Date:

Practical 1

AIM: Write a Lexical Analyzer program that identifies any 10 keywords from C language and identifiers following all the naming conventions of the C program.

THEORY:

Lexical analysis is the first phase of a compiler. It reads the source code character by character, identifies tokens, and categorizes them based on their type.
: A token is a string of characters grouped together to form a meaningful entity in a programming language.

1) **Keywords**: Reserved words in C that have a predefined meaning.
2) **Identifiers**: User-defined names for variables, functions, etc., that follow specific naming conventions in C. They must follow these naming conventions:
    1) The first character must be an alphabet (a-z, A-Z) or an underscore (_).
    2) Subsequent characters can be alphabets, digits (0-9), or underscores.
    3) Identifiers are case-sensitive.
    4) Keywords cannot be used as identifiers.

: Components of a Lexical Analyzer

A lexical analyzer consists of several main components:

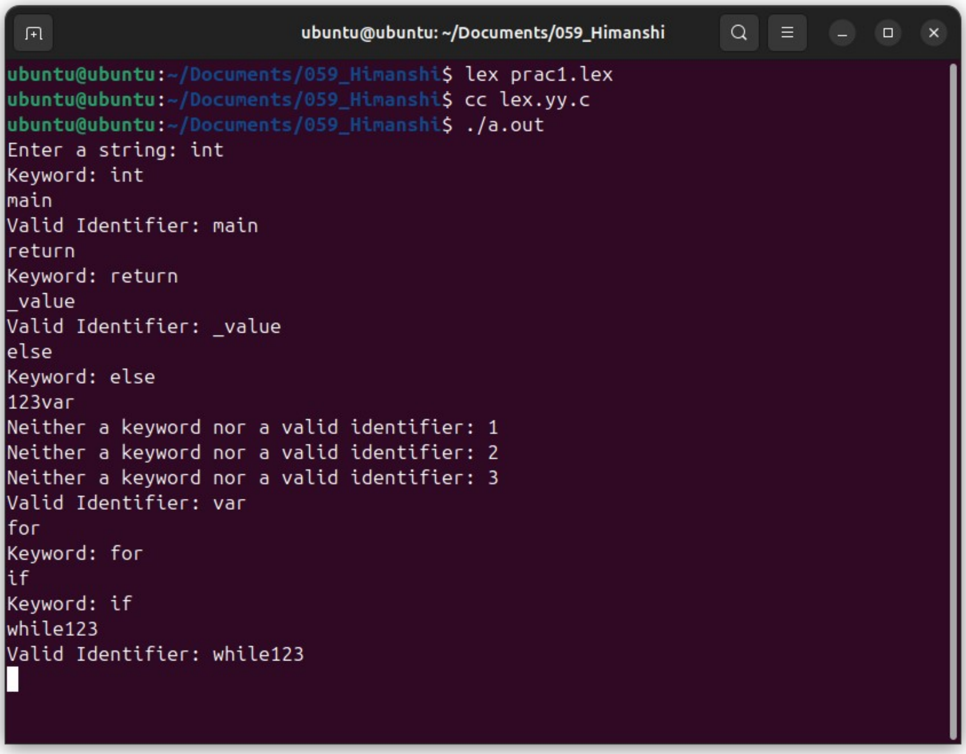
1) **Input Buffer**: Stores the input source code.
2) **Scanner/Tokenizer**: Scans the input character by character, recognizing lexemes and classifying them into tokens.
3) **Symbol Table**: Stores information about identifiers.

CODE:

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

#define keyword_count 10
char keywords[keyword_count][10] = {
   "void", "int", "float", "char", "double", "return", "if", "else", "for",
"while"
};

int is_keyword(char* str) {
for (int i = 0; i < keyword_count; i++) {
if (strcmp(str, keywords[i]) == 0) {
return 1;
}
}
```

```
return 0;
}
%}

%option noyywrap

%%

[ \t\n]+
"void"|"int"|"float"|"char"|"double"|"return"|"if"|"else"|"for"|"while" {
printf("Keyword: %s\n", yytext);
}
[a-zA-Z_][a-zA-Z0-9_]*     {
if (is_keyword(yytext)) {
printf("Keyword: %s\n", yytext);
} else {
printf("Valid Identifier: %s\n", yytext);
}
}
.                        { printf("Neither a keyword nor a valid identifier: %s\
n", yytext); }
%%

int main() {
printf("Enter a string: ");
yylex();
return 0;
}
```

OUTPUT:



