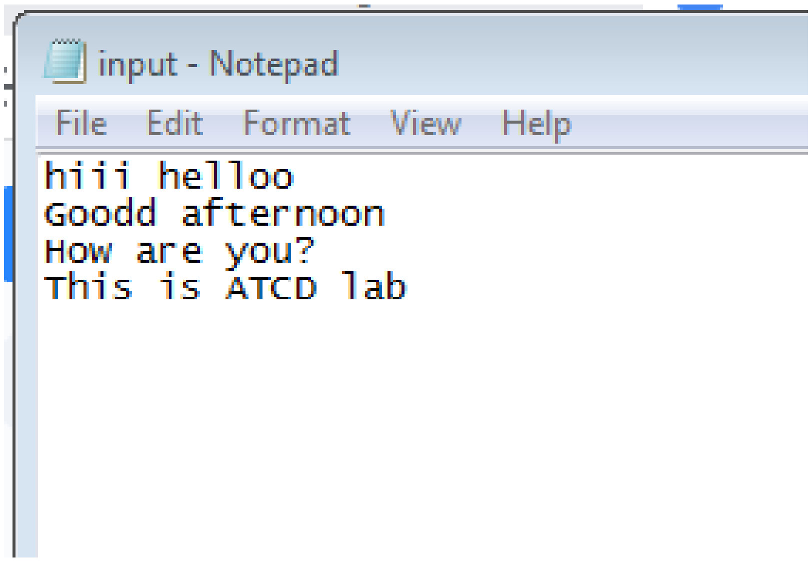
Practical - 1

Develop a 'C' Program finds white spaces, number of newline characters from the given input.

→ Code :

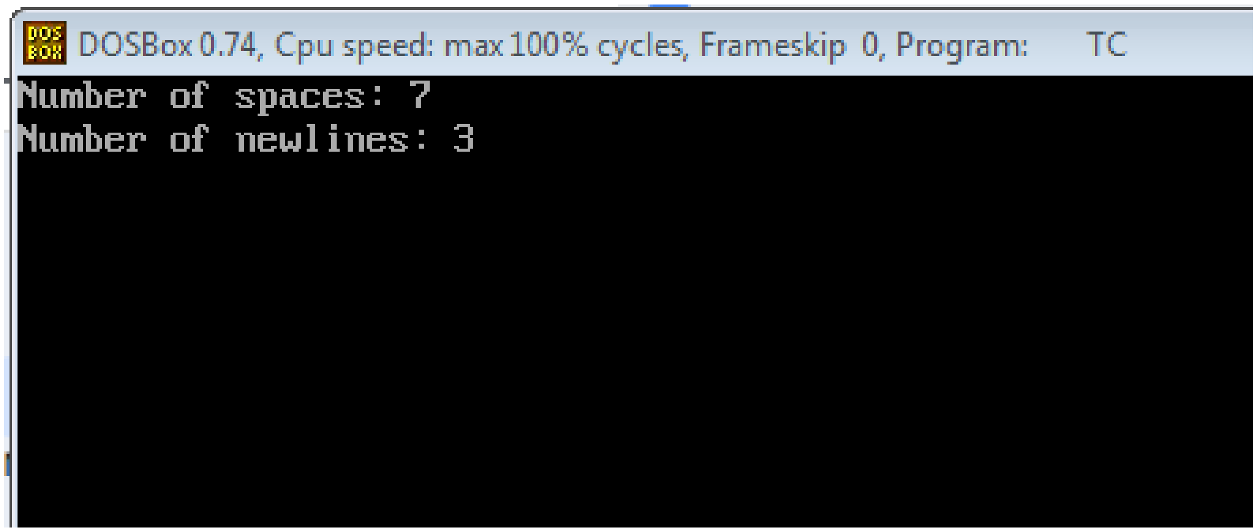
```
#include <stdio.h>
#include <conio.h>

void main() {
    FILE *fp;
    char ch;
    int spaces = 0, newlines = 0;

    clrscr();  // Clear screen (Turbo C specific)

    fp = fopen("input.txt", "r");  // Open file for reading

    if (fp == NULL) {
        printf("Error opening file.\n");
    } else {
        while ((ch = fgetc(fp)) != EOF) {
            if (ch == ' ')
                spaces++;
            if (ch == '\n')
                newlines++;
        }

        fclose(fp);  // Close the file

        printf("Number of spaces: %d\n", spaces);
        printf("Number of newlines: %d\n", newlines);
    }

    getch();  // Wait for key press
}
```

→ Input :



```
hiii helloo
Goodd afternoon
How are you?
This is ATCD lab
```

→ Output :


DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Program: TC

```
Number of spaces: 7
Number of newlines: 3
```

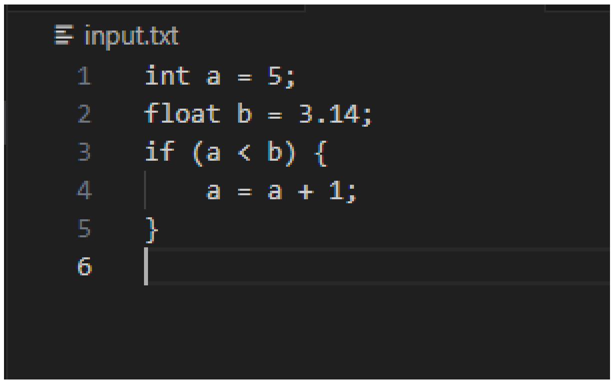
Practical - 2

Implement a lexical analyzer (Scanner program) to recognize identifiers, keywords and constants from the given input file and store them separately.

