

**院 系：计 算 机 学 院**

**实验课程：编译原理**

**实验项目：TINY扩充语言的语法分析**

**指导老师：黄煜廉**

**专 业：计算机科学与技术**

**班 级：2018级 3 班**

**学 生：**

**学 号：**

**华南师范大学教务处**

**SLR(1)分析器的生成**

**一．实验目的**

设计一个应用软件，以实现SLR(1)分析器的生成。

1. **实验内容**
2. 要提供一个源程序编辑界面，让用户输入文法规则（可保存、打开存有文法规则的文件）。
3. 检查该文法是否需要进行文法的扩充。
4. 求出该文法各非终结符号的first集合与follow集合，并提供窗口以便用户可以查看这些集合结果。
5. 需要提供窗口以便用户可以查看文法对应的LR(0)DFA图。（此处用表格呈现）
6. 需要提供窗口以便用户可以查看该文法是否为SLR(1)文法。（如果非SLR(1)文法，可查看其原因）。
7. 需要提供窗口以便用户可以查看文法对应的SLR(1)分析表。（如果该文法为SLR(1)文法时）。

**三．设计思路**

**数据结构构建：**

1. 读取文法规则时，以‘->’作为分界线，将左边的为终结符号当成key；右边的符号序列作为value，若存在‘|’则分割value。采用键值对map<string, vector<vector< string>>>的方法记录文法规则。

如：[A->Bb B|Cc C]可以改为[ [A:< <Bb , B>,<Cc, C> >] ]

1. FIRST集合：根据集合的规则，非终结符的first集合用set< string>来存储，所有非终结符的first集合用map<string, set< string>>来存储

如：[A->Bb B|Cc C]的first集合为[ A:<Bb,Cc> ][设Bb,Cc均为终结符]

1. FOLLOW集合：在求解完first集合之后进行follow集合的求解。存储方式与first集合相同。
2. 由上课时所讲的知识可将消去左递归，其转变得到EBNF如下：
3. 构建Item类，存储每条规则不同中的‘.’的位置，如：E->.E|(n)，包含类型记录（归约/移进）。
4. LR(0)的DFA图中每个节点包含多个 Item，因此节点用vector< Item >存储，所有结点用vector<vector< Item>>来存储
5. 在构建LR(0)DFA图时，可同时判断该文法是否为SLR(1)文法

**部分思路：**

1. 先进行划分，按照行划分，句划分，词划分的顺序来对读入的数据进行分割。
2. 求first时，遍历m两遍，每次均按上课讲解求first。
3. 求follow时，遍历所有语法语句两次，每次均按上课讲解内容求follow，且要求first集合要比follow先。
4. 构建LR(0)和SLR(1)都需要先构建Item集，从起始状态即start开始，把相关的语句对应的Item和new\_item联系起来，然后往后顺着相关语句走。
5. 构建Item集时，对于下一位为终结符，则index+1，为非终结符则先往终结符的路走，回来后再index+1。

