Lab - 1

Aim: Practice of LEX / YACC in compiler writing.

a. Write a program to print “Compiler” when input is “Hi” else print “Wrong Input”
b. Write a program to check whether a number is even or odd.
c. Write a program to find the greatest of two numbers.
d. Write a program to create a basic calculator.

a) Code:

```
%{
#include <stdio.h>
%}
%%
^Hi$ { printf("Compiler\n"); }
.|\n { printf("Wrong Input\n"); }
%%
int main() {
 yylex(); // Start lexical analysis
 return 0;
}
int yywrap(){
 return 0;
}
```

Output:

```
C:\Users\mso28\Lex>flex exp1.l

C:\Users\mso28\Lex>gcc lex.yy.c

C:\Users\mso28\Lex>a.exe
Hi
Compiler

Bye
Wrong Input
```

b) Code:

```
%{
#include <stdio.h>
%}
%%
[0-9]+ {
```

```
 if (atoi(yytext) % 2 == 0)
 printf("Even\n");
 else
 printf("Odd\n");}
.* { printf("Wrong Input\n"); }
%%
int main() {
 yylex(); // Start lexical analysis
 return 0;}
int yywrap(){
 return 0;
}
```

Output:

```
C:\Users\mso28\Lex>a.exe
23
Odd

46
Even
```

c) **Code:**

```
%{
#include <stdio.h>
int a, b, c = 0;
%}
%%
[0-9]+ {
 if (c == 0)
 a = atoi(yytext);
 else
 b = atoi(yytext);
}
\n {
 if (a > b)
 printf("%d is greater\n", a);
 else if (a < b)
 printf("%d is greater\n", b);
 else
 printf("Both are equal\n");
 c = 0;
}
. {
 c = 1;
```

```
}
%%
int main() {
 printf("Enter two space-separated integers: \n");
 yylex();
 return 0;
}
int yywrap() {
 return 0;
}
```

Output:

```
C:\Users\mso28\Lex>a.exe
Enter two space-separated integers :-
21 4
21 is greater
5 6
6 is greater
8 8
Both are equal
```

d) Code:

```
%{
#include <stdio.h>
int a, b, c = 0;
%}
%%
[0-9]+ {
 if (c == 0)
 a = atoi(yytext);
 else
 b = atoi(yytext);
}
\n {
 if (a > b)
 printf("%d is greater\n", a);
 else if (a < b)
 printf("%d is greater\n", b);
 else
 printf("Both are equal\n");
 c = 0;
}
. {
 c = 1;
```

```
}
%%
int main() {
 printf("Enter two space-separated integers: \n");
 yylex();
 return 0;
}
int yywrap() {
 return 0;
}
```

Output:

```
C:\Users\mso28\Lex>a.exe
5 + 12
Result is :  17.000000
3 * 4
Result is :  12.000000
9 - 6
Result is :  3.000000
18/4
Result is :  4.500000
```

Lab - 2

Aim: Write a program to check whether a string belongs to the grammar or not.

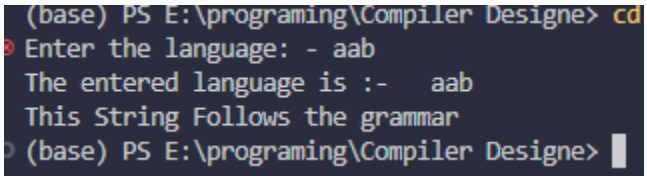
2. a) S -> aS
   S -> Sb
   S -> ab
   String of the form : aab
2. b) S -> aSa
   S -> bSb
   S -> a
   S -> b
   The Language generated is : All Odd Length Palindromes
2. c) S -> aSbb
   S -> abb
   The Language generated is : anb2n , where n>1
2. d) S -> aSb
   S -> ab
   The Language generated is : anbn, where n>0

a) Code:

```
#include <iostream>

using namespace std;

int check(string str){

   if(str.size() < 2 || str[str.size() -1 ] != 'b' || str[0] == 'b'){
      return 0;
   }

   for (int i = 0; i < str.size() - 1; i++)
   {
      if (str[i] != 'a' && str[i] != 'b') return 0;
      if (str[i] == 'b' && str[i + 1] == 'a') return 0;
   }
return 1;
}

int main ()
```

```
{
    string lang;
    cout<<"Enter the language: - ";
    cin >> lang;
    cout<<"The entered language is :-   " << lang << endl;
    auto checked = check(lang);
    if(checked) cout<<"This String Follows the grammar";
    else cout<<"This string does not follow the grammar";
    return 1;
}
```

Output:-



b) **Code :-**

```
#include<bits/stdc++.h>

using namespace std;

bool check(string lang){
    if(lang.size() == 1){
        return true;
    }

    if(lang.size()%2 == 0){
        return false;
    }

    int left = 0;
    int right = lang.size() - 1;

    while(left < right){
        if(lang[left] == lang[right]){
            right--;
            left++;
```

```
        }
        else{
            return false;
        }
    }
    return true;

}

int main(){
    string lang;
    cout<<"Enter the language: - ";
    cin >> lang;
    cout<<"The entered language is :-   " << lang << endl;
    auto checked = check(lang);
    if(checked) cout<<"This String Follows the grammar";
    else cout<<"This string does not follow the grammar";
    return 1;
}
```

Output :-

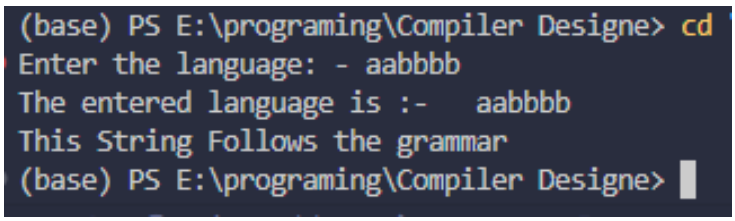
```
(base) PS E:\programing\Compiler Designe> cd "e:
Enter the language: - abbabba
The entered language is :-   abbabba
This String Follows the grammar
(base) PS E:\programing\Compiler Designe>
```

c) **Code :-**

```
#include <iostream>

using namespace std;

int check(string str){

    if(str.size() < 2 || str[str.size() -1 ] != 'b' || str[0] == 'b'){
        return 0;
    }

    int countOfA = 0;
```

```
    int countOfB = 1;

    for (int i = 0; i < str.size() - 1; i++)
    {
        if (str[i] != 'a' && str[i] != 'b') return 0;
        if (str[i] == 'b' && str[i + 1] == 'a') return 0;

        if(str[i] == 'a') countOfA++;
        else if(str[i] == 'b') countOfB++;
    }

    if(countOfB != 2*countOfA) return 0;

return 1;
}

int main ()
{
    string lang;
    cout<<"Enter the language: - ";
    cin >> lang;
    cout<<"The entered language is :-   " << lang << endl;
    auto checked = check(lang);
    if(checked) cout<<"This String Follows the grammar";
    else cout<<"This string does not follow the grammar";
    return 1;
}
```

Output: -

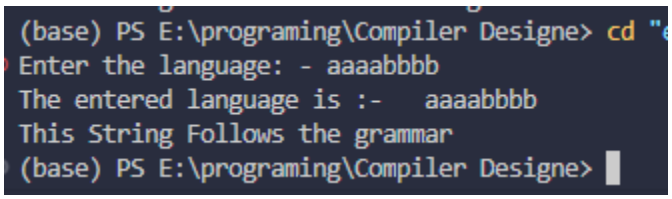


d) Code :-

```
#include <iostream>
using namespace std;
```

```
int check(string str){
    if(str.size() < 2 || str[str.size() -1 ] != 'b' || str[0] == 'b'){
        return 0;
    }
    int countOfA = 0;
    int countOfB = 1;
    for (int i = 0; i < str.size() - 1; i++)
    {
        if (str[i] != 'a' && str[i] != 'b') return 0;
        if (str[i] == 'b' && str[i + 1] == 'a') return 0;
        if(str[i] == 'a') countOfA++;
        else if(str[i] == 'b') countOfB++;
    }

    if(countOfB != 2*countOfA) return 0;

return 1;
}

int main ()
{
    string lang;
    cout<<"Enter the language: - ";
    cin >> lang;
    cout<<"The entered language is :-   " << lang << endl;
    auto checked = check(lang);
    if(checked) cout<<"This String Follows the grammar";
    else cout<<"This string does not follow the grammar";
    return 1;
}
```

Output: -



Lab - 3

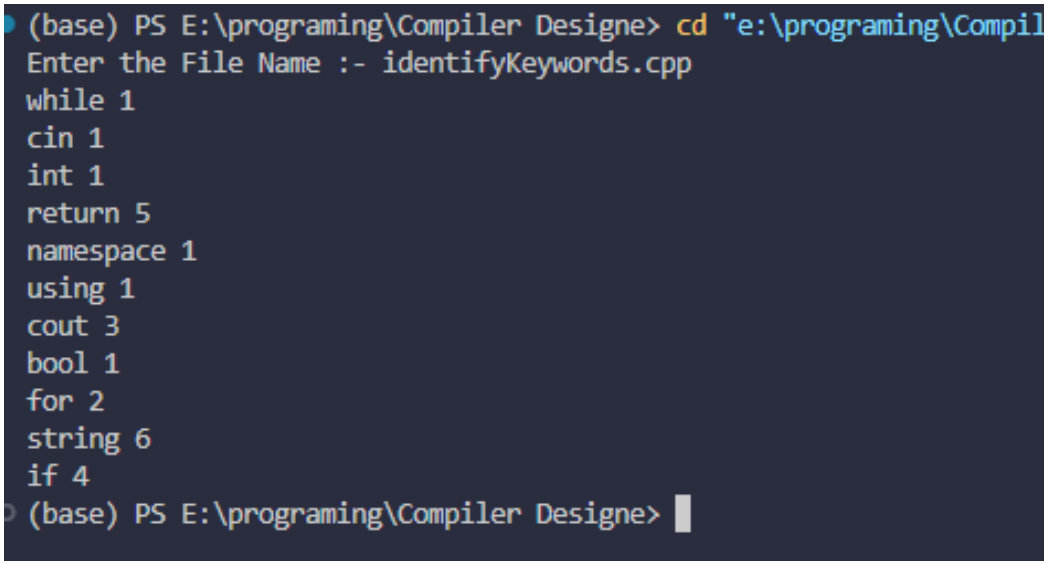
Aim: Write a program to check whether a string includes a Keyword or not.

Code :

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

using namespace std;

bool isKeyword(string word)
{
    string keywords[] = {"int", "float", "double", "char", "string", "bool", "void", "long", "short", "signed", "unsigned", "auto", "const", "extern", "register", "static", "volatile", "struct", "union", "enum", "typedef", "sizeof", "if", "else", "switch", "case", "default", "break", "continue", "for", "do", "while", "goto", "return", "using", "namespace", "std", "cout", "cin", "endl", "main"};
    for (int i = 0; i < 40; i++)
    {
        if (word == keywords[i])
        {
            return true;
        }
    }
    return false;
}

string readFile(string fileName)
{

    ifstream file(fileName);

    if (!file.is_open())
    {
        cerr << "Error: Could not open the file." << endl;
        return "File Not Found";
```

```
    }

    string fileContents((istreambuf_iterator<char>(file)),
                        istreambuf_iterator<char>());

    file.close();

    return fileContents;
}

int main()
{
    unordered_map<string, int> keywords;
    string fileName = "";
    cout << "Enter the File Name :- ";
    cin >> fileName;
    string fileContents = readFile(fileName);
    if (fileContents == "File Not Found")
    {
        cout << "No file Named " << fileName << " Found";
        return 0;
    }
    istringstream iss(fileContents);
    string word;

    while (iss >> word)
    {
        if (isKeyword(word))
        {
            keywords[word]++;
        }
    }
    for (auto i : keywords)
    {
        cout << i.first << " " << i.second << endl;
    }
}
```

Output :-


(base) PS E:\programing\Compiler Designe> cd "e:\programing\Compil
Enter the File Name :- identifyKeywords.cpp
while 1
cin 1
int 1
return 5
namespace 1
using 1
cout 3
bool 1
for 2
string 6
if 4
(base) PS E:\programing\Compiler Designe>

Lab - 4

Aim: Write a program to remove left recursion from a grammar.

Code :

```
#include <bits/stdc++.h>
using namespace std;
vector<string> g;

class NonTerminal
{
   string name;
   vector<string> productionRules;

public:
   NonTerminal(string name)
   {
      this->name = name;
   }
   string getName()
   {
      return name;
   }
   void setRules(vector<string> rules)
   {
      productionRules.clear();
      for (auto rule : rules)
      {
         productionRules.push_back(rule);
      }
   }
   vector<string> getRules()
   {
      return productionRules;
   }
   void addRule(string rule)
   {
      productionRules.push_back(rule);
   }
   void printRule()
```

```
    {
        string toPrint = "";
        toPrint += name + " ->";
        for (string s : productionRules)
        {
            toPrint += " " + s + " |";
        }
        toPrint.pop_back();
        g.push_back(toPrint);
    }
};
class Grammar
{
    vector<NonTerminal> nonTerminals;

public:
    void addRule(string rule)
    {
        bool nt = 0;
        string parse = "";

        for (char c : rule)
        {
            if (c == ' ')
            {
                if (!nt)
                {
                    NonTerminal newNonTerminal(parse);

                    nonTerminals.push_back(newNonTerminal);
                    nt = 1;
                    parse = "";
                }
                else if (parse.size())
                {
                    nonTerminals.back().addRule(parse);
                    parse = "";
                }
            }
            else if (c != '|' && c != '-' && c != '>')
```

```
        {
            parse += c;
        }
    }
    if (parse.size())
    {
        nonTerminals.back().addRule(parse);
    }
}
void inputData(vector<string> &s)
{
    for (int i = 0; i < s.size(); i++)
    {
        addRule(s[i]);
    }
}
void solveNonImmediateLR(NonTerminal &A, NonTerminal &B)
{
    string nameA = A.getName();
    string nameB = B.getName();
    vector<string> rulesA, rulesB, newRulesA;
    rulesA = A.getRules();
    rulesB = B.getRules();
    for (auto rule : rulesA)
    {
        if (rule.substr(0, nameB.size()) == nameB)
        {
            for (auto rule1 : rulesB)
            {
                newRulesA.push_back(rule1 +

                                    rule.substr(nameB.size()));
            }
            else
            {
            }
        }

        newRulesA.push_back(rule);
    }
```

```
    A.setRules(newRulesA);
}
void solveImmediateLR(NonTerminal &A)
{
    string name = A.getName();
    string newName = name + "'";

    vector<string> alphas, betas, rules, newRulesA, newRulesA1;
    rules = A.getRules();
    for (auto rule : rules)
    {
        if (rule.substr(0, name.size()) == name)
        {
            alphas.push_back(rule.substr(name.size()));
        }
        else
        {
        }
    }

    betas.push_back(rule);

    if (!alphas.size())
        return;
    if (!betas.size())
        newRulesA.push_back(newName);
    for (auto beta : betas)
        newRulesA.push_back(beta + newName);
    for (auto alpha : alphas)
        newRulesA1.push_back(alpha + newName);
    A.setRules(newRulesA);
    newRulesA1.push_back("\u03B5");
    NonTerminal newNonTerminal(newName);
    newNonTerminal.setRules(newRulesA1);
    nonTerminals.push_back(newNonTerminal);
}
void applyAlgorithm()
{
    int size = nonTerminals.size();
    for (int i = 0; i < size; i++)
```

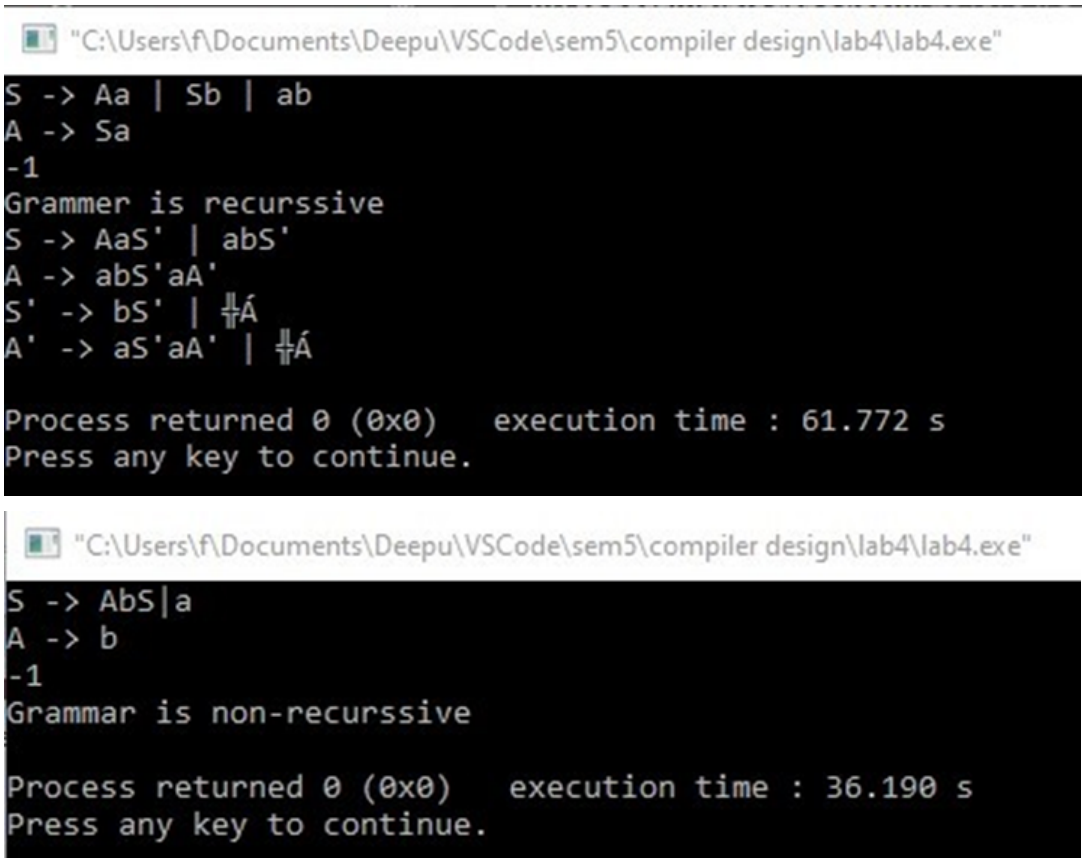
```
        {
            for (int j = 0; j < i; j++)
            {
                solveNonImmediateLR(nonTerminals[i],

                                    nonTerminals[j]);
            }
        }
        solveImmediateLR(nonTerminals[i]);
    }
    void printRules()
    {
        for (auto nonTerminal : nonTerminals)
        {
            nonTerminal.printRule();
        }
    }
};
int main()
{
    vector<string> v;
    string s;
    getline(cin, s);
    while (s != "-1")
    {
        v.push_back(s);
        getline(cin, s);
    }
    Grammar grammar;
    grammar.inputData(v);
    grammar.applyAlgorithm();
    grammar.printRules();
    if (v.size() == g.size())
    {
        cout << "Grammar is non-recurssive" << endl;
    }
    else
    {
        cout << "Grammer is recurssive" << endl;
        for (auto i : g)
```

```
    {
        cout << i << endl;
    }
  }
```

Output:

```
   return 0;
}
```

Output:-



Lab - 5

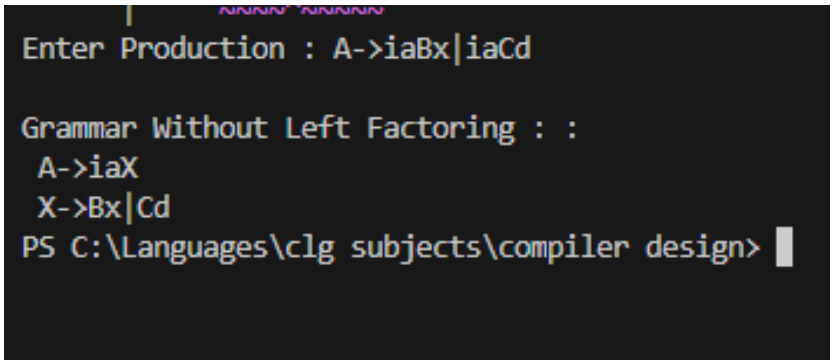
Aim: Write a program to perform left factoring on a grammer.

Code :

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

int main()
{

   char gram[20], part1[20], part2[20], modifiedGram[20], newGram[20],
tempGram[20];
   int i, j = 0, k = 0, l = 0, pos;
   printf("Enter Production : A->");
   gets(gram);
   for (i = 0; gram[i] != '|'; i++, j++)
      part1[j] = gram[i];
   part1[j] = '\0';
   for (j = ++i, i = 0; gram[j] != '\0'; j++, i++)
      part2[i] = gram[j];
   part2[i] = '\0';
   for (i = 0; i < strlen(part1) || i < strlen(part2); i++)
   {
      if (part1[i] == part2[i])
      {
         modifiedGram[k] = part1[i];
         k++;
         pos = i + 1;
      }
   }
   for (i = pos, j = 0; part1[i] != '\0'; i++, j++)
   {
      newGram[j] = part1[i];
   }
   newGram[j++] = '|';
   for (i = pos; part2[i] != '\0'; i++, j++)
   {
      newGram[j] = part2[i];
   }
```

```
    modifiedGram[k] = 'X';
    modifiedGram[++k] = '\0';
    newGram[j] = '\0';
    printf("\nGrammar Without Left Factoring : : \n");
    printf(" A->%s", modifiedGram);
    printf("\n X->%s\n", newGram);
}
```

Output:



Lab - 6

Aim: Write a program to show all the operations of a stack.

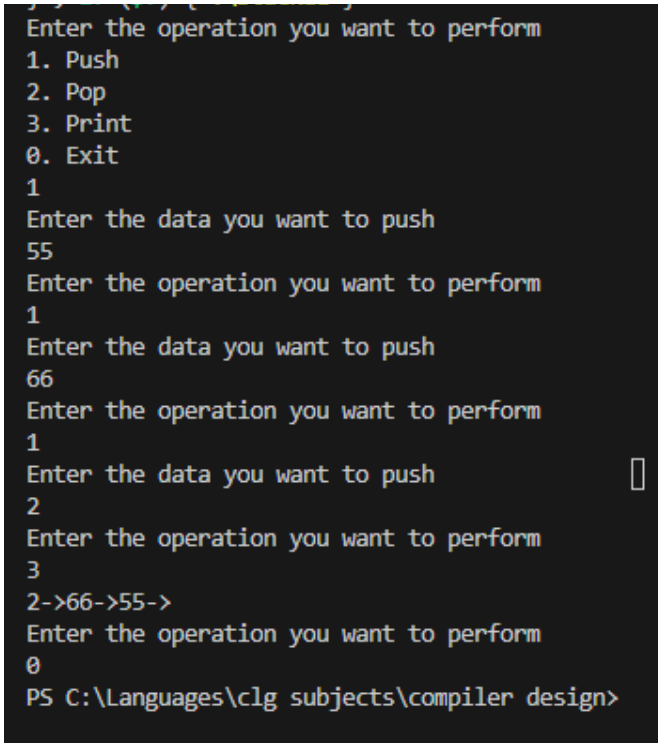
Code :

```
#include <stdio.h>
#include<string.h>
#include<bits/stdc++.h>
using namespace std;

class Node{
   public:
   Node* next;
   int data;

   Node(int data){
      this->data = data;
      this->next = NULL;
   }
};

class StackLL{
   public:
   Node* top;

   StackLL(){
      top = NULL;
   }

   void push(int data){
      Node *newNode = new Node(data);
      if(this->top == NULL){
         newNode->next = NULL;
         top = newNode;
         return;
      }
      newNode->next = top;
      top = newNode;
   }
```

```
    void pop(){
        if(this->top == NULL){
            cout<<"There is nothing to pop" <<endl;
            return;
        }
        Node * temp;
        temp = this->top;
        top = temp->next;
        cout<<temp->data<<" poped";
        delete(temp);
    }

    void printStack(){
        if(this->top == NULL){
            cout<<"there is nothing to print "<<endl;
            return;
        }
        Node* temp = top;
        while (temp != NULL){
            cout<<temp->data<<"->";
            temp = temp->next;
        }
        cout<<endl;

    }
};

int main(){
    StackLL stk;
    cout<<"Enter the operation you want to perform"<<endl;
    cout<<"1. Push"<<endl;
    cout<<"2. Pop"<<endl;
    cout<<"3. Print"<<endl;
    cout<<"0. Exit"<<endl;
    int choice;
    cin>>choice;
    while(choice != 0){
        switch (choice)
        {
        case 1:
```

```
            cout<<"Enter the data you want to push"<<endl;
            int data;
            cin>>data;
            stk.push(data);
            break;
        case 2:
            stk.pop();
            break;
        case 3:
            stk.printStack();
            break;
        default:
            cout<<"Enter the correct choice"<<endl;
            break;
        }
        cout<<"Enter the operation you want to perform"<<endl;
        cin>>choice;
    }
    return 0;
}
```

Output:



Lab - 7

Aim: Write a program to find out the leading of the non-terminals in a grammar.

