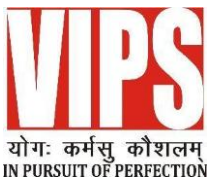


VIVEKANANDA INSTITUTE OF PROFESSIONAL STUDIES - TECHNICAL CAMPUS
Grade A++ Accredited Institution by NAAC
NBA Accredited for MCA Programme; Recognized under Section 2(f) by UGC;
Affiliated to GGSIP University, Delhi; Recognized by Bar Council of India and AICTE
An ISO 9001:2015 Certified Institution

SCHOOL OF ENGINEERING & TECHNOLOGY

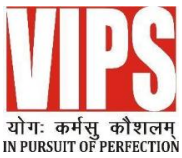
BTECH Programme: CSE (B)

Course Title: Compiler Design Lab

Course Code: CIC-351

Submitted By:

Name: Rudra Sharma
Enrollment No: 10417702722



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.

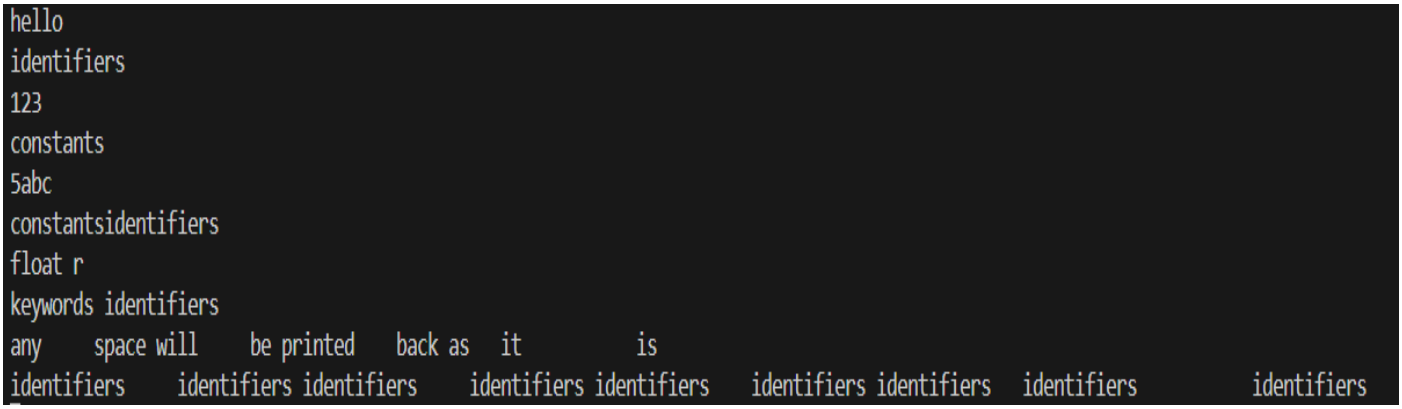
A) Demo/Boilerplate code

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:



B) Code to implement C language rules for identifiers and keywords

CODE:

```
%option noyywrap

%%
"auto"|"break"|"case"|"char"|"const"|"continue"|"default"|"do"|"double"|"else"|
"enum"|"extern"|"float"|"for"|"goto"|"if"|"inline"|"int"|"long"|"register"|"res
trict"|"return"|"short"|"signed"|"sizeof"|"static"|"struct"|"switch"|"typedef"|
"union"|"unsigned"|"void"|"volatile"|"while" printf("keyword: %s\n", yytext);

[0-9]+ printf("constant: %s\n", yytext);

[a-zA-Z_][a-zA-Z0-9_]* printf("identifier: %s\n", yytext);
%%

int main()
{
    yylex();
    return 0;
}
```

OUTPUT:

```
auto
keyword: auto

0
constant: 0

0526
constant: 0526

_hello
identifier: _hello

Abcd_
identifier: Abcd_

9aBc
constant: 9
identifier: aBc

if
keyword: if

else
keyword: else

break
keyword: break
```

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<iostream>
using namespace std;

int main() {
    int a = 5, b = 10;
    int sum = a + b;
    cout << "Sum: " << sum << endl;
    return 0;
}
```

CODE:

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

using namespace std;

// List of 62 C++ keywords
vector<string> cpp_keywords = {
    "alignas", "alignof", "and", "and_eq", "asm", "auto", "bitand", "bitor",
"bool", "break", "case", "catch",
    "char", "char16_t", "char32_t", "class", "compl", "const", "constexpr",
"const_cast", "continue", "decltype",
    "default", "delete", "do", "double", "dynamic_cast", "else", "enum",
"explicit", "export", "extern", "false",
    "float", "for", "friend", "goto", "if", "inline", "int", "long", "mutable",
"namespace", "new", "noexcept",
    "not", "not_eq", "nullptr", "operator", "or", "or_eq", "private",
"protected", "public", "register", "reinterpret_cast",
    "return", "short", "signed", "sizeof", "static", "static_assert",
"static_cast", "struct", "switch", "template",
    "this", "throw", "true", "try", "typedef", "typeid", "typename", "union",
"unsigned", "using", "virtual", "void",
    "volatile", "wchar_t", "while", "xor", "xor_eq"};

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

```
int main()
{

    string filePath = "C:\Users\varun\OneDrive\Desktop\coding> cd
"c:\Users\varun\OneDrive\Desktop\coding\c\" ; if ($?) { gcc os.c -o os } ; if
($?) { .\os }";

    // Open input file
    ifstream file(filePath);
    if (!file)
    {
        cerr << "Error: Could not open input file." << endl;
        return 1;
    }

    // Read the entire file into a string
    stringstream buffer;
    buffer << file.rdbuf();
    string programText = buffer.str();

    // Output the string read from input.txt
    cout << "Program text from input.txt:" << endl;
    cout << programText << endl;

    // Prepare to count keywords
    unordered_map<string, int> keyword_count;
    stringstream ss(programText);
    string word;

    // Parse the program and count keyword frequency
    while (ss >> word)
    {
        // Remove punctuation from word
        word.erase(remove_if(word.begin(), word.end(), ::ispunct), word.end());

        if (is_keyword(word))
        {
            keyword_count[word]++;
        }
    }

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

    return 0;
}
```

OUTPUT:

```
PS C:\Users\rishab\Downloads\CD LAB,07817702722_Rishab_Negi> cd "c:\Users\rishab\Downloads\CD LAB,07817702722_Rishab_Negi\exp2\" ; if ($?) { g++ exp2.cpp -o exp2 } ; if ($?) { .\exp2 }
Program text from input.txt:
#include<iostream>
using namespace std;

int main() {
    int a = 5, b = 10;
    int sum = a + b;
    cout << "Sum: " << sum << endl;
    return 0;
}


Keywords found and their frequencies:
int : 3
namespace : 1
return : 1
using : 1
PS C:\Users\rishab\Downloads\CD LAB,07817702722_Rishab_Negi\exp2>
```

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 (Common):

```
%{
#include <stdio.h>
#include "sample.tab.h"
int c;
extern int yylval;
%}
%%
" "       ;
[a-z]     {
            c = yytext[0];
            yylval = c - 'a';
            return(LETTER);
          }
[0-9]     {
            c = yytext[0];
            yylval = c - '0';
            return(DIGIT);
          }
[^a-z0-9\b]    {
                 c = yytext[0];
                 return(c);
```

A) Sample code

CODE:

```
%{
#include <stdio.h>
int regs[26];
int base;
%}
%start list
%token DIGIT LETTER
%left '|'
%left '&'
%left '+' '-'
%left '*' '/' '%'
%%                    /* beginning of rules section */
list:
          /* empty */
        | list stat '\n'
        | list error '\n' { printf("errorrrrrr\n"); yyerrok; }
        ;
stat:
```

```
          expr { printf("%d\n", $1); }
        | LETTER '=' expr { printf("here\n"); regs[$1] = $3; }
        ;
expr:
          '(' expr ')' { $$ = $2; }
        | expr '+' expr { $$ = $1 + $3; }
        | expr '-' expr { $$ = $1 - $3; }
        | LETTER { $$ = regs[$1]; }
        | number
        ;
number:
          DIGIT {
                 //printf("first here!!");
                 $$ = $1; base = ($1 == 0) ? 8 : 10; }
        | number DIGIT { printf("%d %d \n", $1, $2); $$ = base * $1 + $2; }
        ;
%%
main() {
    return (yyparse());
}
yyerror(s)
char *s;
{
    fprintf(stderr, "%s\n", s);
}
yywrap() {
    return (1);
}
```

OUTPUT:

B) Code to include * , / , % operations

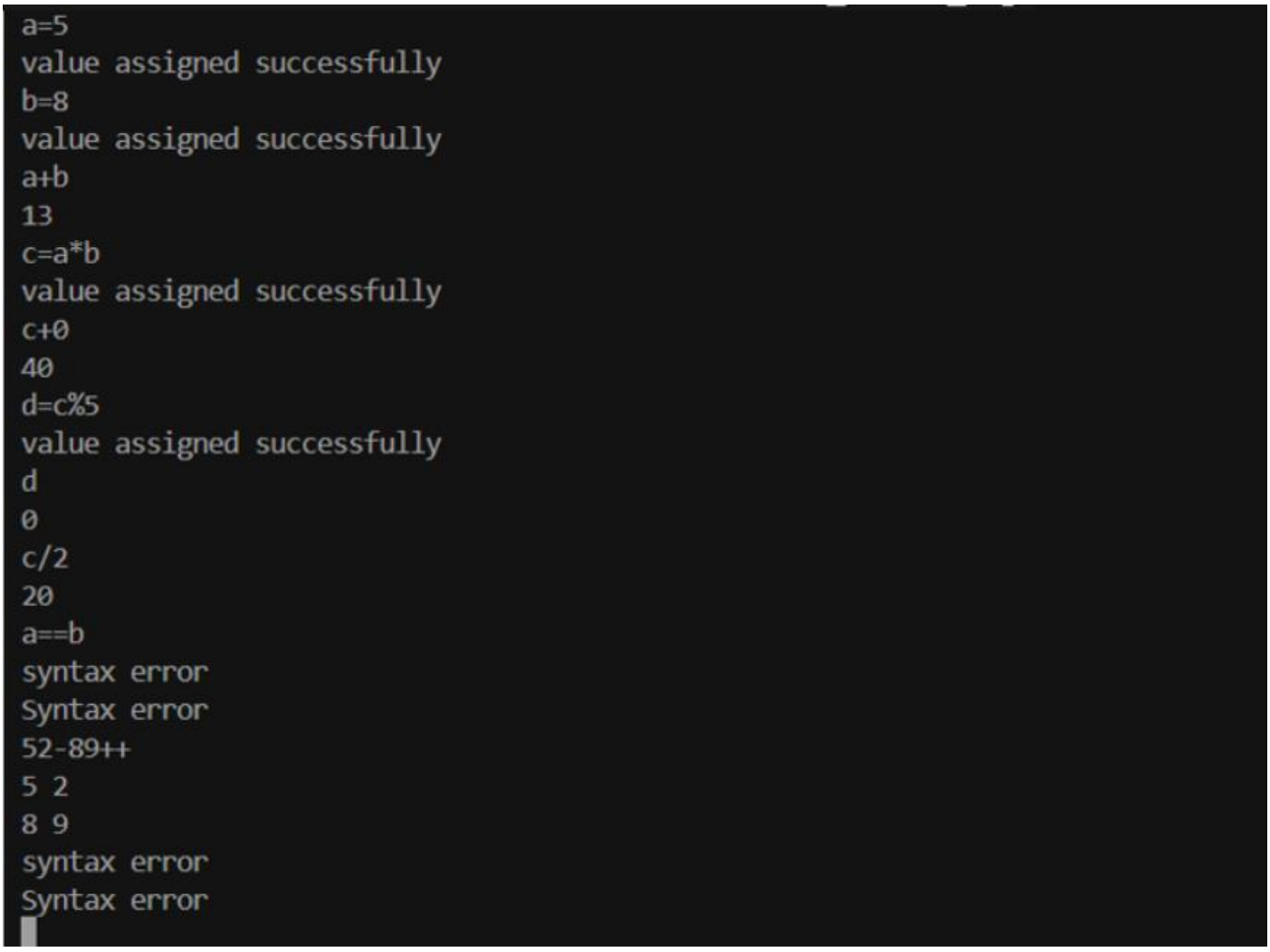
CODE:

```
%{
#include <stdio.h>
int regs[26];
int base;
%}
%start list
%token DIGIT LETTER
%left '|'
%left '&'
%left '+' '-'
%left '*' '/' '%'   /* operator precedence is low to high from top to bottom*/

%%                  /* beginning of rules section */
list:
           /*empty */
         | list stat '\n'
        | list error '\n' { printf("Syntax error\n"); yyerrok; }
        ;
stat:
        expr { printf("%d\n", $1); }
        | LETTER '=' expr { printf("value assigned successfully\n"); regs[$1] =
$3; }
        ;
expr:
        '(' expr ')' { $$ = $2; }
        | expr '+' expr { $$ = $1 + $3; }
        | expr '-' expr { $$ = $1 - $3; }
        | expr '*' expr { $$ = $1 * $3; }
        | expr '/' expr { $$ = $1 / $3; }
        | expr '%' expr { $$ = $1 % $3; }
        | LETTER { $$ = regs[$1]; }
        | number
        ;
number:  DIGIT
         {
           //printf("first here!!");
           $$ = $1; base = ($1 == 0) ? 8 : 10; }
        | number DIGIT { printf("%d %d \n", $1, $2); $$ = base * $1 + $2; }
        ;
%%
main() {
    return (yyparse());
}
yyerror(s)
char *s;
{
```

```
    fprintf(stderr, "%s\n", s);
}
yywrap() {
    return (1);
}
```

OUTPUT:



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, α is a sequence of symbols, and β is a non-recursive alternative. The issue arises when the parser attempts to parse a string that matches the recursive part (Aα), 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:

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' \mid \varepsilon$$

Where A' is a new non-terminal, and ε 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 <ctype.h>

#define MAX 100

void removeLeftRecursion(char nonTerminal, char alpha[][MAX], char beta[][MAX],
int alphaCount, int betaCount)
{
    printf("Left Recursive Grammar detected!\n");

    // New Non-Terminal for recursive rules, append `'` to the non-terminal
    char newNonTerminal[MAX];
    sprintf(newNonTerminal, "%c'", nonTerminal);

    // Print the modified production rule without left recursion
    printf("The modified grammar is:\n");
    printf("%c -->", nonTerminal);
    for (int i = 0; i < betaCount; i++)
    {
        printf(" %s%s", beta[i], newNonTerminal);
        if (i < betaCount - 1)
        {
            printf(" |");
        }
    }
    printf("\n");

    printf("%s -->", newNonTerminal);
    for (int i = 0; i < alphaCount; i++)
    {
        printf(" %s%s", alpha[i], newNonTerminal);
        if (i < alphaCount - 1)
        {
            printf(" |");
        }
    }
    printf(" | e\n"); // Adding epsilon (denoted as 'e')
}

int main()
{
    char input[MAX], nonTerminal, production[MAX];
    char alpha[MAX][MAX], beta[MAX][MAX]; // Stores alpha (left-recursive) and
beta (non-left-recursive) parts
    int alphaCount = 0, betaCount = 0;

    // Take the production rule as input
```

```
    printf("Enter the production rule (e.g., A-->Aa|b): ");
    fgets(input, MAX, stdin);
    input[strcspn(input, "\n")] = '\0'; // Remove trailing newline character

    // Extract the non-terminal and the production part
    sscanf(input, "%c-->%s", &nonTerminal, production);

    // Validate the input
    if (!isupper(nonTerminal))
    {
        printf("Invalid non-terminal. It should be an uppercase letter.\n");
        return 1;
    }

    // Split the production into tokens separated by '|'
    char *token = strtok(production, "|");

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

    // Check if left recursion exists
    if (alphaCount == 0)
    {
        printf("No Left Recursion present.\n");
    }
    else
    {
        removeLeftRecursion(nonTerminal, alpha, beta, alphaCount, betaCount);
    }

    return 0;
}
```

OUTPUT:

```
PS C:\Users\varun\OneDrive\Desktop\coding> cd "c:\Users\varun\OneDrive\Desktop\coding\c\" ; if ($?) { gcc os.c -o os } ; if ($?) { .\os }
Enter the production rule (e.g., A-->Aa|b): A-->a|b
No Left Recursion present.
PS C:\Users\varun\OneDrive\Desktop\coding\c>
```

```
PS C:\Users\varun\OneDrive\Desktop\coding\c> cd "c:\Users\varun\OneDrive\Desktop\coding\c\" ; if ($?) { gcc os.c -o os } ; if ($?) { .\os }
Enter the production rule (e.g., A-->Aa|b): E-->E*E|E+E|-E|id
Left Recursive Grammar detected!
The modified grammar is:
E --> -EE' | idE'
E' --> *EE' | +EE' | e
PS C:\Users\varun\OneDrive\Desktop\coding\c>
```

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 α, 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 β or γ 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:

$$A \rightarrow \alpha A'$$
$$A' \rightarrow \beta \mid \gamma$$

By doing this, the parser can first match the common prefix α and then decide between β and γ, 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

// Function to check for left factoring
void leftFactoring(char *input)
{
    char nonTerminal;
    char productions[MAX][MAX], commonPrefix[MAX], newProduction[MAX][MAX];
    int i, j, k, len, commonLen = 0, prodCount = 0;

    // Split the input into non-terminal and productions
    sscanf(input, "%c --> %[^\n]", &nonTerminal, input);

    // Splitting the production rules by '|'
    char *token = strtok(input, "|");
    while (token != NULL)
    {
        strcpy(productions[prodCount++], token);
        token = strtok(NULL, "|");
    }

    // Find the common prefix
    len = strlen(productions[0]);
    for (i = 1; i < prodCount; i++)
    {
        for (j = 0; j < len && j < strlen(productions[i]); j++)
        {
            if (productions[0][j] != productions[i][j])
                break;
        }
        len = j; // Update length to the smallest common prefix
    }

    // If len is 0, no left factoring
    if (len == 0)
    {
        printf("No Left Factoring present\n");
        return;
    }

    // Copy the common prefix
    strncpy(commonPrefix, productions[0], len);
    commonPrefix[len] = '\0';

    // Generate the new grammar
```

```
    printf("Left Factoring Grammar:\n");
    printf("%c --> %s%c'\n", nonTerminal, commonPrefix, nonTerminal);
    printf("%c' --> ", nonTerminal);

    for (i = 0; i < prodCount; i++)
    {
        if (strlen(productions[i]) == len)
        {
            printf("e"); // if the production is just the common prefix, append
epsilon
        }
        else
        {
            printf("%s", productions[i] + len); // print the remaining part of
the production
        }
        if (i < prodCount - 1)
        {
            printf(" | ");
        }
    }
    printf("\n");
}

int main()
{
    char input[MAX];

    // Example input
    printf("Enter production rule (A --> aB|aC|aD): ");
    fgets(input, MAX, stdin);
    input[strlen(input) - 1] = '\0'; // Remove trailing newline

    leftFactoring(input);

    return 0;
}
```

OUTPUT:

```
PS C:\Users\varun\OneDrive\Desktop\coding> cd "c:\Users\varun\OneDrive\Desktop\coding\c\" ; if ($?) { gcc os.c -o os } ; if ($?) { .\os }
Enter production rule (A --> aB|aC|aD): A-->aB|bc
No Left Factoring present
PS C:\Users\varun\OneDrive\Desktop\coding\c>
```

```
PS C:\Users\varun\OneDrive\Desktop\coding\c> cd "c:\Users\varun\OneDrive\Desktop\coding\c\" ; if ($?) { gcc os.c -o os } ; if ($?) { .\os }
Enter production rule (A --> aB|aC|aD): A-->aB|aC|aD
Left Factoring Grammar:
A --> aA'
A' --> B | C | D
PS C:\Users\varun\OneDrive\Desktop\coding\c>
```

LEARNING OUTCOMES:

Program 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 ε is also included in its FIRST set.

1.Notation

- **FIRST(X)**: The FIRST set of non-terminal XXX.
- **ε**: 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: FIRST(X)={X}

3.1 For Non-Terminal Symbols

- If there is a production of the form: X→Y1Y2…Yn. ○ If Y1 is a terminal, add it to FIRST(X).
  - ○ If Y1 is a non-terminal:
    - ✦ Add all symbols in FIRST(Y1) (excluding ε) to FIRST(X).
    - ✦ If Y1 can derive ε, continue to Y2, and so on, until a terminal is found or all symbols are processed.
  - ○ If all symbols Y1,Y2,…,Yn can derive ε, include ε in FIRST(X).

3.2 Epsilon Productions

- If a non-terminal XXX has a production that can derive ε (e.g., X→e), then: ε∈ FIRST(X)

CODE:

```
#include <iostream>
#include <map>
#include <set>
#include <vector>
#include <sstream>
using namespace std;
map<char, set<char>> firstSets; map<char, vector<string>> productions;
void computeFirst(char nonTerminal) {
    if (firstSets.count(nonTerminal)) return;
    set<char> firstSet;
    for (const string &production : productions[nonTerminal])
    {
        for (char symbol : production)
        {
            if (islower(symbol) || symbol == 'e'){
                firstSet.insert(symbol);
                break;
            }
            else{
                computeFirst(symbol);
                set<char> symbolFirstSet = firstSets[symbol];
                firstSet.insert(symbolFirstSet.begin(), symbolFirstSet.end());
                if (!symbolFirstSet.count('e'))
                break;
            }
        }
    }
    firstSets[nonTerminal] = firstSet;
}
int main()
{
    int n;
    cout << "Enter the number of productions: ";
    cin >> n;
    cin.ignore();
    cout << "Enter the productions (e.g., S-->aA|b):" << endl;
    while (n--)
    {
        string line, token;
        getline(cin, line);
        char nonTerminal = line[0];
        stringstream ss(line.substr(4)); // Adjusted to correctly skip the arrow
        while (getline(ss, token, '|')) productions[nonTerminal].push_back(token);
```

```
    }
    for (const auto &entry : productions)
        computeFirst(entry.first);
    cout << "FIRST sets:" << endl;
    for (const auto &entry : firstSets)
    {
        cout << "FIRST(" << entry.first << ") = { ";
        for (char symbol : entry.second)
            cout << (symbol == 'e' ? "epsilon" : string(1, symbol)) << " ";
            cout << "}" << endl;
    }
    return 0;
}
```

OUTPUT:

```
PS C:\Users\varun\OneDrive\Desktop\coding> cd "c:\Users\varun\OneDrive\Desktop\coding\c++\" ; if ($?) { g++ firstprogram.c++ -o firstprogram } ; if ($?) { .\firstprog
ram }
Enter the number of productions: 2
Enter the productions (e.g., S-->aA|b):
S-->aA|e
B-->p|q
FIRST sets:
FIRST(B) = { p q }
FIRST(S) = { a epsilon }
PS C:\Users\varun\OneDrive\Desktop\coding\c++>
```

LEARNING OUTCOMES:

Program 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 <iostream>
#include <map>
#include <stack>
#include <sstream>
#include <vector>
using namespace std;
map<string, char> productions;
void shift(stack<string> &p, const string &i, int &j, vector<string> &steps)
{
   p.push(string(1, i[j++]));
   steps.push_back("Shift: " + i.substr(j) + " | Stack: " + p.top());
}
bool reduce(stack<string> &p, vector<string> &steps)
{
   for (const auto &prod : productions)
   {
      string top;
      for (size_t i = 0; i < prod.first.size(); ++i)
      {
         if (p.empty())
            return false;
         top = p.top() + top;
         p.pop();
      }

      if (top == prod.first)
      {
         p.push(string(1, prod.second));
         steps.push_back("Reduce: " + top + " -> " + prod.second);
         return true;
      }
      else
      {
         for (char c : top)
         {
            p.push(string(1, c));
         }
      }
   }
   return false;
}
bool srParser(const string &input, vector<string> &steps)
{
```

```
    stack<string> p;
    int j = 0;
    while (j < input.size() || p.size() > 1)
    {
        if (j < input.size())
        {
            shift(p, input, j, steps);
        }
        else if (!reduce(p, steps))
        {
            return false;
        }
    }
    return p.size() == 1 && p.top() == "S";
}
int main()
{
    int n;
    cout << "Enter the number of productions: ";
    cin >> n;
    cin.ignore();

    cout << "Enter the productions (e.g., S-->aA|b):" << endl;
    while (n--)
    {
        string line, token;
        getline(cin, line);
        char nonTerminal = line[0];
        stringstream ss(line.substr(4)); // Adjusted to correctly skip the arrow
        while (getline(ss, token, '|')) productions[token] = nonTerminal;
    }

    string input;
    cout << "Enter the input string: ";
    cin >> input;
    vector<string> steps;
    bool accepted = srParser(input, steps);

    // Output the steps
    for (const string &step : steps) {
    cout << step << endl;
}

// Output the result
cout << (accepted ? "Accepted" : "Not Accepted") << endl;
return 0;
}
```

OUTPUT:

```
PS C:\Users\varun\OneDrive\Desktop\coding> cd "c:\Users\varun\OneDrive\Desktop\coding\c++\" ; if ($?) { g++ firstprogram.c++ -o firstprogram } ; if ($?) { .\firstprog
ram }
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
Reduce: aA -> S
Accepted
PS C:\Users\varun\OneDrive\Desktop\coding\c++> cd "c:\Users\varun\OneDrive\Desktop\coding\c++\" ; if ($?) { g++ firstprogram.c++ -o firstprogram } ; if ($?) { .\first
program }
Enter the number of productions: 2
Enter the productions (e.g., S-->aA|b):
S-->aA|b
A-->c|d
Enter the input string: av
Shift: v | Stack: a
Shift:  | Stack: v
Not Accepted
PS C:\Users\varun\OneDrive\Desktop\coding\c++>
```

LEARNING OUTCOMES:

Program 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 <iostream>
#include <vector>
#include <sstream>

using namespace std;

// Function to check if the grammar is operator precedence bool
isOperatorPrecedent(const vector<string>& grammar) {
for (const auto &production : grammar)
{
   // Split the production into left-hand side (lhs) and right-hand side (rhs)
   size_t arrowPos = production.find("-->");
   string lhs = production.substr(0, arrowPos);
   string rhs = production.substr(arrowPos + 3); // Skip the arrow

   // Split rhs by '|'
   stringstream ss(rhs);
   string part;

   while (getline(ss, part, '|'))
   {   // Check for epsilon
      if (part.find('e') != string::npos) {
      return false; // Epsilon is not allowed
   }

   // Check for consecutive non-terminals
   char prevChar = '\0';
   for (char ch : part)
   {
      if (isupper(ch))
      { // Non-terminal
         if (isupper(prevChar))
         { // Previous was also a non-terminal
         return false; // Invalid: two non-terminals together
         }
      }
      prevChar = ch; // Update previous character
   }
}
}
return true; // Passed all checks
}
int main()
```

```
{
    vector<string> grammar;
    string line;

    cout << "Enter grammar productions (type 'end' to finish):" << endl;

    while (true)
    {
        getline(cin, line);
        if (line == "end")
        {
            break; // Stop taking input when 'end' is entered
        }
        // Ensure the input follows the rules
        if (line.find("-->") != string::npos) {
        grammar.push_back(line); // Valid production
    }
    else
    {
        cout << "Invalid production format. Please follow the rules." << endl;
    }
}
// Check if the grammar is operator precedence
if (isOperatorPrecedent(grammar)) {
cout << "The grammar is operator precedent." << endl;
}
else
{
    cout << "The grammar is not operator precedent." << endl;
}

return 0;
}
```

OUTPUT:

```
PS C:\Users\varun\OneDrive\Desktop\coding> cd "c:\Users\varun\OneDrive\Desktop\coding\c++\" ; if ($?) { g++ firstprogram.c++ -o firstprogram } ; if ($?) { .\firstprog
ram }
Enter grammar productions (type 'end' to finish):
S-->a+b
A-->c
B-->d
end
The grammar is operator precedent.
PS C:\Users\varun\OneDrive\Desktop\coding\c++> cd "c:\Users\varun\OneDrive\Desktop\coding\c++\" ; if ($?) { g++ firstprogram.c++ -o firstprogram } ; if ($?) { .\first
program }
Enter grammar productions (type 'end' to finish):
s-->a+b
A-->c-d
B-->e
end
The grammar is not operator precedent.
PS C:\Users\varun\OneDrive\Desktop\coding\c++>
```

LEARNING OUTCOMES: