

**VIVEKANANDA INSTITUTE OF PROFESSIONAL STUDIES - TECHNICAL CAMPUS**

**Grade A++ Accredited Institution by NAAC**

NBA Accredited for MCA Programme; Recognized under Section 2(f) by UGC;  
Affiliated to GGSIP University, Delhi; Recognized by Bar Council of India and AICTE  
An ISO 9001:2015 Certified Institution

**SCHOOL OF ENGINEERING & TECHNOLOGY**

## **BTECH Programme: Computer Science Engineering**

**Course Title:** Compiler Design Lab

**Course Code:** CIC-351

**Batch:** 2022-26

**Submitted By:**

**Name:** Saurav Malik

**Enrollment No:** 06017702722



**VIVEKANANDA INSTITUTE OF PROFESSIONAL STUDIES - TECHNICAL CAMPUS**

**Grade A++ Accredited Institution by NAAC**

NBA Accredited for MCA Programme; Recognized under Section 2(f) by UGC;  
Affiliated to GGSIP University, Delhi; Recognized by Bar Council of India and AICTE  
An ISO 9001:2015 Certified Institution

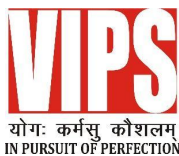
**SCHOOL OF ENGINEERING & TECHNOLOGY**

**VISION OF INSTITUTE**

To be an educational institute that empowers the field of engineering to build a sustainable future by providing quality education with innovative practices that supports people, planet and profit.

**MISSION OF INSTITUTE**

To groom the future engineers by providing value-based education and awakening students' curiosity, nurturing creativity and building capabilities to enable them to make significant contributions to the world.



**VIVEKANANDA INSTITUTE OF PROFESSIONAL STUDIES - TECHNICAL CAMPUS**

**Grade A++ Accredited Institution by NAAC**

NBA Accredited for MCA Programme; Recognized under Section 2(f) by UGC;  
Affiliated to GGSIP University, Delhi; Recognized by Bar Council of India and AICTE  
An ISO 9001:2015 Certified Institution

**SCHOOL OF ENGINEERING & TECHNOLOGY**

## **DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING**

### **VISION OF DEPARTMENT**

To achieve excellence in computer science and fostering research, innovation, and entrepreneurship in students, in order to contribute to the nation's sustainable development.

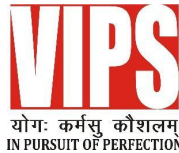
### **MISSION OF DEPARTMENT**

**M1:** To encourage outcome-based learning techniques to develop a center of excellence that satisfies the industry requirements.

**M2:** To develop teamwork and problem-solving abilities, support lifelong learning, and instil a sense of ethical and societal obligations.

**M3:** To impart top-notch experiential learning to gain proficiency in contemporary software tools and to meet the current and futuristic demands.

**M4:** To inculcate knowledge of fundamental concepts and pioneering technologies through research on inter-disciplinary as well as core concepts of computer science.


**VIVEKANANDA INSTITUTE OF PROFESSIONAL STUDIES - TECHNICAL CAMPUS**
**Grade A++ Accredited Institution by NAAC**

NBA Accredited for MCA Programme; Recognized under Section 2(f) by UGC;  
 Affiliated to GGSIP University, Delhi; Recognized by Bar Council of India and AICTE  
 An ISO 9001:2015 Certified Institution

**SCHOOL OF ENGINEERING & TECHNOLOGY**

# INDEX

S. No	EXP.	Allot. Date	Sub. Date	Marks			Remark	Updated Marks	Faculty Signature
				Lab. Assess (15 Marks)	Class Part. (5 Marks)	Viva (5 Marks)			
1	Write a Lexical Analyzer program that identifies any 10 keywords from C language and identifiers following all the naming conventions of the C program.								
2	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								

3	Write a Syntax Analyzer program using Yacc tool that will have grammar rules for the operators : *,/,%.								
4	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.								
5	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.								
6	Write a program to find out the FIRST of the Non-terminals in a grammar.								
7	Write a program to Implement Shift Reduce parsing for a String.								
8	Write a program to check whether a grammar is operator precedent.								

## **PROGRAM-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:**

A Lexical Analyzer, commonly referred to as a scanner or tokenizer, is the first crucial phase of a compiler. Its purpose is to read the raw source code provided as input and break it down into meaningful atomic units known as tokens. These tokens are the smallest identifiable components of a program, representing variables, constants, operators, delimiters, and keywords. In the broader context of compiler design, the lexical analyzer sits between the source code and the syntax analyzer (parser), serving as an intermediary to preprocess the input into a token stream, which the syntax analyzer uses to further validate the structure of the program.

In languages like C, keywords are reserved words predefined by the language specification. They possess special meaning and serve as the building blocks for syntactic structures. Examples include `int`, `if`, `while`, `return`, and `char`. These words cannot be used for any other purpose, such as naming variables or functions. Recognizing keywords is a fundamental task for the lexical analyzer since any misidentification of a keyword can lead to serious semantic or syntactic errors down the compilation pipeline.

On the other hand, identifiers are names provided by the programmer for variables, functions, arrays, structures, etc. Identifiers in C must follow specific naming conventions, such as beginning with a letter (a-z, A-Z) or underscore (`_`) and being followed by letters, digits (0-9), or underscores. Moreover, C is case-sensitive, meaning `variable`, `Variable`, and `VARIABLE` are considered different identifiers. The lexical analyzer must enforce these conventions and distinguish valid identifiers from invalid ones and from keywords.

The process of lexical analysis relies heavily on regular expressions or finite state automata to recognize the different types of tokens. For example, a regular expression may be used to identify any word that matches the pattern of a keyword. Likewise, patterns are set up to distinguish between identifiers and other tokens like numeric constants or operators.

Internally, the lexical analyzer needs to handle several challenges. Whitespace and comments must be ignored, as they are irrelevant to the logic of the program but present in the source code for readability and clarity. Similarly, multi-character operators like `++`, `--`, and `>=` must be recognized as single tokens despite their multi-character nature. In essence, the lexical analyzer dissects the input character stream and identifies its structure while discarding non-relevant elements.

## Sample CODE:

```
%option noyywrap
%%
boolean|float|int|if|char printf("keywords");
[0-9][0-9]* printf("constants");
[a-zA-Z][a-zA-Z0-9]* printf("identifiers");
%%
int main()
{
    yylex();
}
```

## OUTPUT:

```
PROBLEMS OUTPUT PORTS DEBUG CONSOLE TERMINAL
sauravmalik@Sauravs-MacBook-Air 06017702722_Saurav % flex ABC.lec
flex: can't open ABC.lec
sauravmalik@Sauravs-MacBook-Air 06017702722_Saurav % flex ABC.lex
sauravmalik@Sauravs-MacBook-Air 06017702722_Saurav % gcc lex.yy.c
sauravmalik@Sauravs-MacBook-Air 06017702722_Saurav % ./a
zsh: no such file or directory: ./a
sauravmalik@Sauravs-MacBook-Air 06017702722_Saurav % ./a.out
int
keywords
main
identifiers
3242main
constantsidentifiers
```

## CODE:

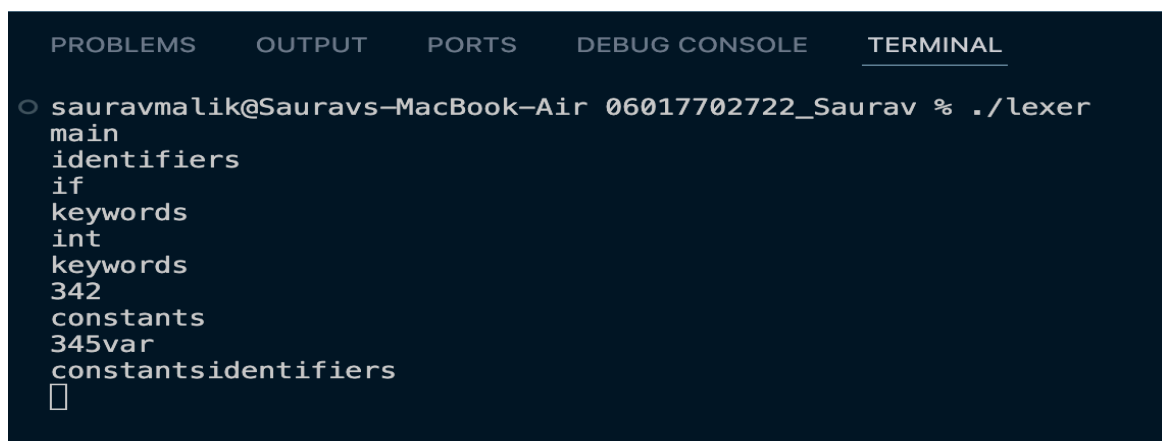
```
#include <stdio.h>
#include <string.h>

#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:



```
PROBLEMS OUTPUT PORTS DEBUG CONSOLE TERMINAL
o sauravmalik@Sauravs-MacBook-Air 06017702722_Saurav % ./lexer
main
identifiers
if
keywords
int
keywords
342
constants
345var
constantsidentifiers
□
```

## LEARNING OUTCOMES:



## **PROGRAM 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:**

In programming languages, keywords form the foundational elements used to define the language's grammar and functionality. A keyword is a predefined and reserved word that carries a specific meaning and serves a dedicated role in constructing a program's syntax. For instance, in C, keywords such as `int`, `while`, `return`, and `if` are essential for defining control structures, data types, and function operations. A keyword, once defined by the language's grammar, cannot be used for any other purpose, such as naming variables or functions.

The goal of this experiment is to create a C program that takes an input string from a text file (e.g., `input.txt`) and identifies the frequency of the keywords appearing in the string. The process involves file handling, where the program opens the input text file and reads its contents into a memory buffer. The string is then processed word by word, checking if any word matches a predefined list of keywords.

Tokenization is a fundamental step in this process. The input string is broken into individual components called tokens, using delimiters such as spaces, punctuation marks, and special characters. Each token is compared to the list of known keywords in C. The keywords are usually stored in a data structure such as an array or hash table for quick access. For every match found, the frequency counter associated with that keyword is incremented. The program may also need to handle case-sensitivity, ensuring that keywords are matched precisely.

This experiment showcases a practical application of text processing and word counting, techniques widely used in areas like natural language processing (NLP) and data mining. File handling and string manipulation are essential programming skills in C, as they allow programs to interact with external data sources, read from them, and process the content.

The output of the program is a report that lists each keyword along with its frequency count in the input string. This type of program can be expanded further to handle more complex tasks, such as syntax highlighting in text editors, keyword analysis in compilers, and frequency analysis in large datasets.

## Input File Text:

```
#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;
}
```

## CODE:

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

// List of 32 C keywords
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";

    // Extract words from the line
    std::istringstream iss(line);
    std::string word;
    while (iss >> word) {
        // Check if the word is a keyword and update its count
        if (is_keyword(word)) {
            keyword_count[word]++;
        }
    }
}

infile.close();

// Output the C program read from the file
std::cout << "Program read from input.txt:\n";
std::cout << program_code << std::endl;

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

return 0;
}
```

## **OUTPUT:**



```
PROBLEMS  OUTPUT  PORTS  DEBUG CONSOLE  TERMINAL

sauravmalik@Sauravs-MacBook-Air 06017702722_Saurav % cd "/Users/sauravmalik/Desktop/06017702722_Saurav/" && g++ -std=c++11 counter.cpp -o counter && ./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
```

**LEARNING OUTCOMES:**

## **PROGRAM 3**

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

### **THEORY:**

A Syntax Analyzer, or parser, is the second phase of the compilation process. It is responsible for verifying whether a given sequence of tokens, generated by the lexical analyzer, conforms to the rules of the programming language's grammar. The grammar defines the syntactic structure of valid statements and expressions, typically described using context-free grammar (CFG). The parser ensures that the token stream can be reduced into a valid structure, such as an expression or statement, according to the grammar rules.

Yacc (Yet Another Compiler Compiler) is a tool used to generate parsers based on a given grammar. It reads a set of grammar rules defined in Backus-Naur Form (BNF) and generates C code to parse inputs according to those rules. The Yacc tool is widely used in the construction of compilers and interpreters. It works in conjunction with a lexical analyzer, often implemented using Lex, to form the front-end of a compiler.

In this experiment, Yacc is used to create grammar rules for the arithmetic operators \* (multiplication), / (division), and % (modulus) in C. These operators are part of the core arithmetic operation set in the C language. They follow specific rules of precedence and associativity. Multiplication, division, and modulus all have the same level of precedence, and their associativity is left-to-right, meaning expressions like  $a * b / c$  are evaluated from left to right.

The grammar rules defined in Yacc must reflect this precedence and associativity. The rules will guide the parser in correctly interpreting expressions involving these operators, ensuring that they are evaluated in the correct order. The parser also generates a parse tree, which represents the syntactic structure of the expression. This tree serves as the basis for further stages in the compilation process, such as optimization and code generation.

The use of Yacc in this context demonstrates how parsers are constructed from grammatical rules and how they contribute to building compilers and interpreters. It also illustrates the importance of handling operator precedence and associativity, which are essential for the correct evaluation of expressions in a language.

## Lex Code :

```
%{
#include <stdio.h>
#include "exp3.tab.h"
}%

%%

[0-9]+ { yylval = atoi(yytext); return NUMBER; }
"+"    { return ADD; }
"-"    { return SUB; }
"*"    { return MUL; }
"/"    { return DIV; }
\n     { return '\n'; } // Return newline to Bison for
processing
[ \t]   { /* ignore whitespace */ }
.       { /* ignore any other character */ }

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

## Yacc Code :

**Code to include \*, /, % operations**

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

void yyerror(const char *s);
int yylex(void);
}%
%token NUMBER
%token ADD SUB MUL DIV
%left ADD SUB
%left MUL DIV
%%
input:
    | input expression '\n' { /* Accept and ignore newline
*/ };
expression:
    | expression ADD expression { printf("result = %d\n", $1 +
$3); }
    | expression SUB expression { printf("result = %d\n", $1 -
$3); }
```

```

    | expression MUL expression { printf("result = %d\n", $1 *
$3); }
    | expression DIV expression {
        if ($3 == 0) {
            yyerror("Attempted to divide by zero.");
            YYERROR;
        } else {
            printf("result = %d\n", $1 / $3);
        }
    }
    | NUMBER { $$ = $1; }
;

%%
void yyerror(const char *s) {
    fprintf(stderr, "%s\n", s);
}
int main(void) {
    printf("Enter expressions to evaluate (Ctrl+C to exit):
\n");
    yyparse();
    return 0;
}

```

## **OUTPUT:**



```

PROBLEMS  OUTPUT  PORTS  DEBUG CONSOLE  TERMINAL

● sauravmalik@Sauravs-MacBook-Air 06017702722_Saurav % bison -dy P3.yacc
● sauravmalik@Sauravs-MacBook-Air 06017702722_Saurav % flex P3.lex
● sauravmalik@Sauravs-MacBook-Air 06017702722_Saurav % gcc -o my_program y.tab.c lex.yy.c
● sauravmalik@Sauravs-MacBook-Air 06017702722_Saurav % ./my_program
Enter expressions to evaluate (Ctrl+C to exit):
15 + 3
result = 18
16 - 8
result = 8
20 / 0
Attempted to divide by zero.
○ sauravmalik@Sauravs-MacBook-Air 06017702722_Saurav %

```

## **LEARNING OUTCOMES:**

## **PROGRAM 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:**

In context-free grammars (CFGs), left recursion refers to a situation where a non-terminal symbol in a grammar rule immediately refers to itself as the first symbol on the right-hand side of the production. This phenomenon poses a significant problem for top-down parsers, particularly recursive descent parsers, which may enter an infinite recursion loop while attempting to process left-recursive grammars.

A production rule with left recursion follows the form:  $A \rightarrow A\alpha \mid \beta$

Where  $A$  is a non-terminal,  $\alpha$  is a sequence of symbols, and  $\beta$  is a non-recursive alternative. The issue arises when the parser attempts to parse a string that matches the recursive part ( $A\alpha$ ), leading it to call itself indefinitely.

To overcome this problem, left recursion must be eliminated from the grammar. This is achieved by refactoring the left-recursive grammar into an equivalent non-left-recursive form. For the example above, the left-recursive production can be rewritten as:

$$\begin{aligned} A &\rightarrow \beta A' \\ A' &\rightarrow \alpha A' \mid \epsilon \end{aligned}$$

Where  $A'$  is a new non-terminal, and  $\epsilon$  represents the empty string. This transformation allows top-down parsers to process the grammar without encountering infinite recursion.

The task of the C program is to take a production rule as input, check whether it exhibits left recursion, and if so, transform the grammar into a non-left-recursive form. The program identifies left-recursive patterns by analyzing the structure of the production rule. Once detected, the program outputs the refactored grammar that avoids recursion.

Left recursion elimination is crucial for making grammars suitable for parsers that rely on top-down parsing techniques. The refactored grammar is easier to parse and is less prone to errors caused by infinite recursion.



## 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");
    }
}

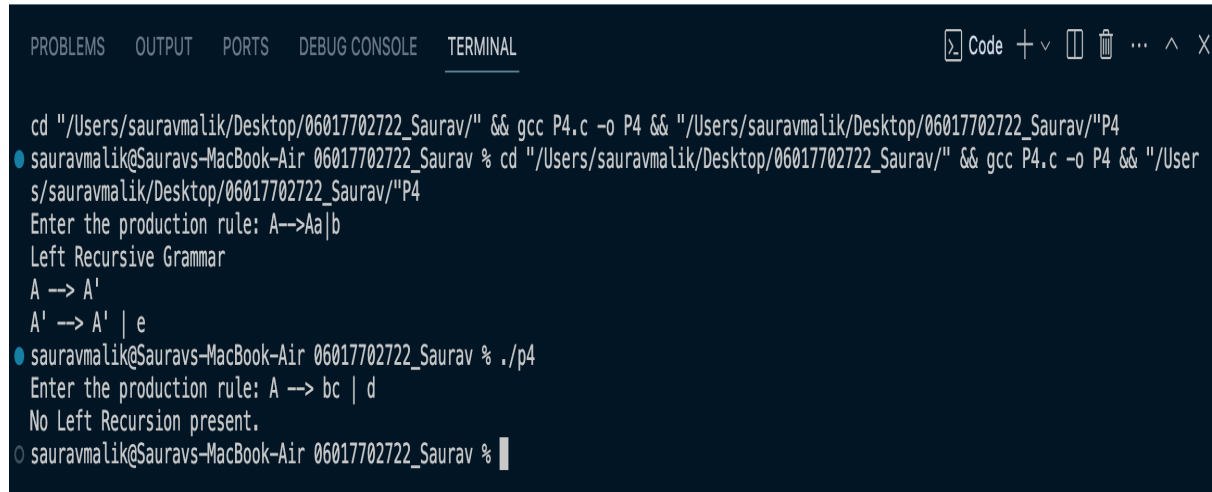
int main() {
    char input[MAX];

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

    checkLeftRecursion(input);
}
```

```
    return 0;  
}
```

## **OUTPUT:**



```
PROBLEMS OUTPUT PORTS DEBUG CONSOLE TERMINAL Code + - [ ] [ ] ... ^ X  
  
cd "/Users/sauravmalik/Desktop/06017702722_Saurav/" && gcc P4.c -o P4 && "/Users/sauravmalik/Desktop/06017702722_Saurav/"P4  
sauravmalik@Sauravs-MacBook-Air 06017702722_Saurav % cd "/Users/sauravmalik/Desktop/06017702722_Saurav/" && gcc P4.c -o P4 && "/User  
s/sauravmalik/Desktop/06017702722_Saurav/"P4  
Enter the production rule: A-->Aa|b  
Left Recursive Grammar  
A --> A'  
A' --> A' | e  
sauravmalik@Sauravs-MacBook-Air 06017702722_Saurav % ./p4  
Enter the production rule: A --> bc | d  
No Left Recursion present.  
sauravmalik@Sauravs-MacBook-Air 06017702722_Saurav %
```

## **LEARNING OUTCOMES:**

## **PROGRAM 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:**

In context-free grammar, left factoring is a technique used to eliminate ambiguity in production rules where multiple alternatives share a common prefix. Left factoring makes a grammar suitable for top-down parsers like LL parsers, which rely on deterministic decision-making based on the current input token.

Consider the following production rule:  $A \rightarrow \alpha\beta \mid \alpha\gamma$

Both alternatives share the common prefix  $\alpha$ , which makes it difficult for a top-down parser to decide which path to take. This leads to ambiguity, as the parser cannot immediately determine whether to follow the  $\beta$  or  $\gamma$  branch without looking ahead in the input.

Left factoring transforms this ambiguous grammar into a non-ambiguous form by factoring out the common prefix. The factored form looks like this:

$$\begin{aligned} A &\rightarrow \alpha A' \\ A' &\rightarrow \beta \mid \gamma \end{aligned}$$

By doing this, the parser can first match the common prefix  $\alpha$  and then decide between  $\beta$  and  $\gamma$ , based on the remaining input. This process eliminates the need for lookahead and makes grammar easier to parse using top-down techniques.

In this experiment, the C program takes a production rule as input and checks whether it can be left-factored. If left-factoring is possible, the program outputs the factored grammar. The program achieves this by analyzing the structure of the production rule and identifying common prefixes between different alternatives.

Left factoring is essential for making grammar more efficient and parser-friendly. It ensures that grammars are deterministic, allowing parsers to make decisions without backtracking or excessive lookahead. This improves the performance and reliability of the parsing process.

**CODE:**

```

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

#define MAX 100

int commonPrefixLength(char *str1, char *str2) {
    int i = 0;
    while (str1[i] && str2[i] && str1[i] == str2[i]) {
        i++;
    }
    return i;
}

void checkLeftFactoring(char *input) {
    char nonTerminal, alpha[MAX], betas[MAX][MAX];
    char *token;
    int i = 0, prefixLength = 0;
    int minPrefixLength = MAX;
    char prefix[MAX];

    nonTerminal = input[0];

    token = strtok(input + 3, "|");
    while (token != NULL) {
        strcpy(betas[i++], token);
        token = strtok(NULL, "|");
    }

    for (int k = 1; k < i; k++) {
        prefixLength = commonPrefixLength(betas[0], betas[k]);
        if (prefixLength < minPrefixLength) {
            minPrefixLength = prefixLength;
        }
    }

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

    strncpy(prefix, betas[0], minPrefixLength);
    prefix[minPrefixLength] = '\0';

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

    printf("%c --> %s%c'", nonTerminal, prefix, nonTerminal);

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

```

```

        if (strncmp(betas[k], prefix, minPrefixLength) != 0) {
            printf("%s", betas[k]);
        }
    }
    printf("\n");

    printf("%c' --> ", nonTerminal);
    for (int k = 0; k < i; k++) {
        if (k > 0) printf("|");
        if (strlen(betas[k]) > minPrefixLength) {
            printf("%s", betas[k] + minPrefixLength);
        } else {
            printf("e"); // epsilon
        }
    }
    printf("\n");
}

int main() {
    char input[MAX];
    printf("Enter a production rule: ");
    fgets(input, MAX, stdin);

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

    checkLeftFactoring(input);

    return 0;
}

```

## OUTPUT:

```

PROBLEMS OUTPUT PORTS DEBUG CONSOLE TERMINAL
sauravmalik@Sauravs-MacBook-Air 06017702722_Saurav % cd "/Users/sauravmalik/Desktop/06017702722_Saurav/" && gcc P5.c -o P5 && "/Users/sauravmalik/Desktop/06017702722_Saurav/"P5
Enter a production rule: A -> aB|cd
No Left Factoring
sauravmalik@Sauravs-MacBook-Air 06017702722_Saurav % ./P5
Enter a production rule: A -> Aa | cd
No Left Factoring
sauravmalik@Sauravs-MacBook-Air 06017702722_Saurav %

```

## LEARNING OUTCOMES:

# Experiment 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 grammar is the set of terminal symbols that can appear at the beginning of any string derived from that non-terminal. If the non-terminal can derive an empty string (epsilon), then  $\epsilon$  is also included in its FIRST set.

### 1. Notation

- **FIRST(X)**: The FIRST set of non-terminal XXX.
- $\epsilon$ : Represents the empty string.

### 2. Calculation of FIRST Sets

To calculate the FIRST set for a non-terminal, follow these rules:

### 3. For Terminal Symbols

- If X is a terminal, then:  $\text{FIRST}(X) = \{X\}$

#### 3.1 For Non-Terminal Symbols

- If there is a production of the form:  $X \rightarrow Y_1 Y_2 \dots Y_n$ .
  - If  $Y_1$  is a terminal, add it to  $\text{FIRST}(X)$ .
  - If  $Y_1$  is a non-terminal:
    - Add all symbols in  $\text{FIRST}(Y_1)$  (excluding  $\epsilon$ ) to  $\text{FIRST}(X)$ .
    - If  $Y_1$  can derive  $\epsilon$ , continue to  $Y_2$ , and so on, until a terminal is found or all symbols are processed.
  - If all symbols  $Y_1, Y_2, \dots, Y_n$  can derive  $\epsilon$ , include  $\epsilon$  in  $\text{FIRST}(X)$ .

#### 3.2 Epsilon Productions

- If a non-terminal XXX has a production that can derive  $\epsilon$  (e.g.,  $X \rightarrow \epsilon$ ), then:  $\epsilon \in \text{FIRST}(X)$ .

## Code:

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

#define MAX_PRODUCTIONS 100
#define MAX_SYMBOLS 100

typedef struct
{
    char nonTerminal;
    char productions[MAX_PRODUCTIONS][MAX_SYMBOLS];
    int prodCount;
} Production;

Production productions[MAX_PRODUCTIONS];
int productionCount = 0;

typedef struct
```

```

{
    char nonTerminal;
    char firstSet[MAX_SYMBOLS];
    int firstCount;
} FirstSet;

FirstSet firstSets[MAX_PRODUCTIONS];
int firstSetCount = 0;

int findFirstSet(char nonTerminal)
{
    for (int i = 0; i < firstSetCount; i++)
    {
        if (firstSets[i].nonTerminal == nonTerminal)
        {
            return i;
        }
    }
    return -1;
}

void addToFirstSet(int index, char symbol)
{
    for (int i = 0; i < firstSets[index].firstCount; i++)
    {
        if (firstSets[index].firstSet[i] == symbol)
            return;
    }
    firstSets[index].firstSet[firstSets[index].firstCount++] = symbol;
}

void computeFirst(char nonTerminal)
{
    int index = findFirstSet(nonTerminal);
    if (index == -1)
    {
        index = firstSetCount++;
        firstSets[index].nonTerminal = nonTerminal;
        firstSets[index].firstCount = 0;
    }

    for (int i = 0; i < productionCount; i++)
    {
        if (productions[i].nonTerminal == nonTerminal)
        {
            for (int j = 0; j < productions[i].prodCount; j++)
            {
                char *production = productions[i].productions[j];
                for (int k = 0; production[k] != '\0'; k++)
                {
                    char symbol = production[k];

                    if (islower(symbol) || symbol == 'e')
                    {
                        addToFirstSet(index, symbol);
                        break;
                    }
                    else
                    {
                        computeFirst(symbol);
                        int symbolIndex = findFirstSet(symbol);
                        for (int m = 0; m <
firstSets[symbolIndex].firstCount; m++)

```

```

        {
            if (firstSets[symbolIndex].firstSet[m] != 'e')
            {
                addToFirstSet(index,
firstSets[symbolIndex].firstSet[m]);
            }
        }
        if (strchr(firstSets[symbolIndex].firstSet, 'e')
== NULL)
            break;
    }
}
}
}
}

int main()
{
    int n;
    printf("Enter the number of productions: ");
    scanf("%d", &n);
    getchar(); // Consume newline character

    printf("Enter the productions (e.g., S-->aA|b):\n");
    for (int i = 0; i < n; i++)
    {
        char line[MAX_SYMBOLS];
        fgets(line, sizeof(line), stdin);
        line[strcspn(line, "\n")] = 0; // Remove newline

        productions[productionCount].nonTerminal = line[0];
        char *token = strtok(line + 4, "|"); // Skip the arrow and split
by '|'

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

    for (int i = 0; i < productionCount; i++)
    {
        computeFirst(productions[i].nonTerminal);
    }

    printf("FIRST sets:\n");
    for (int i = 0; i < firstSetCount; i++)
    {
        printf("FIRST(%c) = { ", firstSets[i].nonTerminal);
        for (int j = 0; j < firstSets[i].firstCount; j++)
        {
            printf("%s ", firstSets[i].firstSet[j] == 'e' ? "epsilon" :
(char[]){firstSets[i].firstSet[j], '\0'});
        }
        printf("}\n");
    }
    return 0;
}

```



## Output:

```
cd "/Users/sauravmalik/Desktop/06017702722_Saurav/" && gcc P6.c -o P6 && "/Users/sauravmalik/Desktop/06017702722_Saurav/"P6
sauravmalik@Sauravs-MacBook-Air 06017702722_Saurav % cd "/Users/sauravmalik/Desktop/06017702722_Saurav/" && gcc P6.c -o P6 && "/Users
/sauravmalik/Desktop/06017702722_Saurav/"P6
Enter the number of productions: 3
Enter the productions (e.g., S-->aA|b):
S-->AB|e
A-->aA|B
B-->b
FIRST sets:
FIRST(S) = { a b epsilon }
FIRST(A) = { a b }
FIRST(B) = { b }
```

## Learning Outcome:

# Experiment 7

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

## Theory:

Shift-reduce parsing is a fundamental technique used in syntax analysis, particularly in the context of bottom-up parsers. This method utilizes a stack to hold symbols and employs two primary operations—shift and reduce—to derive a parse tree for a given input string. Below is a detailed explanation of shift-reduce parsing, its mechanisms, and an example.

## 1. Definition of Shift-Reduce Parsing

### 1.1 What is Shift-Reduce Parsing?

Shift-reduce parsing is a bottom-up parsing technique where the parser begins with the input string and works backward towards the start symbol of the grammar. It makes use of a stack to manage the symbols currently being processed. The two primary operations in this parsing technique are:

- **Shift:** Move the next input symbol onto the stack.
- **Reduce:** Replace a sequence of symbols on the stack that matches the right-hand side of a production rule with the corresponding non-terminal.

## 2. Operations in Shift-Reduce Parsing

### 2.1 Shift Operation

- The **shift** operation involves taking the next symbol from the input string and pushing it onto the stack. This operation continues until a valid reduction can occur.

### 2.2 Reduce Operation

- The **reduce** operation occurs when the top symbols of the stack match the right-hand side of a production rule. The parser pops these symbols off the stack and pushes the corresponding non-terminal onto the stack.

## Code:

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

#define MAX_PRODUCTIONS 100
#define MAX_SYMBOLS 100

typedef struct
{
    char input[MAX_SYMBOLS];
    char nonTerminal;
} Production;

Production productions[MAX_PRODUCTIONS];
int productionCount = 0;

typedef struct
{
    char stack[MAX_SYMBOLS];
    int top;
} Stack;
```

```

void initStack(Stack *s)
{
    s->top = -1;
}

void push(Stack *s, const char *symbol)
{
    if (s->top < MAX_SYMBOLS - 1)
    {
        strcpy(&s->stack[++(s->top)], symbol);
    }
}

void pop(Stack *s, int count)
{
    if (s->top >= count - 1)
    {
        s->top -= count;
    }
}

const char *top(Stack *s)
{
    return &s->stack[s->top];
}

void shift(Stack *p, const char *input, int *j, char
steps[MAX_PRODUCTIONS][MAX_SYMBOLS], int *stepCount)
{
    char symbol[2] = {input[*j], '\0'};
    push(p, symbol);
    (*j)++;
    snprintf(steps[(stepCount)++], MAX_SYMBOLS, "Shift: %s | Stack: %s",
&input[*j], symbol);
}

int reduce(Stack *p, char steps[MAX_PRODUCTIONS][MAX_SYMBOLS], int
*stepCount)
{
    for (int i = 0; i < productionCount; i++)
    {
        int len = strlen(productions[i].input);
        char topSequence[MAX_SYMBOLS] = "";

        for (int j = len - 1; j >= 0; j--)
        {
            if (p->top < 0)
                return 0;
            strncat(topSequence, top(p), 1);
            pop(p, 1);
        }

        if (strcmp(topSequence, productions[i].input) == 0)
        {
            char symbol[2] = {productions[i].nonTerminal, '\0'};
            push(p, symbol);
            snprintf(steps[(stepCount)++], MAX_SYMBOLS, "Reduce: %s ->
%c", topSequence, productions[i].nonTerminal);
            return 1;
        }
        else
        {
            for (int k = len - 1; k >= 0; k--)

```

```

        {
            char symbol[2] = {topSequence[k], '\0'};
            push(p, symbol);
        }
    }
}
return 0;
}

int srParser(const char *input, char steps[MAX_PRODUCTIONS][MAX_SYMBOLS],
int *stepCount)
{
    Stack p;
    initStack(&p);
    int j = 0;

    while (j < strlen(input) || p.top > 0)
    {
        if (j < strlen(input))
        {
            shift(&p, input, &j, steps, stepCount);
        }
        else if (!reduce(&p, steps, stepCount))
        {
            return 0;
        }
    }
    return p.top == 0 && p.stack[0] == 'S';
}

int main()
{
    int n;
    printf("Enter the number of productions: ");
    scanf("%d", &n);
    getchar(); // Consume newline

    printf("Enter the productions (e.g., S-->aA|b):\n");
    for (int i = 0; i < n; i++)
    {
        char line[MAX_SYMBOLS];
        fgets(line, sizeof(line), stdin);
        line[strlen(line)] = 0; // Remove newline

        productions[productionCount].nonTerminal = line[0];
        char *token = strtok(line + 4, "|"); // Skip the arrow

        while (token)
        {
            strcpy(productions[productionCount++].input, token);
            token = strtok(NULL, "|");
        }
    }

    char input[MAX_SYMBOLS];
    printf("Enter the input string: ");
    scanf("%s", input);

    char steps[MAX_PRODUCTIONS][MAX_SYMBOLS];
    int stepCount = 0;
    int accepted = srParser(input, steps, &stepCount);

    for (int i = 0; i < stepCount; i++)

```

```

    {
        printf("%s\n", steps[i]);
    }

    printf("%s\n", accepted ? "Accepted" : "Not Accepted");
    return 0;
}

```

## Output:

```

cd "/Users/sauravmalik/Desktop/06017702722_Saurav/" && gcc P7.c -o P7 && "/Users/sauravmalik/Desktop/06017702722_Saurav/"P7
sauravmalik@Sauravs-MacBook-Air 06017702722_Saurav % cd "/Users/sauravmalik/Desktop/06017702722_Saurav/" && gcc P7.c
-o P7 && "/Users/sauravmalik/Desktop/06017702722_Saurav/"P7
Enter the number of productions:
2
Enter the productions (e.g., S-->aA|b):
S-->aA|b
A-->c|d
Enter the input string: ac
Shift: c | Stack: a
Shift: | Stack: c
Reduce: c -> A
Accepted
sauravmalik@Sauravs-MacBook-Air 06017702722_Saurav % cd "/Users/sauravmalik/Desktop/06017702722_Saurav/" && gcc P7.c
-o P7 && "/Users/sauravmalik/Desktop/06017702722_Saurav/"P7
Enter the number of productions: 2
Enter the productions (e.g., S-->aA|b):
S-->aA|b
A-->c|d
Enter the input string: ag
Shift: g | Stack: a
Shift: | Stack: g
Not Accepted

```

## Learning Outcome:

# Experiment 8

**Aim:** Write a program to check whether a grammar is operator precedent

## Theory:

Operator precedence grammar is a type of context-free grammar that is designed to specify the precedence (order of evaluation) and associativity (direction of evaluation) of operators in expressions. This allows parsers to correctly interpret and evaluate expressions according to the rules of arithmetic or logical operations.

### 1. Key Concepts

- **Precedence:** Refers to the rules that determine which operator takes priority when evaluating an expression. Higher precedence operators are evaluated before lower precedence ones.
- **Associativity:** Determines the order in which operators of the same precedence level are processed. For example, left-to-right associativity means that operations are performed from left to right.

### 2. Structure of Operator Precedence Grammar

#### 2.1 Productions

Operator precedence grammars typically include productions that define the syntax of expressions, operators, and their relationships. The productions often include:

- **Non-terminals:** Represent different levels of expressions or operations (e.g., E for expressions, T for terms).
- **Terminals:** Represent actual operators and operands (e.g., +, -, \*, a, b).

#### 2.2 Precedence Levels

In this grammar, the precedence is defined as follows:

- **Highest Precedence:** Multiplication (\*) and division (/) have higher precedence than addition (+) and subtraction (-).
- **Lowest Precedence:** Addition and subtraction have lower precedence than multiplication and division.
- **Associativity:**
  - Addition and subtraction are left associative.
  - Multiplication and division are also left associative.

### 3. Rules to check whether operator precedent:

- a. no two non-terminals should appear together.
- b. no epsilon present in the right side of any of the productions

## Code:

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

#define MAX_PRODUCTIONS 100
#define MAX_SYMBOLS 100

// Structure to hold grammar productions
typedef struct
{
    char lhs[MAX_SYMBOLS];
    char rhs[MAX_SYMBOLS];
} Production;

Production grammar[MAX_PRODUCTIONS];
```

```

int productionCount = 0;

// Function to check if the grammar is operator precedence
int isOperatorPrecedent()
{
    for (int i = 0; i < productionCount; i++)
    {
        char *rhs = grammar[i].rhs;
        int len = strlen(rhs);

        // Check for epsilon (represented as 'e' here)
        if (strchr(rhs, 'e') != NULL)
        {
            return 0; // Epsilon is not allowed in operator precedence
        }

        // Check for consecutive non-terminals
        char prevChar = '\0';
        for (int j = 0; j < len; j++)
        {
            char ch = rhs[j];
            if (isupper(ch))
            { // Non-terminal
                if (isupper(prevChar))
                {
                    // Previous was also a non-terminal
                    return 0; // Invalid: two non-terminals together
                }
            }
            prevChar = ch; // Update previous character
        }
    }
    return 1; // Passed all checks
}

int main()
{
    printf("Enter grammar productions (type 'end' to finish):\n");
    char line[MAX_SYMBOLS];

    while (1)
    {
        fgets(line, sizeof(line), stdin);
        line[strlen(line)] = '\0'; // Remove newline character

        if (strcmp(line, "end") == 0)
        {
            break; // Stop taking input when 'end' is entered
        }

        // Ensure the input follows the correct format
        char *arrowPos = strstr(line, "-->");
        if (arrowPos == NULL)
        {
            printf("Invalid production format. Please follow the rules.\n");
            continue;
        }

        // Extract lhs and rhs of the production
        strncpy(grammar[productionCount].lhs, line, arrowPos - line);
        grammar[productionCount].lhs[arrowPos - line] = '\0'; // Null-terminate lhs
    }
}

```

```
        strcpy(grammar[productionCount].rhs, arrowPos + 3);    // Copy rhs
        productionCount++;
    }

    // Check if the grammar is operator precedence
    if (isOperatorPrecedent())
    {
        printf("The grammar is operator precedent.\n");
    }
    else
    {
        printf("The grammar is not operator precedent.\n");
    }

    return 0;
}
```

## Output:

```
cd "/Users/sauravmalik/Desktop/06017702722_Saurav/" && gcc P8.c -o P8 && "/Users/sauravmalik/Desktop/06017702722_Saurav/"P8
sauravmalik@Sauravs-MacBook-Air 06017702722_Saurav % cd "/Users/sauravmalik/Desktop/06017702722_Saurav/" && gcc P8.c -o P8 && "/Users
/sauravmalik/Desktop/06017702722_Saurav/"P8
Enter grammar productions (type 'end' to finish):
S-->a+b
A-->c
B-->d
end
The grammar is operator precedent.
sauravmalik@Sauravs-MacBook-Air 06017702722_Saurav % ./P8
Enter grammar productions (type 'end' to finish):
S-->a+b
A-->c-d
B-->e
end
The grammar is not operator precedent.
```

## Learning Outcome: