**A close-up of a sign

Description automatically generated**

**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 (A)**

**Course Title:**Compiler Design Lab

**Course Code:** CIC-351

**A close-up of a sign

Description automatically generated**

**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.

**A close-up of a sign

Description automatically generated**

**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.

**A close-up of a sign

Description automatically generated**

**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. | Date | Marks | | | Remark | Updated Marks | Faculty Signature |
| Laboratory Assessment (15 Marks) | Class Participation (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. |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |

# Experiment 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**, also known as a **lexer** or **scanner**, is a program that processes the input source code and converts it into a sequence of tokens. These tokens represent syntactic constructs such as keywords, identifiers, constants, and operators.

In C, a lexical analyzer performs:

**Tokenization**: It breaks down the source code into tokens.

* + **Keywords**: Reserved words in C that have predefined meanings (e.g., int, if, else).
  + **Identifiers**: Names used for variables, functions, or arrays. These must follow naming conventions.
* Begin with a letter (A-Z or a-z) or an underscore (\_).
* The subsequent characters can be letters, digits (0-9), or underscores.
* Identifiers are case-sensitive.
* They must not match any C keywords.
* Eg: main, variable1, foo\_bar, \_temp
  + **Operators**: Symbols like +, -, \*, and =.
  + **Literals**: Constant values like numbers or strings.
  + **Punctuation**: Characters like ;, ,, and {} that define structure.

**Lexical Analysis Steps:**

1. Read the input source code character by character.
2. Detect valid keywords and identifiers by comparing the input against known keywords and applying identifier rules.
3. Return the detected tokens for further processing (e.g., by a parser).

**Structure of a Lex Program:**

A Lex program has three sections, separated by %%:

1. **Declarations Section**:
   * Here, you define any global variables, header files (e.g., #include), and regular expressions for tokens.
2. **Rules Section**:
   * This section contains regular expression rules for token patterns and the actions to perform when a match is found.
   * Each line consists of a regular expression followed by C code, which gets executed when the pattern is matched.
3. **User Code Section**:
   * Any additional C code, such as the main() function, is defined here. You can write custom functions to handle specific tasks.

**How Lex Works with Yacc:**

* Lex is often used alongside **Yacc** (Yet Another Compiler Compiler), which is a parser generator. Lex identifies tokens, and Yacc performs the **syntax analysis** on these tokens.
* **Workflow**:
  1. Lex analyzes the input to produce tokens.
  2. Yacc processes these tokens to build a syntax tree or check for grammar rules.

**Advantages of Lex:**

1. **Automation**: Lex generates efficient C code for lexical analysis, saving the programmer from writing a manual scanner.
2. **Pattern Matching**: It uses regular expressions, which are more concise and readable than traditional manual scanning code.
3. **Integration**: Works well with Yacc for building complete language processors (e.g., compilers, interpreters).

**Code:**

%option noyywrap

%%

break|return|void|struct|for|boolean|float|int|if|char printf("keywords");

[0-9][0-9]\* printf("constants");

[a-zA-Z][a-zA-Z0-9]\* printf("identifiers");

%%

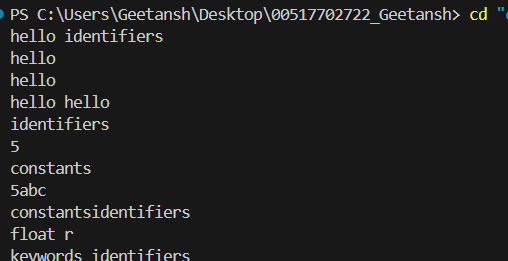
int main()

