xx大学学生实验报告

开课学院及实验室：计算机科学与工程实验室　　2023年11月23日

学院		**年级/专业/班**		**姓名**		**学号**	
实验课程名称		编译原理实验				**成绩**	
实验项目名称		语法分析				**指导老师**	

组员分工表（前面3列、后面2列的内容不得修改）

学号	姓名	班级	项目角色	业务分工	评语	成绩
			负责人			
			组员			
			组员			

一、实验目的

设计、编制并调试一个语法分析程序，加深对语法分析原理的理解。

二、基本知识

1、上下文无关文法

2、无左递归、无回溯文法

3、LL(1)分析法

三、实验环境

1、Windows操作系统

2、C/C++/Java语言

四、实验要求

1、做好实验预习，掌握并熟悉本实验中所使用的编程、测试环境及相应的软件

2、写出实验报告

特别注意，不得使用与编译原理相关的第三方工具、构件，如 lex、yacc、antlar。但可以使用通用场景用途的工具，比如 STL、vue 等。

五、实验内容

C 语言子集文法如下：

<函数定义> -> <修饰词闭包><类型><变量>(<参数声明>){<函数块>}
<修饰词闭包> -> <修饰词><修饰词闭包> | $
<修饰词> -> describe
<类型> -> type <取地址>
<取地址> -> <星号闭包>
<星号闭包> -> <星号><星号闭包> | $
<星号> -> *
<变量> -> <标志符><数组下标>
<标志符> -> id
<数组下标> -> [<因式>] | $
<因式> -> (<表达式>) | <变量> | <数字>
<数字> -> digit
<表达式> -> <因子><项>
<因子> -> <因式><因式递归>
<因式递归> -> * <因式><因式递归> | / <因式><因式递归> | $
<项> -> + <因子><项> | - <因子><项> | $
<参数声明> -> <声明><声明闭包> | $
<声明> -> <修饰词闭包><类型><变量><赋初值>
<赋初值> -> = <右值> | $
<右值> -> <表达式> | { <多个数据> }
<多个数据> -> <数字><数字闭包>
<数字闭包> -> , <数字><数字闭包> | $
<声明闭包> -> , <声明><声明闭包> | $
<函数块> -> <声明语句闭包><函数块闭包>
<声明语句闭包> -> <声明语句><声明语句闭包> | $
<声明语句> -> <声明> ;
<函数块闭包> -> <赋值函数><函数块闭包> | <for 循环><函数块闭包> | <条件语句><函数块闭包> | <函数返回><函数块闭包> | $
<赋值函数> -> <变量><赋值或函数调用>
<赋值或函数调用> -> = <右值> ; | (<参数列表>) ;
<参数列表> -> <参数><参数闭包>
<参数闭包> -> , <参数><参数闭包> | $
<参数> -> <标志符> | <数字> | <字符串>
<字符串> -> string
<for 循环> -> for (<赋值函数><逻辑表达式> ; <后缀表达式>) { <函数块> }
<逻辑表达式> -> <表达式><逻辑运算符><表达式>
<逻辑运算符> -> < | > | == | !=
<后缀表达式> -> <变量><后缀运算符>
<后缀运算符> -> ++ | --

<条件语句> -> if (<逻辑表达式>) { <函数块> } <否则语句>
<否则语句> -> else { <函数块> } | $
<函数返回> -> return <因式> ;

输入实验 1 中的 C 语言程序 token 流，判断源程序是否符合给定 C 语言子集语法

1、系统名称

LL1 语法分析器

2、系统定义

本系统是一个通用的语法分析器。系统通过读取文法规则文件，并根据规则构造分析表，然后根据分析表对输入的 token 流进行语法分析。

本系统的外部参与者主要有编程人员，其主要使用 LL1 语法分析器进行编译器或解释器的开发，以便分析和处理源代码；系统管理员：负责维护 LL1 语法分析器的运行环境和配置。

本系统的核心功能为对输入的 token 流进行语法分析。主要功能有文法读取功能，允许用户导入文法；Token 流读取功能，允许用户导入待分析的 Token 流；First 集计算功能，根据给定文法自动计算非终结符的 First 集；Follow 集计算功能，根据给定文法自动计算非终结符的 Follow 集；LL1 分析表构造功能，基于计算得到的 First 集和 Follow 集，自动构造 LL1 分析表；语法分析功能，利用构建好的 LL1 分析表，对输入的 Token 流进行语法分析，检测语法错误。

本系统具有多平台支持，要求要求使用支持 C11 标准的 C++编译器和相关库，以确保系统的可靠性和跨平台性。

本系统的利益相关方主要有编程人员，直接受益于 LL1 语法分析器，通过它可以更方便地开发编译器或解释器，提高代码分析的准确性和效率；系统管理员，确保 LL1 语法分析器的稳定运行，负责系统的部署、更新和维护；软件开发团队，参与 LL1 语法分析器的开发，通过协作确保系统的功能完备、性能良好。

总体业务功能与流程包括，文法和 Token 流输入，用户通过导入文法和待分析的 Token 流；First 集和 Follow 集计算，系统根据输入的文法自动计算非终结符的 First 集和 Follow 集；LL1 分析表构造，利用计算得到的 First 集和 Follow 集，系统自动生成 LL1 分析表；语法分析，用户触发语法分析操作，系统根据 LL1 分析表对输入的 Token 流进行语法分析，检测语法错误；结果输出功能，将语法分析结果输出给用户，包括语法正确与否的信息，如果有错误，还需提供错误信息的定位和提示。

3、需求分析

本系统主要业务需求如下：

1. 功能需求：

a) 读取并解析文法文件

b) 求文法的 first 集和 follow 集

c) 求文法的 select 集并构造 LL1 分析表

d) 针对分析表对输入的 token 流进行语法分析

需要输出语法分析表
需要展示语法分析的过程
需要输出语法错误的位置

2. 性能需求：

a) 本系统在性能方面没有明确的特定要求

3. 接口需求：

a) 语法分析器应具备能够读入文法规则的功能，以便灵活地定义文法规则
b) 提供接口计算文法的 First 集和 Follow 集，用于后续 LL1 分析表的构建
c) 构建 LL1 分析表，提供文法的产生式、非终结符、终结符和对应的 Select 集等信息
d) 对输入的 Token 流进行语法分析，输出分析过程，若有错误需要报告语法错误和位置
e) 提供接口输出构建好的 LL1 分析表，以便用户查看

4、系统设计

针对各需求点，描述是如何实现的。需列出详细的设计要点。

4.1 体系结构设计

本系统采用 C11 标准 C++语言实现，根据功能划分为四层，分别为：

1. 文法解析层

- **职责：**负责解析输入的文法规则，提取产生式、非终结符、终结符等信息，并进行基本的语法检查
- **接口：**提供函数或接口以加载、解析、处理文法规则

2. First、Follow 层

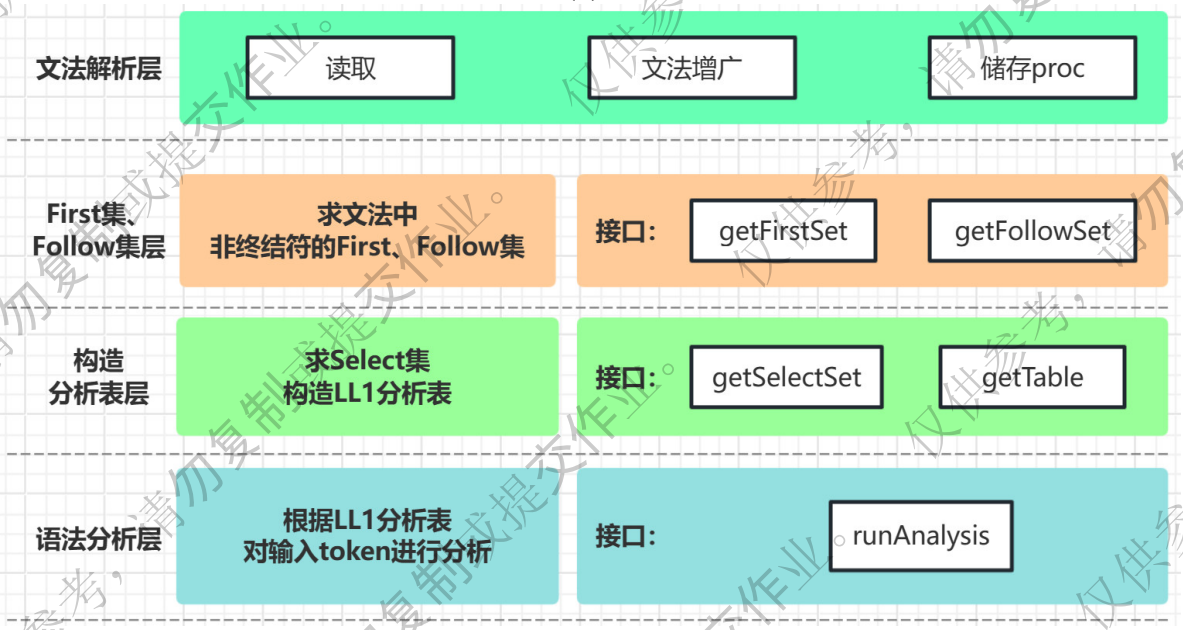
- **职责：**计算文法规则中每个非终结符的 First 集合和 Follow 集合
- **接口：** 提供函数或接口以计算和获取 First 和 Follow 集合

3. 构造分析表层

- **职责：** 基于计算得到的 First 集合和 Follow 集合，构造 LL(1)分析表
- **接口：** 提供函数或接口以计算 Select 集、生成 LL(1)分析表

4. 语法分析层

- **职责**：利用 LL(1)分析表进行语法分析，根据输入的符号串进行语法分析，若出现错误则报告语法错误
- **接口**：提供函数或接口以执行 LL(1)语法分析



4.2 类设计

本系统所有层封装在类 Grammer 中

1. 文法解析层

- **问题描述**

本层问题是如何以一种合适的方式读取文法，并且需要在预处理后存储下来。

- **整体的解决思路、流程或算法**

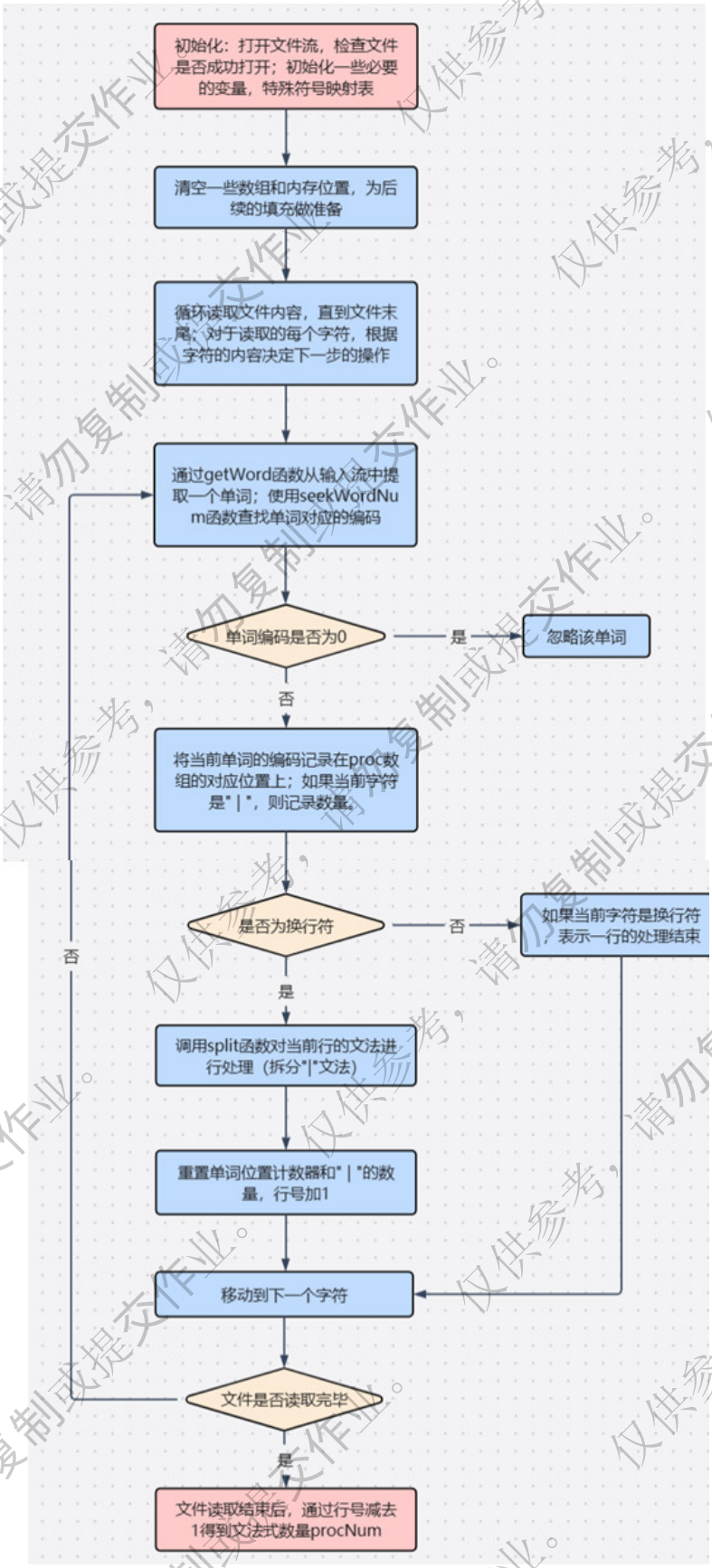
整体的解决思路：

在类 Grammer 构造函数中，需要给不同的文法终结符编号，然后每一条产生式都以编号后的形式储存下来：首先打开给定路径的文法文件，并初始化相关变量和映射表。随后，通过循环读取文件内容，逐行解析文法规则。在每一行中，通过 getWord 函数获取单词并映射为对应的编码，将编码记录到 proc 数组中，同时处理" | "进行文法增广。最终，计算产生式的数量，完成文法的初始化。其中，使用了各种数组和映射表来存储文法的各类信息，以及相应的处理逻辑。

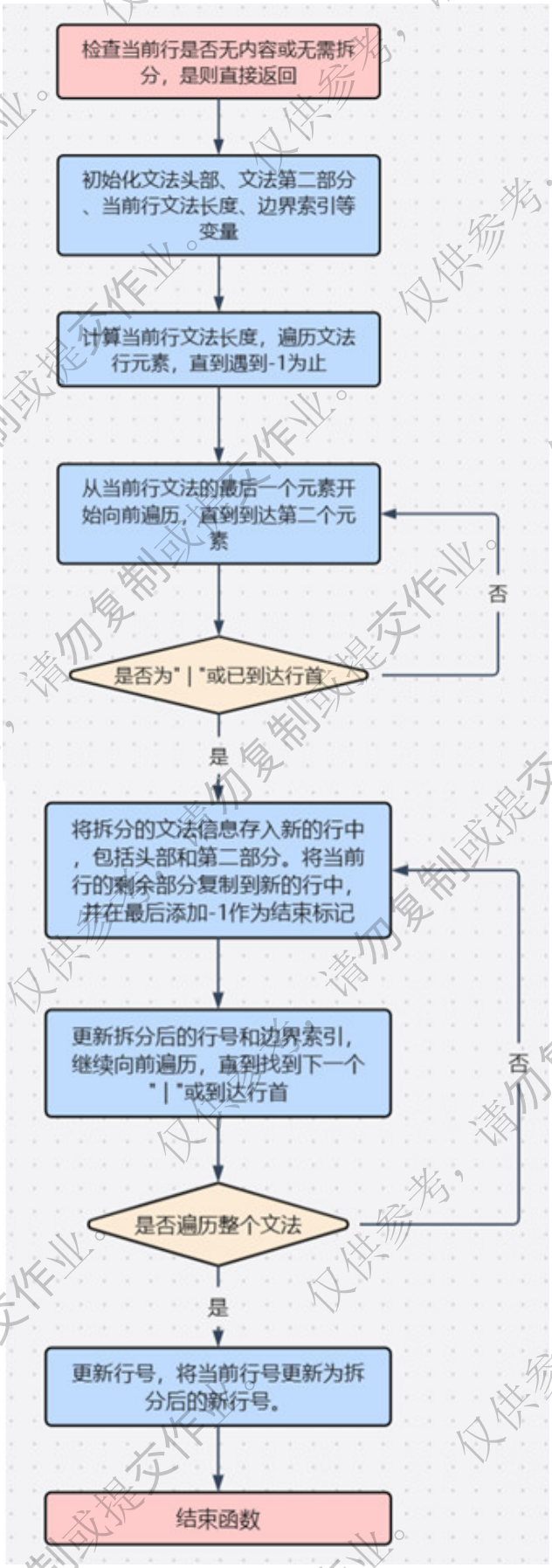
函数 seekWordNum()用于获取文法产生式中每一个符号的编码，split()函数用于进行文法增广操作。

算法的流程图：

类 Grammer 构造函数



文法增广函数 split()



- **本层提供接口**

seekWordNum()	获取产生式符号对应编码
split()	文法增广
proc	编码后的产生式
nonTerMap	非终结符映射表
terMap	终结符映射表
specialMap	文法中的特殊符号映射表，包括-> | $

2. First、Follow 集层

- **问题描述**

First 集：对于文法中的每个非终结符号，它的 First 集包含可以作为该非终结符号推导出的字符串的开头终结符号集合。具体来说，对于非终结符号 A，它的 First(A)包含了所有由 A 推导出的字符串的开头终结符号集合。求解 First 集的过程涉及到对文法的推导规则进行分析，直到找到所有可能的开始终结符号。

Follow 集：对于文法中的每个非终结符号，它的 Follow 集包含了在句型中该非终结符号右边可能出现的终结符号集合。具体来说，对于非终结符号 A，它的 Follow(A)包含了所有可能在句型中跟随 A 出现的终结符号集合。求解 Follow 集通常需要对文法进行反复分析，以确定哪些终结符号可以跟随着特定的非终结符号。

- **整体的解决思路、流程或算法**

整体的解决思路：

First 集求解的过程主要分为以下六个步骤：

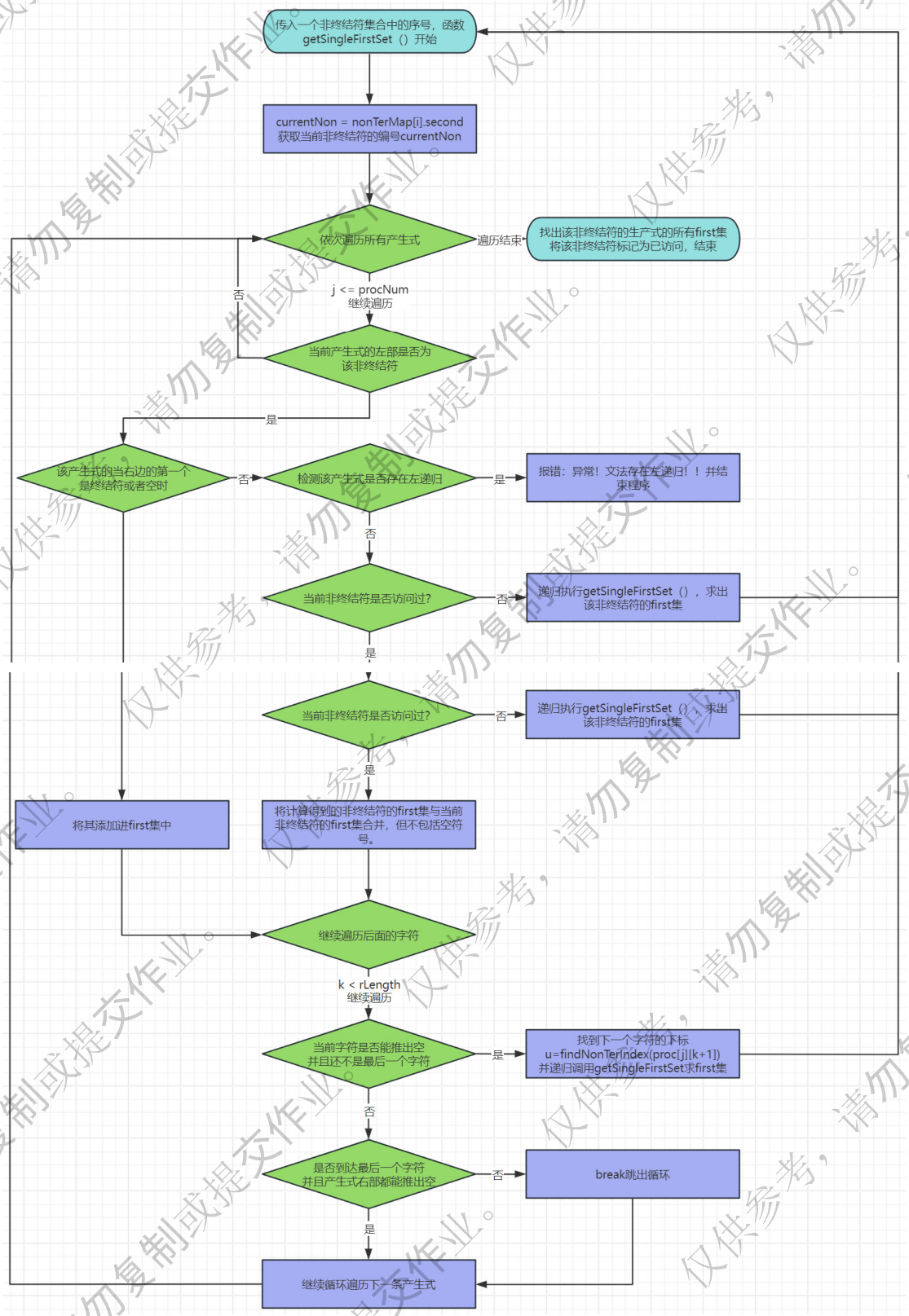
1. 初始化和迭代遍历产生式：遍历文法的所有产生式，检查每个产生式是否以当前非终结符开始。
2. 处理终结符和空串：如果产生式右侧的第一个符号是终结符或者空串，将其加入到当前非终结符的 First 集中。
3. 处理非终结符：如果产生式右侧的第一个符号是另一个非终结符，通过递归调用计算当前非终结符的右侧第一个非终结符的 First 集，然后将这个非终结符的 First 集合并入当前非终结符的 First 集中，但不包括空串。
4. 终结符能推出空时的后续遍历：循环处理产生式右侧的剩余字符，如果后续字符能推导出空串，则继续处理下一个字符。如果产生式右侧字符序列到达末尾仍能推导出空串，将空串加入当前非终结符的 First 集中。
5. 标记访问状态，对程序进行优化： 标记已经处理过的非终结符，避免重复计算。
6. 递归遍历：在处理非终结符时，如果遇到新的非终结符，递归调用来计算其 First 集。

Follow 集部分：

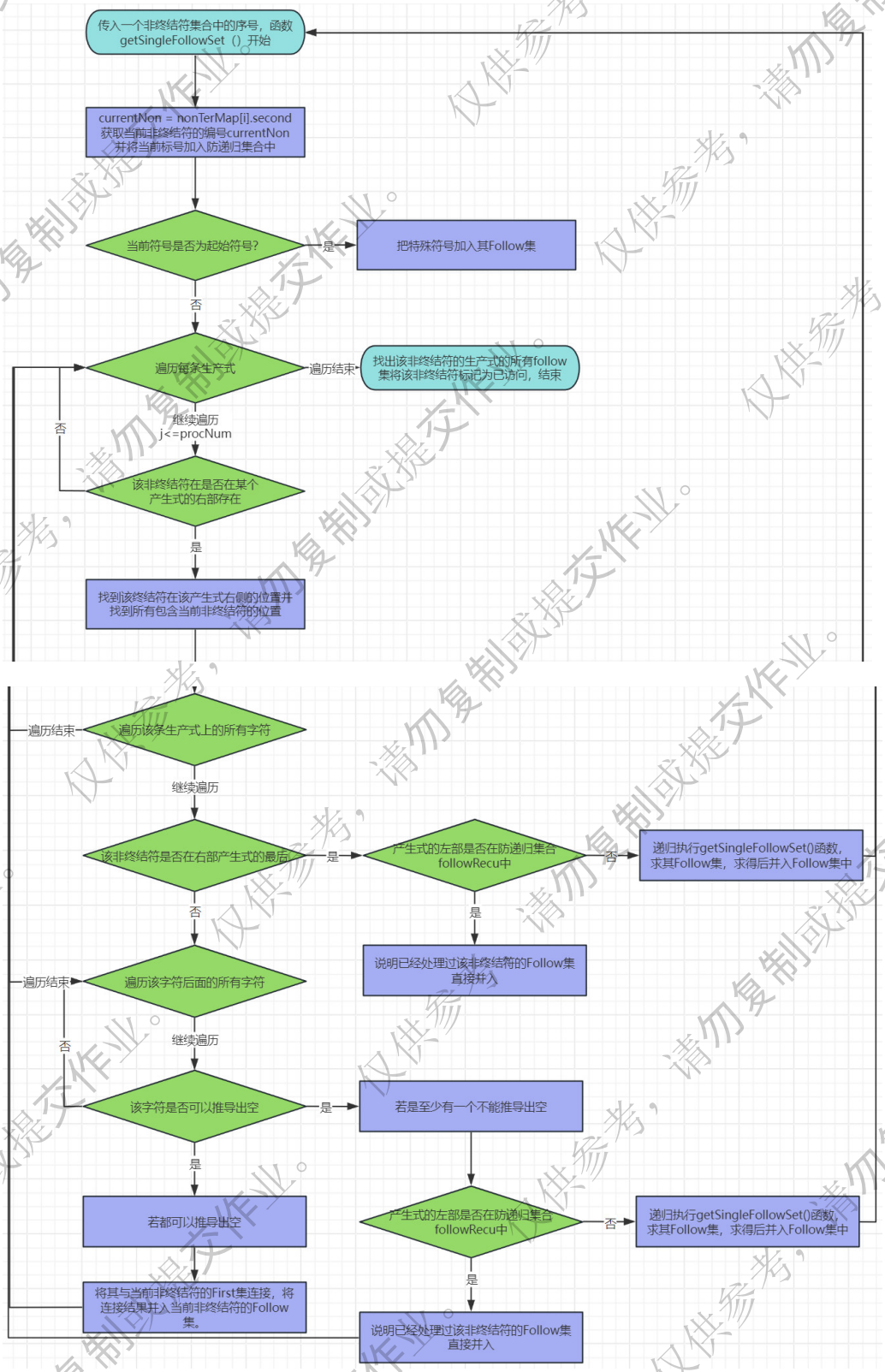
1. 初始化和递归防护：
   a) 获取当前非终结符的编号 currentNon。
   b) 将 currentNon 加入 followRecu 集合中，防止在递归过程中重复处理同一个非终结符。
2. 处理起始符号：
   a) 如果当前非终结符是起始符号，将特殊符号（通常用于表示输入串的结束，如$）加入到它的 Follow 集中。
3. 遍历产生式：
   a) 对于文法中的每一个产生式，检查当前非终结符是否出现在产生式的右侧。
4. 在产生式右侧找到非终结符：
   a) 获取产生式左侧的非终结符编号 leftNum。
   b) 寻找当前非终结符在产生式右侧所有出现的位置，存储在 kArray 中。
5. 对 kArray 中的每个位置进行遍历，对每个位置的非终结符执行以下操作：
   情况 1：非终结符位于产生式右侧末尾
   a) 如果当前非终结符是产生式右侧的最后一个符号，将左侧非终结符的 Follow 集加入到当前非终结符的 Follow 集中。
   b) 如果左侧非终结符的 Follow 集已在 followRecu 中，说明已经处理过，直接并入。
   c) 否则，递归地计算左侧非终结符的 Follow 集，并将其加入当前非终结符的 Follow 集中。
   情况 2：非终结符后面还有符号
   a) 检查当前非终结符之后的所有符号是否都能推导出空（即是否都是可空的）。
   b) 如果所有符号都可空，将左侧非终结符的 Follow 集并入当前非终结符的 Follow 集。
   c) 同时，计算从当前位置到产生式末尾符号序列的 First 集（不包括空）并将其并入当前非终结符的 Follow 集。
6. 在 followVisit 数组中将当前非终结符标记为已处理。

