

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

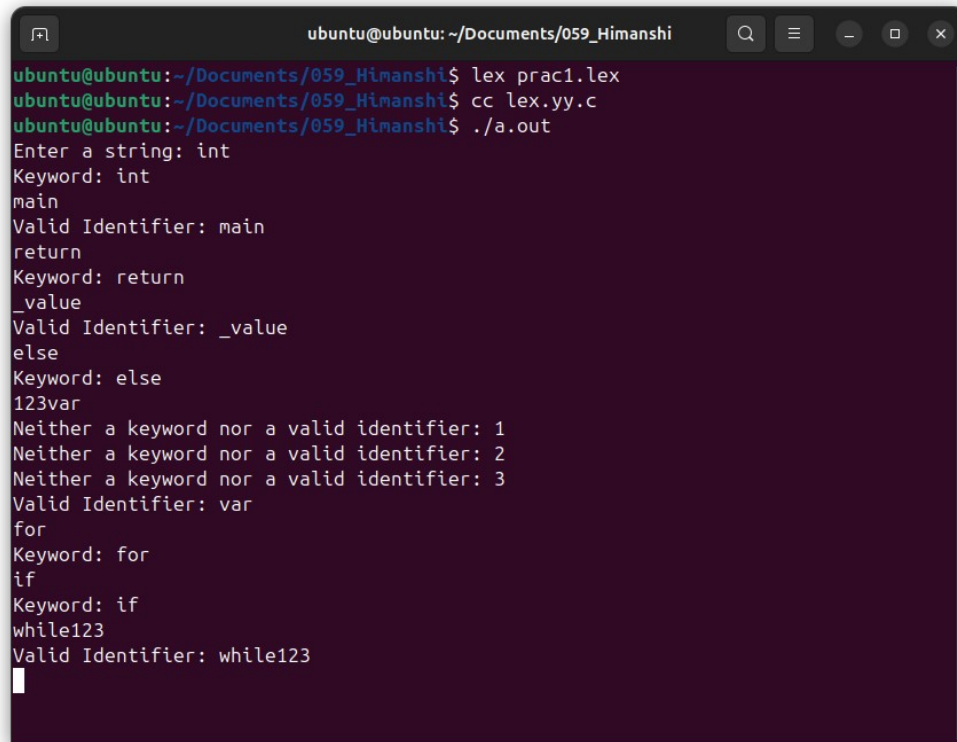
%%

[ \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:

A terminal window titled 'ubuntu@ubuntu: ~/Documents/059_Himanshi' with standard window controls. The terminal shows the execution of a Lex program. The user enters 'int' and 'return', which are recognized as keywords. Then, they enter '_value', which is recognized as a valid identifier. Next, they enter 'else', recognized as a keyword. Then, they enter '123var', which is rejected as neither a keyword nor a valid identifier. This is followed by 'var', which is recognized as a valid identifier. Then, they enter 'for', recognized as a keyword. Then, they enter 'if', recognized as a keyword. Then, they enter 'while123', which is recognized as a valid identifier. The terminal ends with a cursor on a new line.

```
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();

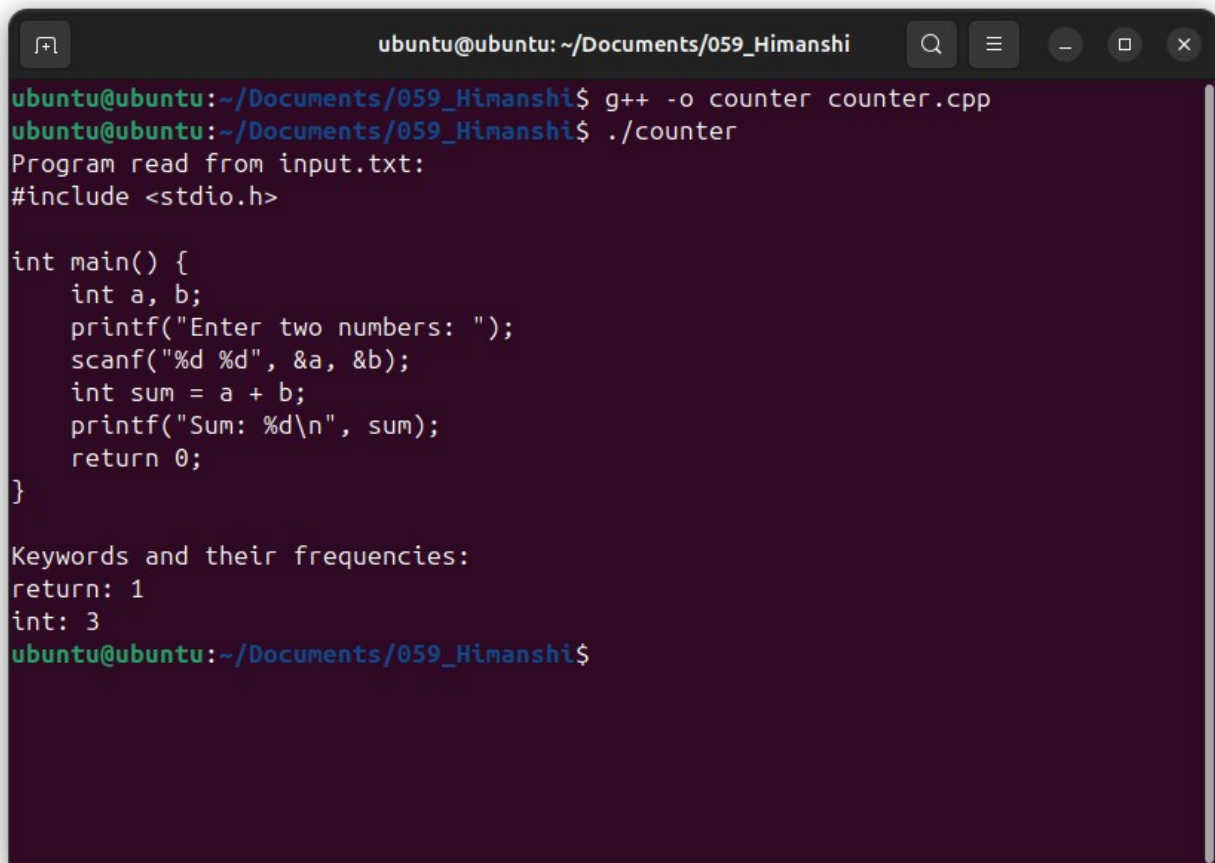
std::cout << "Program read from input.txt:\n";
std::cout << program_code << std::endl;

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

return 0;
}

```

OUTPUT:



A terminal window titled 'ubuntu@ubuntu: ~/Documents/059_Himanshi' showing the compilation and execution of a C++ program. The user runs 'g++ -o counter counter.cpp' and './counter'. The output shows the program reading from 'input.txt', displaying the source code, and then printing 'Keywords and their frequencies:' followed by 'return: 1' and 'int: 3'.

```

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: /
```

```
ubuntu@ubuntu: ~/Documents/059_Himanshi
ubuntu@ubuntu:~/Documents/059_Himanshi$ ./a.out
Enter an expression:
9%4
Modulus operation: %
```

```
ubuntu@ubuntu: ~/Documents/059_Himanshi
ubuntu@ubuntu:~/Documents/059_Himanshi$ ./a.out
Enter an expression:
4+7
Syntax error
ubuntu@ubuntu:~/Documents/059_Himanshi$
```

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;
```

```

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];

