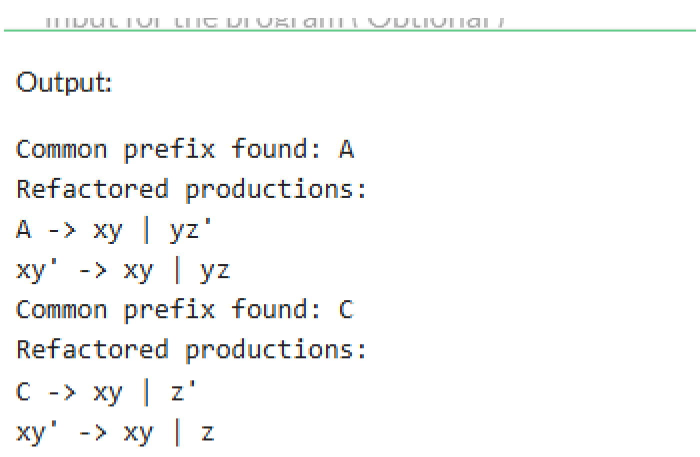
LAB PROGRAMS DAY-2

7. Implement a C program to eliminate left factoring.

```
#include <stdio.h>

#include <string.h>

void eliminateLeftFactoring(char productions[][10], int n)

{

    for (int i = 0; i < n; i++) {

        for (int j = i + 1; j < n; j++) {

            if (productions[i][0] == productions[j][0])

            {

                printf("Common prefix found: %c\n", productions[i][0]);

                printf("Refactored productions:\n");

                printf("%c -> %s | %s'\n", productions[i][0], productions[i] + 1, productions[j] +
1);

                printf("%s' -> %s | %s\n", productions[i] + 1, productions[i] + 1, productions[j] +
1);

            }

        }

    }

}

int main() {

    char productions[5][10] = {"Axy", "Ayz", "Bz", "Cxy", "Cz"};

    int n = 5;
    eliminateLeftFactoring(productions, n);
    return 0;
}
```



Output:

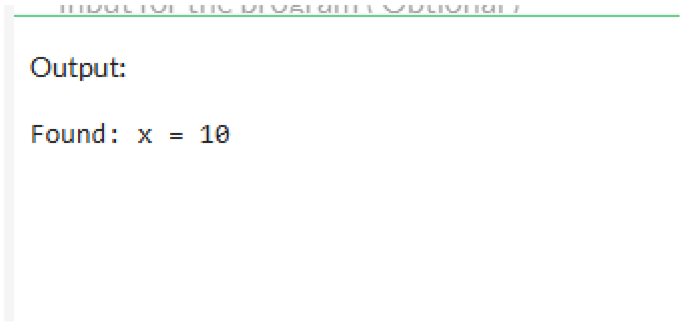
```
Common prefix found: A
Refactored productions:
A -> xy | yz'
xy' -> xy | yz
Common prefix found: C
Refactored productions:
C -> xy | z'
xy' -> xy | z
```

8. Implement a C program to perform symbol table operations.

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

#define TABLE_SIZE 100

typedef struct Symbol {
    char name[50];
    int value;
    struct Symbol* next;
} Symbol;

typedef struct SymbolTable {
    Symbol* table[TABLE_SIZE];
} SymbolTable;
```

```
unsigned int hash(char* name) {
    unsigned int hash = 0;
    while (*name) {
        hash = (hash << 5) + *name++;
    }
    return hash % TABLE_SIZE;
}

SymbolTable* createSymbolTable() {
    SymbolTable* st = malloc(sizeof(SymbolTable));
    memset(st->table, 0, sizeof(st->table));
    return st;
}

void insert(SymbolTable* st, char* name, int value) {
    unsigned int index = hash(name);
    Symbol* newSymbol = malloc(sizeof(Symbol));
    strcpy(newSymbol->name, name);
    newSymbol->value = value;
    newSymbol->next = st->table[index];
    st->table[index] = newSymbol;
}

Symbol* lookup(SymbolTable* st, char* name) {
    unsigned int index = hash(name);
    Symbol* symbol = st->table[index];
    while (symbol) {
        if (strcmp(symbol->name, name) == 0) {
            return symbol;
        }
        symbol = symbol->next;
    }
    return NULL;
}

void deleteSymbol(SymbolTable* st, char* name) {
    unsigned int index = hash(name);
    Symbol* symbol = st->table[index];
    Symbol* prev = NULL;
    while (symbol) {
        if (strcmp(symbol->name, name) == 0) {
            if (prev) {
                prev->next = symbol->next;
            } else {
                st->table[index] = symbol->next;
            }
            free(symbol);
            return;
        }
        prev = symbol;
        symbol = symbol->next;
    }
}
```

```
int main() {
    SymbolTable* st = createSymbolTable();
    insert(st, "x", 10);
    Symbol* found = lookup(st, "x");
    if (found) {
        printf("Found: %s = %d\n", found->name, found->value);
    }
    deleteSymbol(st, "x");
    free(st);
    return 0;
}
```



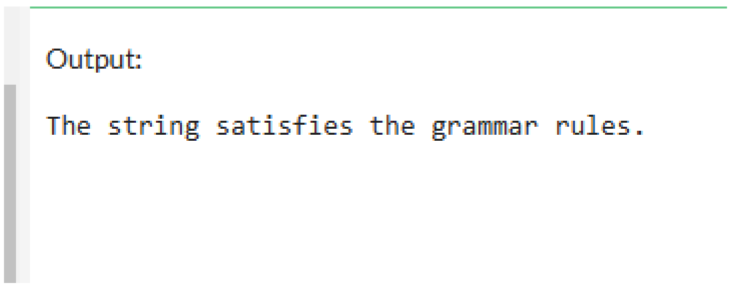
Output:

```
Found: x = 10
```

9.All languages have Grammar. When people frame a sentence we usually say whether the sentence is framed as per the rules of the Grammar or Not. Similarly use the same ideology , implement to check whether the given input string is satisfying the grammar or not .

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
int isValidGrammar(const char *str)
{
    if (str == NULL || strlen(str) == 0)
    {
        return 0;
    }
    if (!isupper(str[0]))
    {
        return 0;
    }
    if (str[strlen(str) - 1] != '.')
    {
        return 0;
    }
    return 1;
}
int main()
{
    const char *testString = "Hello, world.";
    if (isValidGrammar(testString))
    {
        printf("The string satisfies the grammar rules.\n");
    }
    else
    {
        printf("The string does not satisfy the grammar rules.\n");
    }
    return 0;
}
```

}


Output:

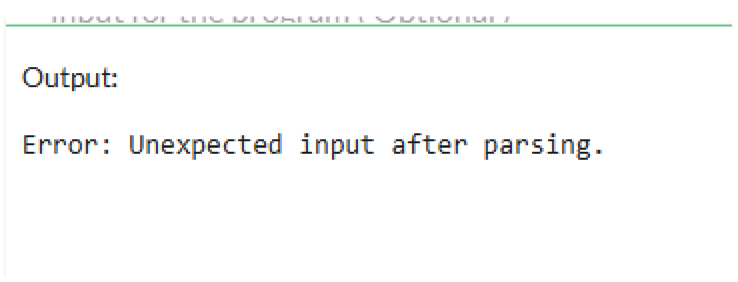
```
The string satisfies the grammar rules.
```

10.Write a C program to construct recursive descent parsing.

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
const char *input;
char lookahead;
void next_token() {
    lookahead = *input++;
}
void expression();
void term();
void factor();
void expression()
{
    term();
    while (lookahead == '+' || lookahead == '-') {
        next_token();
        term();
    }
}
```

```
void term()
 {
    factor();
    while (lookahead == '*' || lookahead == '/') {
        next_token();
        factor();
    }
}
void factor()
{
    if (isdigit(lookahead))
{
        while (isdigit(lookahead))
 {
            next_token();
        }
    }
else if (lookahead == '(')
 {
        next_token();
        expression();
        if (lookahead == ')')
 {
            next_token();
        } else
 {
            printf("Error: Missing closing parenthesis\n");
```

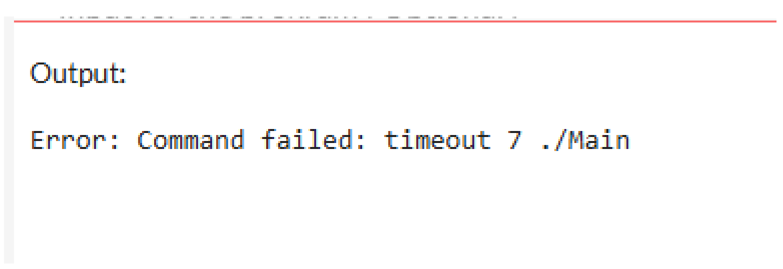
```
                exit(1);
            }
        }
    else
    {
            printf("Error: Unexpected character %c\n", lookahead);
            exit(1);
        }
    }
int main()
{
    input = "3 + 5 * ( 10 - 4 )";
    next_token();
    expression();
    if (lookahead == '\0')
     {
            printf("Parsing completed successfully.\n");
        }
     else
     {
            printf("Error: Unexpected input after parsing.\n");
        }
    return 0;
}
```



```
Output:

Error: Unexpected input after parsing.
```

11. In a class of Grade 3, Mathematics Teacher asked for the Acronym PEMDAS?. All of them are thinking for a while. A smart kid of the class Kishore of the class says it is Parentheses, Exponentiation, Multiplication, Division, Addition, Subtraction. Can you write a C Program to help the students to understand about the operator precedence parsing for an expression containing more than one operator, the order of evaluation depends on the order of operations.

```
#include <stdio.h>

#include <stdlib.h>

int main()

{

    double result;

    char expression[100];

    printf("Enter a mathematical expression (e.g., 3 + 5 * 2): ");

    fgets(expression, sizeof(expression), stdin);

    result = atof(expression);

    printf("The result of the expression is: %.2f\n", result);

    return 0;

}
```



```
Output:

Error: Command failed: timeout 7 ./Main
```

12.The main function of the Intermediate code generation is producing three address code statements for a given input expression. The three address codes help in determining the sequence in which operations are actioned by the compiler. The key work of Intermediate code generators is to simplify the process of Code Generator. Write a C Program to Generate the Three address code representation for the given input statement.

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>


void generateTAC(char *expression)

{

    char *token;

    char temp[10];

    int count = 1;

    token = strtok(expression, " ");

    while (token != NULL) {

        if (strcmp(token, "+") == 0 || strcmp(token, "-") == 0 ||

            strcmp(token, "*") == 0 || strcmp(token, "/") == 0) {

            printf("t%d = %s %s %s\n", count, temp, token, strtok(NULL, " "));

            sprintf(temp, "t%d", count);

            count++;

        } else {

            sprintf(temp, "%s", token);

        }

        token = strtok(NULL, " ");

    }

}

int main()
```

```
{
    char expression[100];
    printf("Enter an expression: ");
    fgets(expression, sizeof(expression), stdin);
    expression[strcspn(expression, "\n")] = 0;
    generateTAC(expression);
    return 0;
}
```