```
ubuntu@ubuntu: ~/Documents/059_Himanshi
ubuntu@ubuntu:~/Documents/059_Himanshi$ lex prac1.lex
ubuntu@ubuntu:~/Documents/059_Himanshi$ cc lex.yy.c
ubuntu@ubuntu:~/Documents/059_Himanshi$ ./a.out
Enter a string: int
Keyword: int
main
Valid Identifier: main
return
Keyword: return
_value
Valid Identifier: _value
else
Keyword: else
123var
Neither a keyword nor a valid identifier: 1
Neither a keyword nor a valid identifier: 2
Neither a keyword nor a valid identifier: 3
Valid Identifier: var
for
Keyword: for
if
Keyword: if
while123
Valid Identifier: while123
```

LEARNING OUTCOMES:

Date:

Practical 2

AIM: Write a C program that takes as input string from the text file (let's say input.txt), and identifies and counts the frequency of the keywords appearing in that string.

THEORY:

1. File Handling:
   - The program will read the contents of the file `input.txt`.
   - It opens the file, processes the contents, and then closes the file.
2. Tokenization:
   - The program will parse the input text word by word. Each word will be compared to a predefined list of C keywords.
   - String Tokenization: The input string will be split into tokens (words) based on delimiters such as spaces, newlines, punctuation marks, etc.
3. Keyword Identification:
   - The program will have a predefined list of C keywords. Each word (token) from the input will be checked to see if it matches any of the keywords.
4. Frequency Counting:
   - For each keyword found, its frequency count will be increased. The program will keep track of the count of each keyword and display it after the complete input has been processed.
5. Output:
   - The program will output the list of keywords and their respective frequencies.

CODE:

counter.cpp

```
#include <iostream>
#include <fstream>
#include <unordered_map>
#include <string>
#include <sstream>
#include <vector>
#include <algorithm>

const std::vector<std::string> c_keywords = {
   "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"
};

// Function to check if a word is a keyword
bool is_keyword(const std::string& word) {
    return std::find(c_keywords.begin(), c_keywords.end(), word) !=
c_keywords.end();
}

int main() {
    std::ifstream infile("input.txt");
    if (!infile) {
        std::cerr << "Error: input.txt file not found!" << std::endl;
        return 1;
    }

    std::string line, program_code;
    std::unordered_map<std::string, int> keyword_count;

    // Read the file and store its contents
    while (std::getline(infile, line)) {
        program_code += line + "\n";

        std::istringstream iss(line);
        std::string word;
        while (iss >> word) {
            if (is_keyword(word)) {
                keyword_count[word]++;
            }
        }
    }
```

```
    infile.close();

    std::cout << "Program read from input.txt:\n";
    std::cout << program_code << std::endl;

    std::cout << "Keywords and their frequencies:\n";
    for (const auto& pair : keyword_count) {
        std::cout << pair.first << ": " << pair.second << std::endl;
    }

    return 0;
}
```

OUTPUT:

```
ubuntu@ubuntu: ~/Documents/059_Himanshi
ubuntu@ubuntu:~/Documents/059_Himanshi$ g++ -o counter counter.cpp
ubuntu@ubuntu:~/Documents/059_Himanshi$ ./counter
Program read from input.txt:
#include <stdio.h>

int main() {
    int a, b;
    printf("Enter two numbers: ");
    scanf("%d %d", &a, &b);
    int sum = a + b;
    printf("Sum: %d\n", sum);
    return 0;
}

