**项目测试报告**

**任务二**

**1、准备工作**

**tiny的文法：**

|  |
| --- |
| program | stmt-sequence | statement | if-stmt | repeat-stmt | assign-stmt | read-stmt | write-stmt | exp | simple-exp | comparison-op | addop | term | mulop | factor  SEMI | if | then | end | else | repeat | until | ASSIGN | read | ID | number | write | LT | EQ | LTEQ | NE | RTEQ | RT | PLUS | MINUS | MULTIPLY | DIVIDE | MOD | LPAN | RPAN | NUMBER  program -> stmt-sequence  stmt-sequence -> stmt-sequence SEMI statement  stmt-sequence -> statement  statement -> if-stmt  statement -> repeat-stmt  statement -> assign-stmt  statement -> read-stmt  statement -> write-stmt  if-stmt -> if exp then stmt-sequence end  if-stmt -> if exp then stmt-sequence else stmt-sequence end  repeat-stmt -> repeat stmt-sequence until exp  assign-stmt -> ID ASSIGN exp  read-stmt -> read ID  write-stmt -> write exp  exp -> simple-exp comparison-op simple-exp  exp -> simple-exp  comparison-op -> LT  comparison-op -> EQ  comparison-op -> LTEQ  comparison-op -> NE  comparison-op -> RTEQ  comparison-op -> RT  simple-exp -> simple-exp addop term  simple-exp -> term  addop -> PLUS  addop -> MINUS  term -> term mulop factor  term -> factor  mulop -> MULTIPLY  mulop -> DIVIDE  mulop -> MOD  factor -> LPAN exp RPAN  factor -> NUMBER  factor -> ID |

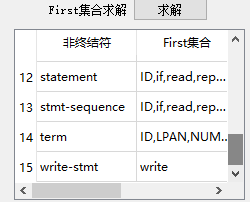
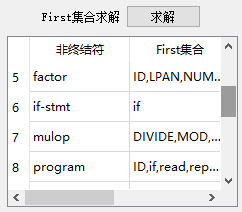
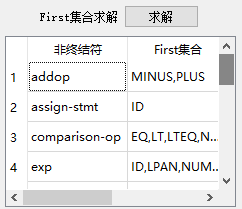
**语义函数：**

|  |
| --- |
| 0  0 -1 1  0 0  0  0  0  0  0  0 1 -1 2 -1  0 1 -1 2 -1 3 -1  0 1 -1 2  1 0 2  0 1  0 1  1 0 2  0  0  0  0  0  0  0  1 0 2  0  0  0  1 0 2  0  0  0  0  -1 0 -1  0  0 |

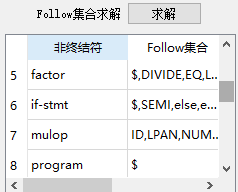
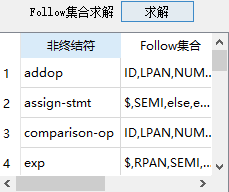
**输入文法：**



**2、求解first集合**



**3、求解follow集合**

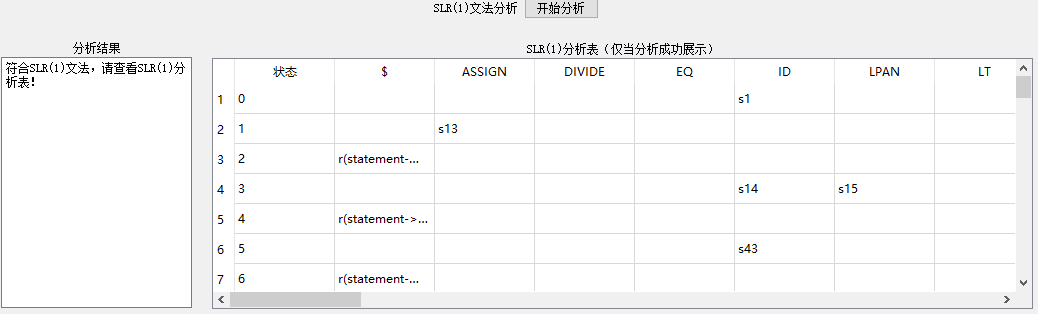


**4、LR0**



**项目太多，未能展示完全**

**5、SLR1表**



**项目太多，未能展示完全**

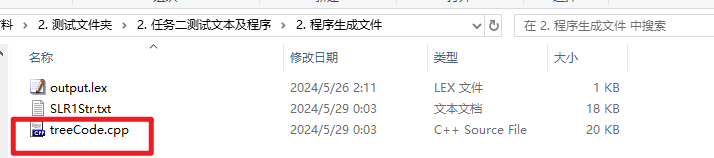
**6、语法树代码生成**

**注意：lex所在路径即为代码生成路径**



**7、查看语法树生成代码文件**

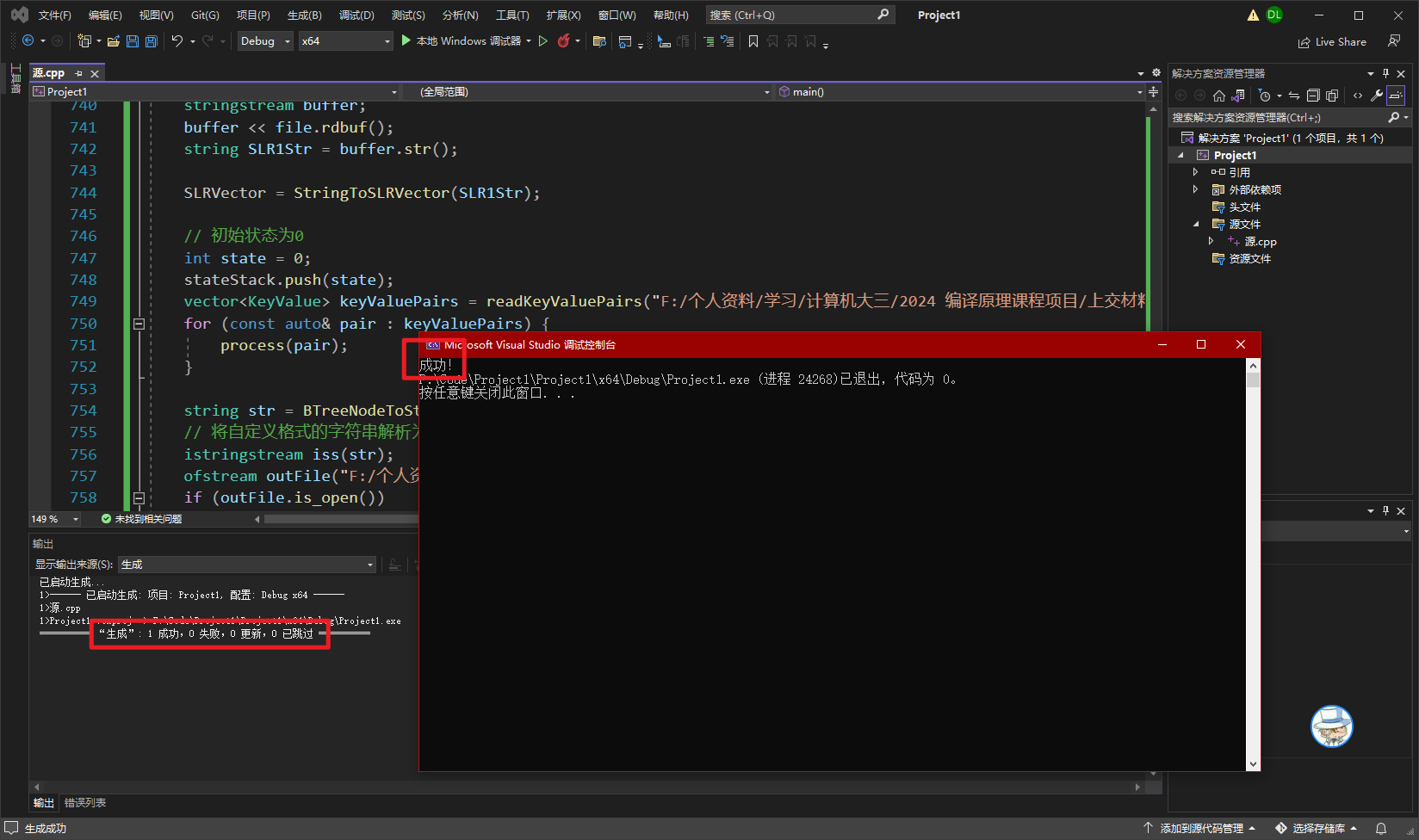
**PS:请用C++11以上进行编译**



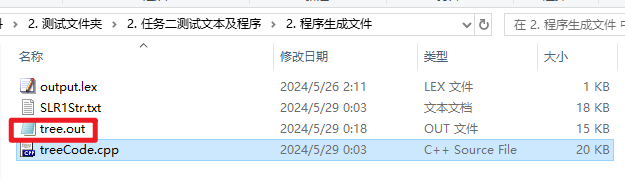
具体语法树代码如下：

|  |
| --- |
| #include <iostream>  #include <stack>  #include <vector>  #include <map>  #include <string>  #include <sstream>  #include <fstream>  using namespace std;  map<string, int> grammarMap = { {"program->stmt-sequence" , 0},{"stmt-sequence->stmt-sequence SEMI statement" , 1},{"stmt-sequence->statement" , 2},{"statement->if-stmt" , 3},{"statement->repeat-stmt" , 4},{"statement->assign-stmt" , 5},{"statement->read-stmt" , 6},{"statement->write-stmt" , 7},{"if-stmt->if exp then stmt-sequence end" , 8},{"if-stmt->if exp then stmt-sequence else stmt-sequence end" , 9},{"repeat-stmt->repeat stmt-sequence until exp" , 10},{"assign-stmt->ID ASSIGN exp" , 11},{"read-stmt->read ID" , 12},{"write-stmt->write exp" , 13},{"exp->simple-exp comparison-op simple-exp" , 14},{"exp->simple-exp" , 15},{"comparison-op->LT" , 16},{"comparison-op->EQ" , 17},{"comparison-op->LTEQ" , 18},{"comparison-op->NE" , 19},{"comparison-op->RTEQ" , 20},{"comparison-op->RT" , 21},{"simple-exp->simple-exp addop term" , 22},{"simple-exp->term" , 23},{"addop->PLUS" , 24},{"addop->MINUS" , 25},{"term->term mulop factor" , 26},{"term->factor" , 27},{"mulop->MULTIPLY" , 28},{"mulop->DIVIDE" , 29},{"mulop->MOD" , 30},{"factor->LPAN exp RPAN" , 31},{"factor->NUMBER" , 32},{"factor->ID" , 33} };  // 定义一个结构体来表示每一行的键值对  struct KeyValue {  string key;  string value;  KeyValue() {}  KeyValue(string \_key) {  key = \_key;  }  KeyValue(string \_key, string \_value) {  key = \_key;  value = \_value;  }  };  // 函数读取并分隔每一行的键值对  vector<KeyValue> readKeyValuePairs(const string& filename) {  ifstream file(filename);  vector<KeyValue> keyValuePairs;  if (!file) {  return keyValuePairs;  }  string line;  while (getline(file, line)) {  // 使用 stringstream 将每一行分隔成键和值  stringstream ss(line);  string key, value;  getline(ss, key, ':'); // 以 ':' 分隔键  getline(ss, value); // 获取剩下的作为值  // 去除键和值两端的空格  key.erase(0, key.find\_first\_not\_of(" \t"));  key.erase(key.find\_last\_not\_of(" \t") + 1);  value.erase(0, value.find\_first\_not\_of(" \t"));  value.erase(value.find\_last\_not\_of(" \t") + 1);  // 将键值对添加到向量中  keyValuePairs.push\_back({ key, value });  }  file.close();  return keyValuePairs;  }  // 定义语法树节点结构  struct BTreeNode {  string kind;  string value;  vector<BTreeNode\*> nodeList;  BTreeNode(string kind, string val) :kind(kind), value(val) {}  };  stack<BTreeNode\*> treeStack; // 树节点栈  stack<KeyValue> strStack; // 分析栈  stack<int> stateStack; // 状态栈  // 定义数据结构  struct SLRUnit  {  string index;  map<string, string> m;  };  vector<SLRUnit> SLRVector;  // 将自定义格式的字符串解析成 SLRVector  vector<SLRUnit> StringToSLRVector(const string& str)  {  vector<SLRUnit> vec;  SLRUnit unit;  istringstream iss(str); // 创建一个输入流  string line;  int index = 0;  while (getline(iss, line))  {  if (line == "SLRUnit")  {  unit = SLRUnit(); // 创建一个新的 SLRUnit  unit.index = to\_string(index); // 标记序号  }  else if (line == "{")  {  // 忽略  }  else if (line == "}")  {  vec.push\_back(unit); // 将完成的 SLRUnit 添加到 vector 中  }  else if (line.size() > 0)  {  // 解析键值对  size\_t pos = line.find(": ");  if (pos != string::npos)  {  string key = line.substr(pos + 2); // 获取键的内容  getline(iss, line); // 读取下一行，这一行应该是值  pos = line.find(": ");  if (pos != string::npos)  {  string value = line.substr(pos + 2); // 获取值的内容  unit.m[key] = value; // 添加到当前 SLRUnit 中  }  }  }  }  return vec;  }  // 将 BTreeNode 转换为自定义格式的字符串  string BTreeNodeToString(BTreeNode\* node, int depth = 0)  {  ostringstream oss; // 创建一个输出流  string indent(depth \* 4, ' '); // 缩进  oss << indent << "BTreeNode\n";  oss << indent << "{\n";  oss << indent << " kind: " << node->kind << "\n";  oss << indent << " value: " << node->value << "\n";  if (!node->nodeList.empty())  {  oss << indent << " nodeList:\n";  for (const auto& child : node->nodeList)  {  oss << BTreeNodeToString(child, depth + 1); // 递归转换子节点  }  }  oss << indent << "}\n";  return oss.str(); // 返回输出流中的字符串  }  // program->stmt-sequence  string fun0() {  BTreeNode\* newNode0 = treeStack.top();  strStack.pop();  stateStack.pop();  treeStack.pop();  treeStack.push(newNode0);  return "program";  }  // stmt-sequence->stmt-sequence SEMI statement  string fun1() {  BTreeNode\* newNode2 = treeStack.top();  strStack.pop();  stateStack.pop();  treeStack.pop();  string symbol1 = strStack.top().value;  strStack.pop();  stateStack.pop();  BTreeNode\* newNode1 = new BTreeNode("SEMI", symbol1);  BTreeNode\* newNode0 = treeStack.top();  strStack.pop();  stateStack.pop();  treeStack.pop();  BTreeNode\* rootNode = new BTreeNode("-1", "stmt-sequence");  rootNode->nodeList.push\_back(newNode0);  rootNode->nodeList.push\_back(newNode2);  treeStack.push(rootNode);  return "stmt-sequence";  }  // stmt-sequence->statement  string fun2() {  BTreeNode\* newNode0 = treeStack.top();  strStack.pop();  stateStack.pop();  treeStack.pop();  treeStack.push(newNode0);  return "stmt-sequence";  }  // statement->if-stmt  string fun3() {  BTreeNode\* newNode0 = treeStack.top();  strStack.pop();  stateStack.pop();  treeStack.pop();  treeStack.push(newNode0);  return "statement";  }  // statement->repeat-stmt  string fun4() {  BTreeNode\* newNode0 = treeStack.top();  strStack.pop();  stateStack.pop();  treeStack.pop();  treeStack.push(newNode0);  return "statement";  }  // statement->assign-stmt  string fun5() {  BTreeNode\* newNode0 = treeStack.top();  strStack.pop();  stateStack.pop();  treeStack.pop();  treeStack.push(newNode0);  return "statement";  }  // statement->read-stmt  string fun6() {  BTreeNode\* newNode0 = treeStack.top();  strStack.pop();  stateStack.pop();  treeStack.pop();  treeStack.push(newNode0);  return "statement";  }  // statement->write-stmt  string fun7() {  BTreeNode\* newNode0 = treeStack.top();  strStack.pop();  stateStack.pop();  treeStack.pop();  treeStack.push(newNode0);  return "statement";  }  // if-stmt->if exp then stmt-sequence end  string fun8() {  string symbol4 = strStack.top().value;  strStack.pop();  stateStack.pop();  BTreeNode\* newNode4 = new BTreeNode("end", symbol4);  BTreeNode\* newNode3 = treeStack.top();  strStack.pop();  stateStack.pop();  treeStack.pop();  string symbol2 = strStack.top().value;  strStack.pop();  stateStack.pop();  BTreeNode\* newNode2 = new BTreeNode("then", symbol2);  BTreeNode\* newNode1 = treeStack.top();  strStack.pop();  stateStack.pop();  treeStack.pop();  string symbol0 = strStack.top().value;  strStack.pop();  stateStack.pop();  BTreeNode\* newNode0 = new BTreeNode("if", symbol0);  newNode0->nodeList.push\_back(newNode1);  newNode0->nodeList.push\_back(newNode3);  treeStack.push(newNode0);  return "if-stmt";  }  // if-stmt->if exp then stmt-sequence else stmt-sequence end  string fun9() {  string symbol6 = strStack.top().value;  strStack.pop();  stateStack.pop();  BTreeNode\* newNode6 = new BTreeNode("end", symbol6);  BTreeNode\* newNode5 = treeStack.top();  strStack.pop();  stateStack.pop();  treeStack.pop();  string symbol4 = strStack.top().value;  strStack.pop();  stateStack.pop();  BTreeNode\* newNode4 = new BTreeNode("else", symbol4);  BTreeNode\* newNode3 = treeStack.top();  strStack.pop();  stateStack.pop();  treeStack.pop();  string symbol2 = strStack.top().value;  strStack.pop();  stateStack.pop();  BTreeNode\* newNode2 = new BTreeNode("then", symbol2);  BTreeNode\* newNode1 = treeStack.top();  strStack.pop();  stateStack.pop();  treeStack.pop();  string symbol0 = strStack.top().value;  strStack.pop();  stateStack.pop();  BTreeNode\* newNode0 = new BTreeNode("if", symbol0);  newNode0->nodeList.push\_back(newNode1);  newNode0->nodeList.push\_back(newNode3);  newNode0->nodeList.push\_back(newNode5);  treeStack.push(newNode0);  return "if-stmt";  }  // repeat-stmt->repeat stmt-sequence until exp  string fun10() {  BTreeNode\* newNode3 = treeStack.top();  strStack.pop();  stateStack.pop();  treeStack.pop();  string symbol2 = strStack.top().value;  strStack.pop();  stateStack.pop();  BTreeNode\* newNode2 = new BTreeNode("until", symbol2);  BTreeNode\* newNode1 = treeStack.top();  strStack.pop();  stateStack.pop();  treeStack.pop();  string symbol0 = strStack.top().value;  strStack.pop();  stateStack.pop();  BTreeNode\* newNode0 = new BTreeNode("repeat", symbol0);  newNode0->nodeList.push\_back(newNode1);  newNode0->nodeList.push\_back(newNode3);  treeStack.push(newNode0);  return "repeat-stmt";  }  // assign-stmt->ID ASSIGN exp  string fun11() {  BTreeNode\* newNode2 = treeStack.top();  strStack.pop();  stateStack.pop();  treeStack.pop();  string symbol1 = strStack.top().value;  strStack.pop();  stateStack.pop();  BTreeNode\* newNode1 = new BTreeNode("ASSIGN", symbol1);  string symbol0 = strStack.top().value;  strStack.pop();  stateStack.pop();  BTreeNode\* newNode0 = new BTreeNode("ID", symbol0);  newNode1->nodeList.push\_back(newNode0);  newNode1->nodeList.push\_back(newNode2);  treeStack.push(newNode1);  return "assign-stmt";  }  // read-stmt->read ID  string fun12() {  string symbol1 = strStack.top().value;  strStack.pop();  stateStack.pop();  BTreeNode\* newNode1 = new BTreeNode("ID", symbol1);  string symbol0 = strStack.top().value;  strStack.pop();  stateStack.pop();  BTreeNode\* newNode0 = new BTreeNode("read", symbol0);  newNode0->nodeList.push\_back(newNode1);  treeStack.push(newNode0);  return "read-stmt";  }  // write-stmt->write exp  string fun13() {  BTreeNode\* newNode1 = treeStack.top();  strStack.pop();  stateStack.pop();  treeStack.pop();  string symbol0 = strStack.top().value;  strStack.pop();  stateStack.pop();  BTreeNode\* newNode0 = new BTreeNode("write", symbol0);  newNode0->nodeList.push\_back(newNode1);  treeStack.push(newNode0);  return "write-stmt";  }  // exp->simple-exp comparison-op simple-exp  string fun14() {  BTreeNode\* newNode2 = treeStack.top();  strStack.pop();  stateStack.pop();  treeStack.pop();  BTreeNode\* newNode1 = treeStack.top();  strStack.pop();  stateStack.pop();  treeStack.pop();  BTreeNode\* newNode0 = treeStack.top();  strStack.pop();  stateStack.pop();  treeStack.pop();  newNode1->nodeList.push\_back(newNode0);  newNode1->nodeList.push\_back(newNode2);  treeStack.push(newNode1);  return "exp";  }  // exp->simple-exp  string fun15() {  BTreeNode\* newNode0 = treeStack.top();  strStack.pop();  stateStack.pop();  treeStack.pop();  treeStack.push(newNode0);  return "exp";  }  // comparison-op->LT  string fun16() {  string symbol0 = strStack.top().value;  strStack.pop();  stateStack.pop();  BTreeNode\* newNode0 = new BTreeNode("LT", symbol0);  treeStack.push(newNode0);  return "comparison-op";  }  // comparison-op->EQ  string fun17() {  string symbol0 = strStack.top().value;  strStack.pop();  stateStack.pop();  BTreeNode\* newNode0 = new BTreeNode("EQ", symbol0);  treeStack.push(newNode0);  return "comparison-op";  }  // comparison-op->LTEQ  string fun18() {  string symbol0 = strStack.top().value;  strStack.pop();  stateStack.pop();  BTreeNode\* newNode0 = new BTreeNode("LTEQ", symbol0);  treeStack.push(newNode0);  return "comparison-op";  }  // comparison-op->NE  string fun19() {  string symbol0 = strStack.top().value;  strStack.pop();  stateStack.pop();  BTreeNode\* newNode0 = new BTreeNode("NE", symbol0);  treeStack.push(newNode0);  return "comparison-op";  }  // comparison-op->RTEQ  string fun20() {  string symbol0 = strStack.top().value;  strStack.pop();  stateStack.pop();  BTreeNode\* newNode0 = new BTreeNode("RTEQ", symbol0);  treeStack.push(newNode0);  return "comparison-op";  }  // comparison-op->RT  string fun21() {  string symbol0 = strStack.top().value;  strStack.pop();  stateStack.pop();  BTreeNode\* newNode0 = new BTreeNode("RT", symbol0);  treeStack.push(newNode0);  return "comparison-op";  }  // simple-exp->simple-exp addop term  string fun22() {  BTreeNode\* newNode2 = treeStack.top();  strStack.pop();  stateStack.pop();  treeStack.pop();  BTreeNode\* newNode1 = treeStack.top();  strStack.pop();  stateStack.pop();  treeStack.pop();  BTreeNode\* newNode0 = treeStack.top();  strStack.pop();  stateStack.pop();  treeStack.pop();  BTreeNode\* rootNode = new BTreeNode("-1", "simple-exp");  rootNode->nodeList.push\_back(newNode1);  rootNode->nodeList.push\_back(newNode0);  rootNode->nodeList.push\_back(newNode2);  treeStack.push(rootNode);  return "simple-exp";  }  // simple-exp->term  string fun23() {  BTreeNode\* newNode0 = treeStack.top();  strStack.pop();  stateStack.pop();  treeStack.pop();  treeStack.push(newNode0);  return "simple-exp";  }  // addop->PLUS  string fun24() {  string symbol0 = strStack.top().value;  strStack.pop();  stateStack.pop();  BTreeNode\* newNode0 = new BTreeNode("PLUS", symbol0);  treeStack.push(newNode0);  return "addop";  }  // addop->MINUS  string fun25() {  string symbol0 = strStack.top().value;  strStack.pop();  stateStack.pop();  BTreeNode\* newNode0 = new BTreeNode("MINUS", symbol0);  treeStack.push(newNode0);  return "addop";  }  // term->term mulop factor  string fun26() {  BTreeNode\* newNode2 = treeStack.top();  strStack.pop();  stateStack.pop();  treeStack.pop();  BTreeNode\* newNode1 = treeStack.top();  strStack.pop();  stateStack.pop();  treeStack.pop();  BTreeNode\* newNode0 = treeStack.top();  strStack.pop();  stateStack.pop();  treeStack.pop();  BTreeNode\* rootNode = new BTreeNode("-1", "term");  rootNode->nodeList.push\_back(newNode1);  rootNode->nodeList.push\_back(newNode0);  rootNode->nodeList.push\_back(newNode2);  treeStack.push(rootNode);  return "term";  }  // term->factor  string fun27() {  BTreeNode\* newNode0 = treeStack.top();  strStack.pop();  stateStack.pop();  treeStack.pop();  treeStack.push(newNode0);  return "term";  }  // mulop->MULTIPLY  string fun28() {  string symbol0 = strStack.top().value;  strStack.pop();  stateStack.pop();  BTreeNode\* newNode0 = new BTreeNode("MULTIPLY", symbol0);  treeStack.push(newNode0);  return "mulop";  }  // mulop->DIVIDE  string fun29() {  string symbol0 = strStack.top().value;  strStack.pop();  stateStack.pop();  BTreeNode\* newNode0 = new BTreeNode("DIVIDE", symbol0);  treeStack.push(newNode0);  return "mulop";  }  // mulop->MOD  string fun30() {  string symbol0 = strStack.top().value;  strStack.pop();  stateStack.pop();  BTreeNode\* newNode0 = new BTreeNode("MOD", symbol0);  treeStack.push(newNode0);  return "mulop";  }  // factor->LPAN exp RPAN  string fun31() {  string symbol2 = strStack.top().value;  strStack.pop();  stateStack.pop();  BTreeNode\* newNode2 = new BTreeNode("RPAN", symbol2);  BTreeNode\* newNode1 = treeStack.top();  strStack.pop();  stateStack.pop();  treeStack.pop();  string symbol0 = strStack.top().value;  strStack.pop();  stateStack.pop();  BTreeNode\* newNode0 = new BTreeNode("LPAN", symbol0);  treeStack.push(newNode1);  return "factor";  }  // factor->NUMBER  string fun32() {  string symbol0 = strStack.top().value;  strStack.pop();  stateStack.pop();  BTreeNode\* newNode0 = new BTreeNode("NUMBER", symbol0);  treeStack.push(newNode0);  return "factor";  }  // factor->ID  string fun33() {  string symbol0 = strStack.top().value;  strStack.pop();  stateStack.pop();  BTreeNode\* newNode0 = new BTreeNode("ID", symbol0);  treeStack.push(newNode0);  return "factor";  }  // 定义一个存储函数指针的数组  string(\*funcArray[])() = { fun0,fun1,fun2,fun3,fun4,fun5,fun6,fun7,fun8,fun9,fun10,fun11,fun12,fun13,fun14,fun15,fun16,fun17,fun18,fun19,fun20,fun21,fun22,fun23,fun24,fun25,fun26,fun27,fun28,fun29,fun30,fun31,fun32,fun33 };  void process(const KeyValue& line) {  string key = line.key;  string value = line.value;  // 拿到当前栈的状态  int state = stateStack.top();  // 找到下一个状态字符串  string nextStateStr = SLRVector[state].m[key == "EOF" ? "$" : key];  int nextState;  switch (nextStateStr[0]) {  case 's': // 下一步  // 放入字符栈  strStack.push(line);  nextState = stoi(nextStateStr.substr(1));  stateStack.push(nextState);  break;  case 'r': // 要规约了  {  string res;  size\_t startPos = nextStateStr.find("("); // 查找左括号的位置  size\_t endPos = nextStateStr.find(")"); // 查找右括号的位置  // 如果找到了左右括号  if (startPos != string::npos && endPos != string::npos) {  startPos++; // 从左括号的下一个位置开始提取内容  res = nextStateStr.substr(startPos, endPos - startPos);  }  int i = grammarMap[res];  string left = funcArray[i](); // 调用对应的函数  state = stateStack.top();  nextStateStr = SLRVector[state].m[left];  nextState = stoi(nextStateStr);  stateStack.push(nextState);  strStack.push(KeyValue(left));  // 继续放入当前字符  process(line);  break;  }  case 'A': // ACCEPT  // 生成语法树  cout << "成功！";  break;  default:  cout << "状态表出错！";  }  }  int main() {  ifstream file("F:/个人资料/学习/计算机大三/2024 编译原理课程项目/上交材料/2. 测试文件夹/2. 任务二测试文本及程序/2. 程序生成文件/SLR1Str.txt");  stringstream buffer;  buffer << file.rdbuf();  string SLR1Str = buffer.str();  SLRVector = StringToSLRVector(SLR1Str);  // 初始状态为0  int state = 0;  stateStack.push(state);  vector<KeyValue> keyValuePairs = readKeyValuePairs("F:/个人资料/学习/计算机大三/2024 编译原理课程项目/上交材料/2. 测试文件夹/2. 任务二测试文本及程序/2. 程序生成文件/output.lex");  for (const auto& pair : keyValuePairs) {  process(pair);  }  string str = BTreeNodeToString(treeStack.top());  // 将自定义格式的字符串解析为 BTreeNode  istringstream iss(str);  ofstream outFile("F:/个人资料/学习/计算机大三/2024 编译原理课程项目/上交材料/2. 测试文件夹/2. 任务二测试文本及程序/2. 程序生成文件/tree.out");  if (outFile.is\_open())  {  outFile << str;  outFile.close();  }  return 0;  } |

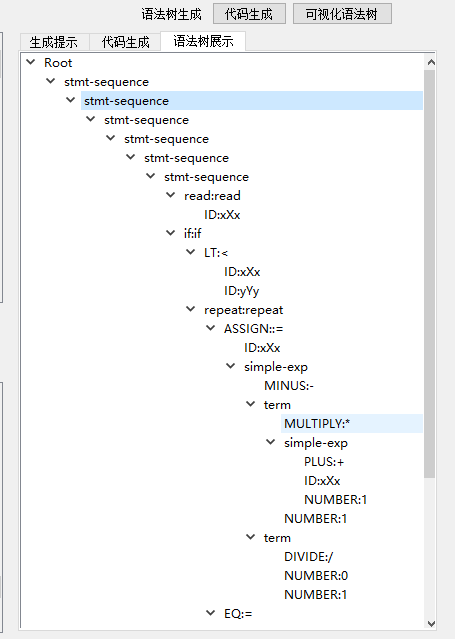
编译运行：



得到语法树：



**8、可视化语法树**



**测试结果**

任务二测试完全通过