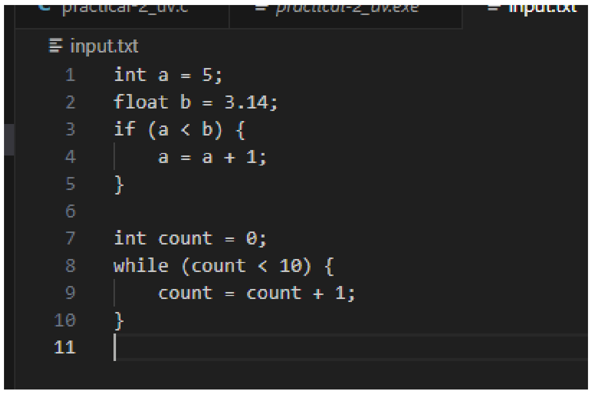
→ Code :

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

char keywords[32][10] = {
    "auto", "break", "case", "char", "const", "continue",
"default", "do", "double",
    "else", "enum", "extern", "float", "for", "goto", "if",
"int", "long", "register",
    "return", "short", "signed", "sizeof", "static", "struct",
"switch", "typedef",
    "union", "unsigned", "void", "volatile", "while"
};

int isKeyword(char *word) {
    int i;
    for (i = 0; i < 32; i++) {
        if (strcmp(keywords[i], word) == 0)
            return 1;
    }
    return 0;
}

int isConstant(char *word) {
    int i = 0, hasDecimal = 0;

    if (word[0] == '\0') return 0;

    while (word[i]) {
        if (!isdigit(word[i])) {
            if (word[i] == '.' && !hasDecimal)
                hasDecimal = 1;
            else
                return 0;
        }
        i++;
    }
    return 1;
}

int isType(char *word) {
    if (strcmp(word, "int") == 0 || strcmp(word, "float") == 0
|| strcmp(word, "char") == 0 ||
        strcmp(word, "double") == 0 || strcmp(word, "long") == 0
|| strcmp(word, "short") == 0) {
        return 1;
    }
    return 0;
}

void main() {
    FILE *fp;
    char ch, buffer[50], type[10];
    int i, isVar = 0;

    fp = fopen("input.txt", "r");

    if (fp == NULL) {
        printf("Cannot open file.\n");
        return;
    }

    printf("Variables/Identifiers with Types:\n");
    printf("Constants:\n");

    while ((ch = fgetc(fp)) != EOF) {
        i = 0;

        if (isalpha(ch) || ch == '_' || isdigit(ch)) {
            buffer[i++] = ch;
            while ((ch = fgetc(fp)) != EOF && (isalnum(ch) || ch
== '_')) {
                buffer[i++] = ch;
            }

            buffer[i] = '\0';
            ungetc(ch, fp);
            if (isType(buffer)) {
                strcpy(type, buffer);
                isVar = 1;
            }
            else if (!isKeyword(buffer)) {
                if (isConstant(buffer)) {
                    printf("Constant: %s\n", buffer);
                }
                else if (isVar) {
                    printf("Variable/Identifier: %s (Type:
%s)\n", buffer, type);
                    isVar = 0;
                }
            }
        }
    }

    fclose(fp);
}
```

→ Input :

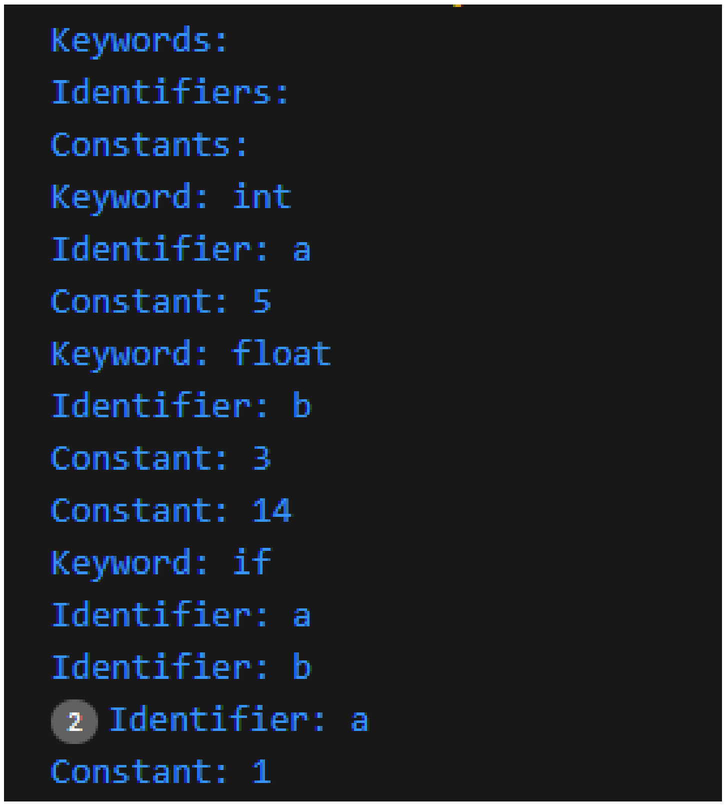


```
int a = 5;
float b = 3.14;
if (a < b) {
    a = a + 1;
}

```

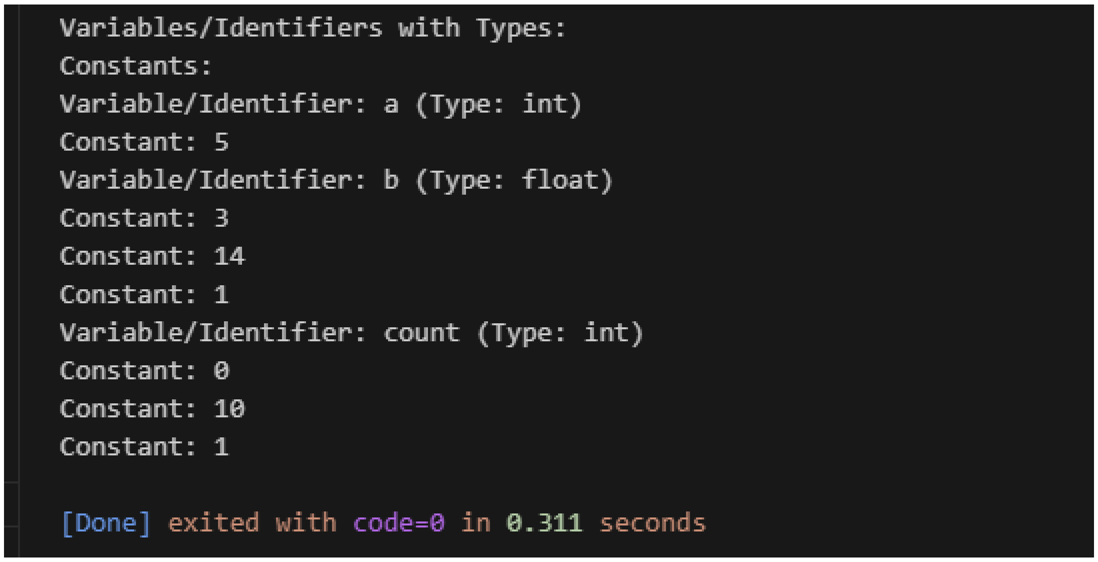


```
int a = 5;
float b = 3.14;
if (a < b) {
    a = a + 1;
}

int count = 0;
while (count < 10) {
    count = count + 1;
}

```

→ Output :



```
Keywords:
Identifiers:
Constants:
Keyword: int
Identifier: a
Constant: 5
Keyword: float
Identifier: b
Constant: 3
Constant: 14
Keyword: if
Identifier: a
Identifier: b
2 Identifier: a
Constant: 1
```



```
Variables/Identifiers with Types:
Constants:
Variable/Identifier: a (Type: int)
Constant: 5
Variable/Identifier: b (Type: float)
Constant: 3
Constant: 14
Constant: 1
Variable/Identifier: count (Type: int)
Constant: 0
Constant: 10
Constant: 1

[Done] exited with code=0 in 0.311 seconds
```

Practical - 3

Write a C program to simulate lexical analyzer to validate arithmetic operators, relational operators and logical operators.

→ Code :

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

// Function to check for arithmetic operators
int isArithmeticOperator(const char *token) {
    return (strcmp(token, "+") == 0 || strcmp(token, "-") == 0
||
            strcmp(token, "*") == 0 || strcmp(token, "/") == 0
||
            strcmp(token, "%") == 0);
}

// Function to check for relational operators
int isRelationalOperator(const char *token) {
    return (strcmp(token, "==") == 0 || strcmp(token, "!=") == 0
||
            strcmp(token, ">") == 0 || strcmp(token, "<") == 0
||
            strcmp(token, ">=") == 0 || strcmp(token, "<=") ==
0);
}

// Function to check for logical operators
int isLogicalOperator(const char *token) {
    return (strcmp(token, "&&") == 0 || strcmp(token, "||") == 0
|| strcmp(token, "!") == 0);
}
```

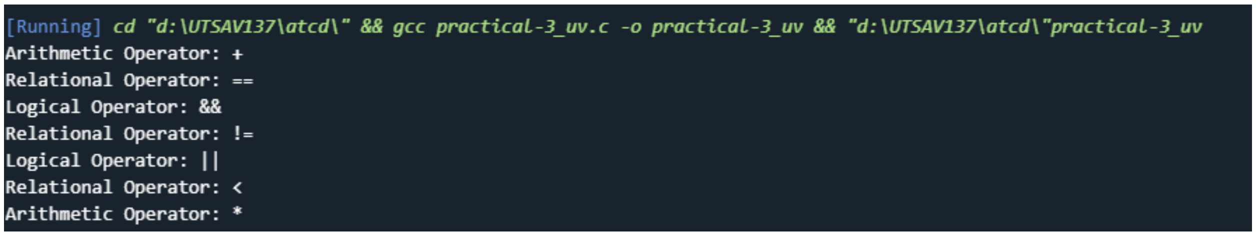
```
// Tokenize input based on whitespace and special characters
void analyzeLexically(char *line) {
    char token[3];
    int i = 0, j = 0;

    while (line[i] != '\0') {
        if (isspace(line[i])) {
            i++;
            continue;
        }

        if (ispunct(line[i])) {
            token[0] = line[i];
            token[1] = '\0';
            token[2] = '\0';

            // Check for 2-character operators (e.g., ==, >=,
&&, etc.)
            if ((line[i+1] == '=' || line[i+1] == '&' ||
line[i+1] == '|') && !isspace(line[i+1])) {
                token[1] = line[i+1];
                token[2] = '\0';
                i++;
            }

            if (isArithmeticOperator(token)) {
                printf("Arithmetic Operator: %s\n", token);
            } else if (isRelationalOperator(token)) {
                printf("Relational Operator: %s\n", token);
            } else if (isLogicalOperator(token)) {
                printf("Logical Operator: %s\n", token);
            }

            i++;
        } else {
            // Skip identifiers or constants (not needed for
this task)
            while (!isspace(line[i]) && !ispunct(line[i]) &&
line[i] != '\0') {
                i++;
            }
        }
    }
}

int main() {
    FILE *file = fopen("input.txt", "r");
    if (!file) {
        perror("Error opening file");
        return 1;
    }

    char line[256];
    while (fgets(line, sizeof(line), file)) {
        analyzeLexically(line);
    }

    fclose(file);
    return 0;
}
```

→ **Input :**

1. a + b == c && d != e || f < g * h
2. u * v + p - q ^ r ! g

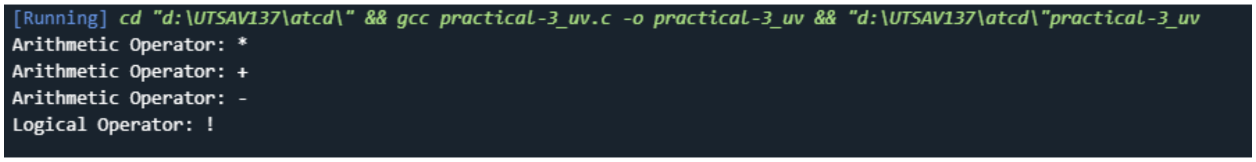
→ **Output :**

1.



```
[Running] cd "d:\UTSAV137\atcd\" && gcc practical-3_uv.c -o practical-3_uv && "d:\UTSAV137\atcd\"practical-3_uv
Arithmetic Operator: +
Relational Operator: ==
Logical Operator: &&
Relational Operator: !=
Logical Operator: ||
Relational Operator: <
Arithmetic Operator: *
```

2.



```
[Running] cd "d:\UTSAV137\atcd\" && gcc practical-3_uv.c -o practical-3_uv && "d:\UTSAV137\atcd\"practical-3_uv
Arithmetic Operator: *
Arithmetic Operator: +
Arithmetic Operator: -
Logical Operator: !
```

Practical - 4

Convert the following regular expression (R.E.) into DFA and Write a ‘C’ program to simulate the DFA for given input Strings.

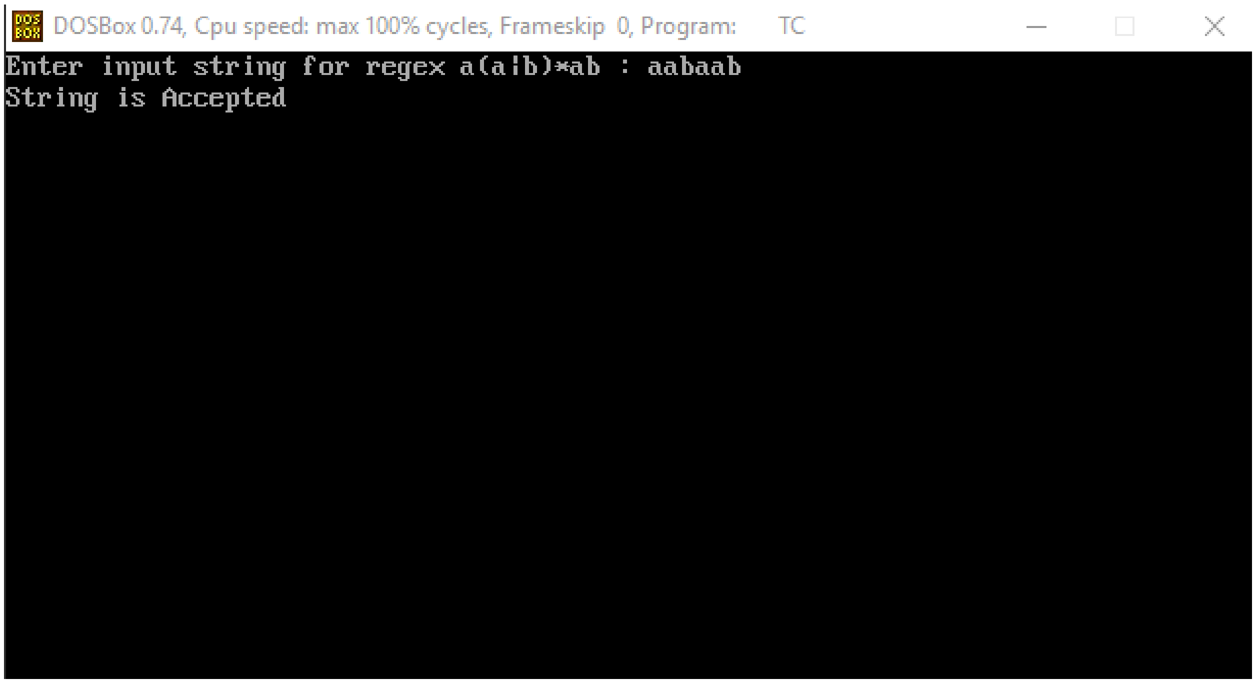
(i) a(a|b)* ab

→ Code :

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

int isAccepted(char *input) {
    int state = 0;
    int i = 0;
    char c;

    while ((c = input[i]) != '\0') {
        switch(state) {
            case 0:
                if (c == 'a')
                    state = 1;
                else
                    return 0;
                break;
            case 1:
                if (c == 'a')
                    state = 2;
                else if (c == 'b')
                    state = 1;
                else
                    return 0;
                break;
            case 2:
                if (c == 'a')
                    state = 2;
                else if (c == 'b')
                    state = 3;
                else
```

```
                    return 0;
                break;
            case 3:
                if (c == 'a')
                    state = 2;
                else if (c == 'b')
                    state = 1;
                else
                    return 0;
                break;
            default:
                return 0;
        }
        i++;
    }

    if (state == 3)
        return 1;
    else
        return 0;
}

void main() {
    char input[100];
    clrscr();

    printf("Enter input string for regex a(a|b)*ab : ");
    scanf("%s", input);

    if (isAccepted(input))
        printf("String is Accepted\n");
    else
        printf("String is Rejected\n");

    getch();
}
```

→ Output :


DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Program: TC

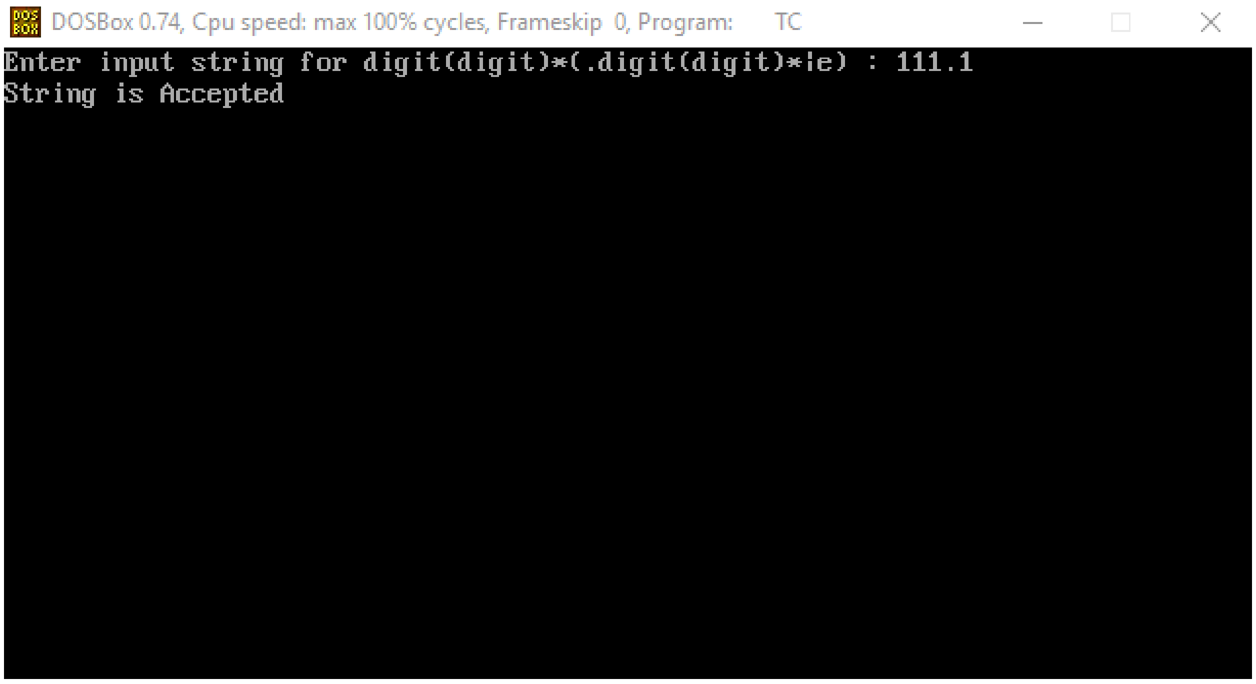
```
Enter input string for regex a(a|b)*ab : aabaab
String is Accepted
```

(ii) digit(digit)*(.digit(digit)*|ε)

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

int isDigit(char c) {
    return (c >= '0' && c <= '9');
}

int isAccepted(char *input) {
    int state = 0;  // q0
    int i = 0;
    char c;

    while ((c = input[i]) != '\0') {
        switch(state) {
            case 0:
                if (isDigit(c))
                    state = 1;
                else
                    return 0;
                break;
            case 1:
                if (isDigit(c))
                    state = 1;
                else if (c == '.')
                    state = 2;
                else
                    return 0;
                break;
            case 2:
                if (isDigit(c))
                    state = 3;
                else
                    return 0;
                break;
            case 3:
                if (isDigit(c))
                    state = 3;
                else
                    return 0;
                break;
            default:
                return 0;
        }
        i++;
    }

    if (state == 1 || state == 3)
        return 1;  // accepted
    else
        return 0;  // rejected
}

void main() {
    char input[100];
    clrscr();

    printf("Enter input string for
digit(digit)*(.digit(digit)*|e) : ");
    scanf("%s", input);

    if (isAccepted(input))
```

```
        printf("String is Accepted\n");
    else
        printf("String is Rejected\n");

    getch();
}
```

→ Output :



DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Program: TC

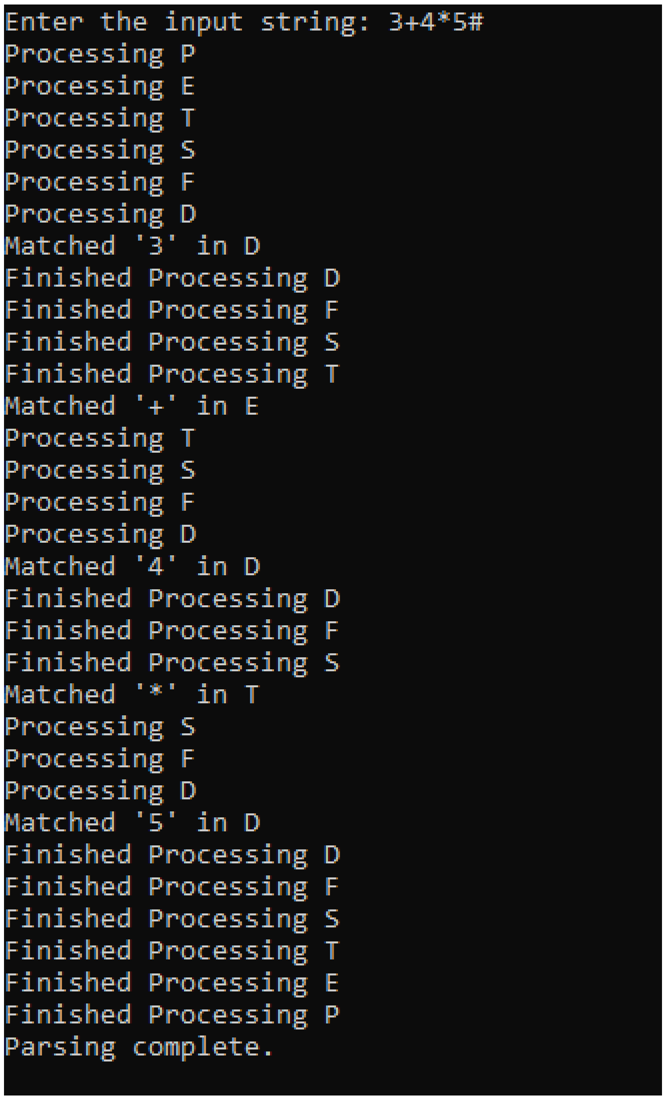
```
Enter input string for digit(digit)*(.digit(digit)*|e) : 111.1
String is Accepted
```

Practical - 5

Implement Recursive Descent Parser program in 'C' for the following Grammar.
P ---> E '#'
E ---> T {('+'|'-') T}
T ---> S {('*'|'/') S}
S ---> F '^' S | F
F ---> D | '(' E ')'
D ---> 0|1|......|9.
Write a program in a way that it will trace the processing of different non-terminals of above grammar for given input string.

→ Code :
```
#include <stdio.h>
#include <ctype.h>
#include <string.h>

char input[100];
int index = 0;

char lookahead() {
    return input[index];
}

void match(char expected) {
    if (lookahead() == expected) {
        index++;
    } else {
        printf("Syntax error: Expected '%c', but found '%c'\n",
expected, lookahead());
        exit(1);
    }
}

void P() {
    printf("Processing P\n");
    E();
    match('#');
    printf("Finished Processing P\n");
}

void E() {
    printf("Processing E\n");
    T();
    while (lookahead() == '+' || lookahead() == '-') {
        printf("Matched '%c' in E\n", lookahead());
        index++;
        T();
    }
    printf("Finished Processing E\n");
}

void T() {
    printf("Processing T\n");
    S();
    while (lookahead() == '*' || lookahead() == '/') {
        printf("Matched '%c' in T\n", lookahead());
        index++;
        S();
    }
    printf("Finished Processing T\n");
}

void S() {
    printf("Processing S\n");
    F();
    if (lookahead() == '^') {
        printf("Matched '^' in S\n");
        index++;
        S();
    }
    printf("Finished Processing S\n");
}

void F() {
    printf("Processing F\n");
    if (isdigit(lookahead())) {
        D();
    } else if (lookahead() == '(') {
        match('(');
        E();
        match(')');
    } else {
        printf("Syntax error: Expected digit or '(' but found
'%c'\n", lookahead());
        exit(1);
    }
    printf("Finished Processing F\n");
}

void D() {
    printf("Processing D\n");
    if (isdigit(lookahead())) {
        printf("Matched '%c' in D\n", lookahead());
        index++;
    } else {
        printf("Syntax error: Expected a digit, but found
'%c'\n", lookahead());
        exit(1);
    }
    printf("Finished Processing D\n");
}

void parse(char* str) {
    strcpy(input, str);
    index = 0;
    P();
}

int main() {
    char inputString[100];
    printf("Enter the input string: ");
    fgets(inputString, sizeof(inputString), stdin);
    inputString[strcspn(inputString, "\n")] = '\0';

    parse(inputString);
    printf("Parsing complete.\n");
    return 0;
}
```

→ Output :



```
Enter the input string: 3+4*5#
Processing P
Processing E
Processing T
Processing S
Processing F
Processing D
Matched '3' in D
Finished Processing D
Finished Processing F
Finished Processing S
Finished Processing T
Matched '+' in E
Processing T
Processing S
Processing F
Processing D
Matched '4' in D
Finished Processing D
Finished Processing F
Finished Processing S
Matched '*' in T
Processing S
Processing F
Processing D
Matched '5' in D
Finished Processing D
Finished Processing F
Finished Processing S
Finished Processing T
Finished Processing E
Finished Processing P
Parsing complete.
```

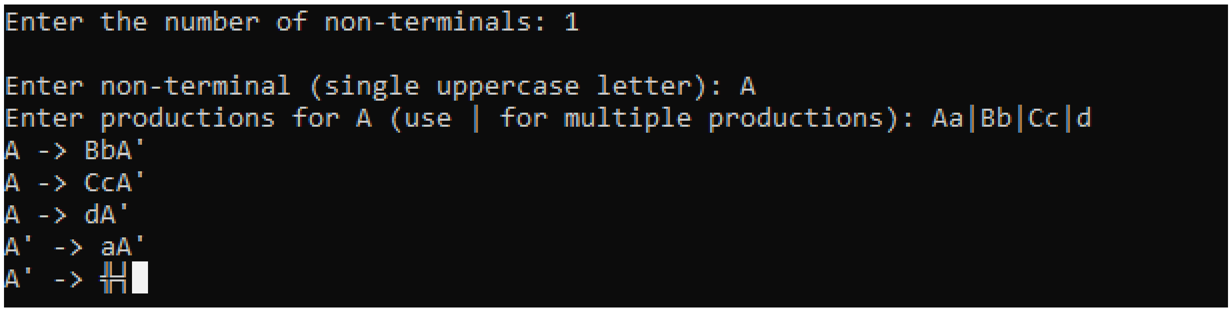
Practical - 6

Write a program to remove left recursion from a given grammar.

→ Code :

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

#define MAX 100

void removeLeftRecursion(char nonTerminal, char *productions) {
    char alpha[MAX][MAX], beta[MAX][MAX];
    int alphaCount = 0, betaCount = 0;

    char *token = strtok(productions, "|");

    while (token != NULL) {
        if (token[0] == nonTerminal) {
            // Left-recursive: A -> Aα
            strcpy(alpha[alphaCount++], token + 1);
        } else {
            // Non-left-recursive: A -> β
            strcpy(beta[betaCount++], token);
        }
        token = strtok(NULL, "|");
    }

    if (alphaCount == 0) {
        // No left recursion
        printf("%c -> %s\n", nonTerminal, productions);
    } else {
        // Create new non-terminal as a string
        char newNonTerminal[3];
        snprintf(newNonTerminal, sizeof(newNonTerminal), "%c'",
nonTerminal);

        // Print β rules: A -> βA'
        for (int i = 0; i < betaCount; i++) {
            printf("%c -> %s%s\n", nonTerminal, beta[i],
newNonTerminal);
        }

        // Print α rules: A' -> αA'
        for (int i = 0; i < alphaCount; i++) {
            printf("%s -> %s%s\n", newNonTerminal, alpha[i],
newNonTerminal);
        }

        // Add epsilon production to A'
        printf("%s -> ε\n", newNonTerminal);
    }
}

int main() {
    int n;
    char nonTerminal;
    char productions[MAX];

    printf("Enter the number of non-terminals: ");
    scanf("%d", &n);
    getchar(); // Clear newline

    for (int i = 0; i < n; i++) {
        printf("\nEnter non-terminal (single uppercase letter):
");
        scanf("%c", &nonTerminal);
        getchar(); // Clear newline

        printf("Enter productions for %c (use | for multiple
productions): ", nonTerminal);
        fgets(productions, sizeof(productions), stdin);
        productions[strcspn(productions, "\n")] = 0; // Remove
newline

        removeLeftRecursion(nonTerminal, productions);
    }

    return 0;
}
```

→ Output :



```
Enter the number of non-terminals: 1

Enter non-terminal (single uppercase letter): A
Enter productions for A (use | for multiple productions): Aa|Bb|Cc|d
A -> BbA'
A -> CcA'
A -> dA'
A' -> aA'
A' -> ╫
```

Practical - 7

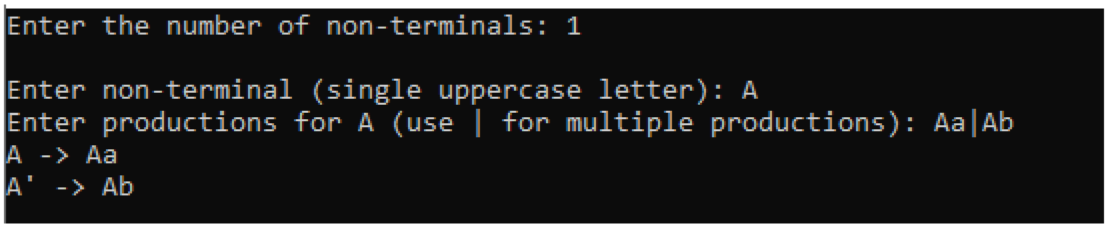
Write a program to left factor the given grammar.

→ Code :

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

#define MAX 100
#define ALPHABET_SIZE 26

// Structure to store the production for a non-terminal
typedef struct {
    char prefix[MAX];
    char suffix[MAX][MAX];
    int suffixCount;
} Production;

// Function to remove left factoring from the grammar
void leftFactor(char nonTerminal, char *productions) {
    Production prod;
    int i = 0;
    int len = strlen(productions);
    char *token = strtok(productions, "|");

    while (token != NULL) {
        // Check if the first production starts with the same
prefix
        if (i == 0) {
            // Save the first prefix (beginning of the
production)
            strcpy(prod.prefix, token);
        } else {
            // If not matching prefix, store the suffix
            if (strncmp(token, prod.prefix, strlen(prod.prefix))
== 0) {
                // This is a common prefix, add to the suffix
                strcpy(prod.suffix[prod.suffixCount++], token +
strlen(prod.prefix));
```

```
            } else {
                // Add this token to the suffix as is
                strcpy(prod.suffix[prod.suffixCount++], token);
            }
        }
        token = strtok(NULL, "|");
        i++;
    }

    // If there's a common prefix, factor it
    if (prod.suffixCount > 0) {
        char newNonTerminal[3];
        snprintf(newNonTerminal, sizeof(newNonTerminal), "%c'",
nonTerminal);

        // Print the new production after factoring
        printf("%c -> %s%c\n", nonTerminal, prod.prefix,
newNonTerminal);

        // Print the suffix rules for the new non-terminal
        for (i = 0; i < prod.suffixCount; i++) {
            printf("%s -> %s\n", newNonTerminal,
prod.suffix[i]);
        }
    } else {
        // No factoring required, just print the original
productions
        printf("%c -> %s\n", nonTerminal, productions);
    }
}

int main() {
    int n;
    char nonTerminal;
    char productions[MAX];

    printf("Enter the number of non-terminals: ");
    scanf("%d", &n);
    getchar(); // Clear newline

    for (int i = 0; i < n; i++) {
        printf("\nEnter non-terminal (single uppercase letter): 
");
        scanf("%c", &nonTerminal);
        getchar(); // Clear newline

        printf("Enter productions for %c (use | for multiple 
productions): ", nonTerminal);
        fgets(productions, sizeof(productions), stdin);
        productions[strcspn(productions, "\n")] = 0; // Remove 
newline

        leftFactor(nonTerminal, productions);
    }

    return 0;
}
```

→ Output :



```
Enter the number of non-terminals: 1

Enter non-terminal (single uppercase letter): A
Enter productions for A (use | for multiple productions): Aa|Ab
A -> Aa
A' -> Ab
```