Keywords and their frequencies:
return: 1
int: 3
ubuntu@ubuntu:~/Documents/059_Himanshi$
```

LEARNING OUTCOMES:

Date:

Practical 3

AIM: Write a Syntax Analyzer program using Yacc tool that will have grammar rules for the operators : *,/,%.

THEORY:

1) A **Syntax Analyzer**, also known as a **Parser**, is the second phase of the compilation process following **Lexical Analysis**.
2) While the Lexical Analyzer identifies tokens in the source code, the Syntax Analyzer checks whether the tokens form a valid sequence according to the grammar of the programming language.
3) The Yacc (Yet Another Compiler Compiler) tool is widely used to implement parsers and generates the syntax analyzer based on grammar rules defined by the programmer.

Yacc (Yet Another Compiler Compiler)

1) Yacc is a tool used to generate a parser that interprets and checks the syntax of a given source code according to the rules defined by the grammar.
2) Yacc works alongside **Lex** (Lexical Analyzer) to create a complete syntax analyzer.
3) Yacc generates a **LALR (Look-Ahead LR)** parser, which is an efficient type of bottom-up parser used in many compilers.

Yacc Program Structure

The Yacc file is typically divided into three sections:

1) **Definition Section**: Define tokens and declare precedence and associativity of operators.
2) **Rules Section**: Define grammar rules and their corresponding actions.
3) **User Code Section**: Optional C code that defines supporting functions (such as printing output or handling errors).

CODE:

pthree.lex

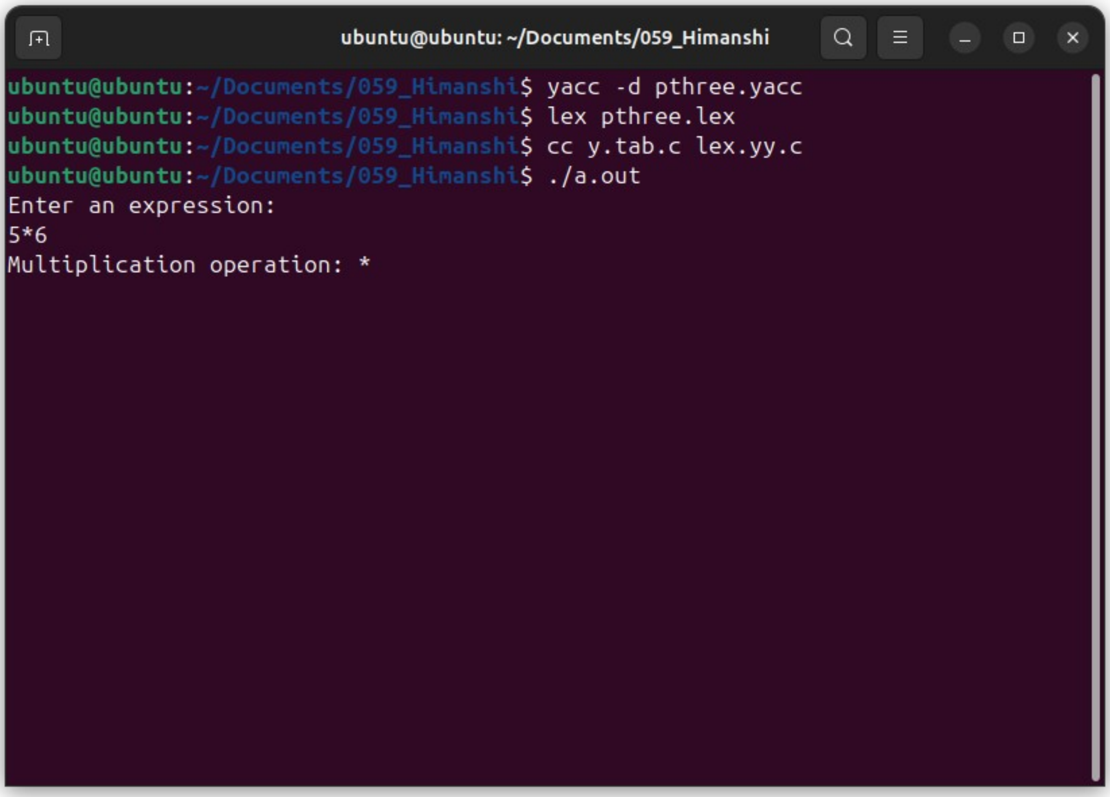
```
%{
#include "y.tab.h"
%}