算法的流程图：

求 First 集的函数——getSingleFirstSet ()：

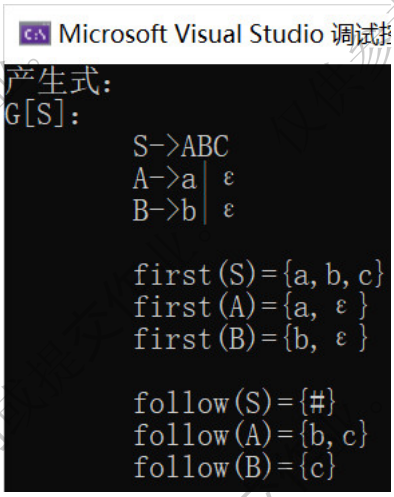


求 Follow 集的函数——getSingleFollowSet ()：

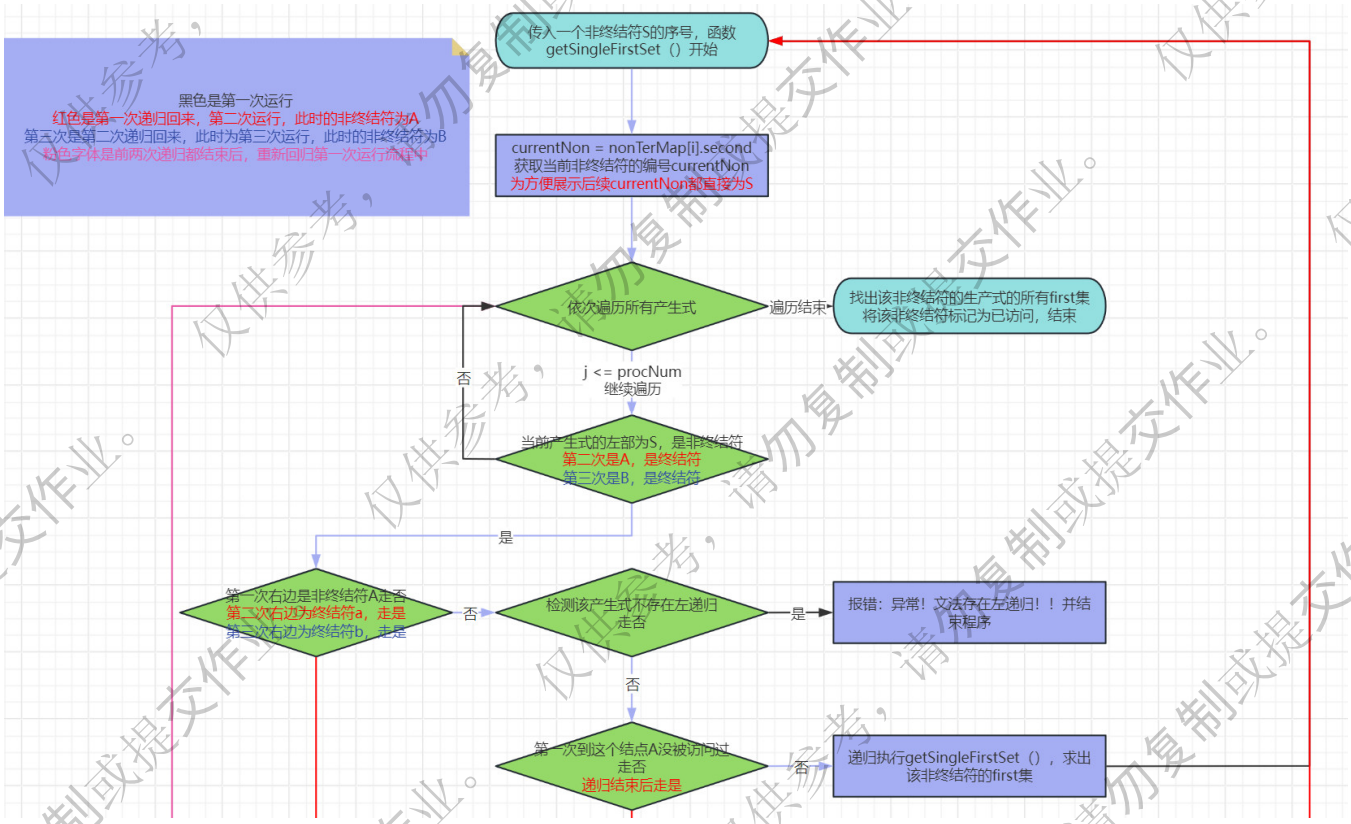


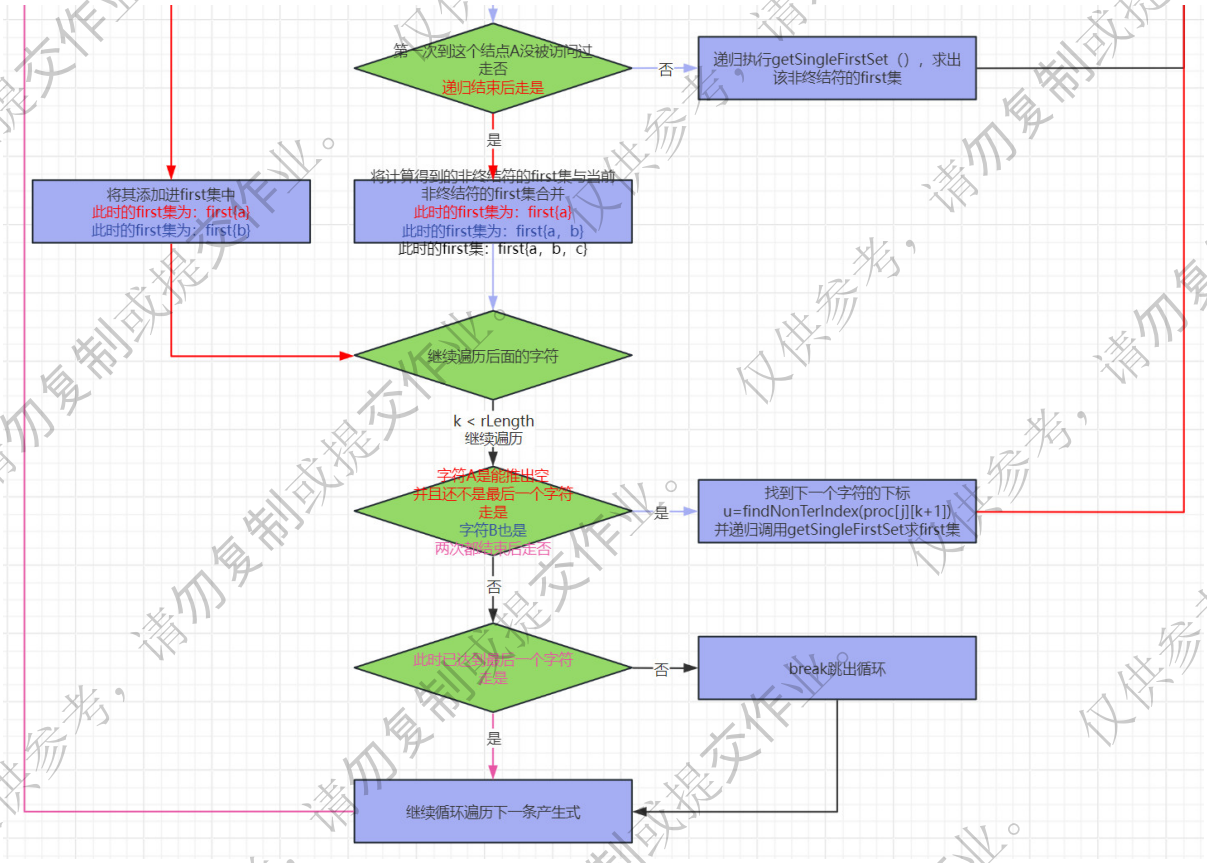
因本层比较关键，进行了单元测试如下：

- 本层基本测试



因为本次程序设计中包涵了众多的递归调用，所以我以求 S 的 first 集为例，过了一遍流程图，以下是展示：





➢ **本层提供接口**

getFirstSet ()	求 first 集
getFollowSet ()	求 follow 集
first	first 存储
follow	follow 存储

3. 构造分析表层

➢ **问题描述**

本层的主要问题是计算文法中每个产生式的 Select 集、然后根据 selcet 集构造分析表。Select 集是用于确定预测分析表中每个产生式的输入符号的集合。对于每个产生式，需要根据右部的符号集合计算 Select 集。如果右部能够推导出空串，还需要考虑左部的 Follow 集。最终，将计算得到的 Select 集合存储在 select_数组中。

➢ **整体的解决思路、流程或算法**

编写两个函数 getSelectSet、getTable 函数：

其中，getSelectSet 函数遍历文法的每个产生式，获取产生式左侧的非终结符。根据产生式右侧的符号集合计算 Select 集。如果右侧能推导出空串，还考虑左侧非终结符的 Follow 集。然后，将计算得到的 Select 集存储在 select_数组中。

getTable 函数遍历所有产生式，找到产生式左侧非终结符在非终结符映射表中的位置。遍历 Select 集合，找到每个终结符在终结符映射表中的位置。将对应

位置的产生式右侧存储到 LL(1)预测分析表的相应位置。输出构建好的预测分析表到文件中。

这两个函数协同工作，先计算 Select 集，再根据 Select 集构建 LL(1)预测分析表。Select 集决定了产生式在预测分析表中的位置以及对应的输入符号。getTable 函数则将这些信息写入预测分析表。

- **本层提供接口**

getSelectSet ()	求 select 集
getTable ()	求分析表
select_	select 存储
analytable	分析表存储

4. 语法分析层

- **问题描述**

本层的主要问题是如何读取词法分析器输出的 token 流，如何利用 LL1 分析表来进行语法分析，并且输出语法分析的过程。

- **整体的解决思路、流程或算法**

针对**如何读取词法分析器输出的 token 流的问题**，我构建了一个类 Lexer，Lexer 初始化时将打开实验 1 中输出的 token 文件然后将 token 读取为一个 token 流链表。本系统的主类 Grammar 将公有继承 Lexer，然后就可以解决如何读取词法分析器输出的 token 流问题，下面详细介绍类 Lexer：

- **构造函数 Lexer::Lexer():**

该构造函数用于初始化 Lexer 类的实例。通过读取文件中的 keyMapping、opMapping 和 limMapping 文件，将关键字、运算符和界限符的映射信息加载到对应的 keyMap、opMap 和 limMap 容器中。

- **成员函数 Lexer::scanner(const char filename):**

该函数用于从文件中读取内容并构造链表。

打开指定文件，逐行读取文件内容，然后调用 createNode 函数创建一个节点，并将该节点添加到链表中。

- **成员函数 Lexer::createNode(const string &line):**

该函数用于从文件内容的一行创建一个新节点。使用 stringstream 解析每一列的内容，将类型、词素、行号、列号、长度等信息设置到新节点中。

再者，针对**如何**利用 LL1 分析表来进行语法分析问题，我构建了两个栈来进行语法分析，详细如下：

- **初始化栈：**

通过 initStack 函数初始化两个栈 s1（符号栈）和 s2（输入栈）。

- **读取 Token 流：**

遍历链表，将词法分析器输出的 Token 流读取到数组 reserve 中。

- **反向入栈：**

将 reserve 数组中的 Token 反向入栈 s2 中，同时在栈底添加特殊符号 # 和文法的起始符号。

- **LL(1) 分析：**

进入主循环，不断进行 LL(1) 分析。

每一步中，显示符号栈和输入栈的内容，并取出符号栈和输入栈的栈顶元素。

如果两者相等，表示匹配成功，同时弹出符号栈和输入栈的栈顶元素。

否则，根据 LL(1) 分析表查找相应的推导式，并将推导式右侧的符号反向入栈 s1。

- **错误处理：**

如果在 LL(1) 分析表中找不到对应的推导式，说明出现了错误。程序输出错误信息，并终止分析过程。

- **显示分析结果：**

在每一步 LL(1) 分析后，显示符号栈、输入栈以及匹配或推导的信息。

- **成功或失败判断：**

如果符号栈和输入栈同时剩余 #，则分析成功，输出成功信息。

如果符号栈和输入栈不匹配，或者 LL(1) 分析表中无法找到对应的推导式，输出失败信息。

通过以上步骤，该语法分析器使用 LL(1) 分析表对输入的 Token 流进行自顶向下的语法分析。在每一步分析中，根据当前符号栈的栈顶元素和输入栈的栈顶元素，查找 LL(1) 分析表，选择相应的推导式进行匹配或推导。这个过程反复执行，直到成功完成分析或者发现错误。最终，程序输出相应的分析结果。

- **本层提供接口**

runAnalysis ()　　语法分析

4.3 构件设计

本系统无构件设计。

5、测试

5.1 测试需求分析

需求点 1：文法文件读取与解析

- 测试点 1： 正常情况下的文法文件读取与解析
  - 输入: 包含正确文法的文件路径
  - 预期输出: 系统成功读取并解析文法文件，将内容正确地储存起来。
- 测试点 2：错误情况下的文法文件读取与解析
  - 输入: 包含错误文法的文件路径
  - 预期输出: 系统能够正确识别错误，输出相应的错误信息，并不进行储存

需求点 2： 能够正确根据文法求得 first、follow 集

- 测试点 1：正常情况下的 first、follow 集求解
  - 输入: 包含正确文法的文件路径

- 预期输出: 系统能够正确求得文法的 first、follow 集

需求点 3：能够正确求得 select 集、LL(1)分析表

- 测试点 1：正常情况下的 select 集、LL(1)分析表求解
  - 输入: 包含正确文法的文件路径
  - 预期输出: 系统能够正确根据文法的 first 集、follow 集求得 select 集以及 LL(1)分析表

需求点 4：能够正常读取词法分析器输出的 token 流

- 测试点 1：正常情况下的 token 流读取
  - 输入: 包含正确 token 流的文件路径
  - 预期输出: 系统能够成功读取词法分析器输出的 token 流。
- 测试点 2：错误情况下的 token 流读取
  - 输入: 包含正确 token 流的文件路径
  - 预期输出: 系统能够成功读取词法分析器输出的 token 流。

需求点 5：能够分析 token 流是否符合语法要求，若不能输出出错位置

- 测试点 1: 正常情况下的语法分析
  - 输入: 包含正确 token 流的文件路径
  - 预期输出: 系统能够成功进行语法分析，输出相应的分析结果。
- 测试点 2: 错误情况下的语法分析
  - 输入: 包含错误 token 流的文件路径
  - 预期输出: 系统能够正确识别错误，输出相应的错误信息，并不进行语法分析。

5.2 测试设计

5. 2. 1 体系结构

1.文法文件读取与解析模块

职责：负责从文件中读取文法内容并解析，生成相应的数据结构。

接口：提供函数用于文法文件的读取、解析，并返回文法数据结构。

2. 编码模块

职责：为终结符和非终结符唯一编码。

接口：提供函数用于为文法中的每一个终结符或非终结符分配唯一编码。

3. First、Follow 集求解模块

职责：根据给定的文法求得相应的 First、Follow 集。

接口：提供函数用于计算文法的 First、Follow 集。

4. Select 集、LL(1)分析表求解模块

职责：根据 First、Follow 集求得 Select 集以及 LL(1)分析表。

接口：提供函数用于计算文法的 Select 集和 LL(1)分析表。

5. 词法分析器输出的 Token 流读取模块

职责：读取词法分析器输出的 Token 流。

接口：提供函数用于从文件中读取 Token 流。

6. 语法分析模块

职责：利用 LL(1)分析表进行语法分析，检查 Token 流是否符合语法要求。

接口：提供函数用于执行 LL(1)语法分析，输出分析结果或错误信息。

7. 测试模块

职责：执行测试用例，比较实际输出和预期输出。

接口：提供测试用例输入，检查实际输出是否符合预期输出。

5.2.2 测试用例

针对具体的测试需求，在测试体系结构设计的框架下，设计相应的测试数据和测试流程，以检验最终的系统是否具备预期的特性。

测试用例 1：

输入：错误的文法文件路径

预取输出：报错，停止运行程序

测试用例 2：

输入：错误的词法分析器输出文件路径

预取输出：报错，停止运行程序

测试用例 3：

输入：正确的文法、token 路径、无语法错误源程序

```
int main()
{
    int i = 7;
    int j = 666;
    char c[3] = {2, 1, 3, 43};

    for (i = 0; i < 20; i++)
    {
        int TEAM_1 = 32116160100;
        for (j = i + 1; j < 20; j--)
        {
            if (j == 19)
            {
                int fwm = j;
            }
        }
        if (j > 19)
        {
            for (j = i + 1; j < 20; j--)
            {
                int wh = j;
                int cqz = j + 1;
            }
        }
    }
    return -1;
}
```

预取输出：语法分析结果

测试用例 4：

输入：正确的文法、token 路径、有语法错误源程序（无;）

```
int main()
{
    int i = 7;
    int j = 666;
    char c[3] = {2, 1, 3, 43}
                              无;
    for (i = 0; i < 20; i++)
    {
        int TEAM_1 = 32116160100;
        for (j = i + 1; j < 20; j--)
        {
            if (j == 19)
            {
                int fwm = j;
            }
        }
        if (j > 19)
        {
            for (j = i + 1; j < 20; j--)
            {
                int wh = j;
                int cqz = j + 1;
            }
        }
    }
    return -1;
}
```

预取输出：语法分析结果，语法错误位置

测试用例 5：

输入：正确的文法、token 路径、有语法错误源程序（括号不闭合）

```
int main()
{
    int i = 7;
    int j = 666;
    char c[3] = {2, 1, 3, 43};
    for (i = 0; i < 20; i++
    {
        int TEAM_1 = 32116160100;
        for (j = i + 1; j < 20; j--)
        {
            if (j == 19)
            {
                int fwm = j;
            }
        }
        if (j > 19)
        {
            for (j = i + 1; j < 20; j--)
            {
                int wh = j;
                int cqz = j + 1;
            }
        }
    }
    return -1;
}
```

括号不闭合

预取输出：语法分析结果，语法错误位置

测试用例 6：

输入：正确的文法、token 路径、有语法错误源程序（缺少=）

```
int main()
{
    int i = 7;
    int j = 666;
    char c[3] = {2, 1, 3, 43};

    for (i = 0; i < 20; i++)
    {
        int TEAM_1  32116160100;
        for (j = i + 1; j < 20; j--)
        {
            if (j == 19)
            {
                int fwm = j;
            }
        }
        if (j > 19)
        {
            for (j = i + 1; j < 20; j--)
            {
                int wh = j;
                int cqz = j + 1;
            }
        }
    }
    return -1;
}
```

没有=

预取输出：语法分析结果，语法错误位置

5.3 测试执行情况记录及测试报告分析

在系统/模块完成后，执行测试代码，记录、统计测试结果，并对测试结果进行统计、分析，识别出相应的软件缺陷及其风险。

测试用例 1：不存在文法文件情况

测试结果：

```
==================== 语法分析 ====================
文法文件打开失败！请检查路径
```

测试结果：成功！符合预期

测试用例 2：不存在 token 文件情况

测试结果：

```
==================== 语法分析 ====================
错误：无法打开文件
```

测试结果：成功！符合预期

测试用例 3：无语法错误的 token 流

测试结果：

```
==================== 词法分析 ====================
1.Lex规则文本解析
2.正规式转NFA
3.NFA转DFA
4.生成词法分析器代码
已生成词法分析器./Lexer/myScannerCode.c

词法分析完毕，请查看token文件

==================== 语法分析 ====================
语法分析完毕，请查看 语法分析结果.gr 文件
(base) fwm@YingFeng:~/project/C_Lexer_Grammer$
```

输出的 LL1 分析表：

```
**********************************预测分析表*******************************
analytable[<函数定义>][(] =
analytable[<函数定义>][)] =
analytable[<函数定义>][{] =
analytable[<函数定义>][}] =
analytable[<函数定义>][describe] = <函数定义>->(修饰词闭包><类型><变量>(<参数声明>){<函数块>}
analytable[<函数定义>][type] = <函数定义>->(修饰词闭包><类型><变量>(<参数声明>){<函数块>}
analytable[<函数定义>][*] =
analytable[<函数定义>][id] =
analytable[<函数定义>][[] =
analytable[<函数定义>][]] =
analytable[<函数定义>][digit] =
analytable[<函数定义>][/] =
analytable[<函数定义>][+] =
analytable[<函数定义>][-] =
analytable[<函数定义>][=] =
analytable[<函数定义>][,] =
analytable[<函数定义>][;] =
analytable[<函数定义>][string] =
analytable[<函数定义>][for] =
analytable[<函数定义>][<] =
analytable[<函数定义>][>] =
analytable[<函数定义>][==] =
analytable[<函数定义>][!=] =
analytable[<函数定义>][++] =
analytable[<函数定义>][--] =
analytable[<函数定义>][if] =
analytable[<函数定义>][else] =
analytable[<函数定义>][return] =


analytable[<修饰词闭包>][(] =
analytable[<修饰词闭包>][)] =
analytable[<修饰词闭包>][{] =
analytable[<修饰词闭包>][}] =
analytable[<修饰词闭包>][describe] = <修饰词闭包>->(修饰词><修饰词闭包>
analytable[<修饰词闭包>][type] = <修饰词闭包>->$
analytable[<修饰词闭包>][*] =
analytable[<修饰词闭包>][id] =
analytable[<修饰词闭包>][[] =
analytable[<修饰词闭包>][]] =
analytable[<修饰词闭包>][digit] =
analytable[<修饰词闭包>][/] =
analytable[<修饰词闭包>][+] =
analytable[<修饰词闭包>][-] =
analytable[<修饰词闭包>][=] =
analytable[<修饰词闭包>][,] =
analytable[<修饰词闭包>][;] =
analytable[<修饰词闭包>][string] =
analytable[<修饰词闭包>][for] =
```

输出的 LL1 分析过程 1：

```
符号栈:<函数定义> #
输入栈:type id ( ) { type id = digit ; type id = digit ; type id [ digit ] = { digit , digit , digit , digit } ; for ( id = digit ; id < digit ; id ++ ) { type id = digit ; for ( id = id + digit ; id < digit ; id -- ) { if ( id == digit ) { type id = id ; } } if ( id > digit ) { for ( id = id + digit ; id < digit ; id -- ) { type id = id ; type id = id + digit ; } } return digit ; } #
推出式:<函数定义>->(修饰词闭包><类型><变量>(<参数声明>){<函数块>}

符号栈:(修饰词闭包><类型><变量>(<参数声明>){<函数块>}#
输入栈:type id ( ) { type id = digit ; type id = digit ; type id [ digit ] = { digit , digit , digit , digit } ; for ( id = digit ; id < digit ; id ++ ) { type id = digit ; for ( id = id + digit ; id < digit ; id -- ) { if ( id == digit ) { type id = id ; } } if ( id > digit ) { for ( id = id + digit ; id < digit ; id -- ) { type id = id ; type id = id + digit ; } } } return digit ; } #
推出式:<修饰词闭包>->$

符号栈:<类型><变量>(<参数声明>){<函数块>}#
输入栈:type id ( ) { type id = digit ; type id = digit ; type id [ digit ] = { digit , digit , digit , digit } ; for ( id = digit ; id < digit ; id ++ ) { type id = digit ; for ( id = id + digit ; id < digit ; id -- ) { if ( id == digit ) { type id = id ; } } if ( id > digit ) { for ( id = id + digit ; id < digit ; id -- ) { type id = id ; type id = id + digit ; } } } return digit ; } #
推出式:<类型>->type<取地址>

符号栈:type <取地址><变量>(<参数声明>){<函数块>}#
输入栈:type id ( ) { type id = digit ; type id = digit ; type id [ digit ] = { digit , digit , digit , digit } ; for ( id = digit ; id < digit ; id ++ ) { type id = digit ; for ( id = id + digit ; id < digit ; id -- ) { if ( id == digit ) { type id = id ; } } if ( id > digit ) { for ( id = id + digit ; id < digit ; id -- ) { type id = id ; type id = id + digit ; } } } return digit ; } #
匹配!

符号栈:<取地址><变量>(<参数声明>){<函数块>}#
输入栈:id ( ) { type id = digit ; type id = digit ; type id [ digit ] = { digit , digit , digit , digit } ; for ( id = digit ; id < digit ; id ++ ) { type id = digit ; for ( id = id + digit ; id < digit ; id -- ) { if ( id == digit ) { type id = id ; } } if ( id > digit ) { for ( id = id + digit ; id < digit ; id -- ) { type id = id ; type id = id + digit ; } } } return digit ; } #
推出式:<取地址>->(星号闭包>

符号栈:(星号闭包><变量>(<参数声明>){<函数块>}#
输入栈:id ( ) { type id = digit ; type id = digit ; type id [ digit ] = { digit , digit , digit , digit } ; for ( id = digit ; id < digit ; id ++ ) { type id = digit ; for ( id = id + digit ; id < digit ; id -- ) { if ( id == digit ) { type id = id ; } } if ( id > digit ) { for ( id = id + digit ; id < digit ; id -- ) { type id = id ; type id = id + digit ; } } } return digit ; } #
推出式:<星号闭包>->$

符号栈:<变量>(<参数声明>){<函数块>}#
输入栈:id ( ) { type id = digit ; type id = digit ; type id [ digit ] = { digit , digit , digit , digit } ; for ( id = digit ; id < digit ; id ++ ) { type id = digit ; for ( id = id + digit ; id < digit ; id -- ) { if ( id == digit ) { type id = id ; } } if ( id > digit ) { for ( id = id + digit ; id < digit ; id -- ) { type id = id ; type id = id + digit ; } } } return digit ; } #
推出式:<变量>->(标志符><数组下标>

符号栈:(标志符><数组下标>(<参数声明>){<函数块>}#
输入栈:id ( ) { type id = digit ; type id = digit ; type id [ digit ] = { digit , digit , digit , digit } ; for ( id = digit ; id < digit ; id ++ ) { type id = digit ; for ( id = id + digit ; id < digit ; id -- ) { if ( id == digit ) { type id = id ; } } if ( id > digit ) { for ( id = id + digit ; id < digit ; id -- ) { type id = id ; type id = id + digit ; } } } return digit ; } #
推出式:<标志符>->id

符号栈:id <数组下标>(<参数声明>){<函数块>}#
输入栈:id ( ) { type id = digit ; type id = digit ; type id [ digit ] = { digit , digit , digit , digit } ; for ( id = digit ; id < digit ; id ++ ) { type id = digit ; for ( id = id + digit ; id < digit ; id -- ) { if ( id == digit ) { type id = id ; } } if ( id > digit ) { for ( id = id + digit ; id < digit ; id -- ) { type id = id ; type id = id + digit ; } } } return digit ; } #
匹配!

符号栈:<数组下标>(<参数声明>){<函数块>}#
输入栈:( ) { type id = digit ; type id = digit ; type id [ digit ] = { digit , digit , digit , digit } ; for ( id = digit ; id < digit ; id ++ ) { type id = digit ; for ( id = id + digit ; id < digit ; id -- ) { if ( id == digit ) { type id = id ; } } if ( id > digit ) { for ( id = id + digit ; id < digit ; id -- ) { type id = id ; type id = id + digit ; } } } return digit ; } #
推出式:<数组下标>->$
```

输出的 LL1 分析过程 2：

```
符号栈:<函数块闭包> } <函数块闭包> } #
输入栈:} return digit ; } #
推出式:<函数块闭包>->$

符号栈:} <函数块闭包> } #
输入栈:} return digit ; } #
匹配!

符号栈:<函数块闭包> } #
输入栈:return digit ; } #
推出式:<函数块闭包>-><函数返回><函数块闭包>

符号栈:<函数返回> <函数块闭包> } #
输入栈:return digit ; } #
推出式:<函数返回>->return<因式>;

符号栈:return <因式> ; <函数块闭包> } #
输入栈:return digit ; } #
匹配!

符号栈:<因式> ; <函数块闭包> } #
输入栈:digit ; } #
推出式:<因式>-><数字>

符号栈:<数字> ; <函数块闭包> } #
输入栈:digit ; } #
推出式:<数字>->digit

符号栈:digit ; <函数块闭包> } #
输入栈:digit ; } #
匹配!

符号栈:; <函数块闭包> } #
输入栈:; } #
匹配!

符号栈:<函数块闭包> } #
输入栈:} #
推出式:<函数块闭包>->$

符号栈:} #
输入栈:} #
匹配!

符号栈:#
输入栈:#
成功!
```

LL1分析结束

测试结果：成功！符合预期

测试用例 4：有语法错误的 token 流（没有；）

测试结果：

```
==================== 词法分析 ====================
1.Lex规则文本解析
2.正规式转NFA
3.NFA转DFA
4.生成词法分析器代码
已生成词法分析器./Lexer/myScannerCode.c

词法分析完毕，请查看token文件

==================== 语法分析 ====================


============ 出现错误！ ============
===== 错误出现在下列两个token间 =====
token     line      colunm
}         5         28
for       7         4
语法分析完毕，请查看 语法分析结果.gr 文件
```

可以看到程序可以展示到出现语法错误的位置，从语法分析过程文件中也可以看到：

语法分析结果.gr

文件 编辑 查看

```
( id = id + digit ; id < digit ; id -- ) { type id = id ; type id = id + digit ; } } } return digi
推出式:<数字闭包>->,<数字><数字闭包>

符号栈:, <数字> <数字闭包> } ; <声明语句闭包> <函数块闭包> } #
输入栈:, digit } for ( id = digit ; id < digit ; id ++ ) { type id = digit ; for ( id = id + digit
( id = id + digit ; id < digit ; id -- ) { type id = id ; type id = id + digit ; } } } return digi
匹配!

符号栈:<数字> <数字闭包> } ; <声明语句闭包> <函数块闭包> } #
输入栈:digit } for ( id = digit ; id < digit ; id ++ ) { type id = digit ; for ( id = id + digit ;
( id = id + digit ; id < digit ; id -- ) { type id = id ; type id = id + digit ; } } } return digi
推出式:<数字>->digit

符号栈:digit <数字闭包> } ; <声明语句闭包> <函数块闭包> } #
输入栈:digit } for ( id = digit ; id < digit ; id ++ ) { type id = digit ; for ( id = id + digit ;
( id = id + digit ; id < digit ; id -- ) { type id = id ; type id = id + digit ; } } } return digi
匹配!

符号栈:<数字闭包> } ; <声明语句闭包> <函数块闭包> } #
输入栈:} for ( id = digit ; id < digit ; id ++ ) { type id = digit ; for ( id = id + digit ; id < d
id + digit ; id < digit ; id -- ) { type id = id ; type id = id + digit ; } } } return digit ; } #
推出式:<数字闭包>->$

符号栈:} ; <声明语句闭包> <函数块闭包> } #
输入栈:} for ( id = digit ; id < digit ; id ++ ) { type id = digit ; for ( id = id + digit ; id < d
id + digit ; id < digit ; id -- ) { type id = id ; type id = id + digit ; } } } return digit ; } #
匹配!

符号栈:; <声明语句闭包> <函数块闭包> } #
输入栈:for ( id = digit ; id < digit ; id ++ ) { type id = digit ; for ( id = id + digit ; id < dig
+ digit ; id < digit ; id -- ) { type id = id ; type id = id + digit ; } } } return digit ; } #


============ 出现错误！ ============
===== 错误出现在下列两个token间 =====
token     line      column
}         5         28
for       7         4
```

语法分析过程

测试结果：成功！符合预期

测试用例 5：有语法错误的 token 流（括号不闭合）

测试结果：

可以看到程序可以展示到出现语法错误的位置

测试结果：成功！符合预期

测试用例 6：有语法错误的 token 流（没有=）

测试结果：

测试结果：成功！符合预期

6、实验中遇到的困难及解决方法

本次实验中我们小组遇到了许多难题，好在在不断的学习、检索下都成功解决了难题。

在文法解析层，首当其冲的难题是需要设计合适的符合 LL1 规范的文法，我们通过查询资料找到了关于 C 语言子集的文法，并在此基础上设计了我们的文法。其次，文法需要进行储存，若直接储存字符串会带来将程序变得复杂，我们设计了一套编码方法，将文法编码后存储。最后，还需要对文法进行增广，以消除|。

在 first、follow 层遇到最大的问题首当其冲的递归部分，因为需要反复推导文法进一步求解下一个非终结符所对应的 first 集或 follow 集，就不可避免的使用大量的递归，而递归就导致了难以正确的计算 first 集与 follow 集，并且出现问题后排查起来非常困难。

第二困难的就是空字符的处理相关内容，空字符的存在使得计算 First 集和 Follow 集时需要考虑到推导出空的情况，极大增加了程序实现时的复杂性。

第三困难的就是对算法的理解与实现，对 First 集和 Follow 集的算法理解不深入，导致程序在应对一些“特殊”情况时出现问题，就比如求 first 集时，右部第一个非终结符可以推出空的情况，一开始我没有考虑到，导致了后续实验结果总是出问题。

虽然有着以上诸多困难，但本次实验确实锻炼到了我对递归、算法本身以及对程序优化方面的理解，例如递归时：理清文法的逻辑结构非常重要。使用辅助非终结符和迭代算法能够有效地解决递归与循环依赖问题，但需要谨慎设计以避免引入新的复杂性。

再例如算法细节中的空字符部分：需要仔细处理可空产生式，确保在计算 First 集和 Follow 集时考虑到空字符的影响。可以使用循环、迭代等手段来处理可空产生式的影响。属实让我受益良多。

在分析表构造层和语法分析层，构建好分析表后需要查表对 token 流进行语法分析，如何读取 token 流、如何查表分析困惑了我很久。最后我先设计了一个类 Lexer，该类会将 token 流构造为一个链表，然后令语法分析类 Grammar 继承 Lexer 类以使用 token 流链表；然后是解决查表对 token 流部分，为了解决这个问题，我设计了两个栈，s1 用于存放符号栈，s2 用于存放输入栈。首先，通过 initStack 函数初始化这两个栈。随后，我遍历词法分析器输出的 Token 流，将 Token 读取到数组 reserve 中。为了构建输入栈 s2，我反向入栈 reserve 数组中的 Token，同时在栈底添加特殊符号#和文法的起始符号。进入主循环后，程序不断进行 LL(1)分析，每一步中显示符号栈和输入栈的内容，取出栈顶元素进行匹配或推导。在 LL(1)分析表中查找相应的推导式，并将推导式右侧的符号反向入栈 s1。对于错误处理，若找不到对应的推导式，程序输出错误信息并终止分析过程。最终，成功或失败的判断通过符号栈和输入栈是否同时剩余#来实现。

7、心得体会

在完成这个实验的过程中，我们遇到了一系列的挑战和困难，但通过不断地学习和合作，我们获得了许多宝贵的经验和体会。

首先，文法的设计和编码是整个实验的起点，也是一个相对困难的任务。理

解和符合 LL(1)文法规范需要深入理解编程语言的语法结构，而在实践中我们选择了 C 语言子集的文法。通过查阅资料和小组合作，我们最终设计出了符合要求的文法，并进行了有效的编码和储存。

其次，First 集和 Follow 集的计算是实验中一个复杂且关键的环节。递归计算和处理空字符推导十分困难，但通过合理的逻辑设计和算法优化，我们成功地克服了这些问题。对于递归问题，理清文法的逻辑结构非常关键，而对于空字符的处理，则需要在计算 First 集和 Follow 集时谨慎对待可空产生式的情况。

LL(1)分析表的构建也是一个需要深刻理解文法和算法的任务。通过逐步构建表格、查阅资料、团队协作，我们解决了一系列关于分析表的问题。这一阶段的工作帮助我更好地理解了文法与分析表之间的关系，加深了对 LL(1)分析的认识。

最后，设计并实现词法分析&语法分析器联合工作是整个实验的亮点之一。构建 Lexer 类、处理 Token 流、设计栈等步骤需要将理论知识转化为实际的程序结构。通过细致的设计和调试，我们成功地完成了对 Token 流的语法分析，实现了 LL(1)分析过程。

总结：这个实验让我们更深入地理解了编程语言的语法分析原理，提高了我的算法设计和实现能力。通过克服实验中的各种难题，我们不仅学到了专业知识，也培养了解决问题的能力。这次经历对我的编译原理课程学习和职业发展都具有积极的影响。

8、源代码

// main.c

```
#include <iomanip>
#include "Grammer.h"

using namespace std;

int main()
{
    Grammer G = Grammer("./Grammer/wenfa");

    // 获取词法分析部分输入
    G.scanner("./token");

    // 语法分析部分
    // G.printProc();
    // G.printTer();
    // G.printNonTer();
```

```
    // 计算 first 集
    G.getFirstSet();
    // G.printFirstSet();

    // 计算 follow 集
    G.getFollowSet();
    // G.printFollowSet();

    // 计算 select 集
    G.getSelectSet();
    // G.printSelectSet();

    // 构造分析表
    G.getTable();

    // 对 token 流进行语法分析
    G.runAnalysis();

    return 0;
}
```

// Grammer.h

```
#ifndef _SYNANALYSIS_H
#define _SYNANALYSIS_H

#include <iostream>
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <fstream>
#include <vector>
#include <iomanip>
#include "Lexer.h"
#include "Grammer.h"

#define GRAMMAR_ARROW 2000   //->
#define GRAMMAR_OR 2001      // |
#define GRAMMAR_NULL 2002    // 空值
#define GRAMMAR_SPECIAL 2003 // 特殊符号#
#define GRAMMAR_BASE 2010    // 动态生成的基值

#define Stack_Size 5000 // 栈的最大容量
#define Max_Proc 500    // 产生式的最大数量
#define Max_Length 500  // 产生式的最大长度
```

```
#define Max_NonTer 60   // 非终结符集合的最大数量
#define Max_Ter 60      // 终结符集合的最大数量
#define Max_Length2 100 // 分析表中产生式右部的长度

using namespace std;

// 栈元素
typedef struct
{
    char *token;
    int type;
    int line;
    int col;
} elemType;
// 分析栈
typedef struct
{
    elemType elem[Stack_Size];
    int top;
} SeqStack;

// 语法分析器类
class Grammer : public Lexer
{
public:
    explicit Grammer(const char *file_path);
    void getFirstSet();
    void getFollowSet();
    void getSelectSet();
    void getTable();
    void runAnalysis();

    void printProc();
    void printTer();
    void printNonTer();
    void printFirstSet();
    void printFollowSet();
    void printSelectSet();
    const char *searchMap(int num);
    void showStack(SeqStack *S);

private:
    int procNum;                // 产生式数量，proc 的维数都是从 1 开始的
```

```
    int proc[Max_Proc][Max_Length]{};// 产生式的数组，里边存储了终结符或者非终结符对应的编号
    int first[Max_Proc][Max_Length]{};     // 存储 First 集的数组
    int follow[Max_Proc][Max_Length]{};    // 存储 Follow 集的数组
    int select_[Max_Proc][Max_Length]{};   // 存储 Select 集的数组
    int analytable[Max_NonTer][Max_Ter][Max_Length2]; // 预测分析表，analytable[i][j]表示非终结符 i 和终结符 j 对应的产生式编号
    int connectFirst[Max_Length]{};  // 将某些 First 集结合起来的集合
    int firstVisit[Max_Proc]{};       // 记录某非终结符的 First 集是否已经求过
    int followVisit[Max_Proc]{};     // 记录某非终结符的 Follow 集是否已经求过
    int empty[Max_Proc]{};          // 可推出空的非终结符的编号
    int emptyRecu[Max_Proc]{};// 在求可推出空的非终结符的编号集时使用的防治递归的集合
    int followRecu[Max_Proc]{};       // 在求 Follow 集时使用的防治递归的集合
    fstream analyResult;              // 分析结果输出文件流
    vector<pair<const char *, int>> nonTerMap;// 非终结符映射表,不可重复
    vector<pair<const char *, int>> terMap; // 终结符映射表,不可重复
    vector<pair<const char *, int>> specialMap;// 文法中的特殊符号映射表，包括-> | $(空)

    // 拆分包含" | "的文法行
    static void split(int p[][Max_Length], int &line, int orNum);
    // 查找 word 对应编码
    int seekWordNum(char *word);
    // 计算能够直接推出空的非终结符集合
    void getNullSet(int currentNum);
    // 将来源集合合并至目标集合中
    void joinSet(int *destination, const int *source, int type);
    // 计算一个非终结符的 first 集(传入的参数是在非终结符集合中的序号)
    void getSingleFirstSet(int i);
    // 将 First 结合起来的函数
    void connectFirstSet(int *p);
    // 判断该非终结符是否能推出空，但终结符也可能传入，但没关系
    int canNonTer2Null(int currentNon);
    void getSingleFollowSet(int i);
    int dynamicNonTer(char *word);
    int isNonTer(int n);
    int isTer(int n);
    int inEmpty(int n);
    int inEmptyRecu(int n);
    int inFollowRecu(int n);
    int inProcRight(int n, const int *p);
    char *getWord(ifstream &infstream, char *array, char &ch);
    int findNonTerIndex(int nonTerSymbol);
```

```
    int findTerIndex(int TerSymbol);
    int getProcLength(const int proc[]);
};

#endif
```

// Grammer.cpp

```
#include "Grammer.h"

// 构造函数
Grammer::Grammer(const char *file_path)
{
    ifstream infstream(file_path); // 文件输入流
    char ch;                       // 用于存储读取的字符
    char array[30];                // 存储读取的单词的字符数组
    char *word;                    // 指向存储单词的动态分配内存
    int i;                         // 循环计数器
    int codeNum;                   // 存储单词对应的编码
    int line = 1;                  // 记录当前处理的行数
    int count = 0;                 // 记录当前处理的单词在一行中的位置
    int orNum = 0;                 // 记录当前处理的" | "的数量

    if (!infstream.is_open())
    {
        cout << "文法文件打开失败！请检查路径" << endl;
        exit(1);
    }

    // 初始化特殊符号映射表等
    specialMap.clear();
    specialMap.emplace_back("->", GRAMMAR_ARROW);
    specialMap.emplace_back("|", GRAMMAR_OR);
    specialMap.emplace_back("$", GRAMMAR_NULL);
    specialMap.emplace_back("#", GRAMMAR_SPECIAL);
    nonTerMap.clear();
    terMap.clear();

    // 清空
    memset(proc, -1, sizeof(proc));
    memset(first, -1, sizeof(first));
    memset(follow, -1, sizeof(follow));
    memset(select_, -1, sizeof(select_));
    memset(connectFirst, -1, sizeof(connectFirst));
    memset(firstVisit, 0, sizeof(firstVisit));
```

```
    memset(followVisit, 0, sizeof(followVisit));
    memset(empty, -1, sizeof(empty));
    memset(emptyRecu, -1, sizeof(emptyRecu));
    memset(followRecu, -1, sizeof(followRecu));
    memset(analytable, -1, sizeof(analytable));

    ch = infstream.get();
    i = 0;

    // 循环读取文件内容，直到文件末尾
    while (ch != EOF)
    {
        // 获取一个单词
        word = getWord(infstream, array, ch);

        // 获取单词对应的编码
        codeNum = seekWordNum(word);

        // 如果不是空白行，则处理当前单词
        if (codeNum != 0)
        {
            count++;
            // 如果是" | "，记录数量
            if (codeNum == GRAMMAR_OR)
                orNum++;
            // 记录当前单词的编码到 proc 数组中
            proc[line][count] = codeNum;
        }

        // 如果当前字符为换行符，表示一行的处理结束，拆分"|"文法
        if (ch == '\n')
        {
            // 文法增广
            split(proc, line, orNum);
            count = 0; // 重置
            orNum = 0; // 重置
            line++;    // 行号++
            ch = infstream.get();
        }
    }

    // 产式式数量
    procNum = line - 1;
}
```

```
// ============================ tools ============================ //
const char *Grammer::searchMap(int num)
{
    // 标志符
    if (num == IDENTIFER)
        return "id";

    // 处理文法中的特殊符号
    for (auto &i : specialMap)
        if (i.second == num)
            return i.first;

    // 处理非终结符
    for (auto &i : nonTerMap)
        if (i.second == num)
            return i.first;

    // 处理终结符
    for (auto &i : terMap)
        if (i.second == num)
            return i.first;
}
// 获取文法中的一个单词
char *Grammer::getWord(ifstream &infstream, char *array, char &ch)
{
    int i = 0;

    // 跳过空格和制表符
    while (ch == ' ' || ch == '\t')
        ch = infstream.get();

    // 读取一个单词
    while (ch != ' ' && ch != '\n' && ch != EOF)
    {
        array[i++] = ch;
        ch = infstream.get();
    }

    // 跳过空格和制表符
    while (ch == ' ' || ch == '\t')
        ch = infstream.get();

    // 动态分配内存存储单词
```

```
    char *word = new char[i + 1];
    memcpy(word, array, i);
    word[i] = '\0';

    return word;
}
// 分割函数，用于将包含" | "的文法行进行拆分
void Grammer::split(int p[][Max_Length], int &line, int orNum)
{
    // 如果当前行无内容或者没有需要拆分的文法，则直接返回
    if (p[line][1] == -1 || orNum == 0)
        return;

    int head = p[line][1];        // 记录文法头部
    int push = p[line][2];        // 记录文法第二部分（紧跟在头部之后）
    int length = 0;               // 用于计算当前行文法的长度
    int right, left;              // 用于拆分文法的边界索引
    int lineTrue = line + orNum; // 计算拆分后新的行号

    // 计算当前行文法的长度
    for (length = 3;; length++)
        if (p[line][length] == -1)
            break;

    length--;

    for (left = length, right = length; left >= 2;)
    {
        // 如果找到了" | "或者已经到达行首
        if (p[line][left] == GRAMMAR_OR || left == 2)
        {
            // 将拆分的文法信息存入新的行
            p[line + orNum][1] = head;
            p[line + orNum][2] = push;
            for (int i = left + 1; i <= right; i++)
                p[line + orNum][i - left + 2] = p[line][i];
            p[line + orNum][right - left + 3] = -1;

            right = left = left - 1;
            orNum--;
        }
        else
            left--;
    }
```

```
    // 更新行号
    line = lineTrue;
}
// 查找 word 对应编码
int Grammer::seekWordNum(char *word)
{
    // 处理文法中的特殊符号
    for (auto &i : specialMap)
        if (strcmp(word, i.first) == 0)
            return i.second;

    // 先搜索终结符映射表中有没有此终结符
    for (auto &i : terMap)
        if (strcmp(word, i.first) == 0)
            return i.second;

    // 在关键字映射表中查找
    for (auto &i : keyMap)
    {
        if (strcmp(word, i.first) == 0)
        {
            terMap.push_back(make_pair(word, i.second));
            return i.second;
        }
    }

    // 在运算符映射表中查找
    for (auto &i : opMap)
    {
        if (strcmp(word, i.first) == 0)
        {
            terMap.push_back(make_pair(word, i.second));
            return i.second;
        }
    }

    // 在限界符映射表中查找
    for (auto &i : limMap)
    {
        if (strcmp(word, i.first) == 0)
        {
            terMap.push_back(make_pair(word, i.second));
            return i.second;
```

```
        }
    }
    // 处理标志符
    if (strcmp(word, "id") == 0)
    {
        terMap.push_back(make_pair(word, IDENTIFER));
        return IDENTIFER;
    }
    // 处理关键字、运算符、限界符表，即非终结符
    else
        return dynamicNonTer(word);
}
// 将来源集合加入至目标集合中，type = 1 代表包括空（$），type = 2 代表不包括空
void Grammer::joinSet(int *destination, const int *source, int type)
{
    for (int i = 0; source[i] != -1; i++)
    {
        int element = source[i];

        // 检查目标集合中是否已经包含该元素
        bool exists = false;
        for (int j = 0; destination[j] != -1; j++)
        {
            if (destination[j] == element)
            {
                exists = true;
                break;
            }
        }

        // 如果目标集合中不包含该元素，则添加
        if (!exists)
        {
            // 如果 type = 2 且元素为 GRAMMAR_NULL，则不添加
            if (type == 2 && element == GRAMMAR_NULL)
                continue;

            // 添加元素到目标集合中
            int index = 0;
            while (destination[index] != -1)
                index++;

            destination[index] = element;
            destination[index + 1] = -1;
```

```
        }
    }
}
// 查找非终结符在 nonTerMap 中的位置
int Grammer::findNonTerIndex(int nonTerSymbol)
{
    for (int k = 0; k < nonTerMap.size(); k++)
        if (nonTerMap[k].second == nonTerSymbol)
            return k;

    return -1;
}
// 查找终结符在 TerMap 中的位置
int Grammer::findTerIndex(int TerSymbol)
{
    for (int k = 0; k < terMap.size(); k++)
        if (terMap[k].second == TerSymbol)
            return k;

    return -1;
}
// 获取产生式长度
int Grammer::getProcLength(const int proc[])
{
    int length = 3;
    for (length = 3; proc[length] != -1; length++)
        ;
    return length;
}
// 计算能够直接推出空的非终结符集合
void Grammer::getNullSet(int currentNum)
{
    // 遍历所有产生式
    for (int j = 1; j <= procNum; j++)
    {
        // 如果右边的第一个符号是当前字符，且产生式长度只有 1
        if (proc[j][3] == currentNum && proc[j][4] == -1)
        {
            // 将产生式左部加入能够推出空的集合
            int item[2];
            item[0] = proc[j][1];
            item[1] = -1;
            joinSet(empty, item, 1);
```

```
            // 递归处理产生式左部
            getNullSet(proc[j][1]);
        }
    }
}
// 判断该非终结符是否能推出空
int Grammer::canNonTer2Null(int currentNon)
{
    int item[2];
    item[0] = currentNon;
    item[1] = -1;

    int result = 1;
    int mark = 0;

    joinSet(emptyRecu, item, 1); // 先将此符号并入防递归集合中
    if (inEmpty(currentNon) == 1)
        return 1;

    for (int j = 1; j <= procNum; j++)
    {
        if (proc[j][1] == currentNon)
        {
            // 求出长度
            int rLen = getProcLength(proc[j]);

            // 如果右侧只有一个非终结符，并且这个非终结符可以推出空
            if (rLen - 2 == 1 && inEmpty(proc[j][rLen]))
                return 1;

            // 如果右侧只有一个终结符
            else if (rLen - 2 == 1 && isTer(proc[j][rLen]))
                return 0;

            // 右侧不只有一个非终结符
            else
            {
                // 检查该产生式右边有没有重复符号（重复就递归爆炸了啊啊啊！！！）
                for (int k = 3; k <= rLen; k++)
                    if (inEmptyRecu(proc[j][k]))
                        mark = 1;
                if (mark == 1)
                    continue;
                else
```

```
                {
                    // 没有重复的话，就挨个递归检查是否可以推导出空
                    for (int k = 3; k <= rLen; k++)
                    {
                        result *= canNonTer2Null(proc[j][k]); // 其实是挨个与运算，有一个可以推导出空，整个都可以
                        item[0] = proc[j][k];                 // 经典集合操作，不赘述
                        item[1] = -1;
                        joinSet(emptyRecu, item, 1); // 防止递归
                    }
                }
            }
            // 当前产生式能推导出空，寄！
            if (result == 1)
                return 1;
        }
    }
    return 0;
}
// 将 First 结合起来的函数
void Grammer::connectFirstSet(int *p)
{
    int i = 0;
    int flag = 0;
    int item[2];
    // 如果 P 的长度为 1
    if (p[1] == -1)
    {
        if (p[0] == GRAMMAR_NULL)
        {
            connectFirst[0] = GRAMMAR_NULL;
            connectFirst[1] = -1;
        }
        else
        {
            for (i = 0; i < nonTerMap.size(); i++)
            {
                if (nonTerMap[i].second == p[0])
                {
                    flag = 1;
                    joinSet(connectFirst, first[i], 1);
                    break;
                }
```

```
            }
            // 也可能是终结符
            if (flag == 0)
            {
                for (i = 0; i < terMap.size(); i++)
                {
                    if (terMap[i].second == p[0])
                    {
                        item[0] = terMap[i].second;
                        item[1] = -1;
                        joinSet(connectFirst, item, 2); // 终结符的First
集就是其本身
                        break;
                    }
                }
            }
        }
        // 如果 p 的长度大于 1
        else
        {
            for (i = 0; i < nonTerMap.size(); i++)
            {
                if (nonTerMap[i].second == p[0])
                {
                    flag = 1;
                    joinSet(connectFirst, first[i], 2);
                    break;
                }
            }
            // 也可能是终结符
            if (flag == 0)
            {
                for (i = 0; i < terMap.size(); i++)
                {
                    if (terMap[i].second == p[0])
                    {
                        item[0] = terMap[i].second;
                        item[1] = -1;
                        joinSet(connectFirst, item, 2); // 终结符的 First 集就
是其本身
                        break;
                    }
                }
```

```
        }
        flag = 0;
        int length = 0;
        for (length = 0;; length++)
            if (p[length] == -1)
                break;

        for (int k = 0; k < length; k++)
        {
            emptyRecu[0] = -1; // 相当于初始化这个防递归集合

            // 如果右部的当前字符能推出空并且还不是最后一个字符，就将之后的一个字符并入 First 集中
            if (canNonTer2Null(p[k]) == 1 && k < length - 1)
            {
                int u = 0;
                for (u = 0; u < nonTerMap.size(); u++)
                {
                    // 注意是记录下一个符号的位置
                    if (nonTerMap[u].second == p[k + 1])
                    {
                        flag = 1;
                        joinSet(connectFirst, first[u], 2);
                        break;
                    }
                }
                // 也可能是终结符
                if (flag == 0)
                {
                    for (u = 0; u < terMap.size(); u++)
                    {
                        // 注意是记录下一个符号的位置
                        if (terMap[u].second == p[k + 1])
                        {
                            item[0] = terMap[i].second;
                            item[1] = -1;
                            joinSet(connectFirst, item, 2);
                            break;
                        }
                    }
                }
                flag = 0;
            }
            // 到达最后一个字符，并且产生式右部都能推出空，将$并入 First 集中
```

```
            else if (canNonTer2Null(p[k]) == 1 && k == length - 1)
            {
                item[0] = GRAMMAR_NULL;
                item[1] = -1;
                joinSet(connectFirst, item, 1);
            }
            else
                break;
        }
    }
}
// 动态生成非终结符，在基点的基础上，确保不和终结符冲突
int Grammer::dynamicNonTer(char *word)
{
    int i = 0;
    int dynamicNum;
    for (i = 0; i < nonTerMap.size(); i++)
    {
        if (strcmp(word, nonTerMap[i].first) == 0)
            return nonTerMap[i].second;
    }
    if (i == nonTerMap.size())
    {
        if (i == 0)
        {
            dynamicNum = GRAMMAR_BASE;
            nonTerMap.push_back(make_pair(word, dynamicNum));
        }
        else
        {
            dynamicNum = nonTerMap[nonTerMap.size() - 1].second + 1;
            nonTerMap.push_back(make_pair(word, dynamicNum));
        }
    }
    return dynamicNum;
}
// 判断某个标号是不是非终结符的标号,1 代表是，0 代表否
int Grammer::isNonTer(int n)
{
    for (int i = 0; i < nonTerMap.size(); i++)
        if (nonTerMap[i].second == n)
            return 1;

    return 0;
```

```
}
// 判断某个标号是不是终结符的标号，1 代表是，0 代表否
int Grammer::isTer(int n)
{
    for (int i = 0; i < terMap.size(); i++)
        if (terMap[i].second == n)
            return 1;

    return 0;
}
// 判断某个标号在不在此时的 empty 集中，1 代表是，0 代表否
int Grammer::inEmpty(int n)
{
    // 当前 Empty 集的长度
    int emptyLength = 0;
    for (emptyLength = 0;; emptyLength++)
        if (empty[emptyLength] == -1)
            break;

    for (int i = 0; i < emptyLength; i++)
        if (empty[i] == n)
            return 1;

    return 0;
}
// 判断某个标号在不在此时的 emptyRecu 集中，1 代表是，0 代表否
int Grammer::inEmptyRecu(int n)
{
    // 当前 Empty 集的长度
    int emptyLength = 0;
    for (emptyLength = 0;; emptyLength++)
        if (emptyRecu[emptyLength] == -1)
            break;

    for (int i = 0; i < emptyLength; i++)
        if (emptyRecu[i] == n)
            return 1;

    return 0;
}
// 判断某个标号在不在此时的 followRecu 集中，1 代表是，0 代表否
int Grammer::inFollowRecu(int n)
{
    int followLength = 0;
```

```
    for (followLength = 0;; followLength++)
        if (followRecu[followLength] == -1)
            break;

    for (int i = 0; i < followLength; i++)
        if (followRecu[i] == n)
            return 1;

    return 0;
}
// 判断某个标号是不是在产生式的右边
int Grammer::inProcRight(int n, const int *proc)
{
    // 直接看右边
    for (int i = 3;; i++)
    {
        if (proc[i] == -1)
            return 0;
        if (proc[i] == n)
            return 1;
    }
}

// 输出文法
void Grammer::printProc()
{
    cout << "=======================================文法
=======================================\n";
    // 遍历处理后的文法
    for (int k = 1; k <= procNum; k++)
    {
        for (int j = 1; j < Max_Length; j++)
        {
            if (proc[k][j] != -1)
                cout << searchMap(proc[k][j]) << " ";
            else
                break;
        }
        cout << endl;
    }

    cout << endl;
    for (int k = 1; k <= procNum; k++)
    {
```

```
        for (int j = 1; j < Max_Length; j++)
        {
            if (proc[k][j] != -1)
                cout << proc[k][j] << " ";
            else
                break;
        }
        cout << endl;
    }
}
// 输出文法中的终结符
void Grammer::printTer()
{
    cout << "\n\n==================================文法终结符
==================================\n";
    for (auto &item : terMap)
        cout << item.first << "\t\t\t" << item.second << endl;
}
// 输出文法中的非终结符
void Grammer::printNonTer()
{
    cout << "\n\n==================================文法非终结符
==================================\n";

    for (auto &item : nonTerMap)
        cout << item.first << "\t\t\t" << item.second << endl;
    cout << endl;
}
// 输出 first 集
void Grammer::printFirstSet()
{
    cout <<
"\n**********************************First*****************************
**\n";
    for (int i = 0; i < nonTerMap.size(); i++)
    {
        printf("First[%s] = ", nonTerMap[i].first);
        for (int j = 0;; j++)
        {
            if (first[i][j] == -1)
                break;
            cout << searchMap(first[i][j]) << " ";
        }
        cout << endl;
```

```
    }
}
// 输出follow集
void Grammer::printFollowSet()
{
    cout << "\n***********************************Follow集
*****************************\n";

    for (int i = 0; i < nonTerMap.size(); i++)
    {
        printf("Follow[%s] = ", nonTerMap[i].first);
        for (int j = 0;; j++)
        {
            if (follow[i][j] == -1)
                break;

            cout << searchMap(follow[i][j]) << " ";
        }
        cout << endl;
    }
}
// 输出select集
void Grammer::printSelectSet()
{
    cout << "\n***********************************Select集
*****************************\n";

    for (int i = 0; i < procNum; i++)
    {
        cout << "select[" << i + 1 << "] = ";
        for (int j = 0;; j++)
        {
            if (select_[i][j] == -1)
                break;

            cout << searchMap(select_[i][j]) << " ";
        }
        cout << endl;
    }
}
bool isDescribe(const string &token)
{
    if (token == "auto" ||
        token == "const" ||
```

```
        token == "unsigned" ||
        token == "signed" ||
        token == "static")
    {
        return true;
    }
    return false;
}
bool isType(const string &token)
{
    if (token == "int" ||
        token == "char" ||
        token == "double" ||
        token == "float" ||
        token == "void")
    {
        return true;
    }
    return false;
}

// ============================ first ============================ //
// 计算一个非终结符的 first 集(传入的参数是在非终结符集合中的序号)
void Grammer::getSingleFirstSet(int i)
{
    int currentNon = nonTerMap[i].second; // 当前的非终结符编号
    // 依次遍历全部产生式
    for (int j = 1; j <= procNum; j++) // j 代表第几个产生式
    {
        // 找到该非终结符的产生式
        if (currentNon == proc[j][1]) // 注意从 1 开始
        {
            // 当右边的第一个是终结符或者空的时候，直接加入 first 集
            if (isTer(proc[j][3]) == 1 || proc[j][3] == GRAMMAR_NULL)
            {
                // 并入当前非终结符的 first 集中
                int item[2];
                item[0] = proc[j][3];
                item[1] = -1; // 集合最后一个元素标记为-1，便于遍历加入
                joinSet(first[i], item, 1);
            }

            // 当右边的第一个是非终结符的时候
            else if (isNonTer(proc[j][3]) == 1)
```

```
            {
                // 左递归处理
                if (proc[j][3] == currentNon)
                {
                    cout << "异常！文法存在左递归！！" << endl;
                    exit(1);
                }

                // 记录下右边第一个非终结符的位置
                int k = findNonTerIndex(proc[j][3]);

                // 当右边第一个非终结符还未访问过的时候，递归
                if (firstVisit[k] == 0)
                {
                    getSingleFirstSet(k);
                    firstVisit[k] = 1;
                }

                // 如果first[k]此时有空值的话，暂时不把空值并入first[i]中
                joinSet(first[i], first[k], 2);

                // 获取产生式右侧长度
                int rLength = getProcLength(proc[j]);

                // 循环处理后面的
                for (k = 3; k < rLength; k++)
                {
                    emptyRecu[0] = -1; // 相当于初始化这个防递归集合

                    // 如果右部的当前字符能推出空并且还不是最后一个字符，就将之后的一个字符并入First集中
                    if (canNonTer2Null(proc[j][k]) == 1 && k < rLength - 1)
                    {
                        // 找到下一个符号的下标
                        int u = findNonTerIndex(proc[j][k + 1]);

                        // 如果没被访问过，递归啊啊啊啊
                        if (firstVisit[u] == 0)
                        {
                            getSingleFirstSet(u);
                            firstVisit[u] = 1;
                        }
                        joinSet(first[i], first[u], 2);
```

```
                    }

                    // 到达最后一个字符，并且产生式右部都能推出空,将$并入 First
集中
                    else if (canNonTer2Null(proc[j][k]) == 1 && k ==
rLength - 1)
                    {
                        int item[2];
                        item[0] = GRAMMAR_NULL;
                        item[1] = -1;
                        joinSet(first[i], item, 1);
                    }
                    else
                        break;
                }
            }
        }
    }
    // 标记该非终结符已近被使用
    firstVisit[i] = 1;
}
// 计算文法中所有非终结符的 first 集合
void Grammer::getFirstSet()
{
    // 先求出能直接推出空的非终结符集合
    getNullSet(GRAMMAR_NULL);

    // 循环求每一个非终结符的 fist 集
    for (int i = 0; i < nonTerMap.size(); i++)
        getSingleFirstSet(i);
}

// ================================= follow
================================= //
// 计算一个非终结符的 follow 集(传入的参数是在非终结符集合中的序号)
void Grammer::getSingleFollowSet(int i)
{
    int currentNon = nonTerMap[i].second; // 当前的非终结符标号
    int result = 1;

    // 将当前标号加入防递归集合中
    int item[2];
    item[0] = currentNon;
    item[1] = -1;
```

```
    joinSet(followRecu, item, 1);

    // 如果当前符号就是开始符号,把特殊符号加入其 Follow 集（标识输入串的开始位置）
    if (proc[1][1] == currentNon)
    {
        item[0] = GRAMMAR_SPECIAL;
        item[1] = -1;
        joinSet(follow[i], item, 1);
    }

    // 开始遍历每一个产生式
    for (int j = 1; j <= procNum; j++)
    {
        // 如果该非终结符在某个产生式的右部存在
        if (inProcRight(currentNon, proc[j]) == 1)
        {
            int flag = 0;
            int k = 0;
            int rLen = 1;
            int leftNum = proc[j][1]; // 产生式的左边
            int kArray[Max_Length2];
            memset(kArray, -1, sizeof(kArray));

            // 找到该终结符在该产生式右侧的位置
            int h = findNonTerIndex(leftNum);

            // 找到所有包含当前非终结符的位置
            for (rLen = 1;; rLen++)
            {
                if (currentNon == proc[j][rLen + 2])
                    kArray[k++] = rLen;
                if (proc[j][rLen + 2] == -1)
                    break;
            }
            rLen--;

            // 遍历 kArray
            for (int y = 0;; y++)
            {
                // 遍历完了
                if (kArray[y] == -1)
                    break;

                // 如果该非终结符在右部产生式的最后
```

```
                if (kArray[y] == rLen)
                {
                    // 产生式的左部在防递归集合 followRecu 中，说明已经处理过该
非终结符的 Follow 集
                    // 直接并入
                    if (inFollowRecu(leftNum) == 1)
                    {
                        joinSet(follow[i], follow[h], 1);
                        continue;
                    }
                    // 如果没有访问过，递归
                    if (followVisit[h] == 0)
                    {
                        getSingleFollowSet(h);
                        followVisit[h] = 1;
                    }
                    // 递归回来后，直接并入
                    joinSet(follow[i], follow[h], 1);
                }

                // 如果不在最后
                else
                {
                    result = 1;
                    //  遍历从位置 kArray[y]+1 到 rLen 的符号
                    for (int n = kArray[y] + 1; n <= rLen; n++)
                    {
                        // 重置一下防止递归
                        emptyRecu[0] = -1;
                        // 挨个测试 kArray[y]+1 往后是否可以推导出空
                        result *= canNonTer2Null(proc[j][n + 2]);
                    }

                    // 至少有一个不能推导出空
                    if (result == 1)
                    {
                        // 如果在左部防递归集合 followRecu 中，说明已经处理过该
非终结符的 Follow 集，直接并入
                        if (inFollowRecu(leftNum) == 1)
                        {
                            joinSet(follow[i], follow[h], 1);
                            continue;
                        }
                        // 如果不在，说明还没求过，递归去求！！！
```

```
                    if (followVisit[h] == 0)
                    {
                        getSingleFollowSet(h);
                        followVisit[h] = 1;
                    }
                    // 求完了放进去
                    joinSet(follow[i], follow[h], 1);
                }

                // 如果都可以推导出空
                int item[Max_Length];
                memset(item, -1, sizeof(item));
                for (int n = kArray[y] + 1; n <= rLen; n++)
                    item[n - kArray[y] - 1] = proc[j][n + 2];

                // 数组尾部设置-1 表示结尾
                item[rLen - kArray[y]] = -1;
                connectFirst[0] = -1; // 重新初始化
                connectFirstSet(item);
                joinSet(follow[i], connectFirst, 2);
            }
        }
    }
    followVisit[i] = 1;
}
// 计算文法中所有非终结符的 follow 集合
void Grammer::getFollowSet()
{
    for (int i = 0; i < nonTerMap.size(); i++)
    {
        followRecu[0] = -1;
        getSingleFollowSet(i);
    }
}

// ============================== select ============================== //
// 求 Select 集,注意 Select 集中不能含有空($),因而 Type=2
void Grammer::getSelectSet()
{
    for (int i = 1; i <= procNum; i++) // i 代表第几个产生式
    {
        int leftNum = proc[i][1]; // 产生式的左边
        int h = 0;
```

```
        int result = 1;
        for (h = 0; h < nonTerMap.size(); h++)
            if (nonTerMap[h].second == leftNum)
                break;

        int rightLength = 1;
        for (rightLength = 1;; rightLength++)
            if (proc[i][rightLength + 2] == -1)
                break;

        rightLength--;

        // 如果右部推出式的长度为 1 并且是空,select[i-1] = follow[左边]
        if (rightLength == 1 && proc[i][rightLength + 2] == GRAMMAR_NULL)
            joinSet(select_[i - 1], follow[h], 2);

        // 如果右部不是空的时候,select[i-1] = first[右部全部]
        // 如果右部能够推出空，select[i-1] = first[右部全部] ^ follow[左边]
        else
        {
            int temp2[Max_Length];
            int n = 0;
            memset(temp2, -1, sizeof(temp2));
            for (n = 1; n <= rightLength; n++)
                temp2[n - 1] = proc[i][n + 2];

            temp2[rightLength] = -1;
            connectFirst[0] = -1; // 应该重新初始化一下
            connectFirstSet(temp2);
            joinSet(select_[i - 1], connectFirst, 2);

            for (n = 1; n <= rightLength; n++)
            {
                emptyRecu[0] = -1;
                result *= canNonTer2Null(proc[i][n + 2]);
            }

            // 如果右部能推出空，将 follow[左边]并入 select[i-1]中
            if (result == 1)
                joinSet(select_[i - 1], follow[h], 2);
        }
    }
}
```

```
// ========================= 构造分析表 ========================= //
// 输出预测分析表
void Grammer::getTable()
{
    // 打开文件用于写入预测分析表
    fstream outfile;
    outfile.open("./LL1 分析表.gr", ios::out);

    // 遍历所有产生式
    for (int i = 0; i < procNum; i++)
    {
        int m = 0; // 非终结符的序号

        // 查找产生式左部对应的非终结符在 nonTerMap 中的位置
        for (int t = 0; t < nonTerMap.size(); t++)
        {
            if (nonTerMap[t].second == proc[i + 1][1])
            {
                m = t;
                break;
            }
        }

        // 遍历该产生式的所有 SELECT 集合
        for (int j = 0;; j++)
        {
            if (select_[i][j] == -1)
                break;

            // 查找终结符在 terMap 中的位置
            for (int k = 0; k < terMap.size(); k++)
            {
                if (terMap[k].second == select_[i][j])
                {
                    int n = 0;

                    // 将产生式右部写入预测分析表
                    for (n = 1; n <= Max_Length2; n++)
                    {
                        analytable[m][k][n - 1] = proc[i + 1][n];

                        if (proc[i + 1][n] == -1)
                            break;
                    }
```

```
                    break;
                }
            }
        }
    }

    // 写入预测分析表到文件
    outfile << endl
            << "**********************************预测分析表
*****************************" << endl;
    for (int i = 0; i < nonTerMap.size(); i++)
    {
        for (int j = 0; j < terMap.size(); j++)
        {
            outfile << "analytable[" << nonTerMap[i].first << "][" <<
terMap[j].first << "] = ";

            // 输出预测分析表中的产生式右部
            for (int k = 0;; k++)
            {
                if (analytable[i][j][k] == -1)
                    break;
                outfile << searchMap(analytable[i][j][k]);
            }
            outfile << endl;
        }
        outfile << endl
                << endl;
    }

    // 关闭文件
    outfile.close();
}

// ================================== 分析源程序
================================== //
void initStack(SeqStack *S) /*初始化顺序栈*/
{
    S->top = -1;
}
int push(SeqStack *S, const elemType &e) /*进栈*/
{
    if (S->top == Stack_Size - 1)
        return 0;
```

```
    S->top++;
    S->elem[S->top] = e;
    return 1;
}
int pop(SeqStack *S) /*出栈*/
{
    if (S->top == -1)
        return 0;
    else
    {
        S->top--;
        return 1;
    }
}
int getTop(SeqStack *S, elemType *e) /*取栈顶元素*/
{
    if (S->top == -1)
        return 0;
    else
    {
        *e = S->elem[S->top];
        return 1;
    }
}
int getLen(elemType *array)
{
    for (int len = 0;; len++)
        if (array[len].type == -1)
            return len;
}
void Grammer::showStack(SeqStack *S) /*显示栈的字符，先输出栈底元素*/
{
    for (int i = S->top; i >= 0; i--)
        analyResult << searchMap(S->elem[i].type) << " ";
}

// 进行语法分析
void Grammer::runAnalysis()
{
    // 打开结果文件
    analyResult.open("./语法分析结果.gr", ios::out);

    SeqStack s1; // 符号栈
    SeqStack s2; // 输入栈
```

```
    int i = 0;
    elemType e;
    elemType prevE;

    elemType reserve[Stack_Size]; // 辅助反向入栈变量
    memset(reserve, -1, sizeof(reserve));

    Node *p = head; // 遍历链表指针

    // 初始化两个栈
    initStack(&s1);
    initStack(&s2);
    // # 和 第一条产生式入栈
    e.type = GRAMMAR_SPECIAL;
    push(&s1, e);
    e.type = proc[1][1];
    push(&s1, e);
    e.type = GRAMMAR_SPECIAL;
    push(&s2, e);

    // 读取 lex 的 token 流到 reserve
    while (p != NULL)
    {
        // 使用 isDescribe 和 isType 函数判断 token 的类型
        if (isDescribe(p->lexeme))
        {
            e.type = DESCRIBE;
            e.token = (char *)malloc(strlen(p->lexeme.c_str()) + 1);
            strcpy(e.token, p->lexeme.c_str());
            e.line = p->line;
            e.col = p->column;
            reserve[i++] = e;
        }
        else if (isType(p->lexeme))
        {
            e.type = TYPE;
            e.token = (char *)malloc(strlen(p->lexeme.c_str()) + 1);
            strcpy(e.token, p->lexeme.c_str());
            e.line = p->line;
            e.col = p->column;
            reserve[i++] = e;
        }
        else
        {
```

```
            e.type = p->type;
            e.token = (char *)malloc(strlen(p->lexeme.c_str()) + 1);
            strcpy(e.token, p->lexeme.c_str());
            e.line = p->line;
            e.col = p->column;
            reserve[i++] = e;
        }
        p = p->next;
    }

    // 反向入栈
    for (i = getLen(reserve); i > 0; i--)
        push(&s2, reserve[i - 1]);

    // 开始分析
    elemType e1, e2;

    while (true)
    {
        int flag = 0;
        int h1, h2;
        analyResult << "符号栈:";
        showStack(&s1);
        analyResult << endl;
        analyResult << "输入栈:";
        showStack(&s2);
        analyResult << endl;

        // 取栈顶元素
        prevE = e2;
        getTop(&s1, &e1);
        getTop(&s2, &e2);

        // 当符号栈和输入栈都剩余#，无错误
        if (e1.type == GRAMMAR_SPECIAL && e2.type == GRAMMAR_SPECIAL)
        {
            analyResult << "成功!" << endl;
            break;
        }

        if (e1.type == GRAMMAR_SPECIAL && e2.type != GRAMMAR_SPECIAL)
        {
            analyResult << "失败!" << endl;
            break;
```

```
        }

        // 符号栈的栈顶元素和输入串的栈顶元素相同时，同时弹出
        if (e1.type == e2.type)
        {
            pop(&s1);
            pop(&s2);
            flag = 1;
        }
        // 查表分析
        else
        {
            // 记录下非终结符的位置
            h1 = findNonTerIndex(e1.type);
            // 记录下终结符的位置
            h2 = findTerIndex(e2.type);

            if (analytable[h1][h2][0] == -1)
            {
                analyResult << "\n\n============ 出现错误！
============\n";
                analyResult << "===== 错误出现在下列两个 token 间 =====\n";
                analyResult << setw(10) << left << "token" << setw(10)
<< "line" << setw(10) << "colunm" << endl;
                analyResult << setw(10) << left << prevE.token <<
setw(10) << prevE.line << setw(10) << prevE.col << endl;
                analyResult << setw(10) << left << e2.token << setw(10)
<< e2.line << setw(10) << e2.col << endl;

                cout << "\n\n============ 出现错误！ ============\n";
                cout << "===== 错误出现在下列两个 token 间 =====\n";
                cout << setw(10) << left << "token" << setw(10) <<
"line" << setw(10) << "colunm" << endl;
                cout << setw(10) << left << prevE.token << setw(10) <<
prevE.line << setw(10) << prevE.col << endl;
                cout << setw(10) << left << e2.token << setw(10) <<
e2.line << setw(10) << e2.col << endl;
                break;
            }
            else
            {
                // 计算推导式的长度
                int len = getProcLength(analytable[h1][h2]);
```

```
                pop(&s1);
                // 如果不是空的话，反向入栈
                if (analytable[h1][h2][2] != GRAMMAR_NULL)
                {
                    for (int k = len - 1; k >= 2; k--)
                    {
                        e.type = analytable[h1][h2][k];
                        e.line = e.col = -1;
                        push(&s1, e);
                    }
                }
            }
        }

        if (flag == 1)
            analyResult << "匹配!" << endl
                        << endl;
        else
        {
            analyResult << "推出式:";
            int w = 0;
            // 记录下推导式的长度
            for (w = 0; analytable[h1][h2][w] != -1; w++)
                analyResult << searchMap(analytable[h1][h2][w]);
            analyResult << endl
                        << endl;
        }
    }

    // 关闭结果文件
    analyResult.close();
}
```

// Lexer.h

```
#pragma once

#include <vector>
#include <iostream>
#include <fstream>
#include <cstring>
#include <string>
#include <sstream>

using namespace std;
```

```
#define DESCRIBE 4000
#define TYPE 4001
#define STRING 24
#define DIGIT 26
#define IDENTIFER 25


// 定义链表节点的结构体
struct Node
{
    int type;
    string lexeme;
    int line;
    int column;
    int length;
    Node *next;
};

// 词法分析类
class Lexer
{
public:
    Lexer();
    void scanner(const char *);

protected:
    Node *head;
    Node *tail;
    vector<pair<const char *, int> > keyMap;
    vector<pair<const char *, int> > opMap;
    vector<pair<const char *, int> > limMap;

    void displayList();
    static Node *createNode(const string &line);
};
```

// Lexer.cpp

```
#include "Lexer.h"

using namespace std;

Lexer::Lexer()
{
    string line;
```

```
    string token;
    int num;
    head = tail = nullptr;

    // 初始化 keyMap
    ifstream file_1("./Grammer/Mapping/keyMapping");
    if (!file_1.is_open())
    {
        cerr << "错误：无法打开文件\n";
        return;
    }
    keyMap.clear();
    while (getline(file_1, line))
    {
        stringstream ss(line);
        ss >> token >> num;
        char* tokenCopy = new char[token.size() + 1];
        strcpy(tokenCopy, token.c_str());
        keyMap.emplace_back(tokenCopy,num);
    }
    file_1.close();

    // 初始化 opMap
    ifstream file_2("./Grammer/Mapping/opMapping");
    if (!file_2.is_open())
    {
        cerr << "错误：无法打开文件\n";
        return;
    }
    opMap.clear();
    while (getline(file_2, line))
    {
        stringstream ss(line);
        ss >> token >> num;
        char* tokenCopy = new char[token.size() + 1];
        strcpy(tokenCopy, token.c_str());
        opMap.emplace_back(tokenCopy,num);
    }
    file_2.close();

    // 初始化 limMap
    ifstream file_3("./Grammer/Mapping/limMapping");
    if (!file_3.is_open())
    {
```

```
        cerr << "错误：无法打开文件\n";
        return;
    }
    limMap.clear();
    while (getline(file_3, line))
    {
        stringstream ss(line);
        ss >> token >> num;
        char* tokenCopy = new char[token.size() + 1];
        strcpy(tokenCopy, token.c_str());
        limMap.emplace_back(tokenCopy,num);
    }
    file_3.close();
}

// 从文件读取内容并构造链表的函数
void Lexer::scanner(const char *filename)
{
    ifstream file(filename);
    if (!file.is_open())
    {
        cerr << "错误：无法打开文件\n";
        return;
    }

    string line;
    while (getline(file, line))
    {
        Node *newNode = createNode(line);
        if (head == nullptr)
            head = tail = newNode;
        else
        {
            tail->next = newNode;
            tail = newNode;
        }
    }

    file.close();
}

// 从文件内容的一行创建新节点的函数
Node *Lexer::createNode(const string &line)
{
```

```
    Node *newNode = new Node;

    // 使用 stringstream 来解析每一列的内容
    stringstream ss(line);

    // 读取每一列的内容并设置到节点中
    ss >> newNode->type >> newNode->lexeme >> newNode->line >> newNode-
>column >> newNode->length;

    // 设置 next 指针为 nullptr，因为这是新链表的最后一个节点
    newNode->next = nullptr;

    return newNode;
}

// 显示链表内容的函数
void Lexer::displayList()
{
    Node *current = head;
    while (current != nullptr)
    {
        cout << current->type << "\t" << current->lexeme << "\t"
             << current->line << "\t" << current->column << "\t" <<
current->length << "\n";
        current = current->next;
    }
}
```

// wenfa

```
<函数定义> -> <修饰词闭包><类型><变量>(<参数声明>){<函数块>}
<修饰词闭包> -> <修饰词><修饰词闭包> | $
<修饰词> -> describe
<类型> -> type <取地址>
<取地址> -> <星号闭包>
<星号闭包> -> <星号><星号闭包> | $
<星号> -> *
<变量> -> <标志符><数组下标>
<标志符> -> id
<数组下标> -> [ <因式> ] | $
<因式> -> ( <表达式> ) | <变量> | <数字>
<数字> -> digit
<表达式> -> <因子><项>
<因子> -> <因式><因式递归>
<因式递归> -> * <因式><因式递归> | / <因式><因式递归> | $
```

```
<项> -> + <因子> <项> | - <因子> <项> | $
<参数声明> -> <声明> <声明闭包> | $
<声明> -> <修饰词闭包> <类型> <变量> <赋初值>
<赋初值> -> = <右值> | $
<右值> -> <表达式> | { <多个数据> }
<多个数据> -> <数字> <数字闭包>
<数字闭包> -> , <数字> <数字闭包> | $
<声明闭包> -> , <声明> <声明闭包> | $
<函数块> -> <声明语句闭包> <函数块闭包>
<声明语句闭包> -> <声明语句> <声明语句闭包> | $
<声明语句> -> <声明> ;
<函数块闭包> -> <赋值函数> <函数块闭包> | <for 循环> <函数块闭包> | <条件语句> <函数
块闭包> | <函数返回> <函数块闭包> | $
<赋值函数> -> <变量> <赋值或函数调用>
<赋值或函数调用> -> = <右值> ; | ( <参数列表> ) ;
<参数列表> -> <参数> <参数闭包>
<参数闭包> -> , <参数> <参数闭包> | $
<参数> -> <标志符> | <数字> | <字符串>
<字符串> -> string
<for 循环> -> for ( <赋值函数> <逻辑表达式> ; <后缀表达式> ) { <函数块> }
<逻辑表达式> -> <表达式> <逻辑运算符> <表达式>
<逻辑运算符> -> < | > | == | !=
<后缀表达式> -> <变量> <后缀运算符>
<后缀运算符> -> ++ | --
<条件语句> -> if ( <逻辑表达式> ) { <函数块> } <否则语句>
<否则语句> -> else { <函数块> } | $
<函数返回> -> return <因式> ;
```