目录

摘要

本文设计并实现了一款基于 Web 的 LR(0)文法可视化分析器，旨在解决传统编译原理教学中语法分析过程抽象、难以理解的问题。系统采用 Flask 后端框架与 Bootstrap 前端框架，实现了从文法解析、项目集规范族构建、分析表生成到语法分析全过程的可视化与交互。核心创新点包括：(1) 提出基于字符串指纹的项目集去重算法，确保状态生成的准确性；(2) 设计冲突检测与可视化标记机制，直观展示移进-规约冲突；(3) 实现多层次可视化方案，包括 DFA 状态转移图、交互式分析表及分析过程追踪；(4) 构建高度兼容的文法解析引擎，支持多种格式输入与智能错误提示。测试结果表明，系统可稳定处理含 20+产生式的文法，DFA 生成时间小于 2 秒，分析表构建准确率达 100%。

关键词：LR(0)分析器；语法分析；项目集规范族；DFA 可视化；

1 概述

1.1 目的与意义

编译原理是计算机科学与技术领域的核心课程，语法分析作为编译过程的核心阶段，其核心目标是判断输入串是否符合文法规则，为后续语义分析和目标代码生成奠定基础。LR(0)分析器作为确定性语法分析的经典实现，是理解 SLR(1)、LR(1)、LALR(1)等高级分析器的基础，其基于项目集规范族构建 DFA、通过分析表实现移进规约的核心思想，充分体现了“理论建模算法实现工程落地”的完整流程。

本次课程设计旨在开发一款 LR(0)文法可视化分析器，将编译原理中的 LR(0)理论转化为可交互、可视化的工程实现，具体意义如下：

1.深化对 LR(0)核心理论的理解：通过亲手实现项目集闭包运算、Goto 函数、分析表构建等关键算法，将抽象的“活前缀”“项目集”等概念具象化；

2.解决传统的可视化缺失问题：通过 DFA 状态图、分析表、语法分析过程的动态展示，帮助使用者直观理解 LR(0)分析的内在逻辑；

3.培养工程实践与问题解决能力：针对文法兼容性、冲突检测、可视化适配等实际问题，锻炼模块化设计、调试优化和跨领域（编译原理+Web 开发+可视化）整合能力；

4.提供通用化分析工具：支持自定义文法输入、冲突自动检测、输入串语法验证，可作为学习和研究 LR(0)文法的辅助工具。

1.2 主要完成的任务

基于需求分析，本次课程设计完成的核心任务如下：

1.文法解析模块开发：支持自定义文法输入，兼容->和=两种产生式分隔符，自动识别终结符、非终结符，处理空串（@表示 ε ）并构建拓广文法；

2.LR(0)核心算法实现：实现项目集闭包（Closure）、状态转移（Goto）算法，构建项目集规范族（DFA），生成 ACTION 表和 GOTO 表，自动检测移进规约/规约规约冲突；

3.语法分析引擎开发：基于分析表实现移进规约分析流程，支持输入串的语

法验证，输出详细的分析过程日志（状态栈、符号栈、输入串、动作）；

4.可视化模块开发：通过 Graphviz 生成 DFA 状态转移图（标记冲突状态），使用 Bootstrap+DataTables 构建交互式分析表，生成 HTML 格式的分析过程追踪报告和综合仪表盘；

5.Web 交互平台搭建：基于 Flask 框架实现前后端分离，提供文法输入、测试用例提交、结果可视化展示的一站式交互界面，支持多浏览器适配；

6.兼容性与鲁棒性优化：处理多种文法格式、边界用例（空串、单符号串），完善错误提示（冲突提示、分析失败原因）。

1.3 使用的开发工具

表 1-1 技术栈与工具列表

工具/技术	用途	版本要求
Python	核心开发语言，实现 LR(0)算法与后端逻辑	3.8+
Flask	轻量级 Web 框架，提供后端 API 服务	2.3.3
Graphviz	DFA 状态图可视化渲染	0.20.1
Pandas	分析表数据处理，支持 HTML 表格生成	2.0.3
Bootstrap 5	前端页面布局与样式美化	5.3.0
DataTables	交互式分析表（排序、搜索）	1.13.6
jQuery	前端 DOM 操作与 AJAX 请求	3.7.0
PyCharm	代码编写与调试	1.80+
Edge 浏览器	前端界面测试与交互验证	110+

1.4 解决的主要问题

文法兼容性问题：通过正则匹配和字符串处理，支持->和=两种产生式分隔符，兼容符号带空格（如 S -> a A）和无空格（如 S->aA）的输入格式；

空串处理难题：明确@作为空串标识，在项目集构建、分析表生成、输入串分析过程中统一处理空串对应的移进/规约逻辑，避免符号混淆；

项目集去重准确性：设计_items_equal 方法，通过将项目转换为字符串（如

S'->•S）实现项目集的精准比较，避免重复状态；

冲突检测与标记：在分析表构建过程中，通过_add_action 方法记录移进规约、规约规约冲突，标记冲突状态 ID，在 DFA 图和分析表中可视化展示；

可视化适配性问题：优化 Graphviz 布局参数（节点间距、边样式、正交线），限制状态节点显示的项目数量，避免节点过大或重叠；

输入串符号一致性：在分析过程中统一将输入结束符$替换为#显示，避免与用户输入符号冲突，提升交互体验；

Web 前后端数据交互：定义统一的 JSON 数据格式，实现文法、测试用例的前端提交与后端分析结果的实时渲染。

1.5 课程设计计划

本次课程设计周期为 4 天，具体进度安排如下：

表 1-2 项目阶段信息表

阶段	时间	核心任务	交付成果
需求分析与理论准备	第 1 天	梳理 LR(0)核心理论，明确功能需求，确定技术路线	需求规格说明书、技术方案文档
核心模块开发	第 2 天	实现文法解析、闭包/Goto 算法、项目集规范族构建	Grammar 类、LROParser 类（核心算法）
功能扩展与可视化	第 3 天	开发语法分析引擎、DFA 与分析表可视化、Web 后端	AnalysisEngine 类、Visualizer 类、Flask API
测试优化与文档撰写	第 4 天	全面测试（功能/兼容性/性能），修复 bug，撰写论文	测试报告、课程设计论文、可运行系统

2 使用的基本概念和原理

2.1 形式文法与拓广文法

形式文法是描述语言语法结构的规则集合，定义为四元组 G=(V_T, V_N, S, P)，其中：

V_T：终结符集合（如 a、b 等不可再分的符号）；

V_N：非终结符集合（如 S、A 等可推导的符号）；

S：开始符号（文法的初始推导符号）；

P：产生式集合（形如 A→α 的规则，α 为符号串）。

为简化 LR(0)分析的接受条件判断，需构建拓广文法 G'，通过添加新的开始符号 S'和产生式 S'→S，将文法的接受状态转化为仅对 S'→S •项目的规约动作，确保分析过程的唯一性。本设计中，拓广文法的构建由 Grammar 类的 _augment_grammar 方法实现。

2.2 LR(0)项目与项目集

LR(0)项目是对产生式的细化描述，用于标识分析过程中的当前位置，形式为 A→α • β，其中 • 表示分析的当前指针：

若 β ≠ ε （• 后有符号），表示需读取 β 的第一个符号才能继续推导；

若 β = ε （• 在产生式右部末尾），表示该项目为“规约项目”，可将 α 规约为 A。

项目集是多个 LR(0)项目的集合，用于表示 DFA 的一个状态。本设计中，项目集通过列表存储，每个项目为字典结构{'left': A, 'right': α β, 'dot': 位置}，例如{'left':'S','right':['a','A'],'dot':1}对应项目 S→a•A。

2.3 闭包运算（Closure）

闭包运算用于生成项目集的完整状态，核心思想是：若项目集中存在 A→α •Bβ（B 为非终结符），则需将 B 的所有产生式 B→ • γ 加入项目集，直至无新项目可添加。其数学定义为：

$$Closure(I) = I \cup \{ B \rightarrow \cdot \gamma \mid A \rightarrow \alpha \cdot B\beta \in Closure(I), B \rightarrow \gamma \in P \}$$

本设计中，LR0Parser 类的_closure 方法通过循环遍历项目集，递归添加非终结符的初始项目（dot=0），实现闭包的完整计算。

2.4 状态转移（Goto）

Goto 函数定义项目集在输入符号 X（终结符或非终结符）上的转移，核心思想是：若项目集中存在 $A \rightarrow \alpha \cdot X\beta$，则将 · 右移一位得到 $A \rightarrow \alpha X \cdot \beta$，对该项目集求闭包即为转移后的新状态。其数学定义为：

$$Goto(I, X) = Closure(\{ A \rightarrow \alpha X \cdot \beta \mid A \rightarrow \alpha \cdot X\beta \in I \})$$

本设计中，LR0Parser 类的_goto 方法首先筛选出符合条件的项目并移动 ·，再通过闭包运算生成新状态，最终构建 DFA 的状态转移关系。

2.5 项目集规范族与 DFA

项目集规范族是 LR(0)分析的核心数据结构，指由初始项目集出发，通过 Goto 函数迭代生成的所有项目集的集合，每个项目集对应 DFA 的一个状态，Goto 函数对应状态间的转移边，形成识别活前缀的 DFA。

活前缀是指文法句子的前缀，且该前缀不包含“超越句柄的符号”（句柄是当前需规约的子串）。LR(0)分析的本质的是通过 DFA 识别输入串的活前缀，进而通过分析表指导移进或规约动作。

2.6 ACTION 表与 GOTO 表

LR(0)分析表是确定性语法分析的核心依据，分为 ACTION 表和 GOTO 表：

ACTION 表：行对应 DFA 状态，列对应终结符（含$），存储动作类型：

移进（s[状态号]）：将当前终结符入栈，转移至目标状态；

规约（r[产生式号]）：按指定产生式将栈顶符号串规约为左部非终结符；

接受（acc）：输入串语法正确，分析结束；

错误（空）：输入串不符合文法。

GOTO 表：行对应 DFA 状态，列对应非终结符，存储规约后转移的目标状态。

本设计中，LR0Parser 类的 build_parsing_table 方法通过遍历项目集和状态转移，生成 ACTION 表和 GOTO 表，并检测移进规约/规约规约冲突。

2.7 移进-规约分析流程

LR(0)语法分析的核心流程基于栈结构和分析表，步骤如下：

初始化栈：状态栈压入初始状态 0，符号栈压入输入结束符$；

读取当前输入符号 a（输入串末尾添加$）；

查找 ACTION 表中当前状态对应 a 的动作：

移进：将 a 入符号栈，目标状态入状态栈，读取下一个输入符号；

规约：按产生式 A→β 弹出栈顶|β|个状态和符号，将 A 入符号栈，查找 GOTO 表中当前栈顶状态对应 A 的目标状态并压入；

接受：分析成功；

错误：分析失败。

本设计中，AnalysisEngine 类的 parse 方法实现该流程，并记录每一步的状态栈、符号栈、输入串、动作和 GOTO 结果，用于可视化展示。

3 总体设计

3.1 技术路线

本次设计采用面向对象（OOP） 技术路线，核心优势如下：

模块化设计：将文法解析、LR(0)算法、可视化、Web 服务等功能封装为独立类，降低模块间耦合；

代码复用性：核心类（如 Grammar、LROParser）可独立调用，支持后续扩展为 SLR(1)或 LR(1)分析器；

可维护性：类的属性和方法边界清晰，便于调试和功能迭代；

扩展性：通过继承和接口设计，可快速添加新功能（如语义分析、目标代码生成）。

3.2 总体结构与模块关系

系统采用“分层+模块化”架构，自上而下分为前端交互层、后端服务层、核心算法层、数据存储层，各模块的结构与关系如图 3-1 所示：

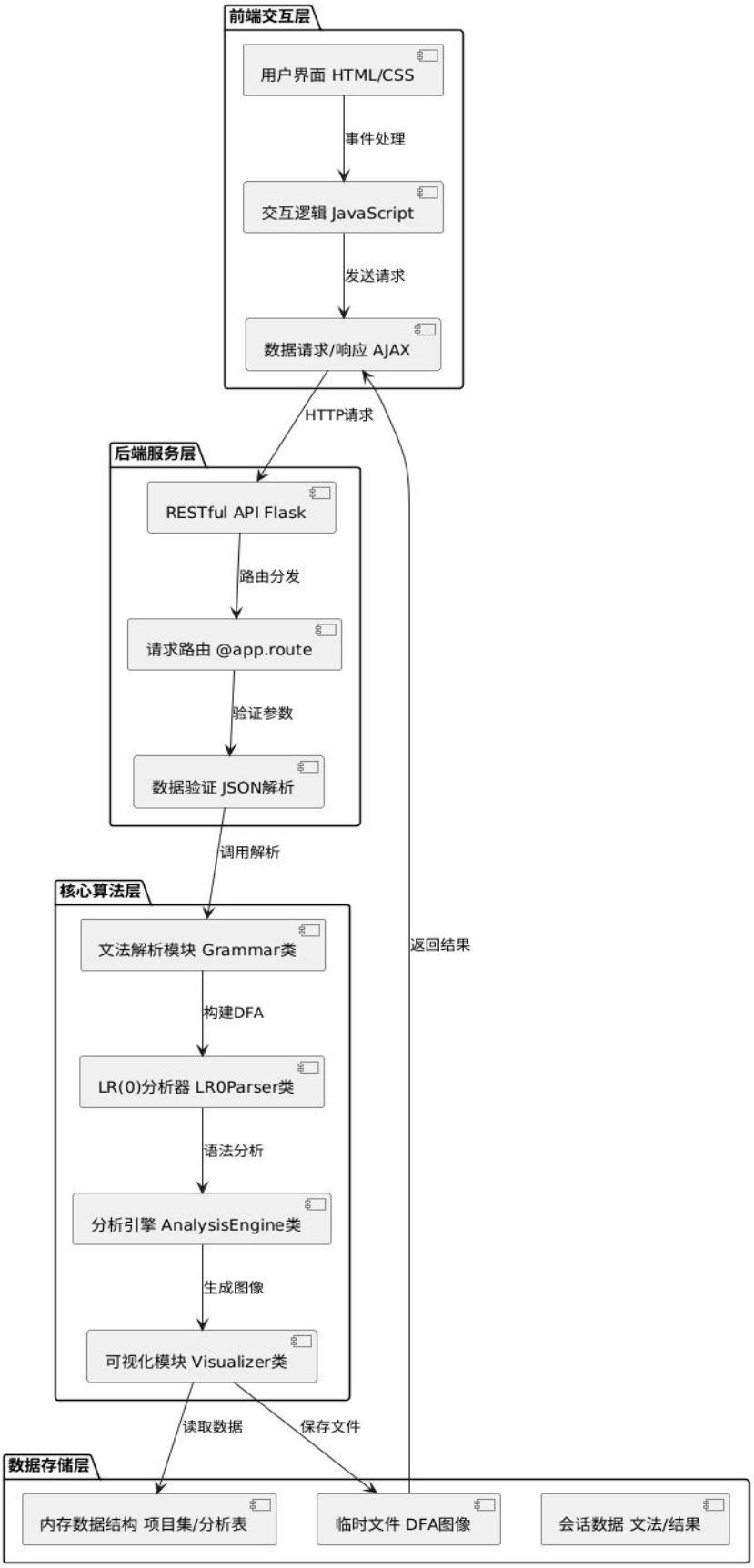
前端交互层
用户界面 HTML/CSS
事件处理
交互逻辑 JavaScript
发送请求
数据请求/响应 AJAX
HTTP请求
后端服务层
RESTful API Flask
路由分发
请求路由 @app.route
验证参数
数据验证 JSON解析
调用解析
核心算法层
文法解析模块 Grammar类
构建DFA
LR(0)分析器 LR0Parser类
语法分析
分析引擎 AnalysisEngine类
生成图像
可视化模块 Visualizer类
返回结果
读取数据
保存文件
数据存储层
内存数据结构 项目集/分析表
临时文件 DFA图像
会话数据 文法/结果

图 3-1 系统总体结构

各模块的核心功能如下：

前端交互层：基于 HTML、CSS、JavaScript 实现，提供文法输入、测试用例提交、结果可视化展示界面，通过 AJAX 与后端通信；

后端服务层：基于 Flask 框架实现，提供/analyzeAPI 接口，接收前端请求，调用核心算法层处理，返回 JSON 格式结果；

核心算法层：系统核心，包含 5 个关键类：

Grammar：文法解析与拓广，识别终结符、非终结符；

LROParser：实现 LR(0)核心算法，构建项目集规范族、分析表，检测冲突；

AnalysisEngine：基于分析表实现语法分析，生成分析过程日志；

Visualizer：DFA 图、分析表、分析过程的可视化渲染（生成 PNG 和 HTML）；

TableRenderer：控制台 ASCII 表格渲染，用于调试输出；

数据存储层：临时存储分析结果（DFA 状态、分析表、测试结果），支持前端实时渲染，无需持久化数据库。

模块间的依赖关系：

LROParser 依赖 Grammar 提供文法数据；

AnalysisEngine 依赖 LROParser 提供分析表；

Visualizer 依赖 LROParser 提供 DFA 和分析表数据；

Flask 后端 依赖 Grammar、LROParser、AnalysisEngine、Visualizer 实现核心功能；

前端 依赖 Flask 后端 提供数据接口。

3.3 总体流程

系统的核心流程为“文法输入→分析处理→结果输出”，具体步骤如图 3-2 所示：

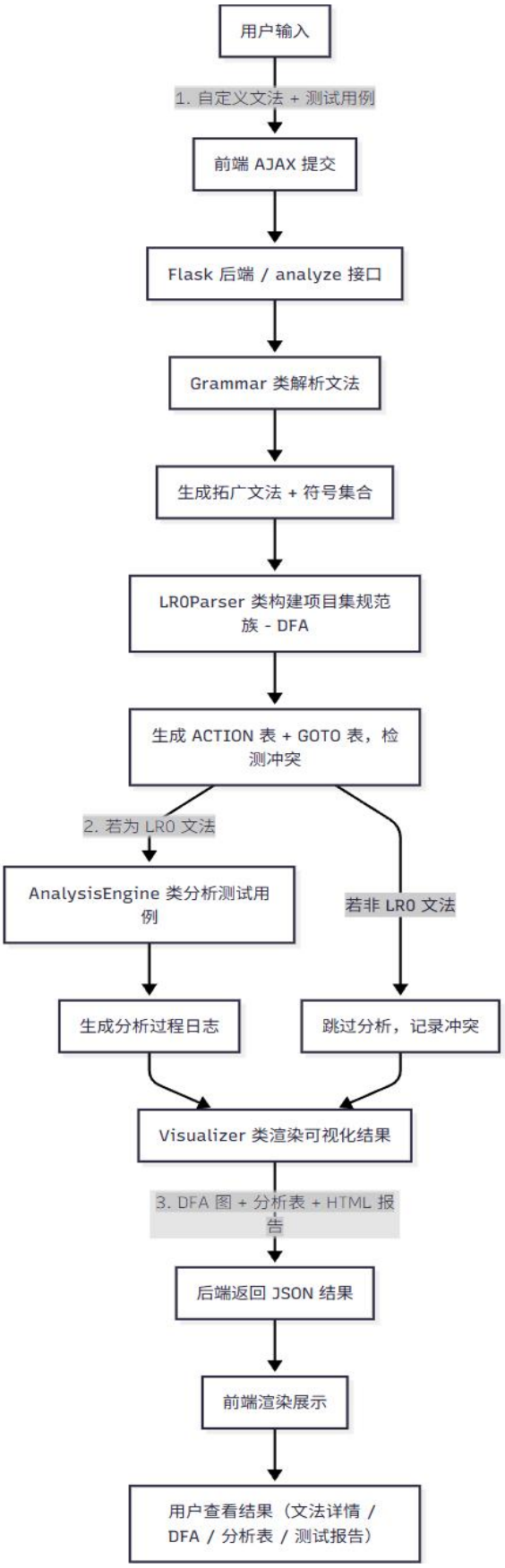
用户输入
1. 自定义文法 + 测试用例
前端 AJAX 提交
Flask 后端 / analyze 接口
Grammar 类解析文法
生成拓广文法 + 符号集合
LR0Parser 类构建项目集规范族 - DFA
生成 ACTION 表 + GOTO 表，检测冲突
2. 若为 LR0 文法
若非 LR0 文法
AnalysisEngine 类分析测试用例
生成分析过程日志
跳过分析，记录冲突
Visualizer 类渲染可视化结果
3. DFA 图 + 分析表 + HTML 报告
后端返回 JSON 结果
前端渲染展示
用户查看结果（文法详情 / DFA / 分析表 / 测试报告）

图 3-2 系统总体流程

1.前端用户输入自定义文法（如 S>aA|bB; A>a; B>b）和测试用例（如 aa、ab）；

2..前端通过 AJAX 将数据提交至 Flask 后端的/analyze 接口；

3.后端接收数据后，调用 Grammar 类解析文法，生成拓广文法和符号集合；

4.调用 LROParser 类构建项目集规范族（DFA）和分析表，检测冲突；

5.若文法为 LR(0)，调用 AnalysisEngine 类对每个测试用例进行语法分析，生成分析日志；

6.调用 Visualizer 类生成 DFA 状态图、交互式分析表、分析过程 HTML 报告；

7.后端将文法信息、DFA 数据、分析表、测试结果封装为 JSON 返回前端；

8.前端渲染结果，展示文法详情、DFA 图、分析表、测试用例的分析过程和结果。

4 详细设计

4.1 核心模块流程与算法

4.1.1 文法解析流程

Grammar 类的核心功能是将用户输入的原始文法字符串解析为结构化数据，流程如图 4-1 所示：

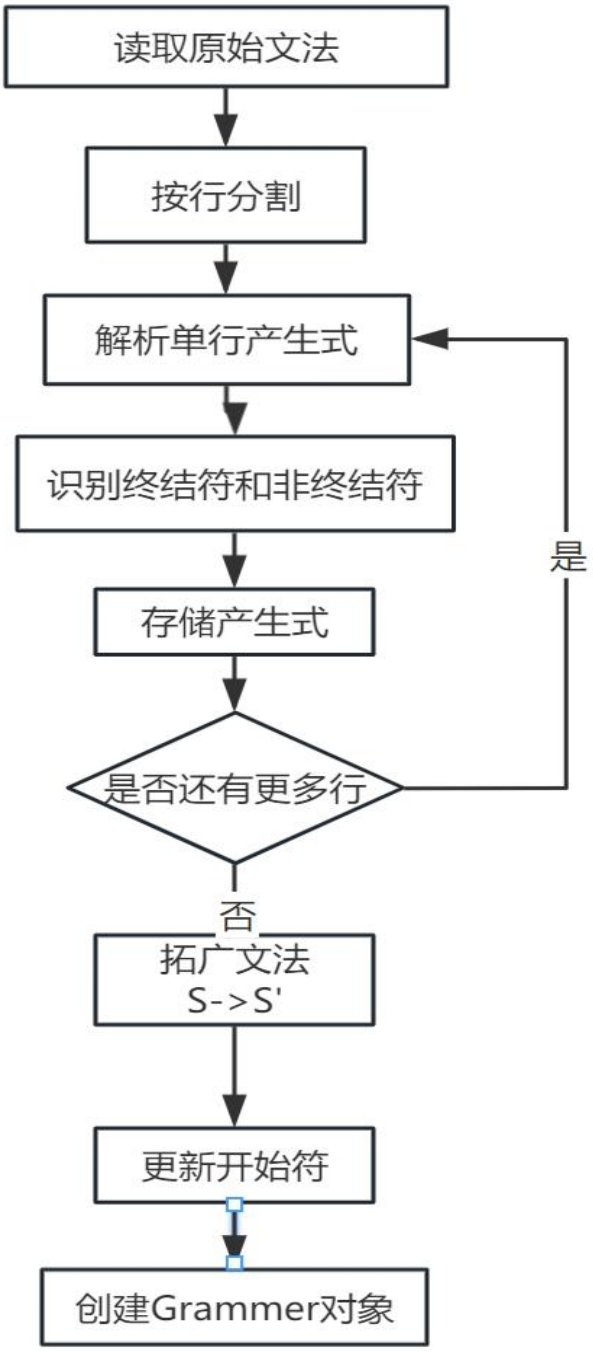

图 4-1 文法解析流程

1. 输入原始文法字符串，按行分割并过滤空行和注释行；

2. 对每行产生式，兼容>和=分隔符，拆分左部（LHS）和右部（RHS）；

3. 处理 RHS 的可选分支（|分隔），将每个分支拆分为符号列表；

4. 识别终结符（非大写字母且非@）和非终结符（大写字母）；

5. 处理空串：若 RHS 为空，替换为@；

6. 构建拓广文法：添加 S'→S 作为第 0 号产生式，更新开始符号和非终结符集合；

7. 输出结构化产生式、终结符集合、非终结符集合、开始符号。

4.1.2 项目集闭包运算算法

LR0Parser 类的_closure 方法实现闭包运算，算法流程如图 4-2 所示：

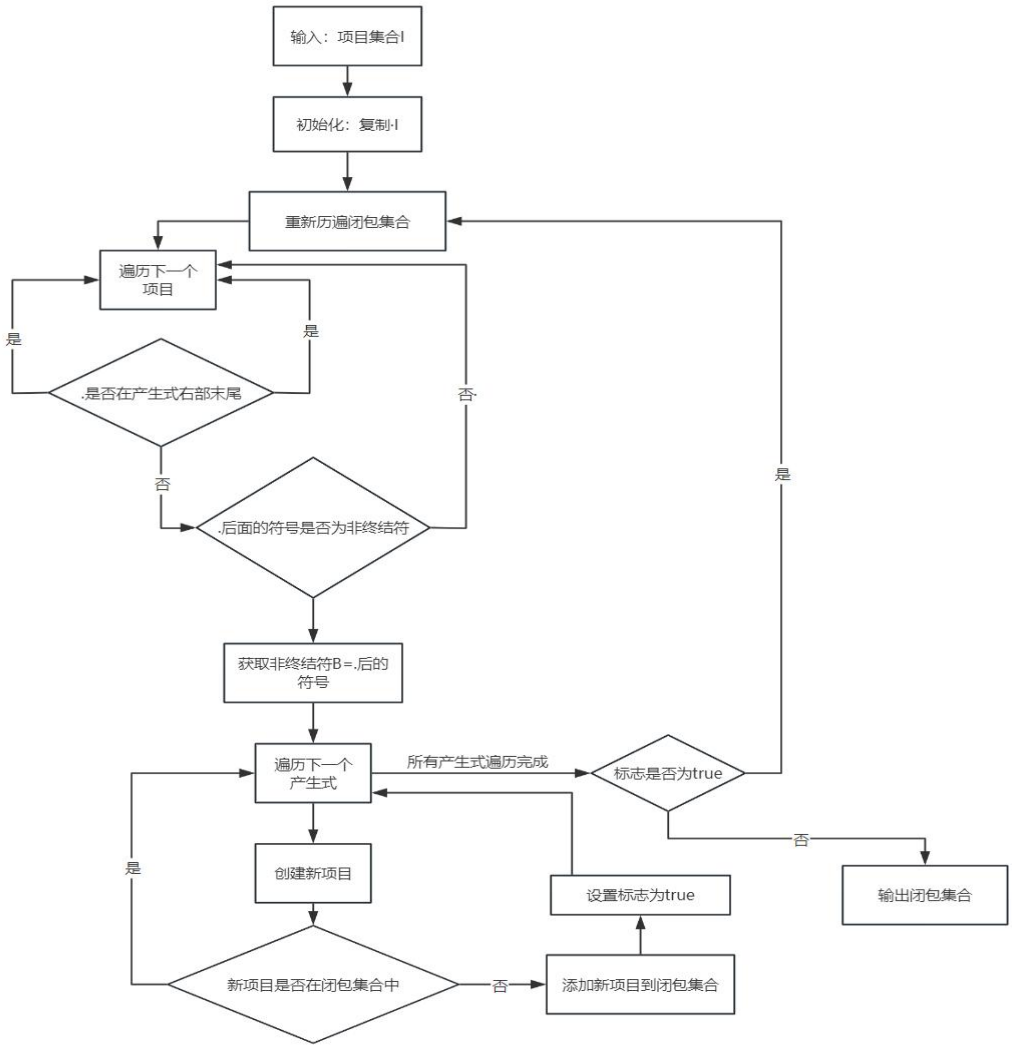

图 4-2 闭包运算流程

1.输入初始项目集 I，创建副本作为闭包初始集合；

2.遍历闭包集合中的每个项目，检查·是否在产生式右部末尾：

a.若·后为非终结符 B，遍历文法中所有以 B 为左部的产生式；

b.为每个产生式创建初始项目（dot=0），判断是否已存在于闭包集合；

c.若不存在，添加至闭包集合，并标记“添加新项目”；

3.重复步骤 2，直至无新项目可添加；

4.输出闭包集合 Closure(I)。

4.1.3 项目集规范族构建流程

LR0Parser 类的 build_canonical_collection 方法构建项目集规范族，流程如图 4-3 所示：

1.从拓广文法的初始项目 S’→·S 出发，计算闭包得到初始项目集 I0，加入状态列表；

2.初始化待处理状态队列，将 I0 的索引（0）加入队列；

3.取出队列头部的状态 Ii，遍历其中所有项目，收集·后非@的符号（终结符和非终结符）；

4.对每个符号 X，调用 Goto 函数计算 Goto(Ii, X)，得到新项目集 Ij；

5.检查 Ij 是否已存在于状态列表：

若不存在，添加至状态列表，记录状态转移(i, X)→新索引，将新索引加入队列；

若已存在，记录状态转移(i, X)→已存在索引；

6.重复步骤 35，直至队列空；

7.输出状态列表（项目集规范族）和状态转移表。

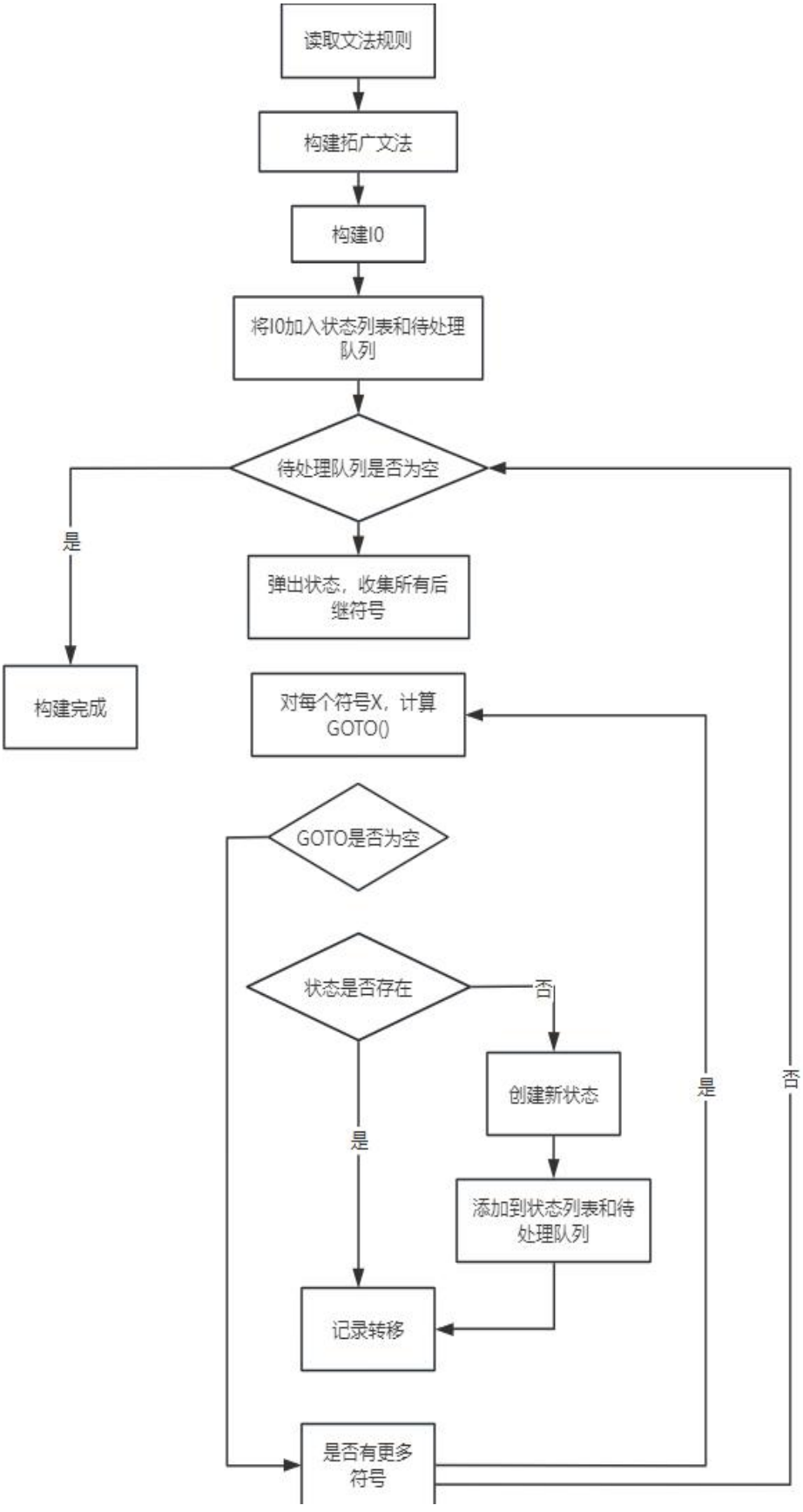

图 4-3 项目集规范族构建流程

4.1.4 语法分析流程

AnalysisEngine 类的 parse 方法实现移进-规约分析，进而实现对输入串的分析，流程如图 4-4 所示：

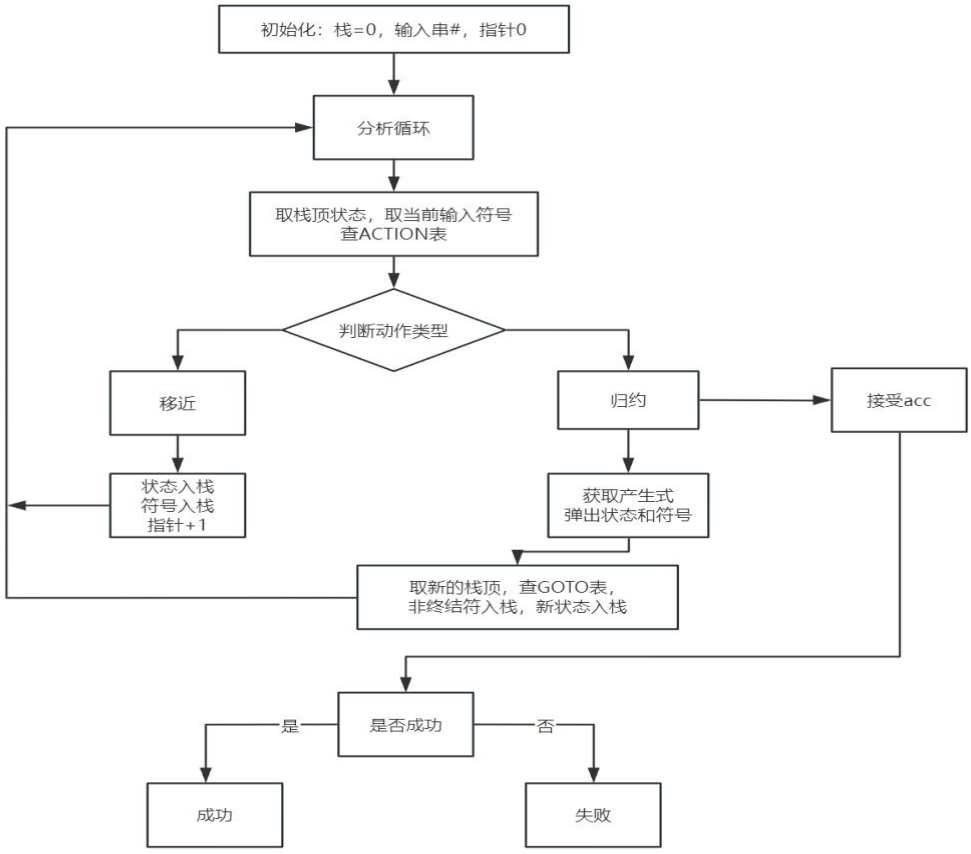

图 4-4 语法分析流程

1.初始化状态栈（压入 0）、符号栈（压入$）、输入指针（ptr=0）、步骤计数器（step=1）；

2.读取输入串，末尾添加$，生成输入符号列表；

3.循环：

获取当前栈顶状态 top_state 和当前输入符号 current_char；

查找 ACTION 表中 top_state 对应 current_char 的动作；

记录当前步骤的状态栈、符号栈（$替换为#）、剩余输入（$替换为#）、动作、GOTO 值；

4.处理动作：

空（错误）：返回分析失败和分析日志；

移进（s[状态号]）：目标状态入栈，当前符号入栈，指针 ptr+1；

规约（r[产生式号]）：按产生式右部长度弹出栈顶状态和符号，将左部非终结符入栈，查找 GOTO 表并压入目标状态，更新 GOTO 值；

接受（acc）：返回分析成功和分析日志；

5.步骤计数器 step+1；

6.输出分析结果（成功/失败）和详细分析日志。

4.2 主要类、属性与函数设计

4.2.1 Grammar 类（文法处理）

（1）属性：

productions：list[dict]，结构化产生式列表，每个元素为{'left': str, 'right': list[str]}。

terminals：set[str]，终结符集合（含$）。

non_terminals：set[str]，非终结符集合（含拓广文法开始符号）。

start_symbol：str，拓广文法的开始符号（如 S'）。

（2）方法

__init__(raw_productions: str):构造函数

功能：接收原始文法字符串并初始化文法对象。

_parse_grammar(raw_text: str)：核心文法解析器

功能：将文本形式的文法转换为结构化数据。

_augment_grammar(self):构建拓广文法

功能：添加 S'→S 产生式。

get_production_str(self, index: int):

功能：返回指定索引产生式的字符串形式（如 S'->S)。

4.2.2 LR0Parser 类（LR(0)核心算法）

（1）属性：

grammar：Grammar 对象，存储文法规则和符号信息。

states：list[list[dict]]，DFA 状态列表，每个状态是一个项目集列表。

transitions：dict[tuple[int, str], int]，状态转移表，键为(状态 ID, 符号)，值为目标状态 ID。

action_table：dict[int, dict[str, str]]，ACTION 分析表，外层键为状态 ID，内层键为终结符。

goto_table：dict[int, dict[str, int]]，GOTO 分析表，外层键为状态 ID，内层键为非终结符。

is_lr0：bool，LR(0)文法标志位，True 表示无冲突。

conflicts：list[str]，冲突记录列表，存储检测到的冲突详细信息。

conflict_state_ids：set[int]，冲突状态 ID 集合，便于快速查找冲突状态。

（2）方法：

__init__(grammar: Grammar)：构造函数

功能：接收 Grammar 对象并初始化分析器数据结构。

_get_item_str(item: dict)：辅助方法

功能：将项目对象转为字符串形式用于去重比较。

_items_equal(set1: list[dict], set2: list[dict])：辅助方法

功能：判断两个项目集是否相同。

_closure(items: list[dict])：核心算法

功能：计算 LR(0)项目集的闭包。

_goto(items: list[dict], symbol: str)：核心算法

功能：计算 GoTo(I, X)转移。

build_canonical_collection()：

功能：构建识别活前缀的 DFA（项目集规范族）。

build_parsing_table()：

功能：生成 LR(0)分析表并检测冲突。

_add_action(state: int, symbol: str, action: str)：

功能：向分析表添加动作，处理冲突检测。

print_dfa()：

功能：打印 DFA 状态集信息到控制台。

print_table()：

功能：打印格式化的分析表到控制台。

4.2.3 AnalysisEngine 类（语法分析引擎）

（1）属性

parser：LROParser 对象，存储已构建的分析表和文法信息。

（2）方法

__init__(parser: LROParser)：构造函数，接收已构建好的 LROParser 对象。

parse(input_string: str)：核心分析方法，对输入串执行 LR(0)语法分析。

4.2.4 Visualizer 类（可视化模块）

（1）属性

output_dir：统一管理所有可视化输出文件的存储位置。

（2）方法

render_dfa(states, transitions, terminals, conflict_states=None)

states:项目集,transitions:状态转移,terminals:终结符,conflict_states:冲突状态

功能：使用 Graphviz 绘制高质量的 LR(0) DFA 状态转换图。

render_trace_html(trace_data, input_str, filename="trace_log.html")：

功能：生成详细的输入串分析过程追踪表格。

render_dashboard(info_dict, filename="index.html")：

功能：生成综合性仪表盘，集成所有分析结果。

4.2.5 Flask 后端核心函数（app.py）

(1)方法

analyze_grammar(grammar_text, input_strings=None)：

功能: 核心分析函数，协调各个模块完成完整的 LR(0)文法分析流程

analyze()：

功能: 分析请求处理函数，接收前端 JSON 请求并返回分析结果

index()：

功能: 主页路由处理函数，渲染前端界面

返回值: 渲染后的 index.html 模板

5 编码实现

5.1 开发环境设置

5.1.1 环境配置

Python 版本：3.13.16；

核心依赖 (requirements.txt)：

```
Flask =3.1.2              # Web 框架
graphviz>=0.21            # DFA 图生成
pandas>=2.3.3             # 数据处理
```

所有依赖如下表 5-1：

表 5-1 Package Version 信息表

Package	Version
blinker	1.9.0
click	8.3.1
colorama	0.4.6
flask	3.1.2
graphviz	0.21
itsdangerous	2.2.0
jinja2	3.1.6
markupsafe	3.0.3
numpy	2.4.0
pandas	2.3.3
pip	25.1.1
python - dateutil	2.9.0.post0
pytz	2025.2
six	1.17.0
tzdata	2025.3
werkzeug	3.1.4

5.1.2 项目结构

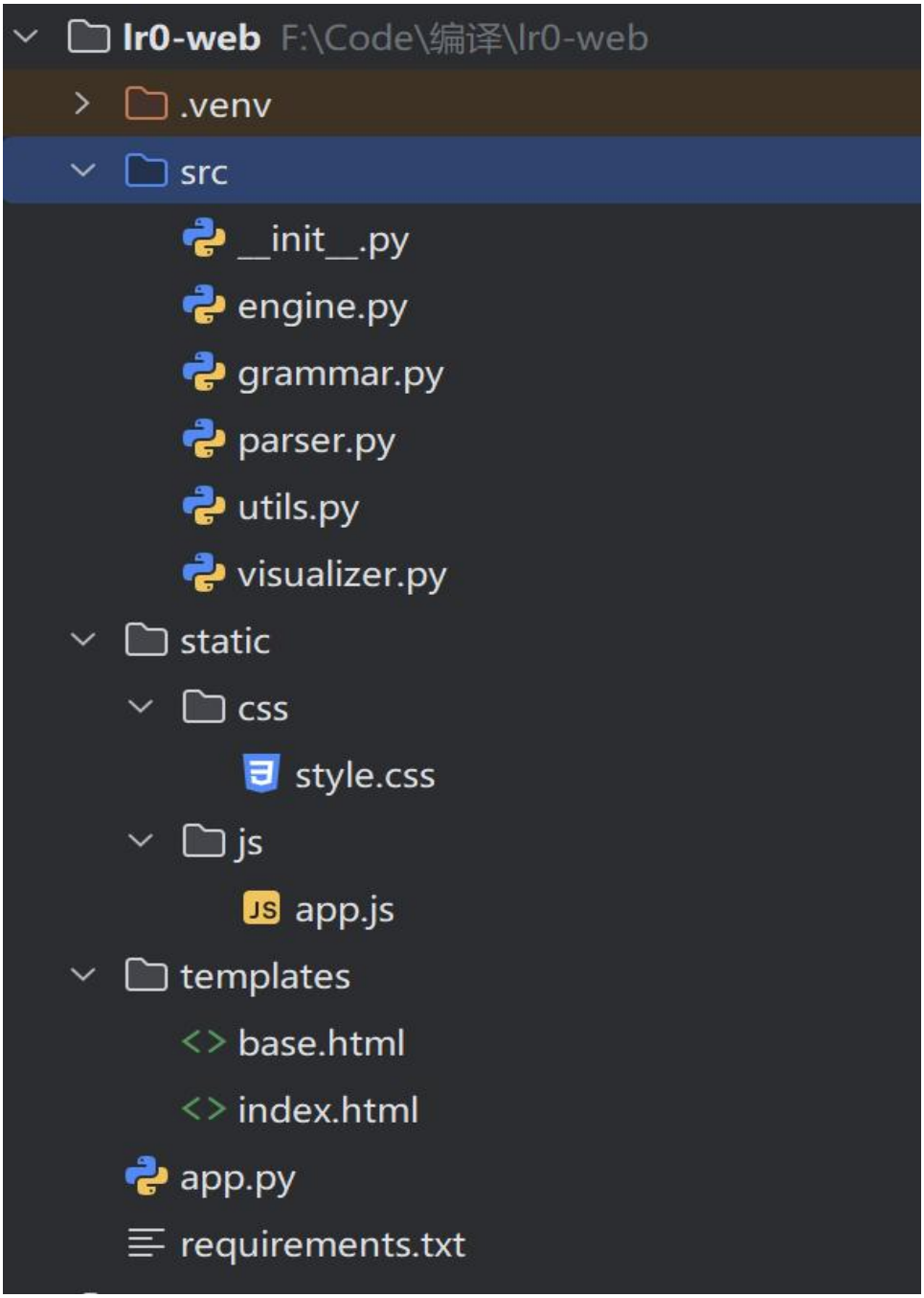

图 5-1 项目结构

5.2 程序设计注意事项

文法兼容性：需支持>和=两种分隔符，处理符号带空格和无空格的情况，过滤注释行（#开头）；

符号识别准确性：严格区分终结符（非大写字母且非@）和非终结符（大写

字母），避免混淆；

空串处理：统一使用@表示空串，在项目集构建、分析表生成、分析过程中确保逻辑一致；

冲突检测完整性：在_add_action 方法中，需检测同一状态符号对应的多个动作，记录冲突并标记冲突状态；

可视化适配性：DFA 图需避免节点重叠，分析表需支持中文对齐，分析过程需清晰展示栈变化；

符号显示一致性：输入结束符$在前端展示时替换为#，避免与用户输入符号冲突；

错误处理：完善文法格式错误、Graphviz 渲染失败、分析表查找错误等场景的提示信息。

5.3 关键构件/插件的特点和使用

5.3.1 Graphviz

特点：开源图形可视化工具，支持 DOT 语言描述的图形渲染，适用于状态图、流程图生成；

使用：在 Visualizer.render_dfa 中，通过 Digraph 对象定义节点和边，设置布局参数（rankdir=LR、nodesep=2.0 等），调用 render 方法生成 PNG 文件；

优化：通过调整 dpi、splines、margin 等参数，解决节点重叠和边交叉问题。

5.3.2 Flask

特点：轻量级 Python Web 框架，路由简单，支持 RESTful API，便于快速构建后端服务；

使用：通过@app.route 装饰器定义/（首页）和/analyze（分析接口），request.get_json()接收前端数据，jsonify 返回结果；

优势：无需复杂配置，支持调试模式，便于开发和测试。

5.3.3 Bootstrap 5

特点：响应式前端框架，提供现成的组件（卡片、表格、按钮、折叠面板），支持多设备适配；

使用：在 index.html 中引入 CSS 和 JS 文件，使用 container、card、table 等类构建界面，确保在 PC 和移动端的适配性。

5.3.4 DataTables

特点：基于 jQuery 的交互式表格插件，支持排序、搜索、分页，提升分析表的可读性；

使用：在前端页面中初始化 DataTable 对象，禁用分页和信息展示，适配分析表的静态数据展示需求。

5.4 主要程序代码设计及注释

5.4.1 LR0Parser._closure（闭包运算核心代码）

```
    def _closure(self, items):
        """计算闭包 Closure(I)"""
        closure_set = [item.copy() for item in items]
        while True:
            added_new = False
            for item in closure_set:
                rhs = item['right']
                dot = item['dot']
                if dot < len(rhs):
                    symbol = rhs[dot]
                    if symbol in self.grammar.non_terminals:
                        for prod in self.grammar.productions:
                            if prod['left'] == symbol:
                                new_item = {'left': prod['left'],
'right': prod['right'], 'dot': 0}
                                exists = False
                                for existing in closure_set:
                                    if existing == new_item:
                                        exists = True
                                        break
                                if not exists:
                                    closure_set.append(new_item)
                                    added_new = True
```

```
    def _goto(self, items, symbol):
        """计算 GoTo(I, X)"""
        next_items = []
        for item in items:
            rhs = item['right']
            dot = item['dot']
            if dot < len(rhs) and rhs[dot] == symbol:
                new_item = item.copy()
                new_item['dot'] += 1
                next_items.append(new_item)
        return self._closure(next_items)
```

5.4.2 build_canonical_collection（构建识别活前缀的 DFA）

```
    def build_canonical_collection(self):
        """构建识别活前缀的 DFA"""
        print("正在构建项目集规范族 (DFA)...")
        start_prod = self.grammar.productions[0]
        initial_item = {'left': start_prod['left'], 'right':
start_prod['right'], 'dot': 0}
        initial_state = self._closure([initial_item])

        self.states.append(initial_state)
        to_process = [0]

        while to_process:
            current_idx = to_process.pop(0)
            current_items = self.states[current_idx]

            symbols = set()
            for item in current_items:
                if item['dot'] < len(item['right']):
                    symbol = item['right'][item['dot']]
                    # 关键修改：跳过 ε 符号（@），不为其创建转移
                    if symbol != '@':
                        symbols.add(symbol)

            for sym in sorted(list(symbols)):
                next_state_items = self._goto(current_items, sym)
                if not next_state_items:
                    continue
```

```
                existing_idx = -1
                for idx, state in enumerate(self.states):
                    if self._items_equal(state, next_state_items):
                        existing_idx = idx
                        break

                if existing_idx == -1:
                    self.states.append(next_state_items)
                    new_idx = len(self.states) - 1
                    self.transitions[(current_idx, sym)] = new_idx
                    to_process.append(new_idx)
                else:

self.transitions[(current_idx,sym)]=existing_idx
```

5.4.3 LR0Parser.build_parsing_table（分析表构建与冲突检测）

```
    def build_parsing_table(self):
        """生成分析表并检测冲突"""
        n_states = len(self.states)
        for i in range(n_states):
            self.action_table[i] = {}
            self.goto_table[i] = {}

        for i, state_items in enumerate(self.states):
            # 1. 移进 (Shift)
            for (src, sym), dest in self.transitions.items():
                if src == i and sym in self.grammar.terminals:
                    # 关键修改：确保 $ 也添加移进动作
                    self._add_action(i, sym, f"s{dest}")

            # GOTO 表
            for (src, sym), dest in self.transitions.items():
                if src == i and sym in self.grammar.non_terminals:
                    self.goto_table[i][sym] = dest
           # 2. 规约 (Reduce) 和 接受 (Accept)
            for item in state_items:
                if item['dot'] == len(item['right']) or
(item['right'] == ['@'] and item['dot'] == 0):
                    # 接受动作：当点在最右端且左部是拓广文法的开始
符号
                    if item['left'] == self.grammar.start_symbol and
item['dot'] == len(item['right']):
                     self._add_action(i, '$', "acc")
```

```
                    else:
                        # 规约动作
                        prod_idx = -1
                        for idx, p in
enumerate(self.grammar.productions):
                            if p['left'] == item['left'] and
p['right'] == item['right']:
                                prod_idx = idx
                                break

                        if prod_idx >= 0:
                            action_str = f"r{prod_idx}"
                            # LR(0) 核心：对所有终结符都进行规约（包
括 $）
                            for term in self.grammar.terminals:
                                if term !='$':  # $ 可能有接受动作，
避免覆盖
                                    self._add_action(i, term,
action_str)
                            # 额外添加 $ 的规约动作（如果没有接受动
作冲突的话）
                            if '$' not in self.action_table[i] or
self.action_table[i]['$'] != 'acc':
                                self._add_action(i, '$',
action_str)

    def _add_action(self, state, symbol, action):
        """
        添加动作，支持显式记录冲突
        """
        if symbol in self.action_table[state]:
            existing = self.action_table[state][symbol]
            # 如果动作不一样，说明冲突 (例如 s3 vs r2)
            if existing != action and action not in existing:
                self.is_lr0 = False

                # [核心修复] 记录冲突状态 ID
                self.conflict_state_ids.add(state)

                new_val = existing + "/" + action
                self.action_table[state][symbol] = new_val
```

```
                conflict_msg = f"State {state}, Symbol '{symbol}':
Conflict [{new_val}]"
                if conflict_msg not in self.conflicts:
                    self.conflicts.append(conflict_msg)
        else:
            self.action_table[state][symbol] = action
```

5.4.4 AnalysisEngine（语法分析核心代码）

```
class AnalysisEngine:
    def __init__(self, parser):
        self.parser = parser

    def parse(self, input_string: str):
        if not self.parser.is_lr0:
            return False, []

        stack = [0]
        symbol_stack = ['$']  # 内部保持 $

        trace_log = []

        input_tokens = list(input_string) + ['$']  # 内部保持 $
        ptr = 0
        step = 1

        while True:
            top_state = stack[-1]
            current_char = input_tokens[ptr]
            action =
self.parser.action_table[top_state].get(current_char)
            state_stack_str = " ".join(map(str, stack))
            symbol_stack_str = "".join(symbol_stack).replace('$',
'#')  # $ 替换为 #
            remaining_input =
"".join(input_tokens[ptr:]).replace('$', '#')  # $ 替换为 #
            # 初始化 GOTO 列
            goto_value = ""

            step_info = {
                "step": step,
                "state_stack": state_stack_str,
                "symbol_stack": symbol_stack_str,
```

```
                "input": remaining_input,  # 这里已经是替换后的值
                "action": str(action) if action else "ERROR",
                "goto": goto_value
            }
            trace_log.append(step_info)

            if action is None:
                return False, trace_log

            # === SHIFT ===
            if action.startswith('s'):
                next_state = int(action[1:])
                stack.append(next_state)
                symbol_stack.append(current_char)
                ptr += 1

            # === REDUCE ===
            elif action.startswith('r'):
                prod_idx = int(action[1:])
                production =
self.parser.grammar.productions[prod_idx]
                lhs = production['left']
                rhs = production['right']

                pop_len = len(rhs)
                if pop_len == 1 and rhs[0] == '@':
                    pop_len = 0


                if pop_len > 0:
                    stack = stack[:-pop_len]
                    symbol_stack = symbol_stack[:-pop_len]

                current_top = stack[-1]
                if lhs in self.parser.goto_table[current_top]:
                    goto_state =
self.parser.goto_table[current_top][lhs]
                    stack.append(goto_state)
                    symbol_stack.append(lhs)
```

```
                    # === 更新 GOTO 列 ===
                    trace_log[-1]['goto'] = str(goto_state)  # 英文小写
                else:
                    return False, trace_log

            # === ACCEPT ===
            elif action == 'acc':

                return True, trace_log

            step += 1
```

5.4.5 Visualizer（DFA 可视化代码）

```
    def render_dfa(self, states, transitions, terminals, conflict_states=None):
        # 1. 绘制节点 (状态)
        for i, items in enumerate(states):
            is_conflict = i in conflict_states
            title_bg = "#ffcccc" if is_conflict else "#E0E0E0"
            title_text = f"I{i}   " if is_conflict else f"I{i}"
            border_color = "red" if is_conflict else "black"

            # === 优化：限制每个状态显示的项目数量 ===
            display_items = items
            if len(items) > 8:  # 如果项目太多，只显示前 8 个
                display_items = items[:8]
                # 添加省略号提示
                title_text = f"I{i} ({len(items)}项)"

            # 构造 HTML 表格 - 简化显示
            label_html = f'''<<TABLE BORDER="0" CELLBORDER="1" CELLSPACING="0" CELLPADDING="3" COLOR="{border_color}">
                        <TR><TD BGCOLOR="{title_bg}" BORDER="1"
```

```
            for item in display_items:
                rhs = item['right'][:]
                rhs.insert(item['dot'], '•')
                lhs_esc = html.escape(item['left'])
                rhs_esc = html.escape("".join(rhs))

                # 简化显示：对于长产生式进行截断
                if len(rhs_esc) > 15:
                    rhs_esc = rhs_esc[:15] + "..."

                bg_color = "#ffffff"
                if    item['dot']    ==    len(item['right'])    or
(item['right'] == ['@'] and item['dot'] == 0):
                    bg_color = "#e6fffa"

                label_html      +=      f'<TR><TD      ALIGN="LEFT"
BGCOLOR="{bg_color}"          COLSPAN="2">{lhs_esc}          &rarr;
{rhs_esc}</TD></TR>'

            # 如果项目太多被截断，显示提示
            if len(items) > len(display_items):
                label_html     +=     f'<TR><TD     ALIGN="CENTER"
BGCOLOR="#f0f0f0"   COLSPAN="2">...   还   有     {len(items)   -
len(display_items)} 项</TD></TR>'

            label_html += "</TABLE>>"

            # 为节点设置固定大小，防止节点过大
            dot.node(str(i),
                     label=label_html,
                     _attributes={'width': '1.2', 'height': '0.8'}
if len(items) > 5 else {})
        # 2. 绘制边 (转移) - 优化边的显示
        for (start_idx, sym), end_idx in transitions.items():
            if sym in terminals:
                color = "#0056b3"  # 蓝色
                style = "solid"
                penwidth = "1.5"
                # 为终结符添加特殊标签
                if sym == '#':  # 处理结束符
                    label = " # "
                else:
                    label = f" {sym} "
```

```
            else:
                color = "#d9534f"  # 红色
                style = "dashed"
                penwidth = "1.2"
                label = f" {sym} "

            # === 优化：调整边的位置，减少交叉 ===
            dot.edge(str(start_idx), str(end_idx),
                     label=label,
                     color=color,
                     style=style,
                     fontcolor=color,
                     penwidth=penwidth,
                     # 添加约束，减少不必要的弯曲
                     constraint='true',
                     # 边的标签位置调整
                     labeldistance='2.5',
                     labelangle='25')

        # 3. 保存并渲染
        output_path = os.path.join(self.output_dir, 'dfa_graph')
        try:
            # 尝试不同的布局引擎
            dot.engine = 'dot'  # 使用 dot 引擎，更适合层次结构

            dot.render(output_path, view=False, cleanup=True)
            print(f"      -> [Graphviz] DFA 高清图已生成: {output_path}.png")

            # 如果第一次效果不好，尝试不同的随机种子
            if len(states) > 15:  # 状态较多时才尝试
                for attempt in range(3):
                    dot.attr(start=str(attempt + 10))  # 改变随机种子
                    alt_path = os.path.join(self.output_dir, f'dfa_graph_alt{attempt}')
                    dot.render(alt_path, view=False, cleanup=True)
                    print(f"   -> [Graphviz] 备选布局 {attempt + 1} 已生成")
```

5.5 解决的技术难点与经常犯的错误

5.5.1 技术难点及解决方案

(1) 项目集的精准去重：

难点：项目集是字典列表，直接比较难以判断是否相等；

解决方案：设计_get_item_str 方法将项目转换为字符串（如 S→a·A），通过集合比较判断项目集是否相等。

(2) 冲突状态的准确检测与标记：

难点：同一状态符号可能对应多个动作（移进/规约），需完整记录冲突并标记冲突状态；

解决方案：在_add_action 方法中，检测动作冲突时，更新 is_lr0 标志位，记录冲突状态 ID 和冲突信息，在 DFA 图和分析表中可视化标记。

(3) DFA 可视化的节点重叠与边交叉：

难点：项目集数量较多时，节点易重叠，边交叉导致图形混乱；

解决方案：调整 Graphviz 布局参数（nodesep 增大节点间距、splines=ortho 使用正交线、rankdir=LR 从左到右布局），限制每个节点显示的项目数量，对长产生式截断处理。

(4) 输入串中$与#的替换一致性：

难点：$作为输入结束符，需在分析过程中隐藏，避免与用户输入符号冲突；

解决方案：在 AnalysisEngine.parse 中，将符号栈和输入串中的$替换为#后再记录日志，在前端展示时统一使用#作为结束符。

(5) Web 前后端数据交互的格式一致性

难点：后端生成的分析表、分析日志需转换为前端可渲染的格式；

解决方案：定义统一的 JSON 数据格式，后端将复杂数据（如项目集、分析表）结构化，前端通过 AJAX 接收后，按格式渲染 DFA 图、分析表和分析过程。

5.5.2 经常犯的错误及修正

(1) 终结符与非终结符的混淆：

错误：在构建 GOTO 表时，误将终结符作为列名，导致 GOTO 表生成错误；

修正：严格区分符号类型，GOTO 表仅处理非终结符，ACTION 表仅处理终结符，通过 grammar.terminals 和 grammar.non_terminals 过滤符号类型。

(2) 空串产生式的闭包计算错误：

错误：未处理 A→@（空串）的项目，导致闭包计算不完整；

修正：在_closure 方法中，允许·后为非终结符时，遍历所有含该非终结符的产生式，包括空串产生式。

(3) 分析表中$符号的动作覆盖：

错误：规约动作覆盖了$符号的接受动作，导致分析成功时无法触发 acc；

修正：在添加规约动作时，判断$符号是否已存在接受动作，若存在则不覆盖。

(4) 可视化时 HTML 转义错误：

错误：产生式中含特殊字符（如<、&），导致 HTML 渲染失败；

修正：使用 html.escape 函数对产生式的左部和右部进行转义，避免 HTML 语法冲突。

(5) Flask 后端跨域问题：

错误：前端 AJAX 请求后端时，出现跨域访问限制；

修正：在 Flask 中添加跨域支持（安装 flaskcors，添加 CORS(app)），允许前端跨域请求。

6 测试和试运行

6.1 测试方法

本次测试采用“分层测试+全覆盖测试”策略，确保系统功能的正确性和鲁棒性：

单元测试：对核心模块的关键方法进行独立测试，验证算法正确性（如Grammar._parse_grammar、LROParser._closure、AnalysisEngine.parse）；

集成测试：测试模块间的协作流程，验证从文法输入到结果输出的端到端功能（如文法解析→DFA 生成→分析表构建→输入串分析→可视化展示）；

功能测试：验证所有核心功能（文法兼容性、冲突检测、输入串分析、可视化）是否正常工作；

兼容性测试：测试不同浏览器（Chrome、Firefox、Edge）、不同 Python 版本（3.8、3.9、3.10）的运行情况；

边界测试：测试特殊用例（空串文法、单符号文法、含冲突的文法、超长输入串）；

用户测试：邀请同学使用系统，收集反馈，优化交互体验。

6.2 测试用例与结果

6.2.1 测试用例设计

表 6-1 文法测试信息表

测试类型	测试用例	预期结果
合法 LR(0)文法	文法：S -> a A | b B A -> c A | d B -> c B | d 测试用例：bccd,ad,abd	文法判定为 LR(0)，无冲突；bccd、ad 分析成功，abd 分析失败
表达式文法	文法：S -> E E -> (E + E)|(E * E)| + 测试用例：+，(+*+)	文法判定为 LR(0)；+，(+*+)分析成功
非 LR(0)文法(移进规约冲突)	文法：S -> A c S -> a c A -> a	文法判定为非 LR(0)，检测到冲突；aa 分析失败

文法格式兼容性	文法：S = a A ｜ b B A = c A ｜ d B = c B ｜ d（使用=分隔） 测试用例：bccd、ad	文法解析成功，判定为 LR(0)； bccd、ad 分析成功

6.2.2 测试结果示例

（1）合法 LR(0)文法测试：

输入文法：S -> a A ｜ b B，A -> c A ｜ d，B -> c B ｜ d

测试用例：bccd，ad，aa

测试结果：

文法判定：LR(0)文法，无冲突；

DFA 生成：成功生成 11 个状态的 DFA 图；

分析表：ACTION 表和 GOTO 表无冲突；

输入串分析：分析成功

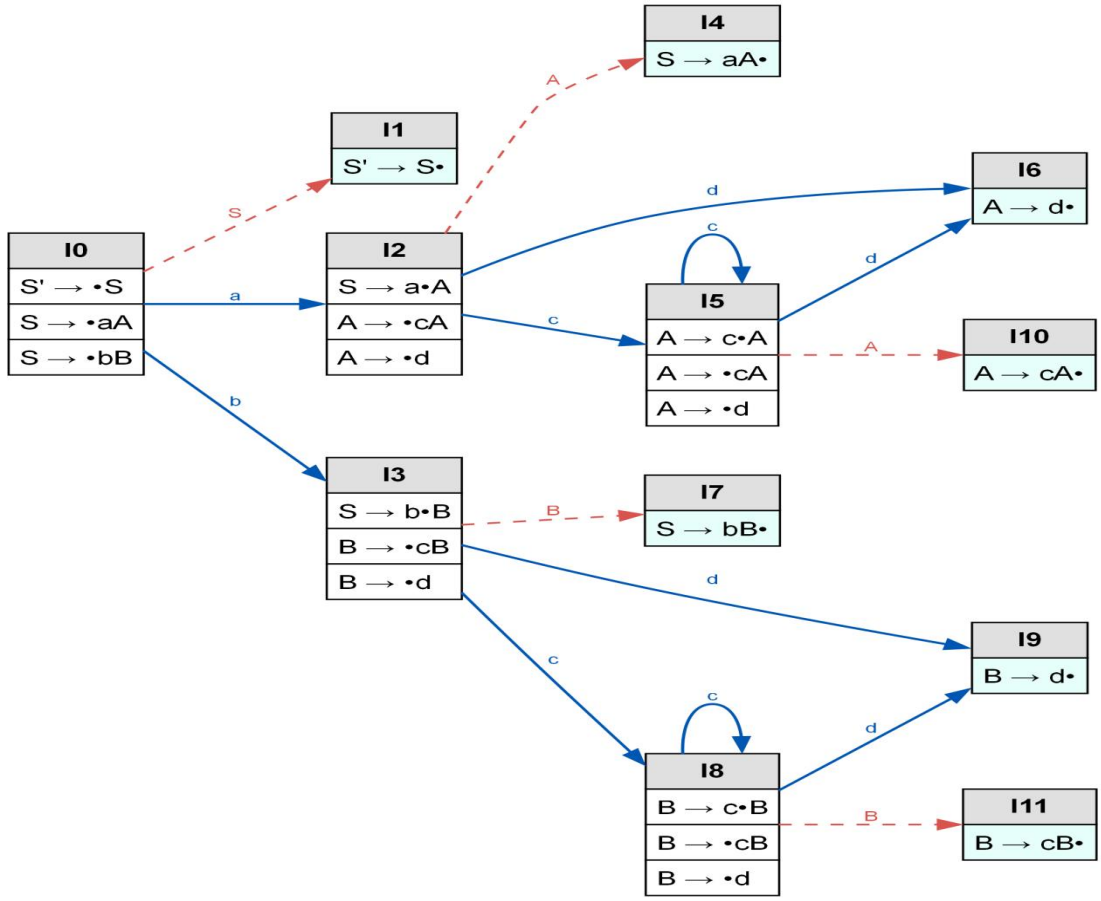

图 6-1 LR(0)项目集规范族

LR(0) 分析表

State	ACTION					GOTO		
	a	b	c	d	#	A	B	S
0	s2	s3	-	-	-	-	-	1
1	-	-	-	-	acc	-	-	-
2	-	-	s5	s6	-	4	-	-
3	-	-	s8	s9	-	-	7	-
4	r1	r1	r1	r1	r1	-	-	-
5	-	-	s5	s6	-	10	-	-
6	r4	r4	r4	r4	r4	-	-	-
7	r2	r2	r2	r2	r2	-	-	-
8	-	-	s8	s9	-	-	11	-
9	r6	r6	r6	r6	r6	-	-	-
10	r3	r3	r3	r3	r3	-	-	-
11	r5	r5	r5	r5	r5	-	-	-

图 6-2 LR(0)分析表

输入串分析结果

接受 输入: bccd

步骤	状态栈	符号栈	输入串	ACTION	GOTO
1	0	#	bccd#	s3	
2	0 3	#b	ccd#	s8	
3	0 3 8	#bc	cd#	s8	
4	0 3 8 8	#bcc	d#	s9	
5	0 3 8 8 9	#bccd	#	r6	11
6	0 3 8 8 11	#bccB	#	r5	11
7	0 3 8 11	#bcB	#	r5	7
8	0 3 7	#bB	#	r2	1
9	0 1	#S	#	acc	

分析结果: 输入串 bccd 是该文法的句子。

接受 输入: ad

拒绝 输入: aa

图 6-3 对输入串分析结果

（2）非 LR(0)文法测试：

输入文法：S -> A c，S -> a c，A -> a 测试用例：aa

测试结果：

文法判定：非 LR(0)文法，检测到冲突：State 3, Symbol 'a': Conflict [s4/r0]；

DFA 生成：冲突状态 3 标记为红色；

输入串分析：分析失败，ACTION 表中状态 3 对应 a 的动作为 s4/r0，无法确定动作。

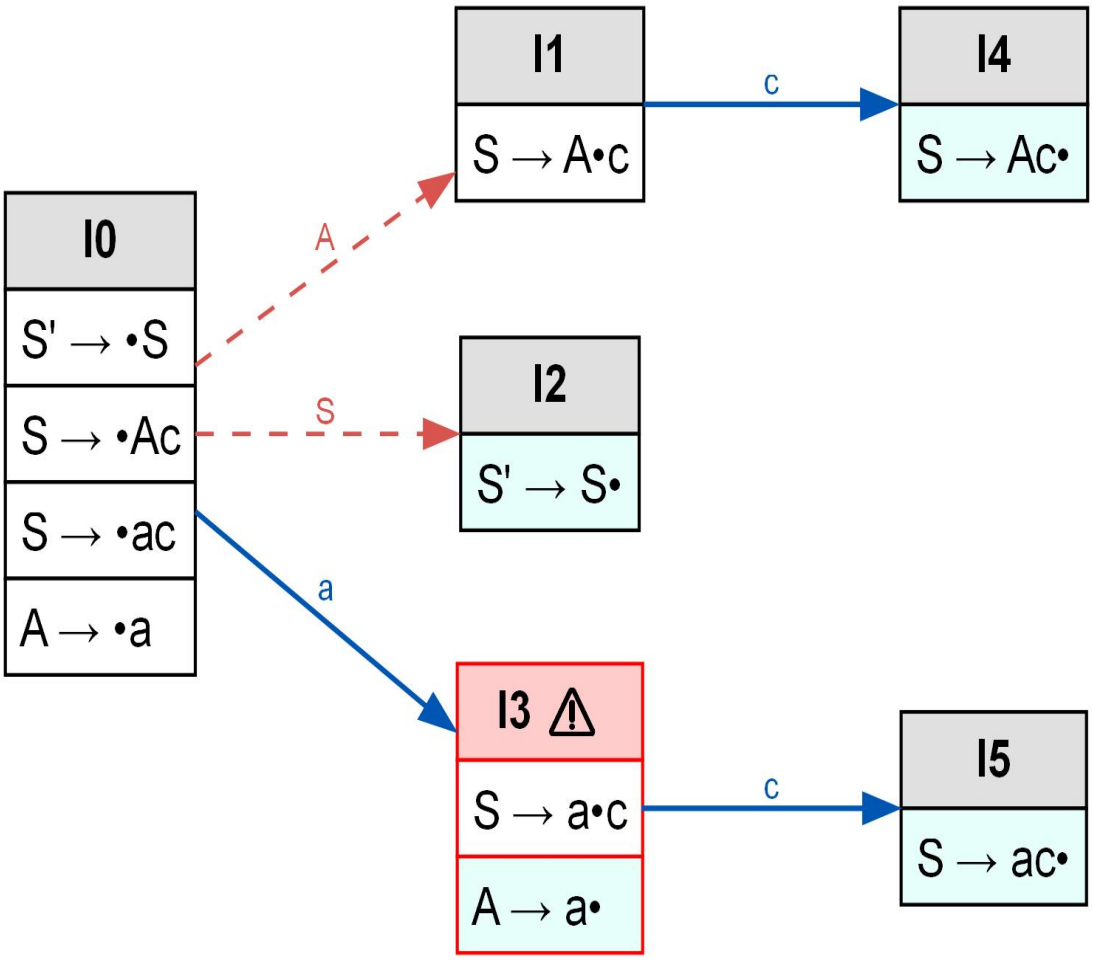

图 6-4 非 LR(0)项目集规范族

6.3 测试中出现的问题及解决方法

表 6-2 测试问题分析与解决表

测试问题	原因分析	解决方法
含@的产生式闭包计算不完整	未处理 A→@的项目，闭包中未添加该项目	在_closure 方法中，允许·后为非终结符时，遍历所有含该非终结符的产生式，包括 A→@
DFA 图中节点重叠严重	项目集数量多，Graphviz 默认布局参数不合理	调整 nodesep（节点间距）、ranksep（层级间距），使用 ortho 正交线，限制节点显示的项目数量
分析表中$符号显示为#，但动作中仍含$	仅在日志展示时替换$，未处理动作中的$	在 app.py 的分析表构建中，将动作中的$替换为#后再返回前端
前端分析表渲染时，ACTION 和 GOTO 列分界错误	未正确识别终结符和非终结符的分界位置	在 app.js 的 renderParsingTable 中，通过查找第一个非终结符的位置确定 ACTION 和 GOTO 列的分界
Graphviz 渲染失败	未安装 Graphviz 软件或未配置环境变量	提示用户安装 Graphviz 并配置环境变量，提供备选引擎（neato）
超长输入串分析时，分析日志展示混乱	未分页或滚动展示分析日志	在前端使用折叠面板和滚动容器，优化长日志的展示体验

7 总结

7.1 课程设计完成情况

本次课程设计已完成LR(0)文法可视化分析器的全部核心功能，达到了预期目标：

文法处理：支持自定义文法输入，兼容>和=分隔符，正确识别终结符、非终结符，处理空串并构建拓广文法；

LR(0)核心算法：实现了闭包运算、Goto函数、项目集规范族构建、分析表生成、冲突检测等关键算法；

语法分析：基于分析表实现移进规约分析，支持输入串的语法验证，生成详细的分析日志；

可视化展示：生成DFA状态转移图（标记冲突状态）、交互式分析表、分析过程追踪报告和综合仪表盘；

Web交互：基于Flask实现前后端分离，提供直观的用户界面，支持多浏览器适配。

未完成的部分（可扩展方向）：

语义分析集成：当前仅支持语法分析，可添加语义分析模块，生成中间代码（如四元式）或直接计算表达式结果；

目标代码生成：支持将中间代码转换为汇编语言或机器语言；

文法优化：添加文法简化、左递归消除、回溯消除等自动优化功能；

性能优化：针对大规模文法（多状态、多产生式）优化算法效率，减少时间和空间复杂度。

7.2 收获与经验

理论与实践的深度融合：通过实现 LR(0)核心算法，深刻理解了项目集、闭包、Goto 函数、分析表等抽象概念的内在逻辑，将编译原理教材中的理论知识转化为可执行的代码；

面向对象设计的优势：采用模块化的面向对象设计，使各模块边界清晰、耦合度低，便于调试和功能扩展，例如后续扩展 SLR(1)时，可继承 LR0Parser 类，仅修改分析表构建逻辑；

可视化对调试的帮助：DFA 图和分析过程的可视化不仅提升了用户体验，也为调试提供了极大便利，通过可视化可以快速定位项目集构建错误、冲突检测遗漏等问题；

工程问题的解决能力：在开发过程中，遇到了符号识别、冲突检测、可视化适配等一系列工程问题，通过查阅文档、调试代码、优化算法，逐步提升了问题分析和解决能力；

跨领域技术的整合能力：本次设计整合了编译原理、Python 编程、Web 开发（Flask）、可视化技术（Graphviz）、前端开发（HTML/CSS/JavaScript）等多个领域的知识，提升了跨领域技术的整合能力。

7.3 教训与感受

细节决定成败：编译原理算法对细节要求极高，一个微小的错误（如符号类型判断错误、$和#的替换不一致）会导致整个系统功能异常，后续开发中需更加注重细节，编写单元测试验证关键逻辑；

前期设计的重要性：初期未充分考虑文法的兼容性和边界用例，导致后期频繁修改代码，后续开发应在前期进行充分的需求分析和设计，明确功能边界和异常处理方案；

文档和注释的必要性：核心算法（如闭包、Goto）逻辑复杂，缺乏注释会导致后期维护困难，应养成编写详细注释和文档的习惯，提高代码的可读性和可维护性；

用户体验的重要性：初期仅关注功能实现，忽略了用户交互体验（如分析表展示混乱、DFA 图节点重叠），后续通过优化可视化布局和前端界面，显著提升了系统的易用性，意识到“功能正确”和“用户好用”同样重要。

本次课程设计不仅提升了技术能力，更让我体会到了编译原理的魅力——通过严谨的算法和模型，将复杂的语法规则转化为确定性的分析过程。虽然开发过程中遇到了诸多困难，但通过不断调试和优化，最终实现了预期目标，这种从“理论”到“实践”的跨越，是本次课程设计最大的收获。

8 参考文献

[1] 王生原,董渊,张素琴,吕映芝,蒋维杜.编译原理（第 3 版）[M].北京:清华大学出版社,, 2009: 123-148.

[2]袁国忠. Python 编程：从入门到实践 [M]. 北京：人民邮电出版社，2016: 150-180.

[3] Miguel Grinberg. Flask Web 开发：基于 Python 的 Web 应用开发实战 [M]. 北京：人民邮电出版社，2018: 30-60.

[4] Graphviz 官方文档. https://graphviz.org/documentation/, 2024.

[5] Flask 官方文档. https://flask.palletsprojects.com/, 2024.

[6] DataTables 官方文档. https://datatables.net/manual/, 2024.

[7] 王树森. LR (0) 文法分析器的设计与实现 [J]. 计算机工程与应用，2018, 54 (12): 102-106.