%%

[0-9]       { return DIGIT; }
[ \t\n]     { /* Ignore whitespace */ }
.           { return yytext[0]; }
```

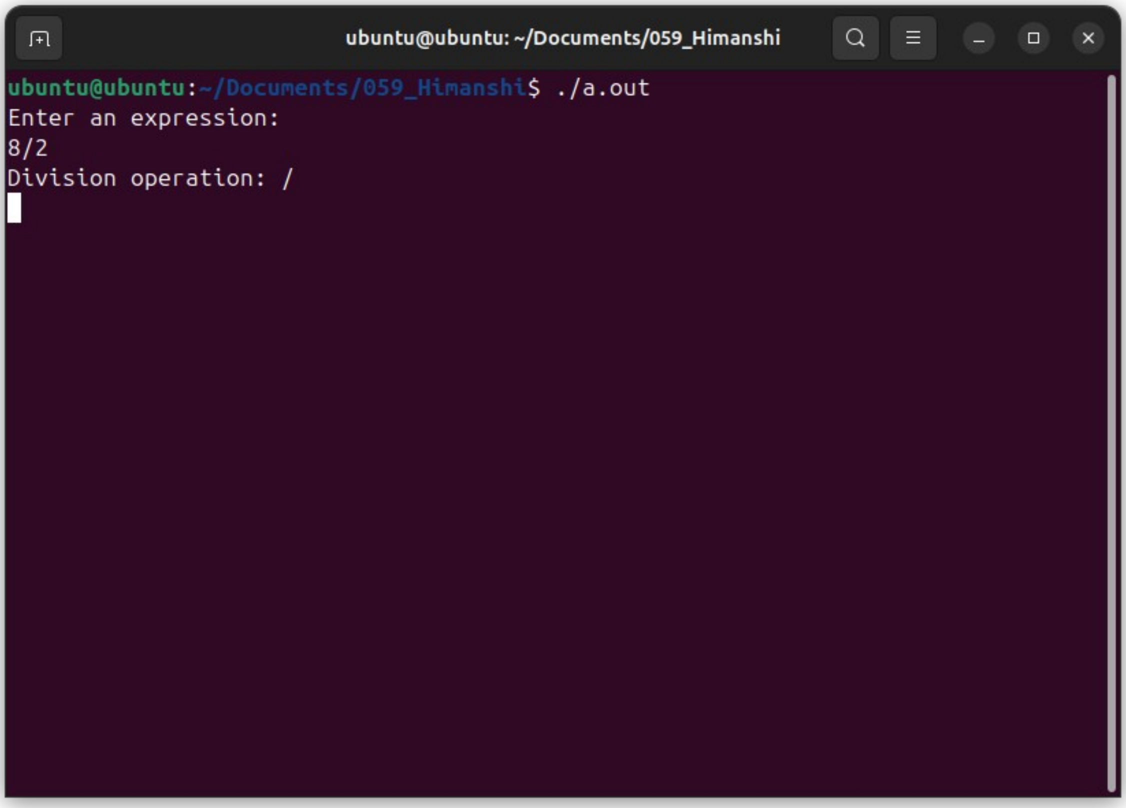
```
%%

int yywrap() {
    return 1;
}
```

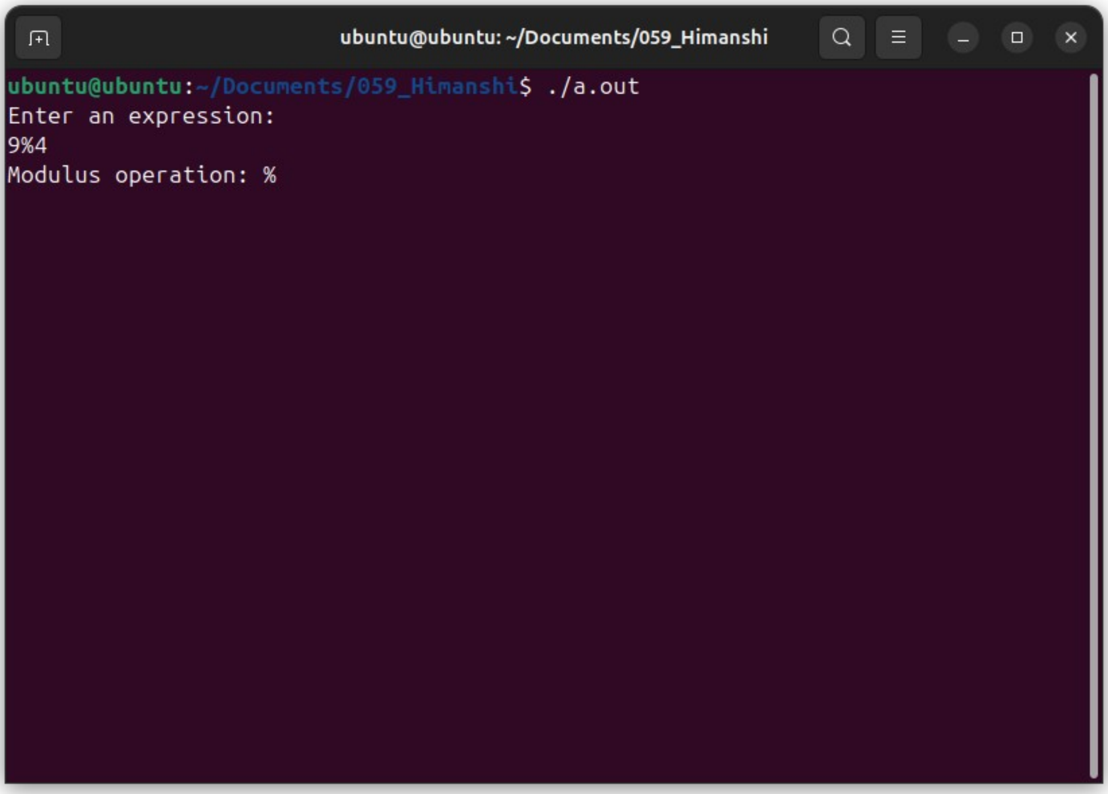
pthree.yacc

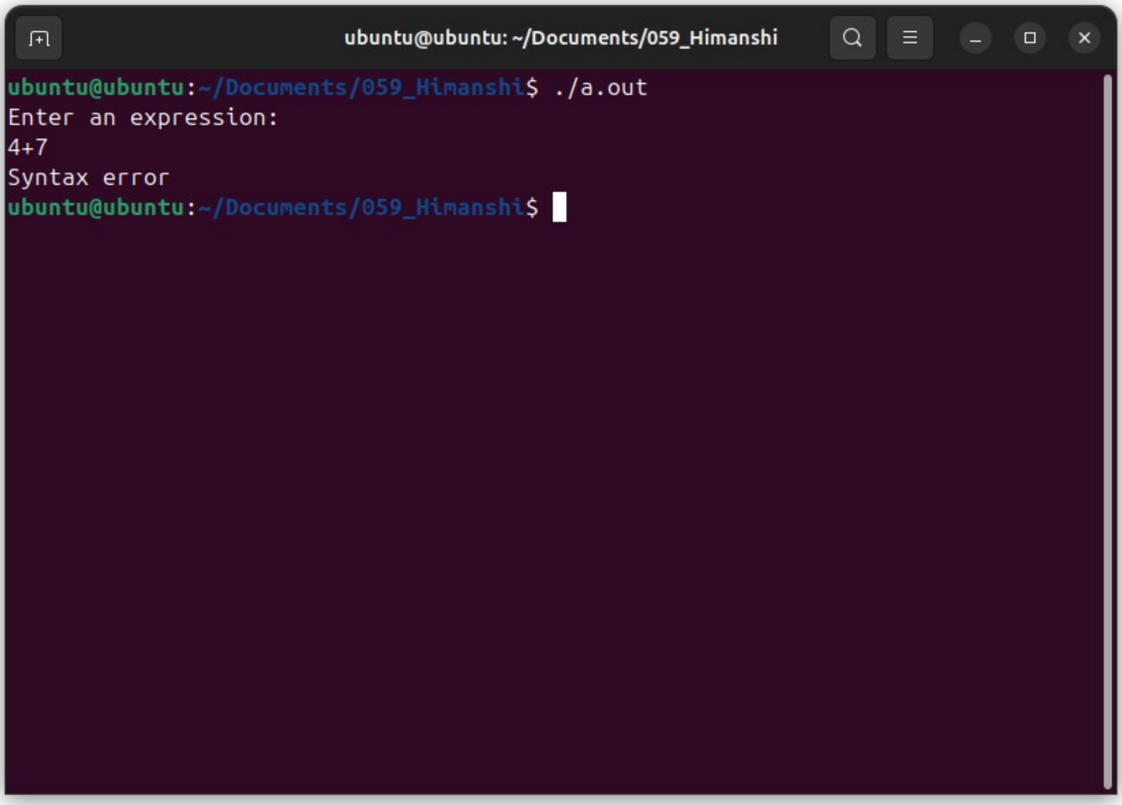
```
%{
#include <stdio.h>

int yylex();
void yyerror(const char *s);
%}

%token DIGIT
%left '*' '/' '%'

%%

expr:  expr '*' expr       { printf("Multiplication operation: *\n"); }
    | expr '/' expr        { printf("Division operation: /\n"); }
    | expr '%' expr        { printf("Modulus operation: %%\n"); }
    | DIGIT                { /* Base case: single digit */ }
    ;

%%

int main() {
    printf("Enter an expression:\n");
    return yyparse();
}

void yyerror(const char *s) {
    printf("Syntax error\n");
}
```

OUTPUT:


ubuntu@ubuntu: ~/Documents/059_Himanshi
ubuntu@ubuntu:~/Documents/059_Himanshi$ yacc -d pthree.yacc
ubuntu@ubuntu:~/Documents/059_Himanshi$ lex pthree.lex
ubuntu@ubuntu:~/Documents/059_Himanshi$ cc y.tab.c lex.yy.c
ubuntu@ubuntu:~/Documents/059_Himanshi$ ./a.out
Enter an expression:
5*6
Multiplication operation: *


ubuntu@ubuntu: ~/Documents/059_Himanshi
ubuntu@ubuntu:~/Documents/059_Himanshi$ ./a.out
Enter an expression:
8/2
Division operation: /





LEARNING OUTCOMES:

Date:

Practical 4

AIM: To write a C program that takes the single line production rule in a string as input and checks if it has Left-Recursion or not and give the unambiguous grammar, in case, if it has Left- Recursion.

THEORY:

1) Grammar defines the syntactical structure of a programming language. A grammar consists of a set of production rules that describe how strings in a language are derived from a starting symbol (also known as the start symbol).
2) Left recursion is problematic for parsers that use a top-down parsing strategy.
3) When a parser encounters a left-recursive rule, it could keep applying the rule infinitely without making any progress. Therefore, left-recursive grammars need to be transformed into an equivalent grammar that is non-left-recursive.

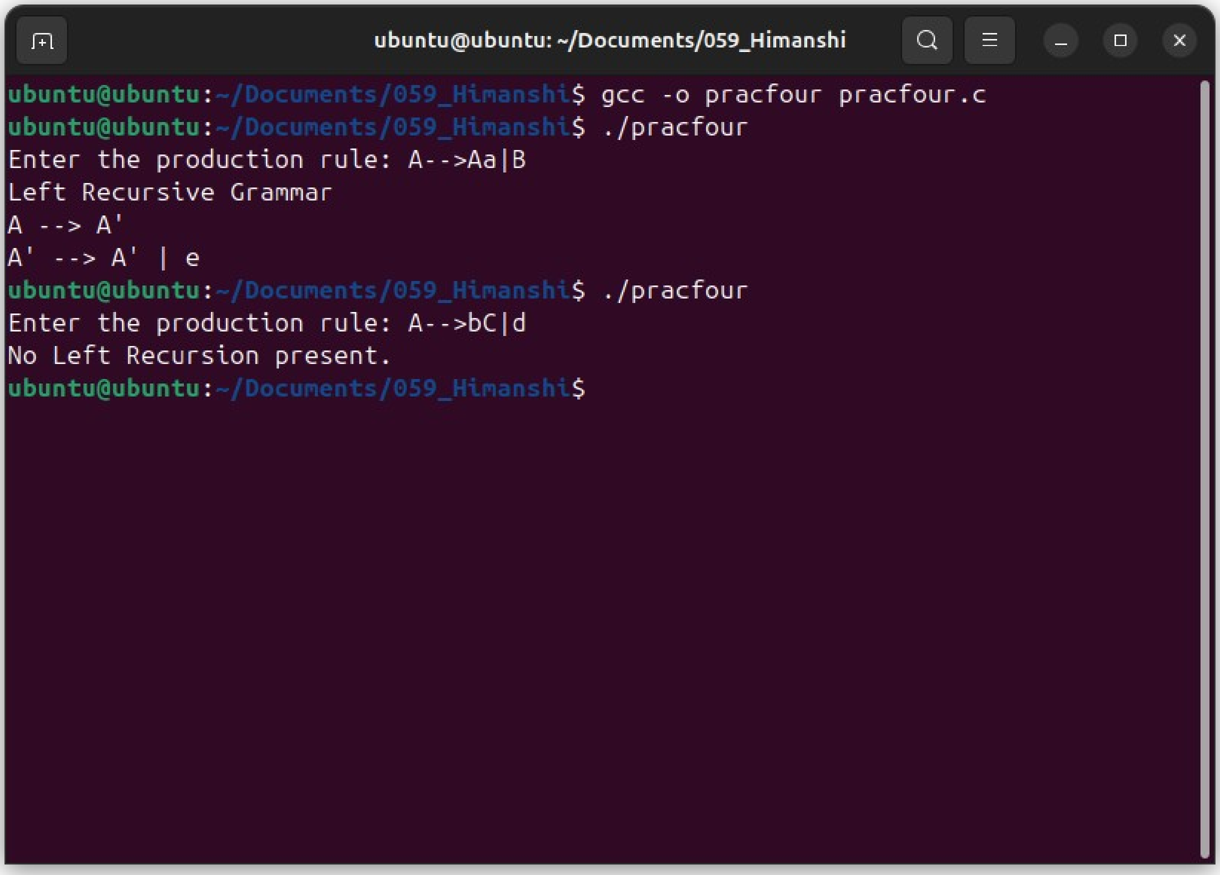
CODE:

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

#define MAX 100

void checkLeftRecursion(char* input) {
    char nonTerminal = input[0];
    char alpha[MAX], beta[MAX];
    char* production = strstr(input, "-->") + 3;
    char *token = strtok(production, "|");
    bool hasLeftRecursion = false;
```

```c
    int alphaCount = 0, betaCount = 0;

    while (token != NULL) {
        if (token[0] == nonTerminal) {
            hasLeftRecursion = true;
            strcpy(alpha + alphaCount, token + 1);
            alphaCount += strlen(token) - 1;
            strcat(alpha, "|");
        } else {
            strcpy(beta + betaCount, token);
            betaCount += strlen(token);
            strcat(beta, "|");
        }
        token = strtok(NULL, "|");
    }

    if (betaCount > 0) beta[betaCount - 1] = '\0';
    if (alphaCount > 0) alpha[alphaCount - 1] = '\0';

    if (hasLeftRecursion) {
        printf("Left Recursive Grammar\n");
        printf("%c --> %s%c'\n", nonTerminal, beta, nonTerminal);
        printf("%c' --> %s%c' | e\n", nonTerminal, alpha, nonTerminal);
    } else {
        printf("No Left Recursion present.\n");
    }
}

int main() {
    char input[MAX];

    printf("Enter the production rule: ");
    fgets(input, MAX, stdin);
    input[strcspn(input, "\n")] = 0;
```

```
    checkLeftRecursion(input);

    return 0;
}
```

OUTPUT:



LEARNING OUTCOMES:

Date:

Practical 5

AIM: To write a C program that takes the single line production rule in a string as input and checks if it has Left-Factoring or not and give the unambiguous grammar, in case, if it has Left- Factoring.

THEORY:

Ambiguity in Grammar

Ambiguity in grammar arises when a string generated by the grammar has more than one possible parse tree or derivation. Ambiguous grammars can lead to problems in the parsing phase of a compiler, especially for top-down parsers, which require a single, unambiguous derivation. Two common sources of ambiguity are Left Recursion and Left-Factoring.

Left-Factoring

Left-Factoring is a technique used to remove ambiguity from a grammar when two or more alternatives of a production rule share a common prefix. In such cases, the parser can face difficulty in choosing the correct alternative at the beginning of parsing because the decision to choose a path cannot be made until more input symbols are read.

$A \rightarrow \alpha\beta1 \mid \alpha\beta2$

When there are common prefixes in the right-hand side of a production, parsers may struggle to decide which rule to apply without looking ahead further in the input string. This issue is especially significant for top-down parsers like Recursive Descent Parsers, which cannot backtrack. To resolve this, left-factoring must be applied to rewrite the grammar and eliminate the common prefix.

CODE:

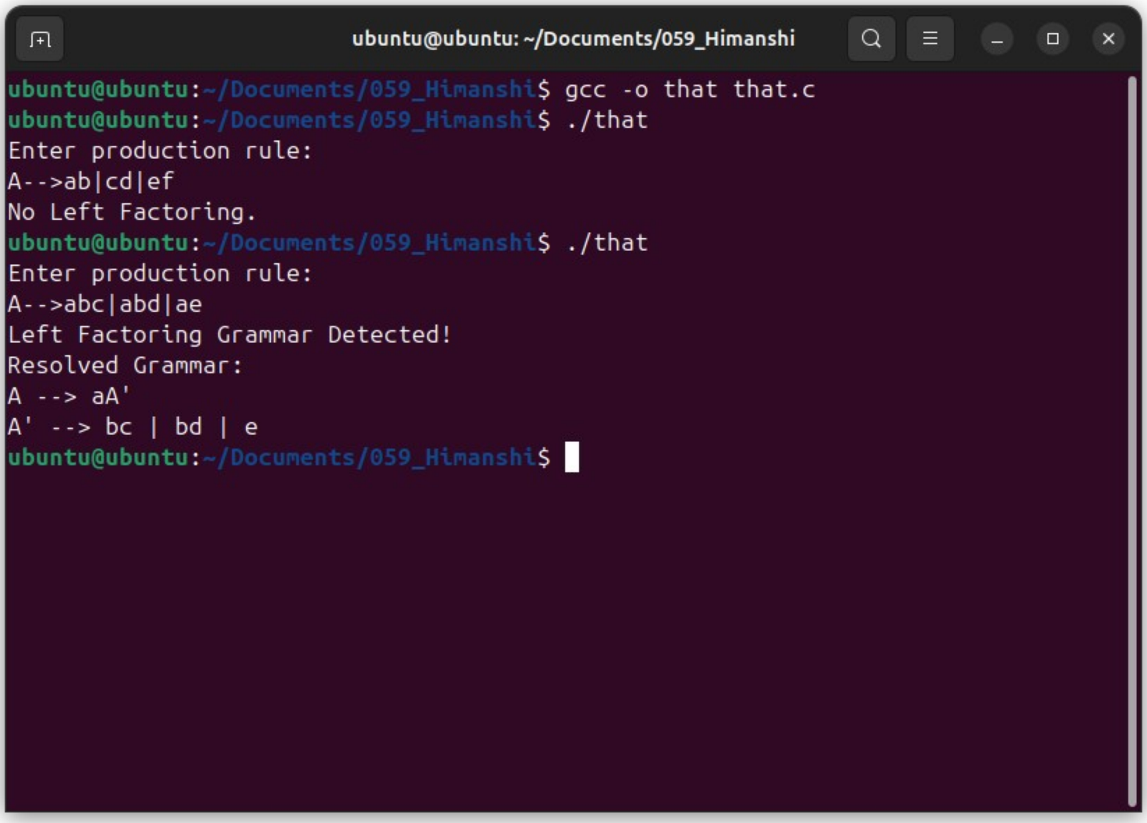
```
#include <stdio.h>
#include <string.h>
#include <stdbool.h>

void findLongestCommonPrefix(char productions[][100], int n, char* prefix)
{
    int min_len = strlen(productions[0]);

    for (int i = 1; i < n; i++) {
        if (strlen(productions[i]) < min_len) {
            min_len = strlen(productions[i]);
        }
    }

    for (int i = 0; i < min_len; i++) {
```

```
        char current_char = productions[0][i];
        bool is_common = true;

