LP Assignment 2

BT19CSE**122** - Deep Walke
22-02-2022

PROBLEM STATEMENT

Given the grammar in the file, generate the parsing table also output FIRST and FOLLOW sets for each non-terminal:

Roll Numbers ending with: **2**, 6: **LALR**

LALR PARSER

CODE

Language: C++

INPUT FORMAT

TERMINALS and NONTERMINALS should be declared as a list of strings first as shown below
'e' is considered as epsilon

example :

TERMINALS ( )
NONTERMINALS S
S -> (S)
S -> e

lalr.cpp

DATA STRUCTURES AND GLOBAL COUNTERS

- **Terminals & Nonterminals** are stored as vectors of strings.
- **Productions** are stored as an unordered_map where nonterminals are mapped to its rules.
- **First and Follow sets** are a map from nonterminal to its first/follow sets.
- **productionsMap** has a new start state and index to each state.
- **item** is a pair of a 2-d vector and vector of string which will basically represent a canonical item. It stores the production along with its corresponding set of lookahead symbols.
- **canonicalItems** is a vector<vector<item>>, it stores all the canonical items generated.
- **parseTableCLR** and **parseTableLALR** are basically the parse tables for respective grammars.
- **mergeIndex** is a vector<vector<int>> which stores indexes, these indexes are basically the indices of states which has to be merged
- **srConflict, rrConflict, clrConflicts, srCount, rrCount** are used to take care of all the conflicts.

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

vector<string> terminals;
vector<string> nonterminals;
unordered_map<string, vector<vector<string>>> productions;

unordered_map<string, vector<string>> first;
unordered_map<string, vector<string>> follow;
map<int, vector<vector<string>>> productionsMap;
typedef pair<vector<vector<string>>, vector<string>> item;
// in item above: vector<vector<string>> is the production, vector<string> is the lookahead
vector<vector<item>> canonicalItems;
vector<vector<string>> parseTableCLR;
vector<vector<string>> parseTableLALR;
vector<vector<int>> mergeIndex;
set<int> indexSet;
bool isLALR = true;
bool srConflict = false;
int srCount = 0;
bool rrConflict = false;
int rrCount = 0;
int clrConflicts=0;
```

HELPER FUNCTIONS

displayProductions()

Printing the **productions**

```cpp
void displayProductions()
{
    cout << "\nProductions" << endl;
    for (auto &i : productions)
    {
        cout << i.first << "->";
        for (auto &j : i.second)
        {
            for (auto &k : j)
            {
                cout << k << " ";
            }
            cout << "| ";
        }
        cout << endl;
    }
}
```

displayFirst()

Printing the **first set**

```cpp
void displayFirst()
{
    cout << "First Set\n";
    for (auto &i : first)
    {
        cout << i.first << " -> ";
        for (auto &j : i.second)
        {
            cout << j << " ";
        }
        cout << endl;
    }
}
```

displayFollow()

Printing the **follow set**

```cpp
void displayFollow()
{
    cout << "Follow Set\n";
    for (auto &i : follow)
    {
        cout << i.first << " -> ";
        for (auto &j : i.second)
        {
            cout << j << " ";
        }
        cout << endl;
    }
}
```

getProductionsMap()

Adding elements to productionMap and printing it.
In productionsMap, we are mapping each production with an index.

```cpp
void getProductionsMap()
{
    int index = 0;
    for (auto x : productions)
    {
        vector<vector<string>> v;
        v.push_back({x.first}); // pushing the vector with string s

        for (auto i : x.second)
        {
            v.push_back(i);
            productionsMap[index] = v;
            index++;
            v.pop_back();
        }
    }
    for (auto x : productionsMap)
    {
        cout << x.first << "  " << x.second[0][0] << " -> ";
        for (auto i : x.second[1])
        {
            cout << i;
        }
        cout << endl;
    }
}
```

getCLRTable()

Printing all terminals along with “$” and nonterminals along the x-axis except augmented start and state number along the y-axis, with values from 2d vector of strings, parseTableCLR.

```cpp
void getCLRTable()
{
    // CLR parse table
    cout << "CLR Parse Table : " << endl;
    cout << "State ";
    for (auto &i : terminals)
    {
        cout << i << "    ";
    }

    cout << "$    ";
    for (auto &i : nonterminals)
    {
        cout << i << "    ";
    }
    cout << endl;

    for (int i = 0; i < parseTableCLR.size(); i++)
    {
        cout << i << "     ";
        for (int j = 0; j < parseTableCLR[i].size(); j++)
        {
            cout << parseTableCLR[i][j] << "  ";
        }
        cout << endl;
    }
}
```

getLALRTable()

- ➔ Duplicating the CLR parse table and then merging rows whose productions in the reduced state are the same.
- ➔ And then deleting the extra rows in the LALR Table.
- ➔ Printing all terminals along with "$" and nonterminals along the x-axis except augmented start and state number along the y-axis, with values from 2d vector of strings, parseTableLALR.

```
void getLALRTable()
{
    //this vector will help us in printing which states got merged in lalr table
    vector<int> indexLALR;
    int i = 0;
    // duplicating the CLR table to LALR
    for (auto it : parseTableCLR)
    {
        indexLALR.push_back(i);
        parseTableLALR.push_back(it);
        i++;
    }
    //count is basically the number of extra rows in the end of LALR table that needs to be removed
    int count = -mergeIndex.size();
    for (int i = 0; i < mergeIndex.size(); i++)
    {
        int ind = mergeIndex[i][0];
        count += mergeIndex[i].size();
        int temp = indexLALR[ind];
        for (int j = 1; j < mergeIndex[i].size(); j++)
        {
            temp = temp * 10 + mergeIndex[i][j];
            indexLALR.erase(indexLALR.begin() + mergeIndex[i][j]);
            //here we are merging rows which have same production map
            for (int k = 0; k < parseTableLALR[ind].size(); k++)
            {
                //checking for rr conflict we a
                if (parseTableLALR[ind][k][0] == 'r')
                {
                    if (parseTableLALR[mergeIndex[i][j]][k][0] == 'r')
                    {
                        rrConflict = true;
                        rrCount++;
                    }
                    else if (parseTableLALR[mergeIndex[i][j]][k][0] == 's')
                    {
                        srConflict = true;
                        srCount++;
                    }
                    parseTableLALR[ind][k] +=parseTableLALR[mergeIndex[i][j]][k];
                }
```

```cpp
                }
                else if (parseTableLALR[ind][k][0] == 's')//sr conflict check
                {
                    if (parseTableLALR[mergeIndex[i][j]][k][0] == 'r')
                    {
                        srConflict = true;
                        srCount++;
                    }
                    parseTableLALR[ind][k] +=  parseTableLALR[mergeIndex[i][j]][k];
                }
                else
                {
                    parseTableLALR[ind][k] = parseTableLALR[mergeIndex[i][j]][k];
                }
            }
        }
        indexLALR[ind] = temp;
    }
    //popping all the extra layers at the end of LALR table
    while (count--)
        parseTableLALR.pop_back();

    //printing the LALR Table

    cout << "LALR Table : " << endl;
    cout << "State ";
    for (auto &i : terminals)
    {
        cout << i << "    ";
    }

    cout << "$    ";
    for (auto &i : nonterminals)
    {
        cout << i << "    ";
    }
    cout << endl;

    for (int i = 0; i < parseTableLALR.size(); i++)
    {
        cout << indexLALR[i] << "     ";
        for (int j = 0; j < parseTableLALR[i].size(); j++)
        {
            cout << parseTableLALR[i][j] << "  ";
        }
        cout << endl;
    }
```

check()

- This function is to find all the reduced states where production is the same and which indexes can be merged.

```
vector<int> check(item itm, int index)
{
    //check function to get the reduced items which can be merged
    int ind = 0;
    vector<int> v;

    for (auto it1 : canonicalItems)
    {
        if (ind == 0)
        {
            ind++;
            continue;
        }
        for (auto it2 : it1)
        {
            if (it2.first == itm.first)
            {
                v.push_back(ind);
            }
        }
        ind++;
    }
    return v;
}
```

First Sets for Grammar

getFirst()

Rules to compute FIRST set:

- ➔ If x is a terminal, then FIRST(x) = { ‘x’ }
- ➔ If x-> ∈, is a production rule, then add ∈ to FIRST(x).
- ➔ If X->Y1 Y2 Y3....Yn is a production,
    - ◆ FIRST(X) = FIRST(Y1)
    - ◆ If FIRST(Y1) contains ∈ then FIRST(X) = { FIRST(Y1) – ∈ } U { FIRST(Y2) }
    - ◆ If FIRST (Yi) contains ∈ for all i = 1 to n, then add ∈ to FIRST(X).

```
void getFirst(string s)
{
Source Control (Ctrl+Shift+G) = 1;
    for (auto &i : productions[s])
    {
        for (auto &p : i)
        {
            if (flag && find(begin(terminals), end(terminals), p) != terminals.end())
            {
                if (find(begin(first[s]), end(first[s]), p) == first[s].end())
                {
                    first[s].push_back(p);
                }
                flag = 0;
                break;
            }
            else if (flag && find(begin(nonterminals), end(nonterminals), p) != nonterminals.end())
            {
                flag = 0;
                if (first.find(p) == first.end())
                {
                    getFirst(p);
                }
                for (auto &k : first[p])
                {
                    if (k != string(1, 'e') && find(begin(first[s]), end(first[s]), k) == first[s].end())
                    {
                        first[s].push_back(k);
                    }
                    if (k == string(1, 'e'))
                    {
                        flag = 1;
                    }
                }
                if (!flag)
                {
                    break;
                }
            }
            else
            {
                if (find(begin(first[s]), end(first[s]), p) == first[s].end())
                {
                    first[s].push_back(p);
                }
            }
        }
        if (flag)
        {
            if (find(begin(first[s]), end(first[s]), string(1, 'e')) == first[s].end())
            {
                first[s].push_back(string(1, 'e'));
            }
        }
    }
}
```

- ➔ This function helps us to get the **first** value for a **string s** which is passed as a parameter.

getFollow()

Rules to compute FOLLOW set:

- ➔ FOLLOW(S) = { $ }   // where S is the starting Non-Terminal
- ➔ If A -> pBq is a production, where p, B, and q are any grammar symbols, then everything in FIRST(q) except Є is in FOLLOW(B).
- ➔ If A->pB is a production, then everything in FOLLOW(A) is in FOLLOW(B).
- ➔ If A->pBq is a production and FIRST(q) contains Є,
- ➔   then FOLLOW(B) contains { FIRST(q) – Є } U FOLLOW(A)

```
void getFollow(string s)
{
    bool flag = 0;
    for (auto x : productions)
    {
        for (auto &i : x.second)
        {
            for (auto &p : i)
            {
                if (p == s)
                {
                    flag = 1;
                    continue;
                }
                if (flag && find(begin(nonterminals), end(nonterminals), p) != nonterminals.end())
                {
                    flag = 0;
                    for (auto k : first[p])
                    {
                        if (k != string(1, 'e'))
                        {
                            if (find(begin(follow[s]), end(follow[s]), k) == follow[s].end())
                            {
                                follow[s].push_back(k);
                            }
                        }
                        else
                        {
                            flag = 1;
                        }
                    }
                }
```

```
            else if (flag && find(begin(terminals), end(terminals), p) != terminals.end())
            {
                if (find(begin(follow[s]), end(follow[s]), p) == follow[s].end())
                {
                    follow[s].push_back(p);
                }
                flag = 0;
            }
        }
        if (flag)
        {
            if (follow.find(x.first) == follow.end())
            {
                getFollow(x.first);
            }
            for (auto k : follow[x.first])
            {
                if (find(begin(follow[s]), end(follow[s]), k) == follow[s].end())
                {
                    follow[s].push_back(k);
                }
            }
        }
      }
    }
}
```

- ➔ Function to calculate the **follow** value for a **string s.**
- ➔ Calculating **follow** for all the nonterminals, by running a loop in main().

getClosure()

```
string getClosure(vector<item> it)
{
    vector<item> closure;
    int row = canonicalItems.size();
    // initializing the rows of parse table with a vector of size nonterminals.size()+terminals.size()
    parseTableCLR.push_back(vector<string>(nonterminals.size() + terminals.size(), "   "));
    for (auto itr : it)
    {
        // storing production map in p
        vector<vector<string>> p = itr.first;
        // getting lookaheads from item
        vector<string> lookaheads = itr.second;
        // symbol is a start state string
        string symbol = p[0][0];
        // setting production to the starting productio
        // at index 0 of p S' is there so setting production to (S)
        vector<string> production = p[1];
        closure.push_back(itr);
        queue<vector<vector<string>>>q; //if nonterminal to expand has nonterminals as start of its production
        q.push(p);
        while(!q.empty()){
            vector<vector<string>> ele=q.front();
            production=ele[1];
            q.pop();
            int i = 0;
            while (production[i] != ".")
            {
                i++;
            }
            i++;
```

- ➔ Get vector<item> as a parameter.
- ➔ **Row** stores which state we are calculating.
- ➔ Pushing new row into parseTable with number of columns = **terminals size+ nonterminals size.**
- ➔ Iterating through all items received as parameters.
- ➔ The queue is maintained as there might be nonterminals present after the **"."** which has to be expanded in a **dfs(Depth First Search)** way.
- ➔ Iterating through each production till **"."** is encountered.

```cpp
// if "." is at the end or  no "." found
if (i == production.size()){
    // if last string in non terminals is equals to symbol then the state is accepted
    if (nonterminals.back() == symbol){
        parseTableCLR[canonicalItems.size()][terminals.size()] = "acc"; // acc is accepted state
    }else{
        // finding which production to reduce to
        int pos = -1;
        production.pop_back();
        for (auto itr2 : productionsMap){
            // setting the pos
            if (itr2.second[0][0] == symbol && itr2.second[1] == production){
                pos = itr2.first;
            }
        }
        if(pos!=-1){
            for (auto itr : lookaheads){
                // if lookahead is "$" then pos will be same as calculated above.
                // else pos will be calculated for the lookahead and added to the table
                int posTerminal;
                        if(itr=="$"){
                            posTerminal=terminals.size();
                        }else{
                            posTerminal=find(terminals.begin(), terminals.end(), itr)-terminals.begin();
                        }
                        if(parseTableCLR[row][posTerminal]=="   " || parseTableCLR[row][posTerminal]=="r"+to_string(pos)+" ")
                            parseTableCLR[row][posTerminal]="r"+to_string(pos)+" ";
                        else
                            clrConflicts++;
            }
```

- ➔ If **"."** is at the end of the production check if the LHS symbol is the augmented start and add **"acc"** to the **$** column of that row.
- ➔ The reduced state should be added to all columns whose symbols are in the lookahead set if it is not the augmented start symbol.
- ➔ To begin, determine which production number it has to be reduced to.
- ➔ Then respectively added to all the columns whose symbols are in the lookahead set.
- ➔ If the element is already full while adding to the table, conflicts have formed; therefore, increasing the conflict count.

```cpp
else
{
    // checking if next index of production has a terminal
    if (find(terminals.begin(), terminals.end(), production[i + 1]) != terminals.end())
    {
        // if it is a terminal then we will add it to st
        if (find(st.begin(), st.end(), production[i + 1]) == st.end())
        {
            st.push_back(production[i + 1]);
        }
    }
    else
    { // if next index is non terminal
        int flag = 0;
        // as we need to get the first of non-terminals
        vector<string> st2 = first[production[i + 1]];
        for (auto itr : st2)
        {
            // if first is not an epsilon then push it to st
            if (itr != "e")
            {
                if (find(st.begin(), st.end(), itr) == st.end())
                {
                    st.push_back(itr);
                }
            }
            else
            { // first is epsilon
                flag = 1;
            }
        }
        // if we get an epsilon then we have to check for (i+2)th production
        int j = i + 2;
```

- ➔ If **"."** is not at the end of the production, should check if the symbol next to the **"."** is a nonterminal.
- ➔ If it is a nonterminal, should expand and add the productions of the nonterminal to the closure.
- ➔ If epsilon is part of the closure, just add **"."** as production and check if the next symbol is nonterminal and expand in a **dfs(Depth First Search)** way.

```
        while (flag && j < production.size())
        {
            flag = 0;
            // again getting the first for (i+2)th production and so on
            st2 = first[production[j]];
            for (auto itr : st2)
            {
                // if first is not an epsilon then push it to st
                if (itr != "e")
                {
                    if (find(st.begin(), st.end(), itr) == st.end())
                    {
                        st.push_back(itr);
                    }
                }
                else
                {
                    // first is episilon continue the loop till we get a non epsilon or production ends
                    flag = 1;
                }
            }
        }

        if (flag)
        {
            // if an epsilon was encountered then we will push lookaheads to st.
            for (auto itr : lookaheads)
            {
                if (find(st.begin(), st.end(), itr) == st.end())
                {
                    st.push_back(itr);
                }
            }
        }
    }
}

// iterating through productions with ith string of production
for (auto it2 : productions[production[i]])
{
    vector<string> v2 = it2;
    v.clear();
```

- ➔ The lookaheads of these productions are also calculated at the same time.
- ➔ If the symbol following the expanded nonterminal is a terminal, add it to the lookahead set.
- ➔ If it is a nonterminal, add the first set of that nonterminal as the lookaheads.
- ➔ If there is **"e" (epsilon)** in the first set, add the first set of the next nonterminal.
- ➔ Continue this until no epsilon is found or the end of production is reached.
- ➔ If the end is reached and the flag is still true, add all the lookaheads of the initial production also as the lookaheads.

```
                    v.push_back(".");
                    // here we are storing the production in v
                    for (auto it3 : v2)
                    {
                        if (it3 != "e")
                            v.push_back(it3);
                    }
                    temp.push_back(v);
                    // here we made a item with the production and lookaheads and pushed it to closure
                    closure.push_back(make_pair(temp, st));
                    temp.pop_back();
                }
            }
        }
    }
}

string returnString = "";

// HERE WE ARE DECIDING THE SHIFT STATES
// here we are checking if closure items are already present in canonical items or not


if (find(canonicalItems.begin(), canonicalItems.end(), closure) == canonicalItems.end())
{
    canonicalItems.push_back(closure);
    // setting the shift state to 1+ to number of items we have
    //  so row is the size of vector of canonical items.
    returnString = "s" + to_string(row) + " ";
}
else
{
    // if closure is already present in canonical items then we will set the shift state to the index of closure
    int pos = find(canonicalItems.begin(), canonicalItems.end(), closure) - canonicalItems.begin();
    parseTableCLR.pop_back(); // no need of row which we inserted at start
    return "s" + to_string(pos) + " ";
}
```

- ➔ All the productions of the nonterminals are added to the closure.
- ➔ Checking if the created canonical item is already an existing canonical item, find the position of the canonical item in the vector, and return it.
- ➔ If it is a new canonical item, push it into the vector of canonical items and call function which will calculate the next closure.
- ➔ Else find and return the state. (returning shift state)

```
// dfs to getClosure of next elements
set<string> symbols; // to store symbols where we will call dfs
vector<item> items;
// here we are getting the items from closure
for (auto it2 = closure.begin(); it2 != closure.end(); it2++)
{
    item itm = *it2;
    vector<vector<string>> temp;
    temp.push_back(itm.first[0]);
    vector<string> p2 = itm.first[1];
    int i = 0;
    while (p2[i] != ".")
    {
        i++;
    }

    if (p2.size() == 1)
    { // case of S->e (where e is epsilon)
        vector<string> st = itm.second;
        temp.push_back(p2);
        temp[1][0] = "e";
        int posOfProduction = -1;
        for (auto itr2 : productionsMap)
        {
            // if production is found then we will set the posOfProduction to the ley of the productionsMap
            if (itr2.second[0][0] == temp[0][0] && itr2.second[1] == temp[1])
            {
                posOfProduction = itr2.first;
            }
        }
        for (auto itr : st)
        {
            // getting the index of itr from terminals
            int pos = find(terminals.begin(), terminals.end(), itr) - terminals.begin();
            // setting the reduced state to the index of the itr
            if(parseTableCLR[row][pos]=="   "|| parseTableCLR[row][pos]=="r"+to_string(posOfProduction)+" "){
                    parseTableCLR[row][pos]="r"+to_string(posOfProduction)+" ";
            }else{//if already filled increase conflict
                clrConflicts++;
            }
        }
    }
```

- ➔ Closure contains all the items in the closure.
- ➔ Iterate through every item and iterate in the production till **"."**
- ➔ If the production is just **-> "e"**, find the index of the production.
- ➔ To the columns whose symbols are in the lookahead set, add the reduced state to the table.

```cpp
else if (i < p2.size() - 1 && symbols.find(p2[i + 1]) == symbols.end())
{
    // if lookahead is not present in symbols then we will add it to symbols
    items.clear();
    p2[i] = p2[i + 1];
    p2[i + 1] = ".";
    string sym = p2[i];
    // adding it to the symbol
    symbols.insert(sym);
    temp.push_back(p2);
    itm = make_pair(temp, itm.second);
    items.push_back(itm);
    // this loop is to check if multiple productions can transition on the same symbol to diff state
    for (auto it3 = it2 + 1; it3 != closure.end(); it3++)
    {
        item itm2 = *it3;
        temp.clear();
        temp.push_back(itm2.first[0]);
        vector<string> p3 = itm2.first[1];
        int j = 0;
        while (p3[j] != ".")
        {
            j++;
        }
        if (p3[j + 1] == sym)
        {
            p3[j] = p3[j + 1];
            p3[j + 1] = ".";
            temp.push_back(p3);
            itm2 = make_pair(temp, itm2.second);
            items.push_back(itm2);
        }
    }
```

- ➔ Find the symbol after **"."** to determine which symbol will transition to the next state, then insert it into the set of symbols so we can see which symbol transition has already occurred.
- ➔ More than one production may transition on the same symbols.
- ➔ As a result, locate all such productions and add them to items, which is a vector of items.

```
            // callig the getClosure again with the items
            string ret = getClosure(items);
            // if symbol sym is present in terminals
            // set the reduced value(r4,r5 etc to the parse table)
            if (find(terminals.begin(), terminals.end(), sym) != terminals.end())
            {
                int pos = find(terminals.begin(), terminals.end(), sym) - terminals.begin();

                if(parseTableCLR[row][pos]=="    "|| parseTableCLR[row][pos]==ret){
                    parseTableCLR[row][pos]=ret;
                }else{
                    clrConflicts++;
                }
            }
            else
            {
                // if it is not in terminals then find the index in non-terminals
                // and "ret" to parseTableCLR
                int pos = find(nonterminals.begin(), nonterminals.end(), sym) - nonterminals.begin();
                ret.erase(0, 1);
                if(parseTableCLR[row][terminals.size()+1+pos]=="    "||parseTableCLR[row][terminals.size()+1+pos]==ret)
                    parseTableCLR[row][terminals.size()+1+pos]=ret;
                else
                    clrConflicts++;
            }
        }
    }
    return returnString;
}
```

- ➔ Call the getClosure() function by sending the vector of items as an argument.
- ➔ Get the returned state from the function and add it to the parse table in the column of that symbol where the transition has taken place.
- ➔ Also, **return the canonical item number**.

lalrParser

```
void lalrParser()
{
    int index = 0;
    vector<int> v;
    for (auto it1 : canonicalItems)
    {
        //checking if index is already present in this set ot not
        if (indexSet.find(index) == indexSet.end())
        {
            index++;
            continue;
        }
        for (auto it2 : it1)
        {
            //calling the check function to check if any other reduced state has same production or not
            v = check(it2, index);
            if (v.size() > 1)
            {
                sort(v.begin(), v.end());
                //mergeIndex is basically a vector of vector where every vector contains indexes that has to be merged
                if (find(mergeIndex.begin(), mergeIndex.end(), v) == mergeIndex.end())
                {
                    mergeIndex.push_back(v);
                }
            }
        }
        index++;
    }
}
```

- indexSet has indexes that are in the reduced state, so we are calling the check function only for reduced states.
- After check function, we are checking if we got anything in the vector, if yes then we are checking if this vector already exists in the mergexIndex matrix or not.
- If this array is not there then you can add it to the mergeIndex.

main() function

Getting input from a file “inpt.txt”

```cpp
int main()
{
    int i, j;
    ifstream file;
    file.open("input.txt");
    string line;
    if (file.is_open())
    {
        while (getline(file, line))
        {
            if (line.substr(0, 9) == "TERMINALS")
            {
                string temp;
                for (i = 10; i < line.size(); i++)
                {
                    if (line[i] == ' ')
                    {
                        terminals.push_back(temp);
                        temp.clear();
                    }
                    else
                        temp += line[i];
                }
                terminals.push_back(temp);
            }
```

- ➔ Here, we are storing all the terminals in a vector called **terminals**

```
            else if (line.substr(0, 12) == "NONTERMINALS")
            {
                string temp;
                for (i = 13; i < line.size(); i++)
                {
                    if (line[i] == ' ')
                    {
                        nonterminals.push_back(temp);
                        temp.clear();
                    }
                    else
                        temp += line[i];
                }
                nonterminals.push_back(temp);
            }
```

➔ Here, we are storing all the non-terminals in a vector called **nonterminals**

```
            else
            {
                vector<string> prod;
                string temp = line.substr(5); // a -> b (so b is 5th index)
                // int i=0;
                for (auto i : temp)
                {
                    if (i != ' ')
                        prod.push_back(string(1, i)); // pushing the string into prod vector
                    // string(1,i) is used to convert char to string i.e. 'a' to "a"
                    // string(1,i) means 1 times i in a string
                }
                // pushing the production into productions map
                // and using .substr function because our map takes string as key not char
                productions[line.substr(0, 1)].push_back(prod);
            }
        }
    }

    file.close();
    // follow of first state = $
    follow[nonterminals[0]].push_back(string(1, '$'));
```

➔ Here, we have map **productions**, where the key is a string which is basically a non-terminal. Value for these productions is a vector<vector<string>> which will store the production values.
➔ After getting the required things from the input file, we are closing the file.
➔ As we know the follow of starting state is **"$"**, so we are setting it to **"$".**

```cpp
    // displaying terminals and non-terminals
    cout << "Terminals: \n";
    for (auto &i : terminals)
    {
        cout << i << " ";
    }

    cout << "\nNon Terminals" << endl;
    for (auto &i : nonterminals)
    {
        cout << i << " ";
    }

    // displaying productions
    displayProductions();

    // calling getFirst function, to get the first values of all nonterminals
    for (auto &i : nonterminals)
    {
        getFirst(i);
    }

    displayFirst();

    // calling getFollow function, to get the follow values of all nonterminals
    for (auto &i : nonterminals)
    {
        getFollow(i);
    }
```

```cpp
    displayFollow();

    // Augmenting grammar
    string firstNT = nonterminals[0];
    string augmentStart = firstNT + "'";                        // to make S to S'
    nonterminals.push_back(augmentStart);                       // adding S' to nonterminals
    productions[augmentStart].push_back(vector<string>()); // adding empty string to  make :S'->
    productions[augmentStart][0].push_back(firstNT);        // adding S to S'

    // displaying augmented productions
    cout << "\n\nAugmented grammar productions : " << endl;
    getProductionsMap();

    vector<vector<string>> temp;
    temp.push_back({augmentStart});
    temp.push_back({".", firstNT});

    vector<string> st = {"$"};
    // item it=make_pair(temp,st);
    vector<item> items;
    //PUSHING S'-> .S to items
    items.push_back(make_pair(temp, st));

    getClosure(items);

    cout << "\n\n";

    //to print CLR table
    getCLRTable();
```

```cpp
//Printing all the items
cout << "Canonical Items : \n";
int index = 0;
for (auto it1 : canonicalItems)
{
    cout << "State " << index << " \n";
    string prev;
    bool flag = true;
    for (auto it2 : it1)
    {
        for (auto it3 : it2.first)
        {
            for (auto it4 : it3)
            {
                cout << it4 << " ";
                prev = it4;
            }
        }
        cout << ":";
        for (auto it4 : it2.second)
        {
            cout << it4 << " ";
        }
        cout << "\n";
        if (prev != ".")
        {
            flag = false;
        }
        // cout<<index<<" "<<prev<<"\n";
    }
    if (flag)
    {
        indexSet.insert(index);
    }
    index++;
}
```

```cpp
    //finding reduced states which can be merged
    lalrParser();

    cout << "\n\n";

    //converting CLR table to LALR table
    getLALRTable();

    cout << "\n\n";

    if(clrConflicts>0){
        cout<<"\nThis is not a CLR Grammar as there are "<<clrConflicts<<" conflicts\n\n";
        cout << "Therefore, The given grammar is not LALR\n";
    }
    else{
        cout << "Number of S R Conflicts: " << srCount << endl;
        cout << "Number of R R Conflicts: " << rrCount << endl;

        if (!srConflict && !rrConflict)
            cout << "The given grammar is LALR\n";
        else
            cout << "The given grammar is not LALR\n";
    }
    return 0;
}
```

Execution Screenshots

Example 1 (LALR Grammar)

INPUT

```
A2 > input.txt
TERMINALS ( )
NONTERMINALS S
S -> ( S )
S -> e
```

TERMINALS |NON TERMINALS ||PRODUCTIONS
FIRST SET || FOLLOW SET
AUGMENTED GRAMMAR with INDICES

```
Terminals:
( )
Non Terminals
S
Productions
S->( S ) | e |
First Set
S -> ( e
Follow Set
S -> $ )


Augmented grammar productions :
0  S' -> S
1  S -> (S)
2  S -> e
```

CLR PARSE TABLE

```
CLR Parse Table :
State (    )    $    S    S'
0     s2        r2   1
1               acc
2     s5   r2        3
3          s4
4               r1
5     s5   r2        6
6          s7
7          r1
```

CANONICAL ITEMS

```
Canonical Items :
State 0
S' . S :$
S . ( S ) :$
S . :$
State 1
S' S . :$
State 2
S ( . S ) :$
S . ( S ) :)
S . :)
State 3
S ( S . ) :$
State 4
S ( S ) . :$
State 5
S ( . S ) :)
S . ( S ) :)
S . :)
State 6
S ( S . ) :)
State 7
S ( S ) . :)
```

LALR TABLE

```
LALR Table :
State (    )    $    S    S'
0     s2        r2   1
1               acc
2     s5   r2        3
3          s4
47          r1   r1
5     s5   r2        6
6          s7


Number of S R Conflicts: 0
Number of R R Conflicts: 0
The given grammar is LALR

[Done] exited with code=0 in
```

Example 2 (Not LALR Grammar)

INPUT

```
TERMINALS a b c d e
NONTERMINALS S A B
S -> aAd
S -> bBd
S -> aBe
S -> bAe
A -> c
B -> c
```

TERMINALS |NON TERMINALS ||PRODUCTIONS
FIRST SET || FOLLOW SET

```
Terminals:
a b c d e
Non Terminals
S A B
Productions
B->c |
A->c |
S->a A d | b B d | a B e | b A e |
First Set
B -> c
A -> c
S -> a b B d e A
Follow Set
B -> d e
A -> d e
S -> $
```

AUGMENTED GRAMMAR with INDICES

```
Augmented grammar productions :
0  S' -> S
1  B -> c
2  A -> c
3  S -> aAd
4  S -> bBd
5  S -> aBe
6  S -> bAe
```

CLR PARSE TABLE

```
CLR Parse Table :
State a    b    c    d    e    $    S    A    B    S'
0     s2   s7                       1
1                              acc
2               s5                       3   6
3                    s4
4                              r3
5                    r2        r1
6
7               s10                       11   8
8                    s9
9                              r4
10                    r1        r2
11
```

CANONICAL ITEMS

```
Canonical Items :
Run and Debug (Ctrl+Shift+
S' . S :$
S . a A d :$
S . b B d :$
S . a B :$
S . b A :$
State 1
S' S . :$
State 2
S a . A d :$
A . c :d
S a . B :$
B . c :$
State 3
S a A . d :$
State 4
S a A d . :$
State 5
A c . :d
B c . :$
State 6
S a B . :$
```

```
State 7
S b . B d :$
B . c :d
S b . A :$
A . c :$
State 8
S b B . d :$
State 9
S b B d . :$
State 10
B c . :d
A c . :$
State 11
S b A . :$
```

LALR TABLE

```
LALR Table :
State a    b    c    d    e    $    S    A    B    S'
0     s2   s7                       1
1                              acc
2               s5                       3   6
3                    s4
4                              r3
60                    r2 r1        r1 r2
6
7               s10                       11   8
8                    s9
9                              r4
11                    r1         r2


Number of S R Conflicts: 0
Number of R R Conflicts: 2
The given grammar is not LALR
```