编号：

**《编译原理》实验报告**

**题 目 ： 自下而上语法分析**

**姓 名 ：**

**学 号 ：**

**院 （系）： 计算机科学与技术**

**指导教师：**

**成 绩 ：**

**2024年 12月 5日**

1. **实验目的**

（1）理解 LR（0）语法分析方法的原理，掌握 LR(0)分析表的构造，设计相关数据结构和程序结构，加深对自下而上语法分析方法的理解。

（2）理解 LR 语法分析方法的原理，设计相关数据结构和程序结构，加深对自下而上语法分析方法的理解。

1. **问题描述**

我实现了一个实现一个 LR(0)语法分析器。 LR(0)分析器是一种自下而上的语法分析方法，主要用于根据给定的上下文无关文法（CFG）进行词法分析和语法分析。LR(0)分析器属于一种自动机的实现方法，通过状态转换和分析表来进行句子的语法分析。

**本实验需要实现的功能有：**

1. 构造文法的 LR预测分析表；
2. 构造DFA：使用LR(0)方法构造识别活前缀的DFA。
3. 构造分析表：生成Action表和Goto表。
4. 输入文法：从指定文件读取文法描述（非终结符、终结符、产生式和起始符号）。
5. 输出项目集簇：输出LR(0)分析器的项目集簇。
6. 输出DFA：输出DFA的状态转换。
7. 输出分析表：输出Action表和Goto表到文件（扩展名为.lrtbl）。
8. LR(0)文法判断：判断文法是否为LR(0)文法，并输出结果。
9. 分析符号串：使用LR(0)分析器分析输入符号串，输出分析过程和结果。
10. 总控程序：构建LR(0)语法分析器的总控程序，协调上述功能的执行。

**实现原理和方法：**

**1. 文法构建与输入加载**

在LR分析器中，首先需要构建符合上下文无关文法的文法。文法由终结符、非终结符、产生式和起始符号组成。产生式的形式为A->α，其中A是非终结符，α是符号串，可以是终结符、非终结符或空串（ε）。文法信息通过文件输入加载到分析器中，供后续分析使用。

**2. 项目与项目集族的构建**

LR分析器使用项目集族来模拟分析过程。一个项目是产生式的扩展形式，包含一个“点”，表示当前正在匹配产生式右部的具体位置。闭包操作在此过程中起到关键作用。当项目中的非终结符后面有符号，并且该非终结符有产生式时，闭包操作会扩展该非终结符的产生式。通过这种方式，分析器可以生成所有相关的项目。DFA（确定性有限自动机）用来构造项目集，每个状态代表一个项目集，状态之间的转移表示如何根据输入符号（终结符或非终结符）从一个状态转移到另一个状态。初始项目集由起始符号生成，并通过闭包操作得到。

**3. 状态转移与状态转移表**

LR分析器通过计算状态之间的转移来构建状态转移表（transitions）。如果项目集内存在A->X·YZ，并且Y是某个符号（终结符或非终结符），则可以从A->X·YZ状态转移到A->XY·Z状态。这些转移通过查找文法的产生式来确定，并生成相应的转移关系。状态转移表通过广度优先搜索（BFS）算法逐步生成完整的项目集族，确保分析过程能够正确处理所有可能的符号串。

**4. 分析表的构建：Action表与Goto表**

LR分析器的核心是Action表和Goto表。Action表包含三种主要操作：移进（shift）、规约（reduce）和接受（accept）。移进操作发生在栈顶符号与输入符号匹配时，规约操作发生在栈顶符号与产生式的右部匹配时，接受操作则在输入符号串完全匹配并符合起始符号时发生。Goto表用于处理非终结符的转移。Action表的填充是通过当前状态和输入符号的匹配来实现的。例如，当栈顶符号是非终结符且符合某个产生式时，查表并执行规约。状态转移时，如果栈顶符号与输入符号匹配且当前状态有合法转移，则执行移进操作；否则，进行规约操作。

**5. LR分析器的执行过程**

在LR分析器执行过程中，首先读取文法和分析表，初始化分析器。分析器模拟自底向上的分析过程，使用栈来模拟语法分析。输入符号串从队列中逐个取出，进行分析。当栈顶符号与输入符号相同，分析器会执行移进操作并继续分析；如果栈顶符号是非终结符，则查表使用相应的产生式，逆序将产生式的右部符号压入栈中；如果栈顶符号与输入符号不匹配，或者在分析表中找不到合适的产生式，则表示分析失败。如果所有输入符号被成功消耗且栈中只剩下起始符号，则表示分析成功。整个分析过程的输出包括栈信息、输入符号和当前的操作（移进、规约或接受）。

1. **软件设计方法的选择**

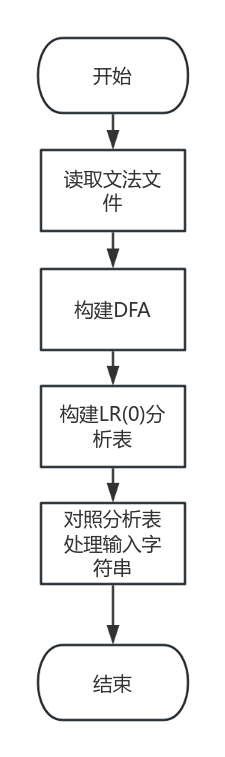
**表1 软件设计方法及开发语言和环境**

|  |  |
| --- | --- |
| **软件设计方法** | **结构化设计方法** |
| 开发语言 | C++ |
| 开发环境 | windows11，Visual Studio 2022 |

**表2 创建的模型**

|  |  |
| --- | --- |
| **分析模型** | **设计模型** |
| 系统流程图、数据字典 | 类图、函数接口 |

1. **分析模型**

**系统流程图：**

在LR分析器的系统流程图中，整个过程从文法加载开始，到最终的语法分析结束，每一步都通过不同的模块和数据流动来实现分析。以下是对主要流程的分段描述。

系统的第一步是加载文法信息。此步骤通过读取输入文件或者直接由用户提供的文法定义，解析文法中的终结符、非终结符、产生式以及起始符号。文法信息在数据结构中进行存储，以便后续处理。在文法加载完毕后，我们生成初始的项目集，并进行闭包操作，以得到初始状态的集合。初始项目集通常由起始符号和相关产生式构成，这些信息为后续的状态转移和分析表构建提供了基础。

状态转移是LR分析器的核心部分，项目集计算是构建整个分析流程的基础。在这个阶段，LR分析器通过闭包操作生成初始项目集，并利用广度优先搜索（BFS）来遍历项目集之间的状态转移。每个项目集代表了当前分析过程中不同产生式的状态。当遇到输入符号时，LR分析器会根据当前状态和符号进行状态转移，生成新的项目集。状态之间的转移关系是通过分析输入符号与产生式右侧的符号匹配的结果来确定的。通过状态转移，最终形成一个完整的项目集族。

分析表的构建是LR分析器的另一个关键步骤。根据生成的项目集族，分析表分为Action表和Goto表两部分。Action表记录了不同状态下，遇到不同输入符号时应采取的动作，主要包括移进、规约和接受操作。Goto表则记录了在遇到非终结符时如何进行状态转移。分析表的填充过程通过查找项目集中的每个 **图1自下而上语法分析器**

项目，结合输入符号来确定具体的操作。如果当前状态和输入符号匹配，系统会选择移进操作；如果某个非终结符的右侧与栈顶符号匹配，则执行规约操作；否则，系统会检查是否可以接受输入符号，确认分析是否完成。

一旦分析表构建完成，LR分析器便开始进行实际的语法分析。分析过程模拟自底向上的语法分析，使用栈来模拟分析过程。在这个阶段，输入符号串中的符号逐个与栈顶符号进行匹配。若栈顶符号与当前输入符号相同，则出栈并继续分析；如果栈顶符号为非终结符，则通过查表来选择相应的产生式，并将产生式的右部逆序压入栈中进行后续的推导。如果栈顶符号与当前输入符号不匹配，或没有找到有效的产生式，则表示语法分析失败。最终，当所有输入符号被成功消耗，且栈中只剩下起始符号时，分析过程成功结束。

通过上述四个主要步骤，LR分析器完成了从文法加载到语法分析的全过程。整个过程不仅涉及了文法的解析、状态转移的计算，还包括了分析表的构建和实际的语法分析，确保了对输入符号串的正确解析。

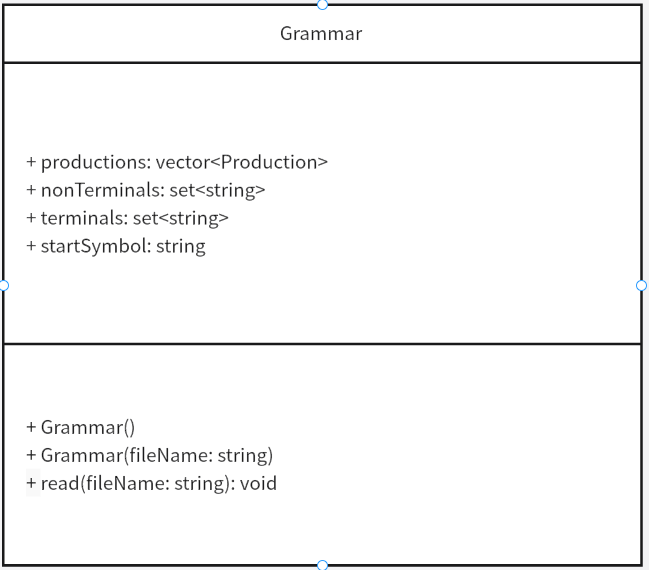
**数据字典：**

**表3 数据字典**

|  |  |  |
| --- | --- | --- |
| **数据名称** | 数据类型 | 数据作用 |
| **nonTerminals** | set<string> | 文法中的所有非终结符 |
| **terminals** | set<string> | 文法中的所有终结符 |
| **Production** | struct{int,int} | 产生式结构 |
| **productions** | vector<Production> | 文法的产生式集合 |
| **startSymbol** | string | 文法的起始符号 |
| **Item** | struct{int,int} | 项目结构 |
| **itemSets** | vector<set<Item>> | 项目集族 |
| **transitions** | map<pair<int, string>, int> | 状态转移表 |
| **actionTable** | map<pair<int, string>, string> | Action 表 |
| **gotoTable** | map<pair<int, string>, int> | Goto 表 |

1. **设计模型**

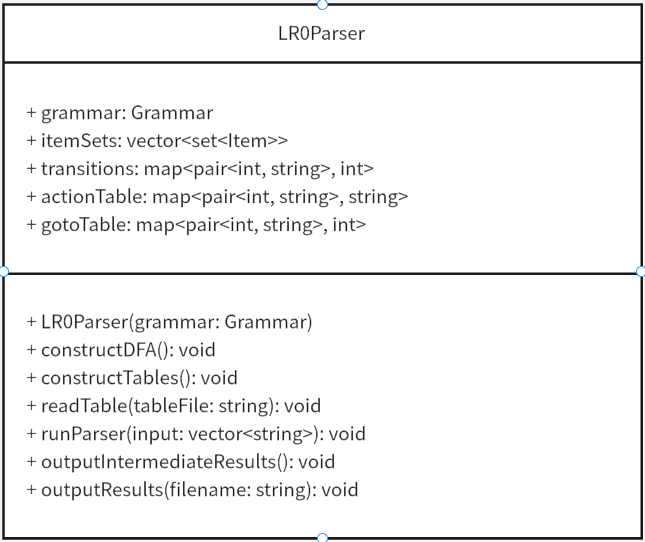
**类图：**

**Grammar类：**

Grammar 类表示文法的核心结构，负责存储与文法相关的各个组件。在这个类中，包含了一个 vector<Production> 类型的成员变量 productions，它存储了文法的所有产生式。每个产生式描述了一个非终结符和一个符号串之间的推导关系。nonTerminals 和 terminals 分别是 set<string> 类型的成员变量，分别表示文法中的非终结符集合和终结符集合。startSymbol 是文法的起始符号，通常是文法的第一个非终结符，它决定了语法分析的起始点。这个类还提供了一个构造函数 Grammar()，用于创建一个空文法对象，  **图2 Gramma类**

以及一个带参数的构造函数 Grammar(fileName)，通过读取文件加载文法定义。read(fileName) 方法则是该类的核心功能之一，用于从文件中读取文法信息，并解析出文法的产生式、非终结符、终结符和起始符号。

**LR0Parser 类**

LR0Parser 类则是语法分析的核心，负责处理具体的语法分析任务。这个类的成员变量 grammar 是一个 Grammar 类型的对象，它提供了 LR(0) 分析所需要的文法信息。itemSets 是一个 vector<set<Item>> 类型的成员，表示项目集族，它将被用于构造确定性有限自动机（DFA）。DFA 是语法分析器的核心部分，用于控制分析过程的状态转移。transitions、actionTable 和 gotoTable 分别是表示状态转移表、ACTION 表和 GOTO 表的成员变量，分别用于处理语法分析过程中的状态转移、移进、规约等操作。constructDFA() 方法用于构造 DFA 状态集，通过闭包和转移操作生成所有可能的项目集。constructTables() 方法基于 DFA 和文法规则生成 ACTION 和 GOTO 表，这些表将被用于实际的语法分析。readTable(tableFile) 方法则用于读取外部的分析表文件，方便后续的分析操作。runParser(input) 方法负责执行语法分析过程，处理输入的符号串并进行语法判断，确保输入符合文法规则。为了帮助调试，outputInt **图3 LR0Parser类**

ermediateResults() 方法可以输出分析过程中的中间状态，outputResults(filename) 方法则用于输出最终的分析结果和生成的表格。

**主要函数接口：**

**表4 函数接口**

|  |  |  |  |
| --- | --- | --- | --- |
| **函数名** | 传入参数类型 | 返回参数类型 | 描述 |
| **Grammar::read** | const string&filename | void | 读取文法文件 |
| **LR0Parser::readTable** | const string& tableFile | void | 读取分析表文件 |
| **LR0Parser::runParser** | const vector<string>& input | void | 执行语法分析 |
| **read\_input** | string filename | vector<string> | 读取输入字符串 |
| **LR0Parser::findItemSet** | const set<Item>& items | int | 查找项目集编号 |
| **LR0Parser::closure** | const set<Item>& items | set<Item> | 计算闭包 |
| **LR0Parser::constructDFA** | 无 | void | 构造DFA |
| **LR0Parser::constructTables** | 无 | void | 构造分析表 |
| **LR0Parser::outputIntermediateResults** | 无 | void | 输出中间结果 |
| **LR0Parser::outputResults** | const string& filename | void | 输出分析表到文件 |

1. **主要算法描述**

**（1）构造DFA（如下图4）**

这个constructDFA函数的目标是构建一个LR(0)自动机的DFA（确定性有限自动机），以便为后续的LR语法分析提供状态转移和项目集。首先，函数通过closure方法从初始项集（包含起始文法符号的项目）开始构建状态空间。初始项集通过将项目{0, 0}（表示文法的第一个产生式的开始位置）传递给closure函数生成。

接着，函数初始化了一个队列q，用于存储需要处理的项目集的索引，并将初始项目集的索引0推入队列中。随后，函数进入一个循环，逐个处理队列中的项目集。对于每个当前项目集，函数会通过遍历其中的每个项目，计算它的可能转移。具体来说，如果当前项目中的点（dot）位置不在产生式的最后，它将查找下一个符号并创建一个新的项目，该项目的点位置向前移动一位。

然后，函数根据这些转移生成新的项目集，并使用closure函数计算该项目集的闭包。closure函数确保将所有通过文法规则可以推出的项目都包含在内，以保持状态的正确性。如果新生成的项目集还没有出现过（即它的索引在itemSets中找不到），则将其添加到itemSets中，并将该项目集的索引推入队列，以便进一步处理。

最后，transitions映射记录了从当前项目集到新项目集的状态转移。对于每一个符号（如文法的终结符或非终结符），函数会记录当前项目集在该符号下的转移到新状态的映射。

这个过程通过不断生成和合并项目集，逐步构建整个DFA，直到队列为空，表示所有项目集和转移都已计算完毕，DFA也就完全构建完成了。

**（2）构造分析表（如下图5）**

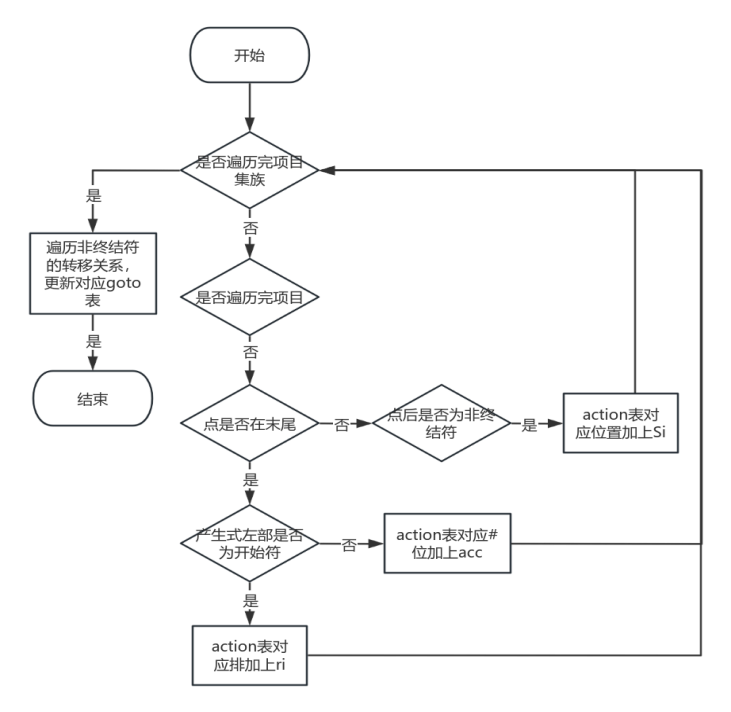
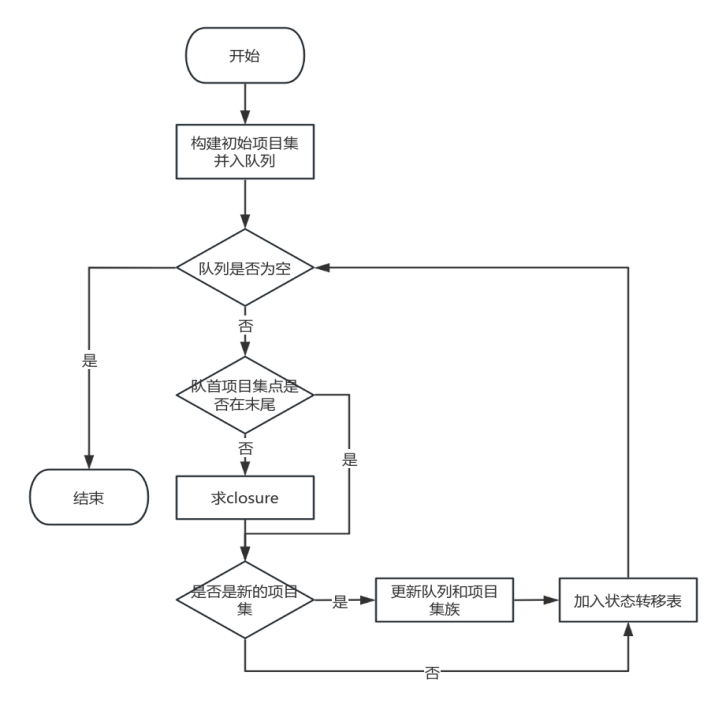
constructTables函数通过遍历所有状态和项目集来构建LR(0)分析器的动作表（actionTable）和转移表（gotoTable）。首先，函数检查每个项目的点位置，如果点已经到达产生式的末尾，说明可以进行规约操作或接受。如果当前项目的左侧符号是文法的开始符号，那么就在动作表中为输入符号#设置接受操作（acc）。如果不是开始符号，那么对于每一个终结符和结束符号#，都设置规约动作（r）。如果点未到末尾，说明可以进行移进操作，程序根据当前状态和输入符号，查找对应的转移状态，并为动作表设置移进操作（s）。接下来，函数为每个非终结符填充转移表（gotoTable），记录每个状态和非终结符的状态转移。这个过程保证了分析器可以根据文法的项目集和转移关系，准确地进行移进、规约等动作。

**（3）语法分析器（如下图6）**

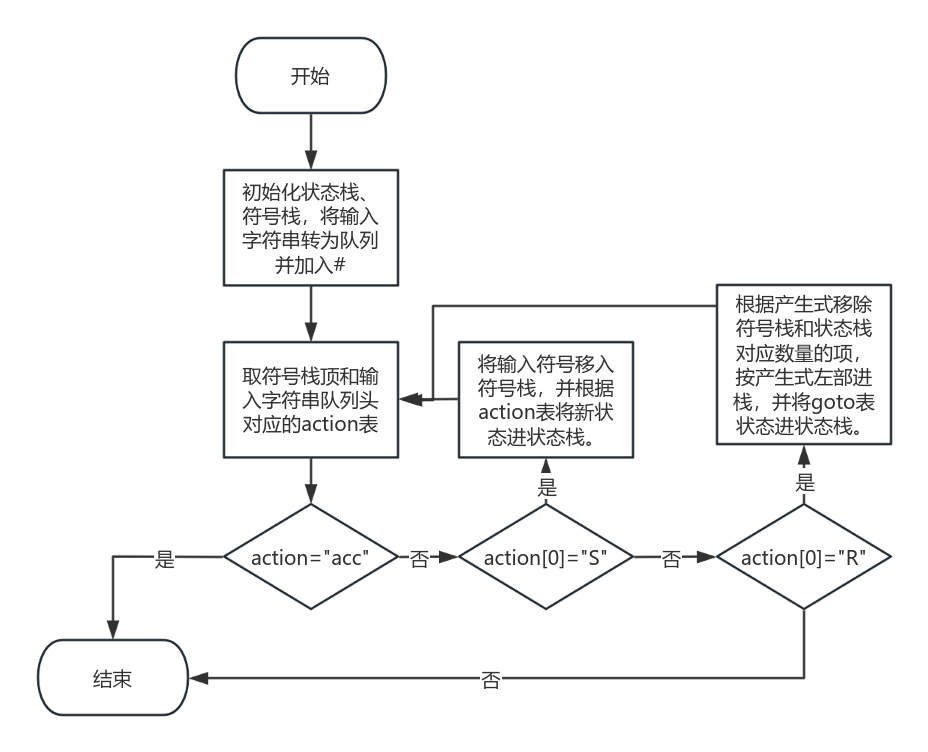
这个函数实现了一个基于LR(0)分析表的语法分析器，用于解析输入串并输出分析过程。首先，函数初始化状态栈和符号栈，状态栈从0开始，符号栈以#作为栈底符号。同时，将输入串加载到队列中，并在队列末尾添加结束符#。在分析过程中，程序通过状态栈的栈顶状态和当前输入符号从动作表中查找对应的动作。如果动作是“acc”，表示分析成功，程序终止并输出“成功接收”。如果动作是移进操作（以s开头），程序将目标状态推入状态栈，将当前输入符号推入符号栈，并移除输入队列的栈顶符号，继续下一轮分析。

如果动作是规约操作（以r开头），程序会根据产生式的右部长度，从状态栈和符号栈中依次弹出相应数量的项。然后，通过转移表（gotoTable）查找规约后需要跳转的目标状态，将该状态推入状态栈，并将产生式的左部符号推入符号栈，完成规约操作。程序会输出当前规约的情况，并显示被规约的产生式。

若遇到动作表返回未知动作或查表失败，程序会输出错误信息并终止分析，通常这意味着输入串无法被当前文法解析。整个过程通过移进和规约交替进行，直到输入串被接受或遇到错误，模拟了LR分析器的工作机制，成功解析输入串或检测到语法错误。



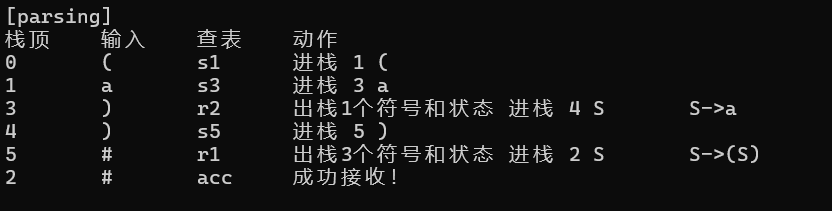
**图4 构造DFA 图5构造分析表**



**图6 语法分析器**

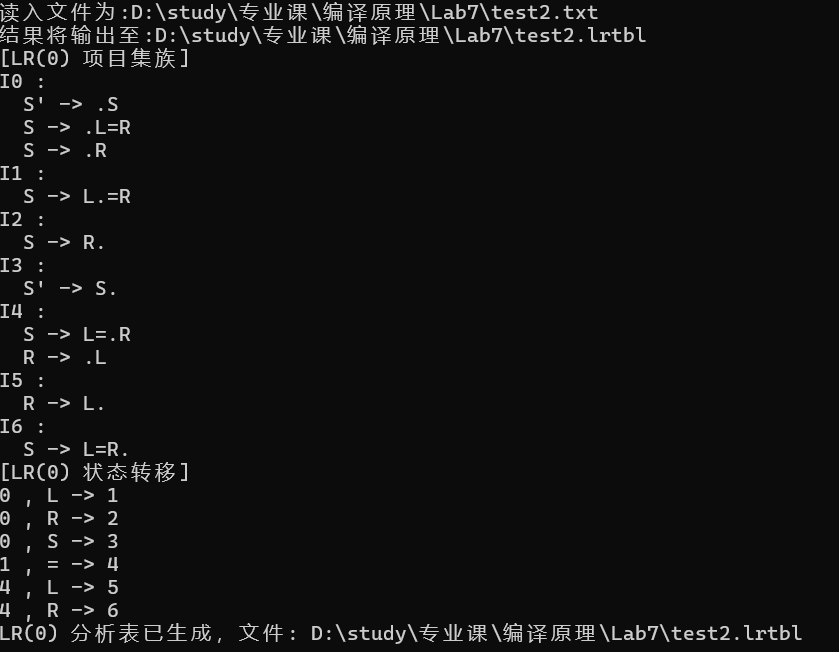
1. **测试数据与测试效果**

**（1）LR 语法分析器的设计与实现：**



**图7 LR 语法分析器的设计与实现**

**（2）LR(0)分析表的构造：**



**图8 LR(0)分析表的构造**

1. **实验总结**

在本次实验中，我实现了一个基于LR(0)分析表的语法分析器，这个过程让我对编译原理中的语法分析部分有了更深的理解。通过实验，我深入学习了LR(0)分析的基本原理，掌握了如何通过状态集和转移计算来构建DFA，并利用此DFA生成分析表。通过这些知识，我不仅加深了对编译原理的理解，还提升了我的编程能力，特别是在处理复杂数据结构和算法实现方面。

此外，这个实验也让我体会到了调试和测试的重要性。在实现过程中，很多问题往往是由于细节问题导致的，因此在每个环节进行充分的调试，确保每个步骤的正确性，能有效减少后续的错误。

虽然本次实验的LR(0)分析器已经能够处理一些简单的输入串，但LR(0)文法的表达能力较为有限，很多常见的编程语言文法无法满足LR(0)的要求。因此，未来我计划深入学习LR(1)和更高级别的分析方法，例如SLR和LALR分析，以应对更复杂的语言解析任务。

总结来说，这次实验让我在编译原理方面获得了宝贵的经验，同时也让我对编译器的设计与实现有了更清晰的认识，今后我将继续在此基础上扩展我的学习。