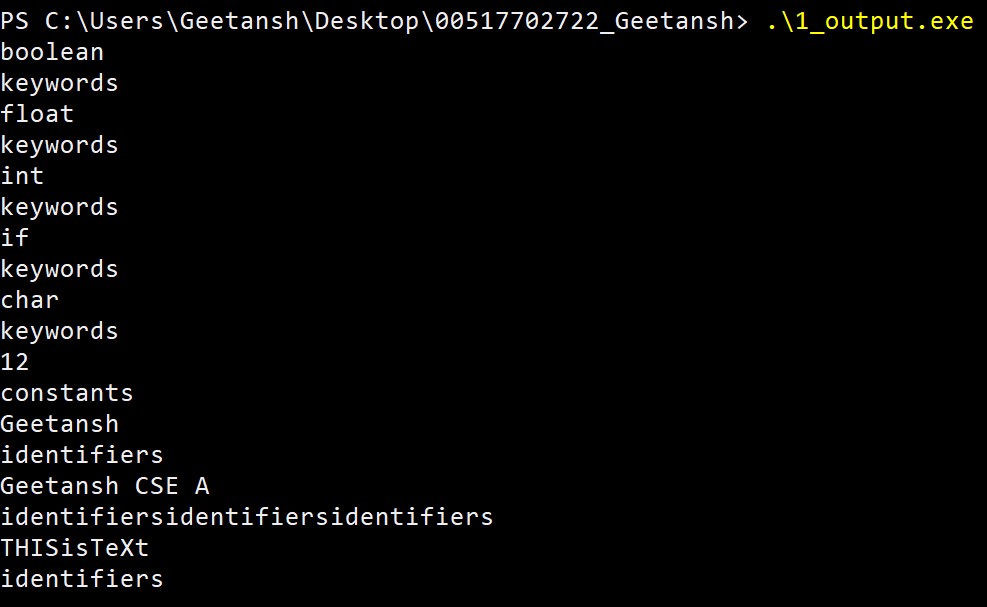
{

yylex();

}

**Output:**

****



**Learning Outcomes:**

# Experiment 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. Keywords in C:**

* **Keywords** are reserved words in the C language that have special meanings and cannot be used as identifiers (e.g., int, return, for, etc.).
* The keywords in C are: 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

**2. Reading Input from a File:**

* To analyze the text, the program must read a string from a file. This can be done using functions like fopen() to open the file, fgets() or fscanf() to read from it, and fclose() to close the file.

**3. Tokenization:**

* To identify words in the input string, the program will use a **tokenizer**. A common way to tokenize strings is by using strtok() which breaks the input string into tokens (words) based on a set of delimiters (e.g., spaces, punctuation, etc.).

**4. Keyword Matching:**

* Once the input is tokenized, the program will compare each word with a list of predefined C keywords to determine if the word is a keyword.

**5. Counting Frequency:**

* The program will maintain a count of how many times each keyword appears in the input. This can be done using an array that tracks the frequency of each keyword.

**Code:**

#include <bits/stdc++.h>

using namespace std;

int main() {

    // Path to input file

    const string inputFilePath = "input.txt";

    // Load keywords

    unordered\_set<string> 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"};

    // Create a map to store keyword counts

    unordered\_map<string, int> keywordCount;

    // Read the input file

    ifstream inputFile(inputFilePath),ss(inputFilePath);

    if (!inputFile) {

        cerr << "Error opening file: " << inputFilePath << endl;

        return 1;

    }

     string token;

        while (inputFile >> token) {

            if (keywords.find(token) != keywords.end()) {

                keywordCount[token]++;

            }

        }

        // output the string in file

        cout << "Content of " << inputFilePath<<":   ";

        while (ss >> token) {

            cout<<token;

        }

        cout<<endl;

        for (const auto& pair : keywordCount) {

            if (pair.second > 0) {

                cout << pair.first << ": " << pair.second << endl;

            }

        }

        inputFile.close();

    return 0;

}

**input.txt:**

**“** #include <stdio.h>

int main() {

int a, b, sum;

printf ("Enter two numbers: ");

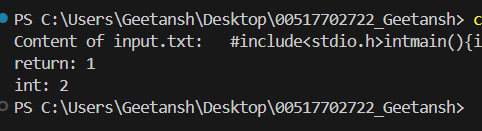
scanf ("%d %d", &a, &b);

sum = a + b;

printf ("Sum: %d\n", sum);

return 0;

} **”**

**Output:  
**

**Learning Outcomes:**

# Experiment 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 a component of a compiler that takes the output of the lexical analyzer (i.e., a stream of tokens) and builds a syntax tree based on the grammar rules of the programming language. For this task, we are going to create a syntax analyzer using **Yacc (Yet Another Compiler Compiler)**, which will analyze expressions involving the operators \*, /, and %.

**Key Concepts:**

1. **Yacc**:
   * **Yacc** is a parser generator that works with **LALR(1)** (Look-Ahead LR) parsing. It takes a set of **grammar rules** and generates a C program that can parse tokens from the lexical analyzer.
   * Yacc grammar consists of **non-terminals**, **terminals (tokens)**, and **production rules** that define how valid expressions are formed in the language.
   * The Yacc-generated parser interacts with a **Lex scanner** (for lexical analysis) to receive tokens.
2. **Grammar Rules**:
   * The grammar defines how operators \*, /, and % are used in expressions. These operators are typically associated with **multiplicative precedence** in programming languages, meaning they are evaluated before addition and subtraction.
   * We define grammar rules in **Backus-Naur Form (BNF)** to specify valid expressions, with each rule representing part of the syntax structure.
3. **Lex**:
   * **Lex** is used to generate the lexical analyzer (scanner). It tokenizes the input and recognizes keywords, operators, and identifiers.
   * Lex interacts with Yacc by passing **tokens**. For example, \*, /, and % would be passed as tokens from Lex to Yacc.
4. **Action and Semantic Values**:
   * The rules in Yacc are associated with **actions** that are executed when a rule is recognized. These actions generally involve operations on the semantic values of tokens, such as evaluating expressions or storing values.
   * **Semantic values** (in Yacc) allow us to perform computations during parsing (e.g., evaluating the result of 3 \* 4).

**Code:**

**calc.yacc**

%{

#include <stdio.h>

#include <stdlib.h>

int regs[26]; // Array to hold variable values

int base;

void yyerror(const char \*s);

%}

%start list

%token DIGIT LETTER

%left '|'

%left '&'

%left '+' '-'

%left '\*' '/' '%'

%% // Beginning of rules section

list:

/\* empty \*/

| list stat '\n'

| list error '\n'

{

printf("error: invalid statement\n");

yyerrok; // Recover from error

}

;

stat:

expr

{

printf("%d\n", $1); // Print the result of the expression

}

|

LETTER '=' expr

{

printf("Stored: %c = %d\n", $1 + 'A', $3);

regs[$1] = $3; // Store the value in the register

}

;

expr:

'(' expr ')'

{

$$ = $2; // Return the value inside parentheses

}

|

expr '+' expr

{

$$ = $1 + $3; // Addition

}

|

expr '-' expr

{

$$ = $1 - $3; // Subtraction

}

|

expr '\*' expr

{

$$ = $1 \* $3; // Multiplication

}

|

expr '/' expr

{

if ($3 == 0)

yyerror("division by zero");

else

$$ = $1 / $3; // Division

}

|

expr '%' expr

{

if ($3 == 0)

yyerror("division by zero");

else

$$ = $1 % $3; // Modulus

}

|

LETTER

{

$$ = regs[$1]; // Retrieve value from register

}

|

number

;

number:

DIGIT

{

$$ = $1; // Return the digit value

base = ($1 == 0) ? 8 : 10; // Set base

}

|

number DIGIT

{

printf("%d %d\n", $1, $2);

$$ = base \* $1 + $2; // Construct the number

}

;

%% // End of grammar rules

int main(void)

{

return yyparse(); // Start parsing

}

void yyerror(const char \*s)

{

fprintf(stderr, "error: %s\n", s); // Print error message

}

int yywrap(void)

{

return 1; // End of input

}

**calc.lex**

extern int yylval;

%}

%%

" " ;

[a-z] {

c = yytext[0];

yylval = c - 'a';

return(LETTER);

}

[0-9] {

c = yytext[0];

yylval = c - '0';

return(DIGIT);

}

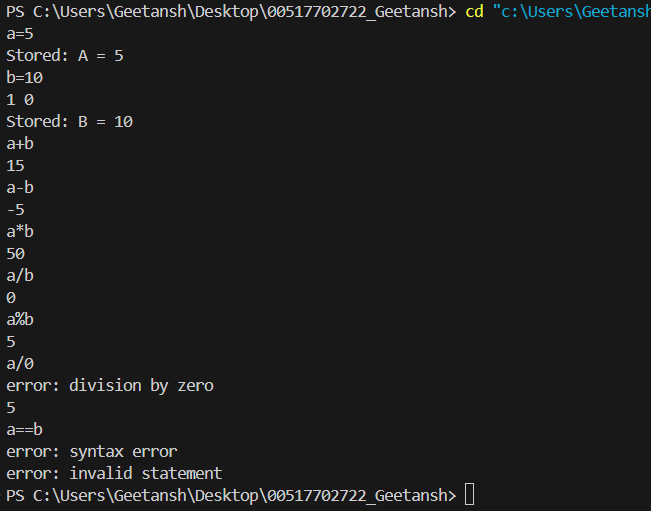
[^a-z0-9\b] {

c = yytext[0];

return(c);

}

**Output:**

****

**Learning Outcomes:**

# Experiment 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, **left recursion** occurs when a non-terminal in a production rule refers to itself as the first symbol on the right-hand side. Left recursion can lead to non-termination or inefficiencies when parsing, especially with **top-down parsers** like **recursive descent parsers**. Hence, it is necessary to detect and eliminate left recursion in grammars to make them **unambiguous** and suitable for efficient parsing.

**Key Concepts:**

1. **Context-Free Grammar**:
   * A **context-free grammar** (CFG) is a set of rules that define the structure of valid strings for a language. Each rule consists of a **non-terminal symbol** on the left-hand side and a combination of terminal and non-terminal symbols on the right-hand side.
   * A production rule in CFG is typically written as: A → Aα | β, where A is a non-terminal, α and β are strings of terminal and/or non-terminal symbols.
2. **Left Recursion**:
   * **Left recursion** occurs when a non-terminal symbol on the left-hand side of a production rule appears as the first symbol on the right-hand side. For example, in the rule A → Aα, the non-terminal A refers to itself.
   * Example:
     + A → Aα | β (left recursive)
     + Here, A on the left appears first in the right-hand side of the production.
3. **Right Recursion**:
   * **Right recursion** occurs when a non-terminal symbol appears on the right-hand side but not in the first position. It doesn't cause issues for top-down parsers.
   * Example:
     + A → βA | α (right recursive)
4. **Elimination of Left Recursion**:
   * Left recursion can be eliminated by converting a left-recursive grammar into a **right-recursive** form. The general technique involves transforming:
     + A → Aα | β
     + into an equivalent **non-recursive form**:
       - A → βA'
       - A' → αA' | ε
     + Here, A' is a new non-terminal symbol, α is the recursive part, β is the non-recursive part, and ε represents the **empty string**.

**Code:**

#include <stdio.h>

#include <string.h>

#include <ctype.h>

#define MAX 100

void checkLeftRecursion(char nonTerminal, char \*productions) {

    char alpha[MAX], beta[MAX];

    int isLeftRecursive = 0;

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

    int alphaIndex = 0, betaIndex = 0;

    while (token != NULL) {

        // Trim whitespace

        while (\*token == ' ') token++;

        if (token[0] == nonTerminal) {

            isLeftRecursive = 1;

            strcpy(&alpha[alphaIndex], token + 1); // Copy everything after the non-terminal

            alphaIndex += strlen(token + 1);

            alpha[alphaIndex++] = '|'; // Add separator

        } else {

            strcpy(&beta[betaIndex], token);

            betaIndex += strlen(token);

            beta[betaIndex++] = '|'; // Add separator

        }

        token = strtok(NULL, "|");

    }

    if (alphaIndex > 0) alpha[alphaIndex - 1] = '\0'; // Remove last '|'

    if (betaIndex > 0) beta[betaIndex - 1] = '\0'; // Remove last '|'

    if (isLeftRecursive) {

        printf("Left Recursive Grammar Detected.\n");

        printf("%c -> %s%c'\n", nonTerminal, beta, nonTerminal);

        printf("%c' -> %s%c' | e\n", nonTerminal, alpha, nonTerminal);

    } else {

        printf("No Left Recursion detected.\n");

    }

}

int validateProductionRule(char \*input) {

    if (strstr(input, "-->") != NULL) {

        printf("Error: '->' should not be greater than 3 characters long.\n");

        return 0;

    }

    if (!isupper(input[0]) || strncmp(input + 1, "->", 2) != 0) {

        printf("Error: Invalid production rule format.\n");

        return 0;

    }

    return 1;

}

int main() {

    char input[MAX], nonTerminal, productions[MAX];

    printf("Enter the production rule : ");

    fgets(input, MAX, stdin);

    input[strcspn(input, "\n")] = '\0'; // Remove the newline character

    if (!validateProductionRule(input)) {

        return 0;

    }

    nonTerminal = input[0];

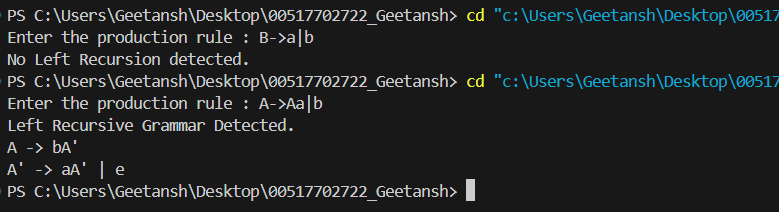
    strcpy(productions, input + 3); // Skip the non-terminal and "->"

    checkLeftRecursion(nonTerminal, productions);

    return 0;

}

**Output:**

****

**Learning Outcomes:**

# Experiment 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 the design of parsers for context-free grammars, **left factoring** is a process used to transform a grammar to remove ambiguity and make it suitable for **top-down parsing**. Left factoring is necessary when two or more production rules for the same non-terminal begin with the same symbols, leading to **non-determinism**. This makes it difficult for parsers to decide which production to apply. The solution is to rewrite the grammar by factoring out the common prefix, resulting in an unambiguous grammar.

**Key Concepts:**

1. **Context-Free Grammar (CFG)**:
   * A **context-free grammar** consists of a set of production rules where each rule describes how a **non-terminal** can be expanded into a sequence of terminal and non-terminal symbols.
   * A production rule is typically written as A → α | β, where A is a non-terminal and α and β are sequences of terminal and/or non-terminal symbols.
2. **Left Factoring**:
   * **Left factoring** is a situation in grammar when two or more productions for the same non-terminal begin with the same prefix. For example:
     + A → αβ1 | αβ2
   * In this case, the prefix α is common to both productions, leading to **non-determinism**, as a top-down parser cannot decide which production to apply by looking only at α.
   * **Eliminating Left Factoring**:
     + To eliminate left factoring, we rewrite the grammar to factor out the common prefix, introducing a new non-terminal symbol. The grammar is transformed as follows:
       - Original: A → αβ1 | αβ2
       - Transformed: A → αA'
         * A' → β1 | β2
3. **Ambiguity in Grammar**:
   * A grammar is **ambiguous** if a string can have more than one valid parse tree. Left factoring is one of the causes of ambiguity and non-determinism in parsers, making it difficult to decide the correct production to apply.
4. **Why Left Factoring is Important**:
   * **Top-down parsers**, especially **LL(1) parsers**, require grammars to be left-factored to ensure they can select the appropriate production based on one lookahead token. Left factoring resolves the ambiguity by restructuring the grammar, making it **deterministic** and easy to parse.
   * **Code:**

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#define MAX\_PRODUCTIONS 10

#define MAX\_LENGTH 100

void leftFactor(char \*input) {

    char nonTerminal;

    char productions[MAX\_PRODUCTIONS][MAX\_LENGTH];

    char commonPrefix[MAX\_LENGTH];

    int productionCount = 0;

    // Extract the non-terminal and productions

    sscanf(input, "%c --> %[^\n]", &nonTerminal, productions[0]);

    char \*token = strtok(productions[0], "|");

    while (token != NULL && productionCount < MAX\_PRODUCTIONS) {

        // Trim whitespace

        while (\*token == ' ') token++;

        strcpy(productions[productionCount], token);

        productionCount++;

        token = strtok(NULL, "|");

    }

    // Check for left factoring and find the common prefix

    int prefixLength = 0;

    int foundCommonPrefix = 1; // Assume there is a common prefix at start

    // Find the longest common prefix

    while (foundCommonPrefix) {

        for (int i = 1; i < productionCount; i++) {

            if (productions[i][prefixLength] != productions[0][prefixLength]) {

                foundCommonPrefix = 0;

                break;

            }

        }

        if (foundCommonPrefix) {

            prefixLength++;

        }

    }

    // If there's no common prefix

    if (prefixLength == 0) {

        printf("No Left Factoring present.\n");

        return;

    }

    // Create the left-factored grammar

    strncpy(commonPrefix, productions[0], prefixLength);

    commonPrefix[prefixLength] = '\0'; // Null terminate the common prefix

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

    printf("%c --> %s%c'\n", nonTerminal, commonPrefix, nonTerminal);

    printf("%c' --> ", nonTerminal);

    int first = 1;

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

        if (strncmp(productions[i], commonPrefix, prefixLength) == 0) {

            if (!first) {

                printf(" | ");

            }

            printf("%s", productions[i] + prefixLength);

            first = 0;

        }

    }

    printf(" | e\n");

}

int main() {

    char input[MAX\_LENGTH];

    printf("Enter a production rule : ");

    fgets(input, sizeof(input), stdin);

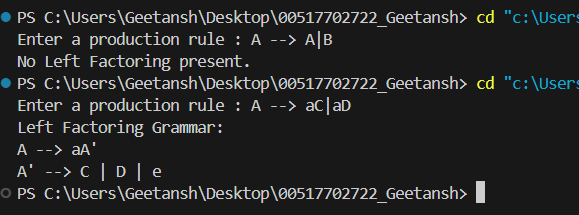
    input[strcspn(input, "\n")] = 0; // Remove the newline character

    leftFactor(input);

    return 0;

}

**Output:**

****

**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 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}

* 1. **For Non-Terminal Symbols**
* If there is a production of the form: X→Y1Y2…Yn.

o If Y1 is a terminal, add it to FIRST(X).

o 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.

o 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>

#include <cctype> // For islower function

using namespace std;

map<char, set<char>> firstSets;

map<char, vector<string>> productions;

void computeFirst(char nonTerminal) {

// If first set is already computed for this nonTerminal, no need to recompute

if (firstSets.count(nonTerminal)) return;

set<char> firstSet;

// Iterate through all productions of the non-terminal

for (const string& production : productions[nonTerminal]) {

for (char symbol : production) {

if (islower(symbol) || symbol == 'e') {

// If symbol is a terminal or epsilon, add it to the first set

firstSet.insert(symbol);

break;

} else {

// If symbol is a non-terminal, compute its first set recursively

computeFirst(symbol);

set<char> symbolFirstSet = firstSets[symbol];

firstSet.insert(symbolFirstSet.begin(), symbolFirstSet.end());

// If the first set of the symbol contains epsilon, continue to next symbol

if (!symbolFirstSet.count('e')) break;

}

}

}

firstSets[nonTerminal] = firstSet; // Store the computed first set for the non-terminal

}

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);

// The non-terminal is the first character of the line

char nonTerminal = line[0];

stringstream ss(line.substr(4)); // Skip the "S-->" part of the production

// Split the production by '|'

while (getline(ss, token, '|')) {

productions[nonTerminal].push\_back(token);

}

}

// Compute first sets for all non-terminals

for (const auto& entry : productions) {

computeFirst(entry.first);

}

// Output the first sets

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)) << " ";

}

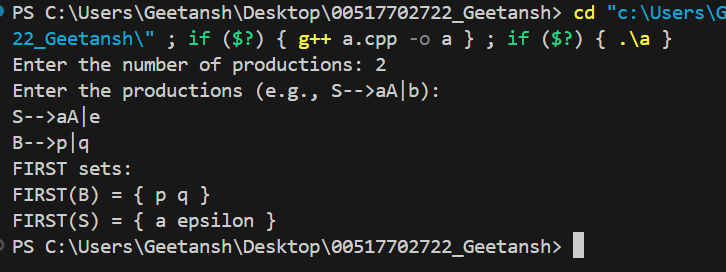
cout << "}" << endl;

}

return 0;

}

**Output:**



SHIVAM KUMAR LAL CSE-B

**Learning Outcomes:**

**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.

1. **Operations in Shift-Reduce Parsing** 
   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. **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 + " -> " + string(1, 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;

}

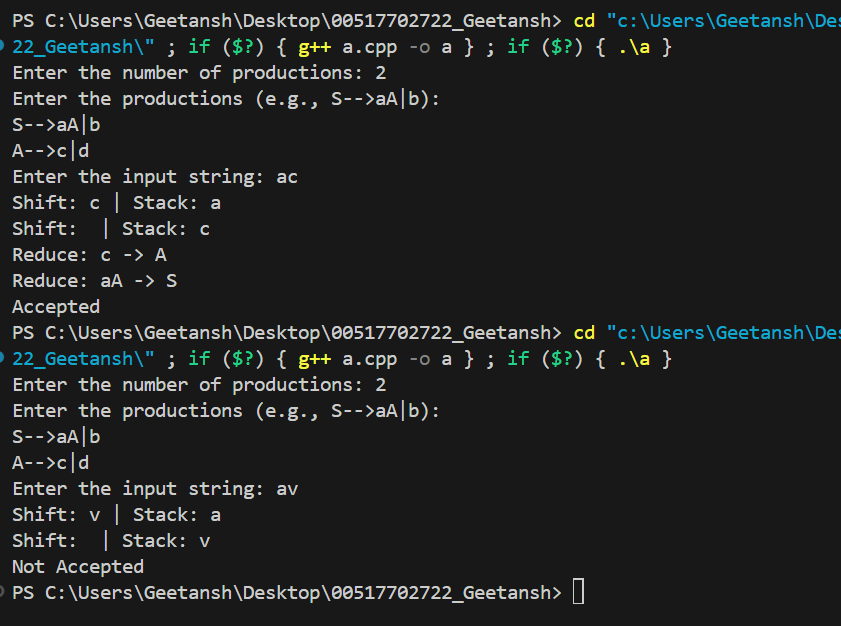
// Output the result

cout << (accepted ? "Accepted" : "Not Accepted") << endl;

return 0;

}

**Output:**



**Learning Outcomes:**

**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.

1. **Structure of Operator Precedence Grammar** 
   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).
  1. **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.

1. **Rules to check whether operator precedent:** 
   * + 1. **no two non-terminals should appear together.**
       2. **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 precedence." << endl;

} else {

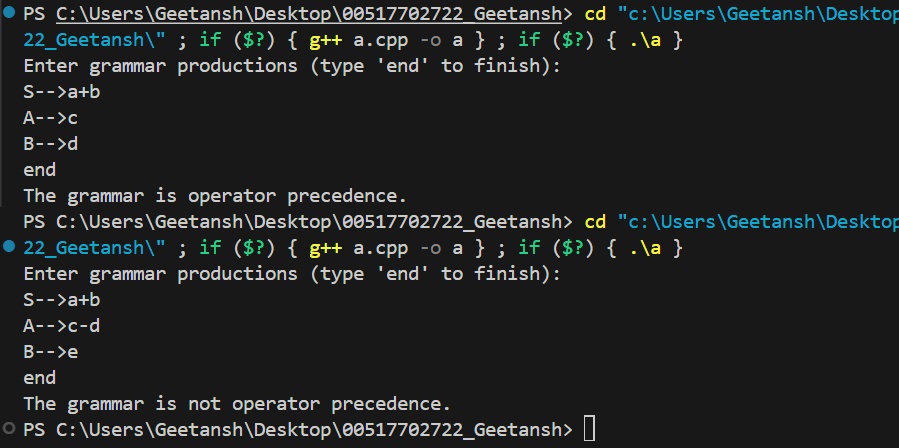
cout << "The grammar is not operator precedence." << endl;

}

return 0;

}

**Output:**



**Learning Outcomes:**