CC Lab Final Report

Lexor code:

```
import java.io.*;
import java.util.ArrayList;
import java.util.Scanner;
import java.util.regex.Pattern;

public class Lexor {
    // two pointers named ch and temp
    // two buffer lexical analyzer implemented
    // one pointer remains still while the other pointer keeps moving

    public static void main(String[] args) {
        // write your code here
        ArrayList<String> keywords = new ArrayList<>();

        //Defining types
        keywords.add("int");
        keywords.add("float");
        keywords.add("while");
        keywords.add("main");
        keywords.add("if");
        keywords.add("else");
        keywords.add("new");

        //Pushing required keywords
        Pattern constants = Pattern.compile("^[0-9][0-9]*(([\\.][0-9][0-
9]*)?([e][+|-][0-9][0-9]*)?)?$");
        Pattern variables = Pattern.compile("^[A-Za-z|_][A-Za-z|0-9]*$");
        Pattern operators = Pattern.compile("^[-,*,+,/,>,<,&,|,=]$");

        ArrayList<Node> data = new ArrayList<>();

        File file = new
File("C:\\Users\\talha\\IdeaProjects\\CClabs\\test.txt");
        // opened the test code file
        try {
            // scanner to read the file
            Scanner read = new Scanner(file);

            int i = 1; // to note the line number
            while (read.hasNext()) {

                // looping over the input file
                String line = read.nextLine(); // reading each next line
                String temp = "";
                boolean qoute = false; // checking if the input is a string
to there can be soaces in a strin to maintain this

                for (char ch : line.toCharArray()) { // first buffer
pointer
                    if (ch == '(' || ch == ')' || ch == '{' || ch == '}') {
                        // case 1 if the input is brackets
                        boolean flag = true;

                        // check if it already exists in the list then
```

```
increment it
                        for (Node n : data) {
                            if (n.data.equals("" + ch)) {
                                n.count++;
                                flag = false;
                                break;
                            }
                        }
                        if (flag) {
                            // if not already added add the lexem in the
table
                            data.add(new Node("" + ch, "Brackets", i));
                        }
                        temp = "";
                        continue;

                    } else if (ch == '\"' && !qoute) {// case found an
opening qoutation
                        qoute = true;
                    } else if (ch == '\"' && qoute) {// case found closing
braket
                        temp += ch;
                        data.add(new Node(temp, "String", i));// add the
string to the list

                        // as string are not checked for repetation so just
add them
                        temp = "";
                        qoute = false;
                        continue;
                    } else if (ch == ' ' && !qoute) {
                        // case found a space
                        if (temp.equals(" ")) { // sub case check if the
second pointer if pointing to a
                            // substring that is all spaces if yes rest the
pointers and continue to next iteration
                            temp = "";
                            continue;
                        }
                        boolean flag = true; // checking if not already in
the data table
                        for (Node n : data) {
                            if (n.data.equals(temp)) {
                                n.count++; // if found increase the count
                                flag = false;
                                break;
                            }
                        }
                        if (flag) { // case if not found in data table
                            if (keywords.contains(temp)) { // sub case
would be a keyword
                                data.add(new Node(temp, "Keyword", i));
                                temp = "";
                                continue;
                            }
                            if (constants.matcher(temp).find()) { // sub
case would be a constant
                                data.add(new Node(temp, "Constant", i));
                                temp = "";
                                continue;
                            }
```

```
                            if (variables.matcher(temp).find()) { // sub
case would be a variable
                                data.add(new Node(temp, "Variable", i));
                                temp = "";
                                continue;
                            }
                            if (operators.matcher(temp).find()) { // sub
case would be an operator
                                data.add(new Node(temp, "Operator", i));
                                temp = "";
                                continue;
                            }
                            // if any sub case match then continue to next
point after resetting the pointers
                        }
                        temp = "";
                    }
                    temp += ch; //  2nd point to store the substring
                }
                i++;
            }
            System.out.println("---------------------------------------------
---------------------------------------------------");
            System.out.printf("%5s %14s %14s %15s", "Lexem", "Type", "Line
num", "Repetition");
            System.out.println();
            System.out.println("---------------------------------------------
---------------------------------------------------");
            for (Node n : data) { // printing the content in the data table
                System.out.format("%7s %14s %7s %10s", n.data, n.type,
n.line, n.count);
                System.out.println();
//                System.out.println(n.toString());
            }
            System.out.println("---------------------------------------------
----------------------------------------------------");
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}

class Node {

    // lexem class
    String data;
    int count;
    String type;
    int line;

    public Node(String data, String type, int line) {
        this.data = data;
        this.type = type;
        this.line = line;
        this.count = 1;
    }

    // method to convert a lexem row into string to print it on console
    public String toString() {
        return "-------------\nLexem: " + data + "\nType: " + type + "
Found on Line: " + line + " Repeated: " + count + " times: " ;
```

```
    }

}
```

Lexor Output:

```
--------------------------------------------------------------------------------------------
Lexem           Type       Line num      Repetition
--------------------------------------------------------------------------------------------
    int        Keyword       1          2
   main        Keyword       1          1
      (       Brackets       1          3
      )       Brackets       1          3
      {       Brackets       1          1
  float        Keyword       2          1
   test       Variable       2          1
      =       Operator       2          1
      9       Constant       2          1
 printf       Variable       4          1
 "Sum of "          String       4          1
  "and"         String       4          1
  "is "         String       4          1
      +       Operator       4          1
      }       Brackets       5          1
---------------------------------------------------------------------------------------------


Process finished with exit code 0
```

Parser code:

```
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
```

```python
    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
```

```python
    # 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)
```

```python
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
```

```python
    # 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('I' + 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)
```

Parser Output:

```
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
```

```
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 ( 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		S6				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			

```
Process finished with exit code 0
```

LL1 Parser code:

```python
def removeLeftRecursion(rulesDiction):
   # for rule: A->Aa|b
   # result: A->bA',A'->aA'|#

   # 'store' has new rules to be added
   store = {}
   # traverse over rules
   for lhs in rulesDiction:
      # alphaRules stores subrules with left-recursion
      # betaRules stores subrules without left-recursion
      alphaRules = []
      betaRules = []
      # get rhs for current lhs
      allrhs = rulesDiction[lhs]
      for subrhs in allrhs:
         if subrhs[0] == lhs:
            alphaRules.append(subrhs[1:])
         else:
            betaRules.append(subrhs)
      # alpha and beta containing subrules are separated
      # now form two new rules
      if len(alphaRules) != 0:
         # to generate new unique symbol
         # add ' till unique not generated
         lhs_ = lhs + "'"
         while (lhs_ in rulesDiction.keys()) \
               or (lhs_ in store.keys()):
            lhs_ += "'"
         # make beta rule
         for b in range(0, len(betaRules)):
            betaRules[b].append(lhs_)
```

```
            rulesDiction[lhs] = betaRules
            # make alpha rule
            for a in range(0, len(alphaRules)):
                alphaRules[a].append(lhs_)
            alphaRules.append(['#'])
            # store in temp dict, append to
            # - rulesDiction at end of traversal
            store[lhs_] = alphaRules
    # add newly generated rules generated
    # - after removing left recursion
    for left in store:
        rulesDiction[left] = store[left]
    return rulesDiction


def LeftFactoring(rulesDiction):
    # for rule: A->aDF|aCV|k
    # result: A->aA'|k, A'->DF|CV

    # newDict stores newly generated
    # - rules after left factoring
    newDict = {}
    # iterate over all rules of dictionary
    for lhs in rulesDiction:
        # get rhs for given lhs
        allrhs = rulesDiction[lhs]
        # temp dictionary helps detect left factoring
        temp = dict()
        for subrhs in allrhs:
            if subrhs[0] not in list(temp.keys()):
                temp[subrhs[0]] = [subrhs]
            else:
                temp[subrhs[0]].append(subrhs)
        # if value list count for any key in temp is > 1,
        # - it has left factoring
        # new_rule stores new subrules for current LHS symbol
        new_rule = []
        # temp_dict stores new subrules for left factoring
        tempo_dict = {}
        for term_key in temp:
            # get value from temp for term_key
            allStartingWithTermKey = temp[term_key]
            if len(allStartingWithTermKey) > 1:
                # left factoring required
                # to generate new unique symbol
                # - add ' till unique not generated
                lhs_ = lhs + "'"
                while (lhs_ in rulesDiction.keys()) \
                        or (lhs_ in tempo_dict.keys()):
                    lhs_ += "'"
                # append the left factored result
                new_rule.append([term_key, lhs_])
                # add expanded rules to tempo_dict
                ex_rules = []
                for g in temp[term_key]:
                    ex_rules.append(g[1:])
                tempo_dict[lhs_] = ex_rules
            else:
                # no left factoring required
                new_rule.append(allStartingWithTermKey[0])
        # add original rule
```

```python
        newDict[lhs] = new_rule
        # add newly generated rules after left factoring
        for key in tempo_dict:
            newDict[key] = tempo_dict[key]
    return newDict


# 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
# use 'rules' list, and 'diction' dict from above

# follow function input is the split result on
# - Non-Terminal whose Follow we want to compute
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 computeAllFirsts():
    global rules, nonterm_userdef, \
        term_userdef, diction, firsts
    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

    print(f"\nRules: \n")
    for y in diction:
        print(f"{y}->{diction[y]}")
    print(f"\nAfter elimination of left recursion:\n")

    diction = removeLeftRecursion(diction)
    for y in diction:
        print(f"{y}->{diction[y]}")
    print("\nAfter left factoring:\n")

    diction = LeftFactoring(diction)
    for y in diction:
        print(f"{y}->{diction[y]}")

    # calculate first for each rule
    # - (call first() on all RHS)
    for y in list(diction.keys()):
        t = set()
        for sub in diction.get(y):
            res = first(sub)
            if res != None:
                if type(res) is list:
                    for u in res:
                        t.add(u)
                else:
                    t.add(res)

        # save result in 'firsts' list
        firsts[y] = t

    print("\nCalculated firsts: ")
    key_list = list(firsts.keys())
    index = 0
    for gg in firsts:
        print(f"first({key_list[index]}) "
            f"=> {firsts.get(gg)}")
```

```
        index += 1


def computeAllFollows():
    global start_symbol, rules, nonterm_userdef,\
        term_userdef, diction, firsts, follows
    for NT in diction:
        solset = set()
        sol = follow(NT)
        if sol is not None:
            for g in sol:
                solset.add(g)
        follows[NT] = solset

    print("\nCalculated follows: ")
    key_list = list(follows.keys())
    index = 0
    for gg in follows:
        print(f"follow({key_list[index]})"
            f" => {follows[gg]}")
        index += 1


# create parse table
def createParseTable():
    import copy
    global diction, firsts, follows, term_userdef
    print("\nFirsts and Follow Result table\n")

    # find space size
    mx_len_first = 0
    mx_len_fol = 0
    for u in diction:
        k1 = len(str(firsts[u]))
        k2 = len(str(follows[u]))
        if k1 > mx_len_first:
            mx_len_first = k1
        if k2 > mx_len_fol:
            mx_len_fol = k2

    print(f"{{:<{10}}} "
        f"{{:<{mx_len_first + 5}}} "
        f"{{:<{mx_len_fol + 5}}}"
        .format("Non-T", "FIRST", "FOLLOW"))
    for u in diction:
        print(f"{{:<{10}}} "
            f"{{:<{mx_len_first + 5}}} "
            f"{{:<{mx_len_fol + 5}}}"
            .format(u, str(firsts[u]), str(follows[u])))

    # create matrix of row(NT) x [col(T) + 1($)]
    # create list of non-terminals
    ntlist = list(diction.keys())
    terminals = copy.deepcopy(term_userdef)
    terminals.append('$')

    # create the initial empty state of ,matrix
    mat = []
    for x in diction:
        row = []
        for y in terminals:
```

```
            row.append('')
        # of $ append one more col
        mat.append(row)

    # Classifying grammar as LL(1) or not LL(1)
    grammar_is_LL = True

    # rules implementation
    for lhs in diction:
        rhs = diction[lhs]
        for y in rhs:
            res = first(y)
            # epsilon is present,
            # - take union with follow
            if '#' in res:
                if type(res) == str:
                    firstFollow = []
                    fol_op = follows[lhs]
                    if fol_op is str:
                        firstFollow.append(fol_op)
                    else:
                        for u in fol_op:
                            firstFollow.append(u)
                    res = firstFollow
                else:
                    res.remove('#')
                    res = list(res) +\
                        list(follows[lhs])
            # add rules to table
            ttemp = []
            if type(res) is str:
                ttemp.append(res)
                res = copy.deepcopy(ttemp)
            for c in res:
                xnt = ntlist.index(lhs)
                yt = terminals.index(c)
                if mat[xnt][yt] == '':
                    mat[xnt][yt] = mat[xnt][yt] \
                                 + f"{lhs}->{' '.join(y)}"
                else:
                    # if rule already present
                    if f"{lhs}->{y}" in mat[xnt][yt]:
                        continue
                    else:
                        grammar_is_LL = False
                        mat[xnt][yt] = mat[xnt][yt] \
                                     + f",{lhs}->{' '.join(y)}"

    # final state of parse table
    print("\nGenerated parsing table:\n")
    frmt = "{:>12}" * len(terminals)
    print(frmt.format(*terminals))

    j = 0
    for y in mat:
        frmt1 = "{:>12}" * len(y)
        print(f"{ntlist[j]} {frmt1.format(*y)}")
        j += 1

    return (mat, grammar_is_LL, terminals)
```

```python
def validateStringUsingStackBuffer(parsing_table, grammarll1,
                        table_term_list, input_string,
                        term_userdef,start_symbol):

   print(f"\nValidate String => {input_string}\n")

   # for more than one entries
   # - in one cell of parsing table
   if grammarll1 == False:
      return f"\nInput String = " \
         f"\"{input_string}\"\n" \
         f"Grammar is not LL(1)"

   # implementing stack buffer

   stack = [start_symbol, '$']
   buffer = []

   # reverse input string store in buffer
   input_string = input_string.split()
   input_string.reverse()
   buffer = ['$'] + input_string

   print("{:>20} {:>20} {:>20}".
      format("Buffer", "Stack","Action"))

   while True:
      # end loop if all symbols matched
      if stack == ['$'] and buffer == ['$']:
         print("{:>20} {:>20} {:>20}"
            .format(' '.join(buffer),
                  ' '.join(stack),
                  "Valid"))
         return "\nValid String!"
      elif stack[0] not in term_userdef:
         # take font of buffer (y) and tos (x)
         x = list(diction.keys()).index(stack[0])
         y = table_term_list.index(buffer[-1])
         if parsing_table[x][y] != '':
            # format table entry received
            entry = parsing_table[x][y]
            print("{:>20} {:>20} {:>25}".
               format(' '.join(buffer),
                     ' '.join(stack),
                     f"T[{stack[0]}][{buffer[-1]}] = {entry}"))
            lhs_rhs = entry.split("->")
            lhs_rhs[1] = lhs_rhs[1].replace('#', '').strip()
            entryrhs = lhs_rhs[1].split()
            stack = entryrhs + stack[1:]
         else:
            return f"\nInvalid String! No rule at " \
               f"Table[{stack[0]}][{buffer[-1]}]."
      else:
         # stack top is Terminal
         if stack[0] == buffer[-1]:
            print("{:>20} {:>20} {:>20}"
               .format(' '.join(buffer),
                     ' '.join(stack),
                     f"Matched:{stack[0]}"))
            buffer = buffer[:-1]
```

```
            stack = stack[1:]
        else:
            return "\nInvalid String! " \
               "Unmatched terminal symbols"


# DRIVER CODE - MAIN

# NOTE: To test any of the sample sets, uncomment ->
# 'rules' list, 'nonterm_userdef' list, 'term_userdef' list
# and for any String validation uncomment following line with
# 'sample_input_String' variable.

sample_input_string = None

# sample set 1 (Result: Not LL(1))
# rules=["A -> S B | B",
#   "S -> a | B c | #",
#   "B -> b | d"]
# nonterm_userdef=['A','S','B']
# term_userdef=['a','c','b','d']
# sample_input_string="b c b"

# sample set 2 (Result: LL(1))
# rules=["S -> A | B C",
#   "A -> a | b",
#   "B -> p | #",
#   "C -> c"]
# nonterm_userdef=['A','S','B','C']
# term_userdef=['a','c','b','p']
# sample_input_string="p c"

# sample set 3 (Result: LL(1))
# rules=["S -> A B | C",
#   "A -> a | b | #",
#   "B-> p | #",
#   "C -> c"]
# nonterm_userdef=['A','S','B','C']
# term_userdef=['a','c','b','p']
# sample_input_string="a c b"

# sample set 4 (Result: Not LL(1))
# rules = ["S -> A B C | C",
#      "A -> a | b B | #",
#      "B -> p | #",
#      "C -> c"]
# nonterm_userdef=['A','S','B','C']
# term_userdef=['a','c','b','p']
# sample_input_string="b p p c"

# sample set 5 (With left recursion)
# rules=["A -> B C c | g D B",
#   "B -> b C D E | #",
#   "C -> D a B | c a",
#   "D -> # | d D",
#   "E -> E a f | c"
#   ]
# nonterm_userdef=['A','B','C','D','E']
# term_userdef=["a","b","c","d","f","g"]
# sample_input_string="b a c a c"
```

```
# sample set 6
# rules=["E -> T E'",
#   "E' -> + T E' | #",
#   "T -> F T'",
#   "T' -> * F T' | #",
#   "F -> ( E ) | id"
# ]
# nonterm_userdef=['E','E\'','F','T','T\'']
# term_userdef=['id','+','*','(',')']
# sample_input_string="id * * id"
# example string 1
# sample_input_string="( id * id )"
# example string 2
# sample_input_string="( id ) * id + id"

# sample set 7 (left factoring & recursion present)
rules=["S -> A k O",
   "A -> A d | a B | a C",
   "C -> c",
   "B -> b B C | r"]

nonterm_userdef=['A','B','C']
term_userdef=['k','O','d','a','c','b','r']
sample_input_string="a r k O"

# sample set 8 (Multiple char symbols T & NT)
# rules = ["S -> NP VP",
#       "NP -> P | PN | D N",
#       "VP -> V NP",
#       "N -> championship | ball | toss",
#       "V -> is | want | won | played",
#       "P -> me | I | you",
#       "PN -> India | Australia | Steve | John",
#       "D -> the | a | an"]
#
# nonterm_userdef = ['S', 'NP', 'VP', 'N', 'V', 'P', 'PN', 'D']
# term_userdef = ["championship", "ball", "toss", "is", "want",
#             "won", "played", "me", "I", "you", "India",
#             "Australia","Steve", "John", "the", "a", "an"]
# sample_input_string = "India won the championship"

# diction - store rules inputed
# firsts - store computed firsts
diction = {}
firsts = {}
follows = {}

# computes all FIRSTs for all non terminals
computeAllFirsts()
# assuming first rule has start_symbol
# start symbol can be modified in below line of code
start_symbol = list(diction.keys())[0]
# computes all FOLLOWs for all occurrences
computeAllFollows()
# generate formatted first and follow table
# then generate parse table

(parsing_table, result, tabTerm) = createParseTable()

# validate string input using stack-buffer concept
if sample_input_string != None:
```

```
    validity = validateStringUsingStackBuffer(parsing_table, result,
                                              tabTerm, sample_input_string,
                                              term_userdef,start_symbol)
    print(validity)
else:
    print("\nNo input String detected")
```

LL1 Parser Output:

```
Rules:

S->[['A', 'k', 'O']]
A->[['A', 'd'], ['a', 'B'], ['a', 'C']]
C->[['c']]
B->[['b', 'B', 'C'], ['r']]

After elimination of left recursion:

S->[['A', 'k', 'O']]
A->[['a', 'B', "A'"], ['a', 'C', "A'"]]
C->[['c']]
B->[['b', 'B', 'C'], ['r']]
A'->[['d', "A'"], ['#']]

After left factoring:

S->[['A', 'k', 'O']]
A->[['a', "A''"]]
A''->[['B', "A'"], ['C', "A'"]]
C->[['c']]
B->[['b', 'B', 'C'], ['r']]
A'->[['d', "A'"], ['#']]
```

```
Calculated firsts:
first(S) => {'a'}
first(A) => {'a'}
first(A'') => {'r', 'b', 'c'}
first(C) => {'c'}
first(B) => {'r', 'b'}
first(A') => {'#', 'd'}

Calculated follows:
follow(S) => {'$'}
follow(A) => {'k'}
follow(A'') => {'k'}
follow(C) => {'k', 'c', 'd'}
follow(B) => {'k', 'c', 'd'}
follow(A') => {'k'}
```

```
Firsts and Follow Result table

Non-T      FIRST               FOLLOW
S          {'a'}               {'$'}
A          {'a'}               {'k'}
A''        {'r', 'b', 'c'}     {'k'}
C          {'c'}               {'k', 'c', 'd'}
B          {'r', 'b'}          {'k', 'c', 'd'}
A'         {'#', 'd'}          {'k'}

Generated parsing table:

           k           O           d           a           c           b           r           $
S                                           S->A k O
A                                           A->a A''
A''                                                    A''->C A'   A''->B A'   A''->B A'
C                                                         C->c
B                                                                 B->b B C        B->r
A'        A'->#                   A'->d A'
```

```
Validate String => a r k 0

              Buffer                  Stack             Action
            $ 0 k r a                   S $        T[S][a] = S->A k 0
            $ 0 k r a               A k 0 $        T[A][a] = A->a A''
            $ 0 k r a          a A'' k 0 $              Matched:a
              $ 0 k r            A'' k 0 $     T[A''][r] = A''->B A'
              $ 0 k r           B A' k 0 $             T[B][r] = B->r
              $ 0 k r           r A' k 0 $              Matched:r
                $ 0 k             A' k 0 $          T[A'][k] = A'->#
                $ 0 k                k 0 $              Matched:k
                  $ 0                  0 $              Matched:0
                    $                    $                  Valid

Valid String!

Process finished with exit code 0
```

Semantic Analyser Code:

```
import pandas as pd
import copy
try:
    a=pd.read_csv("input.csv")
    print("\a One thing in this program is that it takes an input of csv file.\n\aThe formate of the csv file is in following manner: ")
    print(a)
    print("\a The output of this program is based on above csv file.\n\aYou can take the different input csv file for diffrent required output.")
    print("\a One thing is to be noticed that in the csv file, there are two columns one is left and other is right for left and right values respectively.")
    print("\a There is only one left side variable for each equation and it may be possible that more than one varible in right side.")
    print("\a The operators and operands which are used in right side must be space separated from each other.")
    print("\a This program is case sensitive, this means that 'd*10' and '10*d' are treated as different equation.\nIf you want to solve this problem then you can use the 'CODE OPTIMIZATION TECHNIQUE'.")
    print('\t')
    c=a.shape# It will gives an tuple of numbers of rows and columns
    #print(c)
    l=[]
    o=list("+-*/")#If you want to add more operator youn can use that as well
    o1=[]
    r=[]
    for i in range(c[0]):# Here c[0] is 0th element of tuple c, which is a.shape (c=a.shape)
```

```
        l=l+[a['left'][i]]
        d=a['right'][i]
        x=d.split()
        l=l+x
    #print(l)
    sizel=len(l)
    for z in range(sizel):

        #print(sizel)
        if(l[z] in o):
            o1=o1+[l[z]]
    o1=list(set(o1))
    #print(o1)
    li=copy.deepcopy(l)# if you use li=l then it may occures some un usual error further in program.
    for x in o1:
        if(x in li):
            li.remove(x)
    li=list(set(li))
    #print(li)
    for b in range(len(li)):
        r=r+["R"+str(b)]
    #print(r)
    i=1
    ak=0
    z=0
    ACounter=0
    akm=[]
    while(i):
        if(ak==len(l)):
            i=0
        elif(l[ak].isalpha() and l[ak]==a['left'][z]):
            print("MOV "+str(l[ak])+' , '+str(r[li.index(l[ak])]))
            akm=akm+[r[li.index(l[ak])]]
            ak+=1
        elif(((l[ak].isalpha()) and (l[ak] in a['right'][z]))and (l[ak] not in o1)):
            print("MOV "+str(l[ak])+' , '+str(r[li.index(l[ak])]))
            akm=akm+[r[li.index(l[ak])]]
            ak+=1
            ACounter+=1
            if((len(a['right'][z])==1)and (len(akm)==2)):
               print("STOR "+str(akm[len(akm)-1])+' , '+str(akm[0]))
               #print(akm)
               akm.clear()
               z+=1
               print("\t")
        elif((l[ak] in a['right'][z]) and ((l[ak]in o1)and l[ak]=="+")):
            print("MOV "+str(l[ak+1])+' , '+str(r[li.index(l[ak+1])]))
            akm=akm+[r[li.index(l[ak+1])]]
            print("ADD "+str(akm[len(akm)-2])+' , '+str(akm[len(akm)-1]))
            akm.pop(len(akm)-2)
            #print(akm)
            print("STOR "+str(akm[len(akm)-1])+' , '+str(akm[0]))
            #print(ak)
            #print(ACounter)
            ak+=2
            ACounter+=2
            #print(ACounter)
            if(len(a['right'][z].split(" "))==ACounter):
                   #print(akm)
```

```
                akm.clear()
                z+=1
                ACounter=0
                #print(z)
                print("\t")
        elif((l[ak] in a['right'][z]) and ((l[ak]in o1)and l[ak]=="-")):
            print("MOV "+str(l[ak+1])+' , '+str(r[li.index(l[ak+1])]))
            akm=akm+[r[li.index(l[ak+1])]]
            print("SUB "+str(akm[len(akm)-2])+' , '+str(akm[len(akm)-1]))
            akm.pop(len(akm)-2)
            print("STOR "+str(akm[len(akm)-1])+' , '+str(akm[0]))
            ak+=2
            ACounter+=2
            #print(ACounter)
            if(len(a['right'][z].split(" "))==ACounter):
                #print(akm)
                akm.clear()
                z+=1
                ACounter=0
                #print(z)
                print("\t")
        elif((l[ak] in a['right'][z]) and ((l[ak]in o1)and l[ak]=="*")):
            print("MOV "+str(l[ak+1])+' , '+str(r[li.index(l[ak+1])]))
            akm=akm+[r[li.index(l[ak+1])]]
            print("MUL "+str(akm[len(akm)-2])+' , '+str(akm[len(akm)-1]))
            akm.pop(len(akm)-2)
            print("STOR "+str(akm[len(akm)-1])+' , '+str(akm[0]))
            ak+=2
            ACounter+=2
            #print(ACounter)
            if(len(a['right'][z].split(" "))==ACounter):
                #print(akm)
                akm.clear()
                z+=1
                ACounter=0
                #print(z)
                print("\t")
        elif((l[ak] in a['right'][z]) and ((l[ak]in o1)and l[ak]=="/")):
            print("MOV "+str(l[ak+1])+' , '+str(r[li.index(l[ak+1])]))
            akm=akm+[r[li.index(l[ak+1])]]
            print("DIV "+str(akm[len(akm)-2])+' , '+str(akm[len(akm)-1]))
            akm.pop(len(akm)-2)
            print("STOR "+str(akm[len(akm)-1])+' , '+str(akm[0]))
            akm.clear()
            ak+=2
            ACounter+=2
            if(len(a['right'][z].split(" "))==ACounter):
                #print(akm)
                akm.clear()
                z+=1
                ACounter=0
                #print(z)
                print("\t")
        elif((l[ak].isnumeric())and(l[ak] in a['right'][z])):
            print("MOV "+str(l[ak])+' , '+str(r[li.index(l[ak])]))
            akm=akm+[r[li.index(l[ak])]]
            ak+=1
            ACounter+=1
            if((len(akm)==2)and (a['right'][z]==l[ak-1])):
               print("STOR "+str(akm[len(akm)-1])+' , '+str(akm[0]))
               akm.clear()
```

```
                z+=1
                ACounter=0
                #print(z)
                print("\t")
        elif((l[ak] not in o1)or (l[ak] not in string.ascii_lowercase)):
            print("\f Error!\n\f Please enter valid syntax for three
address code.\n\f Check your csv file...")
            print(f"\f Error description...\nError in line number {z} and
place number {ak}.")
            print(f"\f Error element is {a['right'][z]}.")
            break
except (FileNotFoundError):
    print("Please check you input file. It may possible that file doesn't
exist.")
    print("Also check the file name that is given in input section at the
starting place.")
except(ArithmeticError):
    print("An arithmetic error is caused due to which program is not
proceed futher.Please check for the solution.")
except(IndexError):
    print("List index out of range.")
except:
    print("An exceptions occurred.")
```

Code Optimization and generation:

```
import pandas as pd
import copy
try:
    a=pd.read_csv("input.csv")
    print("\a One thing in this program is that it takes an input of csv
file.\n\aThe formate of the csv file is in following manner: ")
    print(a)
    print("\a The output of this program is based on above csv file.\n\aYou
can take the different input csv file for diffrent required output.")
    print("\a One thing is to be noticed that in the csv file, there are
two columns one is left and other is right for left and right values
respectively.")
    print("\a There is only one left side variable for each equation and it
may be possible that more than one varible in right side.")
    print("\a The operators and operands which are used in right side must
be space separated from each other.")
    print("\a This program is case sensitive, this means that 'd*10' and
'10*d' are treated as different equation.\nIf you want to solve this
problem then you can use the 'CODE OPTIMIZATION TECHNIQUE'.")
    print('\t')
    c=a.shape# It will gives an tuple of numbers of rows and columns
    #print(c)
    l=[]
    o=list("+-*/")#If you want to add more operator youn can use that as
well
    o1=[]
    r=[]
    for i in range(c[0]):# Here c[0] is 0th element of tuple c, which is
a.shape (c=a.shape)
        l=l+[a['left'][i]]
        d=a['right'][i]
        x=d.split()
```

```
        l=l+x
    #print(l)
    sizel=len(l)
    for z in range(sizel):

        #print(sizel)
        if(l[z] in o):
            o1=o1+[l[z]]
    o1=list(set(o1))
    #print(o1)
    li=copy.deepcopy(l)# if you use li=l then it may occures some un usual
error further in program.
    for x in o1:
        if(x in li):
            li.remove(x)
    li=list(set(li))
    #print(li)
    for b in range(len(li)):
        r=r+["R"+str(b)]
    #print(r)
    i=1
    ak=0
    z=0
    ACounter=0
    akm=[]
    while(i):
        if(ak==len(l)):
            i=0
        elif(l[ak].isalpha() and l[ak]==a['left'][z]):
            print("MOV "+str(l[ak])+' , '+str(r[li.index(l[ak])]))
            akm=akm+[r[li.index(l[ak])]]
            ak+=1
        elif(((l[ak].isalpha()) and (l[ak] in a['right'][z]))and (l[ak] not
in o1)):
            print("MOV "+str(l[ak])+' , '+str(r[li.index(l[ak])]))
            akm=akm+[r[li.index(l[ak])]]
            ak+=1
            ACounter+=1
            if((len(a['right'][z])==1)and (len(akm)==2)):
               print("STOR "+str(akm[len(akm)-1])+' , '+str(akm[0]))
               #print(akm)
               akm.clear()
               z+=1
               print("\t")
        elif((l[ak] in a['right'][z]) and ((l[ak]in o1)and l[ak]=="+")):
            print("MOV "+str(l[ak+1])+' , '+str(r[li.index(l[ak+1])]))
            akm=akm+[r[li.index(l[ak+1])]]
            print("ADD "+str(akm[len(akm)-2])+' , '+str(akm[len(akm)-1]))
            akm.pop(len(akm)-2)
            #print(akm)
            print("STOR "+str(akm[len(akm)-1])+' , '+str(akm[0]))
            #print(ak)
            #print(ACounter)
            ak+=2
            ACounter+=2
            #print(ACounter)
            if(len(a['right'][z].split(" "))==ACounter):
                    #print(akm)
                    akm.clear()
                    z+=1
                    ACounter=0
```

```python
                #print(z)
                print("\t")
        elif((l[ak] in a['right'][z]) and ((l[ak]in o1)and l[ak]=="-")):
            print("MOV "+str(l[ak+1])+' , '+str(r[li.index(l[ak+1])]))
            akm=akm+[r[li.index(l[ak+1])]]
            print("SUB "+str(akm[len(akm)-2])+' , '+str(akm[len(akm)-1]))
            akm.pop(len(akm)-2)
            print("STOR "+str(akm[len(akm)-1])+' , '+str(akm[0]))
            ak+=2
            ACounter+=2
            #print(ACounter)
            if(len(a['right'][z].split(" "))==ACounter):
                #print(akm)
                akm.clear()
                z+=1
                ACounter=0
                #print(z)
                print("\t")
        elif((l[ak] in a['right'][z]) and ((l[ak]in o1)and l[ak]=="*")):
            print("MOV "+str(l[ak+1])+' , '+str(r[li.index(l[ak+1])]))
            akm=akm+[r[li.index(l[ak+1])]]
            print("MUL "+str(akm[len(akm)-2])+' , '+str(akm[len(akm)-1]))
            akm.pop(len(akm)-2)
            print("STOR "+str(akm[len(akm)-1])+' , '+str(akm[0]))
            ak+=2
            ACounter+=2
            #print(ACounter)
            if(len(a['right'][z].split(" "))==ACounter):
                #print(akm)
                akm.clear()
                z+=1
                ACounter=0
                #print(z)
                print("\t")
        elif((l[ak] in a['right'][z]) and ((l[ak]in o1)and l[ak]=="/")):
            print("MOV "+str(l[ak+1])+' , '+str(r[li.index(l[ak+1])]))
            akm=akm+[r[li.index(l[ak+1])]]
            print("DIV "+str(akm[len(akm)-2])+' , '+str(akm[len(akm)-1]))
            akm.pop(len(akm)-2)
            print("STOR "+str(akm[len(akm)-1])+' , '+str(akm[0]))
            akm.clear()
            ak+=2
            ACounter+=2
            if(len(a['right'][z].split(" "))==ACounter):
                #print(akm)
                akm.clear()
                z+=1
                ACounter=0
                #print(z)
                print("\t")
        elif((l[ak].isnumeric())and(l[ak] in a['right'][z])):
            print("MOV "+str(l[ak])+' , '+str(r[li.index(l[ak])]))
            akm=akm+[r[li.index(l[ak])]]
            ak+=1
            ACounter+=1
            if((len(akm)==2)and (a['right'][z]==l[ak-1])):
               print("STOR "+str(akm[len(akm)-1])+' , '+str(akm[0]))
               akm.clear()
               z+=1
               ACounter=0
               #print(z)
```

```
                print("\t")
        elif((l[ak] not in o1)or (l[ak] not in string.ascii_lowercase)):
            print("\f Error!\n\f Please enter valid syntax for three
address code.\n\f Check your csv file...")
            print(f"\f Error description...\nError in line number {z} and
place number {ak}.")
            print(f"\f Error element is {a['right'][z]}.")
            break
except (FileNotFoundError):
    print("Please check you input file. It may possible that file doesn't
exist.")
    print("Also check the file name that is given in input section at the
starting place.")
except(ArithmeticError):
    print("An arithmetic error is caused due to which program is not
proceed futher.Please check for the solution.")
except(IndexError):
    print("List index out of range.")
except:
    print("An exceptions occurred.")
```

Code Optimization and generation Output:

```
□ One thing in this program is that it takes an input of csv file.
□The formate of the csv file is in following manner:
  left      right
0    b  c + d + f
1    f      b + b
2    r          f
3    a         10
4    s     a + 10
5    d     s - 10
6    g     10 * d
7    j     d * 10
8    c          2
□ The output of this program is based on above csv file.
□You can take the different input csv file for diffrent required output.
□ One thing is to be noticed that in the csv file, there are two columns one is left and other is right for left and right values respectively.
□ There is only one left side variable for each equation and it may be possible that more than one varible in right side.
□ The operators and operands which are used in right side must be space separated from each other.
□ This program is case sensitive, this means that 'd*10' and '10*d' are treated as different equation.
If you want to solve this problem then you can use the 'CODE OPTIMIZATION TECHNIQUE'.
```

```
MOV b , R0
MOV c , R7
MOV d , R5
ADD R7 , R5
STOR R5 , R0
MOV f , R8
ADD R5 , R8
STOR R8 , R0

MOV f , R8
MOV b , R0
MOV b , R0
ADD R0 , R0
STOR R0 , R8

MOV r , R12
MOV f , R8
STOR R8 , R12

MOV a , R11
MOV 10 , R10
STOR R10 , R11
```

```
MOV s , R4
MOV a , R11
MOV 10 , R10
ADD R11 , R10
STOR R10 , R4

MOV d , R5
MOV s , R4
MOV 10 , R10
SUB R4 , R10
STOR R10 , R5

MOV g , R1
MOV 10 , R10
MOV d , R5
MUL R10 , R5
STOR R5 , R1

MOV j , R2
MOV d , R5
MOV 10 , R10
MUL R5 , R10
STOR R10 , R2
```