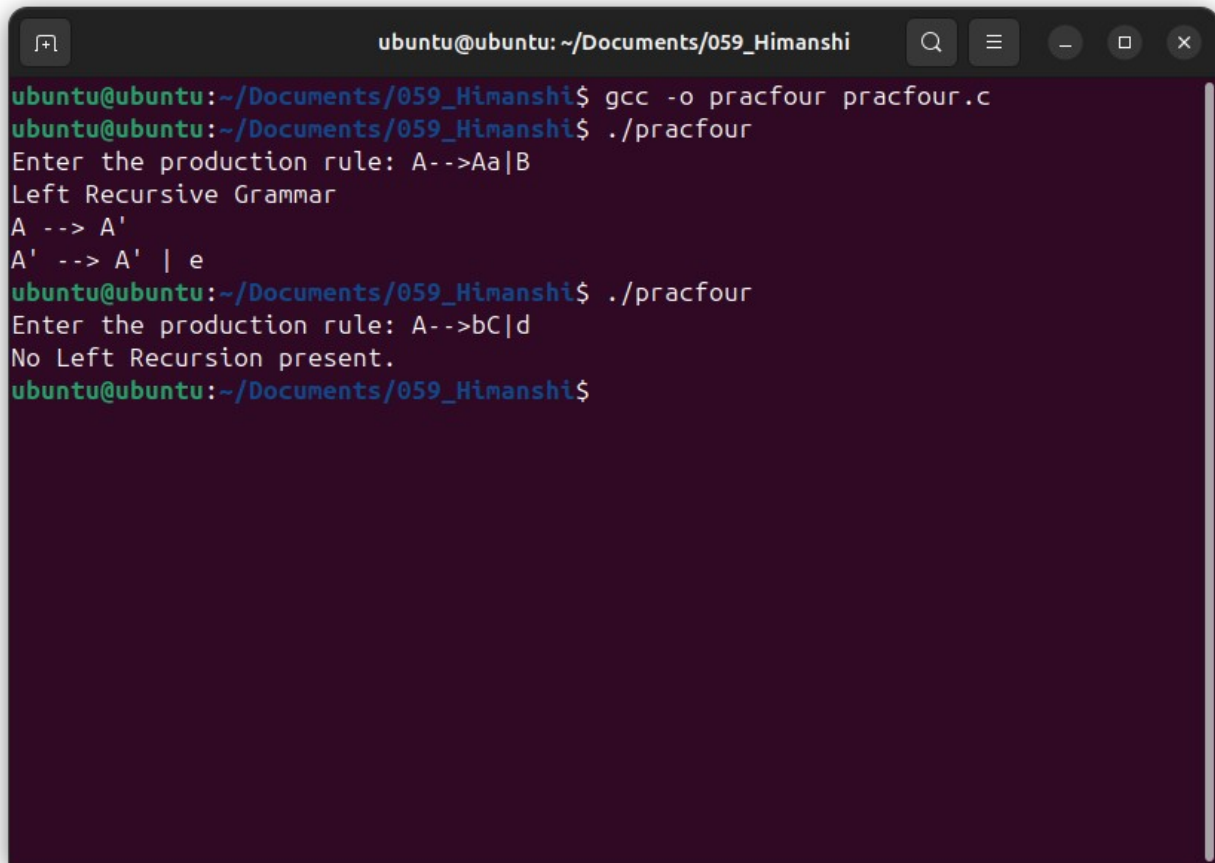
    printf("Enter the production rule: ");
    fgets(input, MAX, stdin);
    input[strcspn(input, "\n")] = 0;

```

```
checkLeftRecursion(input);

return 0;
}
```

OUTPUT:

A terminal window titled 'ubuntu@ubuntu: ~/Documents/059_Himanshi' with search, menu, and window control icons. The terminal shows the compilation and execution of a C program named 'pracfour.c'. The first run uses the production rule 'A-->Aa|B' and outputs 'Left Recursive Grammar'. The second run uses the production rule 'A-->bC|d' and outputs 'No Left Recursion present.'

```
ubuntu@ubuntu:~/Documents/059_Himanshi$ gcc -o pracfour pracfour.c
ubuntu@ubuntu:~/Documents/059_Himanshi$ ./pracfour
Enter the production rule: A-->Aa|B
Left Recursive Grammar
A --> A'
A' --> A' | e
ubuntu@ubuntu:~/Documents/059_Himanshi$ ./pracfour
Enter the production rule: A-->bC|d
No Left Recursion present.
ubuntu@ubuntu:~/Documents/059_Himanshi$
```

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.

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$

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;

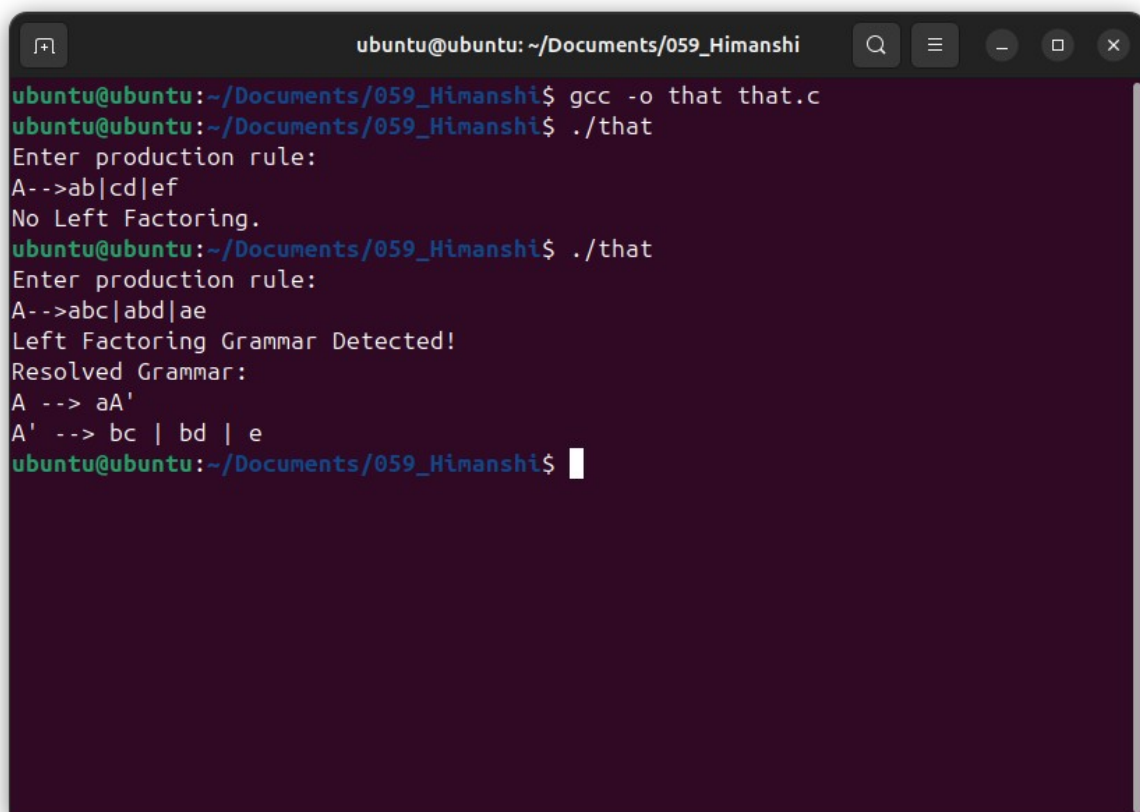
if (sscanf(input, "%c-->%s", &non_terminal, right_side) != 2) {
    printf("Invalid production rule format!\n");
    return 1;
}

checkLeftFactoring(non_terminal, right_side);

return 0;
}

```

OUTPUT:



```

ubuntu@ubuntu: ~/Documents/059_Himanshi
ubuntu@ubuntu:~/Documents/059_Himanshi$ gcc -o that that.c
ubuntu@ubuntu:~/Documents/059_Himanshi$ ./that
Enter production rule:
A-->ab|cd|ef
No Left Factoring.
ubuntu@ubuntu:~/Documents/059_Himanshi$ ./that
Enter production rule:
A-->abc|abd|ae
Left Factoring Grammar Detected!
Resolved Grammar:
A --> aA'
A' --> bc | bd | e
ubuntu@ubuntu:~/Documents/059_Himanshi$

```

LEARNING OUTCOMES:

Date:

Practical 6

AIM: Write a program to find out the FIRST of the Non-terminals in a grammar.

THEORY:

The FIRST set of a non-terminal in a context-free grammar is a set that contains all the terminals that can begin the strings derived from that non-terminal.

The FIRST Set is important.

- The FIRST set helps predict which production rule to use in a given parse state.
- It helps in building predictive parsers (like LL parsers), ensuring they can decide which rule to apply without needing backtracking.

To compute the FIRST set:

1. If there is a production $A \rightarrow a\alpha$ (where a is a terminal), then a is in $\text{FIRST}(A)$.
2. If $A \rightarrow B\alpha$, where B is a non-terminal, then add $\text{FIRST}(B)$ to $\text{FIRST}(A)$ (excluding epsilon, if present).
3. If $A \rightarrow \epsilon$, add ϵ to $\text{FIRST}(A)$.
4. If $A \rightarrow B_1B_2\dots B_n$, continue adding elements from $\text{FIRST}(B)$ until encountering a terminal or non-epsilon entry.

CODE:

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

#define MAX_RULES 100
#define MAX_NON_TERMINALS 26
#define MAX_PRODUCTIONS 50

typedef struct {
    char nonTerminal;
    char productions[MAX_PRODUCTIONS][MAX_PRODUCTIONS]; //
    Array of production rules
    int numProductions;
} GrammarRule;

GrammarRule grammar[MAX_NON_TERMINALS];
```

```

int numGrammarRules = 0;

char firstSets[MAX_NON_TERMINALS][MAX_PRODUCTIONS];
int firstSetSizes[MAX_NON_TERMINALS];

int getNonTerminalIndex(char nt) {
    for (int i = 0; i < numGrammarRules; i++) {
        if (grammar[i].nonTerminal == nt) {
            return i;
        }
    }
    return -1;
}

int isTerminal(char c) {
    return (c >= 'a' && c <= 'z');
}

void computeFirstSet(char nonTerminal) {
    int index = getNonTerminalIndex(nonTerminal);
    if (index == -1) return;

    for (int i = 0; i < grammar[index].numProductions; i++) {
        char *production = grammar[index].productions[i];

        if (isTerminal(production[0])) {
            if (!strchr(firstSets[index], production[0])) {
                firstSets[index][firstSetSizes[index]++] = production[0];
            }
        } else if (production[0] != 'ε') {
            int subIndex = getNonTerminalIndex(production[0]);
            if (subIndex != -1) {
                if (firstSets[subIndex][0] == '\0') {
                    computeFirstSet(production[0]);
                }
            }
        }
    }
}

```

```

        for (int j = 0; j < firstSetSizes[subIndex]; j++) {
            if (!strchr(firstSets[index], firstSets[subIndex][j])) {
                firstSets[index][firstSetSizes[index]++] =
firstSets[subIndex][j];
            }
        }
    }
}
}
}
}

```

```

void inputGrammar() {
    printf("Enter the number of grammar rules: ");
    scanf("%d", &numGrammarRules);

    for (int i = 0; i < numGrammarRules; i++) {
        printf("Enter the non-terminal for rule %d: ", i + 1);
        scanf(" %c", &grammar[i].nonTerminal);

        printf("Enter the productions for %c (separate with '|'): ",
grammar[i].nonTerminal);
        char productionLine[500];
        getchar(); // To capture the newline character
        fgets(productionLine, sizeof(productionLine), stdin);

        char *token = strtok(productionLine, "|");
        int prodCount = 0;

        while (token != NULL) {
            strcpy(grammar[i].productions[prodCount++], token);
            token = strtok(NULL, "|");
        }

        grammar[i].numProductions = prodCount;
    }
}

```

```
    }  
}
```

```
void printFirstSets() {  
    printf("\nFIRST sets:\n");  
    for (int i = 0; i < numGrammarRules; i++) {  
        printf("FIRST(%c) = {", grammar[i].nonTerminal);  
        for (int j = 0; j < firstSetSizes[i]; j++) {  
            if (j > 0) printf(", ");  
            printf("%c", firstSets[i][j]);  
        }  
        printf("}\n");  
    }  
}
```

```
int main() {  
    for (int i = 0; i < MAX_NON_TERMINALS; i++) {  
        firstSetSizes[i] = 0;  
    }  
  
    inputGrammar();  
    for (int i = 0; i < numGrammarRules; i++) {  
        computeFirstSet(grammar[i].nonTerminal);  
    }  
    printFirstSets();  
  
    return 0;  
}
```

OUTPUT:

```
ubuntu@ubuntu: ~/Documents/059_Himanshi
ubuntu@ubuntu:~/Documents/059_Himanshi$ gcc -o prac6 prac6.c
ubuntu@ubuntu:~/Documents/059_Himanshi$ ./prac6
Enter the number of grammar rules: 4
Enter the non-terminal for rule 1: E
Enter the productions for E (separate with '|'): tX
Enter the non-terminal for rule 2: X
Enter the productions for X (separate with '|'): tX|e
Enter the non-terminal for rule 3: T
Enter the productions for T (separate with '|'): xF
Enter the non-terminal for rule 4: F
Enter the productions for F (separate with '|'): E|i

FIRST sets:
FIRST(E) = {t}
FIRST(X) = {t, e}
FIRST(T) = {x}
FIRST(F) = {t, i}
ubuntu@ubuntu:~/Documents/059_Himanshi$
```

LEARNING OUTCOMES:

Date:

Practical 7

AIM: Write a program to Implement Shift Reduce parsing for a String.

THEORY:

Shift-Reduce parsing is a bottom-up parsing method. The parser uses a stack to shift input symbols until it recognizes a pattern that can be reduced to a non-terminal using grammar rules.

Key in Shift-Reduce Parsing

1. Shift: Move (shift) the next input symbol onto the stack.
2. Reduce: When the stack contains a sequence that matches the right-hand side of a production, replace that sequence with the non-terminal on the left-hand side of the production (this is called a reduction).
3. Accept: If the entire input string is parsed, and the stack contains only the start symbol, the parsing is successful.
4. Error: If no valid shift or reduction can be made, a syntax error is detected.

Shift-Reduce Parsing Actions

- SHIFT: Push the next symbol from the input onto the stack.
- REDUCE: Apply a production rule in reverse (i.e., replace a sequence of symbols on the stack that matches a production's right side with the non-terminal on the left side).
- ACCEPT: The input string is fully parsed when the stack contains only the start symbol.
- ERROR: If parsing cannot continue (due to no applicable rules), an error is raised.

CODE:

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

```
#define MAX 100
```

```
typedef struct {
    char lhs;
    char rhs[MAX];
} Production;
```

```
Production productions[MAX];
```

```
int prod_count = 0;
```

```
void add_production(char lhs, const char *rhs);
```

```

void display_productions();
int is_terminal(char ch);
int shift_reduce_parse(char *input);

int main() {
    char input[MAX];
    int result;

    printf("Enter the productions (use '|' for multiple productions, enter 'END' to stop):\n");
    char line[MAX];
    while (1) {
        printf("Production %d: ", prod_count + 1);
        fgets(line, sizeof(line), stdin);

        line[strcspn(line, "\n")] = 0;

        if (strcmp(line, "END") == 0) {
            break;
        }

        char *token = strtok(line, "-->");
        char lhs = token[0];

        token = strtok(NULL, "-->");
        char *rhs = strtok(token, "|");

        while (rhs != NULL) {
            add_production(lhs, rhs);
            rhs = strtok(NULL, "|");
        }
    }

    printf("\nGrammar Productions:\n");
    display_productions();
}

```



```

printf("\nEnter input string to parse: ");
scanf("%s", input);

result = shift_reduce_parse(input);
if (result == 1) {
    printf("\nThe string is accepted.\n");
} else {
    printf("\nThe string is not accepted.\n");
}

return 0;
}

void add_production(char lhs, const char *rhs) {
    productions[prod_count].lhs = lhs;
    strcpy(productions[prod_count].rhs, rhs);
    prod_count++;
}

void display_productions() {
    for (int i = 0; i < prod_count; i++) {
        printf("%c --> %s\n", productions[i].lhs, productions[i].rhs);
    }
}

int is_terminal(char ch) {
    return islower(ch);
}

int shift_reduce_parse(char *input) {
    char stack[MAX] = "";
    char action[MAX];
    int i = 0, j = 0;

    printf("\nParsing Steps:\n");
    printf("Stack\tInput\tAction\n");

```

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

```
while (1) {
```

```
    stack[j++] = input[i++];
```

```
    stack[j] = '\0';
```

```
    printf("%-5s\t%-5s\tShift\n", stack, input + i);
```

```
    int reduced;
```

```
    do {
```

```
        reduced = 0;
```

```
        for (int k = 0; k < prod_count; k++) {
```

```
            int len = strlen(productions[k].rhs);
```

```
            if (j >= len && strcmp(&stack[j - len], productions[k].rhs) == 0) {
```

```
                // Perform the reduction
```

```
                j -= len;
```

```
                stack[j++] = productions[k].lhs;
```

```
                stack[j] = '\0';
```

```
                reduced = 1;
```

```
                printf("%-5s\t%-5s\tReduce by %c --> %s\n", stack, input + i, productions[k].lhs,
```

```
productions[k].rhs);
```

```
            }
```

```
        }
```

```
    } while (reduced);
```

```
    if (j == 1 && stack[0] == productions[0].lhs && input[i] == '\0') {
```

```
        return 1; // Accepted
```

```
    }
```

```
    if (input[i] == '\0' && j != 1) {
```

```
        return 0;
```

```
    }
```

```
}
```

```
}
```

OUTPUT:

```
ubuntu@ubuntu: ~/Documents/059_Himanshi
ubuntu@ubuntu: ~/Documents/059_Himanshi
ubuntu@ubuntu:~/Documents/059_Himanshi$ gcc -o prac7 prac7.c
ubuntu@ubuntu:~/Documents/059_Himanshi$ ./prac7
Enter the productions (use '|' for multiple productions, enter 'END' to stop):
Production 1: S-->aA|b
Production 3: A-->c
Production 4: END

Grammar Productions:
S --> aA
S --> b
A --> c

Enter input string to parse: ac

Parsing Steps:
Stack   Input   Action
-----
a        c      Shift
ac       Shift
aA       Reduce by A --> c
S        Reduce by S --> aA

The string is accepted.
ubuntu@ubuntu:~/Documents/059_Himanshi$
```

```
ubuntu@ubuntu: ~/Documents/059_Himanshi
ubuntu@ubuntu: ~/Documents/059_Himanshi
The string is accepted.
ubuntu@ubuntu:~/Documents/059_Himanshi$ ./prac7
Enter the productions (use '|' for multiple productions, enter 'END' to stop):
Production 1: S-->aA|b
Production 3: A-->c
Production 4: END

Grammar Productions:
S --> aA
S --> b
A --> c

Enter input string to parse: ab

Parsing Steps:
Stack   Input   Action
-----
a        b      Shift
ab       Shift
aS       Reduce by S --> b

The string is not accepted.
ubuntu@ubuntu:~/Documents/059_Himanshi$
```

LEARNING OUTCOMES:

Date:

Practical 8

AIM: Write a program to check whether a grammar is operator precedent.

THEORY:

A grammar is operator-precedence if it does not have:

1. Empty productions (no non-terminal produces an epsilon).
2. Unit productions (no non-terminal produces another non-terminal directly).
3. Ambiguous relations between operators in terms of precedence.

Operator-precedence grammars use operator precedence relations between terminals to define parsing, with a set of rules such as:

- $a < b$ (a has lower precedence than b),
- $a > b$ (a has higher precedence than b),
- $a = b$ (a and b are of equal precedence and can be grouped).

Advantages of Operator Precedence Grammars

- Efficient handling of mathematical expressions.
- Simplifies parsing by clearly defining operator hierarchy.

CODE:

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

#define MAX_PRODUCTIONS 100
#define MAX_PROD_LENGTH 100

int isOperatorPrecedent(char grammar[MAX_PRODUCTIONS]
[MAX_PROD_LENGTH], int grammarCount) {
    for (int i = 0; i < grammarCount; i++) {
        char* arrowPos = strstr(grammar[i], "-->");
        if (arrowPos == NULL) continue;

        int lhsLen = arrowPos - grammar[i];
        char lhs[MAX_PROD_LENGTH];
        strncpy(lhs, grammar[i], lhsLen);
```

```

lhs[lhsLen] = '\0';

char rhs[MAX_PROD_LENGTH];
strcpy(rhs, arrowPos + 3); // Skip "-->"

char* part = strtok(rhs, "|");
while (part != NULL) {
    if (strchr(part, 'e') != NULL) {
        return 0;
    }

    char prevChar = '\0';
    for (int j = 0; part[j] != '\0'; j++) {
        char ch = part[j];
        if (isupper(ch)) {
            if (isupper(prevChar)) {
                return 0;
            }
        }
        prevChar = ch;
    }
    part = strtok(NULL, "|");
}
return 1;
}

int main() {
    char grammar[MAX_PRODUCTIONS][MAX_PROD_LENGTH];
    int grammarCount = 0;

    printf("Enter grammar productions (type 'end' to finish):\n");

    while (1) {
        char line[MAX_PROD_LENGTH];

```

```

fgets(line, sizeof(line), stdin);

line[strcspn(line, "\n")] = '\0';

if (strcmp(line, "end") == 0) {
    break;    }

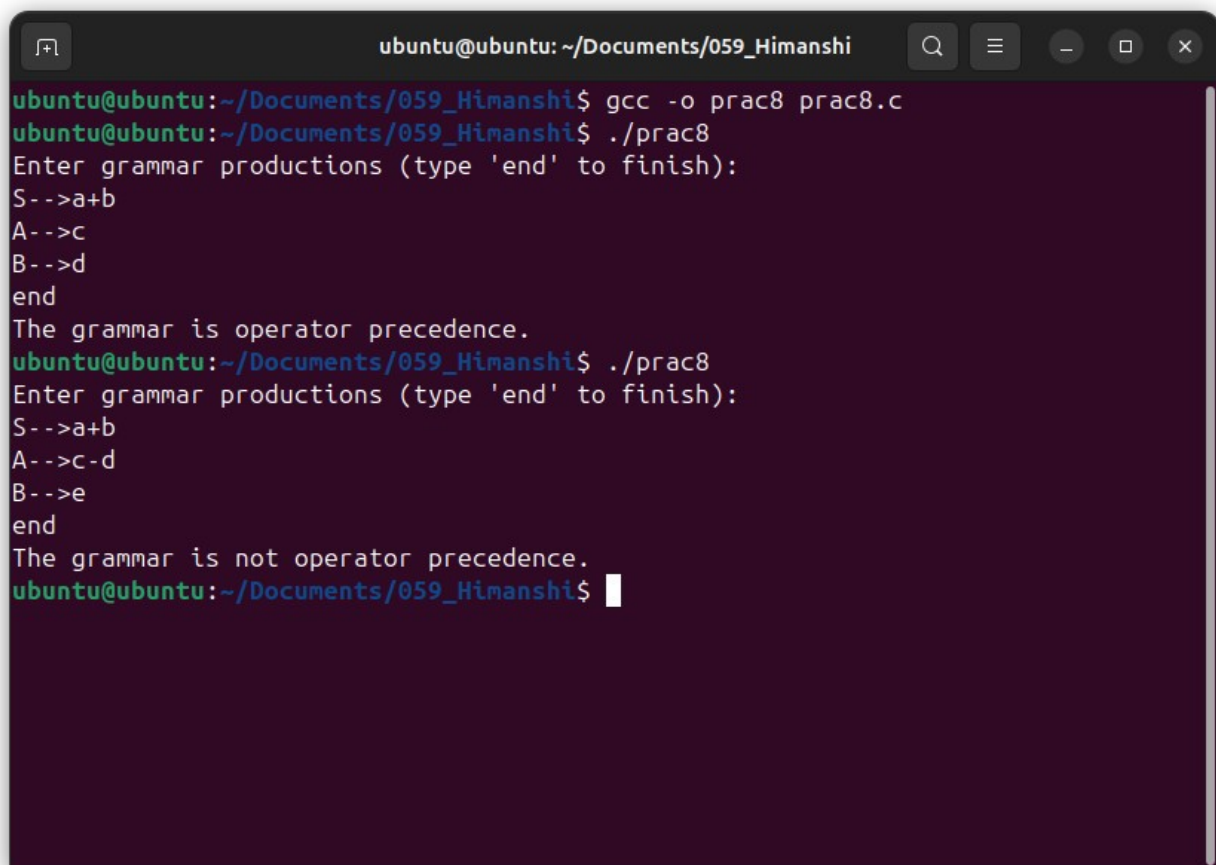
if (strstr(line, "-->") != NULL) {
    strcpy(grammar[grammarCount++], line);
} else {
    printf("Invalid production format. Please follow the rules.\n");
}
}

if (isOperatorPrecedent(grammar, grammarCount)) {
    printf("The grammar is operator precedence.\n");
} else {
    printf("The grammar is not operator precedence.\n");
}

return 0;
}

```

OUTPUT:



```
ubuntu@ubuntu: ~/Documents/059_Himanshi
ubuntu@ubuntu:~/Documents/059_Himanshi$ gcc -o prac8 prac8.c
ubuntu@ubuntu:~/Documents/059_Himanshi$ ./prac8
Enter grammar productions (type 'end' to finish):
S-->a+b
A-->c
B-->d
end
The grammar is operator precedence.
ubuntu@ubuntu:~/Documents/059_Himanshi$ ./prac8
Enter grammar productions (type 'end' to finish):
S-->a+b
A-->c-d
B-->e
end
The grammar is not operator precedence.
ubuntu@ubuntu:~/Documents/059_Himanshi$
```

LEARNING OUTCOMES: