LAB ASSIGNMENT II

Design a SLR parser for the grammar given below:

E→E+T/T
T→T*F/F
F→(E)/id
This will involve three steps:
Generate the Set of Items (5 Marks) (3 Lab of 2 Hrs.)
Generate the Action and GOTO table (5 marks) (3 Labs of 2 Hrs. each)

OR

Design a SLR parser for the any grammar (Generic)

General SLR Parsing Table code with input example as:
E->E+T/T

T->T*F/F

F->(E)/i

```
#include <iostream>

#include <vector>

#include <map>

#include <set>

#include <string>

#include <sstream>

#include <algorithm>

#include <bits/stdc++.h>

using namespace std;

class SLRParser {

public:

struct Production {

char left;

string right;

};

map<char, vector<string>> productions;

vector<Production> productionRules; // Vector to store productions with rule numbers

map<char, set<char>> first, follow;

vector<vector<pair<char, string>>> itemsets;

map<pair<int, char>, int> gotoTable;

map<pair<int, char>, string> actionTable;

vector<char> terminals, nonTerminals;

char startSymbol;

void computeFirst(char symbol, set<char>& firstSet);

void computeFollow(char symbol, set<char>& followSet, set<char>& processed);

vector<pair<char, string>> closure(const vector<pair<char, string>>& kernel);

void buildItemSets();
```

```
void buildParsingTable();
SLRParser(const vector<Production>& prods, char start);
void buildParser();
void displayResults() const;
};
SLRParser::SLRParser(const vector<Production>& prods, char start) : startSymbol(start) {
int ruleNumber = 1; // Rule numbers start at 1
for (const auto& prod : prods) {
productions[prod.left].push_back(prod.right);
productionRules.push_back(prod); // Store the production with its number
if (find(nonTerminals.begin(), nonTerminals.end(), prod.left) == nonTerminals.end()) {
nonTerminals.push_back(prod.left);
}
for (char c : prod.right) {
if (isupper(c)) {
if (find(nonTerminals.begin(), nonTerminals.end(), c) == nonTerminals.end()) {
nonTerminals.push_back(c);
}
} else if (find(terminals.begin(), terminals.end(), c) == terminals.end()) {
terminals.push_back(c);
}
}
}
terminals.push_back('$'); // Add end-of-input marker
}
void SLRParser::computeFirst(char symbol, set<char>& firstSet) {
if (first.find(symbol) != first.end()) {
firstSet.insert(first[symbol].begin(), first[symbol].end());
return;
}
if (find(terminals.begin(), terminals.end(), symbol) != terminals.end()) {
firstSet.insert(symbol);
return;
}
for (const auto& rhs : productions[symbol]) {
if(rhs[0] == symbol) continue;
bool allDeriveEpsilon = true;
for (char c : rhs) {
set<char> tempFirst;
```

```
computeFirst(c, tempFirst);
firstSet.insert(tempFirst.begin(), tempFirst.end());
if (tempFirst.find('e') == tempFirst.end()) {
allDeriveEpsilon = false;
break;
}
}
if (allDeriveEpsilon) {
firstSet.insert('e');
}
}
first[symbol] = firstSet;
}
void SLRParser::computeFollow(char symbol, set<char>& followSet, set<char>& processed) {
if (processed.find(symbol) != processed.end()) return;
processed.insert(symbol);
if (symbol == startSymbol) {
followSet.insert('$');
}
for (const auto& [left, rights] : productions) {
for (const auto& right : rights) {
auto pos = right.find(symbol);
if (pos != string::npos) {
if (pos == right.length() - 1) {
if (left != symbol) {
set<char> tempFollow;
computeFollow(left, tempFollow, processed);
followSet.insert(tempFollow.begin(), tempFollow.end());
}
} else {
set<char> tempFirst;
computeFirst(right[pos + 1], tempFirst);
followSet.insert(tempFirst.begin(), tempFirst.end());
if (tempFirst.find('e') != tempFirst.end()) {
if(left != symbol){
set<char> tempFollow;
computeFollow(left, tempFollow, processed);
followSet.insert(tempFollow.begin(), tempFollow.end());
}
```

```
}
}
}
}
}
follow[symbol] = followSet;
}
vector<pair<char, string>> SLRParser::closure(const vector<pair<char, string>>& kernel) {
vector<pair<char, string>> result = kernel;
bool changed;
do {
changed = false;
vector<pair<char, string>> newItems;
for (const auto& [left, right] : result) {
auto dotPos = right.find('.');
if (dotPos != right.length() - 1) {
char nextSymbol = right[dotPos + 1];
if (isupper(nextSymbol)) {
for (const auto& prod : productions[nextSymbol]) {
pair<char, string> newItem = {nextSymbol, "." + prod};
if (find(result.begin(), result.end(), newItem) == result.end() && find(newItems.begin(), newItems.end(), newItem) == newItems.end()) {
newItems.push_back(newItem);
changed = true;
}
}
}
}
}
result.insert(result.end(), newItems.begin(), newItems.end());
} while (changed);
return result;
}
void SLRParser::buildItemSets() {
vector<pair<char, string>> initialItem = {{startSymbol, "." + productions[startSymbol][0]}};
itemsets.push_back(closure(initialItem));
for (size_t i = 0; i < itemsets.size(); i++) {
map<char, std::vector<pair<char, std::string>>> symbolGroups;
for (const auto& [left, right] : itemsets[i]) {
auto dotPos = right.find('.');
```

```
if (dotPos != right.length() - 1) {
char nextSymbol = right[dotPos + 1];
string newRight = right;
swap(newRight[dotPos], newRight[dotPos + 1]);
symbolGroups[nextSymbol].push_back({left, newRight});
}
}
for (const auto& [symbol, group] : symbolGroups) {
auto newItemset = closure(group);
auto it = find(itemsets.begin(), itemsets.end(), newItemset);
if (it == itemsets.end()) {
gotoTable[{i, symbol}] = itemsets.size();
itemsets.push_back(newItemset);
} else {
gotoTable[{i, symbol}] = distance(itemsets.begin(), it);
}
}
}
}
void SLRParser::buildParsingTable() {
for (size_t i = 0; i < itemsets.size(); i++) {
for (const auto& [left, right] : itemsets[i]) {
auto dotPos = right.find('.');
if (dotPos == right.length() - 1) { // Reduction case
if (left == startSymbol && right == productions[startSymbol][0] + ".") {
actionTable[{i, '$'}] = "acc";
} else {
string productionRight = right.substr(0, right.length() - 1);
int ruleNumber = -1;
// Find the rule number
for (size_t j = 0; j < productionRules.size(); j++) {
if (productionRules[j].left == left && productionRules[j].right == productionRight) {
ruleNumber = j + 1;
break;
}
}
for (char terminal : follow[left]) {
actionTable[{i, terminal}] = "r" + to_string(ruleNumber); // Use rule number
}
```

```
}
} else { // Shift case
char nextSymbol = right[dotPos + 1];
if (find(terminals.begin(), terminals.end(), nextSymbol) != terminals.end()) {
stringstream ss;
ss << "s" << gotoTable[{i, nextSymbol}];
actionTable[{i, nextSymbol}] = ss.str();
}
}
}
for (char nonTerminal : nonTerminals) {
auto it = gotoTable.find({i, nonTerminal});
if (it != gotoTable.end()) {
stringstream ss;
ss << it->second;
actionTable[{i, nonTerminal}] = ss.str();
}
}
}
}
void SLRParser::buildParser() {
for (char nonTerminal : nonTerminals) {
set<char> firstSet;
computeFirst(nonTerminal, firstSet);
}
for (char nonTerminal : nonTerminals) {
set<char> followSet, processed;
computeFollow(nonTerminal, followSet, processed);
}
buildItemSets();
buildParsingTable();
}
void SLRParser::displayResults() const {
cout << "FIRST sets:\n";
for (const auto& [symbol, set] : first) {
cout << symbol << ": {";
for (auto it = set.begin(); it != set.end(); ++it) {
if (it != set.begin()) std::cout << ", ";
cout << *it;
```

```
}
cout << "}\n";
}
cout << "\nFOLLOW sets:\n";
for (const auto& [symbol, set] : follow) {
cout << symbol << ": {";
for (auto it = set.begin(); it != set.end(); ++it) {
if (it != set.begin()) cout << ", ";
cout << *it;
}
cout << "}\n";
}
cout << "\nItem Sets:\n";
for (size_t i = 0; i < itemsets.size(); i++) {
cout << "I" << i << ":\n";
for (const auto& [left, right] : itemsets[i]) {
cout << " " << left << " -> " << right << "\n";
}
cout << "\n";
}
cout << "Parsing Table:\n";
cout << " ";
for (char terminal : terminals) cout << terminal << "\t";
for (char nonTerminal : nonTerminals) cout << nonTerminal << "\t";
cout << "\n";
for (size_t i = 0; i < itemsets.size(); i++) {
cout << i << " ";
for (char symbol : terminals) {
auto it = actionTable.find({i, symbol});
if (it != actionTable.end()) {
for(const auto &action : it->second){
cout << action << " ";
}
}
cout << "\t";
}
for (char symbol : nonTerminals) {
auto it = actionTable.find({i, symbol});
if (it != actionTable.end()) {
```

```
for(const auto &action : it->second){
cout << action << " ";
}
}
cout << "\t";
}
cout << "\n";
}
}
int main() {
vector<SLRParser::Production> productions = {
{'E', "E+T"},
{'E', "T"},
{'T', "T*F"},
{'T', "F"},
{'F', "(E)"},
{'F', "i"}
};
productions.insert(productions.begin(), {'A', "E"});
SLRParser parser(productions, 'A');
parser.buildParser();
parser.displayResults();
return 0;
}
```

```
FIRST sets:
A: {(, i}
E: {(, i}
F: {(, i}
T: {(, i}
FOLLOW sets:
A: {$}
E: {$, ), +}
F: {$, ), *, +}
T: {$, ), *, +}
```

```
Item Sets:
I0:
 A -> .E
 E -> .E+T
 E -> .T
 T -> .T*F
 T -> .F
 F -> .(E)
 F -> .i

I1:
 F -> (.E)
 E -> .E+T
 E -> .T
 T -> .T*F
 T -> .F
 F -> .(E)
 F -> .i

I2:
 A -> E.
 E -> E.+T

I3:
 T -> F.
```

```
I4:
 E -> T.
 T -> T.*F

I5:
 F -> i.

I6:
 F -> (E.)
 E -> E.+T

I7:
 E -> E+.T
 T -> .T*F
 T -> .F
 F -> .(E)
 F -> .i

I8:
 T -> T*.F
 F -> .(E)
 F -> .i

I9:
 F -> (E).

I10:
 E -> E+T.
 T -> T.*F
```

```
I11:
 T -> T*F.
```

Parsing Table:

	+	*	(	)	i	$	A	E	T	F
0			s 1			s 5			2	4	3
1			s 1			s 5			6	4	3
2	s 7						a	c	c		
3	r 5		r 5			r 5			r 5		
4	r 3		s 8			r 3			r 3		
5	r 7		r 7			r 7			r 7		
6	s 7				s 9						
7			s 1			s 5			1	0	3
8			s 1			s 5				1	1
9	r 6		r 6			r 6			r 6		
10	r 2		s 8			r 2			r 2		
11	r 4		r 4			r 4			r 4		