        for (int j = 1; j < n; j++) {
            if (productions[j][i] != current_char) {
                is_common = false;
                break;
            }
        }

        if (is_common) {
            prefix[i] = current_char;
        } else {
            break;
        }
    }
}

void checkLeftFactoring(char non_terminal, char* right_side) {
    char productions[10][100];
    char temp[100];
    char* token;
    int count = 0;

    token = strtok(right_side, "|");
    while (token != NULL) {
        strcpy(productions[count++], token);
        token = strtok(NULL, "|");
    }

    char prefix[100] = "";
    findLongestCommonPrefix(productions, count, prefix);

    int prefix_len = strlen(prefix);
```

```
    if (prefix_len == 0) {
        printf("No Left Factoring.\n");
        return;
    }

    printf("Left Factoring Grammar Detected!\n");
    printf("Resolved Grammar:\n");

    printf("%c --> %s%c'\n", non_terminal, prefix, non_terminal);

    printf("%c' --> ", non_terminal);
    bool first = true;

    for (int i = 0; i < count; i++) {
        if (strncmp(productions[i], prefix, prefix_len) == 0) {
            if (!first) {
                printf(" | ");
            }
            if (strlen(productions[i] + prefix_len) == 0) {
                printf("e");
            } else {
                printf("%s", productions[i] + prefix_len);
            }
            first = false;
        } else {
            printf(" | %s", productions[i]);
        }
    }
    printf("\n");
}

int main() {
    char input[200];
    char non_terminal;
    char right_side[200];
```

```
    printf("Enter production rule:\n");
    fgets(input, sizeof(input), stdin);
    input[strcspn(input, "\n")] = 0;

    if (sscanf(input, "%c-->%s", &non_terminal, right_side) != 2) {
        printf("Invalid production rule format!\n");
        return 1;
    }

    checkLeftFactoring(non_terminal, right_side);

    return 0;
}
```

OUTPUT:



LEARNING OUTCOMES: