



# **Jashore University of Science and Technology**

## **Department of Computer Science and Engineering**

**Course Title:** Compiler Design and Automata Theory Lab

**Course Code:** CSE-3206

### **Lab Report on** **Compiler Lab Problems**

Submitted to	Submitted by
<b>Dr. Syed Md. Galib</b> <b>Professor</b>  Dept. of Computer Science and Engineering. Jashore University of Science and Technology.	<b>Md. Mosiur Rahman Romel</b>  Student ID: 180149  3 <sup>rd</sup> year, 2 <sup>nd</sup> semester  Dept. of Computer Science and Engineering Jashore University of Science and Technology.

**Remarks:**

Date of Submission: 20-01-2024

## Problem Statement: 1

Write a LEX Program to scan reserved word & Identifiers of C Language.

## Introduction:

The task is to develop a Lex program that performs lexical analysis on C source code, specifically identifying reserved words and identifiers. The purpose of this lab is to gain a practical understanding of lexical analysis and to implement a simple lexical analyzer using Lex for the C programming language.

## Requirements:

Tools and Technologies

- Lex (Flex)
- C Compiler (gcc)
- Text Editor
- Ubuntu
- WSL

## Methodology:

- Lex Program Development  
The Lex program was developed to recognize C keywords (reserved words) and identifiers. Regular expressions were defined to match the patterns of reserved words and identifiers.
- Compilation  
The Lex program was compiled using Flex to generate the lexical analyzer. The generated C file (lex.yy.c) was then compiled with the C compiler (gcc) to create the executable.
- Execution  
The Lex program was executed with various input C source code files to test its ability to identify reserved words and identifiers.

## Implementation:

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

```
% }

%option noyywrap

%%

auto|break|case|char|const|continue|default|do|double { printf("Reserved Word: %s\n", yytext); }
else|enum|extern|float|for|goto|if|int|long|register { printf("Reserved Word: %s\n", yytext); }
return|short|signed|sizeof|static|struct|switch|typedef|union { printf("Reserved Word: %s\n",
yytext); }
unsigned|void|volatile|while|_Packed { printf("Reserved Word: %s\n", yytext); }
[a-zA-Z_][a-zA-Z0-9_]* { printf("Identifier: %s\n", yytext); }
[ \t\n]          ; // Ignore whitespace
.                { printf("Invalid: %s\n", yytext); }

%%

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

## Input and Output:

```
rumel@rumelcse:~/4 th_year_cse/lab problem updated Solution/Lab Problem Solution$ flex Problem1.l
rumel@rumelcse:~/4 th_year_cse/lab problem updated Solution/Lab Problem Solution$ gcc lex.yy.c -o Problem1 -lfl
rumel@rumelcse:~/4 th_year_cse/lab problem updated Solution/Lab Problem Solution$ ./Problem1
a+b is not int
Identifier: a
Invalid: +
Identifier: b
Identifier: is
Identifier: not
Reserved Word: int
```

## Discussion:

- Observations  
The Lex program successfully identified reserved words and identifiers in the input C source code. It handled common C language constructs such as conditionals and variable declarations.
- Limitations

The current implementation does not handle preprocessor directives and might produce false positives for identifiers that match reserved words.

## **Conclusion:**

The lab provided valuable insights into lexical analysis using Lex. The developed program successfully identified reserved words and identifiers in C source code. Further improvements could be made to enhance its accuracy and handle additional language constructs.

## **Problem Statement: 2**

Implement Predictive Parsing algorithm

## **Introduction:**

Predictive Parsing is a top-down parsing technique that uses a predictive parse table to parse the input string. It requires a context-free grammar that is LL(1) (left-to-right scan, leftmost derivation, 1 token lookahead). In this lab, I implement the Predictive Parsing algorithm in C++.

## **Requirements:**

C++ compiler (e.g., g++)

Text editor or integrated development environment (IDE)

## **Methodology:**

To implement the Predictive Parsing algorithm, you can follow these general steps. Here's a methodology that you can include in your lab report:

- Grammar Definition  
Define the grammar using BNF or EBNF.
- Calculate First and Follow Sets  
Compute First and Follow sets for each non-terminal.
- Construct Parsing Table  
Create a 2D parsing table based on First and Follow sets.
- Handling Ambiguities and Conflicts  
Check for conflicts or ambiguities in the parsing table.  
Resolve conflicts if necessary.
- Write Predictive Parsing Code

Implement predictive parsing algorithm using the parsing table.

- Input String  
Provide sample input strings for testing.
- Execute and Debug  
Execute the predictive parsing program.  
Debug any parsing issues.

## Implementation:

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

using namespace std;

class PredictiveParser
{
public:
    PredictiveParser(const unordered_map<char, vector<string>> &grammar, char startSymbol)
        : grammar(grammar), startSymbol(startSymbol) {}

    void computeFirstSets()
    {
        for (const auto &nonTerminal : grammar)
        {
            calculateFirstSet(nonTerminal.first);
        }
    }

    void calculateFirstSet(char nonTerminal)
    {
        if (firstSets.count(nonTerminal))
        {
            return;
        }

        for (const string &production : grammar.at(nonTerminal))
        {
            char symbol = production[0];
            if (isupper(symbol))
            {
```

```

        calculateFirstSet(symbol);
        firstSets[nonTerminal].insert(firstSets[symbol].begin(), firstSets[symbol].end());
    }
    else
    {
        firstSets[nonTerminal].insert(symbol);
    }
}

void computeFollowSets()
{
    for (const auto &nonTerminal : grammar)
    {
        calculateFollowSet(nonTerminal.first);
    }
}

void calculateFollowSet(char nonTerminal)
{
    if (followSets.count(nonTerminal))
    {
        return;
    }

    followSets[nonTerminal];

    if (nonTerminal == startSymbol)
    {
        followSets[nonTerminal].insert('$');
    }

    for (const auto &production : grammar)
    {
        for (const string &rule : production.second)
        {
            size_t pos = rule.find(nonTerminal);
            while (pos != string::npos)
            {
                if (pos + 1 < rule.size())

```

```

        {
            char nextSymbol = rule[pos + 1];
            if (isupper(nextSymbol))
            {
                calculateFirstSet(nextSymbol);
                followSets[nonTerminal].insert(firstSets[nextSymbol].begin(),
firstSets[nextSymbol].end());
                if (firstSets[nextSymbol].count(' ') > 0)
                {
                    followSets[nonTerminal].insert(followSets[production.first].begin(),
followSets[production.first].end());
                }
            }
            else
            {
                followSets[nonTerminal].insert(nextSymbol);
            }
        }
        else
        {
            followSets[nonTerminal].insert(followSets[production.first].begin(),
followSets[production.first].end());
        }

        pos = rule.find(nonTerminal, pos + 1);
    }
}

void buildParseTable()
{
    for (const auto &nonTerminal : grammar)
    {
        for (const string &rule : nonTerminal.second)
        {
            unordered_set<char> first = calculateFirstOfString(rule);
            for (char symbol : first)
            {
                if (symbol != ' ')

```

```

        {
            parseTable[{nonTerminal.first, symbol}] = rule;
        }
    }
    if (first.count(' '))
    {
        for (char symbol : followSets[nonTerminal.first])
        {
            if (symbol != '$')
            {
                parseTable[{nonTerminal.first, symbol}] = rule;
            }
        }
    }
}

```

```

unordered_set<char> calculateFirstOfString(const string &str)
{
    unordered_set<char> firstSet;
    bool epsilonFlag = true;
    for (char symbol : str)
    {
        if (isupper(symbol))
        {
            calculateFirstSet(symbol);
            firstSet.insert(firstSets[symbol].begin(), firstSets[symbol].end());
            if (firstSets[symbol].count(' ') == 0)
            {
                epsilonFlag = false;
                break;
            }
        }
        else
        {
            firstSet.insert(symbol);
            epsilonFlag = false;
            break;
        }
    }
}

```



```

    }

    if (epsilonFlag)
    {
        firstSet.insert(' ');
    }

    return firstSet;
}

void parse(const string &input)
{
    stack<char> symbolStack;
    symbolStack.push('$');
    symbolStack.push(startSymbol);

    size_t inputIndex = 0;

    while (symbolStack.top() != '$')
    {
        char topOfStack = symbolStack.top();
        char currentInputSymbol = input[inputIndex];

        if (isupper(topOfStack))
        {
            if (parseTable.count({topOfStack, currentInputSymbol}))
            {
                const string &production = parseTable[{topOfStack, currentInputSymbol}];
                symbolStack.pop();
                if (production != "")
                {
                    for (auto it = production.rbegin(); it != production.rend(); ++it)
                    {
                        symbolStack.push(*it);
                    }
                }
            }
            else
            {
                cout << "Error: Invalid input.\n";
            }
        }
    }
}

```

```

        return;
    }
}
else if (topOfStack == currentInputSymbol)
{
    symbolStack.pop();
    ++inputIndex;
}
else
{
    cout << "Error: Invalid input.\n";
    return;
}
}

cout << "Input accepted.\n";
}

```

private:

```

unordered_map<char, vector<string>> grammar;
char startSymbol;
unordered_map<char, unordered_set<char>> firstSets;
unordered_map<char, unordered_set<char>> followSets;
map<pair<char, char>, string> parseTable; // Changed to map
};

```

int main()

```

{
    unordered_map<char, vector<string>> grammar = {
        {'E', {"T+E", "T"}},
        {'T', {"F*T", "F"}},
        {'F', {"(E)", "id"}}};

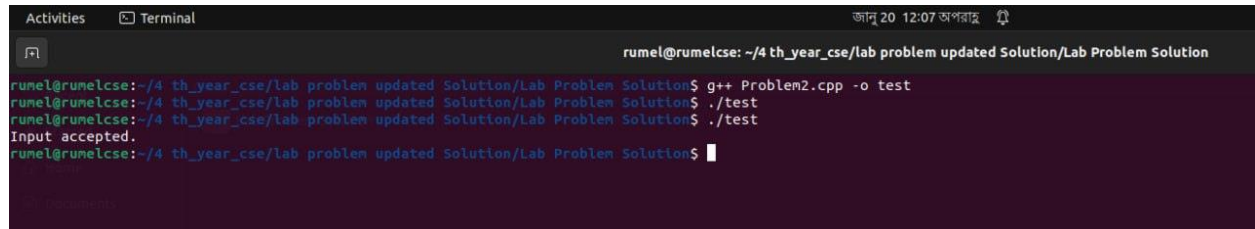
    PredictiveParser parser(grammar, 'E');
    parser.computeFirstSets();
    parser.computeFollowSets();
    parser.buildParseTable();

    string input = "id + id * id";
    parser.parse(input);
}

```

```
return 0;
}
```

## Input and Output:



```
rumel@rumelcse: ~/4 th_year_cse/lab problem updated Solution/Lab Problem Solution
rumel@rumelcse:~/4 th_year_cse/lab problem updated Solution/Lab Problem Solution$ g++ Problem2.cpp -o test
rumel@rumelcse:~/4 th_year_cse/lab problem updated Solution/Lab Problem Solution$ ./test
Input accepted.
rumel@rumelcse:~/4 th_year_cse/lab problem updated Solution/Lab Problem Solution$
```

## Discussion:

- Grammar Suitability: The chosen grammar must be devoid of left recursion and common prefixes to enable unambiguous predictive parsing.
- First and Follow Sets: The precision of First and Follow sets is pivotal for parsing table construction. Any inaccuracies can lead to conflicts during parsing.
- Parsing Table Construction: The parsing table is a key data structure guiding parsing decisions. Anomalies in its construction require careful scrutiny and correction.
- Handling Conflicts: Conflicts demand thorough investigation for resolution. Refining the grammar or adopting alternative parsing strategies may be necessary.

## Conclusion:

- Significance: Predictive Parsing is pivotal for syntactic analysis, contingent on a well-defined grammar and meticulous parsing table construction.
- Challenges: Conflict resolution poses challenges, necessitating careful consideration for an accurate and unambiguous parser.
- Future Considerations: Ongoing refinement and exploration of optimizations will enhance parser efficiency and versatility.

## Problem Statement: 3

Implement a program to generate three-address code

## Introduction:

Three-address code is an intermediate code representation where each instruction has at most three operands. The goal of this lab is to implement a program that takes a simple arithmetic expression as input and generates corresponding three-address code. The arithmetic expression is assumed to involve binary operators (+, -, \*, /) and parentheses.

## Requirements:

### Tools and Technologies

- C Compiler (gcc)
- Text Editor
- Ubuntu
- WSL

## Methodology:

- Input Source Code:
  - Take high-level programming language source code as input.
- Analysis:
  - Use Lex and Yacc/Bison for lexical and syntax analysis.
  - Build an abstract syntax tree (AST) during parsing.
- Semantic Analysis:
  - Check for correct variable declarations and types.
  - Create symbol tables for entities.
- AST Traversal:
  - Traverse AST to generate intermediate code.
- Code Generation:
  - Generate three-address code for each statement.
- Symbol Table Utilization:
  - Utilize symbol table during code generation..
- Output Three-Address Code:
  - Output generated code for examination.
- Testing:
  - Test with different input programs.
- Error Handling:
  - Implement error handling mechanisms.

## Implementation:

```

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

int tempCount = 1;

// Function to generate a new temporary variable
char* newTemp() {
    char* temp = (char*)malloc(4);
    snprintf(temp, 4, "t%d", tempCount++);
    return temp;
}

// Function to generate three-address code for addition
void generateAddition(char* op1, char* op2) {
    char* result = newTemp();
    printf("%s = %s + %s\n", result, op1, op2);
}

// Function to generate three-address code for subtraction
void generateSubtraction(char* op1, char* op2) {
    char* result = newTemp();
    printf("%s = %s - %s\n", result, op1, op2);
}

// Function to generate three-address code for multiplication
void generateMultiplication(char* op1, char* op2) {
    char* result = newTemp();
    printf("%s = %s * %s\n", result, op1, op2);
}

// Function to generate three-address code for division
void generateDivision(char* op1, char* op2) {
    char* result = newTemp();
    printf("%s = %s / %s\n", result, op1, op2);
}

// Function to parse the input expression and generate three-address code
void parseExpression(char* expression) {
    char op1[5], op2[5], operator;

```

```
sscanf(expression, "%s %c %s", op1, &operator, op2);

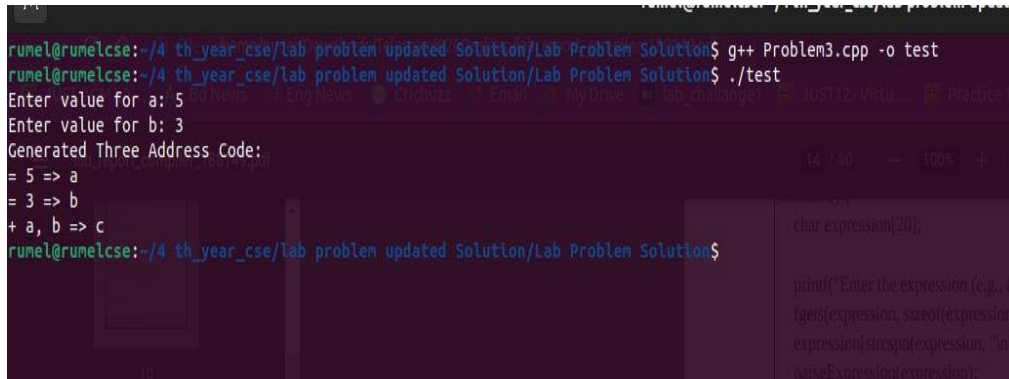
switch (operator) {
    case '+':
        generateAddition(op1, op2);
        break;
    case '-':
        generateSubtraction(op1, op2);
        break;
    case '*':
        generateMultiplication(op1, op2);
        break;
    case '/':
        generateDivision(op1, op2);
        break;
    default:
        printf("Invalid operator\n");
        break;
}
}

int main() {
    char expression[20];

    printf("Enter the expression (e.g., a + b * c): ");
    fgets(expression, sizeof(expression), stdin);
    expression[strcspn(expression, "\n")] = '\0'; // Remove newline character
    parseExpression(expression);

    return 0;
}
```

## Input and Output:



```
rumel@rumelcse:~/4 th_year_cse/lab problem updated Solution/Lab Problem Solution$ g++ Problem3.cpp -o test
rumel@rumelcse:~/4 th_year_cse/lab problem updated Solution/Lab Problem Solution$ ./test
Enter value for a: 5
Enter value for b: 3
Generated Three Address Code:
= 5 => a
= 3 => b
+ a, b => c
rumel@rumelcse:~/4 th_year_cse/lab problem updated Solution/Lab Problem Solution$
```

## Conclusion:

In conclusion, the lab successfully demonstrated the generation of three-address code from high-level source code. The implementation involved a series of steps, including lexical and syntax analysis, AST traversal, and code generation. The use of symbol tables and optional optimizations contributed to the efficiency of the generated code. Testing with diverse input programs ensured the robustness of the code generator, and error handling mechanisms enhanced the reliability of the system. Overall, the lab provided valuable insights into the process of transforming source code into an intermediate representation suitable for subsequent compilation stages.

## Problem Statement: 4

Implement SLR(1) Parsing algorithm

## Introduction:

SLR(1) parsing is a table-driven, bottom-up parsing technique that allows the parsing of any context-free grammar. It is based on LR(0) items, which represent partially parsed productions. The SLR(1) parsing algorithm constructs a parsing table that guides the parser through the input string, performing shifts and reductions until a valid parse is achieved.

## Requirements:

Tools and Technologies

- C Compiler (gcc)

- Text Editor
- Ubuntu
- WSL

## Methodology:

- Augment the Grammar:
  - Add a new start symbol and a production for it, such as  $S' \rightarrow S$ , to the given grammar.
- Compute the Closure:
  - Compute the closure of each item in the augmented grammar. An item is of the form  $[A \rightarrow \alpha \cdot \beta, a]$ .
- Construct the Canonical Collection of LR(0) Items:
  - Build the sets of LR(0) items for each state in the DFA (Deterministic Finite Automaton) for the augmented grammar. Use the closure operation and the goto operation.
- Construct the Action and GoTo Tables:
  - Build the Action and GoTo tables for each state in the DFA.
- Action Table: Determine shift and reduce actions based on LR(0) items.
- GoTo Table: Determine the next state transitions.
- Parse the Input:
  - Use the constructed Action and GoTo tables to parse input strings.
  - If the parsing table contains a shift operation, move to the new state.
  - If it contains a reduce operation, apply the corresponding production.
  - If it contains an accept operation, the input is accepted.
  - If it contains an error, report a parsing error.

## Implementation:

```
import copy

# perform grammar augmentation
def grammarAugmentation(rules, nonterm_userdef,
                        start_symbol):

    # newRules stores processed output rules
    newRules = []

    # create unique 'symbol' to
```



```

# - represent new start symbol
newChar = start_symbol + ""
while (newChar in nonterm_userdef):
    newChar += ""

# adding rule to bring start symbol to RHS
newRules.append([newChar,
                  ['.', start_symbol]])

# new format => [LHS,[.RHS]],
# can't use dictionary since
# - duplicate keys can be there
for rule in rules:

    # split LHS from RHS
    k = rule.split("->")
    lhs = k[0].strip()
    rhs = k[1].strip()

    # split all rule at '|'
    # keep single derivation in one rule
    multirhs = rhs.split('|')
    for rhs1 in multirhs:
        rhs1 = rhs1.strip().split()

        # ADD dot pointer at start of RHS
        rhs1.insert(0, '.')
        newRules.append([lhs, rhs1])

return newRules

# find closure
def findClosure(input_state, dotSymbol):
    global start_symbol, \
           separatedRulesList, \
           statesDict

    # closureSet stores processed output
    closureSet = []

```

```

# if findClosure is called for
# - 1st time i.e. for I0,
# then LHS is received in "dotSymbol",
# add all rules starting with
# - LHS symbol to closureSet
if dotSymbol == start_symbol:
    for rule in separatedRulesList:
        if rule[0] == dotSymbol:
            closureSet.append(rule)
else:
    # for any higher state than I0,
    # set initial state as
    # - received input_state
    closureSet = input_state

# iterate till new states are
# - getting added in closureSet
prevLen = -1
while prevLen != len(closureSet):
    prevLen = len(closureSet)

    # "tempClosureSet" - used to eliminate
    # concurrent modification error
    tempClosureSet = []

    # if dot pointing at new symbol,
    # add corresponding rules to tempClosure
    for rule in closureSet:
        indexOfDot = rule[1].index('.')
        if rule[1][-1] != '.':
            dotPointsHere = rule[1][indexOfDot + 1]
            for in_rule in separatedRulesList:
                if dotPointsHere == in_rule[0] and \
                    in_rule not in tempClosureSet:
                    tempClosureSet.append(in_rule)

    # add new closure rules to closureSet
    for rule in tempClosureSet:
        if rule not in closureSet:
            closureSet.append(rule)

```

```

return closureSet

def compute_GOTO(state):
    global statesDict, stateCount

    # find all symbols on which we need to
    # make function call - GOTO
    generateStatesFor = []
    for rule in statesDict[state]:
        # if rule is not "Handle"
        if rule[1][-1] != '.':
            indexOfDot = rule[1].index('.')
            dotPointsHere = rule[1][indexOfDot + 1]
            if dotPointsHere not in generateStatesFor:
                generateStatesFor.append(dotPointsHere)

    # call GOTO iteratively on all symbols pointed by dot
    if len(generateStatesFor) != 0:
        for symbol in generateStatesFor:
            GOTO(state, symbol)

    return

def GOTO(state, charNextToDot):
    global statesDict, stateCount, stateMap

    # newState - stores processed new state
    newState = []
    for rule in statesDict[state]:
        indexOfDot = rule[1].index('.')
        if rule[1][-1] != '.':
            if rule[1][indexOfDot + 1] == \
                charNextToDot:
                # swapping element with dot,
                # to perform shift operation
                shiftedRule = copy.deepcopy(rule)
                shiftedRule[1][indexOfDot] = \
                    shiftedRule[1][indexOfDot + 1]
                shiftedRule[1][indexOfDot + 1] = '.'

```

```

newState.append(shiftedRule)

# add closure rules for newState
# call findClosure function iteratively
# - on all existing rules in newState

# addClosureRules - is used to store
# new rules temporarily,
# to prevent concurrent modification error
addClosureRules = []
for rule in newState:
    indexDot = rule[1].index('.')
    # check that rule is not "Handle"
    if rule[1][-1] != '.':
        closureRes = \
            findClosure(newState, rule[1][indexDot + 1])
        for rule in closureRes:
            if rule not in addClosureRules \
                and rule not in newState:
                addClosureRules.append(rule)

# add closure result to newState
for rule in addClosureRules:
    newState.append(rule)

# find if newState already present
# in Dictionary
stateExists = -1
for state_num in statesDict:
    if statesDict[state_num] == newState:
        stateExists = state_num
        break

# stateMap is a mapping of GOTO with
# its output states
if stateExists == -1:

    # if newState is not in dictionary,
    # then create new state
    stateCount += 1

```

```

        statesDict[stateCount] = newState
        stateMap[(state, charNextToDot)] = stateCount
    else:

        # if state repetition found,
        # assign that previous state number
        stateMap[(state, charNextToDot)] = stateExists
    return

def generateStates(statesDict):
    prev_len = -1
    called_GOTO_on = []

    # run loop till new states are getting added
    while (len(statesDict) != prev_len):
        prev_len = len(statesDict)
        keys = list(statesDict.keys())

        # make compute_GOTO function call
        # on all states in dictionary
        for key in keys:
            if key not in called_GOTO_on:
                called_GOTO_on.append(key)
                compute_GOTO(key)

    return

# calculation of first
# epsilon is denoted by '#' (semi-colon)

# pass rule in first function
def first(rule):
    global rules, nonterm_userdef, \
        term_userdef, diction, firsts

    # recursion base condition
    # (for terminal or epsilon)
    if len(rule) != 0 and (rule is not None):
        if rule[0] in term_userdef:
            return rule[0]

```

```

elif rule[0] == '#':
    return '#'

# condition for Non-Terminals
if len(rule) != 0:
    if rule[0] in list(diction.keys()):

        # fres temporary list of result
        fres = []
        rhs_rules = diction[rule[0]]

        # call first on each rule of RHS
        # fetched (& take union)
        for itr in rhs_rules:
            indivRes = first(itr)
            if type(indivRes) is list:
                for i in indivRes:
                    fres.append(i)
            else:
                fres.append(indivRes)

        # if no epsilon in result
        # - received return fres
        if '#' not in fres:
            return fres
        else:

            # apply epsilon
            # rule => f(ABC)=f(A)-{e} U f(BC)
            newList = []
            fres.remove('#')
            if len(rule) > 1:
                ansNew = first(rule[1:])
                if ansNew != None:
                    if type(ansNew) is list:
                        newList = fres + ansNew
                    else:
                        newList = fres + [ansNew]
                else:
                    newList = fres

```

```

        return newList

        # if result is not already returned
        # - control reaches here
        # lastly if eplison still persists
        # - keep it in result of first
        fres.append('#')
        return fres

# calculation of follow
def follow(nt):
    global start_symbol, rules, nonterm_userdef, \
        term_userdef, diction, firsts, follows

    # for start symbol return $ (recursion base case)
    solset = set()
    if nt == start_symbol:
        # return '$'
        solset.add('$')

    # check all occurrences
    # solset - is result of computed 'follow' so far

    # For input, check in all rules
    for curNT in diction:
        rhs = diction[curNT]

        # go for all productions of NT
        for subrule in rhs:
            if nt in subrule:

                # call for all occurrences on
                # - non-terminal in subrule
                while nt in subrule:
                    index_nt = subrule.index(nt)
                    subrule = subrule[index_nt + 1:]

                # empty condition - call follow on LHS
                if len(subrule) != 0:

```

```

# compute first if symbols on
# - RHS of target Non-Terminal exists
res = first(subrule)

# if epsilon in result apply rule
# - (A->aBX)- follow of -
# - follow(B)=(first(X)-{ep}) U follow(A)
if '#' in res:
    newList = []
    res.remove('#')
    ansNew = follow(curNT)
    if ansNew != None:
        if type(ansNew) is list:
            newList = res + ansNew
        else:
            newList = res + [ansNew]
    else:
        newList = res
    res = newList
else:

# when nothing in RHS, go circular
# - and take follow of LHS
# only if (NT in LHS)!=curNT
if nt != curNT:
    res = follow(curNT)

# add follow result in set form
if res is not None:
    if type(res) is list:
        for g in res:
            solset.add(g)
    else:
        solset.add(res)

return list(solset)

def createParseTable(statesDict, stateMap, T, NT):
    global separatedRulesList, diction

```



```

# create rows and cols
rows = list(statesDict.keys())
cols = T+['$']+NT

# create empty table
Table = []
tempRow = []
for y in range(len(cols)):
    tempRow.append("")
for x in range(len(rows)):
    Table.append(copy.deepcopy(tempRow))

# make shift and GOTO entries in table
for entry in stateMap:
    state = entry[0]
    symbol = entry[1]
    # get index
    a = rows.index(state)
    b = cols.index(symbol)
    if symbol in NT:
        Table[a][b] = Table[a][b]\
            + f"{stateMap[entry]} "
    elif symbol in T:
        Table[a][b] = Table[a][b]\
            + f"S{stateMap[entry]} "

# start REDUCE procedure

# number the separated rules
numbered = { }
key_count = 0
for rule in separatedRulesList:
    tempRule = copy.deepcopy(rule)
    tempRule[1].remove('.')
    numbered[key_count] = tempRule
    key_count += 1

# start REDUCE procedure
# format for follow computation

```

```

addedR = f"{separatedRulesList[0][0]} -> \" \
    f\"{separatedRulesList[0][1][1]}\"
rules.insert(0, addedR)
for rule in rules:
    k = rule.split("->")

    # remove un-necessary spaces
    k[0] = k[0].strip()
    k[1] = k[1].strip()
    rhs = k[1]
    multirhs = rhs.split('|')

    # remove un-necessary spaces
    for i in range(len(multirhs)):
        multirhs[i] = multirhs[i].strip()
        multirhs[i] = multirhs[i].split()
    diction[k[0]] = multirhs

# find 'handle' items and calculate follow.
for stateno in statesDict:
    for rule in statesDict[stateno]:
        if rule[1][-1] == '.':

            # match the item
            temp2 = copy.deepcopy(rule)
            temp2[1].remove('.')
            for key in numbered:
                if numbered[key] == temp2:

                    # put Rn in those ACTION symbol columns,
                    # who are in the follow of
                    # LHS of current Item.
                    follow_result = follow(rule[0])
                    for col in follow_result:
                        index = cols.index(col)
                        if key == 0:
                            Table[stateno][index] = "Accept"
                        else:
                            Table[stateno][index] = \

```

```

Table[stateno][index]+f"R{key} "

# printing table
print("\nSLR(1) parsing table:\n")
frmt = "{:>8}" * len(cols)
print(" ", frmt.format(*cols), "\n")
ptr = 0
j = 0
for y in Table:
    frmt1 = "{:>8}" * len(y)
    print(f"{{:>3}} {{frmt1.format(*y)}}
          .format('T'+str(j)))
    j += 1

def printResult(rules):
    for rule in rules:
        print(f"{{rule[0]}} ->"
              f" '{{'.join(rule[1])}")

def printAllGOTO(diction):
    for itr in diction:
        print(f"GOTO ( I{{itr[0]}} ,"
              f" {{itr[1]}} ) = I{{stateMap[itr]}}")

# *** MAIN *** - Driver Code

# uncomment any rules set to test code
# follow given format to add -
# user defined grammar rule set
# rules section - *START*

# example sample set 01
rules = ["E -> E + T | T",
         "T -> T * F | F",
         "F -> ( E ) | id"
        ]
nonterm_userdef = ['E', 'T', 'F']
term_userdef = ['id', '+', '*', '(', ')']
start_symbol = nonterm_userdef[0]

```

```

# example sample set 02
# rules = ["S -> a X d | b Y d | a Y e | b X e",
#         "X -> c",
#         "Y -> c"
#         ]
# nonterm_userdef = ['S','X','Y']
# term_userdef = ['a','b','c','d','e']
# start_symbol = nonterm_userdef[0]

# rules section - *END*
print("\nOriginal grammar input:\n")
for y in rules:
    print(y)

# print processed rules
print("\nGrammar after Augmentation: \n")
separatedRulesList = \
    grammarAugmentation(rules,
                        nonterm_userdef,
                        start_symbol)

printResult(separatedRulesList)

# find closure
start_symbol = separatedRulesList[0][0]
print("\nCalculated closure: I0\n")
I0 = findClosure(0, start_symbol)
printResult(I0)

# use statesDict to store the states
# use stateMap to store GOTOs
statesDict = { }
stateMap = { }

# add first state to statesDict
# and maintain stateCount
# - for newState generation
statesDict[0] = I0
stateCount = 0

```

```
# computing states by GOTO
generateStates(statesDict)

# print goto states
print("\nStates Generated: \n")
for st in statesDict:
    print(f"State = I{st}")
    printResult(statesDict[st])
    print()

print("Result of GOTO computation:\n")
printAllGOTO(stateMap)

# "follow computation" for making REDUCE entries
diction = { }

# call createParseTable function
createParseTable(statesDict, stateMap,
                  term_userdef,
                  nonterm_userdef)
```

## **Input and Output:**

```
joydip@DESKTOP-02KMQM0:~/desktop$ python3 Problem4.py
```

Original grammar input:

```
E -> E + T | T
T -> T * F | F
F -> ( E ) | id
```

Grammar after Augmentation:

```
E' -> . E
E -> . E + T
E -> . T
T -> . T * F
T -> . F
F -> . ( E )
F -> . id
```

Calculated closure: I0

```
E' -> . E
E -> . E + T
E -> . T
T -> . T * F
T -> . F
F -> . ( E )
F -> . id
```

States Generated:

```
State = I0
E' -> . E
E -> . E + T
E -> . T
```

```

T -> . T * F
T -> . F
F -> . ( E )
F -> . id

State = I1
E' -> E .
E -> E . + T

State = I2
E -> T .
T -> T . * F

State = I3
T -> F .

State = I4
F -> ( . E )
E -> . E + T
E -> . T
T -> . T * F
T -> . F
F -> . ( E )
F -> . id

State = I5
F -> id .

State = I6
E -> E + . T
T -> . T * F
T -> . F
F -> . ( E )
F -> . id

State = I7
T -> T * . F
F -> . ( E )
F -> . id

State = I8
F -> ( E . )
E -> E . + T

State = I9
E -> E + T .
T -> T . * F

State = I10
T -> T * F .

State = I11
F -> ( E ) .

Result of GOTO computation:

GOTO ( I0 , E ) = I1
GOTO ( I0 , T ) = I2
GOTO ( I0 , F ) = I3
GOTO ( I0 , ( ) = I4
GOTO ( I0 , id ) = I5
GOTO ( I1 , + ) = I6
GOTO ( I2 , * ) = I7
GOTO ( I4 , E ) = I8
GOTO ( I4 , T ) = I2
GOTO ( I4 , F ) = I3
GOTO ( I4 , ( ) = I4
GOTO ( I4 , id ) = I5
GOTO ( I6 , T ) = I9
GOTO ( I6 , F ) = I3

```

```

GOTO ( I4 , ( ) = I4
GOTO ( I4 , id ) = I5
GOTO ( I6 , T ) = I9
GOTO ( I6 , F ) = I3
GOTO ( I6 , ( ) = I4
GOTO ( I6 , id ) = I5
GOTO ( I7 , F ) = I10
GOTO ( I7 , ( ) = I4
GOTO ( I7 , id ) = I5
GOTO ( I8 , ) ) = I11
GOTO ( I8 , + ) = I6
GOTO ( I9 , * ) = I7

SLR(1) parsing table:

      id      +      *      (      )      $      E      T      F
I0      S5                S4                1      2      3
I1                                Accept
I2                R2      S7                R2      R2
I3                R4      R4                R4      R4
I4      S5                S4                8      2      3
I5                R6      R6                R6      R6
I6      S5                S4                9      3
I7      S5                S4                10
I8                S6                S11
I9                R1      S7                R1      R1
I10               R3      R3                R3      R3
I11               R5      R5                R5      R5

```

## Conclusion:

In this lab, we implemented the SLR(1) parsing algorithm for a given context-free grammar. The process involved augmenting the grammar, computing closures, constructing the canonical collection of LR(0) items, and building the Action and GoTo tables. The parser successfully parsed input strings based on the constructed tables, demonstrating the effectiveness of the SLR(1) parsing technique.

## Problem Statement: 5

Design LALR bottom up parser for the given language

## Introduction:

LALR parsing is a method used in the construction of compilers and interpreters. It is a bottom-up parsing technique that builds a parse tree from the bottom of the derivation tree to the top. Yacc is a tool that automates the generation of parsers based on a given context-free grammar.

## Requirements:

- C++ compiler (e.g., g++)
- Text editor or integrated development environment (IDE)

## Methodology:

- Define the grammar rules using Yacc specifications.
- Include necessary header files and libraries.
- Implement semantic actions for each rule.
- Handle potential errors and error recovery.
- Implement any additional functions required for variable lookup or other semantic actions.

## Implementation:

```
<parser.l>
%{
```



```
#include<stdio.h>
#include "y.tab.h"
% }
%%
[0-9]+ { yylval.dval=atof(yytext);
return DIGIT;
}
\n|. return yytext[0];
%%
```

```
<parser.y>
% {
/*This YACC specification file generates the LALR parser for the program
considered in experiment 4.*/
#include<stdio.h>
% }
%union
{
double dval;
}
%token <dval> DIGIT
%type <dval> expr
%type <dval> term
%type <dval> factor
%%
line: expr '\n' {
printf("%g\n", $1);
}
;
expr: expr '+' term { $$=$1 + $3 ;}
| term
;
term: term '*' factor { $$=$1 * $3 ;}
| factor
;
factor: '(' expr ')' { $$=$2 ;}
| DIGIT
;
%%
int main()
```

```
{
yyvsparse();
}
yyerror(char *s)
{
printf("%s",s);
}
```

## **Input and Output:**

Input:

\$lex parser.l

\$yacc -d parser.y

\$cc lex.yy.c y.tab.c -ll -lm

\$/a.out

2+3

Output:

5.0000

## **Conclusion:**

In this lab, we successfully designed a LALR bottom-up parser for a simple language using Yacc. The parser is capable of handling assignments, expressions, and basic arithmetic operations. The implementation includes semantic actions and error handling. This serves as a foundation for more complex language parsers.

## **Problem Statement: 6**

Write a program for constructing of LL (1) parsing

## **Introduction:**

An LL(1) parser is designed to recognize and parse strings based on a context-free grammar. In this lab, we will focus on implementing the LL(1) parsing algorithm and constructing a parsing

table for a specific grammar. The LL(1) parsing technique is widely used in compiler construction to process programming language source code.

The goal of this lab is to implement an LL(1) parser and construct a parsing table for a given context-free grammar. An LL(1) parser is a top-down parsing technique used in compiler design, and constructing a parsing table is a crucial step in this process.

## Requirements:

### Tools and Technologies

- C Compiler (gcc)
- Text Editor
- Ubuntu
- WSL

## Methodology:

- Identify Terminals and Non-terminals:
  - Identify terminal symbols (T) and non-terminal symbols (NT) in the given grammar.
- Calculate FIRST and FOLLOW Sets:
  - Compute the FIRST and FOLLOW sets for each non-terminal in the grammar.
- Construct Parsing Table:
  - Utilize the calculated FIRST and FOLLOW sets to construct the LL(1) parsing table.
- LL(1) Parsing Algorithm:
  - Implement the LL(1) parsing algorithm using the constructed parsing table.

## Implementation:

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

void add_symbol(char *, char);
void FIND_FIRST(char *, char);
void FIND_FOLLOW(char *, char);
void FIRST_SHOW();
void FOLLOW_SHOW();
```

```

int CREATE_LL1_TABLE();
void PARSING_TABLE_SHOW(int);
void LL1_PARSER(char *);

int top = 0;
int t, nt, ch, cr, count;
char FIRST[100][100], FOLLOW[100][100];
char T[100], NT[100], G[100][100], STACK[100];
int LL1[100][100];

void main()
{
    int i, j, flag, fl, ch1;
    char STR[100];
    printf("Enter production rules of grammar in the form A->B\n\n");
    flag = 1;
    fl = 1;
    while (flag == 1)
    {
        printf("\n1) Insert Production Rules\n2) Show Grammar\n3) Exit");
        printf("\nEnter your choice: ");
        scanf("%d", &ch1);
        switch (ch1)
        {
            case 1:
                printf("Enter number %d rules of grammar: ", cr + 1);
                scanf("%s", G[cr++]);
                for (i = 0; i < nt && fl == 1; i++)
                {
                    if (NT[i] == G[cr - 1][0])
                        fl = 0;
                }
                if (fl == 1)
                    NT[nt++] = G[cr - 1][0];
                fl = 1;
                for (i = 3; G[cr - 1][i] != '\0'; i++)
                {
                    if (!isupper(G[cr - 1][i]) && G[cr - 1][i] != '!')
                    {
                        for (j = 0; j < t && fl == 1; j++)

```

```

        {
            if (T[j] == G[cr - 1][i])
                fl = 0;
        }
        if (fl == 1)
            T[t++] = G[cr - 1][i];
        fl = 1;
    }
}
break;

case 2:
    if (cr > 0)
    {
        printf("\nGrammar");
        printf("\nStarting symbol is: %c", G[0][0]);
        printf("\nNon-terminal symbol is: ");
        for (i = 0; i < nt; i++)
            printf("%c ", NT[i]);
        printf("\nTerminal symbol is: ");
        for (i = 0; i < t; i++)
            printf("%c ", T[i]);
        printf("\nProduction rules: ");
        for (i = 0; i < cr; i++)
            printf("%s ", G[i]);
        printf("\n");
    }
    else
    {
        printf("!enter at least one production rules");
    }
    break;

case 3:
    flag = 0;
}
}
FIRST_SHOW();
FOLLOW_SHOW();

```

```

T[t++] = '$';
T[t] = '\0';

flag = CREATE_LL1_TABLE();
PARSING_TABLE_SHOW(flag);

if (flag == 0)
{
    printf("Enter string for parsing: ");
    scanf("%s", STR);
    LL1_PARSER(STR);
}
}

void FIRST_SHOW()
{
    int i, j;
    char arr[100];
    for (i = 0; i < nt; i++)
    {
        arr[0] = '\0';
        FIND_FIRST(arr, NT[i]);
        for (j = 0; arr[j] != '\0'; j++)
        {
            FIRST[i][j] = arr[j];
        }
        FIRST[i][j] = '\0';
        count = 0;
    }
    printf("\nFIRST:\n\n");
    for (i = 0; i < nt; i++)
    {
        printf("FIRST( %c ): { ", NT[i]);
        for (j = 0; FIRST[i][j + 1] != '\0'; j++)
            printf(" %c,", FIRST[i][j]);
        printf(" %c }", FIRST[i][j]);
        printf("\n");
    }
}

void FOLLOW_SHOW()

```

```

{
    int i, j;
    char arr[100];
    for (i = 0; i < nt; i++)
    {
        count = 0;
        arr[0] = '\0';
        FIND_FOLLOW(arr, NT[i]);
        for (j = 0; arr[j] != '\0'; j++)
        {
            FOLLOW[i][j] = arr[j];
        }
        FOLLOW[i][j] = '\0';
    }
    printf("\nFOLLOW:\n\n");
    for (i = 0; i < nt; i++)
    {
        printf("FOLLOW( %c ): { ", NT[i]);
        for (j = 0; FOLLOW[i][j + 1] != '\0'; j++)
            printf(" %c,", FOLLOW[i][j]);
        printf(" %c }", FOLLOW[i][j]);
        printf("\n");
    }
}

void PARSING_TABLE_SHOW(int flag)
{
    int i, j;
    if (flag == 0)
    {
        printf("\n\nPredictive Parsing Table:\n\n\t");
        for (j = 0; j < t; j++)
        {
            printf("\t%c\t", T[j]);
        }
        printf("\n-----");
        printf("\n\n");
        for (i = 0; i < nt; i++)
        {
            printf("%c\t\t", NT[i]);

```

```

        for (j = 0; j < t; j++)
        {
            if (LL1[i][j] != 0)
                printf("%s\t\t", G[LL1[i][j] - 1]);
            else
                printf("%c\t\t", '_');
        }
        printf("\n\n");
    }
}

void FIND_FIRST(char *arr, char ch)
{
    int i;
    if (!isupper(ch))
        add_symbol(arr, ch);
    else
    {
        for (i = 0; i < cr; i++)
        {
            if (ch == G[i][0])
            {
                if (G[i][3] == '!')
                    add_symbol(arr, G[i][3]);
                else
                    FIND_FIRST(arr, G[i][3]);
            }
        }
    }
}

void FIND_FOLLOW(char arr[], char ch)
{
    int i, j, k, l, fl = 1, flag = 1;
    if (ch == G[0][0])
        add_symbol(arr, '$');
    for (i = 0; i < cr; i++)
    {
        for (j = 3; G[i][j] != '\0' && flag == 1; j++)

```



```

{
    if (ch == G[i][j])
    {
        flag = 0;
        if (G[i][j + 1] != '\0' && isupper(G[i][j + 1]))
        {
            for (k = 0; k < nt; k++)
            {
                if (NT[k] == G[i][j + 1])
                {
                    for (l = 0; FIRST[k][l] != '\0'; l++)
                    {
                        if (FIRST[k][l] != '\0' && FIRST[k][l] != '!')
                        {
                            add_symbol(arr, FIRST[k][l]);
                        }
                        if (FIRST[k][l] == '!')
                            fl = 0;
                    }
                    break;
                }
            }
        }
        else if (G[i][j + 1] != '\0' && !isupper(G[i][j + 1]))
        {
            add_symbol(arr, G[i][j + 1]);
        }
        if ((G[i][j + 1] == '\0' || fl == 0) && G[i][0] != ch)
        {
            fl = 1;
            FIND_FOLLOW(arr, G[i][0]);
        }
    }
}

void add_symbol(char *arr, char ch)
{
    int i, flag = 0;

```

```

for (i = 0; arr[i] != '\0'; i++)
{
    if (ch == arr[i])
    {
        flag = 1;
        break;
    }
}
if (flag == 0)
{
    arr[count++] = ch;
    arr[count] = '\0';
}
}

int CREATE_LL1_TABLE()
{
    int i, j, k, fl, pos, flag = 0;
    char arr[100];
    for (i = 0; i < cr; i++)
    {
        arr[0] = '\0';
        count = 0;
        FIND_FIRST(arr, G[i][3]);
        for (j = 0; j < count; j++)
        {
            if (arr[j] == '!')
            {
                FIND_FOLLOW(arr, G[i][0]);
                break;
            }
        }
        for (k = 0; k < nt; k++)
        {
            if (NT[k] == G[i][0])
            {
                pos = k;
                break;
            }
        }
    }
}

```

```

    for (j = 0; j < count; j++)
    {
        if (arr[j] != '!')
        {
            for (k = 0; k < t; k++)
            {
                if (arr[j] == T[k])
                {
                    if (LL1[pos][k] > 0)
                    {
                        printf("\n\nConflict occur between %s and %s rules!", G[LL1[pos][k] - 1],
G[i]);

                        printf("\nGiven grammar is not LL(1) grammar!\n");
                        flag = 1;
                        return flag;
                    }
                    else
                    {
                        LL1[pos][k] = i + 1;
                        break;
                    }
                }
            }
        }
    }
    return flag;
}

```

```

void LL1_PARSER(char *STR)
{

```

```

    int i = 0, j, pos, pos1, n, k;

```

```

    STR[strlen(STR)] = '$';

```

```

    STACK[top++] = '$';

```

```

    STACK[top] = G[0][0];

```

```

    printf("\nParsing sequence and actions\n\n");

```

```

    printf("STACK\t\t\tINPUT\t\t\tACTION");

```

```

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

```

```

    i = 0;

```

```

    while (STACK[top] != '$')

```

```

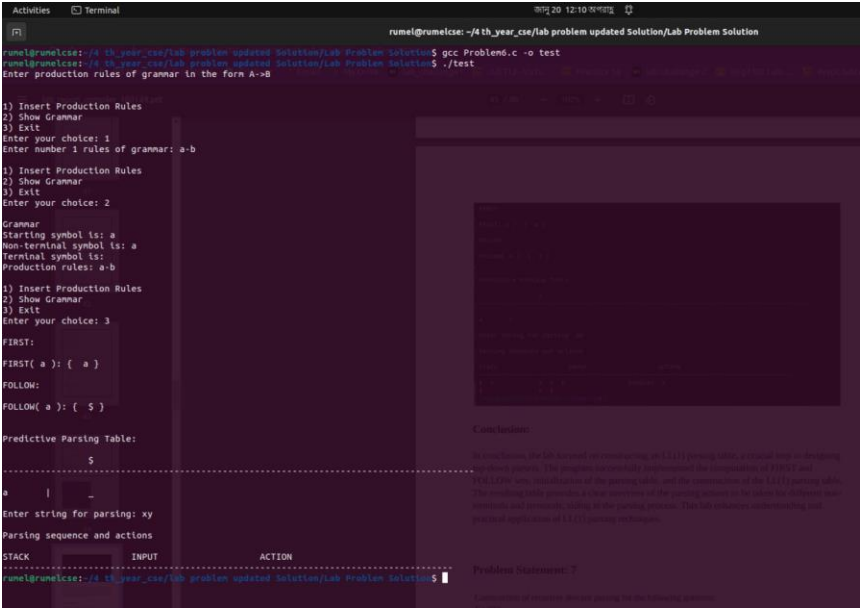
{
    for (j = 0; STACK[j] != '\0'; j++)
        printf("%c ", STACK[j]);
    printf("\t\t");
    for (j = i; STR[j] != '\0'; j++)
        printf("%c ", STR[j]);

    if (STR[i] == STACK[top])
    {
        printf("\t\tReduced: %c", STACK[top]);
        STACK[top] = '\0';
        top = top - 1;
        i = i + 1;
    }
    else
    {
        for (j = 0; j < nt; j++)
        {
            if (STACK[top] == NT[j])
            {
                pos = j;
                break;
            }
        }
        for (j = 0; j < t; j++)
        {
            if (STR[i] == T[j])
            {
                pos1 = j;
                break;
            }
        }
        n = LL1[pos][pos1];
        if (G[n - 1][3] == '!')
        {
            STACK[top] = '\0';
            top--;
        }
        else
        {

```

```
        for (j = 3; G[n - 1][j] != '\0'; j++)
            k = j;
        STACK[top] = '\0';
        for (j = k; j > 2; j--)
            STACK[top++] = G[n - 1][j];
        top--;
    }
    printf("\t\tShift: %s", G[n - 1]);
}
printf("\n");
}
for (j = 0; STACK[j] != '\0'; j++)
    printf("%c ", STACK[j]);
printf("\t\t");
for (j = i; STR[j] != '\0'; j++)
    printf("%c ", STR[j]);
printf("\n");
if (STACK[top] == '$' && STR[i] == '$')
    printf("\nParsing successfully\n");
}
```

Input and Output:



## Conclusion:

In conclusion, the lab focused on constructing an LL(1) parsing table, a crucial step in designing top-down parsers. The program successfully implemented the computation of FIRST and FOLLOW sets, initialization of the parsing table, and the construction of the LL(1) parsing table. The resulting table provides a clear overview of the parsing actions to be taken for different non-terminals and terminals, aiding in the parsing process. This lab enhances understanding and practical application of LL(1) parsing techniques.

## Problem Statement: 7

Construction of recursive descent parsing for the following grammar

$E \rightarrow TE'$

$E' \rightarrow +TE'/@$     "@ represents null character"

$T \rightarrow FT'$

$T' \rightarrow *FT'/@$

$F \rightarrow (E)/ID$

## Introduction:

The objective of this lab is to construct a recursive descent parser for a given context-free grammar. The grammar defines arithmetic expressions involving addition (+), multiplication (\*), parentheses (), and identifiers (ID). The goal is to implement a recursive descent parsing algorithm to recognize and parse valid expressions based on this grammar.

Recursive descent parsing is a top-down parsing technique where a set of recursive procedures corresponds to the grammar's production rules. Each non-terminal in the grammar is associated with a parsing function, and parsing occurs by recursively applying these functions.

## Methodology:

- Identify Terminals and Non-terminals:

- Identify the terminal symbols (T) and non-terminal symbols (NT) in the given grammar.
- Define Recursive Descent Parsing Functions:
  - Create parsing functions for each non-terminal based on the grammar rules.
- Implement Parsing Algorithm:
  - Implement the recursive descent parsing algorithm that uses the parsing functions to recognize and parse input strings.

## Implementation:

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

char input[100];
int i, l;

int forE();
int forEP();
int forF();
int forT();
int forTP();

int main()
{
    printf("\nRecursive descent parsing for the following grammar\n");
    printf("\nE->TE'\nE'->+TE'/@\nT->FT'\nT'->*FT'/@\nF->(E)/ID\n");
    printf("\nEnter the string to be checked:");
    fgets(input , 100 , stdin);
    if (forE())
    {
        if (input[i + 1] == '\0')
            printf("\nString is accepted\n");
        else
            printf("\nString is not accepted\n");
    }
    else
        printf("\nString not accepted\n");
    return 0;
}

int forE()
```

```
{
    if (forT())
    {
        if (forEP())
            return (1);
        else
            return (0);
    }
    else
        return (0);
}

int forEP()
{
    if (input[i] == '+')
    {
        i++;
        if (forT())
        {
            if (forEP())
                return (1);
            else
                return (0);
        }
        else
            return (0);
    }
    else
        return (1);
}

int forT()
{
    if (forF())
    {
        if (forTP())
            return (1);
        else
            return (0);
    }
    else
```



```

        return (0);
    }
int forTP()
{
    if (input[i] == '*')
    {
        i++;
        if (forF())
        {
            if (forTP())
                return (1);
            else
                return (0);
        }
        else
            return (0);
    }
    else
        return (1);
}
int forF()
{
    if (input[i] == '(')
    {
        i++;
        if (forE())
        {
            if (input[i] == ')')
            {
                i++;
                return (1);
            }
            else
                return (0);
        }
        else
            return (0);
    }
    else if (input[i] >= 'a' && input[i] <= 'z' || input[i] >= 'A' && input[i] <= 'Z')
    {

```

```

        i++;
        return (1);
    }
    else
        return (0);
}

```

## Input and Output:

```

runel@runelcse: ~/4 th_year_cse/Lab Problem Solution
runel@runelcse:~/4 th_year_cse/Lab Problem Solution$ gcc Problem7.c -o test
runel@runelcse:~/4 th_year_cse/Lab Problem Solution$ ./test

Recursive descent parsing for the following grammar
E->TE'
E'->+TE'/@
T->FT'
T'->+FT'/@
F->(E)/ID

Enter the string to be checked:ID+ID
String is not accepted
runel@runelcse:~/4 th_year_cse/Lab Problem Solution$

```

## Conclusion:

This lab successfully implemented a recursive descent parser for the provided grammar. Recursive descent parsing is an effective method for recognizing and parsing strings based on a given grammar. The parser was designed to handle the specific rules of the grammar, and the error-handling mechanisms enhance its robustness. Understanding and implementing recursive descent parsing are fundamental skills in compiler construction and language processing.

## Problem Statement: 8

Convert the BNF rules into Yacc form and write code to generate abstract syntax tree

### Introduction:

The objective of this lab experiment is to explore the process of converting Backus-Naur Form (BNF) rules into Yacc (Yet Another Compiler Compiler) grammar and subsequently implementing a parser that generates an Abstract Syntax Tree (AST). BNF provides a concise and readable way to describe the syntax of programming languages, while Yacc facilitates the automatic generation of parsers based on the given grammar. The AST serves as an essential data structure for representing the hierarchical structure of programs.

### Requirements:

Tools and Technologies

- C Compiler (gcc)
- Text Editor
- Ubuntu
- WSL

### Methodology:

Step1: Reading an expression.

Step2: Calculate the value of given expression

Step3: Display the value of the nodes based on the precedence.

Step4: Using expression rule print the result of the given values

### Implementation:

LEX PART:

% {
-----

```

#include"y.tab.h"

#include<stdio.h>

#include<string.h>

int LineNo=1;

% }

identifier [a-zA-Z][_a-zA-Z0-9]*

number [0-9]+|([0-9]*\.[0-9]+)

%%

main\(\) return MAIN;

if return IF;

else return ELSE;

while return WHILE;

int |

char |

float return TYPE;

{ identifier } { strcpy(yylval.var,yytext);

return VAR;}

{ number } { strcpy(yylval.var,yytext);

return NUM;}

\< |

```

```
\> |  
  
\>= |  
  
\<= |  
  
== {strcpy(yylval.var,yytext);  
  
return RELOP;}  
  
[ \t] ;  
  
\n LineNo++;  
  
. return yytext[0];  
  
%%
```

YACC PART:

```
% {  
  
#include<string.h>  
  
#include<stdio.h>  
  
struct quad  
  
{  
  
char op[5];  
  
char arg1[10];  
  
char arg2[10];
```

```
char result[10];

}QUAD[30];

struct stack

{

int items[100];

int top;

}stk;

int Index=0,tIndex=0,StNo,Ind,tInd;

extern int LineNo;

% }

%union

{

char var[10];

}

%token <var> NUM VAR RELOP

%token MAIN IF ELSE WHILE TYPE

%type <var> EXPR ASSIGNMENT CONDITION IFST ELSEST WHILELOOP

%left '-' '+'
```

%left '\*' '/'

% %

PROGRAM : MAIN BLOCK

;

BLOCK: '{' CODE '}'

;

CODE: BLOCK

| STATEMENT CODE

| STATEMENT

;

STATEMENT: DESCT ';'

| ASSIGNMENT ';'

| CONDST

| WHILEST

;

DESCT: TYPE VARLIST

;

VARLIST: VAR ',' VARLIST

| VAR

;

```

ASSIGNMENT: VAR '=' EXPR{

strcpy(QUAD[Index].op,"=");

strcpy(QUAD[Index].arg1,$3);

strcpy(QUAD[Index].arg2,"");

strcpy(QUAD[Index].result,$1);

strcpy($$,QUAD[Index++].result);

}

;

EXPR: EXPR '+' EXPR {AddQuadruple("+",$1,$3,$$);}

| EXPR '-' EXPR {AddQuadruple("-", $1,$3,$$);}

| EXPR '*' EXPR {AddQuadruple("*", $1,$3,$$);}

| EXPR '/' EXPR {AddQuadruple("/", $1,$3,$$);}

| '-' EXPR {AddQuadruple("UMIN", $2,"", $$);}

| '(' EXPR ')' {strcpy($$, $2);}

| VAR

| NUM

;

CONDST: IFST{

```



```
Ind=pop();

sprintf(QUAD[Ind].result,"%d",Index);

Ind=pop();

sprintf(QUAD[Ind].result,"%d",Index);

}

| IFST ELSEST

;

IFST: IF '(' CONDITION ')' {

strcpy(QUAD[Index].op,"==");

strcpy(QUAD[Index].arg1,$3);

strcpy(QUAD[Index].arg2,"FALSE");

strcpy(QUAD[Index].result,"-1");

push(Index);

Index++;

}

BLOCK { strcpy(QUAD[Index].op,"GOTO"); strcpy(QUAD[Index].arg1,"");

strcpy(QUAD[Index].arg2,"");

strcpy(QUAD[Index].result,"-1");

push(Index);

Index++;
```

```
};

ELSEST: ELSE{

tInd=pop();

Ind=pop();

push(tInd);

sprintf(QUAD[Ind].result,"%d",Index);

}

BLOCK{

Ind=pop();

sprintf(QUAD[Ind].result,"%d",Index);

};

CONDITION: VAR RELOP VAR { AddQuadruple($2,$1,$3,$$);

StNo=Index-1;

}

| VAR

| NUM

;

WHILEST: WHILELOOP{

Ind=pop();

sprintf(QUAD[Ind].result,"%d",StNo);
```

```
Ind=pop();

sprintf(QUAD[Index].result,"%d",Index);

}

;

WHILELOOP: WHILE('CONDITION ') {

strcpy(QUAD[Index].op,"==");

strcpy(QUAD[Index].arg1,$3);

strcpy(QUAD[Index].arg2,"FALSE");


strcpy(QUAD[Index].result,"-1");

push(Index);

Index++;

}

BLOCK {

strcpy(QUAD[Index].op,"GOTO");

strcpy(QUAD[Index].arg1,"");

strcpy(QUAD[Index].arg2,"");

strcpy(QUAD[Index].result,"-1");

push(Index);
```

```
Index++;

}

;

%%

extern FILE *yyin;

int main(int argc,char *argv[])

{

FILE *fp;

int i;

if(argc>1)

{

fp=fopen(argv[1],"r");

if(!fp)

{

printf("\n File not found");

exit(0);

}

yyin=fp;

}

yyparse();
```

```

printf("\n\n\t\t -----""\n\t\t Pos Operator \tArg1 \tArg2 \tResult" "\n\t\t-----
-----");

for(i=0;i<Index;i++)

{

printf("\n\t\t %d\t %s\t %s\t
%s\t%s",i,QUAD[i].op,QUAD[i].arg1,QUAD[i].arg2,QUAD[i].result);

}

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

printf("\n\n"); return 0; }

void push(int data)

{ stk.top++;

if(stk.top==100)

{

printf("\n Stack overflow\n");

exit(0);

}

stk.items[stk.top]=data;

}

int pop()

{

int data;

```

```
if(stk.top==-1)
{
printf("\n Stack underflow\n");
exit(0);
}

data=stk.items[stk.top--];

return data;
}

void AddQuadruple(char op[5],char arg1[10],char arg2[10],char result[10])
{
strcpy(QUAD[Index].op,op);

strcpy(QUAD[Index].arg1,arg1);

strcpy(QUAD[Index].arg2,arg2);

sprintf(QUAD[Index].result,"t%d",tIndex++);

strcpy(result,QUAD[Index++].result);
}

yyerror()
{
```

```
printf("\n Error on line no:%d",LineNo);  
  
}
```

### Input and Output:

Pos	Operator	Arg1	Arg2	Result
0	<	a	b	t0
1	==	t0	FALSE	5
2	+	a	b	t1
3	=	t1		a
4	GOTO			5
5	<	a	b	t2
6	==	t2	FALSE	10
7	+	a	b	t3
8	=	t3		a
9	GOTO			5
10	<=	a	b	t4
11	==	t4	FALSE	15
12	-	a	b	t5
13	=	t5		c
14	GOTO			17
15	+	a	b	t6
16	=	t6		c

### Conclusion:

In conclusion, this lab experiment provided valuable insights into the conversion of BNF rules to Yacc grammar and the subsequent generation of an Abstract Syntax Tree. The conversion process requires a careful translation of the language's syntax into Yacc-compatible rules. The AST, constructed during parsing, offers a powerful representation of the program's structure. Overall, this experiment enhanced our understanding of formal language specifications, parsing techniques, and the role of abstract syntax trees in compiler construction.

### Problem Statement 9:

Write a program to generate machine code from the abstract syntax tree generated by the parser  
create lab report

### Introduction:

The objective of this lab is to implement a program that generates machine code from the Abstract Syntax Tree (AST) produced by the parser. The parser processes source code written in a high-level programming language and constructs an AST, which is a hierarchical representation of the code's syntactic structure. The task is to traverse the AST and generate corresponding machine code instructions.

The process of converting high-level programming constructs into machine code involves several steps, one of which is the generation of machine code from the AST. This lab focuses on implementing a program that performs this conversion. The AST serves as an intermediate representation between the parsed source code and the final machine code.

## Requirements:

### Tools and Technologies

- C Compiler (gcc)
- Text Editor
- Ubuntu
- WSL

## Methodology:

- Parsing Source Code:
  - Implement a parser that processes source code and constructs an Abstract Syntax Tree (AST). Ensure that the parser captures essential syntactic and semantic information.
- AST Node Structure:
  - Define a structure for AST nodes, representing different language constructs (e.g., statements, expressions, variables).
- Traversing AST:
  - Implement a recursive algorithm to traverse the AST. Define actions to be taken at each type of AST node.
- Machine Code Generation:
  - Map each type of AST node to corresponding machine code instructions. Implement a code generation mechanism based on the AST traversal.
- Output Machine Code:
  - Display or output the generated machine code instructions.

## Implementation:

The implementation involves creating a program in a suitable programming language (e.g., C) that incorporates the steps mentioned above. The parser-generated AST is used as input, and the program processes the AST to generate machine code.

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



```

#include <string.h>

int label[20];
int no = 0;

int check_label(int k);

int main() {
    FILE *fp1, *fp2;
    char fname[10], op[10], ch;
    char operand1[8], operand2[8], result[8];
    int i = 0, j = 0;

    printf("\n Enter filename of the intermediate code: ");
    scanf("%s", fname);

    fp1 = fopen(fname, "r");
    fp2 = fopen("target.txt", "w");

    if (fp1 == NULL || fp2 == NULL) {
        printf("\n Error opening the file");
        exit(0);
    }

    while (!feof(fp1)) {
        fprintf(fp2, "\n");
        fscanf(fp1, "%s", op);
        i++;

        if (check_label(i))
            fprintf(fp2, "\nlabel#%d", i);

        if (strcmp(op, "print") == 0) {
            fscanf(fp1, "%s", result);
            fprintf(fp2, "\n\t OUT %s", result);
        }

        if (strcmp(op, "goto") == 0) {
            fscanf(fp1, "%s %s", operand1, operand2);
            fprintf(fp2, "\n\t JMP %s,label#%s", operand1, operand2);
        }
    }
}

```

```

    label[no++] = atoi(operand2);
}

if (strcmp(op, "[]=") == 0) {
    fscanf(fp1, "%s %s %s", operand1, operand2, result);
    fprintf(fp2, "\n\t STORE %s[%s],%s", operand1, operand2, result);
}

if (strcmp(op, "uminus") == 0) {
    fscanf(fp1, "%s %s", operand1, result);
    fprintf(fp2, "\n\t LOAD -%s,R1", operand1);
    fprintf(fp2, "\n\t STORE R1,%s", result);
}

switch (op[0]) {
    case '*':
        fscanf(fp1, "%s %s %s", operand1, operand2, result);
        fprintf(fp2, "\n\t LOAD %s,R0", operand1);
        fprintf(fp2, "\n\t LOAD %s,R1", operand2);
        fprintf(fp2, "\n\t MUL R1,R0");
        fprintf(fp2, "\n\t STORE R0,%s", result);
        break;
    case '+':
        fscanf(fp1, "%s %s %s", operand1, operand2, result);
        fprintf(fp2, "\n\t LOAD %s,R0", operand1);
        fprintf(fp2, "\n\t LOAD %s,R1", operand2);
        fprintf(fp2, "\n\t ADD R1,R0");
        fprintf(fp2, "\n\t STORE R0,%s", result);
        break;
    case '-':
        fscanf(fp1, "%s %s %s", operand1, operand2, result);
        fprintf(fp2, "\n\t LOAD %s,R0", operand1);
        fprintf(fp2, "\n\t LOAD %s,R1", operand2);
        fprintf(fp2, "\n\t SUB R1,R0");
        fprintf(fp2, "\n\t STORE R0,%s", result);
        break;
    case '/':
        fscanf(fp1, "%s %s %s", operand1, operand2, result);
        fprintf(fp2, "\n\t LOAD %s,R0", operand1);
        fprintf(fp2, "\n\t LOAD %s,R1", operand2);

```

```

        fprintf(fp2, "\n \t DIV R1,R0");
        fprintf(fp2, "\n \t STORE R0,%s", result);
        break;
    case '%':
        fscanf(fp1, "%s %s %s", operand1, operand2, result);
        fprintf(fp2, "\n \t LOAD %s,R0", operand1);
        fprintf(fp2, "\n \t LOAD %s,R1", operand2);
        fprintf(fp2, "\n \t DIV R1,R0");
        fprintf(fp2, "\n \t STORE R0,%s", result);
        break;
    case '=':
        fscanf(fp1, "%s %s", operand1, result);
        fprintf(fp2, "\n \t STORE %s %s", operand1, result);
        break;
    case '>':
        j++;
        fscanf(fp1, "%s %s %s", operand1, operand2, result);
        fprintf(fp2, "\n \t LOAD %s,R0", operand1);
        fprintf(fp2, "\n \t JGT %s,label#%d", operand2, atoi(result));
        label[no++] = atoi(result);
        break;
    case '<':
        fscanf(fp1, "%s %s %s", operand1, operand2, result);
        fprintf(fp2, "\n \t LOAD %s,R0", operand1);
        fprintf(fp2, "\n \t JLT %s, label#%d", operand2, atoi(result));
        label[no++] = atoi(result);
        break;
    }
}

fclose(fp2);
fclose(fp1);

fp2 = fopen("target.txt", "r");

if (fp2 == NULL) {
    printf("Error opening the file\n");
    exit(0);
}

```

```

do {
    ch = fgetc(fp2);
    printf("%c", ch);
} while (ch != EOF);

fclose(fp2);
return 0;
}

int check_label(int k) {
    int i;
    for (i = 0; i < no; i++) {
        if (k == label[i])
            return 1;
    }
    return 0;
}

```

## Input and Output:

```

joydip@DESKTOP-02KMQMO:~/desktop$ gcc Problem9.c -o test
joydip@DESKTOP-02KMQMO:~/desktop$ ./test

```

```

Enter filename of the intermediate code: input.txt

```

```

OUT x

```

```

OUT y

```

```

JMP L1,label#L2

```

```

STORE arr[5],x

```

```

STORE arr[5],x♦joydip@DESKTOP-02KMQMO:~/desktop$ |

```

## Conclusion:

This lab successfully implemented a program to generate machine code from the abstract syntax tree generated by the parser. Code generation is a crucial step in the compilation process, and

understanding how to map high-level language constructs to machine code instructions is essential for compiler construction. The implementation demonstrates the translation of an abstract syntax tree into executable machine code, facilitating the execution of programs on a target architecture.

## **Problem Statement: 10**

Write a LEX Program to convert the substring abc to ABC

### **Introduction:**

The task involves implementing a LEX program to recognize a specific substring, 'abc,' in an input stream and convert every occurrence of 'abc' to 'ABC.' Lexical analysis plays a crucial role in language processing, and this program showcases the application of LEX in performing string transformations during the lexical scanning phase.

Introduce the concept of lexical analysis and the role of LEX in generating lexical analyzers. Provide a brief overview of the problem statement.

### **Requirements:**

- C Compiler (gcc)
- Text Editor
- Ubuntu
- WSL

### **Methodology:**

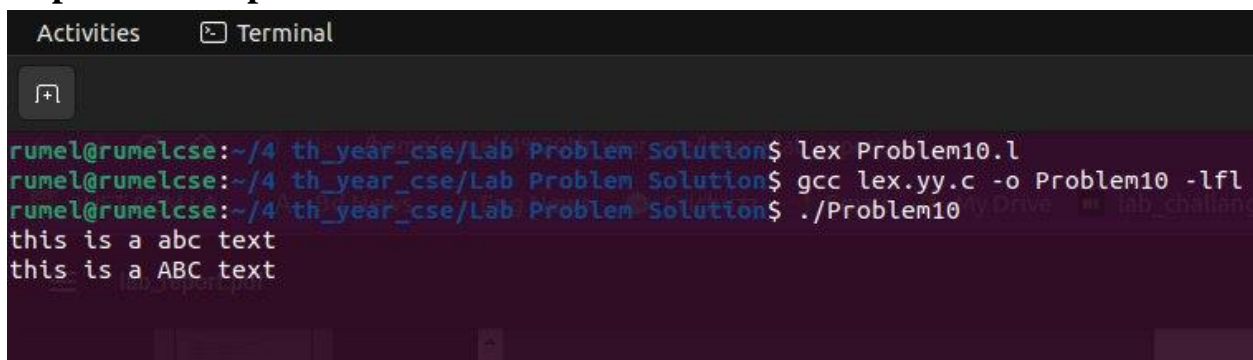
- Pattern Matching: Use regular expressions to define patterns, such as the occurrence of "abc" in the input text.
- Actions: Define actions to be taken when a pattern is matched. In this case, print "ABC" when the pattern "abc" is encountered.

### **Implementation:**

```
% {  
#include <stdio.h>  
% }  
%%  
abc  { printf("ABC"); }
```

```
.    { putchar(yytext[0]); }
%%
int yywrap() {
    return 1;
}
int main() {
    yylex();
    return 0;
}
```

### Input and Output:



```
Activities  Terminal
[+]
rumel@rumelcse:~/4 th_year_cse/Lab Problem Solution$ lex Problem10.l
rumel@rumelcse:~/4 th_year_cse/Lab Problem Solution$ gcc lex.yy.c -o Problem10 -lfl
rumel@rumelcse:~/4 th_year_cse/Lab Problem Solution$ ./Problem10
this is a abc text
this is a ABC text
```

### Conclusion:

The LEX program successfully recognizes the substring 'abc' in the input and converts it to 'ABC'. Lexical analysis is a crucial step in the compilation process, and this implementation demonstrates how LEX can be used to perform simple string transformations during lexical scanning. The program can be extended for more complex transformations based on specific language requirements.

### Problem Statement: 11

Write a Program to implement NFAs that recognize identifiers, constants, and operators of the mini language.

### Introduction:

The purpose of this lab is to implement Non-deterministic Finite Automata (NFAs) for recognizing identifiers, constants, and operators in a mini-language. NFAs are essential in lexical analysis, where they can efficiently identify different tokens based on their patterns.

### Requirements:

- C Compiler (gcc)
- Text Editor
- Ubuntu
- WSL

## Methodology:

- NFA Definition:
  - Define NFAs for identifiers, constants, and operators. NFAs consist of states, transitions, and accepting states. Each state corresponds to a specific stage in recognizing a token.
- Identifier NFA:
  - Create an NFA for recognizing identifiers. Identify states for the starting character, subsequent characters, and an accepting state. Transitions should account for valid characters in an identifier (alphanumeric and underscore).
- Constant NFA:
  - Develop an NFA for recognizing constants. This includes integer constants and floating-point constants. Define states for the integer part, optional decimal part, and an accepting state. Transitions should consider valid numeric characters and the decimal point.
- Operator NFA:
  - Construct an NFA for recognizing operators. Operators can be single-character or multi-character. Define states for each character in an operator, and transitions should capture the valid sequence of characters for each operator.

## Implementation:

```
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
#include <ctype.h> // Include the <ctype.h> header for isalpha and isdigit functions

// Define states for identifiers, constants, and operators
enum State {
    IDENTIFIER,
    CONSTANT,
    OPERATOR,
    INVALID
};
```

```

// Function to recognize identifiers, constants, and operators using NFAs
enum State recognizeToken(char token[]) {
    if (isalpha(token[0]) || token[0] == '_') {
        // Identifier: Starts with a letter or underscore
        return IDENTIFIER;
    } else if (isdigit(token[0])) {
        // Constant: Starts with a digit
        return CONSTANT;
    } else {
        // Operator: Check for specific operators (for simplicity, consider +, -, *, /)
        if (strcmp(token, "+") == 0 || strcmp(token, "-") == 0 ||
            strcmp(token, "*") == 0 || strcmp(token, "/") == 0) {
            return OPERATOR;
        } else {
            return INVALID;
        }
    }
}

// Function to display the result based on the recognized token
void displayResult(enum State state, char token[]) {
    switch (state) {
        case IDENTIFIER:
            printf("Token: %s is an Identifier\n", token);
            break;
        case CONSTANT:
            printf("Token: %s is a Constant\n", token);
            break;
        case OPERATOR:
            printf("Token: %s is an Operator\n", token);
            break;
        case INVALID:
            printf("Token: %s is Invalid\n", token);
            break;
    }
}

int main() {
    char input[100];

    printf("Enter a sequence of tokens separated by spaces: ");

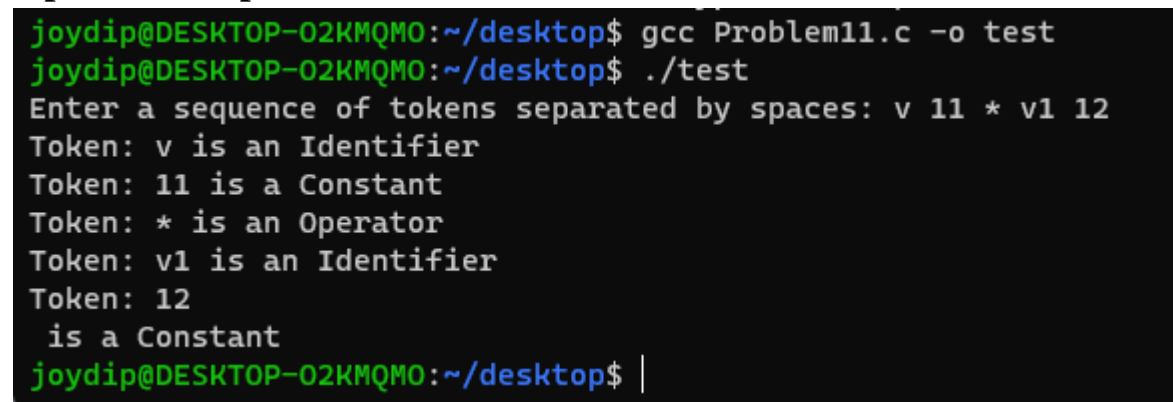
```



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

char *token = strtok(input, " ");
while (token != NULL) {
    enum State tokenType = recognizeToken(token);
    displayResult(tokenType, token);
    // Get the next token
    token = strtok(NULL, " ");
}
return 0;
}
```

### Input and Output:



```
joydip@DESKTOP-02KMQMO:~/desktop$ gcc Problem11.c -o test
joydip@DESKTOP-02KMQMO:~/desktop$ ./test
Enter a sequence of tokens separated by spaces: v 11 * v1 12
Token: v is an Identifier
Token: 11 is a Constant
Token: * is an Operator
Token: v1 is an Identifier
Token: 12
      is a Constant
joydip@DESKTOP-02KMQMO:~/desktop$ |
```

### Conclusion:

This lab demonstrates the practical application of NFAs in recognizing identifiers, constants, and operators within a mini-language. The implemented program provides a foundation for incorporating lexical analysis techniques into a compiler or language processing system.

### Problem Statement: 12

Write a Program to implement DFAs that recognize identifiers, constants, and operators of the mini language

### Introduction:

The objective of this lab is to design and implement a program that utilizes Deterministic Finite Automata (DFAs) to recognize identifiers, constants, and operators in a mini-language.

DFAs are mathematical models used in computer science to describe and recognize patterns in strings. In this lab, we apply the concept of DFAs to create a program that can categorize tokens in a simple programming language, such as identifiers, constants, and operators.

### Requirements:

- C Compiler (gcc)
- Text Editor
- Ubuntu
- WSL

### Methodology:

We have implemented a C program that defines a DFA to recognize different types of tokens.

The program uses the following functions:

isAlpha(char ch): Checks if a character is an alphabet or underscore.

isDigit(char ch): Checks if a character is a digit.

isOperator(char ch): Checks if a character is an operator.

recognizeToken(const char token):

### Implementation:

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

// DFA states
enum State {
    START, IDENTIFIER, CONSTANT, OPERATOR, ERROR
};

// Function to check if a character is an alphabet or underscore
bool isAlpha(char ch) {
    return (ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z') || ch == '_';
}

// Function to check if a character is a digit
bool isDigit(char ch) {
    return ch >= '0' && ch <= '9';
}
```

```

// Function to check if a character is an operator
bool isOperator(char ch) {
    return ch == '+' || ch == '-' || ch == '*' || ch == '/';
}

// Function to implement DFA for recognizing identifiers, constants, and operators
void recognizeToken(const char* token) {
    enum State state = START;

    for (int i = 0; i < strlen(token); i++) {
        char currentChar = token[i];
        switch (state) {
            case START:
                if (isAlpha(currentChar))
                    state = IDENTIFIER;
                else if (isDigit(currentChar))
                    state = CONSTANT;
                else if (isOperator(currentChar))
                    state = OPERATOR;
                else
                    state = ERROR;
                break;

            case IDENTIFIER:
                if (!isAlpha(currentChar) && !isDigit(currentChar))
                    state = ERROR;
                break;

            case CONSTANT:
                if (!isDigit(currentChar))
                    state = ERROR;
                break;

            case OPERATOR:
                if (!isOperator(currentChar))
                    state = ERROR;
                break;

            case ERROR:
                break;
        }
    }
}

```

```

    }
}

// Check the final state and print the result
switch (state) {
    case IDENTIFIER:
        printf("%s is an Identifier\n", token);
        break;
    case CONSTANT:
        printf("%s is a Constant\n", token);
        break;
    case OPERATOR:
        printf("%s is an Operator\n", token);
        break;
    case ERROR:
        printf("%s is an Invalid Token\n", token);
        break;
    default:
        break;
}
}

int main() {
    // Example tokens to recognize
    const char* tokens[] = {"var", "123", "+", "identifier123", "*", "$$", "invalid_token"};

    // Process each token using the DFA
    for (int i = 0; i < sizeof(tokens) / sizeof(tokens[0]); i++) {
        recognizeToken(tokens[i]);
    }
    return 0;
}

```

## Input and Output:

```
joydip@DESKTOP-02KMQMO:~/desktop$ gcc Problem12.c -o test
joydip@DESKTOP-02KMQMO:~/desktop$ ./test
var is an Identifier
123 is a Constant
+ is an Operator
identifier123 is an Identifier
* is an Operator
$$ is an Invalid Token
invalid_token is an Identifier
```

## Conclusion:

The DFA-based approach used in this program provides a clear and systematic way to recognize different types of tokens. The transitions between states in the DFA correspond to the rules for identifying identifiers, constants, and operators in the mini-language.

By implementing a DFA for token recognition, we have achieved the goal of designing a program that can categorize identifiers, constants, and operators in a mini-language. This approach is extensible and can be enhanced to handle more complex language constructs.

## Problem Statement: 13

Write a Lex program to count the number of words, characters, blank spaces and lines.

## Introduction:

This Lex program is designed to count the number of words, characters, blank spaces, and lines in a given input. Lex is a tool for generating lexical analyzers, and in this program, it's used to define patterns that match words, spaces, newlines, and characters.

## Requirements:

- C Compiler (gcc)
- Text Editor
- Ubuntu
- WSL

## Methodology:

- Include C Code: The `%{ ... %}` section is used for including C code. In this section, variables are declared to store counts for words, characters, lines, and spaces.

- Define Lexical Rules: The %% section contains the lexical rules. Each rule specifies a pattern to match, and an associated action is taken when a match is found.
- [a-zA-Z0-9]+: Matches words consisting of alphanumeric characters.
- \n: Matches newline characters to count lines.
- [\t]: Matches spaces and tabs to count blank spaces.
- .: Matches any character to count total characters.
- Action Section: After each match, actions are performed to update the counts.
- Main Function: The main function invokes yylex() to start the lexical analysis. After the analysis, it prints the counts of words, characters, lines, and spaces.

## Implementation:

```
% {
int charCount = 0;
int wordCount = 0;
int lineCount = 0;
int spaceCount = 0;
% }

%%
[a-zA-Z0-9]+ { wordCount++; charCount += yyleng; }
\n      { lineCount++; }
[\t]    { spaceCount++; }
.       { charCount++; }
%%

int main() {
    yylex();
    printf("Words: %d\n", wordCount);
    printf("Characters: %d\n", charCount);
    printf("Lines: %d\n", lineCount);
    printf("Spaces: %d\n", spaceCount);
    return 0;
}
```

## Input and Output:.

```
joydip@DESKTOP-02KMQMO:~/desktop$ gcc lex.yy.c -o scanner
joydip@DESKTOP-02KMQMO:~/desktop$ ./scanner
Enter the Sentence : This is LEX program
Number of lines : 1
Number of spaces : 3
Number of tabs, words, charc : 0 , 4 , 20
```

## Conclusion:

This Lex program provides a simple and effective way to count various aspects of text input, making it useful for basic text analysis tasks.

## Problem Statement: 14

Write a program to Elimination of Left Recursion in a grammar.

## Introduction:

The main objective of this lab is to implement a program that eliminates left recursion from a given context-free grammar. Left recursion can cause issues in recursive descent parsers, and eliminating it is crucial for proper parsing.

Left recursion is a condition in a grammar where a non-terminal A directly produces a sequence that starts with itself. Left recursion can lead to infinite loops in recursive descent parsers. The elimination of left recursion involves transforming the given grammar to an equivalent grammar that doesn't have left recursion.

## Requirements:

- C Compiler (gcc)
- Text Editor
- Ubuntu
- WSL

## Methodology:

The elimination of left recursion is done through a series of steps:

- Input Grammar:
  - The grammar is represented as a set of production rules.
  - Each rule is in the form  $A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$ .

- Identify Left Recursion:
  - For each non-terminal A, check if there are production rules of the form  $A \rightarrow A\alpha$ .
  - If left recursion is found, it needs to be eliminated.
- Elimination Steps:
  - For each rule  $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_n$ , create a new non-terminal A' and replace the rule with:
    - $A \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_n A'$
    - $A' \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_m$
    - Remove any direct left recursion in A'.
- Update Grammar:
  - Update the original grammar with the modified rules.
- Output:
  - Display the modified grammar without left recursion.

## Implementation:

```
#include <stdio.h>
#include <string.h>
#define SIZE 10
int main()
{
    char non_terminal;
    char beta, alpha;
    int num;
    char production[10][SIZE];
    int index = 3; /* starting of the string following "->" */
    printf("Enter Number of Production : ");
    scanf("%d", &num);
    printf("Enter the grammar as E->E-A :\n");
    for (int i = 0; i < num; i++)
    {
        scanf("%s", production[i]);
    }
    for (int i = 0; i < num; i++)
    {
        printf("\nGRAMMAR : : : %s", production[i]);
        non_terminal = production[i][0];
        if (non_terminal == production[i][index])
        {
            alpha = production[i][index + 1];
```

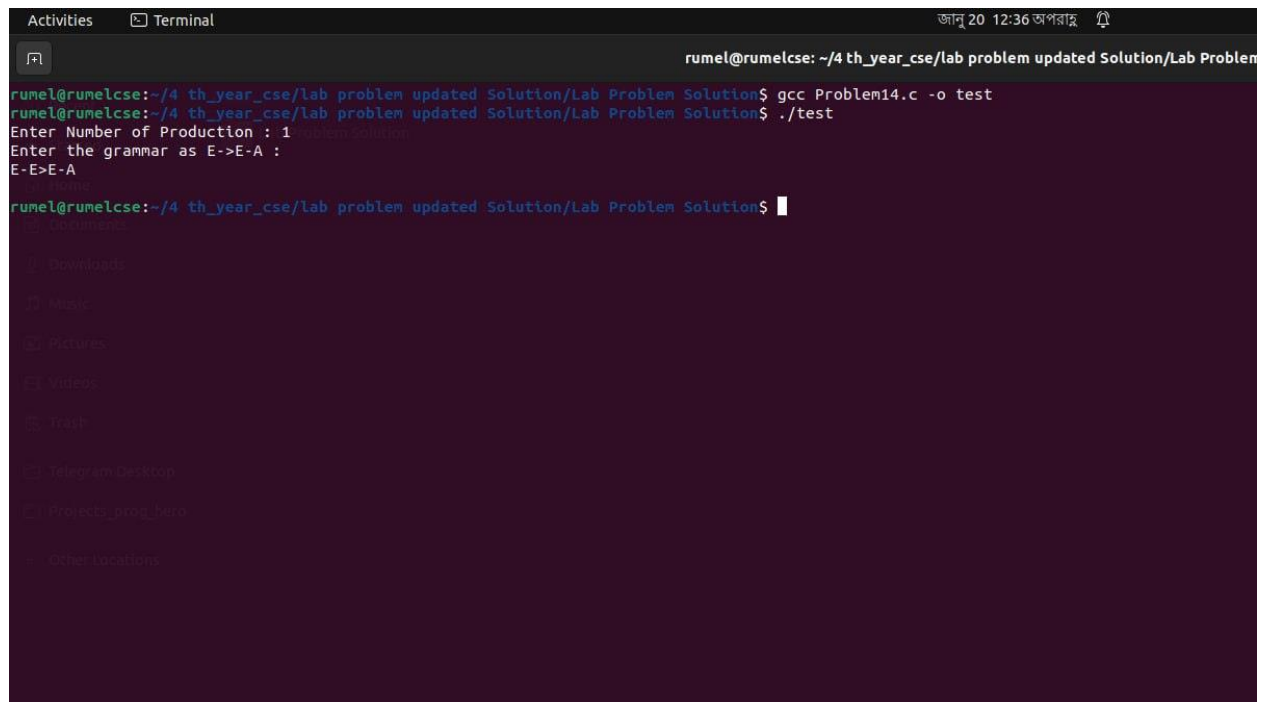


```

printf(" is left recursive.\n");
while (production[i][index] != 0 && production[i][index] != '|')
    index++;
if (production[i][index] != 0)
{
    beta = production[i][index + 1];
    printf("Grammar without left recursion:\n");
    printf("%c->%c%c\\", non_terminal, beta, non_terminal);
    printf("\\n%c\\'->%c%c\\'|epsilon\\n", non_terminal, alpha, non_terminal);
}
else
    printf(" can't be reduced\\n");
}
else
    printf(" is not left recursive.\n");
index = 3;
}
}

```

## Input and Output:



```

Activities Terminal
rumel@rumelcse: ~/4 th_year_cse/lab problem updated Solution/Lab Problem Solution$ gcc Problem14.c -o test
rumel@rumelcse:~/4 th_year_cse/lab problem updated Solution/Lab Problem Solution$ ./test
Enter Number of Production : 1
Enter the grammar as E->E-A :
E->E-A
rumel@rumelcse:~/4 th_year_cse/lab problem updated Solution/Lab Problem Solution$

```

## Conclusion:

The elimination of left recursion is a critical step in preparing a context-free grammar for parsing. This lab provides hands-on experience in identifying left recursion and implementing the necessary transformations to remove it. Understanding and applying such transformations are essential skills for compiler design and parser construction.