

课程实验报告

课 程 名 称： 编译原理

实验项目名称： 上下文无关文法的DFA构建

专 业 班 级： 软件工程2201

姓 名： 滕明君

学 号： 202226010111

指 导 教 师： 黎文伟

完 成 时 间： 2025 年 04 月 26 日

信息科学与工程学院

|  |  |
| --- | --- |
| 实验题目：  上下文无关文法的DFA构建 | |
| 实验目的：   1. 学习和掌握算术表达式文法的LR语法分析 2. 构建TINY语言的完整语法分析器 3. LR(0)自动机构建 4. 语法分析表生成 | |
| 实验环境：  Cangjie、CodeArts IDE for Cangjie | |
| 实验内容及操作步骤：  **一、**  基于给出的数据结构，对文法写出下列功能函数的实现代码：  1) 产生式的FIRST函数求解；  2) 非终结符的FIRST函数求解；  3) 非终结符的FOLLOW函数求解；   1. 产生式的FIRST函数求解；   设计思路：函数从左至右遍历产生式体中的每个文法符号，区分终结符和非终结符处理。遇到终结符时，直接加入First集合并停止推导；遇到非终结符时，合并其First集合，并处理空符的情况。定义循环控制变量nullStand用于判断是否继续推导，当所有符号都可推导为空时添加空符。  代码的大致流程：初始化一个布尔变量nullStand为true，和一个索引`i`从0开始。然后进入一个循环，处理产生式体中的每个文法符号。如果是终结符，就将其加入FIRST集并停止继续处理；如果是非终结符，合并其FIRST集，并检查是否包含空（这里可能是ε）。如果处理完所有符号且nullStand仍为true，就将空符加入FIRST集。  // 核心功能：计算单个产生式的FIRST集合  public func get\_Production\_FIRST(production:Production) {      var nullStand:Bool = true; // 空推导状态标记      var i:Int64 = 0;      // 核心循环：遍历产生式符号      while(nullStand && i<production.pBodySymbolTable.size) {          var currGrammarSymbol = production.pBodySymbolTable[i];            // 关键点1：终结符处理          if(currGrammarSymbol is TerminalSymbol) {              production.pFirstSet.put(currGrammarSymbol as TerminalSymbol); // 直接加入集合              nullStand = false; // 终止推导          }          // 关键点2：非终结符处理          else {              var nts = currGrammarSymbol as NonTerminalSymbol;              production.pFirstSet.putAll(nts.pFirstSet); // 合并FIRST集合              production.pFirstSet.remove(ts\_null); // 临时移除空符                if(!nts.pFirstSet.contains(ts\_null)) {                  nullStand = false; // 发现不可推导为空的符号              }          }          i++;      }      // 关键点3：空符处理      if(nullStand && i == production.pBodySymbolTable.size) {          production.pFirstSet.put(ts\_null); // 所有符号可空时添加      }  }  2）非终结符的FIRST函数求解；  设计思路：函数首先初始化依赖表task\_table和完成集合accomplish\_set，然后进入多轮迭代处理，直到所有非终结符处理完毕。在每轮迭代中，遍历所有非终结符，检查它们的产生式首符号，处理终结符和非终结符的情况，并管理依赖关系。  假设产生式为 E → T E' ：  1. 首符号T是非终结符  2. 检查T是否在accomplish\_set中：  - 若T已完成：将T的FIRST集合合并到E的FIRST集合  - 若T未完成：将T添加到E的依赖表(task\_table)  3. 后续处理E'时，需判断T的FIRST集合是否包含ε  需要特别注意的要点 ：  1. 循环依赖处理 ：  - 通过 if(insert\_nts != nts) 防止非终结符依赖自身  - 使用 task\_table 跟踪未解决的依赖链，通过多轮迭代逐步解决  2. 空产生式处理 ：  - 需要特别注意当产生式体可为空时，需要将ε（ts\_null）加入FIRST集合  - 在合并其他非终结符的FIRST集时需保留ε传播的可能性  public func get\_All\_NTS\_FIRST(){  // ===== 依赖关系管理 =====  // 创建依赖关系表：记录每个非终结符的未解决依赖项  var task\_table:HashMap<NonTerminalSymbol, HashSet<NonTerminalSymbol>> = HashMap<NonTerminalSymbol, HashSet<NonTerminalSymbol>>();  // ... existing code ...  // ===== 多轮迭代处理 =====  // 循环直到所有非终结符完成计算（终止条件：完成集合大小等于总集合大小）  while(accomplish\_set.size != all\_NonTerminalSymbol\_Set.size) {  for(nts in all\_NonTerminalSymbol\_Set) {  var independent = true; // 独立性标志位（当前NT是否无未解决依赖）    // ===== 产生式处理 =====  for(production in nts.pProductionTable) {  // 核心处理逻辑：分析产生式首符号  var grammarSymbol:GrammarSymbol = production.pBodySymbolTable[0];    if(grammarSymbol is NonTerminalSymbol) {  // 处理非终结符依赖链  if(accomplish\_set.contains(insert\_nts)) {  // 依赖项已完成：合并FIRST集合并更新依赖表  nts.pFirstSet.putAll(insert\_nts.pFirstSet);  task\_table[nts].remove(insert\_nts);  } else {  // 检测并处理循环依赖（insert\_nts != nts）  // ... existing code ...  independent = false; // 存在未解决依赖  }  } else {  // 首符号为终结符：直接加入当前NT的FIRST集合  // ... existing code ...  }  }    // ===== 状态更新 =====  // 当所有产生式处理完毕且无依赖时，标记为已完成  if(independent) {  accomplish\_set.put(nts);  }  }  }  }  3）非终结符的FOLLOW函数求解；  设计思路：  将结束符$加入开始符号的FOLLOW  A → αBβ：将FIRST(β)（不含ε）加入B的FOLLOW  A → αB 或 A → αBβ(β→ε)：将FOLLOW(A)加入FOLLOW(B)  代码分为初始化阶段和依赖解析阶段。在初始化阶段，代码遍历所有非终结符及其产生式，建立依赖关系。反向扫描产生式体，从右向左处理符号，合并右侧符号的FIRST集合到当前非终结符的FOLLOW集合中，并处理可能的依赖关系。在依赖解析阶段，使用拓扑排序处理正常依赖，并在检测到循环依赖时进行特殊处理，如强制合并相关FOLLOW集合并标记为已完成。  伪代码：  while 存在未完成的非终结符:  for 每个产生式 A → X1X2...Xn:  for i from n downto 1:  if Xi 是非终结符:  # 处理右侧符号  temp = FIRST(Xi+1...Xn) - {ε}  Xi.FOLLOW |= temp  if ε ∈ FIRST(Xi+1...Xn):  Xi.FOLLOW |= A.FOLLOW  # 记录依赖关系  if A.FOLLOW 未完成:  Xi.dependencies.add(A)  需要特别注意的要点 ：  代码维护了一个依赖图，并通过不断移除已完成的节点来处理依赖关系。但当存在循环依赖时，拓扑排序无法进行，因此需要额外的处理机制。这里的处理方式是在检测到依赖边数量未减少时，强制选择一个未完成的非终结符，合并其相关FOLLOW集合，这可能在某些情况下导致结果不准确，但确保了算法的终止。  代码中的依赖关系管理需要确保不会出现自环边，即非终结符不能依赖自身。这一点通过条件判断temp\_nts != nts来实现，避免了无限循环的产生。  public func get\_All\_NTS\_FOLLOW(){  // ===== 初始化阶段 =====  // 遍历所有非终结符及其产生式，建立FOLLOW依赖关系  for(nts in all\_NonTerminalSymbol\_Set) {  // ... existing code ...    // 反向扫描产生式体（从右向左处理符号）  while(i>=0) {  if(curr\_gs is NonTerminalSymbol) {  // ===== 核心处理逻辑 =====  // 1. 合并右侧符号的FIRST集合（不含空符）  var k:Int64 = i+1;  while(k<=j) {  // 合并右侧符号的FIRST集合到当前非终结符的FOLLOW  temp\_nts.pFollowSet.putAll(FIRST(insert\_k\_gs));  temp\_nts.pFollowSet.remove(ts\_null);  k++;  }  // 2. 动态调整处理区间边界  if(!temp\_nts.pFirstSet.contains(ts\_null)) {  j = i; // 当右侧符号无法推导为空时缩小处理范围  }  // 3. 建立FOLLOW集合的依赖关系  if(nullStand && temp\_nts != nts) {  // 当后续符号都可推导为空时，建立非终结符间的FOLLOW依赖  temp\_nts.pDependentSetInFollow.put(nts);  }  }  // ... existing code ...  }  }  // ===== 依赖解析阶段 =====  // 使用拓扑排序处理正常依赖  while(accomplishment.size != all\_NonTerminalSymbol\_Set.size) {  // ... existing code ...    // ===== 循环依赖处理 =====  if(curr\_dependence\_edge\_num == before\_dependence\_edge\_num) {  // 使用BFS遍历所有相关非终结符  while(!task\_queue.isEmpty()) {  // 广度优先遍历依赖链中的非终结符  var headNts:NonTerminalSymbol = task\_queue[0];  // ... existing code ...  }    // 强制合并循环依赖中的所有FOLLOW集合  for(cNTS in has\_show\_nts) {  deal\_nts.pFollowSet.putAll(cNTS.pFollowSet);  }    // 强制标记为已完成并更新依赖计数  accomplishment.put(deal\_nts);  curr\_dependence\_edge\_num -= dependence\_map[deal\_nts].size;  }  }  }  **二、**  基于给出的数据结构，就LR语法分析写出下列功能函数的实现代码：  1) 一个项集中LR(0)核心项的闭包求解，即实现函数：  void getClosure( ItemSet \*itemSet)；  2) 穷举一个LR(0)项集的变迁，其中包括驱动符的穷举，后继项集的创建，后继项集中核心项的确定，后继项集是否为新项集的判断。即实现函数：  void exhaustTransition(ItemSet \*itemSet)；  3) 文法的LR(0)型DFA求解；  4) 文法是否为SLR(1)文法的判断；  5) LR语法分析表的填写；  1）一个项集中LR(0)核心项的闭包求解，即实现函数：  void getClosure( ItemSet \*itemSet)；  闭包：对于每个项A→α·Bβ，需要将B的所有产生式B→γ的项B→·γ加入闭包。这需要遍历所有相关的非终结符及其产生式。  设计思路：  闭包计算的核心是处理每个项点后的非终结符，将其所有产生式加入闭包，并继续处理这些产生式中的非终结符。BFS确保所有可能的产生式都被覆盖，避免重复处理。  伪代码：  # 伪代码流程  def closure(I):  queue = deque()  # 初始化队列：收集I中所有点号后的非终结符  for item in I:  if item.dot后的符号是B且B是非终结符:  queue.append(B)    # BFS处理  while queue:  B = queue.popleft()  for B→γ in B的所有产生式:  新项 = B→·γ  if 新项不在闭包中:  加入闭包  if γ的第一个符号是C且C是非终结符:  queue.append(C)  return 闭包  需要特别注意的要点 ：  1. 双阶段架构 ：初始化阶段收集核心项符号，BFS阶段展开产生式  2. 防重复机制 ：hasShowSet确保每个非终结符只处理一次  3. 项生成规则 ：为非终结符的每个产生式创建 B→·γ 形式的新项  4. 广度优先优势 ：保证闭包项的生成顺序符合规范要求  5. 类型安全处理 ：通过 is 和 as 运算符确保符号类型转换安全  public func getClosure(itemSet: ItemSet){  // ===== 初始化阶段 =====  // 创建防重复集合和BFS队列  var hasShowSet:HashSet<NonTerminalSymbol> = HashSet<NonTerminalSymbol>();  var prepare\_queue:ArrayList<NonTerminalSymbol> = ArrayList<NonTerminalSymbol>();    // 遍历核心项收集初始非终结符  for(item in itemSet.pItemTable) {  // ... existing code ...  // 核心逻辑：仅处理点号后符号为非终结符的情况  if(curr\_symbol is NonTerminalSymbol) {  // ... existing code ...  }  }  // ===== BFS展开阶段 =====  while(!prepare\_queue.isEmpty()) {  var curr\_NTS:NonTerminalSymbol = prepare\_queue[0];  prepare\_queue.remove(0);  // 遍历当前非终结符的所有产生式  for(production in curr\_NTS.pProductionTable) {  // 核心操作：生成点号在首部的新项  var item:LR0Item = LR0Item(curr\_NTS,production,0,"NONCORE");  itemSet.pItemTable.append(item);  // 处理新项首符号的非终结符  var curr\_symbol:GrammarSymbol = production.pBodySymbolTable[0];  if(curr\_symbol is NonTerminalSymbol) {  // ... existing code ...  // 核心逻辑：发现新非终结符时入队继续展开  if(!hasShowSet.contains(temp\_NTS)) {  // ... existing code ...  }  }  }  }  }  2) 穷举一个LR(0)项集的变迁，其中包括驱动符的穷举，后继项集的创建，后继项集中核心项的确定，后继项集是否为新项集的判断。即实现函数：  void exhaustTransition(ItemSet \*itemSet)；  3) 文法的LR(0)型DFA求解；  状态转移规则 ：  对每个项集I和文法符号X：  创建新项集J = GOTO(I,X)  J包含所有[A→αX·β]形式的项  计算CLOSURE(J)  # 伪代码流程  def build\_automaton():  queue = deque([closure(I0)]) # 初始状态  visited = set()    while queue:  I = queue.popleft()  for X in 所有可能符号:  J = goto(I, X)  if J not in visited:  visited.add(J)  queue.append(J)  record\_transition(I, X, J)  设计思路：  函数使用BFS队列transition\_queue来处理状态项集，初始项集L0被放入队列。然后，遍历队列中的每个项集，收集驱动符号（非终结符和终结符），生成新的项集，并处理状态转移。  关键实现细节：  1. 驱动符号分类：     - 使用drive\_NTS和drive\_TS分别存储不同类型符号     - 通过类型检查(is运算符)实现自动分类  2. 新项集生成：     for(nts in drive\_NTS) {         // 遍历匹配相同非终结符的项         if(curr\_Nts == nts) {             // 创建点号后移的新核心项         }     }  3. 状态去重机制：     if(!contains(Inew)) {         // 计算闭包并注册新状态         getClosure(Inew);         pItemSetTable.append(Inew);     }  public func exhaustTransition(itemSet0:ItemSet) {  // ===== 自动机构建入口 =====  var transition\_queue = ArrayList<ItemSet>();  transition\_queue.append(itemSet0); // 初始化队列放入L0  // ===== 广度优先处理状态 =====  while(!transition\_queue.isEmpty()){  var itemSet = transition\_queue[0];  transition\_queue.remove(0);  // ===== 驱动符号收集阶段 =====  var drive\_NTS = HashSet<NonTerminalSymbol>();  var drive\_TS = HashSet<TerminalSymbol>();  for(item in itemSet.pItemTable) {  // 核心逻辑：仅处理可移入项（点号不在末尾）  if(item.dotPosition < item.production.pBodySymbolTable.size) {  var curr\_symbol = item.production.pBodySymbolTable[item.dotPosition];  // 分类收集驱动符号  if(curr\_symbol is NonTerminalSymbol) {  // ... existing code ...  } else {  // ... existing code ...  }  }  }  // ===== 非终结符转移处理 =====  for(nts in drive\_NTS) {  // 核心操作：生成点号后移的核心项  var Inew = ItemSet(pItemSetTable.size,ArrayList<LR0Item>());  for(item in itemSet.pItemTable) {  if(item.dotPosition < item.production.pBodySymbolTable.size) {  var curr\_Symbol = item.production.pBodySymbolTable[item.dotPosition];  if(curr\_Symbol is NonTerminalSymbol && curr\_Symbol == nts) {  // 创建点号后移的新核心项  var new\_item = LR0Item(..., item.dotPosition+1, "CORE");  Inew.pItemTable.append(new\_item);  }  }  }  // ===== 新状态注册机制 =====  if(!contains(Inew)) {  getClosure(Inew); // 核心操作：计算闭包  pItemSetTable.append(Inew);  transition\_queue.append(Inew);  }  pTransitionEdgeTable.append(...); // 记录转移边  }  // ===== 终结符转移处理（与非终结符对称逻辑）=====  // ... existing code ...  }  }  4) 文法是否为SLR(1)文法的判断；  SLR(1)冲突定义 ：  - 当同一项集中出现以下情况时存在冲突：  - 移进符集合(S) ∩ 规约符FOLLOW集(R) ≠ ∅  - 多个规约项的FOLLOW集存在交集  伪代码：  # 伪代码流程  def check\_conflict():  for 每个项集I in 所有LR(0)项集:  移进符集合 = {a | [A→α·aβ] ∈ I}  规约符集合 = [FOLLOW(B) | [B→γ·] ∈ I]    if 移进符集合与任意FOLLOW集有交集:  return False  if 任意两个FOLLOW集有交集:  return False  return True  设计思路：  代码遍历每个LR(0)项集，收集所有可能的移进和规约动作的终结符集合，然后检查这些集合之间是否有交集。如果有交集，说明存在冲突，文法不是SLR(1)；否则是。  代码使用双重循环，外层遍历项集，内层处理每个项。对于每个项，如果是可移进项，收集对应的终结符；如果是可规约项，添加其FOLLOW集。然后调用judgeSetHasUnion检查冲突。  核心检测流程分解：  1. 冲突集合初始化：  - drive\_nt\_set收集移进动作的终结符（可移入项）  - all\_Set存储所有可能冲突的集合（移进符+各规约符FOLLOW集）  2. 双重循环检测：  var hasSame = judgeSetHasUnion(all\_Set); // 检测集合两两交集  public func judge\_SLR1():Bool {  // ===== 遍历所有LR(0)项集 =====  for(itemSet in pItemSetTable) {  // 冲突检测数据结构初始化  var all\_Set = ArrayList<HashSet<TerminalSymbol>>(); // 存储所有可能冲突的符号集合  var drive\_nt\_set = HashSet<TerminalSymbol>(); // 移进动作的终结符集合  all\_Set.append(drive\_nt\_set); // 首元素为移进符集合  // ===== 遍历项集中每个项目 =====  for(item in itemSet.pItemTable) {  // 可移进项处理（点号不在末尾）  if(item.dotPosition < item.production.pBodySymbolTable.size) {  var currSymbol = item.production.pBodySymbolTable[item.dotPosition];  // 收集移进动作的终结符  if(currSymbol is TerminalSymbol) {  var currTs = currSymbol as TerminalSymbol ?? none\_TerminalSymbol;  if(currTs != none\_TerminalSymbol) {  drive\_nt\_set.put(currTs); // 核心操作：记录移进符  }  }  }  // 可规约项处理（点号在末尾）  else {  var currNts = item.nonTerminalSymbol;  all\_Set.append(currNts.pFollowSet); // 核心操作：添加FOLLOW集合  }  }  // ===== 冲突检测 =====  var hasSame = judgeSetHasUnion(all\_Set);  if(hasSame) return false; // 发现冲突立即返回  }  return true; // 全部项集无冲突  }  5) LR语法分析表的填写；  设计思路：  遍历所有LR(0)项集  对每个项集I：  移进项：收集所有可能的移进符号（终结符）  规约项：收集对应非终结符的FOLLOW集  填充ACTION表：  移进符号→s+目标状态  FOLLOW集符号→r+产生式编号  填充GOTO表：  非终结符→目标状态  public func create\_LR1\_Analysis\_Table(){  // ===== 初始化阶段 =====  // 获取接受状态（acc状态）  var accItemSet:ItemSet = getAcc();  // 初始化ACTION和GOTO表数据结构  for(itemSet in pItemSetTable) {  pActionCellTable.put(itemSet,HashMap<TerminalSymbol, ActionCell>()); // 每个状态的ACTION表（终结符→动作）  pGOTOCellTable.put(itemSet,HashMap<NonTerminalSymbol, GotoCell>()); // 每个状态的GOTO表（非终结符→转移）  }  // ===== 处理状态转移边 =====  for(edge in pTransitionEdgeTable) {  var currGS:GrammarSymbol = edge.driverSymbol;    // 终结符处理（移进动作）  if(currGS is TerminalSymbol) {  var currTS:TerminalSymbol = currGS as TerminalSymbol ?? none\_TerminalSymbol;  if(currTS != none\_TerminalSymbol) {  // 核心操作：创建移进动作（s+目标状态）  var new\_ac:ActionCell = ActionCell(edge.fromItemSet.stateId,edge.driverSymbol.name,"s",edge.toItemSet.stateId);    // 特殊处理接受状态  if(edge.toItemSet.stateId == accItemSet.stateId) {  new\_ac.actionType = "acc"; // 标记为接受动作  }  pActionCellTable[edge.fromItemSet].put(currTS,new\_ac)  }  }  // 非终结符处理（GOTO转移）  else {  var currNTS:NonTerminalSymbol = currGS as NonTerminalSymbol ?? none\_NonTerminalSymbol;  if(currNTS != none\_NonTerminalSymbol) {  // 核心操作：记录GOTO转移  var new\_gt:GotoCell = GotoCell(edge.fromItemSet.stateId,currNTS.name,edge.toItemSet.stateId);  pGOTOCellTable[edge.fromItemSet].put(currNTS,new\_gt);  }  }  }  // ===== 处理规约项 =====  for(itemSet in pItemSetTable) {  for(item in itemSet.pItemTable) {  // 检测规约项（点号在产生式末尾）  if(item.dotPosition == item.production.pBodySymbolTable.size) {  var currHeadNTS:NonTerminalSymbol = item.nonTerminalSymbol;  // 核心操作：遍历FOLLOW集添加规约动作  for(ts in currHeadNTS.pFollowSet) {  var new\_ac:ActionCell = ActionCell(itemSet.stateId,ts.name,"r",item.production.productionId);  pActionCellTable[itemSet].put(ts,new\_ac);  }  }  }  }  }  **三、**  首先以算术运算表达式的文法来验证程序代码的正确性，然后再以TINY语言的文法来验证程序代码的正确性，并得出TINY语言的语法分析表。  文法定义层  // 典型文法构造过程（以算术表达式为例）  public func create\_Arithmetic\_Grammar() {      // 非终结符层次构建      var nts\_E = NonTerminalSymbol("E","NONTERMINAL");      var nts\_T = NonTerminalSymbol("T","NONTERMINAL");        // 运算符终结符定义      var ts\_plus = TerminalSymbol("+","TERMINAL","OPERATOR");        // 产生式定义（体现运算符优先级）      var production = Production(1,3);      production.pBodySymbolTable.append(nts\_E, ts\_plus, nts\_T);  }   * 采用面向对象方式管理符号体系 * 通过产生式层级体现运算符优先级 * 全局符号表实现符号统一管理   自动机构建层  public func create\_Arithmetic\_Grammar\_LR0\_DFA() {      // 初始化核心项      var I0\_0\_item = LR0Item(...);        // 项集闭包计算      getClosure(I0);        // 状态转移构建      exhaustTransition(I0);  }   * 实现LR(0)项集闭包算法 * 自动生成DFA状态转移关系 * 可视化展示项集族   语法分析层  // 状态栈驱动逻辑  while(!pStateStack.isEmpty()) {      // ACTION/GOTO表查询      var indicator = pActionCellTable[...];        // 移进-规约决策      match indicator.actionType {          case "s": handleShift();          case "r": handleReduce();      }  }   * 基于状态栈的移进-规约机制 * 双重哈希表实现快速ACTION/GOTO查询 * 实时状态追踪与错误中断   二、核心数据结构    class TerminalSymbol {          +String name          +String type          +String category      }        class NonTerminalSymbol {          +String name          +ArrayList~Production~ pProductionTable          +HashSet~TerminalSymbol~ pFollowSet      }        class Production {          +Int64 id          +ArrayList~Symbol~ pBodySymbolTable      }        class LR0Item {          +NonTerminalSymbol head          +Production production          +Int64 dotPos      }        TerminalSymbol <|-- NonTerminalSymbol      NonTerminalSymbol "1" \*-- "\*" Production      Production "1" o-- "\*" Symbol      LR0Item --> Production  三、关键算法实现  FIRST/FOLLOW集计算  // 非终结符FIRST集计算  public func get\_All\_NTS\_FIRST() {      // 递归处理空产生式      // 处理终结符首符号  }  // FOLLOW集传播算法  public func get\_All\_NTS\_FOLLOW() {      // 处理产生式右部符号关系      // 实现集合传播规则  }  SLR冲突检测  public func judge\_SLR1() {      // 遍历所有状态项集      // 检查FOLLOW集交集情况      // 返回是否满足SLR(1)条件  }  四、设计亮点  双文法支持架构  同时支持算术表达式和TINY语言文法  通过create\_Arithmetic\_Grammar/create\_TINY切换模式  可扩展产生式系统  // 动态添加产生式示例  nts\_E.pProductionTable.append(      Production(10,2)          .addSymbol(nts\_E)          .addSymbol(ts\_power)          .addSymbol(nts\_F)  );  可视化调试接口  // 输出产生式FIRST集  show\_All\_Production\_FIRST();  // 展示DFA状态转移图  show\_All\_itemSet();  该实现完整实现了从文法定义到语法分析的完整工具链，采用模块化设计便于教学演示和算法验证，其分层架构和面向对象设计为后续扩展LL(1)分析器等组件奠定了良好基础。  收获与体会：  一、理论落地能力提升  1. LR分析具象化 通过实现 create\_Arithmetic\_Grammar\_LR0\_DFA 方法，真正理解了项集闭包计算过程。调试时看到控制台打印的 I0: [E'→·E$] 等状态项，原本抽象的LR(0)自动机概念瞬间具象化。  2. 文法冲突检测 在 judge\_SLR1 方法中处理FOLLOW集交集问题时，深刻体会到教材中"两个产生式FOLLOW集不能相交"的理论约束。当测试用例 num\*(num+num) 触发规约冲突时，通过修改产生式优先级解决问题，这种实战经验非常宝贵。  二、工程实践突破  1. 符号表架构设计 创建 TerminalSymbol 和 NonTerminalSymbol 类时，最初纠结于是否要分离符号类型。当实现到 get\_All\_NTS\_FOLLOW 方法时，发现通过继承体系管理符号属性能显著简化集合运算，这种设计取舍经验是课堂上学不到的。  2. 调试技巧进阶 在 judge\_Sentence\_LR\_Grammar 中设置状态栈打印  通过观察状态跳转序列，成功定位了产生式编号映射错误，这种调试方法论对后续开发复杂系统至关重要。  三、认知维度拓展  1. 工具链认知 通过实现 show\_All\_itemSet 可视化方法，认识到编译器前端各阶段的衔接关系。当看到完整的DFA状态图时，突然理解到Yacc等工具的内部工作原理。  2. 错误处理哲学 在 judge\_Sentence\_LR\_Grammar 中设计立即失败策略：  这种"快速失败"的设计理念，改变了以往过度追求容错率的思维定式，对软件工程质量有了新认知。 | |
| 实  验成绩 |  |