Practical 4

Name: Tushar Harsora

Roll No.: 19BCE509

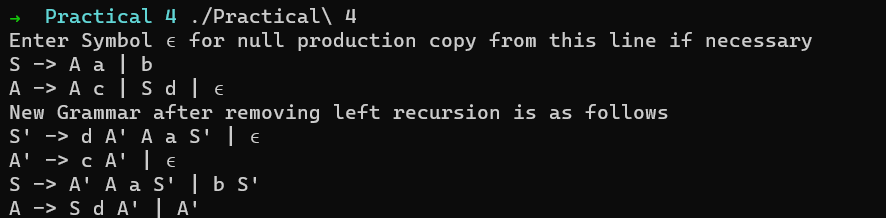
# Explanation

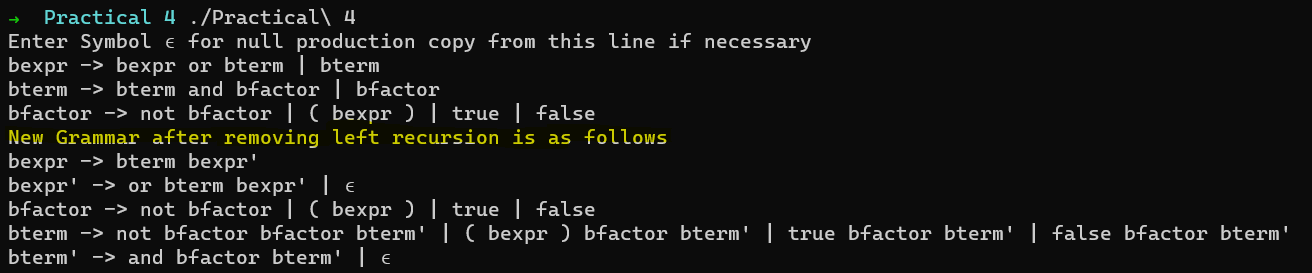
The C++ code is written to eliminate Right Recursion. This code supports **Multiple length identifiers.** The program expects each production in new line and every identifier has to be space separated because space means new identifier is started.

# Code

|  |
| --- |
| #include <iostream>  #include <vector>  #include <string>  #include <map>  #include <unordered\_set>  #include <set>  #include <cassert>  #include <algorithm>  using namespace std;  // trim from start (in place)  static inline void ltrim(std::string& s) {  s.erase(s.begin(), std::find\_if(s.begin(), s.end(), [](unsigned char ch) {  return !std::isspace(ch);  }));  }  // trim from end (in place)  static inline void rtrim(std::string& s) {  s.erase(std::find\_if(s.rbegin(), s.rend(), [](unsigned char ch) {  return !std::isspace(ch);  }).base(), s.end());  }  // trim from both ends (in place)  static inline void trim(std::string& s) {  ltrim(s);  rtrim(s);  }  struct Term {  Term(string s) {  ltrim(s);  while (s.find(" ") != string::npos) {  string id = s.substr(0, s.find(' '));  trim(id);  ids.push\_back(id);  s.erase(0, s.find(" ") + 1);  }  trim(s);  ids.push\_back(s);  }  string to\_string(int pos = 0) const {  string s;  for(; pos < ids.size(); pos++){  s += ids[pos] + " ";  }  s.pop\_back();  assert(s.back() != ' ');  return s;  }  vector<string> ids;  };  struct Production {  public:  Production(string prod) {  const string& str = prod;  auto ind = str.find("->");  if (ind == string::npos) {  cout << "Production " << prod << " is Invalid\n";  exit(0);  }  string non\_terminal = str.substr(0, ind);  trim(non\_terminal);    lhs = non\_terminal;  string rhs = str.substr(ind + 2); // -> skip 2 chars for arrow  vector<string> curr\_prod;  while (rhs.find('|') != string::npos) {  string production = rhs.substr(0, rhs.find('|'));  trim(production);  rhs.erase(0, rhs.find('|') + 1);  Term term(production);  terms.push\_back(term);  }  trim(rhs);  terms.push\_back(Term(rhs));  }  public:  string lhs;  vector<Term> terms;  };  struct Grammar {  public:  Grammar(const vector<string>& input) {  for(int i = 0; i < input.size(); i++) {  Production p(input[i]);  prods.emplace(p.lhs, p);  //prods[p.lhs] = p;  non\_terminals.insert(p.lhs);  if (i == 0) follows[p.lhs].insert("$");  }  for (const auto& prod : prods) {  for (const auto& term : prod.second.terms) {  for (const auto& id : term.ids) {  if (non\_terminals.find(id) == non\_terminals.end()) {  terminals.insert(id);  }  }  }  }  }  void remove\_left\_recursion() {  vector<Production> new\_productions;  for(const auto& pr : prods)  new\_productions.push\_back(pr.second);  for(int i = 0; i < new\_productions.size(); i++){  for(int j = 0; j < i; j++){  vector<Term> new\_rhs;  for(int k = 0; k < new\_productions[i].terms.size(); k++){  if(new\_productions[i].terms[k].ids.size()  && new\_productions[i].terms[k].ids[0] == new\_productions[j].lhs){  for(int l = 0; l < new\_productions[j].terms.size(); l++){  Term new\_term(new\_productions[j].terms[l].to\_string() + " " + new\_productions[i].terms[k].to\_string());  new\_rhs.push\_back(new\_term);  }  }else if(new\_productions[i].terms[k].ids.size()) {  new\_rhs.push\_back(new\_productions[i].terms[k]);  }  }  new\_productions[i].terms = new\_rhs;  }  bool has\_direct\_recursion = false;  for(int j = 0; j < new\_productions[i].terms.size(); j++){  assert(new\_productions[i].terms[j].ids.size());  if(new\_productions[i].terms[j].ids[0] == new\_productions[i].lhs){  has\_direct\_recursion = true;  break;  }  }  if(!has\_direct\_recursion)  continue;    string extra\_production = new\_productions[i].lhs;  // extra\_production += "'";  while(find\_if(new\_productions.begin(), new\_productions.end(), [&extra\_production](const Production& curr){  return curr.lhs == extra\_production;  }) != new\_productions.end()){  extra\_production += "'";  }  vector<Term> rhs\_extra\_production;  vector<Term> curr\_production\_rhs;  for(int j = 0; j < new\_productions[i].terms.size(); j++){  assert(new\_productions[j].terms[j].ids.size());  if(new\_productions[i].terms[j].ids[0] == new\_productions[i].lhs){  string temp = new\_productions[i].terms[j].to\_string(1);  temp += " " + extra\_production;  rhs\_extra\_production.emplace\_back(temp);  }else{  if(new\_productions[i].terms[j].ids[0] == "ϵ"){  curr\_production\_rhs.emplace\_back(extra\_production);  }else{  // curr\_production\_rhs.e  curr\_production\_rhs.push\_back(new\_productions[i].terms[j]);  curr\_production\_rhs.back().ids.push\_back(extra\_production);  }  }  }  new\_productions[i].terms = curr\_production\_rhs;  rhs\_extra\_production.emplace\_back("ϵ");  string temp;  for(int i = 0; i < rhs\_extra\_production.size(); i++){  if(i != 0)  temp += " | ";  temp += rhs\_extra\_production[i].to\_string();  }  string line = extra\_production + " -> " + temp;  new\_productions.emplace\_back(line);  }  prods.clear();  for(auto& prod : new\_productions)  prods.emplace(prod.lhs, prod);  }  vector<string> find\_first(const string& identifier) {  {  auto it = firsts.find(identifier);  if (it != firsts.end()) {  return it->second;  }  if (terminals.find(identifier) != terminals.end())  return { identifier };  }  assert(non\_terminals.find(identifier) != non\_terminals.end()); // assert we are only finding first of non\_terminals  auto it = prods.find(identifier);  assert(it != prods.end());  const Production& curr = it->second;  vector<string> my\_firsts;  for (const auto& term : curr.terms) { // A -> a A | b A here there are 2 terms on rhs  bool has\_null\_production = true;  for (const auto& id : term.ids) { // A -> a A this term has 2 ids 'a' and 'A'  if (id == identifier || id == "ϵ")  continue;  vector<string> first = find\_first(id);  auto null\_pos = find(first.begin(), first.end(), "ϵ");  my\_firsts.insert(my\_firsts.end(), first.begin(), first.end());  has\_null\_production &= (null\_pos != first.end());  if (!has\_null\_production)  break;  }  if (has\_null\_production)  my\_firsts.push\_back("ϵ");  }  firsts[identifier] = my\_firsts;  return my\_firsts;  }  set<string> find\_follow(string identifier) {  assert(non\_terminals.find(identifier) != non\_terminals.end()); // we always call find\_follow on non terminals;    set<string> follow;  for (const auto& prod : prods) {  for (const auto& term : prod.second.terms) {  for (int i = 0; i < term.ids.size(); i++) {  if (term.ids[i] != identifier)  continue;  bool has\_null\_production = true;  for (int j = i + 1; j < term.ids.size(); j++) {  const auto& id = term.ids[j];  const vector<string>& fir = find\_first(id);  bool curr\_has\_null = false;  for (const auto& tok : fir) {  if (tok == "ϵ") {  curr\_has\_null = true;  }  else {  follow.insert(tok);  }  }  has\_null\_production &= curr\_has\_null;  if (!has\_null\_production)  break;  }  if (has\_null\_production) { // all id following us had null so follow(identifier) = follow(lhs)  if (identifier == prod.second.lhs)  continue;  const set<string>& fir\_lhs = find\_follow(prod.second.lhs);  for (const auto& tok : fir\_lhs)  follow.insert(tok);  }  }  }  }  follows[identifier].insert(follow.begin(), follow.end());  return follows[identifier];  }  map<string, Production> prods;  private:  set<string> terminals, non\_terminals;  map<string, vector<string>> firsts;  map<string, set<string>> follows;  };  int main() {  cout << "Enter Symbol ϵ for null production copy from this line if necessary\n";  vector<string> input;  string temp;  while (getline(cin, temp)) {  input.push\_back(temp);  }  Grammar grammar(input);  grammar.remove\_left\_recursion();  cout << "New Grammar after removing left recursion is as follows\n";  for(const auto& production : grammar.prods){  const auto& curr = production.second;  cout << curr.lhs << " -> ";  for(int i = 0; i < curr.terms.size(); i++){  cout << curr.terms[i].to\_string();  if((i + 1) != curr.terms.size())  cout << " | ";  }  cout << endl;  }  return 0;  } |

# Screenshots





In above screenshot we can see identifers bexpr, bterm which are multiple characters. This functionality makes this Program move user friendly and easy to use and can be applied in realworld application.