        if expr[i].isalnum():
            print(expr[i], end='')
        elif expr[i] == '(':
            push(expr[i])
        elif expr[i] == ')':
            op = pop()
            while op != '(':
                print(op, end='')
```

```
            op = pop()
        else:
            while check_priority(peek()) >= check_priority(expr[i]):
                print(pop(), end='')
            push(expr[i])

if __name__ == '__main__':
    main()
```

Output:



Experiment 12

Aim:
WAP to convert infix expression to prefix.

Code:

```
stack = []

def push(c):
    stack.append(c)

def pop():
    return stack.pop()

def peek():
    return stack[-1]

def check_priority(c):
    if c == '^':
        return 3
    elif c == '/' or c == '*':
        return 2
    elif c == '+' or c == '-':
        return 1
    else:
        return 0

def main():
    expr = input('Enter infix expression: ')
    print('Prefix expression: ', end='')
    eq = ''
    expr = expr[::-1]
    expr += '('
    push(')')
    for i in range(len(expr)):
        if expr[i].isalnum():
            eq += expr[i]
        elif expr[i] == ')':
            push(expr[i])
        elif expr[i] == '(':
            op = pop()
```

```
            while op != ')':
                eq += op
                op = pop()
        else:
            while check_priority(peek()) > check_priority(expr[i]):
                eq += pop()
            push(expr[i])
    eq = eq[::-1]
    print(eq)

if __name__ == '__main__':
    main()
```

Output:

```
Enter infix expression: (A+B)*C+D/(E+F*G)^H-I
Prefix expression: -+*+ABC/D^+E*FGHI
```

Experiment 13

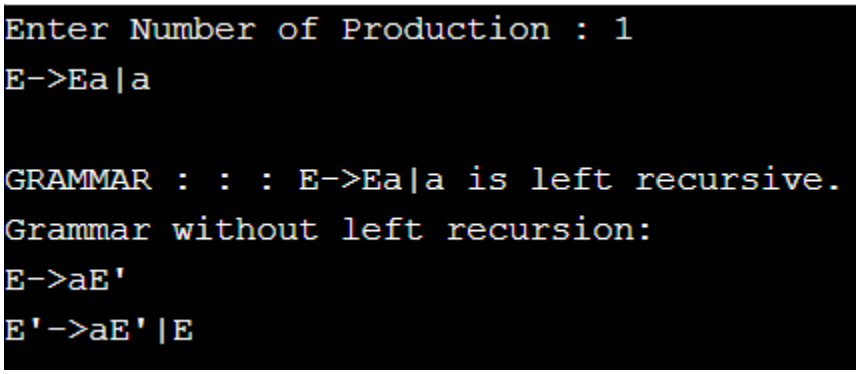
Aim:
WAP To Find Left Recursion and Remove it.

Code:

```
#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("Enter Number of Production : ");
    scanf("%d", &num);
    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");
            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;
   }
}
```

Output:



Experiment 14

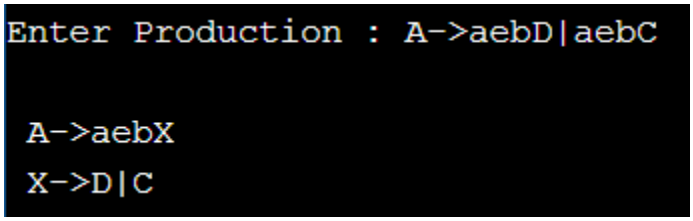
Aim:
WAP To Find Left Factoring and Remove it.

Code:

```
#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("Enter 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("\n A->%s", modifiedGram);
    printf("\n X->%s\n", newGram);
}
```

Output:



Experiment 15

Aim:
WAP to calculate First and Follow sets of given grammar.

Code:

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

void followfirst(char, int, int);
void findfirst(char, int, int);
void follow(char c);

int count, n = 0;
char calc_first[10][100];
char calc_follow[10][100];
int m = 0;
char production[10][10], first[10];
char f[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;
   printf("Enter number of productions :");
   scanf("%d", &count);
   printf("\nEnter %d productions in form A=B where A and B are grammar symbols :\n", count);
   for (i = 0; i < count; i++)
   {
      scanf("%s%c", production[i], &ch);
   }
   int kay;
   char done[100];
   int ptr = -1;
   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;
    for (kay = 0; kay <= ptr; kay++)
        if (c == done[kay])
            xxx = 1;
    if (xxx == 1)
        continue;
    findfirst(c, 0, 0);
    ptr += 1;
    done[ptr] = c;
    printf("\n First(%c)= { ", c);
    calc_first[point1][point2++] = c;
    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[100];
ptr = -1;
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;
   for (kay = 0; kay <= ptr; kay++)
      if (ck == donee[kay])
         xxx = 1;
   if (xxx == 1)
      continue;
   land += 1;
   follow(ck);
   ptr += 1;
   donee[ptr] = ck;
   printf(" Follow(%c) = { ", ck);
   calc_follow[point1][point2++] = ck;
   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++;
    }
    char ter[10];
    for (k = 0; k < 10; k++)
    {
        ter[k] = '!';
    }
    int ap, vp, sid = 0;
    for (k = 0; k < count; k++)
    {
        for (kay = 0; kay < count; kay++)
        {
            if (!isupper(production[k][kay]) && production[k][kay] != '#' && production[k][kay] != '=' &&
production[k][kay] != '\0')
            {
                vp = 0;
                for (ap = 0; ap < sid; ap++)
                {
                    if (production[k][kay] == ter[ap])
                    {
                        vp = 1;
                        break;
                    }
                }
                if (vp == 0)
                {
                    ter[sid] = production[k][kay];
                    sid++;
                }
            }
        }
    }
}

void follow(char c)
{
    int i, j;
    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')
                {
                    followfirst(production[i][j + 1], i, (j + 2));
                }
                if (production[i][j + 1] == '\0' && c != production[i][0])
                {
                    follow(production[i][0]);
                }
            }
        }
    }
}

void findfirst(char c, int q1, int q2)
{
    int j;
    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))
                {
                    findfirst(production[q1][q2], q1, (q2 + 1));
                }
                else
                    first[n++] = '#';
            }
            else if (!isupper(production[j][2]))
            {
                first[n++] = production[j][2];
```

```
            }
            else
            {
                findfirst(production[j][2], j, 3);
            }
        }
    }
}

void followfirst(char c, int c1, int c2)
{
    int k;
    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;
        }
        while (calc_first[i][j] != '!')
        {
            if (calc_first[i][j] != '#')
            {
                f[m++] = calc_first[i][j];
            }
            else
            {
                if (production[c1][c2] == '\0')
                {
                    follow(production[c1][0]);
                }
                else
                {
                    followfirst(production[c1][c2], c1, c2 + 1);
                }
            }
            j++;
        }
    }
}
```

Output:

```
Enter 6 productions in form A=B where A and B are grammar symbols :
S=A
A=aBQ
Q=dQ
Q=#
B=b
C=g

 First(S)= { a, }

 First(A)= { a, }

 First(Q)= { d, #, }

 First(B)= { b, }

 First(C)= { g, }

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

 Follow(S) = { $,  }

 Follow(A) = { $,  }

 Follow(Q) = { $,  }

 Follow(B) = { d, $,  }

 Follow(C) = {  }
```

Experiment 16

Aim:
Write a program for Recursive Descent Calculator.

Code:

```
#include <string>
#include <iostream>
#include <list>
#include <sstream>
using namespace std;
std::string
removeSpaces(const std::string &str)
{
   std::string s(str);
   int j = 0;
   int N = s.size();
   for (int i = 0; i < N; ++i)
   {
      if (s[i] != ' ')
      {
         s[j] = s[i];
         j++;
      }
   }
   s.resize(j);
   return s;
}
void tokenize(const std::string &str,std::list<std::string> &tokens)
{
   std::string num;
   int N = str.size();
   for (int i = 0; i < N; ++i)
   {
      char c = str[i];
      if (isdigit(c))
      {
         num += c;
      }
      else
      {
```

```
            if (!num.empty())
            {
                tokens.push_back(num);
                num.clear();
            }
            std::string token;
            token += c;
            tokens.push_back(token);
        }
    }
    if (!num.empty())
    {
        tokens.push_back(num);
        num.clear();
    }
}
class Calculator
{
    public:
        Calculator(const std::string &expression);
        void next();
        int exp();
        int term();
        int factor();
        int toInt(const std::string &s);
    private:
        std::list<std::string> mTokens;
        std::string mCurrent;
};
Calculator::Calculator(const std::string &expression)
{
    std::string s = removeSpaces(expression);
    tokenize(s, mTokens);
    mCurrent = mTokens.front();
}
void Calculator::next()
{
    mTokens.pop_front();
    if (!mTokens.empty())
    {
        mCurrent = mTokens.front();
    }
    else
    {
```

```
        mCurrent = std::string();
    }
}
int Calculator::exp()
{
    int result = term();
    while (mCurrent == "+" || mCurrent == "-")
    {
        if (mCurrent == "+")
        {
            next();
            result += term();
        }
        if (mCurrent == "-")
        {
            next();
            result -= term();
        }
    }
    return result;
}
int Calculator::term()
{
    int result = factor();
    while (mCurrent == "*" || mCurrent == "/")
    {
        if (mCurrent == "*")
        {
            next();
            result *= factor();
        }
        if (mCurrent == "/")
        {
            next();
            int denominator = factor();
            if (denominator != 0)
            {
                result /= denominator;
            }
            else
            {
                result = 0;
            }
        }
```

```
    }
    return result;
}
int Calculator::factor()
{
    int result;
    if (mCurrent == "(")
    {
        next();
        result = exp();
        next();
    }
    else
    {
        result = toInt(mCurrent);
        next();
    }
    return result;
}
int Calculator::toInt(const std::string &s)
{   std::stringstream ss;
    ss << s;
    int x;
    ss >> x;
    return x;
}
int calculate(std::string s)
{   Calculator calculator(s);
    return calculator.exp();
}
int main()
{
    std::string expression;
    cout << "Input : ";
    cin >> expression;
    std::cout << expression << " -> " << calculate(expression) << std::endl;
    return 0;
}
```

Output:

```
Input : 3+2*2
3+2*2 -> 7
```

Experiment 17

Aim:
WAP to generate LL(1) parsing table.

Code:

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

void followfirst(char, int, int);
void findfirst(char, int, int);
void follow(char c);

int count, n = 0;
char calc_first[10][100];
char calc_follow[10][100];
int m = 0;
char production[10][10], first[10];
char f[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;
   printf("Enter number of productions :");
   scanf("%d", &count);
   printf("\nEnter %d productions in form A=B where A and B are grammar symbols :\n", count);
   for (i = 0; i < count; i++)
   {
      scanf("%s%c", production[i], &ch);
   }
   int kay;
   char done[100];
   int ptr = -1;
   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;
    for (kay = 0; kay <= ptr; kay++)
        if (c == done[kay])
            xxx = 1;
    if (xxx == 1)
        continue;
    findfirst(c, 0, 0);
    ptr += 1;
    done[ptr] = c;
    printf("\n First(%c)= { ", c);
    calc_first[point1][point2++] = c;
    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[100];
ptr = -1;
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;
   for (kay = 0; kay <= ptr; kay++)
      if (ck == donee[kay])
         xxx = 1;
   if (xxx == 1)
      continue;
   land += 1;
   follow(ck);
   ptr += 1;
   donee[ptr] = ck;
   printf(" Follow(%c) = { ", ck);
   calc_follow[point1][point2++] = ck;
   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++;
    }
    char ter[10];
    for (k = 0; k < 10; k++)
    {
        ter[k] = '!';
    }
    int ap, vp, sid = 0;
    for (k = 0; k < count; k++)
    {
        for (kay = 0; kay < count; kay++)
        {
            if (!isupper(production[k][kay]) && production[k][kay] != '#' && production[k][kay] != '=' &&
production[k][kay] != '\0')
            {
                vp = 0;
                for (ap = 0; ap < sid; ap++)
                {
                    if (production[k][kay] == ter[ap])
                    {
                        vp = 1;
                        break;
                    }
                }
                if (vp == 0)
                {
                    ter[sid] = production[k][kay];
                    sid++;
                }
            }
        }
    }
    ter[sid] = '$';
    sid++;
    printf("\n\t\t\t\t\t\t\t The LL(1) Parsing Table for the above grammar :-");
    printf("\n\t\t\t\t\t\t\t^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^\n");

printf("\n\t\t\t==========================================================================
=================================================\n");
    printf("\t\t\t\t|\t");
    for (ap = 0; ap < sid; ap++)
```

```
    {
        printf("%c\t\t", ter[ap]);
    }

printf("\n\t\t\t==========================================================================
=========================================================\n");
    char first_prod[100][100];
    for (ap = 0; ap < count; ap++)
    {
        int destiny = 0;
        k = 2;
        int ct = 0;
        char tem[100];
        while (production[ap][k] != '\0')
        {
            if (!isupper(production[ap][k]))
            {
                tem[ct++] = production[ap][k];
                tem[ct++] = '_';
                tem[ct++] = '\0';
                k++;
                break;
            }
            else
            {
                int zap = 0;
                int tuna = 0;
                for (zap = 0; zap < count; zap++)
                {
                    if (calc_first[zap][0] == production[ap][k])
                    {
                        for (tuna = 1; tuna < 100; tuna++)
                        {
                            if (calc_first[zap][tuna] != '!')
                            {
                                tem[ct++] = calc_first[zap][tuna];
                            }
                            else
                                break;
                        }
                        break;
                    }
                }
                tem[ct++] = '_';
```

```
        }
        k++;
    }
    int zap = 0, tuna;
    for (tuna = 0; tuna < ct; tuna++)
    {
        if (tem[tuna] == '#')
        {
            zap = 1;
        }
        else if (tem[tuna] == '_')
        {
            if (zap == 1)
            {
                zap = 0;
            }
            else
                break;
        }
        else
        {
            first_prod[ap][destiny++] = tem[tuna];
        }
    }
}
char table[100][100 + 1];
ptr = -1;
for (ap = 0; ap < land; ap++)
{
    for (kay = 0; kay < (sid + 1); kay++)
    {
        table[ap][kay] = '!';
    }
}
for (ap = 0; ap < count; ap++)
{
    ck = production[ap][0];
    xxx = 0;
    for (kay = 0; kay <= ptr; kay++)
        if (ck == table[kay][0])
            xxx = 1;
    if (xxx == 1)
        continue;
    else
```

```
    {
        ptr = ptr + 1;
        table[ptr][0] = ck;
    }
}
for (ap = 0; ap < count; ap++)
{
    int tuna = 0;
    while (first_prod[ap][tuna] != '\0')
    {
        int to, ni = 0;
        for (to = 0; to < sid; to++)
        {
            if (first_prod[ap][tuna] == ter[to])
            {
                ni = 1;
            }
        }
        if (ni == 1)
        {
            char xz = production[ap][0];
            int cz = 0;
            while (table[cz][0] != xz)
            {
                cz = cz + 1;
            }
            int vz = 0;
            while (ter[vz] != first_prod[ap][tuna])
            {
                vz = vz + 1;
            }
            table[cz][vz + 1] = (char)(ap + 65);
        }
        tuna++;
    }
}
for (k = 0; k < sid; k++)
{
    for (kay = 0; kay < 100; kay++)
    {
        if (calc_first[k][kay] == '!')
        {
            break;
        }
```

```
            else if (calc_first[k][kay] == '#')
            {
                int fz = 1;
                while (calc_follow[k][fz] != '!')
                {
                    char xz = production[k][0];
                    int cz = 0;
                    while (table[cz][0] != xz)
                    {
                        cz = cz + 1;
                    }
                    int vz = 0;
                    while (ter[vz] != calc_follow[k][fz])
                    {
                        vz = vz + 1;
                    }
                    table[k][vz + 1] = '#';
                    fz++;
                }
                break;
            }
        }
    }
    for (ap = 0; ap < land; ap++)
    {
        printf("\t\t\t   %c\t|\t", table[ap][0]);
        for (kay = 1; kay < (sid + 1); kay++)
        {
            if (table[ap][kay] == '!')
                printf("\t\t");
            else if (table[ap][kay] == '#')
                printf("%c=#\t\t", table[ap][0]);
            else
            {
                int mum = (int)(table[ap][kay]);
                mum -= 65;
                printf("%s\t\t", production[mum]);
            }
        }
        printf("\n");

printf("\t\t\t------------------------------------------------------------------------------------------------------------------------
---");
        printf("\n");
```

```
    }
}

void follow(char c)
{
    int i, j;
    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')
                {
                    followfirst(production[i][j + 1], i, (j + 2));
                }
                if (production[i][j + 1] == '\0' && c != production[i][0])
                {
                    follow(production[i][0]);
                }
            }
        }
    }
}

void findfirst(char c, int q1, int q2)
{
    int j;
    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))
            {
                findfirst(production[q1][q2], q1, (q2 + 1));
            }
            else
                first[n++] = '#';
        }
        else if (!isupper(production[j][2]))
        {
            first[n++] = production[j][2];
        }
        else
        {
            findfirst(production[j][2], j, 3);
        }
    }
  }
}

void followfirst(char c, int c1, int c2)
{
    int k;
    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;
        }
        while (calc_first[i][j] != '!')
        {
            if (calc_first[i][j] != '#')
            {
                f[m++] = calc_first[i][j];
            }
            else
            {
                if (production[c1][c2] == '\0')
                {
                    follow(production[c1][0]);
                }
```

```
            else
            {
               followfirst(production[c1][c2], c1, c2 + 1);
            }
         }
         j++;
      }
   }
}
```

Output:

```
Enter number of productions :6
Enter 6 productions in form A=B where A and B are grammar symbols :
S=A
A=aBQ
Q=dQ
Q=#
B=b
C=g

 First(S)= { a, }
 First(A)= { a, }
 First(Q)= { d, #, }
 First(B)= { b, }
 First(C)= { g, }
-----------------------------------------------
 Follow(S) = { $,  }
 Follow(A) = { $,  }
 Follow(Q) = { $,  }
 Follow(B) = { d, $,  }
 Follow(C) = {  }

The LL(1) Parsing Table for the above grammer :-
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

================================================================================
    |    a       d       b       g       $
================================================================================
   S   |   S=A
--------------------------------------------------------------------------------
   A   |   A=aBQ
--------------------------------------------------------------------------------
   Q   |           Q=dQ                  Q=#
--------------------------------------------------------------------------------
   B   |                   B=b
--------------------------------------------------------------------------------
   C   |                           C=g
--------------------------------------------------------------------------------
```

Experiment 18

Aim:
WAP that recognizes different types of English words.

Code:

```
import nltk

def main():
    string = input('Enter input string:')
    text = nltk.word_tokenize(string)
    types = nltk.pos_tag(text)
    tagdict = nltk.data.load('help/tagsets/upenn_tagset.pickle')
    for t in types:
        print(t[0],' --> ', tagdict[t[1]][0])

if __name__ == '__main__':
    main()
```

Output:

```
Enter input string:The man was running in the rain
The  -->  determiner
man  -->  noun, common, singular or mass
was  -->  verb, past tense
running  -->  verb, present participle or gerund
in  -->  preposition or conjunction, subordinating
the  -->  determiner
rain  -->  noun, common, singular or mass
```

Experiment 19

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

Code:

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

void followfirst(char, int, int);
void findfirst(char, int, int);
void follow(char c);

int count, n = 0;
char calc_first[10][100];
char calc_follow[10][100];
int m = 0;
char production[10][10], first[10];
char f[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;
   printf("Enter number of productions :");
   scanf("%d", &count);
   printf("\nEnter %d productions in form A=B where A and B are grammar symbols :\n", count);
   for (i = 0; i < count; i++)
   {
      scanf("%s%c", production[i], &ch);
   }
   int kay;
```

```
char done[100];
int ptr = -1;
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;
    for (kay = 0; kay <= ptr; kay++)
        if (c == done[kay])
            xxx = 1;
    if (xxx == 1)
        continue;
    findfirst(c, 0, 0);
    ptr += 1;
    done[ptr] = c;
    printf("\n First(%c)= { ", c);
    calc_first[point1][point2++] = c;
    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[100];
ptr = -1;
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;
    for (kay = 0; kay <= ptr; kay++)
        if (ck == donee[kay])
            xxx = 1;
    if (xxx == 1)
        continue;
    land += 1;
    follow(ck);
    ptr += 1;
    donee[ptr] = ck;
    printf(" Follow(%c) = { ", ck);
    calc_follow[point1][point2++] = ck;
    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++;
  }
  char ter[10];
  for (k = 0; k < 10; k++)
  {
    ter[k] = '!';
  }
  int ap, vp, sid = 0;
  for (k = 0; k < count; k++)
  {
    for (kay = 0; kay < count; kay++)
    {
      if (!isupper(production[k][kay]) && production[k][kay] != '#' && production[k][kay] != '=' &&
production[k][kay] != '\0')
      {
        vp = 0;
        for (ap = 0; ap < sid; ap++)
        {
          if (production[k][kay] == ter[ap])
          {
            vp = 1;
            break;
          }
        }
        if (vp == 0)
        {
          ter[sid] = production[k][kay];
          sid++;
        }
      }
    }
  }
  ter[sid] = '$';
  sid++;
  for (ap = 0; ap < sid; ap++)
  {
    printf("%c\t\t", ter[ap]);
  }
```

```
char first_prod[100][100];
for (ap = 0; ap < count; ap++)
{
    int destiny = 0;
    k = 2;
    int ct = 0;
    char tem[100];
    while (production[ap][k] != '\0')
    {
        if (!isupper(production[ap][k]))
        {
            tem[ct++] = production[ap][k];
            tem[ct++] = '_';
            tem[ct++] = '\0';
            k++;
            break;
        }
        else
        {
            int zap = 0;
            int tuna = 0;
            for (zap = 0; zap < count; zap++)
            {
                if (calc_first[zap][0] == production[ap][k])
                {
                    for (tuna = 1; tuna < 100; tuna++)
                    {
                        if (calc_first[zap][tuna] != '!')
                        {
                            tem[ct++] = calc_first[zap][tuna];
                        }
                        else
                            break;
                    }
                    break;
                }
            }
            tem[ct++] = '_';
        }
        k++;
    }
    int zap = 0, tuna;
    for (tuna = 0; tuna < ct; tuna++)
    {
```

```
        if (tem[tuna] == '#')
        {
            zap = 1;
        }
        else if (tem[tuna] == '_')
        {
            if (zap == 1)
            {
                zap = 0;
            }
            else
                break;
        }
        else
        {
            first_prod[ap][destiny++] = tem[tuna];
        }
    }
}
char table[100][100 + 1];
ptr = -1;
for (ap = 0; ap < land; ap++)
{
    for (kay = 0; kay < (sid + 1); kay++)
    {
        table[ap][kay] = '!';
    }
}
for (ap = 0; ap < count; ap++)
{
    ck = production[ap][0];
    xxx = 0;
    for (kay = 0; kay <= ptr; kay++)
        if (ck == table[kay][0])
            xxx = 1;
    if (xxx == 1)
        continue;
    else
    {
        ptr = ptr + 1;
        table[ptr][0] = ck;
    }
}
for (ap = 0; ap < count; ap++)
```

```
{
    int tuna = 0;
    while (first_prod[ap][tuna] != '\0')
    {
        int to, ni = 0;
        for (to = 0; to < sid; to++)
        {
            if (first_prod[ap][tuna] == ter[to])
            {
                ni = 1;
            }
        }
        if (ni == 1)
        {
            char xz = production[ap][0];
            int cz = 0;
            while (table[cz][0] != xz)
            {
                cz = cz + 1;
            }
            int vz = 0;
            while (ter[vz] != first_prod[ap][tuna])
            {
                vz = vz + 1;
            }
            table[cz][vz + 1] = (char)(ap + 65);
        }
        tuna++;
    }
}
for (k = 0; k < sid; k++)
{
    for (kay = 0; kay < 100; kay++)
    {
        if (calc_first[k][kay] == '!')
        {
            break;
        }
        else if (calc_first[k][kay] == '#')
        {
            int fz = 1;
            while (calc_follow[k][fz] != '!')
            {
                char xz = production[k][0];
```

```
                int cz = 0;
                while (table[cz][0] != xz)
                {
                    cz = cz + 1;
                }
                int vz = 0;
                while (ter[vz] != calc_follow[k][fz])
                {
                    vz = vz + 1;
                }
                table[k][vz + 1] = '#';
                fz++;
            }
            break;
        }
    }
}
for (ap = 0; ap < land; ap++)
{
    for (kay = 1; kay < (sid + 1); kay++)
    {
        if (table[ap][kay] == '!');
        else if (table[ap][kay] == '#');
        else
        {
            int mum = (int)(table[ap][kay]);
            mum -= 65;
        }
    }
}
int j;
printf("\n\nPlease enter the desired INPUT STRING = ");
char input[100];
scanf("%s%c", input, &ch);
int i_ptr = 0, s_ptr = 1;
char stack[100];
stack[0] = '$';
stack[1] = table[0][0];
while (s_ptr != -1)
{
    int vamp = 0;
    for (vamp = 0; vamp <= s_ptr; vamp++);
    vamp = i_ptr;
    while (input[vamp] != '\0')
```

```
{
   vamp++;
}
char her = input[i_ptr];
char him = stack[s_ptr];
s_ptr--;
if (!isupper(him))
{
   if (her == him)
   {
      i_ptr++;
   }
   else
   {
      printf("\nString Not Accepted by LL(1) Parser !!\n");
      exit(0);
   }
}
else
{
   for (i = 0; i < sid; i++)
   {
      if (ter[i] == her)
         break;
   }
   char produ[100];
   for (j = 0; j < land; j++)
   {
      if (him == table[j][0])
      {
         if (table[j][i + 1] == '#')
         {
            produ[0] = '#';
            produ[1] = '\0';
         }
         else if (table[j][i + 1] != '!')
         {
            int mum = (int)(table[j][i + 1]);
            mum -= 65;
            strcpy(produ, production[mum]);
         }
         else
         {
            printf("\nString Not Accepted by LL(1) Parser !!\n");
```

```
                    exit(0);
                }
            }
        }
        int le = strlen(produ);
        le = le - 1;
        if (le == 0)
        {
            continue;
        }
        for (j = le; j >= 2; j--)
        {
            s_ptr++;
            stack[s_ptr] = produ[j];
        }
    }
}
if (input[i_ptr] == '\0')
{
    printf("\t\t\t\t\t\t\t\t\tYOUR STRING HAS BEEN ACCEPTED !!\n");
}
else
    printf("\n\t\t\t\t\t\t\t\t\tYOUR STRING HAS BEEN REJECTED !!\n");
}

void follow(char c)
{
    int i, j;
    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')
                {
                    followfirst(production[i][j + 1], i, (j + 2));
                }
                if (production[i][j + 1] == '\0' && c != production[i][0])
                {
```

```
                follow(production[i][0]);
            }
        }
    }
}

void findfirst(char c, int q1, int q2)
{
    int j;
    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))
                {
                    findfirst(production[q1][q2], q1, (q2 + 1));
                }
                else
                    first[n++] = '#';
            }
            else if (!isupper(production[j][2]))
            {
                first[n++] = production[j][2];
            }
            else
            {
                findfirst(production[j][2], j, 3);
            }
        }
    }
}

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

```
    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;
        }
        while (calc_first[i][j] != '!')
        {
            if (calc_first[i][j] != '#')
            {
                f[m++] = calc_first[i][j];
            }
            else
            {
                if (production[c1][c2] == '\0')
                {
                    follow(production[c1][0]);
                }
                else
                {
                    followfirst(production[c1][c2], c1, c2 + 1);
                }
            }
            j++;
        }
    }
}
```

Output:

```
Enter number of productions :7
Enter 7 productions in form A=B where A and B are grammar symbols :
S=ABC
A=abA
A=ab
B=b
B=bC
C=c
C=cC

 First(S)= { a, }
 First(A)= { a, }
 First(B)= { b, }
 First(C)= { c, }
-----------------------------------------------
 Follow(S) = { $,  }
 Follow(A) = { b,  }
 Follow(B) = { c,  }
 Follow(C) = { $, c,  }

Please enter the desired INPUT STRING = abbc
YOUR STRING HAS BEEN ACCEPTED
```

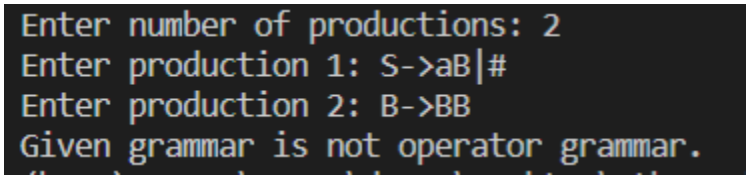
Experiment 20

Aim:
WAP to check whether given grammar is operator grammar or not.

Code:

```
def main():
    n = int(input('Enter number of productions: '))
    grammar = []
    operator_grammar = True
    for i in range(n):
        production = input(f'Enter production {i+1}: ')
        grammar.append(production)
    for production in grammar:
        if '#' in production[3:]:
            operator_grammar = False
        for symbol in range(len(production[3:])-1):
            if production[3:][symbol].isupper() and production[3:][symbol+1].isupper():
                operator_grammar = False
    if operator_grammar:
        print('Given grammar is operator grammar.')
    else:
        print('Given grammar is not operator grammar.')

if __name__ == '__main__':
    main()
```

Output:



Experiment 21

Aim:
WAP which accepts a regular grammar with no left-recursion, and no null-production rules, and then it accepts a string and checks whether a string belongs to the grammar or not.

Code:

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

void followfirst(char, int, int);
void findfirst(char, int, int);
void follow(char c);

int count, n = 0;
char calc_first[10][100];
char calc_follow[10][100];
int m = 0;
char production[10][10], first[10];
char f[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;
   printf("Enter number of productions :");
   scanf("%d", &count);
   printf("\nEnter %d productions in form A=B where A and B are grammar symbols :\n", count);
   for (i = 0; i < count; i++)
   {
      scanf("%s%c", production[i], &ch);
   }
   int kay;
   char done[100];
```

```
int ptr = -1;
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;
   for (kay = 0; kay <= ptr; kay++)
      if (c == done[kay])
         xxx = 1;
   if (xxx == 1)
      continue;
   findfirst(c, 0, 0);
   ptr += 1;
   done[ptr] = c;
   calc_first[point1][point2++] = c;
   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)
      {
         calc_first[point1][point2++] = first[i];
      }
   }
   jm = n;
   point1++;
}
char donee[100];
ptr = -1;
```

```
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;
   for (kay = 0; kay <= ptr; kay++)
      if (ck == donee[kay])
         xxx = 1;
   if (xxx == 1)
      continue;
   land += 1;
   follow(ck);
   ptr += 1;
   donee[ptr] = ck;
   calc_follow[point1][point2++] = ck;
   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)
      {
         calc_follow[point1][point2++] = f[i];
      }
   }
   km = m;
   point1++;
}
char ter[10];
```

```
    for (k = 0; k < 10; k++)
    {
        ter[k] = '!';
    }
    int ap, vp, sid = 0;
    for (k = 0; k < count; k++)
    {
        for (kay = 0; kay < count; kay++)
        {
            if (!isupper(production[k][kay]) && production[k][kay] != '#' && production[k][kay] != '=' &&
production[k][kay] != '\0')
            {
                vp = 0;
                for (ap = 0; ap < sid; ap++)
                {
                    if (production[k][kay] == ter[ap])
                    {
                        vp = 1;
                        break;
                    }
                }
                if (vp == 0)
                {
                    ter[sid] = production[k][kay];
                    sid++;
                }
            }
        }
    }
    ter[sid] = '$';
    sid++;
    for (ap = 0; ap < sid; ap++);
    char first_prod[100][100];
    for (ap = 0; ap < count; ap++)
    {
        int destiny = 0;
        k = 2;
        int ct = 0;
        char tem[100];
        while (production[ap][k] != '\0')
        {
            if (!isupper(production[ap][k]))
            {
                tem[ct++] = production[ap][k];
```

```
            tem[ct++] = '_';
            tem[ct++] = '\0';
            k++;
            break;
        }
        else
        {
            int zap = 0;
            int tuna = 0;
            for (zap = 0; zap < count; zap++)
            {
                if (calc_first[zap][0] == production[ap][k])
                {
                    for (tuna = 1; tuna < 100; tuna++)
                    {
                        if (calc_first[zap][tuna] != '!')
                        {
                            tem[ct++] = calc_first[zap][tuna];
                        }
                        else
                            break;
                    }
                    break;
                }
            }
            tem[ct++] = '_';
        }
        k++;
    }
    int zap = 0, tuna;
    for (tuna = 0; tuna < ct; tuna++)
    {
        if (tem[tuna] == '#')
        {
            zap = 1;
        }
        else if (tem[tuna] == '_')
        {
            if (zap == 1)
            {
                zap = 0;
            }
            else
                break;
```

```
        }
        else
        {
            first_prod[ap][destiny++] = tem[tuna];
        }
    }
}
char table[100][100 + 1];
ptr = -1;
for (ap = 0; ap < land; ap++)
{
    for (kay = 0; kay < (sid + 1); kay++)
    {
        table[ap][kay] = '!';
    }
}
for (ap = 0; ap < count; ap++)
{
    ck = production[ap][0];
    xxx = 0;
    for (kay = 0; kay <= ptr; kay++)
        if (ck == table[kay][0])
            xxx = 1;
    if (xxx == 1)
        continue;
    else
    {
        ptr = ptr + 1;
        table[ptr][0] = ck;
    }
}
for (ap = 0; ap < count; ap++)
{
    int tuna = 0;
    while (first_prod[ap][tuna] != '\0')
    {
        int to, ni = 0;
        for (to = 0; to < sid; to++)
        {
            if (first_prod[ap][tuna] == ter[to])
            {
                ni = 1;
            }
        }
```

```
        if (ni == 1)
        {
            char xz = production[ap][0];
            int cz = 0;
            while (table[cz][0] != xz)
            {
                cz = cz + 1;
            }
            int vz = 0;
            while (ter[vz] != first_prod[ap][tuna])
            {
                vz = vz + 1;
            }
            table[cz][vz + 1] = (char)(ap + 65);
        }
        tuna++;
    }
}
for (k = 0; k < sid; k++)
{
    for (kay = 0; kay < 100; kay++)
    {
        if (calc_first[k][kay] == '!')
        {
            break;
        }
        else if (calc_first[k][kay] == '#')
        {
            int fz = 1;
            while (calc_follow[k][fz] != '!')
            {
                char xz = production[k][0];
                int cz = 0;
                while (table[cz][0] != xz)
                {
                    cz = cz + 1;
                }
                int vz = 0;
                while (ter[vz] != calc_follow[k][fz])
                {
                    vz = vz + 1;
                }
                table[k][vz + 1] = '#';
                fz++;
```

```
            }
            break;
        }
    }
}
for (ap = 0; ap < land; ap++)
{
    for (kay = 1; kay < (sid + 1); kay++)
    {
        if (table[ap][kay] == '!');
        else if (table[ap][kay] == '#');
        else
        {
            int mum = (int)(table[ap][kay]);
            mum -= 65;
        }
    }
}
int j;
printf("\n\nPlease enter the desired INPUT STRING = ");
char input[100];
scanf("%s%c", input, &ch);
int i_ptr = 0, s_ptr = 1;
char stack[100];
stack[0] = '$';
stack[1] = table[0][0];
while (s_ptr != -1)
{
    int vamp = 0;
    for (vamp = 0; vamp <= s_ptr; vamp++);
    vamp = i_ptr;
    while (input[vamp] != '\0')
    {
        vamp++;
    }
    char her = input[i_ptr];
    char him = stack[s_ptr];
    s_ptr--;
    if (!isupper(him))
    {
        if (her == him)
        {
            i_ptr++;
        }
```

```
    else
    {
        printf("\nString Not Accepted by LL(1) Parser !!\n");
        exit(0);
    }
}
else
{
    for (i = 0; i < sid; i++)
    {
        if (ter[i] == her)
            break;
    }
    char produ[100];
    for (j = 0; j < land; j++)
    {
        if (him == table[j][0])
        {
            if (table[j][i + 1] == '#')
            {
                produ[0] = '#';
                produ[1] = '\0';
            }
            else if (table[j][i + 1] != '!')
            {
                int mum = (int)(table[j][i + 1]);
                mum -= 65;
                strcpy(produ, production[mum]);
            }
            else
            {
                printf("\nString Not Accepted by LL(1) Parser !!\n");
                exit(0);
            }
        }
    }
    int le = strlen(produ);
    le = le - 1;
    if (le == 0)
    {
        continue;
    }
    for (j = le; j >= 2; j--)
    {
```

```
            s_ptr++;
            stack[s_ptr] = produ[j];
        }
    }
  }
  if (input[i_ptr] == '\0')
  {
    printf("\t\t\t\t\t\t\t\t\tYOUR STRING HAS BEEN ACCEPTED !!\n");
  }
  else
    printf("\n\t\t\t\t\t\t\t\t\tYOUR STRING HAS BEEN REJECTED !!\n");
}

void follow(char c)
{
  int i, j;
  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')
        {
          followfirst(production[i][j + 1], i, (j + 2));
        }
        if (production[i][j + 1] == '\0' && c != production[i][0])
        {
          follow(production[i][0]);
        }
      }
    }
  }
}

void findfirst(char c, int q1, int q2)
{
  int j;
  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))
                {
                    findfirst(production[q1][q2], q1, (q2 + 1));
                }
                else
                    first[n++] = '#';
            }
            else if (!isupper(production[j][2]))
            {
                first[n++] = production[j][2];
            }
            else
            {
                findfirst(production[j][2], j, 3);
            }
        }
    }
}

void followfirst(char c, int c1, int c2)
{
    int k;
    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;
        }
        while (calc_first[i][j] != '!')
        {
```

```
        if (calc_first[i][j] != '#')
        {
            f[m++] = calc_first[i][j];
        }
        else
        {
            if (production[c1][c2] == '\0')
            {
                follow(production[c1][0]);
            }
            else
            {
                followfirst(production[c1][c2], c1, c2 + 1);
            }
        }
        j++;
    }
  }
}
```

Output:

```
Enter number of productions :3

Enter 3 productions in form A=B where A and B are grammar symbols :
S=AB
A=aa
B=bb

Please enter the desired INPUT STRING = aabb
YOUR STRING HAS BEEN ACCEPTED
```

Experiment 22

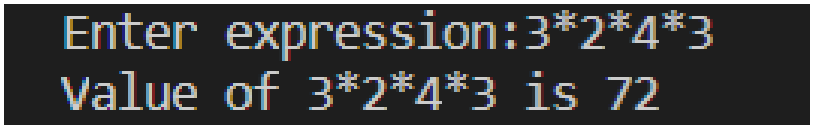
Aim:
Design a parser which accepts a mathematical expression (containing integers only). If the expression is valid, then evaluate the expression else report that the expression is invalid.

Code:

```
def isOperand(c):
    return (c >= '0' and c <= '9')

def value(c):
    return ord(c) - ord('0')

def evaluate(exp):
    len1 = len(exp)
    if (len1 == 0):
        return -1
    res = value(exp[0])
    for i in range(1, len1, 2):
        opr = exp[i]
        opd = exp[i + 1]
        if (isOperand(opd) == False):
            return -1
        if (opr == '+'):
            res += value(opd)
        elif (opr == '-'):
            res -= int(value(opd))
        elif (opr == '*'):
            res *= int(value(opd))
        elif (opr == '/'):
            res /= int(value(opd))
        else:
            return -1
    return res

def main():
    expr1 = input('Enter expression:')
    res = evaluate(expr1)
    print(expr1, "is Invalid") if (res == -1) else print("Value of", expr1, "is", res)
```

```
if __name__ == '__main__':
    main()
```

Output:



Experiment 23

Aim:
WAP to check whether the given production is a reduced production or not in LR(0) canonical item.

Code:

```
def main():
    production = input('Enter production in LR(0) canonical form: ')
    if production[-1] == '.':
        print('Production is reduced.')
    else:
        print('Production is not reduced.')

if __name__ == '__main__':
    main()
```

Output:

```
Enter production in LR(0) canonical form: S->aB.c
Production is not reduced.
```

```
Enter production in LR(0) canonical form: S->aBc.
Production is reduced.
```

Experiment 24

Aim:
WAP to find out all the LR(0) canonical items for a given grammar.

Code:

```
from ordered_set import OrderedSet

I0 = OrderedSet()

def lr0(flag, grammar):
    for i in range(len(grammar[flag])):
        I0.add(f"{flag}->.{grammar[flag][i]}")
        if grammar[flag][i][0].isupper():
            lr0(grammar[flag][i][0], grammar)

def main():
    grammar = dict()
    n = int(input('Enter number of productions:'))
    for i in range(n):
        production = input(f'Enter production {i+1}: ')
        if i == 0:
            start_symbol = production[0]
        if production[0] not in grammar:
            grammar[production[0]] = []
        grammar[production[0]].append(production[3:])
    print('I0 state for LR(0) Canonical Set of Items')
    I0.add(f'{start_symbol}\'->.{start_symbol}')
    lr0(start_symbol, grammar)
    print(*I0, sep='\n')


if __name__ == '__main__':
    main()
```

Output:

```
Enter number of productions:5
Enter production 1: S->AB
Enter production 2: S->BC
Enter production 3: A->aa
Enter production 4: B->bb
Enter production 5: C->cc
I0 state for LR(0) Canonical Set of Items
S'->.S
S->.AB
A->.aa
S->.BC
B->.bb
```

Experiment 25

Aim:
WAP to find out all the LR(1) canonical set of items for a given grammar.

Code:

```
from ordered_set import OrderedSet

I0 = OrderedSet()

def lr1(flag, grammar,lookahead):
    for i in range(len(grammar[flag])):
        I0.add(f"{flag}->.{grammar[flag][i]},{lookahead}")
        if grammar[flag][i][0].isupper():
            if len(grammar[flag][i]) == 1:
                la = lookahead
            elif not grammar[flag][i][1].isupper():
                la = grammar[flag][i][1]
            else:
                la = grammar[grammar[flag][i][1]][0][0]
            lr1(grammar[flag][i][0], grammar, la)

def main():
    grammar = dict()
    n = int(input('Enter number of productions:'))
    for i in range(n):
        production = input(f'Enter production {i+1}: ')
        if i == 0:
            start_symbol = production[0]
        if production[0] not in grammar:
            grammar[production[0]] = []
        grammar[production[0]].append(production[3:])
    print('I0 state for LR(1) Canonical Set of Items')
    I0.add(f'{start_symbol}\'->.{start_symbol},$')
    lr1(start_symbol, grammar, '$')
    print(*I0, sep='\n')

if __name__ == '__main__':
    main()
```

Output:

```
Enter number of productions:5
Enter production 1: S->AB
Enter production 2: S->BC
Enter production 3: A->aa
Enter production 4: B->bb
Enter production 5: C->cc
I0 state for LR(1) Canonical Set of Items
S'->.S,$
S->.AB,$
A->.aa,b
S->.BC,$
B->.bb,c
```