**四．程序源代码**

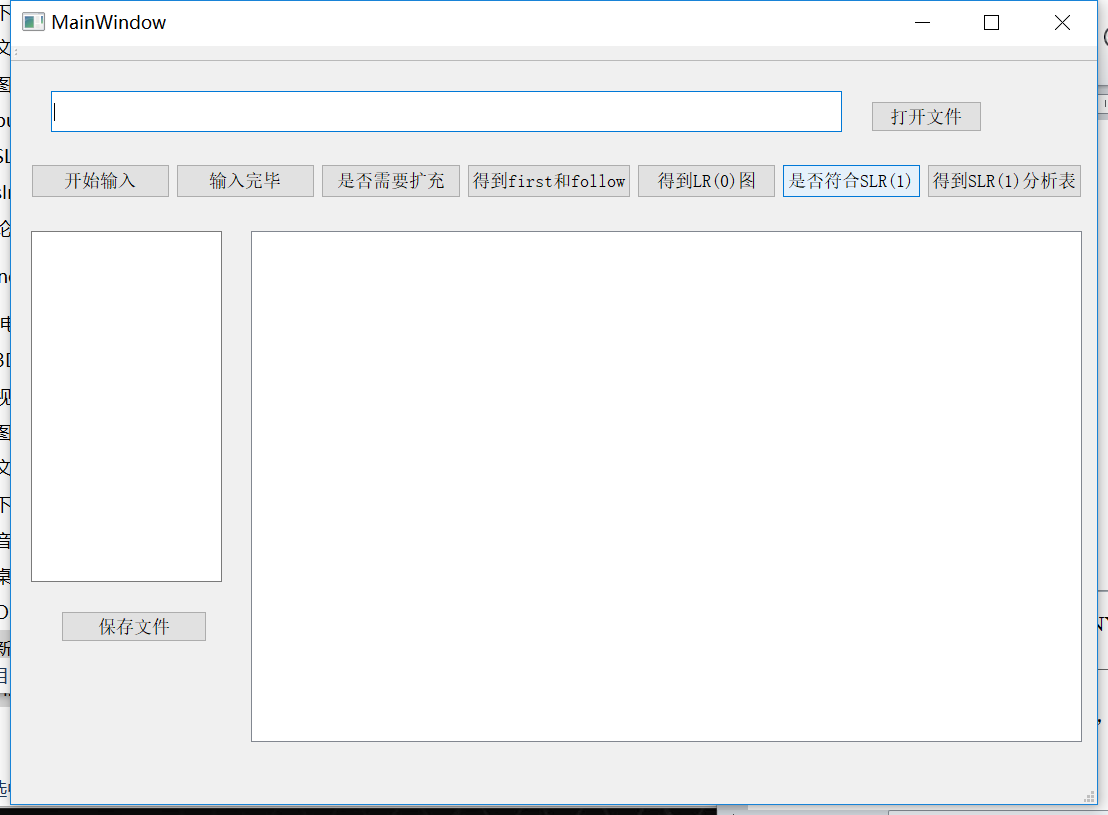
|  |
| --- |
| slr.h |
| #include <string>  #include <set>  #include <vector>  #include <map>  #include <iterator>  #include <iostream>  #include <QString>  using namespace std;  #ifndef \_GRAMMAR\_H\_  #define \_GRAMMAR\_H\_  class **Item** { //项目  public:  string key; //项目的key  int value\_num; //项目key对应的value的编号  int index; //'.'位置  int type; //1为移进项 2为归约项  **Item**(string a = "", int b = 0, int c = 0, int d = 1) {  this->key = a;  this->value\_num = b;  this->index = c;  this->type = d;  }  bool operator == (const Item &other) {  return (this->key == other.key) && (this->value\_num == other.value\_num) && (this->index == other.index) && (this->type == other.type);  }  };  class **slr** {  public:  map<int,string> Content\_num; //表项和序号连接  map<string,int> SLRContent\_num; //表项和序号连接  private:  map<string, vector<vector<string>>> m; //记录文法的每一条产生式，通过键值对的方式  string start; //记录开始符号  set<string> ntSet; //终结符集合  set<string> Vn; //非终结符集合  map<string, set<string>> m\_first; //记录每一个非终结符号对应的first集合  map<string, set<string>> m\_follow; //记录每一个非终结符对应的follow集合  vector<vector<Item>> DFA\_nodes; //记录DFA结点  map<int, map<string, int>> forwards; //记录移进关系  map<int, map<string, int>> backs; //记录归约关系  bool isSLR1; //是否为SLR(1)文法  string reason; //不是SLR(1)文法的原因  bool isExtend; //检测是否需要文法扩充  // vector<vector<string>> DFA; //DFA图  public:  **slr**(string inputString = ""); //构造函数(完成初始化操作)  string **get\_start**(); //得到开始符号  bool **get\_isExtend**();//得到需要文法扩充的判断信息  set<string> **get\_ntSet**(); //得到终结符集  set<string> **get\_first**(string key); //得到key的first集合  set<string> **get\_follow**(string key);//得到key的follow集合  map<string,set<string>> **getfirst**();//返回所有非终结符key的first集合串  map<string,set<string>> **getfollow**();//返回所有非终结符key的follow集合串  map<int,string> **getContent\_num**(); //得到串和编号  string\*\* **get\_DFA**(); //返回DFA图  map<string,int> **getSLRContent\_num**(); //得到串和编号  string\*\* **get\_SLR**();//返回SLR(1)分析图  bool **get\_isSLR1**();// 得到是否为SLR(1)文法判断信息  string **get\_reason**();//得到不是SLR(1)文法的原因  void **show**();  private:  void **first\_follow\_all**(); //求所有非终结符的first和follow函数  set<string> **first**(vector<string> value); //求value序列的first集合  void **node\_relationship**(); //求DFA图，判断是否为SLR(1)  void **extend**(int k); //扩展结点k  };  #endif |

|  |
| --- |
| slr\_scource.cpp |
| #include "slr.h"  #include <sstream>  using namespace std;  //构造函数  //切分，得到first,follow,LR(0)和SLR(1)  slr::**slr**(string inputString) {  //先用字符串分割的方式，找到每一个产生式子串  //再采用字符串流的方法，因为可以读取时可以按照空格、换行等等来分割  //分割出每一个产生式并存于可变数组中  vector<string> v;  int from = 0;  int len = 0;  int i;  //分行  for (i = 0; i < inputString.length(); ++i) {  if (inputString[i] == '\n') {  len = i - from;  v.push\_back(inputString.substr(from, len));  from = i + 1;  }  }  v.push\_back(inputString.substr(from));  //文法检查扩充  isExtend = false;  stringstream ss(v[0]);  string s;  ss >> s;  if (v[0].find('|') != string::npos) {  v.insert(v.begin(), string(s + "'" + " -> " + s));  s += "'";  isExtend = true;  }  this->start = s;  //处理每一个产生式变成键值对  for (auto &x : v) {  string key;  vector<vector<string>> values;  string a;  vector<string> value;  if (x == "")break;  stringstream ss(x + " $");  ss >> key;  ss >> a;  while (1) {  ss >> a;  if (a == "$") {  break;  }  if (a == "|") {  values.push\_back(value);  value.clear();  continue;  }  int from = 0;  int len = 0;  for (int i = 0; i < a.length(); ++i) {  if (a[i] == '(' || a[i] == ')') {  len = i - from;  if (len != 0) {  value.push\_back(a.substr(from, len));  }  from = i + 1;  value.push\_back(string(1, a[i]));  }  }  if (from != i) {  value.push\_back(a.substr(from));  }  }  values.push\_back(value);  this->m[key] = values;  }  //求非终结符集  for (auto it = m.begin(); it != m.end(); ++it) {  this->ntSet.insert(it->first);  }  //求所有终结符的follow集合和follow集合  this->first\_follow\_all();  //求LR(0)DFA、判断是否为SLR(1)文法、SLR(1)表格  this->node\_relationship();  }  //是否要扩充  bool slr::**get\_isExtend**() {  return isExtend;  }  //是否符合SLR(1)  bool slr::**get\_isSLR1**() {  return isSLR1;  }  //不符合SLR(1)原因  string slr::**get\_reason**()  {  return reason;  }  //得到first集合  map<string,set<string>> slr::**getfirst**() {  return m\_first;  }  //得到follow集合  map<string,set<string>> slr::**getfollow**() {  return m\_follow;  }  //得到LR(0)的DFA内容和编号  map<int,string> slr::**getContent\_num**(){  string s;  for (int i = 0; i<this->DFA\_nodes.size(); i++) {  for (auto &p : this->DFA\_nodes[i]) {  s = s+p.key+"->";  for (int j=0; j<this->m[p.key][p.value\_num].size(); j++) {  if (j == p.index) {s=s+".";}  s = s+this->m[p.key][p.value\_num][j]+" ";  }  if (p.index == this->m[p.key][p.value\_num].size()) {  s = s+".";  }  s = s+"\n";  }  Content\_num[i]=s; s.clear();  }  return Content\_num;  }  string\*\* slr::**get\_DFA**() {  int Gsize = DFA\_nodes.size();  string\*\* DFA;  DFA=new string\*[Gsize];  for(int i=0; i<Gsize; i++){  DFA[i]=new string[Gsize];  }  for (auto &x : this->forwards) {  int from = x.first;  for (auto &y : x.second) {  string t = y.first;  int to = y.second;  DFA[from][to]=t;  }  }  return DFA;  }  //获得配对  map<string,int> slr::**getSLRContent\_num**(){  set<string> sSet;  //得到移进和规约  for (auto &x : this->forwards) {  for (auto &y : x.second) {  sSet.insert(y.first);  }  }  for (auto &x : this->backs) {  for (auto &y : x.second) {  sSet.insert(y.first);  }  }  //配起来  int num=0;  for(set<string>::iterator it=sSet.begin(); it!=sSet.end(); it++){  SLRContent\_num[\*it]=num;  num++;  }  return SLRContent\_num;  }  //过的SLR分析表  string\*\* slr::**get\_SLR**(){  stringstream ss;  set<string> sSet;  string\*\* SLR;  getSLRContent\_num();  getContent\_num();  //得到移进和规约  for (auto &x : this->forwards) {  for (auto &y : x.second) {  sSet.insert(y.first);  }  }  for (auto &x : this->backs) {  for (auto &y : x.second) {  sSet.insert(y.first);  }  }  //开辟空间  SLR = new string\*[DFA\_nodes.size()];  for(int i=0; i<DFA\_nodes.size(); i++){  SLR[i]=new string[sSet.size()];  }  //建表  for (int i = 0; i<this->DFA\_nodes.size(); i++) {  for (auto &s : sSet) {  if (this->forwards[i].find(s) != this->forwards[i].end()) { //有无移进项  if (this->ntSet.find(s) != this->ntSet.end()) {  SLR[i][SLRContent\_num[s]] = to\_string(this->forwards[i][s]);  }  else {  SLR[i][SLRContent\_num[s]] = "S" + to\_string(this->forwards[i][s]);  }  }  else if (this->backs[i].find(s) != this->backs[i].end()) { //有无归约项  Item & p = this->DFA\_nodes[i][this->backs[i][s]];  if (p.key == this->start) {  SLR[i][SLRContent\_num[s]] = "AC";  }  else {  string a = "r(";  a += p.key;  a += " -> ";  vector<string> & value = this->m[p.key][p.value\_num];  for (auto &b : value) {  a += b;  a += " ";  }  a[a.length() - 1] = ')';  SLR[i][SLRContent\_num[s]]= a;  }  }  else {  SLR[i][SLRContent\_num[s]]= " ";  }  }  }  return SLR;  }  //得到key的first集合  set<string> slr::**get\_first**(string key) {  if (this->ntSet.find(key) == this->ntSet.end()) {  return set<string>{key};  }  return this->m\_first[key];  }  //得到key的follow集合  set<string> slr::**get\_follow**(string key) {  return m\_follow[key];  }  //批量按序  set<string> slr::**first**(vector<string> value) {  set<string> first\_set;  int k;  for (k = 0; k < value.size(); ++k) {  set<string> first\_set\_k = this->get\_first(value[k]);  for (auto &s : first\_set\_k) {  first\_set.insert(s);  }  if (first\_set\_k.find("E") == first\_set\_k.end()) {  break;  }  first\_set.erase("E");  }  if (k == value.size()) {  first\_set.insert("E");  }  return first\_set;  }  //求first和follow  void slr::**first\_follow\_all**() {  //求所有非终结符的first集合  //初始化  for (auto &nt : this->ntSet) {  this->m\_follow[nt] = set<string>();  }  bool isChange = true;  while (isChange) {  isChange = false;  for (auto &p : this->m) {  string key = p.first;  vector<vector<string>> values = p.second;  for (auto &value : values) {  int k;  for (k = 0; k < value.size(); ++k) {  auto && first\_set\_k = this->get\_first(value[k]);  for (auto &s : first\_set\_k) {  if ((s != "E") && (this->m\_first[key].find(s) == this->m\_first[key].end())) {  this->m\_first[key].insert(s);  isChange = true;  }  }  if (first\_set\_k.find("E") == first\_set\_k.end()) {  break;  }  }  if (k == value.size() && (this->m\_first[key].find("E") == this->m\_first[key].end())) {  this->m\_first[key].insert("E");  isChange = true;  }  }  }  }  //求所有非终结符的follow集合  //初始化  for (auto &nt : this->ntSet) {  this->m\_follow[nt] = set<string>();  }  this->m\_follow[this->start].insert("$");  isChange = true;  while (isChange) {  isChange = false;  for (auto &p : this->m) { //遍历每一个产生式p  string key = p.first;  vector<vector<string>> values = p.second;  for (auto &value : values) {  //遍历value中每一个非终结符  for (auto it = value.begin(); it != value.end(); ++it) {  if (this->ntSet.find(\*it) != this->ntSet.end()) {  //特殊情况  if (it + 1 == value.end()) {  for (auto &x : this->get\_follow(key)) {  if (this->m\_follow[\*it].find(x) == this->m\_follow[\*it].end()) {  this->m\_follow[\*it].insert(x);  isChange = true;  }  }  }  else {  set<string> first\_set = this->first(vector<string>(it + 1, value.end()));  for (auto &x : first\_set) {  if (this->m\_follow[\*it].find(x) == this->m\_follow[\*it].end() && x != "E") {  this->m\_follow[\*it].insert(x);  isChange = true;  }  }  if (first\_set.find("E") != first\_set.end()) { //如果有ε符号  this->m\_follow[\*it].erase("ε");  for (auto &x : this->get\_follow(key)) {  if (this->m\_follow[\*it].find(x) == this->m\_follow[\*it].end()) {  this->m\_follow[\*it].insert(x);  isChange = true;  }  }  }  }  }  }  }  }  }  }  //求LR(0)和SLR(1)  void slr::**node\_relationship**() {  reason = "";  //初始化  vector<Item> node{ Item{this->start, 0, 0, 1} };  this->DFA\_nodes.push\_back(node);  this->isSLR1 = true;  //循环处理每一个结点i  for (int i = 0; i < this->DFA\_nodes.size(); ++i) {  this->forwards[i] = map<string, int>();  this->backs[i] = map<string, int>();  //扩展该结点  extend(i);  //遍历该结点的每一个项目  for (int j = 0; j < this->DFA\_nodes[i].size(); ++j) {  //处理归约项  Item &p = this->DFA\_nodes[i][j];  if (p.type == 2) {  set<string> follow\_set = this->get\_follow(p.key);  for (auto &s : follow\_set) {  if (this->backs[i].find(s) != this->backs[i].end()) {  this->isSLR1 = false;  stringstream ss;  ss << "第" << i << "个结点中归约项目的follow集合有交集\n";  reason += ss.str();  }  this->backs[i][s] = j;  }  }  //处理移进项  else {  string t = this->m[p.key][p.value\_num][p.index];  Item newP(p.key, p.value\_num, p.index + 1, 1);  if (newP.index >= this->m[p.key][p.value\_num].size()) {  newP.type = 2;  }  if (this->forwards[i].find(t) == this->forwards[i].end()) {//转换t未存在  int k = 0;  for (k = 0; k < this->DFA\_nodes.size(); ++k) {  vector<Item> & n = this->DFA\_nodes[k];  if (find(n.begin(), n.end(), newP) != n.end()) {  break;  }  }  if (k < this->DFA\_nodes.size()) { //项目已经存在  this->forwards[i][t] = k;  }  else { //项目未存在  //新建节点  this->DFA\_nodes.push\_back(vector<Item>{newP});  this->forwards[i][t] = this->DFA\_nodes.size() - 1;  }  }  else { //转换t存在  int k = this->forwards[i][t];  vector<Item> & n = this->DFA\_nodes[k];  if (find(n.begin(), n.end(), newP) == n.end()) { //项目不存在  this->DFA\_nodes[k].push\_back(newP);  }  }  }  }  }  if (this->isSLR1) {  for (int i = 0; i < this->DFA\_nodes.size(); ++i) {  set<string> set1, set2, result;  for (auto &x : this->forwards[i]) {  set1.insert(x.first);  }  for (auto &x : this->backs[i]) {  set2.insert(x.first);  }  set\_intersection(begin(*set1*), end(*set1*), begin(*set2*), end(*set2*),  inserter(*result*, begin(*result*)));  if (result.size() != 0) {  this->isSLR1 = false;  stringstream ss;  ss << "第" << i << "个结点的移进项的first集合和归约项的follow集合有交集\n";  reason += ss.str();  }  }  }  }  //扩展  void slr::**extend**(int k) {  vector<Item> &node = this->DFA\_nodes[k];  for (int i = 0; i < node.size(); ++i) {  Item &p = node[i];  if (p.type == 1) {  string extend\_s = this->m[p.key][p.value\_num][p.index];  if (this->ntSet.find(extend\_s) != this->ntSet.end()) {  //可扩展的符号  vector<vector<string>> &values = this->m[extend\_s];  //遍历extend\_s对应的每一个value  for (int j = 0; j < values.size(); ++j) {  Item addP(extend\_s, j, 0, 1);  if (values[j][0] == "E") {//extend\_s -> E特殊情况  addP.index = 1;  addP.type = 2;  }  //判断有无重复  if (find(node.begin() + i, node.end(), addP) == node.end()) {  node.push\_back(addP);  }  }  }  }  }  } |

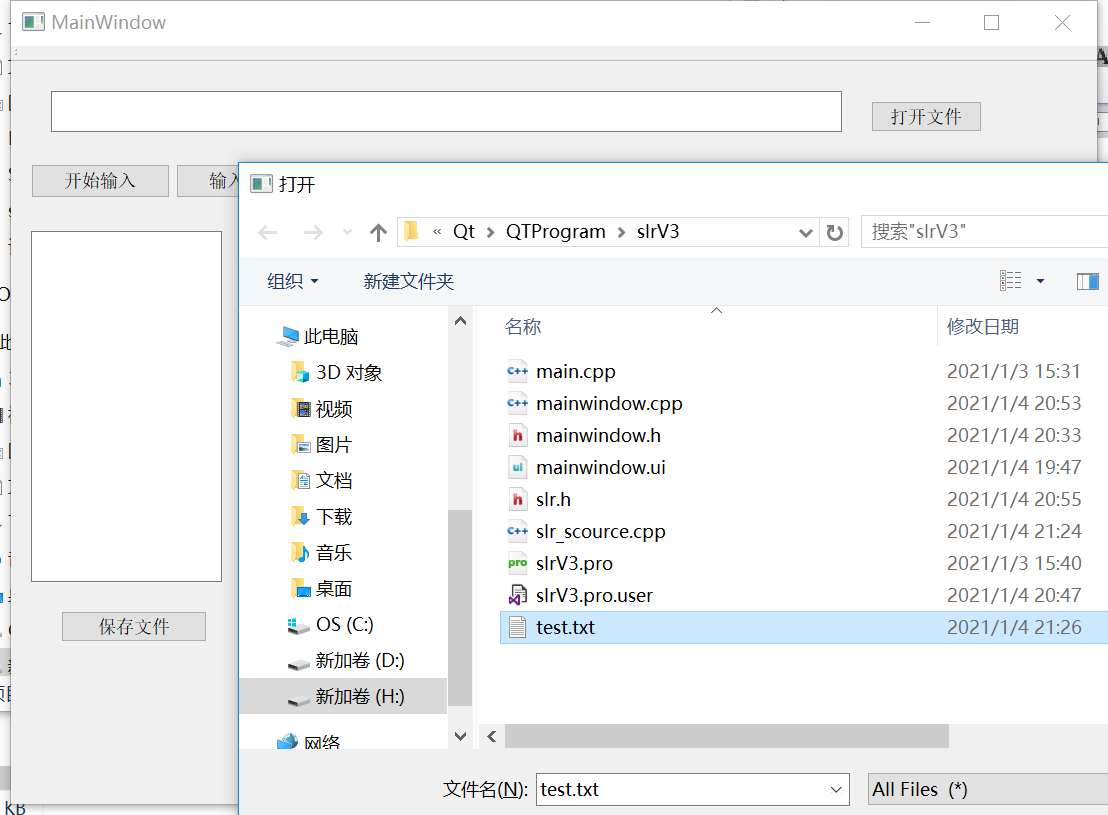
**五．实验结果**

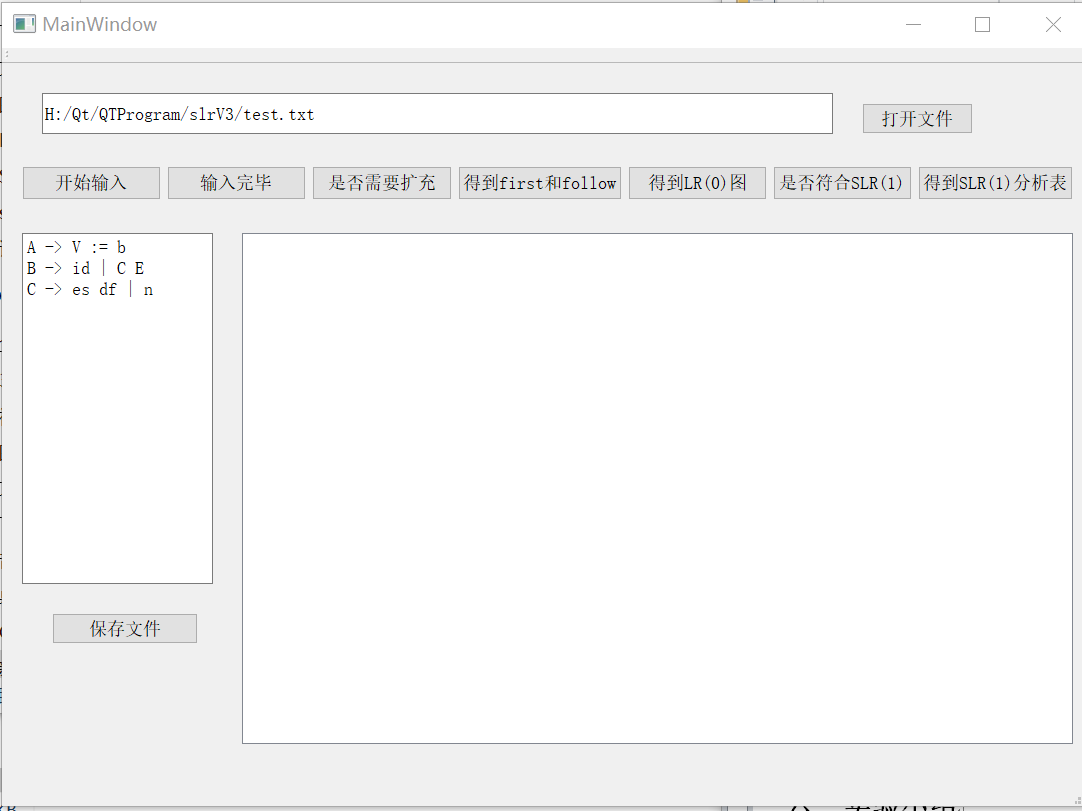
运行程序，截图如下：

1. 一般运行页面

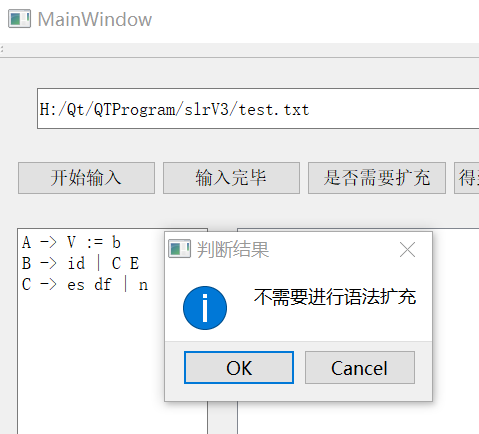


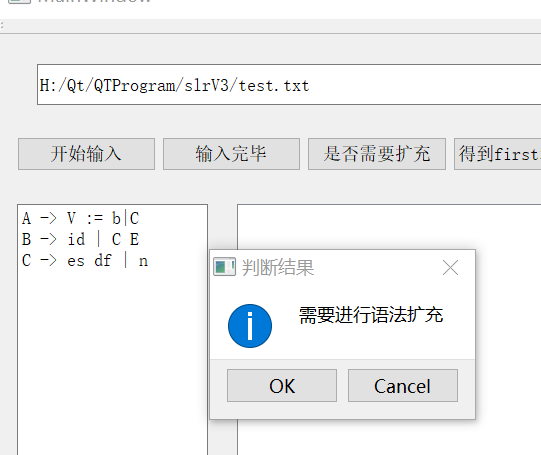
1. 打开文件



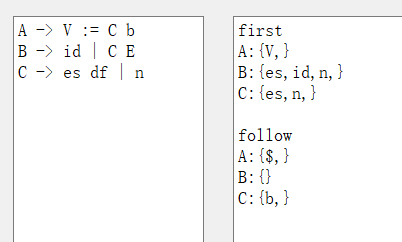


1. 检查是否需要扩充

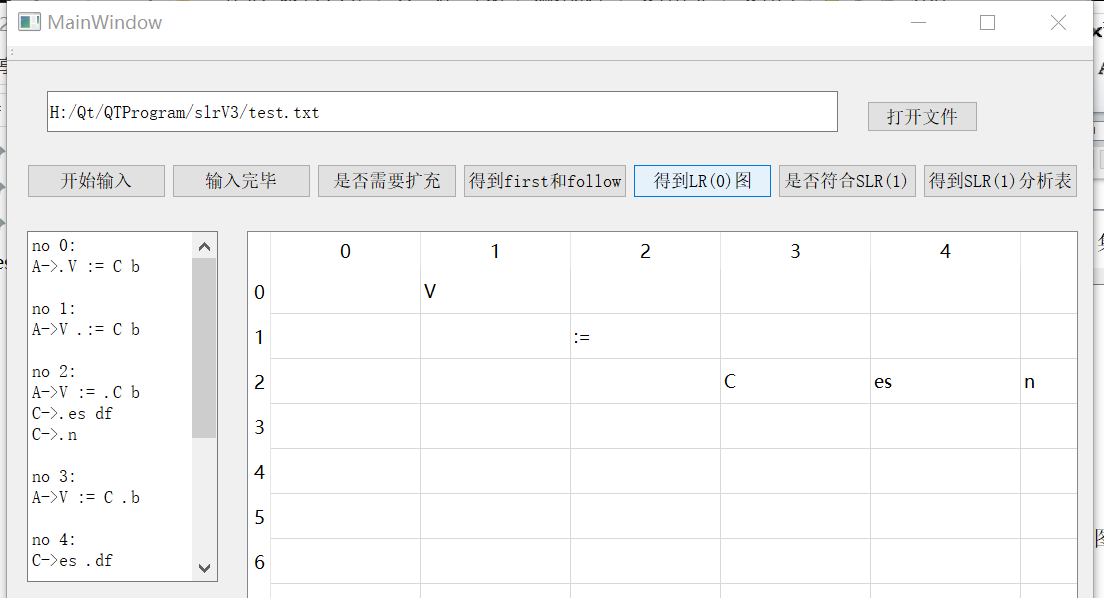




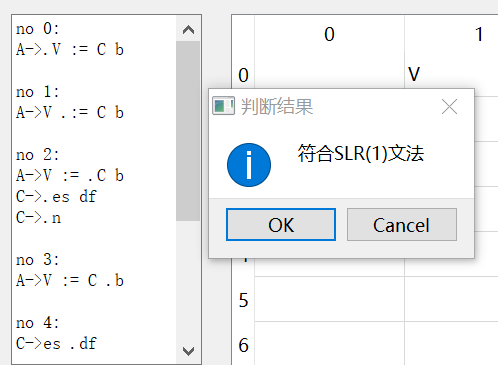
1. 求解first和follow集合



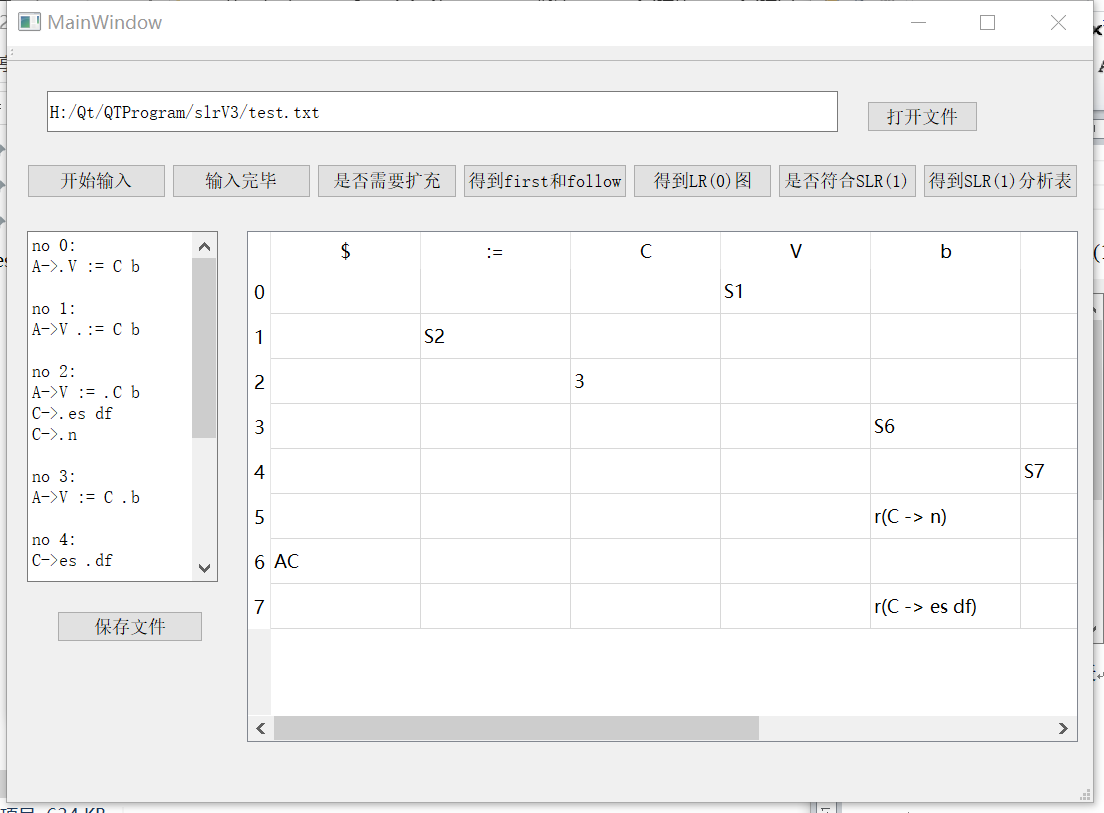
1. 得到LR(0)的DFA图



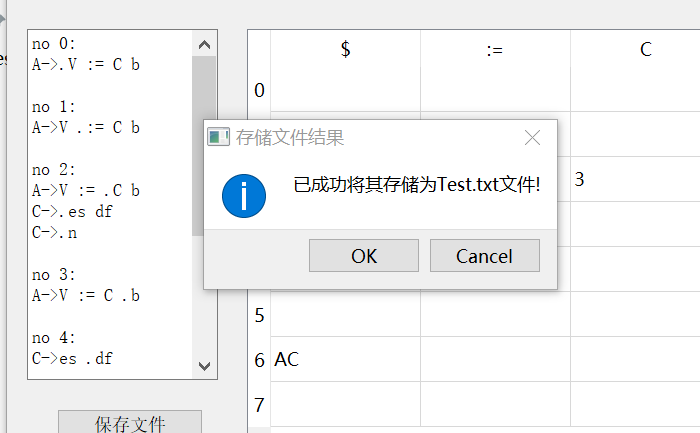
1. 检查是否符号SLR(1)

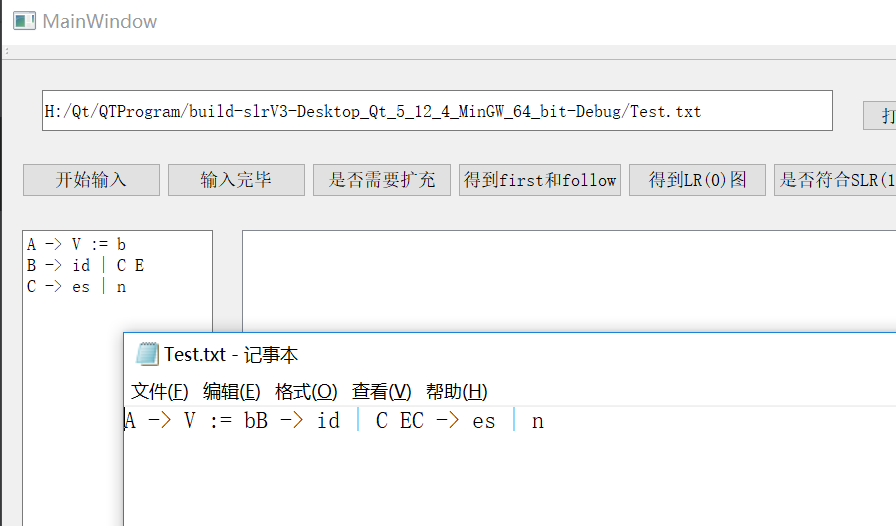


1. 得到SLR(1)分析表



1. 保存文件





**六．实验小结**

本次实验通过对编译原理中SLR(1)语法分析的实现，进一步了解了first和follow集合的求解算法，Item的构建和使用以LR(0)和SLR(1)的构造方法，但经过测试，本程序对于有间接递归的情况处理还有缺陷。

1. **实验小结**

参考资料：

1. :黄煜廉老师的ppt
2. : <https://blog.csdn.net/rl529014/article/details/51345619>
3. :https://www.cnblogs.com/Serenaxy/p/14145468.html#%E5%AE%9E%E9%AA%8C%E8%A6%81%E6%B1%82