

**院 系：计 算 机 学 院**

**实验课程：编译原理**

**实验项目：LR(1)分析生成器**

**指导老师：黄煜廉**

**开课时间：2024 ～ 2025年度第 1学期**

**专 业：网络工程**

**班 级：网工二班**

**学 生：肖翔**

**学 号：20222132002**

**华南师范大学教务处**

|  |
| --- |
| 一、实验内容  1.必做功能：  (1)要提供一个文法输入编辑界面，让用户输入文法规则（可保存、打开存有文法规则的文件）  (2)求出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。【可以采用表格的形式呈现】  (3)需要提供窗口以便用户可以查看文法对应的LR(0)DFA图。（可以用画图的方式呈现，也可用表格方式呈现该图点与边数据）  (4)判断该文法是否为SLR(1)文法。（应提供窗口呈现判断的结果，如果不是SLR（1）文法，需要在窗口中显示其原因）  (5)需要提供窗口以便用户可以查看文法对应的LR(1)DFA图。（可以用画图的方式呈现，也可用表格方式呈现该图点与边数据）  (6)需要提供窗口以便用户可以查看文法对应的LR(1)分析表。【LR(1)分析表采用表格的形式呈现】  (7)应该书写完善的软件文档  (8)应用程序应为Windows界面。  2.选做功能。  (1)需要提供窗口以便用户输入需要分析的句子。  (2)需要提供窗口以便用户查看使用LR(1)分析该句子的过程。【可以使用表格的形式逐行显示分析过程】 |
| 二、实验目的  （1）要提供一个文法规则编辑的界面，以让用户输入文法规则（可输入，可保存、可打开源程序）。  （2）可由用户选择是否生成对应语法规则的First集、Follow集、LR(0)DFA图、LR(1)DFA图、LR(1)分析表。  （3）实验4的实现只能选用的程序设计语言为：C++。  （4）要求应用程序的操作界面应为Windows界面。  （5）应该书写完善的软件文档。 |
| 三、实验文档：  （一）系统概述  1. 系统结构  系统分为7个模块：文法规则输入处理模块、求First集合模块、求Follow集合模块、生成LR(0)DFA图模块、生成LR(1)DFA图模块、生成LR(1)分析表模块以及分析是否为SLR(1)文法模块。  2. 数据结构的选择    图1 系统数据结构（1）    图2 系统数据结构（2）    图3 系统数据结构（3）    图4 系统数据结构（4）    图5 系统数据结构（5）  本系统主要使用了结构体struct、向量vector、集合set、映射map、队列queue等数据结构。以下是对本系统部分数据结构的详细介绍。  （1）grammarMap2用于表示不同终结符对应的文法规则。比如对于以下文法规则，grammarMap2[“Start”]这个集合将包含两个向量[“TESTA”]和[“TESTB”]，这意味着，非终结符Start对应的规则右部是TESTA和TESTB。  Start -> TESTA  Start -> TESTB  TESTA -> a  TESTB -> b  （2）grammarUnit用于表示单条文法规则，其中的gid表示当前文法规则的编号，left表示文法规则的左部，right表示文法规则的右部。  （3）grammarDequeue是一个存储grammarUnit类型数据的队列，它为grammarToInt服务。  （4）grammartoInt用于查找文法规则下标。通过遍历grammarDequeue建立文法单元到编号的映射，即为文法规则生成对应的下标。  （5）firstSets和followSets分别表示非终结字符对应的first集和follow集，firstUnit中的isEpsilon则表示非终结符对应的first集中是否含有空字符串（‘@’）。  （6）dfaCell用于表示LR(0)DFA图状态中的项目，比如某状态中的term->term .mulop factor就表示一个项目，他可以用dfaCell表示。gid是文法编号（grammar id）。它主要用于关联文法产生式和DFA中的项目。cellid是dfaCell自身的编号。它用于在dfaCellVector（存储所有dfaCell的向量）中唯一标识一个dfaCell结构。  假设存在两个不同的dfaCell结构，它们可能对应相同的文法产生式（即gid相同），但它们在DFA中的位置或者其他属性可能不同，此时它们的cellid是不同的。  比如，对于文法产生式A -> aB，在DFA构建过程中，可能在不同的状态中有两个项目都和这个产生式相关，一个是A ->.aB，另一个是A -> a.B。这两个项目的gid相同（因为都对应文法产生式A -> aB），但它们的cellid不同，用于在dfaCellVector中区分这两个不同的项目。  （7）dfaState表示LR(0)DFA图的状态。sid用于唯一标识一个DFA的状态。在整个LR(0)分析过程中，每个状态都有一个不同的编号，方便在构建和遍历DFA图时进行区分和引用。例如，在后续生成状态转移关系以及判断状态是否重复等操作中，通过这个id来准确地操作对应的状态。  originV这个向量存储的是在求闭包操作之前，该状态所包含的项目（用dfaCell表示）的编号信息。在构建DFA状态的过程中，最初放入的项目编号会先记录在这里， originV保留了这个初始的 “底子”，便于一些对比和处理操作，比如判断状态是否重复时会用到其初始状态下项目编号的情况。  cellV存储当前状态所包含的所有项目编号，但它是在整个状态构建完成（包括经过闭包操作等一系列处理后）最终的项目编号集合。可以理解为是这个状态所涵盖的所有LR(0)项目的一个汇总表示，通过这些项目编号能够在dfaCellVector（存储所有dfaCell的向量）中找到对应的具体项目内容。  isEnd用于指示这个状态是否为规约状态，即当前状态是否含有规约项。  nextStateVector用于存储从当前状态出发，通过不同字符可以转移到的下一个状态的相关信息。每个nextStateUnit包含了转移所使用的字符以及目标状态的id，这样就完整地描述了当前状态的状态转移关系，从而构建起整个DFA的状态转移图结构。  right\_VNs用于存储某一状态内已经生成过闭包的非终结符。避免对同一个非终结符重复生成闭包。  （8）生成LR(1)DFA图所用到的结构和LR(0)的类似，不同的是LR(1)的项目结构多了一个属性，即lookahead，用于存储当前项目的向前搜索符。  （9）LR1TableUnit是LR(1)分析表的单元结构，包含动作（action）和去向（goto）。动作，以终结符为键，对应动作字符串（如移进 "s<状态编号>"、规约 "r<文法编号>"、接受 "ACCEPT" 等）为值。去向，以非终结符为键，对应状态编号为值（对于非终结符的转移情况）。  （二）实验过程  1. 求First集  求First集的算法采用讲义中提到的算法（图 6）。    图6 求first集伪代码  具体代码：    图7 求first集代码（1）    图8 求first集代码（2）  此代码的核心部分是遍历产生式右部以计算First集合，即for (auto& g : grammar.second)的复合语句。对于当前非终结符的每个产生式右部g进行遍历，然后通过一个循环遍历产生式右部g的每个字符或字符串（因为非终结符有可能是以单词的形式出现），索引为k。对于每个字符或字符串g[k]，分情况处理：  ①如果字符是'@'（表示空字符串），则跳过该字符，继续处理下一个字符；  ②如果是终结符（通过isTerminal函数判断），则将该终结符插入到一个临时的First集合first\_k中；  ③如果是非终结符，则获取该非终结符已有的First集合（first\_k = firstSets[g[k]].s;）。如果当前字符是终结符或者当前字符所对应的非终结符的First集合中不包含空字符串，那么就可以跳出当前对产生式右部的遍历循环（因为后面的字符对当前非终结符的First集合计算已经没有影响了）。假设有文法规则A -> BC，非终结符A的文法规则中含非终结符B打头，如果B含有空串，则不能跳过，需要扫描B后面的字符C，将字符C的first集合加入到A的first集合当中。  getFirstSets函数的目的是通过不断调用calculateFirstSets函数来迭代计算每个非终结符的First集合，直到所有非终结符的First集合都不再发生变化为止。  2. 求Follow集  求Follow集的算法采用讲义中提到的算法（图 9）。    图9 求follow集伪代码  具体代码：    图10 求follow集代码（1）    图11 求follow集代码（2）  此代码的核心部分是遍历产生式右部以计算Follow集合。  for (auto& g : grammar.second)：对于当前非终结符的每个产生式右部g进行遍历。  for (int i = 0; i < g.size(); ++i)：通过一个内层循环遍历产生式右部g的每个字符，索引为i。对于每个字符g[i]，分情况处理：   1. 如果字符是终结符（通过isTerminal函数判断）或者是空字符串（'@'），则跳过该字符，继续处理下一个字符。 2. 如果当前字符是产生式右部的最后一个字符（即i == g.size() - 1），则进入Case A情况：将当前非终结符nonTerminal的Follow集合中的所有字符插入到一个临时的Follow集合follow\_k中。这是因为在产生式A -> αB（这里B对应g[i]）的情况下，B的Follow集合应该包含A的Follow集合。 3. 如果当前字符不是产生式右部的最后一个字符，则进入Case B情况：首先定义int j = i + 1;，用于从当前字符的下一个字符开始遍历。通过一个内层循环遍历当前字符后面的字符序列。对于每个后续字符g[j]，分情况处理：   （1）如果是终结符，直接将该终结符插入到follow\_k中，并跳出内层循环（因为后面的字符对当前处理已经没有影响了）。  （2）如果是非终结符，首先获取该非终结符的First集合（set<char> first\_beta = firstSets[g[j]].s;），然后将其插入到follow\_k中。接着，如果该非终结符的First集合中不包含空字符串（if (!firstSets[g[j]].isEpsilon)），则跳出内层循环（因为后面的字符对当前处理已经没有影响了）。否则，继续遍历下一个字符（++j;）。换言之，如果有空串在g[j]的first集合中，g[j]就有成为空串的可能，此时相当于g[i]后面跟着的是g[j]后面的字符，当然g[j]后面也可能没有字符，此时g[i]相当于最后一个字符。无论那种情况都要++j，读取g[j]后一位。如果g[j]后面有字符就会继续循环（符合j < g.size()的循环条件），将g[j]后一个字符（即++j后得到的g[j]字符）的first集合加进g[i]的follow。如果g[j]后面没有字符，那么++j后，j等于文法规则的长度，不符合循环条件，退出循环。然后将非终结符的follow集加入到g[i]的follow集中。  （3）如果内层循环遍历完整个后续字符序列（即j == g.size()），说明后续字符序列可能产生空字符串或者全部可空，此时将当前非终结符nonTerminal的Follow集合中的所有字符插入到follow\_k中。这是因为在产生式A -> αBβ（这里B对应g[i]）的情况下，如果β可空，那么B的Follow集合应该包含A的Follow集合。  3. 生成LR(0)DFA图    图11 求LR(0)DFA图的辅助函数  isNewCell函数的目的是判断给定的文法编号gid和点（.）的位置索引index所表示的dfaCell项目是否是新出现的。通过遍历已有的dfaCellVector，如果找到了相同gid和index的项目，就返回其已有的编号，说明不是新结构；如果遍历完都没找到，则返回-1，表示是新结构，后续需要创建新的dfaCell并添加到dfaCellVector中。  isNewState函数用于判断给定的项目编号集合（通过cellIds表示）所代表的状态是否是新的。通过遍历已有的dfaStateVector，对比每个状态的originV（未闭包前的项目编号集合）与传入的cellIds是否完全相同，如果相同则返回该已有状态的sid（编号），说明不是新状态；若遍历完都没有匹配的，则返回-1，意味着是新状态，后续要创建新的dfaState并添加到dfaStateVector中。    图12 求LR(0)DFA图的第一个状态  这个函数用于创建LR(0)的初始状态。首先创建一个空的dfaState结构体实例zero，并给它分配一个唯一的状态编号（通过scnt++实现），然后将其添加到dfaStateVector中。接着创建一个代表初始LR(0)项目（通常是增广后的开始符号产生式，这里假设为E' ->.S，对应的文法编号设为 0，点在最开始位置，索引为 0）的dfaCell结构体实例startCell，给它分配一个项目编号（通过ccnt++），并添加到dfaCellVector中。最后将这个初始项目的编号添加到刚创建的初始状态dfaStateVector[0]的cellV和originV向量中，完成初始状态的构建，它是整个DFA构建的起点。    图13 求LR(0)DFA图核心代码（1）    图14 求LR(0)DFA图核心代码（2）    图15 求LR(0)DFA图核心代码（3）    图16 求LR(0)DFA图核心代码（4）  generateLR0State函数是整个LR(0)状态生成的核心逻辑，通过深度优先搜索（DFS）的方式递归地构建DFA的各个状态以及状态之间的转移关系。  此函数的代码逻辑可分为以下几个部分：  ①避免重复访问。首先通过visitedStates集合来记录已经访问过的状态编号，如果当前传入的stateId已经在该集合中，说明这个状态已经处理过了，直接返回，避免重复构建和陷入死循环。否则，将当前状态编号插入visitedStates集合，表示已访问。  ②求闭包操作。遍历当前状态（由stateId指定）的cellV向量中的每个项目（dfaCell）。对于每个项目，如果点（.）在产生式末尾或者产生式右部为空串，就将当前状态的isEnd标志设为true，表示当前状态涉及规约操作，然后跳过该项目。如果点后面的下一个符号（通过grammarDeque[currentCell.gid].right[currentCell.index]获取）是一个非终结符，并且这个非终结符还没有在当前状态的right\_VNs（前面提到过，right\_VNs用于存储某一状态内已经生成过闭包的非终结符。避免对同一个非终结符重复生成闭包。）集合中处理过，那么就需要进行闭包操作。具体做法是从grammarMap中获取该非终结符对应的所有产生式，为每个产生式创建一个新的dfaCell项目（点在最开始位置，索引为 0），然后通过isNewCell函数判断这个项目是否是新的，如果是新的就分配一个新的项目编号（通过ccnt++），添加到dfaCellVector中，并将项目编号添加到当前状态的cellV向量中，这样就把相关项目添加到当前状态中，完成闭包扩展。  ③生成新状态（临时处理）。接下来，遍历当前状态的cellV向量中的项目（再次遍历是为了生成状态转移相关信息），对于点不在产生式末尾的项目，获取其点后面的下一个字符nextSymbol，以这个字符为键，在tempSave映射中获取或创建一个临时的dfaState（用于暂存新状态信息）。然后创建一个新的dfaCell项目，表示点往后移动一位后的情况（通过nextStateCell表示，其gid不变，index加 1），同样通过isNewCell函数判断是否重复，若不重复则分配新编号并添加到dfaCellVector中，然后将这个项目编号添加到临时状态的cellV和originV向量中。同时，根据nextSymbol是终结符还是非终结符，分别将其添加到VT或VN集合中，方便后续用表格生成LR(0)的DFA图。  ④校验并添加新状态。遍历tempSave中的每个临时状态，通过isNewState函数判断这个临时状态是否与已有的状态重复。如果是新状态，就给它分配一个新的状态编号（通过scnt++），创建一个新的dfaState结构体实例，将临时状态的相关信息复制过来，添加到dfaStateVector中；如果是已有的状态，就获取其已有的状态编号。然后创建一个nextStateUnit结构体实例，记录从当前状态通过nextSymbol字符转移到目标状态（新状态或已有状态）的信息，将其添加到当前状态的nextStateVector中，这样就构建好了当前状态到其他状态的转移关系。  ⑤递归处理下一个状态。最后，遍历当前状态的nextStateVector，获取每个下一个状态的相关信息，对每个下一个状态递归调用generateLR0State函数，继续构建整个DFA的状态图结构，直到所有状态及其转移关系都构建完成。  4. 生成LR(1)DFA图    图17 求LR(1)DFA图的辅助函数  在判断LR(1)的项目和状态是否相同时还要考虑项目和状态中的向前搜索符是否相同。只有两条项目的文法规则编号、点的索引以及向前搜索符完全一致时，才能说明此两条项目在LR(1)中是相同的。只有两个状态的同心项以及向前搜索符完全一致时，才能说明此两个状态在LR(1)中是相同的。    图18 求LR(1)DFA图核心代码（1）    图19 求LR(1)DFA图核心代码（2）    图20 求LR(1)DFA图核心代码（3）    图21 求LR(1)DFA图核心代码（4）    图22 求LR(1)DFA图核心代码（5）  可以看到，求LR(1)的DFA图和求LR(0)的DFA图的过程十分相似（相同部分不再详细展开），不同在于求LR(1)DFA的时候需考虑项目的向前搜索符号。当前项目的点所对应的非终结符的follow集恰恰是由此非终结符所得项目的向前搜索符，也就是说我们可以在求闭包的过程中求项目的向前搜索符。此处将详细展开如何求项目的向前搜索符（lookahead）。  **整体思路：**  求lookahead的代码是在构建 LR(1) 项目的闭包以及状态转移过程中，用于确定新生成的 LR(1) 项目的向前搜索符（lookahead）的逻辑。其核心依据是根据当前正在处理的文法规则中，点号（.）所处位置以及其后面符号的类型（终结符、非终结符等情况），结合已计算出的 First 集合和 Follow 集合来确定合适的 lookahead 值。  **具体情况解释：**  ①点后是非终结符且是文法规则右部最后一个元素的情况。当点号（.）后面紧跟着的符号（通过 nextSymbol 获取）是非终结符，并且这个非终结符已经处于文法规则右部的最后位置（即 currentCell.item.index + 1 >= grammarRight.size()）时，新生成项目的 lookahead 应该是当前项目左部非终结符的 Follow 集合。  例如，对于文法规则 A -> .B，如果要生成新的 LR(1) 项目基于 B 的后续项目，此时 B 后面没有其他符号了，那么新生成项目的 lookahead 就取 A 的 Follow 集合中的元素，通过遍历 followSets[grammarDeque[currentCell.item.gid].left].s（也就是 A 的 Follow 集合），将其中元素添加到 lookahead 字符串中。  ②如果点号后面紧跟着的符号（nextSymbol）是终结符，按照 LR(1) 项目闭包及状态转移的逻辑，此时不需要基于这个终结符去生成新的项目或者做其他特殊处理，所以直接跳过后续关于生成新 LR(1) 项目的相关逻辑，继续处理当前状态下其他的 LR(1) 项目。  ③点后是非终结符且不是文法规则右部最后一个元素的情况。首先获取点号后面字符的下一个符号（通过 nextNextSymbol 获取），这是为了进一步判断后续如何确定 lookahead：  （1）如果 nextNextSymbol 是空字符串（@），由于合法的文法规则右部若有多个终结符或非终结符，最后一个字符不可能是空字符串，所以尝试获取再下一个字符（串）（通过 grammarDeque[currentCell.item.gid].right[currentCell.item.index + 2]）来作为真正要判断的后续符号，以便确定 lookahead。  （2）如果 nextNextSymbol 是终结符，那么很直接地就将这个终结符作为新生成项目的 lookahead，即将 lookahead 赋值为 nextNextSymbol，因为终结符本身就是明确的向前搜索符号了。  （3）如果 nextNextSymbol 是非终结符，此时新生成项目的 lookahead 应该是这个非终结符的 First 集合。所以通过遍历 firstSets[nextNextSymbol].s（也就是 nextNextSymbol 这个非终结符的 First 集合），将其中元素添加到 lookahead 字符串中，以此来确定新生成 LR(1) 项目的向前搜索符。  5. 生成LR(1)分析表    图23 求LR(1)分析表的辅助函数  getActionForTerminal函数的作用是根据给定的状态编号 stateId 和终结符 terminal，判断在该状态下遇到这个终结符时应执行的动作（移进、规约或者无动作）并返回相应的动作字符串。它通过遍历当前状态（dfaStateVectorLR1[stateId]）中的所有项目（通过 cellV 向量中的项目编号获取对应的 dfaCellLR1 项目）来进行判断：  ①如果某个项目中 “点” 的位置不在产生式末尾（即 itemCell.item.index < grammarDeque[itemCell.item.gid].right.length()），并且 “点” 后面的符号就是给定的终结符 terminal，那么说明在此状态下遇到该终结符应该执行移进操作，此时调用 getNextStateId 函数获取通过该终结符转移到的下一个状态编号，并返回形如 "s<状态编号>" 的移进动作字符串。  ②如果某个项目中 “点” 的位置在产生式末尾（即 itemCell.item.index == grammarDeque[itemCell.item.gid].right.length()），并且该项目的向前看符号（lookahead）中包含给定的终结符 terminal（通过 find 函数判断），那么说明在此状态下遇到该终结符应该执行规约操作，返回形如 "r<文法编号>" 的规约动作字符串。  ③如果遍历完所有项目都没有符合上述移进或规约的情况，说明在此状态下遇到该终结符没有对应的有效动作，返回空字符串。  getNextStateId函数根据给定的状态编号 stateId 和一个符号（可以是终结符也可以是非终结符），在当前状态的状态转移信息中查找是否存在通过该符号转移到的下一个状态，如果找到则返回对应的状态编号，若没找到（即不存在这样的转移情况）则返回 -1。    图24 求LR(1)分析表核心代码  generateAnalysisTable函数是生成 LR(1) 分析表的核心函数，它基于已经生成的 LR(1) DFA 表信息构建 LR(1) 分析表。主要执行以下几个步骤：  ①初始化分析表大小。根据已经生成的 LR(1) DFA 状态数量来初始化 LR1Table 的大小，确保每个 LR(1) 状态都有对应的分析表单元来存储信息。  ②填充终结符对应的动作信息。外层循环遍历每个 LR(1) 状态（通过状态编号 stateId），获取当前状态对应的分析表单元引用 tableUnit（通过 LR1Table[stateId]）。  内层循环遍历终结符集合 VTLR1，对于每个终结符 terminal，调用 getActionForTerminal 函数获取在当前状态下遇到该终结符时应执行的动作字符串 action，如果动作字符串不为空（说明存在有效的移进或规约等动作），则将该动作字符串存入当前状态分析表单元的 action map 中，以终结符为键，对应的动作字符串为值（通过 tableUnit.action[terminal] = action;），这样就记录好了每个状态下针对不同终结符的动作信息。  ③填充非终结符对应的去向（goto）信息。同样外层循环遍历每个 LR(1) 状态，内层循环遍历非终结符集合 VNLR1，对于每个非终结符 nonTerminal，调用 getNextStateId 函数获取从当前状态通过该非终结符转移到的下一个状态编号 nextStateId，如果 nextStateId 不为 -1（说明存在这样的转移情况），则将对应的状态编号转换为字符串并存入当前状态分析表单元的 goTo map 中，以非终结符为键，状态编号字符串为值（通过 tableUnit.goTo[nonTerminal] = to\_string(nextStateId);），完成非终结符去向信息的记录。  ④处理接受状态。如果当前状态的规约项目文法左部是文法扩展后的开始符号、项目的向前搜索符号是'$'（在本系统中，能出现这种情况的都是状态第一条项目），则此项目需要ACCEPT。因此，首先获取当前状态的首条项目。然后根据项目结构体中的 index 字段，用于判断 “点” 是否在产生式末尾，与对应的文法产生式的右部长度进行比较。同时还要判断此项目的文法左部是否为文法的真开始符号（即trueStartSymbol，因为文法可能会经过增广处理）。最后需访问 LR1Item 结构体里的 lookahead 字段，用于判断向前看符号是否含有 '$'。如果上述条件均满足，则说明此项目需要被接受，将当前状态分析表单元中对应 '$' 这个终结符的动作设置为 "ACCEPT"（通过 tableUnit.action["$"] = "ACCEPT";），  6. 判断文法是否为SLR(1)文法    图25 判断是否为SLR(1)文法核心代码（1）    图26 判断是否为SLR(1)文法核心代码（2）  （1）SLR1Fun1函数的主要目的是检查文法的DFA状态集合中，是否存在移进-规约冲突情况。函数具体逻辑：  ①循环遍历dfaStateLR1Vector中存储的所有DFA状态。如果当前状态不是规约状态（通过state.isEnd标志判断，true表示是规约状态），则跳过当前循环，继续检查下一个状态。只有规约状态才有可能出现移进-规约冲突，所以只关注这类状态。  ②对于是规约状态的情况，遍历当前规约状态的cellV向量，获取项目编号。根据项目编号cellid从dfaCellVector中获取当前的dfaCell。再根据dfaCell中的gid（文法编号）从grammarDeque中获取对应的文法单元，这样就能获取到具体的文法产生式信息。  ③区分规约项目和移进相关情况。判断当前项目是否为规约项目。如果项目中 “点”（.）的位置在产生式右部的末尾，或者产生式右部为空串（gm.right == "@"），则认为这是一个规约项目，将该产生式左部的非终结符插入到集合a中，因为这个非终结符对应的产生式在当前状态需要进行规约操作。  ④如果不是规约项目，说明可能涉及移进操作。进一步判断当前项目 “点” 后面的字符（gm.right[cell.index]）是否为终结符，如果是终结符，则将其插入到集合rVT中，表示在当前状态下遇到这个终结符时可能需要进行移进操作。  ⑤检查移进-规约冲突。两层嵌套循环遍历集合a中的每个非终结符c（代表规约操作相关符号）和集合rVT中的每个终结符v（代表移进操作相关符号）。检查非终结符c的Follow集合中是否包含终结符v。如果包含，意味着在面对输入符号v时，既可以按照非终结符c对应的产生式进行规约，又可以将v移进，这就产生了移进-规约冲突，此时直接返回true，表示检测到冲突。  ⑥如果遍历完所有状态及其项目后，都没有发现移进-规约冲突，那么函数最终返回false。  （2）SLR1Fun2函数的主要目的是检查文法的DFA状态集合中，是否存在规约-规约冲突情况。此函数前面部分与SLR1Fun1函数相似，在此不再赘述，不同的是检查规约-规约冲突情况的逻辑，在此详细展开：  ①两层嵌套循环遍历集合a中的每一对不同的非终结符c1和c2（因为要检查不同规约项目之间是否存在冲突。  ②对于每一对非终结符c1和c2，分别获取它们的Follow集合，然后定义一个新的集合intersection用于存储它们的交集。  通过set\_intersection函数（这是 C++ STL 中用于求两个集合交集的函数，并非本系统函数）来计算followSetC1和followSetC2的交集，并将结果存储到intersection集合中。  ③检查计算得到的交集集合是否为空，如果不为空，说明非终结符c1和c2对应的规约项目存在规约 - 规约冲突，因为它们的Follow集合有共同的符号，在语法分析遇到这些共同符号时，不知道该按照哪个产生式进行规约，此时函数直接返回true，表示检测到冲突。  ④如果遍历完所有状态及其规约项目后，都没有发现规约-规约冲突，那么函数最终返回false，表示该文法不存在这种类型的冲突。  （三）测试  测试结果详见作业文件夹。 |
| 四、实验总结（心得体会）  通过此次实验，我对 LR(0) 和 LR (1) 分析法的原理、状态机的构建过程、项目集闭包的计算、分析表的生成以及如何判断文法是否为 SLR (1) 文法等核心概念有了透彻的理解。这使我明白了 LR 分析法如何通过对文法规则的分析，构建状态转移图和分析表，从而实现对输入句子的准确语法分析，以及不同变体之间的区别和联系。例如，在求 LR (1) 项目的向前搜索符（lookahead）时，我明白了其计算逻辑是根据点号（.）在文法规则中的位置以及其后符号的类型，结合 First 集和 Follow 集来确定的。  通过此次试验我还清楚地认识到LR(0)文法、LR(1)文法以及SLR(1)文法的区别。LR(0)文法分析能力最弱，当它遇到文法规则出现规约-移进冲突时无法分析，此时则需要在分析文法时引入follow集，这就是SLR(1)文法。SLR(1)分析法能够通过follow集告诉DFA何时规约，从而能够在一定程度内避免移进-规约问题，但是SLR(1)分析法不能完全避免移进-规约问题，也无法避免规约-规约问题。此时则需要再构建DFA图中就考虑向前搜索符，这就是LR(1)分析法。  在实验过程中，不可避免地遇到了各种各样的问题，如逻辑错误、运行时错误、数据结构使用不当等。通过仔细分析错误信息、调试代码（使用qDebug()输出中间结果、逐步跟踪程序执行流程等方法），我逐渐学会了如何定位问题的根源，并找到有效的解决方案。例如，在生成 LR (1) DFA 图的过程中，出现了数组越界访问的错误，通过仔细检查代码中对向量索引的操作，发现是在处理点号后面字符的逻辑中，没有正确判断边界条件。经过修正后，成功解决了问题。  总的来说，本次实验是对自底向上分析方法有关知识的一次综合性实践应用，通过实现 LR (1) 分析生成器，我将理论知识与实际编程紧密结合，不仅加深了对 LR 分析法的理解，还在编程技能、问题解决能力等方面得到了提升。 |
| 五、参考文献：  《编译原理复习提纲》  《编译原理及实践》 |