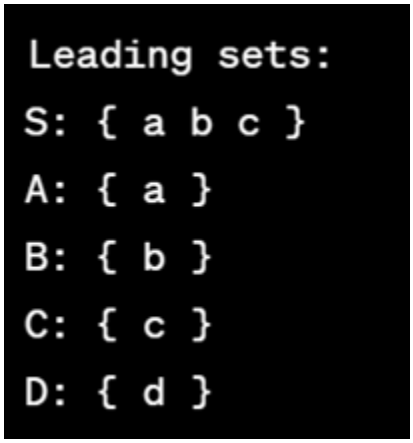
Code :

```
#include <stdio.h>
#include <iostream>
#include <unordered_map>
#include <set>
#include <sstream>
using namespace std;

void addSet(set<char> &destination, const set<char> &source)
{
   destination.insert(source.begin(), source.end());
}
unordered_map<char, set<char>> findLeadingSets(const string &grammar)
{
   unordered_map<char, set<char>> leadingSets;

   istringstream grammarStream(grammar);
   string production;

   while (getline(grammarStream, production))
   {
      istringstream ruleStream(production);

      char nonTerminal;
      ruleStream >> nonTerminal;

      ruleStream.ignore(2);

      string body;
      ruleStream >> body;

      set<char> firstSet;
      for (char symbol : body)
      {
         if (isupper(symbol))
         {
```

```
                addSet(firstSet, leadingSets[symbol]);
                if (leadingSets[symbol].count('$') == 0)
                {
                    break;
                }
            }
            else
            {
                firstSet.insert(symbol);
                break;
            }
        }
        leadingSets[nonTerminal].insert(firstSet.begin(), firstSet.end());
    }

    return leadingSets;
}

void displayLeadingSets(const unordered_map<char, set<char>> &leadingSets)
{
    cout << "Leading sets:\n";
    for (const auto &entry : leadingSets)
    {
        cout << entry.first << ": { ";
        for (char symbol : entry.second)
        {
            cout << symbol << ' ';
        }
        cout << "}\n";
    }
}

int main()
{
    string grammar =
        "S -> AbCd\n"
        "A -> a | $\n"
        "B -> b | $\n"
        "C -> c | $\n"
        "D -> d | $\n";
```

```
    unordered_map<char, set<char>> leadingSets = findLeadingSets(grammar);

    displayLeadingSets(leadingSets);

    return 0;
}
```

Output:



Lab - 8

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

Code :

```
#include <iostream>
#include <stack>
#include <vector>
#include <unordered_map>
using namespace std;

vector<string> productions = {"E->E+T", "E->T", "T->T*F", "T->F", "F->(E)",
"F->id"};

unordered_map<string, unordered_map<string, string>> parsingTable = {
   {"E", {{"(", "shift T"}, {"id", "shift T"}}},
   {"T", {{"(", "shift F"}, {"id", "shift F"}}},
   {"F", {{"(", "shift (E)"}, {"id", "shift id"}}}};

void shift(stack<string> &parseStack, vector<string> &input)
{
   parseStack.push(input[0]);
   input.erase(input.begin());
}

void reduce(stack<string> &parseStack, string production)
{
   size_t arrowPos = production.find("->");
   string rhs = production.substr(arrowPos + 2);

   for (size_t i = 0; i < rhs.size(); ++i)
   {
      parseStack.pop();
   }
   parseStack.push(production.substr(0, arrowPos));
}

void parse(stack<string> &parseStack, vector<string> &input)
{
   while (!input.empty())
```

```
        {
            string stackTop = parseStack.top();
            string nextInput = input[0];

            if (parsingTable[stackTop].find(nextInput) != parsingTable[stackTop].end())
            {
                string action = parsingTable[stackTop][nextInput];
                if (action.substr(0, 5) == "shift")
                {
                    shift(parseStack, input);
                }
                else if (action.substr(0, 5) == "reduce")
                {
                    reduce(parseStack, action.substr(6));
                }
            }
            else
            {
                cout << "Error: Invalid pair (" << stackTop << ", " << nextInput << ")" <<
endl;
                break;
            }

            cout << "Stack: ";
            stack<string> tempStack = parseStack;
            while (!tempStack.empty())
            {
                cout << tempStack.top() << " ";
                tempStack.pop();
            }
            cout << "\tInput: ";
            for (const auto &symbol : input)
            {
                cout << symbol << " ";
            }
            cout << endl;
        }
    }

    int main()
```

```
{
    vector<string> input = {"id", "+", "id", "*", "id", "$"};

    stack<string> parseStack;

    parseStack.push("E");

    parse(parseStack, input);

    return 0;
}
```

Output:

```
Stack: E     Input: id + id * id $
Stack: E id  Input: + id * id $
Stack: E T   Input: + id * id $
Stack: E T + Input: id * id $
Stack: E T   Input: id * id $
Stack: E T F Input: * id $
Stack: E T   Input: * id $
Stack: E T   Input: id $
Stack: E T F Input: $
Stack: E T   Input: $
Stack: E     Input: $
```

Lab - 9

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

Code :

```
#include <iostream>
#include <unordered_map>
#include <unordered_set>
#include <vector>

using namespace std;

struct Production
{
   string nonTerminal;
   string expression;
};

unordered_set<char> calculateFirst(const string &nonTerminal, const
vector<Production> &productions, unordered_map<string, unordered_set<char>>
&firstCache)
{
   if (firstCache.find(nonTerminal) != firstCache.end())
   {
      return firstCache[nonTerminal];
   }

   unordered_set<char> firstSet;

   for (const Production &production : productions)
   {
      if (production.nonTerminal == nonTerminal)
      {
         char firstSymbol = production.expression[0];

         if (isalpha(firstSymbol) && islower(firstSymbol))
         {
            firstSet.insert(firstSymbol);
         }
         else if (isalpha(firstSymbol) && isupper(firstSymbol))
```

```
                {
                    unordered_set<char> firstSetOfSymbol = calculateFirst(string(1,
firstSymbol), productions, firstCache);
                    firstSet.insert(firstSetOfSymbol.begin(), firstSetOfSymbol.end());
                }
                else if (firstSymbol == '\0')
                {
                    firstSet.insert('\0');
                }
            }
        }

        firstCache[nonTerminal] = firstSet;

        return firstSet;
    }

    int main()
    {
        vector<Production> productions = {
            {"S", "aAB"},
            {"A", "b"},
            {"A", ""},
            {"B", "cC"},
            {"C", "d"},
            {"C", ""}};

        unordered_map<string, unordered_set<char>> firstCache;

        cout << "FIRST sets:" << endl;
        for (const Production &production : productions)
        {
            string nonTerminal = production.nonTerminal;
            unordered_set<char> firstSet = calculateFirst(nonTerminal, productions,
firstCache);

            cout << nonTerminal << ": { ";
            for (char symbol : firstSet)
            {
                cout << symbol << ' ';
```

```
        }
        cout << '}' << endl;
    }

    return 0;
}
```

Output:

```
FIRST sets:
S: { a }
A: { b, ϵ }
B: { c, ϵ }
C: { d, ϵ }
```

Lab - 10

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

Code :

```
#include <iostream>
#include <iostream>
#include <vector>
#include <unordered_set>
#include <stack>

using namespace std;

// Data structure to represent a production rule
struct Production
{
    string left;
    string right;
};

// Function to check if the grammar is operator precedence
bool isOperatorPrecedence(const vector<Production> &productions)
{
    unordered_set<string> terminals;
    unordered_set<string> nonTerminals;
    unordered_set<string> operators;

    // Collect terminals, non-terminals, and operators
    for (const Production &production : productions)
    {
        nonTerminals.insert(production.left);
        for (char symbol : production.right)
        {
            string symbolStr(1, symbol);
            if (isalpha(symbol) && islower(symbol))
            {
                terminals.insert(symbolStr);
            }
            else
            {
```

```
                operators.insert(symbolStr);
            }
        }
    }

    // Check for overlapping symbols between terminals and operators
    for (const string &terminal : terminals)
    {
        if (operators.find(terminal) != operators.end())
        {
            return false;
        }
    }

    // Check if the grammar is operator precedence
    for (const Production &production : productions)
    {
        string right = production.right;
        if (right.length() >= 2 && isalpha(right[0]) && isalpha(right[1]) &&
isupper(right[1]))
        {
            return false; // Invalid adjacent non-terminals
        }
    }

    return true;
}

int main()
{
    // Define the grammar with production rules
    vector<Production> productions = {
        {"E", "E+T"},
        {"E", "T"},
        {"T", "T*F"},
        {"T", "F"},
        {"F", "(E)"},
        {"F", "id"}};

    // Check if the grammar is operator precedence
```

```
    if (isOperatorPrecedence(productions))
    {
        cout << "The grammar is operator precedence." << endl;
    }
    else
    {
        cout << "The grammar is not operator precedence." << endl;
    }

    return 0;
}
```

Output: