henryhua0721@foxmail.com

**2151127 华洲琦**

同济大学 计算机科学与技术1班

**类C语言编译器设计报告**

**2023-2024 编译原理课程设计**

**目录**

[**一、设计说明** 3](#_Toc166850589)

[**1.1 实验目的** 3](#_Toc166850590)

[**1.2 实验要求** 3](#_Toc166850591)

[**1.3 设计说明** 3](#_Toc166850592)

[**2.1 核心功能实现** 4](#_Toc166850593)

[**2.1 补充功能实现** 4](#_Toc166850594)

[**三、主要模块** 6](#_Toc166850595)

[**3.1 主页模块** 6](#_Toc166850596)

[**3.2 词法分析模块** 6](#_Toc166850597)

[**3.3 语法分析模块** 7](#_Toc166850598)

[**3.4 语义分析模块** 7](#_Toc166850599)

[**3.5 汇编代码生成模块** 8](#_Toc166850600)

[**四、核心功能函数** 9](#_Toc166850601)

[**4.1 词法分析模块** 9](#_Toc166850602)

[**4.1.1 scnner()** 9](#_Toc166850603)

[**4.2 语法分析模块** 11](#_Toc166850604)

[**4.2.1 CGrammer::readGrammer** 11](#_Toc166850605)

[**3.2.2 LR1\_grammer::generateFirst()** 11](#_Toc166850606)

[**4.2.3 LR1\_grammer::generateFirst()** 12](#_Toc166850607)

[**4.2.4 LR1\_grammer::fillGuiYueTable ()** 13](#_Toc166850608)

[**4.2.5 grammaticalAnalysis** 13](#_Toc166850609)

[**4.3 语义分析模块** 14](#_Toc166850610)

[**4.3.1 SemanticAnalysis::Analysis** 14](#_Toc166850611)

[**4.3.2 SemanticAnalysis::Semantic\_Analysis\_ExtDef（例）** 15](#_Toc166850612)

[**4.3.3 SemanticAnalysis::Semantic\_Analysis\_CreateFunTable\_m（例）** 15](#_Toc166850613)

[**4.4 汇编代码生成模块** 16](#_Toc166850614)

[**4.4.1 Assembly\_Generator::generate** 16](#_Toc166850615)

[**4.4.2 Assembly\_Generator::callTypeHandler（例）** 16](#_Toc166850616)

[**4.4.3 Assembly\_Generator::callTypeHandler（例）** 17](#_Toc166850617)

[**五、界面与结果展示** 18](#_Toc166850618)

[**5.1 界面展示** 18](#_Toc166850619)

[**5.2 文件输出** 19](#_Toc166850620)

[**5.3 结果展示** 21](#_Toc166850621)

[**5.3.1 正确程序结果展示** 21](#_Toc166850622)

[**5.3.2 错误程序结果展示** 22](#_Toc166850623)

[**六、总结与体会** 23](#_Toc166850624)

[**七、附件 - 源代码** 24](#_Toc166850625)

[**7.1 头文件** 24](#_Toc166850626)

[**7.1.1 1\_Lexical\_Analysis.h** 24](#_Toc166850627)

[**7.1.2 1\_Lexical\_PreProcess.h** 24](#_Toc166850628)

[**7.1.3 2\_Grammatical\_Analysis.h** 25](#_Toc166850629)

[**7.1.4 2\_Grammatical\_LR1Grammer.h** 26](#_Toc166850630)

[**7.1.5 3\_Semantic\_Analysis.h** 27](#_Toc166850631)

[**7.1.6 4\_Assembly\_Generator.h** 31](#_Toc166850632)

[**7.1.7 4\_Register\_Info.h** 32](#_Toc166850633)

[**7.2 源文件** 33](#_Toc166850634)

[**7.2.1 1\_Lexical\_Analysis.cpp** 33](#_Toc166850635)

[**7.2.2 1\_Lexical\_PreProcess.cpp** 40](#_Toc166850636)

[**7.2.3 2\_Grammatical\_Analysis.cpp** 43](#_Toc166850637)

[**7.2.4 2\_Grammatical\_LR1Grammer.cpp** 47](#_Toc166850638)

[**7.2.5 2\_Grammatical\_Output.cpp** 59](#_Toc166850639)

[**7.2.6 3\_Semantic\_Analysis.cpp** 61](#_Toc166850640)

[**7.2.7 4\_Assembly\_Generator.cpp** 80](#_Toc166850641)

[**7.2.8 main.cpp** 95](#_Toc166850642)

# **一、设计说明**

## **1.1 实验目的**

1. 掌握使用高级程序语言实现一个一遍完成的、简单语言的编译器的方法；
2. 掌握简单的词法分析器、语法分析器、符号表管理、中间代码生成以及目标代码生成的实现方法；
3. 掌握将生成代码写入文件的技术。

## **1.2 实验要求**

1. 使用高级程序语言作为实现语言，实现一个类C语言的编译器。编码实现编译器的组成部分。
2. 要求的类C编译器是个一遍的编译程序，词法分析程序作为子程序，需要的时候被语法分析程序调用;
3. 使用语法制导的翻译技术，在语法分析的同时生成中间代码，并保存到文件中。
4. 要求输入类C语言源程序，输出中间代码表示的程序;
5. 要求输入类C语言源程序，输出目标代码（可汇编执行）的程序。
6. 实现过程、函数调用的代码编译
7. 拓展类C语言文法，实现包含数组的中间代码以及目标代码生成。

## **1.3 设计说明**

实现了类C语言编译器，在上学期代码上进行补充、整合和优化，总计实现模块如下：

* 系统信息打印模块
* 词法分析模块
* 语法分析模块
* 语义分析模块
* 中间代码生成模块
* 汇编代码生成模块
* 文件输出模块

以上模块均使用C++代码实现，结合STL库等工具进行开发。

**二、功能实现**

系统功能主要划分为：**系统信息打印模块/词法分析模块/语法分析模块/语义分析模块/中间代码生成模块/汇编代码生成模块/文件输出模块**。其中系统信息打印模块和文件输出模块为补充功能，其余皆为类C编译器的核心功能。

## **2.1 核心功能实现**

核心功能实现的模块调用和数据流流动如下图所示：

图示

描述已自动生成

* **词法分析**：将源代码转换为一系列的词法单元（tokens），每个词法单元代表源代码中的一个特定的词法结构（关键字、标识符、常量和运算符等）。模块负责识别和提取源代码中的词法单元，并将它们传递给语法分析模块。
* **语法分析**：接收词法分析模块提供的词法单元序列，并根据语法规则检查源代码的语法正确性。它构建抽象语法树（Abstract Syntax Tree，AST），表示源代码的结构，并进一步验证源代码是否符合语法规范。如果源代码存在语法错误，模块具有足够的容错能力并提供相应的错误信息。
* **语义分析**：语义分析模块在语法分析的基础上，对抽象语法树进行进一步分析。作用为检查源代码中的语义错误（类型错误、未声明的变量等），并生成相应的错误信息。
* **中间代码生成**：将抽象语法树转换为中间表示形式（本项目使用四元式表示）。实现目的为将源代码转换为一种更抽象、更易于优化和后续处理的形式，为后续的编译步骤做准备。
* **汇编代码生成**：将中间代码转换为目标机器的汇编代码。模块包括指令选择、寄存器分配、代码优化等步骤，目的是生成高效的汇编代码。

## **2.1 补充功能实现**

* **系统信息打印模块**实现了系统的基本输出显示、以及系统头部的信息介绍。实现方法为控制台字符画的打印，效果如下图所示：

文本

描述已自动生成

* **文件输出模块**包含编译阶段每个中间内容的输出，实现方式为C++文件读写（以及穿线表语法树的绘制），输出文件列表如下：

图形用户界面, 应用程序

描述已自动生成

# **三、主要模块**

## **3.1 主页模块**

输出系统基本信息和使用的文件信息。

界面效果如下：

文本

描述已自动生成

## **3.2 词法分析模块**

对 C 语言源程序进行预处理，然后进行词法分析，输出类 C 语言的文法符号二元组序列，提供给语法分析器进行语法分析。

实现结果如下：

电脑屏幕截图

描述已自动生成

存储的文件内容如下，左侧为源程序，右侧为文法符号二元组序列：

电脑屏幕截图

描述已自动生成

## **3.3 语法分析模块**

语法分析分为文法读取、FIRST集生成、项目集族生成、LR1分析表生成、规约共5个过程。

* 文法读取过程将读取文件（存储C语言文法），将读取的文法存储到文件中（包括起始产生式位置、终结符集、非终结符集、产生式），内容如下：

文本

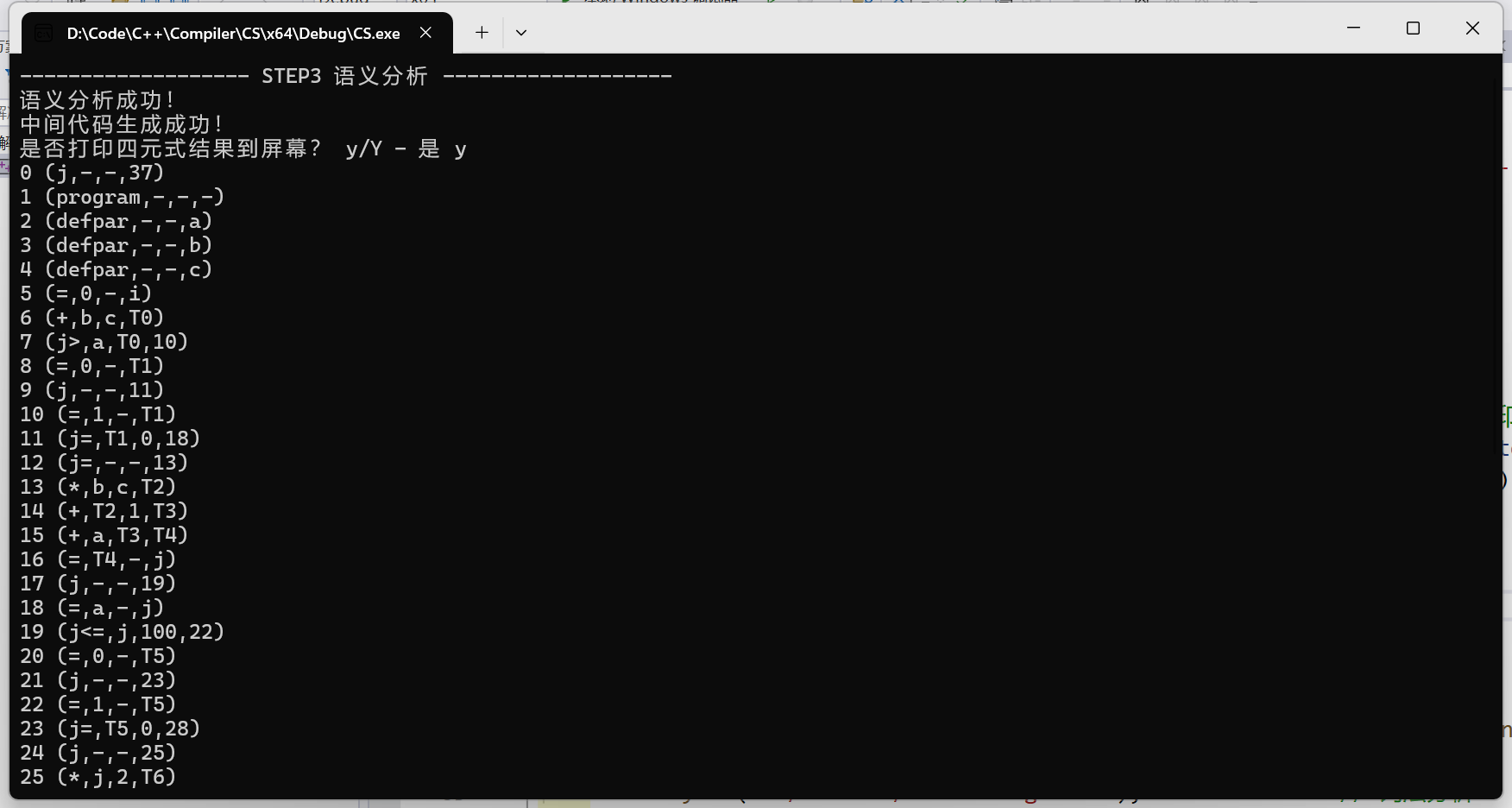
描述已自动生成

* 经过LR1规约后获得的LR1分析表如下：（表过长未完全展示）



## **3.4 语义分析模块**

语义分析在语法分析的基础上进一步实现检查（相较于语法分析，语义分析更关注代码的含义和行为，而不仅仅是语法的正确性），输出为四元式的形式：



同时输出到文件中：



## **3.5 汇编代码生成模块**

该模块使用语义分析获得的中间代码（四元式）生成汇编代码，执行类成员函数函数Assembly\_Generator generator.generate() 实现生成过程。

生成结果如下：

文本

描述已自动生成

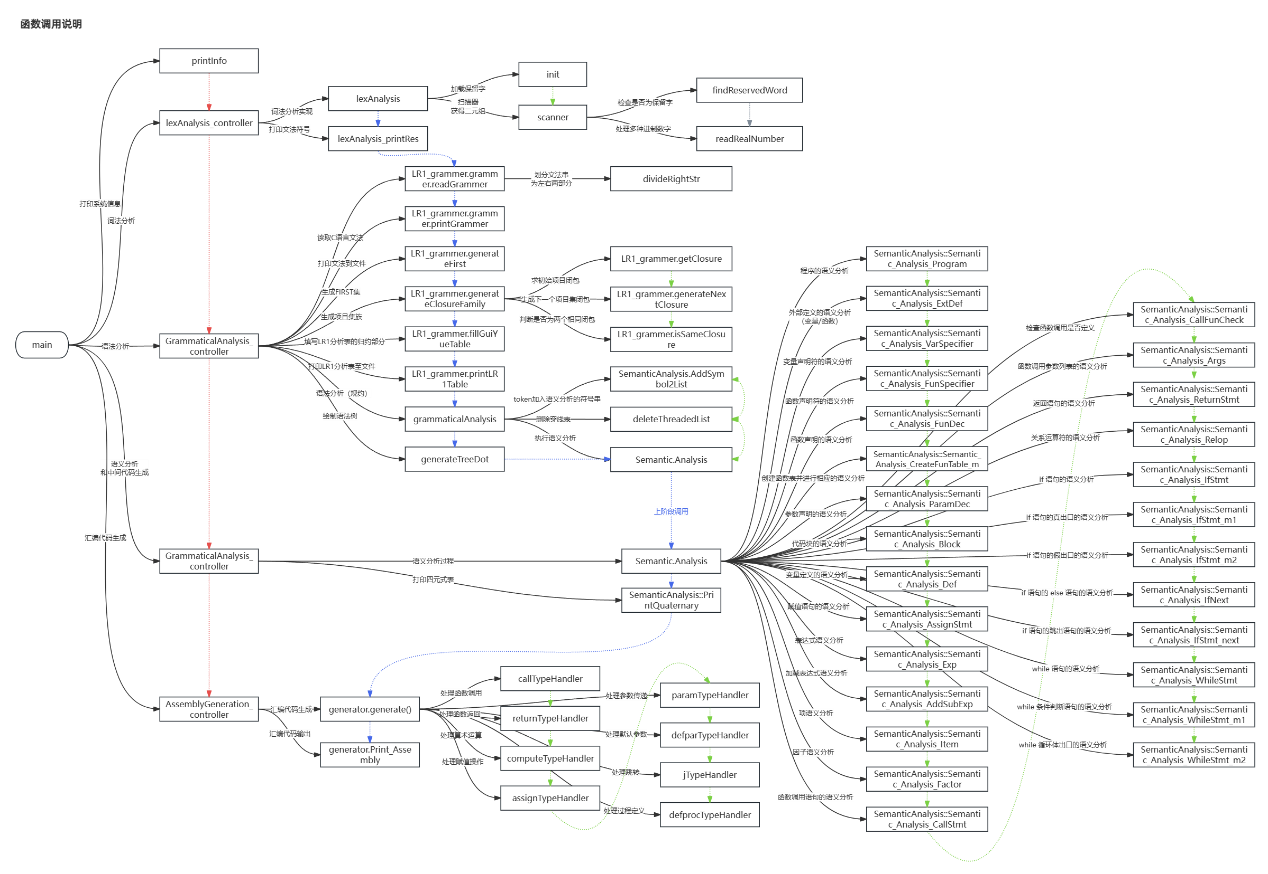
同样存储到.asm文件中

图形用户界面, 文本, 应用程序

描述已自动生成

# **四、核心功能函数**

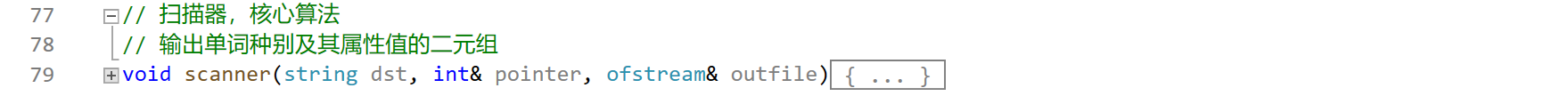
项目所有函数调用流程图如下所示：



以下是各模块核心函数详解：

## **4.1 词法分析模块**

### **4.1.1 scnner()**



**作用**：实现了一个词法分析器的核心功能扫描器，将输入源代码分解成一个个单独的记号（token）。

**变量**：

|  |  |
| --- | --- |
| **变量名** | **作用** |
| dst | 源代码字符串 |
| pointer | 指向源代码字符串中当前字符的索引 |
| outfile | 用于写入识别出的记号的输出流 |
| row | 当前行号 |
| col | 当前列号 |
| lastIsIdentifier | 一个标志，表示上一个记号是否是标识符 |
| coding | 识别出的记号的类型码 |
| word | 识别出的记号的字符串表示 |
| attribute | 记号的额外信息（如数字的值、字符串字面值等） |

实现流程：

1. **预处理**：借助循环实现代码无关内容的处理
   * 跳过空白字符（空格、制表符、换行符、回车符）
   * 删除单行注释（以 // 开头）
   * 删除多行注释（以 /\* 开头，以 \*/ 结尾）
2. **记号识别**：识别记号的类型
   * **标识符或关键字**
     + 字母、数字或下划线开头，代码将读取所有后续的字母、数字或下划线。
     + 使用 findReservedWord 函数检查收集到的单词是否是保留关键字。
     + 是关键字，代码将 coding 设置为相应的类型码，并将 attribute 设置为 "--"。
     + 不是关键字，将 coding 设置为 -1，并将 attribute 设置为单词本身。
   * **数字**
     + 当前字符是数字，代码将读取所有后续的数字和可能的十进制点，以形成数字。
     + 使用 readRealNumber 函数来处理读取数字的细节。
     + 将 coding 设置为数字的类型码，并将 attribute 设置为数字的字符串表示。
   * **字符串字面值**
     + 当前字符是双引号 (")，代码将读取所有后续的字符，直到遇到另一个双引号。
     + 将 coding 设置为字符串字面值的类型码，并将 attribute 设置为字符串字面值本身（包括双引号）。
   * **其他符号**
     + 代码读取当前字符和下一个字符，形成一个双字符符号。
     + 使用 findReservedWord 函数检查该符号是否是保留关键字。
     + 如果是关键字，代码将 coding 设置为相应的类型码，并将 attribute 设置为 "--"。
     + 如果不是关键字，将 coding 设置为符号的类型码，并将 attribute 设置为 "--"。
3. **输出和记号存储**
   * 将识别出的记号写入输出流，格式为：
   * 创建一个包含记号信息的 TOKEN 对象，并将其添加到 lexAnswer 向量中。

## **4.2 语法分析模块**

### **4.2.1 CGrammer::readGrammer**

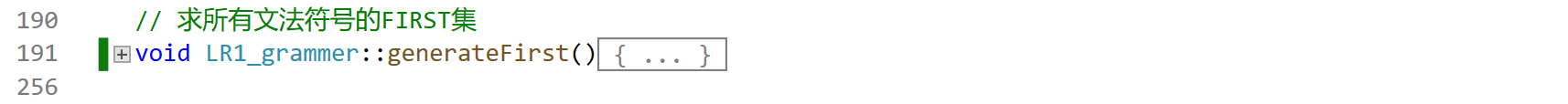


**作用**：读取文件中的文法，存储在 CGrammer 对象中

**实现流程**：

1. **打开文件**
2. **读取文件行**
3. **处理注释和空行**
   * 行首字符是注释符号（EXEG\_SIGN）或为空，则跳过并继续读取下一行。
4. **分割行内容**
   * 使用 divideRightStr 函数将行内容根据分隔符（DERI\_SIGN）分割为左右两部分，分别存储在 left 和 right 变量中。
   * 分割后的结果不是两部分，则会打印错误信息并退出程序。
5. **处理终结符**
   * left 等于 "%VT"，则表示该行定义的是终结符集合。
   * 使用 divideRightStr 函数将 right 部分根据分隔符（SEPE\_SIGN）分割为多个终结符，并将其添加到 Terminals 集合中。
6. **处理产生式**
   * left 不等于 "%VT"，则表示该行定义的是产生式。
   * 使用 divideRightStr 函数将 right 部分根据分隔符（SEPE\_SIGN）分割为多个产生式，并将其添加到 productItems 向量中。
   * 对于每个产生式，将其左部符号添加到 NTterminals 集合中。
   * 如果该产生式的左部符号只有一个字符且该字符是起始符号（INIT\_SIGN），则将该产生式的下标存储在 startItemPos 变量中。

### **3.2.2 LR1\_grammer::generateFirst()**

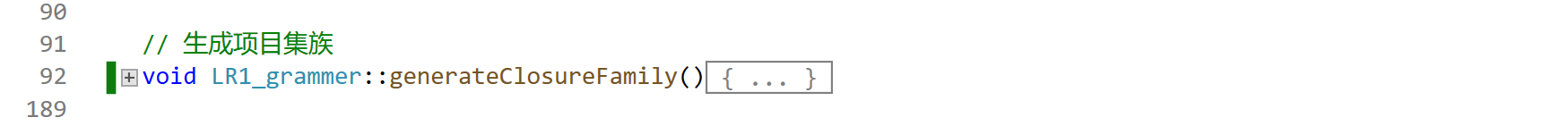


**作用**：计算所有文法符号的FIRST集

**实现流程**：

1. **计算终结符的 First 集**
   * 对于每个终结符，其 First 集就是它自身。
   * 将每个终结符的 First 集存储在 FIRST 字典中。
2. **计算非终结符的FIRST集**
   * 迭代算法计算非终结符的 First 集，反复执行以下步骤，直到所有非终结符的 First 集不再发生变化：
     1. 遍历所有非终结符
     2. 对于每个非终结符，遍历所有包含该非终结符的产生式
     3. 对于每个产生式，检查其右部第一个符号是否为终结符：
        + 如果是终结符，则将该终结符添加到非终结符的 First 集中
        + 如果是非终结符，则将该非终结符的 First 集（不包括空串）添加到非终结符的 First 集中
        + 如果该非终结符可以推导出空串，则继续检查产生式右部的下一个符号
     4. 如果非终结符的 First 集发生了变化，则将 changed 标记设置为 true
   * 如果 changed 标记为 false，则说明所有非终结符的 First 集都已经计算完成，算法结束。
3. 存储FIRST集
   * 将所有非终结符的 First 集存储在 FIRST 字典中

### **4.2.3 LR1\_grammer::generateFirst()**



**作用**：该函数负责生成项目集族

**实现流程**：

1. **创建初始项目集**
   * 创建一个名为 startClosure 的集合，用于存储初始项目集。
   * 创建一个名为 startItem 的 LR1\_item 对象，代表初始项目。
   * 从文法的起始产生式中构造 startItem 对象：
     + 将起始产生式的左部符号设置为 startItem 的左部符号。
     + 将起始产生式的右部符号设置为 startItem 的右部符号。
     + 将 startItem 的点位置设置为 0。
     + 将 startItem 的展望符号设置为 #。
   * 将 startItem 添加到 startClosure 集合中。
   * 调用 getClosure 函数计算 startClosure 的闭包。
   * 将 startClosure 的闭包添加到项目集族 closureFamily 中。
2. **生成项目集族**
   * 遍历项目集族中的每个项目集。
   * 对于每个项目集，遍历所有终结符和非终结符。
   * 对于每个终结符和非终结符，调用 generateNextClosure 函数生成下一个项目集的闭包。
   * 若下一个项目集的闭包不空，则检查项目集族中是否已经存在相同的状态。
   * 如果存在相同状态，则将当前项目集和下一个项目集的闭包之间的转移关系添加到 LR1 分析表中。
   * 如果不存在相同状态，则将下一个项目集的闭包添加到项目集族中，并将当前项目集和下一个项目集的闭包之间的转移关系添加到 LR1 分析表中。

### **4.2.4 LR1\_grammer::fillGuiYueTable ()**

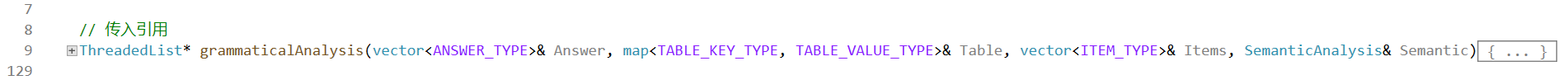


**作用**：填充 LR1 分析表的归约部分。

**实现流程**：

1. **遍历项目集族**
   * 遍历项目集族中的每个项目集。
   * 对于每个项目集，遍历其中的每个项目。
2. **判断归约项目**
   * 如果项目的点位置等于其右部符号的长度，则该项目是一个归约项目。
3. **查找产生式**
   * 遍历文法中的所有产生式，找到与当前项目匹配的产生式。
   * 匹配条件：
     + 产生式的左部符号与项目的左部符号相同。
     + 产生式的右部符号与项目的右部符号相同（不包括点）
4. **填充 LR1 分析表**
   * 如果找到匹配的产生式，则将该产生式的编号和归约动作添加到 LR1 分析表的对应位置。
   * 对于 ACC 产生式，将归约动作设置为 "acc"。
   * 对于其他产生式，将归约动作设置为 "r"。
   * 如果 LR1 分析表的对应位置已经存在其他动作，则会发生冲突。

### **4.2.5 grammaticalAnalysis**



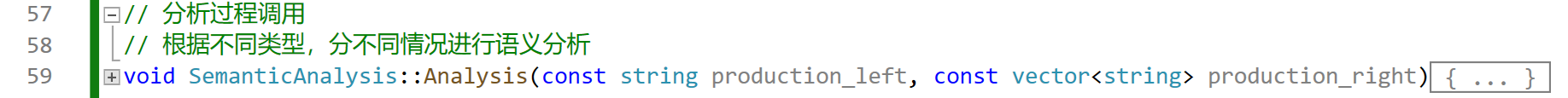
**作用**：语法分析的规约过程。

**实现流程**：

1. **初始化**
   * 创建语义分析对象 Semantic。
   * 将 Program 符号添加到语义分析的符号列表中。
   * 创建一个空 token 并将其添加到答案列表中，作为结束标志。
   * 初始化状态栈、符号栈和穿线表栈。
   * 初始化字符串指针和规约计数器。
2. **循环**
   * 获取当前输入符号。
   * 根据当前状态和符号，查询 LR1 分析表。
   * 根据分析表的结果执行相应的操作：
     + 如果是 "acc"，则说明规约成功，返回穿线表根结点。
     + 如果是 "s"，则将当前符号压入符号栈，并将对应状态压入状态栈，并移动字符串指针。
     + 如果是 "r"，则进行规约操作：
       - 1. 将规约项的右部长度个符号从符号栈和状态栈中弹出。
         2. 创建一个新的穿线表结点，并将规约项的左部符号作为该结点的符号。
         3. 将弹出符号对应的穿线表结点连接到新结点的子结点上。
         4. 将新结点压入穿线表栈。
         5. 将规约项的左部符号压入符号栈，并将对应状态压入状态栈。
     + 如果是其他操作，则说明分析表出错。
3. 在每次规约操作后，调用语义分析对象的 Analysis 函数进行语义分析。

## **4.3 语义分析模块**

### **4.3.1 SemanticAnalysis::Analysis**



**作用**：语义分析的集中调用函数，对于不同的产生式左部，代码会调用相应的语义分析函数进行处理。

**实现流程**：

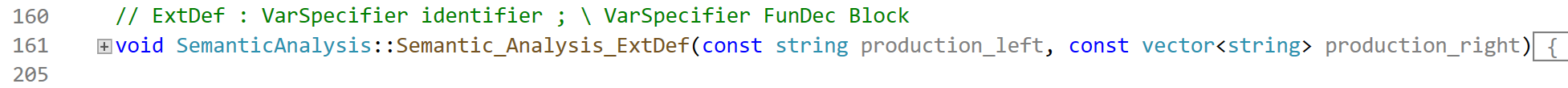
根据给出的左值的不同，可以将语义分析划分为以下子功能模块：

图片包含 图示

描述已自动生成

根据不同的类型实现不一样的语义检查。以下以其中两个语义检查函数为例。

### **4.3.2 SemanticAnalysis::Semantic\_Analysis\_ExtDef（例）**

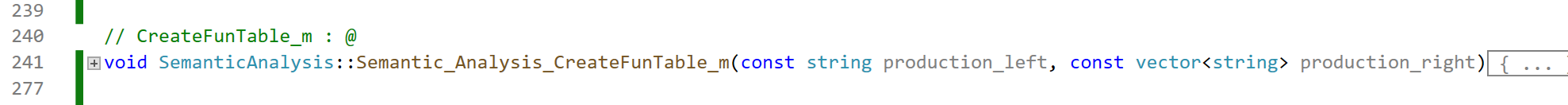


**作用**：对于代码中的外部定义（函数/变量）进行语义分析

**实现流程**：

1. **判断定义类型**
   * 判断外部定义是定义变量 or函数。
2. **定义变量**
   * 获取变量的类型和名称。
   * 判断变量是否在当前层已经定义。
   * 如果变量未定义，则将变量添加到当前层的符号表中。
   * 更新符号列表。
3. **定义函数**
   * 获取函数的名称。
   * 退出当前作用域。
   * 更新符号列表。

### **4.3.3 SemanticAnalysis::Semantic\_Analysis\_CreateFunTable\_m（例）**



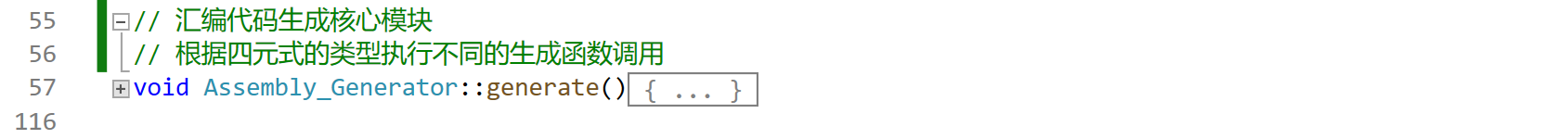
**作用**：创建函数符号表并进行相关语义分析

**实现流程**：

1. **获取函数信息**
   * 获取函数的名称和返回值类型
2. **判断函数是否重定义**
   * 在全局符号表中判断函数名称是否已经存在。
   * 如果函数名称已经存在，则报错并退出程序。
3. **创建函数符号表**
   * 创建一个新的函数符号表，并将其添加到符号表数组中。
   * 在全局符号表中添加一个函数符号项，该符号项包含函数名称、返回值类型、函数符号表索引等信息。
4. **压入函数符号表索引**
   * 将函数符号表索引压入当前作用域栈中
5. **添加返回值变量**
   * 创建一个返回值变量的符号项，并将其添加到函数符号表中。
   * 将返回值变量的四元式索引设置为当前四元式序号。
6. **记录主函数入口**
   * 如果函数名称为 "main"，则记录其入口四元式序号。
7. **更新符号列表**
   * 将函数名称、行号、列号、函数符号表索引等信息添加到符号列表中

## **4.4 汇编代码生成模块**

### **4.4.1 Assembly\_Generator::generate**



**作用**：根据操作类型调用不用函数，生成汇编代码。

**实现流程**：

1. **初始化寄存器使用时间**
   * 将所有寄存器未使用时间设置为最大值
2. **生成标签**
   * 如果当前四元式需要输出标签，则输出
3. **根据操作类型调用不同的处理函数**
   * callTypeHandler： 处理函数调用操作。
   * returnTypeHandler： 处理函数返回操作。
   * computeTypeHandler： 处理算术运算操作。
   * assignTypeHandler： 处理赋值操作。
   * paramTypeHandler： 处理参数传递操作。
   * defparTypeHandler： 处理默认参数操作。
   * jTypeHandler： 处理跳转操作。
   * defprocTypeHandler： 处理过程定义操作。

下面以两个调用函数为例进行解释。

### **4.4.2 Assembly\_Generator::callTypeHandler（例）**

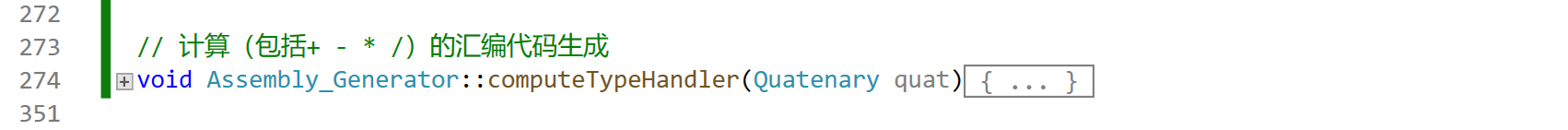


**作用**：处理函数调用操作。

**实现流程**：

1. **保存寄存器现场**
   * 将寄存器 $t0 到 $t7 的值压入栈中
2. **分配栈帧**
   * 将堆栈指针 $sp 减去 32，为新函数分配栈帧
3. **保存返回地址**
   * 将返回地址 $ra 压入栈中
4. **更新堆栈指针**
   * 将堆栈指针 $sp 设置为当前帧指针 $fp 的值
5. **更新栈帧**
   * 将堆栈指针 $sp 减去8，为新函数的局部变量分配空间
6. **跳转到函数**
   * 使用 jal 指令跳转到函数 quat.op1
7. **恢复寄存器现场**
   * 将寄存器 $t0 到 $t7 的值从栈中弹出，恢复现场
8. **恢复堆栈指针**
   * 将堆栈指针 $sp 恢复到上一帧的栈顶，其中 32 是八个寄存器偏移，剩下的是传入参数的偏移
9. **恢复帧指针**
   * 将帧指针 $fp 恢复到上一帧的旧帧指针位置
10. **处理返回值**
    * 如果函数调用时设置了变量接收返回值，则将返回值从寄存器 $v1 中移动到相应的变量中
11. **重置参数数量**
    * 将参数数量 paramsNumber 重置为 0

### **4.4.3 Assembly\_Generator::callTypeHandler（例）**



**作用**：处理算术运算操作

**处理流程**：

1. **分配寄存器**
   * 为操作数 B 和 C 分配寄存器。
   * 如果 B 在内存中，使用 loadVar 函数将 B 值从内存加载到寄存器中。
   * 如果 C 不是立即数，使用 loadVar 函数将 C 值从内存加载到寄存器中。
   * 如果 A 在内存中或尚未分配寄存器，则为 A 分配一个寄存器。
2. **编写汇编代码**
   * 加法：使用 addi 或 add 指令。
   * 减法：使用 subi 或 sub 指令。
   * 乘法：使用 mul 指令。
   * 除法：使用 div 指令。
3. **更新寄存器使用时间**
   * 将参与运算的寄存器的使用时间重置为 0

# **五、界面与结果展示**

## **5.1 界面展示**

|  |  |
| --- | --- |
| **功能模块** | **界面展示** |
| **系统主页** |  |
| **词法分析** |  |
| **语法分析** |  |
| **语义分析** |  |
| **汇编代码生成** |  |
| **结束界面** |  |

## **5.2 文件输出**

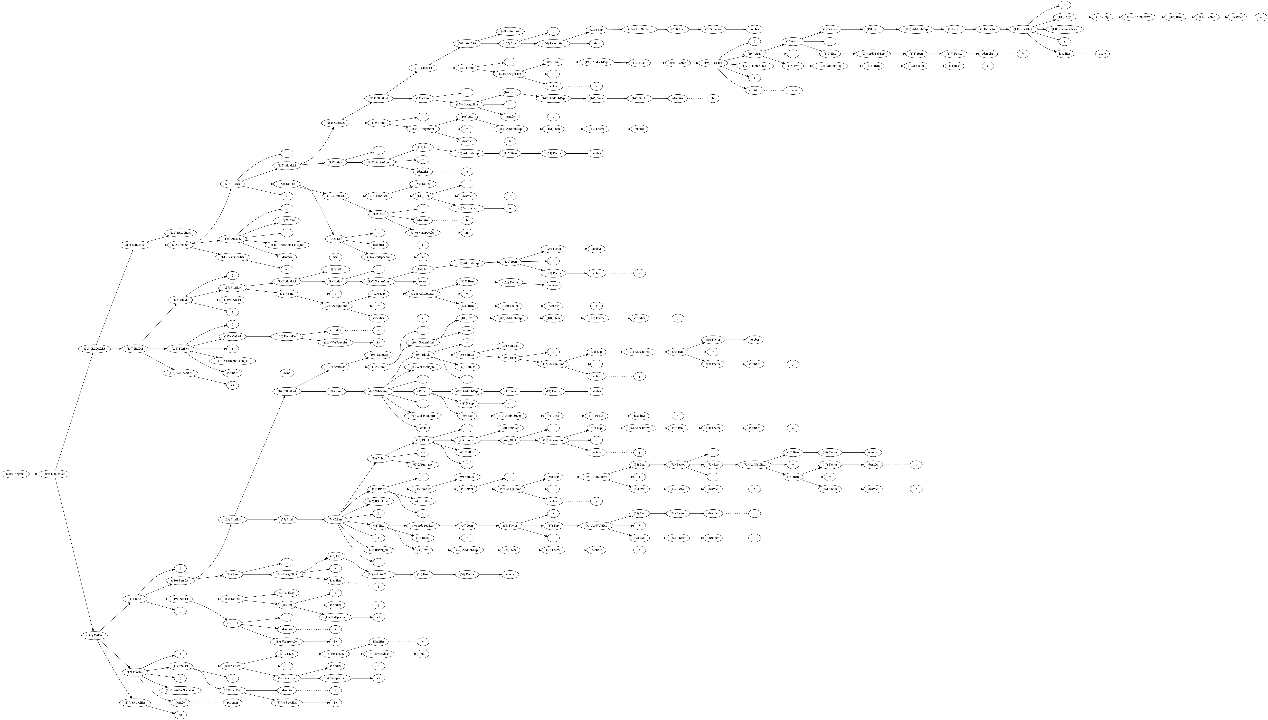
输出的中间结果文件列表如下：

图形用户界面, 应用程序

描述已自动生成

|  |  |
| --- | --- |
| **文件名** | **文件内容** |
| **Lexical\_AnalysisAnswer**  存放词法分析结果 |  |
| **Grammatical\_Grammer**  存放读入的文法表示 |  |
| **Grammatical\_LR1Table**  存放语法分析LR1表 |  |
| **syntaxTree.dot**  语法树节点文件 |  |
| **syntaxTree.png**  语法树图像 | 见表下（此处空间过小不便展示） |
| **Semantic\_Quaternary**  词法分析结果（四元式） |  |
| **Assembly.asm**  输出的汇编代码 |  |

语法树图像（syntaxTree.png）：



## **5.3 结果展示**

### **5.3.1 正确程序结果展示**

5.1模块中已经展示正确程序的编译过程

### **5.3.2 错误程序结果展示**

* **变量重定义**

文本

描述已自动生成

* **变量未定义**

文本

中度可信度描述已自动生成

* **函数调用参数不匹配**

文本

低可信度描述已自动生成

# **六、总结与体会**

本项目使用C++语言实现了一个类C语言编译器，包括了词法分析器、语法分析器、语义分析器/中间代码生成器和目标代码生成器等模块。通过这个作业，我对编译原理的知识有了更深入的理解，并且提升了我C语言面向对象的编程能力。

首先是词法分析器。词法分析器能够识别标准C语言的词法符号，并且能够过滤掉空格、制表符和换行符等无关字符。它还支持注释的识别和错误提示功能。

接下来实现了语法分析器和语义分析器。语法分析器使用LR1文法分析方法，支持函数和变量的声明、定义和表达式语句等语法结构。语义分析器则负责生成中间代码，将输入的语句转换成四元式的形式。在语义分析阶段，我还实现了对变量的定义和重复定义的检查功能。

最后实现的是目标代码生成器，将优化后的四元式序列转化为目标代码（汇编语言）。在目标代码生成阶段，我还对优化后的四元式序列进行了进一步的优化，以生成执行汇编语言文件并输出运算结果。

通过完成这个编译原理课程设计作业，我不仅加深了对编译原理的理解，还提升了我的编程能力。我学会了如何设计和实现一个简单的类C语言编译器，掌握了词法分析、语法分析、语义分析、中间代码生成和目标代码生成等关键技术。这个作业让我更加熟悉了C++语言的使用，并且培养了我解决问题的能力。

# **七、附件 - 源代码**

## **7.1 头文件**

### **7.1.1 1\_Lexical\_Analysis.h**

#pragma once

#include <iostream>

#include <string>

using namespace std;

// 用来描述词法分析结果(tokens)的数据结构

struct TOKEN

{

string token\_key; // 类型：保留字、标识符、number...

string value; // 值

int row, col; // 源文件中的位置，用于出错处理的定位

};

### **7.1.2 1\_Lexical\_PreProcess.h**

#pragma once

#include <iostream>

#include <vector>

#include <set>

using namespace std;

// 类C语言文法的产生式

struct productItem {

string left\_symbol; // 产生式左边的symbol

vector<string> right\_symbol;// 产生式右边的symbol

};

// 类C语言文法类

class CGrammer {

public:

vector<productItem> productItems; // 所有产生式

int startItemPos; // 起始产生式位置

set<string> Terminals; // 所有终结符

set<string> NTterminals; // 所有非终结符

// 特殊符号定义

const char DERI\_SIGN = ':'; // 推导符

const char SEPE\_SIGN = '\\';// 分隔符

const char AND\_SIGN = ' '; // 候选式内部连接符

const char EXEG\_SIGN = '$'; // 注释符

const char INIT\_SIGN = 'S'; // 扩展文法起始符

const char NULL\_SIGN = '@'; // 空串符

CGrammer();

void readGrammer(string filename);

void printGrammer(string filename);

};

### **7.1.3 2\_Grammatical\_Analysis.h**

#pragma once

#include <iostream>

#include <vector>

#include <string>

#include <set>

#include <map>

#include "1\_Lexical\_PreProcess.h"

using namespace std;

// LR1的一个项目

class LR1\_item {

public:

string left\_symbol; // 产生式左边的symbol

vector<string> right\_symbol; // 产生式右边的symbol

int index; // 对应产生式的编号

int dotPos; // 点的位置

string seek\_symbol; // 前向搜索的一个符号

// set中find和count均支持，const关键字必不可少

bool operator==(const LR1\_item& item);

bool operator<(const LR1\_item& item) const;

};

// LR1文法

class LR1\_grammer {

public:

vector<LR1\_item> allItems; // 所有项目

vector<set<LR1\_item>>closureFamily; // 项目集族

map<string, set<string>>FIRST; // FIRST集

map<pair<int, string>, pair<string, int>> LR1\_table;// LR1分析表

CGrammer grammer; // 文法

void printItems(set<LR1\_item> inItems);//打印项目

void generateFirst(); // 求所有文法符号的FIRST集

void printFirst(); // 打印first集

set<string> getStrFirstSet(vector<string> str); // 求一个文法符号串的FIRST集

void getClosure(set<LR1\_item> inItems, set<LR1\_item>& outItems);

bool isSameClosure(set<LR1\_item>closure1, set<LR1\_item>closure2);

void generateClosureFamily(); // 生成项目集族

// 生成下一个项目集（状态）

void generateNextClosure(set<LR1\_item> fromItems, set<LR1\_item>& toItems,

string presentStr);

void fillGuiYueTable(); // 求出项目集后，调用该函数填规约项目

void printLR1Table(string filename);

};

### **7.1.4 2\_Grammatical\_LR1Grammer.h**

#pragma once

#include <iostream>

#include <vector>

#include <string>

#include <set>

#include <map>

#include "1\_Lexical\_PreProcess.h"

using namespace std;

// LR1的一个项目

class LR1\_item {

public:

string left\_symbol; // 产生式左边的symbol

vector<string> right\_symbol; // 产生式右边的symbol

int index; // 对应产生式的编号

int dotPos; // 点的位置

string seek\_symbol; // 前向搜索的一个符号

// set中find和count均支持，const关键字必不可少

bool operator==(const LR1\_item& item);

bool operator<(const LR1\_item& item) const;

};

// LR1文法

class LR1\_grammer {

public:

vector<LR1\_item> allItems; // 所有项目

vector<set<LR1\_item>>closureFamily; // 项目集族

map<string, set<string>>FIRST; // FIRST集

map<pair<int, string>, pair<string, int>> LR1\_table;// LR1分析表

CGrammer grammer; // 文法

void printItems(set<LR1\_item> inItems);//打印项目

void generateFirst(); // 求所有文法符号的FIRST集

void printFirst(); // 打印first集

set<string> getStrFirstSet(vector<string> str); // 求一个文法符号串的FIRST集

void getClosure(set<LR1\_item> inItems, set<LR1\_item>& outItems);

bool isSameClosure(set<LR1\_item>closure1, set<LR1\_item>closure2);

void generateClosureFamily(); // 生成项目集族

// 生成下一个项目集（状态）

void generateNextClosure(set<LR1\_item> fromItems, set<LR1\_item>& toItems,

string presentStr);

void fillGuiYueTable(); // 求出项目集后，调用该函数填规约项目

void printLR1Table(string filename);

};

### **7.1.5 3\_Semantic\_Analysis.h**

#pragma once

#include <iostream>

#include <string>

#include <vector>

#include <stack>

using namespace std;

//错误码

#define FILE\_OPEN\_ERROE 1

#define LEXICAL\_ERROR\_UNDEFINED\_WORD 2

#define GRAMMAR\_ERROR 3

#define GRAMMATICAL\_ERROR\_UNDEFINED\_WORD 4

#define GRAMMATICAL\_ERROR\_CANNOT\_ANALYSIS 5

#define SEMANTIC\_ERROR\_NO\_MAIN 6

#define SEMANTIC\_ERROR\_REDEFINED 7

#define SEMANTIC\_ERROR\_UNDEFINED 8

#define SEMANTIC\_ERROR\_PARAMETER\_NUM 9

//#define SEMANTIC\_ERROR\_UNDEFINED 10

#define SEMANTIC\_ERROR\_NO\_RETURN 11

// 语义分析符号

struct SemanticSymbol

{

string token; // 符号类型

string value; // 值

int row; // 源程序中的位置

int col;

int tableIdx; // 符号所在表的索引

int idx; // 符号在表中的索引

};

// 标识符相关信息，包括函数、变量、临时变量、常量的具体信息

struct IdentifierInfo

{

// 标识符类型:函数、变量、临时变量、常量、返回值

enum idType { FUN, VAR, TMP, CST, RET };

idType idType; // 标识符的类型

string specifierType; // 变(常)量类型/函数返回类型

string identifierName; // 标识符名称/常量值

int funcParameterNum; // 函数参数个数

int funcEntry; // 函数入口地址(四元式的标号)

int funcTableIdx; // 函数的函数符号表在整个程序的符号表列表中的索引

};

// 符号表

struct SemanticSymbolTable

{

// 几种表的类型，分别为全局表、函数表、块级表、临时表

enum TableType { GlobalTable, FunctionTable, BlockTable, TempTable };

TableType table\_type; // 表的类型

vector<IdentifierInfo> table; // 符号表

string table\_name; // 表名

// 构造函数

SemanticSymbolTable(const TableType type, const string name);

// 根据变量名字寻找一个变量 没找到则返回-1，否则返回变量在表中的标号

int FindSymbol(const string name);

// 加入一个变量，返回加入的位置，若之前已经加入则返回-1。

int AddSymbol(const IdentifierInfo id);

};

// 四元式

struct Quatenary

{

int idx; // 四元式标号

string operatorType; // 操作类型

string op1; // 操作数1

string op2; // 操作数2

string result; // 结果

};

// 语义分析

class SemanticAnalysis

{

public:

vector<Quatenary> quaternary; // 四元式

int mainIdx; // main函数对应的四元式标号

int backpatchLevel; // 回填层次

vector<int> backpatchList; // 回填列表

int nextQuaternaryIdx; // 下一个四元式标号

int tmpVarCount; // 临时变量计数

vector<SemanticSymbol> symbolList; // 语义分析过程的符号流

vector<SemanticSymbolTable> tables; // 程序所有符号表

vector<int> currentTableStack; // 当前作用域对应的符号表索引栈

SemanticAnalysis();

void AddSymbol2List(const SemanticSymbol symbol);

void Analysis(const string production\_left, const vector<string> production\_right);

void PrintQuaternary(const string file\_path);

private:

//语义分析函数

void Semantic\_Analysis\_Program(const string production\_left, const vector<string> production\_right);

void Semantic\_Analysis\_ExtDef(const string production\_left, const vector<string> production\_right);

void Semantic\_Analysis\_VarSpecifier(const string production\_left, const vector<string> production\_right);

void Semantic\_Analysis\_FunSpecifier(const string production\_left, const vector<string> production\_right);

void Semantic\_Analysis\_FunDec(const string production\_left, const vector<string> production\_right);

void Semantic\_Analysis\_CreateFunTable\_m(const string production\_left, const vector<string> production\_right);

void Semantic\_Analysis\_ParamDec(const string production\_left, const vector<string> production\_right);

void Semantic\_Analysis\_Block(const string production\_left, const vector<string> production\_right);

void Semantic\_Analysis\_Def(const string production\_left, const vector<string> production\_right);

void Semantic\_Analysis\_AssignStmt(const string production\_left, const vector<string> production\_right);

void Semantic\_Analysis\_Exp(const string production\_left, const vector<string> production\_right);

void Semantic\_Analysis\_AddSubExp(const string production\_left, const vector<string> production\_right);

void Semantic\_Analysis\_Item(const string production\_left, const vector<string> production\_right);

void Semantic\_Analysis\_Factor(const string production\_left, const vector<string> production\_right);

void Semantic\_Analysis\_CallStmt(const string production\_left, const vector<string> production\_right);

void Semantic\_Analysis\_CallFunCheck(const string production\_left, const vector<string> production\_right);

void Semantic\_Analysis\_Args(const string production\_left, const vector<string> production\_right);

void Semantic\_Analysis\_ReturnStmt(const string production\_left, const vector<string> production\_right);

void Semantic\_Analysis\_Relop(const string production\_left, const vector<string> production\_right);

void Semantic\_Analysis\_IfStmt(const string production\_left, const vector<string> production\_right);

void Semantic\_Analysis\_IfStmt\_m1(const string production\_left, const vector<string> production\_right);

void Semantic\_Analysis\_IfStmt\_m2(const string production\_left, const vector<string> production\_right);

void Semantic\_Analysis\_IfNext(const string production\_left, const vector<string> production\_right);

void Semantic\_Analysis\_IfStmt\_next(const string production\_left, const vector<string> production\_right);

void Semantic\_Analysis\_WhileStmt(const string production\_left, const vector<string> production\_right);

void Semantic\_Analysis\_WhileStmt\_m1(const string production\_left, const vector<string> production\_right);

void Semantic\_Analysis\_WhileStmt\_m2(const string production\_left, const vector<string> production\_right);

};

### **7.1.6 4\_Assembly\_Generator.h**

#pragma once

#include <fstream>

#include <string>

#include <vector>

#include <set>

#include <map>

#include "3\_Semantic\_Analysis.h"

#include "4\_Register\_Info.h"

#define REGS\_NUMBER 32 // 寄存器总数

#define DATA\_SEG\_ADDR 0x10010000 // 数据段的起始地址

#define STACK\_SEG\_ADDR 0x10040000 // 堆栈段的起始地址

#define LOCAL\_VAR\_OFFSET -8 // 局部变量在栈中相对于fp的偏移量（与栈帧的设计有关）

// 需要注意的是：生成汇编代码的过程是从头开始的，对于汇编代码而言，是无法看到调用层次与关系的

// 这与中间代码生成部分有很大的不同

using namespace std;

class Assembly\_Generator {

private:

ofstream Asm\_stream; // 汇编文件输出流

vector<Quatenary> quatenary; // 存储中间代码四元式

set<string>J\_DestinationAddress; // 存储四元式中需要跳转的目标地址，用于后续加标签

Reg RegisterInfo[REGS\_NUMBER]; // 寄存器信息

string RVALUE[REGS\_NUMBER]; // RVALUE: 寄存器存储的变量名

map<string, int> AVALUE; // AVALUE: 变量所在的一系列寄存器(-1:被换至内存)

map<string, int>localVariantAddressTable; // 局部变量地址表

map<string, int>globalVariantAddressTable; // 全局变量地址表

int localVarOffset = -4; // 局部变量地址偏移指针

int globalVarAddr = DATA\_SEG\_ADDR; // 全局变量地址指针

int paramsNumber = 2; // 调用参数个数，用于call时设置栈帧返回位移

string procedureName; // 过程块名

void callTypeHandler(Quatenary quat); // 函数调用

void returnTypeHandler(Quatenary quat); // 返回

void jTypeHandler(Quatenary quat); // 跳转（包括j j< j> j= j<= j>=）

void computeTypeHandler(Quatenary quat);// 计算（包括+ - \* /）

void assignTypeHandler(Quatenary quat); // 赋值

void paramTypeHandler(Quatenary quat); // 形参声明

void defparTypeHandler(Quatenary quat); // 实参声明

void defprocTypeHandler(Quatenary quat);// 过程块声明

// 分配一个寄存器给变量A（A := B op C）

int allocateReg(string A, string B = "-", string C = "-");

void saveVar(string var); // 将变量var存入内存

// 选一个寄存器，将变量var放到寄存器中，指定不考虑except1和except2

int loadVar(string var, int except1 = -1, int except2 = -1);

// 通过LRU算法找到最远未被使用的寄存器，指定不考虑except1和except2

int LRU\_GetRegister(int except1, int except2);

public:

Assembly\_Generator(string, const vector<Quatenary>);

~Assembly\_Generator();

void Print\_Assembly(string path);

void generate();

};

### **7.1.7 4\_Register\_Info.h**

#include <string>

using namespace std;

//记录Reg的名称、编号、未使用时间

struct Reg

{

int index = 0; // 编号

string name; // 名称

int unusedTime = 0; // 未使用时间

bool operator < (const Reg& b) {

return unusedTime < b.unusedTime;

};

};

## **7.2 源文件**

### **7.2.1 1\_Lexical\_Analysis.cpp**

#include <iostream>

#include <map>

#include <fstream>

#include <string>

#include <vector>

#include "1\_Lexical\_Analysis.h"

using namespace std;

vector<TOKEN>lexAnswer; // 1、词法分析器的结果

// path为C语言源程序文件路径

void lexAnalysis(const string path); // 调用接口，调用本函数进行词法分析

void init(); // 加载保留字

//string preProcessing(string src); // 源程序预处理

int findReservedWord(string word); // 查找保留字并返回其编码，未找到返回-1

void scanner(string dst, int& pointer, ofstream& outfile); // 扫描器

void readRealNumber(string src, int& pointer, string& dst, int& col); // 数字读取

// 保留字

string reservedWord[] = { "(", ")", "{", "}", ",", ";", "\"",

"=", "+", "-", "\*", "/", "+=", "-=", "\*=", "/=", "%=", "^=", "&=", "|=",

">", "<", ">=", "<=", "==", "!=", "||", "&&",

"else", "if", "int", "float", "double", "return", "void", "while" };

map<string, int>wordMap; // 保留字与其编码的哈希关联

void lexAnalysis(const string path) {

// --------------------

// 读取源程序

// --------------------

ifstream infile;

ofstream outfile;

infile.open(path, ios::in);

if (!infile.is\_open()) {

cout << "文件打开失败" << endl;

exit(-1);

}

string src;

//将文件内容全部写入字符串中

src.assign(std::istreambuf\_iterator<char>(infile), std::istreambuf\_iterator<char>());

// ------------------------

// 扫描输出单词的二元组列表

// ------------------------

init(); //加载保留字

outfile.open("../outfiles/1\_Lexical\_AnalysisAnswer.txt", ios::out | ios::trunc);

if (!outfile.is\_open()) {

cout << "文件打开失败" << endl;

exit(-1);

}

int pointer = 0; // 源程序指针

while (pointer < src.length()) {

scanner(src, pointer, outfile);

if (!src[pointer]) break;

}

outfile.close();

}

// 加载保留字

void init() {

int i = 0;

for (auto word : reservedWord) {

wordMap.insert(pair<string, int>(word, ++i));

}

}

// 查找保留字并返回其编码，未找到返回-1

int findReservedWord(string word) {

auto iter = wordMap.find(word);

return iter == wordMap.end() ? -1 : wordMap[word];

}

// 扫描器，核心算法

// 输出单词种别及其属性值的二元组

void scanner(string dst, int& pointer, ofstream& outfile) {

static int row = 1;

static int col = 1;

static bool lastIsIdentifier = false; // 上一词是否为标识符，用于判断读入实常数时的“-”是负号还是减号

int coding = -1; // 单词编码工作变量

string word = ""; // 单词工作变量

string attribute = ""; // 属性工作变量

//preprocessing

do {

while (dst[pointer] == ' ' || dst[pointer] == '\r' || dst[pointer] == '\n' || dst[pointer] == '\t') {

if (dst[pointer] == ' ' || dst[pointer] == '\t') {

pointer++;

col++;

}

else {

row++;

col = 1;

pointer++;

}

if (pointer >= dst.size())

return;

}

// 删除行注释

if (dst[pointer] == '/' && ((pointer + 1) < dst.length()) && dst[pointer + 1] == '/') {

for (; pointer < dst.length() && dst[pointer] != '\r' && dst[pointer] != '\n'; pointer++)

col++;

if (pointer >= dst.length())//末尾则返回

return;

else {//换行则修改row col

row++;

col = 1;

pointer++;

}

}

if (dst[pointer] == '/' && ((pointer + 1) < dst.length()) && dst[pointer + 1] == '\*') {

for (; pointer < dst.length() && !(dst[pointer - 1] == '\*' && dst[pointer] == '/'); pointer++) {

if (dst[pointer] == '\r' || dst[pointer] == '\n') {

row++;

col = 1;

}

else

col++;

}

if (pointer == dst.length()) {

cerr << "源程序错误！没有找到\"\*/\"!";

}

pointer++;

}

} while ((dst[pointer] == ' ' || dst[pointer] == '\r' || dst[pointer] == '\n')

|| (dst[pointer] == '/' && ((pointer + 1) < dst.length()) && dst[pointer + 1] == '/')

|| (dst[pointer] == '/' && ((pointer + 1) < dst.length()) && dst[pointer + 1] == '\*'));

//预处理完毕，开始记号识别

if (isalpha(dst[pointer]) || isdigit(dst[pointer]) || dst[pointer] == '\_') { // 数字、字母、下划线 开头

if (isalpha(dst[pointer]) || dst[pointer] == '\_') { // 字母、下划线 开头

while (isalpha(dst[pointer]) || isdigit(dst[pointer]) || dst[pointer] == '\_') {

word += dst[pointer++]; //收集，然后下移

col++;

}

coding = findReservedWord(word); // 查表，若是保留字，返回种别码；

attribute = coding == -1 ? word : "--";

word = coding == -1 ? "identifier" : word;

}

else { //数字开头（识别实常数）

word = "number";

readRealNumber(dst, pointer, attribute, col);

}

}

else if (dst[pointer] == '\"') { // 字符串常量

pointer++;

col++;

word += '\"';

while (dst[pointer] != '\"') {

word += dst[pointer];

pointer++;

col++;

}

word += '\"';

attribute = word;

word = "const string";

pointer++;

col++;

}

else { // 开头为其它字符

word = dst[pointer];

word += dst[pointer + 1]; // 超前搜索

if ((coding = findReservedWord(word)) != -1) { // || != <= == >=

pointer += 2;

col += 2;

attribute = "--";

}

else if (!lastIsIdentifier && (word[0] == '-' || word[0] == '.') && isdigit(word[1])) // 说明此处“-”为负号

{

word = "number";

readRealNumber(dst, pointer, attribute, col);

}

else {

word = dst[pointer];

if ((coding = findReservedWord(word)) != -1) { // ( ) \* + , - / ; < >

pointer += 1;

col += 1;

}

else

{

cout << "ERROR: unexpected word " << word << endl;

exit(EXIT\_FAILURE);

}

attribute = "--";

}

}

lastIsIdentifier = (attribute == "--") ? false : true;

outfile << "< " << word << " , " << attribute << " >" << " ROW: " << row << " COL: " << col << endl;

TOKEN token = { word,attribute,row,col };//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*TODO:实现行列计数

lexAnswer.push\_back(token);

}

// 读取实常数的函数

// 实常数的类型：

// 十进制数 / 八进制数：0开头 / 十六进制数：0x开头

// 小数：有且仅有一个小数点

// 指数：有且仅有一个e，且e前必须有数，e后必须整数

// 注：若为小数或指数，则前导0无效；若为整数，则前导0指示八进制

// 注：小数点后可以有一个e，但e后不能有小数点

// 注：若0x前导十六进制，则负号只能在0x前面

void readRealNumber(string src, int& pointer, string& dst, int& col)

{

dst = "";

if (src[pointer] == '-') {

dst += src[pointer++];

col++;

}

bool is\_decimal = true;

if (src[pointer] == '0')

{

dst += src[pointer++];

col++;

if (src[pointer] == 'x' || src[pointer] == 'X') // 16进制

{

dst += src[pointer++];

while (isdigit(src[pointer]) || (src[pointer] >= 'a' && src[pointer] <= 'f')

|| (src[pointer] >= 'A' && src[pointer] <= 'F')) {

dst += src[pointer++];

col++;

}

is\_decimal = false;

}

else // 无法确定是十进制还是八进制，因此使用tmp\_p

{

// 当数字在0-7中，对于两种进制都是合法的输入

while (src[pointer] >= '0' && src[pointer] <= '7') {

dst += src[pointer++];

col++;

}

// 当数字超出0-7的范围，此时不确定这串数字是否是十进制，因此暂存

int tmp\_p = pointer;

string dst\_tmp = dst;

while (isdigit(src[tmp\_p])) {

dst\_tmp += src[tmp\_p++];

}

// 只有遇到e或.，才能确定是十进制，否则就是八进制

if (src[tmp\_p] == 'e' || src[tmp\_p] == 'E' || src[tmp\_p] == '.')

{

is\_decimal = true;

dst = dst\_tmp;

pointer = tmp\_p;

}

else

is\_decimal = false;

}

}

if (is\_decimal)

{

while (isdigit(src[pointer])) {

dst += src[pointer++];

col++;

}

if (src[pointer] == 'e' || src[pointer] == 'E')

{

dst += src[pointer++];

col++;

if (src[pointer] == '-') {

dst += src[pointer++];

col++;

}

while (isdigit(src[pointer])) {

dst += src[pointer++];

col++;

}

}

else if (src[pointer] == '.')

{

dst += src[pointer++];

col++;

while (isdigit(src[pointer])) {

dst += src[pointer++];

col++;

}

if (src[pointer] == 'e' || src[pointer] == 'E')

{

dst += src[pointer++];

col++;

if (src[pointer] == '-') {

dst += src[pointer++];

col++;

}

while (isdigit(src[pointer])) {

dst += src[pointer++];

col++;

}

}

}

}

}

### **7.2.2 1\_Lexical\_PreProcess.cpp**

#include "1\_Lexical\_PreProcess.h"

#include <fstream>

#include <string>

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//工具函数

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//删除str两侧的空格

void delSideBlank(string& str)

{

int length = str.size();

int left = 0;

int right = length-1;

for (; left < length; left++) {

if (str[left] != ' ')

break;

}

for (; right >= 0; right--) {

if (str[right] != ' ')

break;

}

str = str.substr(left,right-left+1);

}

//将str根据seperator划分到substr中

void divideRightStr(string str, char seperator,vector<string>& substr)

{

int length = str.size();

string temp;

for (int i = 0; i < length; i++) {

if (str[i] == seperator) {

delSideBlank(temp);//去除左右两端的空格

substr.push\_back(temp);

temp.clear();

continue;

}

temp += str[i];

}

delSideBlank(temp);

substr.push\_back(temp);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//CGrammer类成员函数实现

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

CGrammer::CGrammer()

{

//empty

}

//读取文件中的文法

void CGrammer::readGrammer(string filename)

{

ifstream infile;

infile.open(filename);

if (!infile) {

cout << "文件打开失败!" << endl;

exit(-1);

}

string line, left, right;

vector<string> left\_and\_right;

while (getline(infile, line, '\n')) {

//消除注释

if (line[0] == EXEG\_SIGN || line.size() == 0) {

line.clear();

continue;

}

right.clear();

left.clear();

left\_and\_right.clear();

divideRightStr(line, DERI\_SIGN, left\_and\_right); //分为左右两部分

if (left\_and\_right.size() != 2) {

cout << "产生式格式不正确!" << endl;

exit(-1);

}

left = left\_and\_right[0];

right = left\_and\_right[1];

//读取终结符

if (left == "%VT") {

vector<string> temp;

divideRightStr(right, SEPE\_SIGN, temp);

for (vector<string>::iterator iter = temp.begin(); iter < temp.end(); iter++)

{

if (\*iter != "")

Terminals.insert(\*iter);

}

}

//读取产生式

else {

vector<string> temp;

divideRightStr(right, SEPE\_SIGN, temp);

int len = temp.size();//产生式个数

for (int i = 0; i < len; i++) {

productItem tempItem;

tempItem.left\_symbol = left;

divideRightStr(temp[i], AND\_SIGN, tempItem.right\_symbol);

productItems.push\_back(tempItem);//插入一个产生式

if (left.size() == 1 && left[0] == INIT\_SIGN)

startItemPos = productItems.size() - 1;//起始产生式的角标

}

//插入非终结符

NTterminals.insert(left);

}

line.clear();

}

infile.close();

}

//打印文法到文件中，测试用

void CGrammer::printGrammer(string filename) {

ofstream outfile;

outfile.open(filename);

if (!outfile) {

cout << "文件打开失败!" << endl;

exit(-1);

}

outfile << "起始产生式位置：" << this->startItemPos << endl;

outfile << "终结符集 = { ";

for (auto str : this->Terminals) {

outfile << str << " ";

}

outfile << "}" << endl;

outfile << "非终结符集 = { ";

for (auto str : this->NTterminals) {

outfile << str << " ";

}

outfile << "}" << endl;

outfile << "产生式：" << endl;

int i = 0;

for (auto pItem : this->productItems) {

outfile << i << "：";

i++;

outfile << pItem.left\_symbol << "-->";

for (auto str : pItem.right\_symbol) {

outfile << str << " ";

}

outfile << endl;

}

}

### **7.2.3 2\_Grammatical\_Analysis.cpp**

#include "2\_Grammatical\_Analysis.h"

#include "2\_Grammatical\_LR1Grammer.h"

#include <stack>

// extern void deleteThreadedList(ThreadedList\* Root);

//extern string START\_SIGN;

// 传入引用

ThreadedList\* grammaticalAnalysis(vector<ANSWER\_TYPE>& Answer, map<TABLE\_KEY\_TYPE, TABLE\_VALUE\_TYPE>& Table, vector<ITEM\_TYPE>& Items, SemanticAnalysis& Semantic)

{

//定义语义分析

Semantic = SemanticAnalysis();

Semantic.AddSymbol2List({ "Program","",-1,-1,-1,-1 });

TOKEN temp = { "#", "--", -1, -1 };

Answer.push\_back(temp);

vector<int>statusStack; // 状态栈

vector<string>symbolsStack; // 符号栈

vector<ThreadedList\*>threadNodeStack; // 穿线表栈，用于绘图

int answerPointer = 0; // 字符串指针

int regulationStepCount = 0; // 当前进行到第几次规约

TABLE\_VALUE\_TYPE tableValue; // 查Table获取的数据存到这里

string action; // 下一步骤，即tableValue.first

statusStack.push\_back(0); // 初始状态为状态0

string symbol;

while (true)

{

symbol = Answer[answerPointer].token\_key;

pair<int, string>cur(statusStack.back(), symbol); // 构造map索引

tableValue = Table[cur]; // 查action表，获取下一步动作

string action = tableValue.first;

if (action == "acc") // 规约成功

{

if (threadNodeStack.size() == 1)

return threadNodeStack[0];

else

{

deleteThreadedList(threadNodeStack[0]);

return nullptr;

}

}

else if (action == "") // 规约失败

{

deleteThreadedList(threadNodeStack[0]);

return nullptr;

}

else if (action == "s") // 移进

{

string curSymbol = Answer[answerPointer].token\_key;

symbolsStack.push\_back(curSymbol); // 将输入串当前字符压入符号栈，并移动指针

statusStack.push\_back(tableValue.second);

ThreadedList\* Node = new(nothrow)ThreadedList; // 构造新的穿线表结点

if (!Node)

{

deleteThreadedList(threadNodeStack[0]);

exit(OVERFLOW);

}

if (curSymbol == "identifier") // 由于其是终结符，因此不用在规约时再考虑之

Node->IdIfExists = Answer[answerPointer].value; // 用于绘制语法树

Node->symbol = curSymbol;

threadNodeStack.push\_back(Node);

//将token加入语义分析的符号串中

Semantic.AddSymbol2List({ Answer[answerPointer].token\_key,Answer[answerPointer].value,Answer[answerPointer].row,Answer[answerPointer].col,-1,-1 });

answerPointer++;

}

else if (action == "r") // 规约

{

++regulationStepCount; // 规约次数+1

int regulationTerm = tableValue.second; // 选定规约项

string regulationLeft = Items[regulationTerm].left\_symbol; // 规约左部，唯一

int regulationRightLength = Items[regulationTerm].right\_symbol.size(); // 右部长度

if (statusStack.size() < regulationRightLength)

{

cout << "ERROR: NOT ENOUGH STATUS!" << endl;

deleteThreadedList(threadNodeStack[0]);

exit(EXIT\_FAILURE);

}

if (Items[regulationTerm].right\_symbol[0] == "@") // 空串特判

regulationRightLength = 0;

ThreadedList\* Node = new(nothrow)ThreadedList(regulationStepCount); // 构造新的穿线表结点，此结点为当前规约使用的产生式左部

if (!Node)

{

deleteThreadedList(threadNodeStack[0]);

exit(OVERFLOW);

}

Node->symbol = regulationLeft;

for (int i = 0; i < regulationRightLength; ++i) // 弹出待规约的右部并连接穿线表

{

symbolsStack.pop\_back();

statusStack.pop\_back();

Node->threads.push\_back(threadNodeStack.back());

threadNodeStack.pop\_back();

}

symbolsStack.push\_back(regulationLeft); // 压入左部

threadNodeStack.push\_back(Node); // 将连接好的非终结符结点压回穿线表结点栈

cur = pair<int, string>(statusStack.back(), symbolsStack.back()); // 构造map索引

tableValue = Table[cur]; // 查表(goto)，获取待压入的状态

statusStack.push\_back(tableValue.second);

//语义分析

//cout <<"Semantic.Analysis: "<< Items[regulationTerm].left\_symbol << endl;

Semantic.Analysis(Items[regulationTerm].left\_symbol, Items[regulationTerm].right\_symbol);

}

else // 不明字符

{

cout << "ERROR: UNKNOWN OPERATION OF LR1TABLE!" << endl;

deleteThreadedList(threadNodeStack[0]);

exit(EXIT\_FAILURE);

}

}

}

// 删除穿线表

void deleteThreadedList(ThreadedList\* Root)

{

// 非递归遍历，使用栈即可

stack<ThreadedList\*>threadedListStack;

threadedListStack.push(Root);

while (!threadedListStack.empty())

{

ThreadedList\* Node = threadedListStack.top();

threadedListStack.pop();

for (int i = 0; i < Node->threads.size(); ++i)

threadedListStack.push(Node->threads[i]);

delete Node;

}

}

### **7.2.4 2\_Grammatical\_LR1Grammer.cpp**

#include "2\_Grammatical\_LR1Grammer.h"

#include <fstream>

#include <iomanip>

bool LR1\_item::operator==(const LR1\_item& item)

{

if (this->left\_symbol != item.left\_symbol) return false;

if (this->dotPos != item.dotPos)return false;

if (this->seek\_symbol != item.seek\_symbol)return false;

if (this->right\_symbol.size() != item.right\_symbol.size())return false;

for (int i = 0; i < this->right\_symbol.size(); i++) {

if (this->right\_symbol[i] != item.right\_symbol[i])

return false;

}

return true;

}

bool LR1\_item::operator<(const LR1\_item& item) const

{

if (this->left\_symbol != item.left\_symbol)

return this->left\_symbol < item.left\_symbol;

else {

if (this->right\_symbol.size() != item.right\_symbol.size())

return this->right\_symbol.size() < item.right\_symbol.size();

else {

for (int i = 0; i < this->right\_symbol.size(); i++) {

if (this->right\_symbol[i] != item.right\_symbol[i])

return this->right\_symbol[i] < item.right\_symbol[i];

}

}

if (this->seek\_symbol != item.seek\_symbol)

return this->seek\_symbol < item.seek\_symbol;

else

return this->dotPos < item.dotPos;

}

}

//打印项目集

void LR1\_grammer::printItems(set<LR1\_item> inItems) {

for (auto pitem : inItems) {

cout << pitem.left\_symbol << " -> ";

int i = 0;

for (auto r : pitem.right\_symbol) {

if (i++ == pitem.dotPos)

cout << "·";

cout << r << " ";

}

if(pitem.dotPos==pitem.right\_symbol.size())

cout << " · ";

cout << " --- " << pitem.seek\_symbol << endl;

}

cout << "/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/" << endl;

}

// 判断是否为两个相同的闭包

bool LR1\_grammer::isSameClosure(set<LR1\_item>closure1, set<LR1\_item>closure2) {

if (closure1.size() != closure2.size())

return false;

int count = 0;

for (auto tmp : closure1) {

for (auto tmp\_ : closure2) {

if (tmp == tmp\_) {

++count;

break;

}

}

}

return count == closure1.size();

}

// 生成下一个项目集（状态）

// 求当前输入符号为presentStr时，状态fromItems转去的状态toItems

void LR1\_grammer::generateNextClosure(set<LR1\_item> fromItems, set<LR1\_item>& toItems, string presentStr) {

if (presentStr == "#") // 不考虑#

return;

for (auto lItem : fromItems) { // 扫描项目集fromItems的每一个项目

if (lItem.dotPos >= lItem.right\_symbol.size()) // dot在末尾

continue;

if (lItem.right\_symbol[lItem.dotPos] != presentStr) // dot后一个符号不是presentStr

continue;

// dot 后是presentStr

LR1\_item temp = lItem;

temp.dotPos = lItem.dotPos + 1; // dot后移一位

toItems.insert(temp); // 把该项目插入到新的项目集

}

getClosure(toItems, toItems); // 求新项目集的闭包

}

// 生成项目集族

void LR1\_grammer::generateClosureFamily() {

set<LR1\_item> startClosure; // 初始项目集

LR1\_item startItem; // 初始项目

productItem startProductItem; // 初始产生式

startProductItem = this->grammer.productItems[this->grammer.startItemPos];

// 根据初始产生式构造初始项目

startItem.left\_symbol = startProductItem.left\_symbol;

for (auto str : startProductItem.right\_symbol) {

startItem.right\_symbol.push\_back(str);

}

startItem.dotPos = 0;

startItem.seek\_symbol = "#";

startClosure.insert(startItem); // 将初始项目加入到初始项目集

this->getClosure(startClosure, startClosure); // 求初始项目的闭包

closureFamily.push\_back(startClosure); // 将初始项目集的闭包加入项目集族

// 遍历项目集族中的每一个项目集

for (int i = 0; i < closureFamily.size(); i++) {

// 对每个项目集 遍历所有终结符

for (auto str : this->grammer.Terminals) {

if (str == "#") // 显然不需要考虑 #

continue;

set<LR1\_item> tempClosure; // 输入str，当前状态会转移到下一个项目集tempClosure

generateNextClosure(closureFamily[i], tempClosure, str); // 生成下一个项目集的闭包

if (tempClosure.empty()) // 下一个项目集为空，说明当前项目集不存在遇到str的转移

continue;

// 检查项目集族中是否已经存在相同的状态

int existIndex = -1;

for (int j = 0; j < closureFamily.size(); j++) {

if (isSameClosure(closureFamily[j], tempClosure)) {

existIndex = j;

break;

}

}

if (existIndex != -1) { // 存在相同状态则不需要插入

pair<int, string>in = pair<int, string>(i, str);

if (LR1\_table.count(in) == 0) { // 检查Action表的对应移进动作是否有冲突

// 填写LR1分析表中的移进动作

LR1\_table[pair<int, string>(i, str)] = pair<string, int>("s", existIndex);

}

else

cout << "ERROR:入口冲突!";

}

else { // 不存在相同状态则需要插入

closureFamily.push\_back(tempClosure);// 将下一个项目集的闭包加入项目集族

pair<int, string>in = pair<int, string>(i, str);

if (LR1\_table.count(in) == 0) { // 检查Action表的对应移进动作是否有冲突

// 填写LR1分析表中的移进动作

LR1\_table[pair<int, string>(i, str)] = pair<string, int>("s", closureFamily.size() - 1);

}

else

cout << "ERROR:入口冲突!";

}

}

// 对每个项目集 遍历所有非终结符

for (auto str : this->grammer.NTterminals) {

set<LR1\_item> tempClosure; // 输入str，当前状态会转移到下一个项目集tempClosure

generateNextClosure(closureFamily[i], tempClosure, str); // 生成下一个项目集的闭包

if (tempClosure.empty()) // 下一个项目集为空，说明当前项目集不存在遇到str的转移

continue;

// 检查项目集族中是否已经存在相同的状态

int existIndex = -1;

for (int j = 0; j < closureFamily.size(); j++) {

if (isSameClosure(closureFamily[j], tempClosure)) {

existIndex = j;

break;

}

}

if (existIndex != -1) { // 存在相同状态则不需要插入

pair<int, string>in = pair<int, string>(i, str);

if (LR1\_table.count(in) == 0) { // 检查GOTO表的对应转移动作是否有冲突

// 填写LR1分析表中的移进动作

LR1\_table[pair<int, string>(i, str)] = pair<string, int>("null", existIndex);

}

else

cout << "ERROR:入口冲突!";

}

else { // 不存在相同状态则需要插入

closureFamily.push\_back(tempClosure); // 将下一个项目集的闭包加入项目集族

pair<int, string>in = pair<int, string>(i, str);

if (LR1\_table.count(in) == 0) { // 检查GOTO表的对应转移动作是否有冲突

// 填写LR1分析表中的移进动作

LR1\_table[pair<int, string>(i, str)] = pair<string, int>("null", closureFamily.size() - 1);

}

else

cout << "ERROR:入口冲突!";

}

}

}

}

// 求所有文法符号的FIRST集

void LR1\_grammer::generateFirst() {

// 求终结符的first集

for (auto str : grammer.Terminals) {

// 终结符的FIRST集是它自身

set<string> firstSet;

firstSet.insert(str);

this->FIRST[str] = firstSet;

}

// 求非终结符的first集

bool changed;

while (true) {

changed = false;

// 遍历所有非终结符

for (auto ntStr : grammer.NTterminals) {

// 遍历所有产生式

for (auto pItem : grammer.productItems) {

// 如果产生式左边不为当前非终结符str，则跳过此产生式

if (pItem.left\_symbol != ntStr)

continue;

// 找到可用产生式，遍历产生式右部

int pos = 0;

// 产生式右部第一个符号是终结符（包括epsilon）

if (grammer.Terminals.find(pItem.right\_symbol[pos]) != grammer.Terminals.end()) {

changed = this->FIRST[ntStr].insert(pItem.right\_symbol[pos]).second || changed;

continue;

}

// 产生式右部以非终结符开始

bool flag = true; // 可推导出空串的标志

for (; pos < pItem.right\_symbol.size(); pos++) {

// 如果是终结符，则停止迭代

if (grammer.Terminals.find(pItem.right\_symbol[pos]) != grammer.Terminals.end()) {

flag = false;

changed = this->FIRST[ntStr].insert(pItem.right\_symbol[pos]).second || changed;

break;

}

// 是非终结符，则把该非终结符除epsilon以外的first集合并到first[ntStr]

set<string> temp = this->FIRST[pItem.right\_symbol[pos]];

temp.erase("@"); //删除epsilon

int size1 = this->FIRST[ntStr].size();

this->FIRST[ntStr].insert(temp.begin(), temp.end());

int size2 = this->FIRST[ntStr].size();

changed = size2 > size1 ? (changed || 1) : (changed || 0); // first集是否发生变化

// 该非终结符是否能推出空串，如是，则继续迭代

flag = flag && this->FIRST[pItem.right\_symbol[pos]].find("@") != this->FIRST[pItem.right\_symbol[pos]].end();

if (!flag)

break;

}

// 产生式右部所有符号都可以推出非终结符，则first集插入Epsilon

if (flag && pos == pItem.right\_symbol.size())

changed = this->FIRST[ntStr].insert("@").second || changed;

}

}

//如果没有改变，说明求first集结束，退出

if (!changed)

break;

}

}

// 求闭包

void LR1\_grammer::getClosure(set<LR1\_item> inItems, set<LR1\_item>& outItems){

// cout << "\*\*\*输入集合：" << endl;

// printItems(inItems); //////////////////////////////////////////////////////////////////////////////////////暂时注释

//闭包包含所有inItems中的项目

vector<LR1\_item> outvec;

for (auto temp : inItems)

outvec.push\_back(temp);

//outItems = inItems;

int item\_size = inItems.size();

//对inItems的每一个形如S->a.Bc ，x 的产生式

for (int pitem = 0; pitem < outvec.size();pitem++) {

//iter->right\_symbol[iter->dotPos]：点后的符号

//归约项·在末尾 直接跳过

if (outvec[pitem].dotPos == outvec[pitem].right\_symbol.size())

continue;

//如果是空项，则转为归约项目

if (outvec[pitem].right\_symbol[0][0] == grammer.NULL\_SIGN) {

outvec[pitem].dotPos++;

continue;

}

//其余项，检查·后面是不是NT

if (grammer.NTterminals.find(outvec[pitem].right\_symbol[outvec[pitem].dotPos]) != grammer.NTterminals.end()) {

vector<string> cx;

for (int i = outvec[pitem].dotPos + 1; i < outvec[pitem].right\_symbol.size(); i++) {

cx.push\_back(outvec[pitem].right\_symbol[i]);

}

cx.push\_back(outvec[pitem].seek\_symbol);

for (auto pgram : grammer.productItems) {

//寻找以pitem.right\_symbol[pitem.dotPos]为左部的产生式

if (pgram.left\_symbol == outvec[pitem].right\_symbol[outvec[pitem].dotPos]) {

//对文法productItems的每一个产生式B->b

set<string> colFirstSet = getStrFirstSet(cx);

//{cx}的FIRST集的每一个元素t，将B->b，t 加入 outItems

LR1\_item temp;

temp.left\_symbol = pgram.left\_symbol;

temp.right\_symbol.assign(pgram.right\_symbol.begin(), pgram.right\_symbol.end());

temp.dotPos = (temp.right\_symbol[0][0] == grammer.NULL\_SIGN) ? 1 : 0;

for (auto pfirst : colFirstSet) {

temp.seek\_symbol = pfirst;

//判断是否已经在outvec中

bool is\_push = 1;

int vecsize = outvec.size();

for (int i = 0; i < vecsize; i++) {

if (outvec[i] == temp) {

is\_push = 0;

break;

}

}

if (is\_push) {

outvec.push\_back(temp);

//cout << temp.left\_symbol << "->";

//int cnt = 0;

//for (auto t : temp.right\_symbol) {

// if (cnt++ == temp.dotPos)

// cout << "·";

// cout << t << " ";

//}

//if (temp.dotPos == temp.right\_symbol.size())

// cout << "·";

//cout << "---" << temp.seek\_symbol << endl;

//

}

}

}

}

}

}

for (auto i : outvec)

outItems.insert(i);

}

// 求一个文法符号串的FIRST集

// 求闭包时使用 因为不可能有@ 所以没考虑

set<string> LR1\_grammer::getStrFirstSet(vector<string> str)

{

int num = str.size();

set<string> strFirst;

if (str.empty())

return strFirst;

//bool is\_epsilon = true;

//依次遍历每一个符号

for (auto itstr : str) {

//终结符

if (grammer.Terminals.find(itstr) != grammer.Terminals.end()) {

strFirst.insert(itstr);

break;

}

//非终结符

auto it = grammer.NTterminals.find(itstr);

if (it != grammer.NTterminals.end()) {

bool is\_continue = 0;

for (auto t : FIRST[\*it]) {

if (t[0] != grammer.NULL\_SIGN)

strFirst.insert(t);

else

is\_continue = 1;

}

if (is\_continue)

continue;

else

break;

}

else {

cout << "ERROR!";

break;

}

}

return strFirst;

}

// 打印first集

void LR1\_grammer::printFirst() {

for (auto str : grammer.Terminals) {

cout << "FIRST(" << str << ") = { ";

for (auto str1 : this->FIRST[str]) {

cout << str1 << " ";

}

cout << "}" << endl;

}

for (auto str : grammer.NTterminals) {

cout << "FIRST(" << str << ") = { ";

for (auto str1 : this->FIRST[str]) {

cout << str1 << " ";

}

cout << "}" << endl;

}

}

void LR1\_grammer::fillGuiYueTable()

{

int setsize = closureFamily.size();

for (int i = 0; i < setsize; i++) {

for (auto it = closureFamily[i].begin(); it != closureFamily[i].end(); it++) {

//归约项目

if (it->right\_symbol.size() == it->dotPos) {

int gramNum = -1;

int proSize = grammer.productItems.size();

for (int t = 0; t < proSize; t++) {

auto gram = grammer.productItems[t];

if (gram.left\_symbol == it->left\_symbol && gram.right\_symbol.size() == it->right\_symbol.size()) {

int nsize = gram.right\_symbol.size();

int n = 0;

for (; n < nsize; n++) {

if (gram.right\_symbol[n] != it->right\_symbol[n])

break;

}

if (n == nsize) {

gramNum = t;

break;

}

}

}

if (gramNum == -1) {

cout << "归约出错!" << endl;

exit(-1);

}

pair<int, string>in = pair<int, string>(i, it->seek\_symbol);

//ACC

if (LR1\_table.count(in) == 0) {//没有冲突

if (gramNum == grammer.startItemPos) {

LR1\_table[in] = pair<string, int>("acc", gramNum);

}

else {

LR1\_table[in] = pair<string, int>("r", gramNum);

}

}

else {

cout << "ERROR:入口冲突!";

}

}

}

}

}

void LR1\_grammer::printLR1Table(string filename)

{

ofstream outfile;

outfile.open(filename);

if (!outfile) {

cout << "文件打开失败!" << endl;

exit(-1);

}

pair<int, string> in;

int stateNum = closureFamily.size();

int vtNum = grammer.Terminals.size();

int nvtNum = grammer.NTterminals.size();

int colNum = vtNum + nvtNum;

outfile << setiosflags(ios::left) << setw(7) << " ";

for (auto str : grammer.Terminals) {

outfile << setiosflags(ios::left) << setw(10) << str;

}

for (auto str : grammer.NTterminals) {

outfile << setiosflags(ios::left) << setw(10) << str;

}

outfile << endl;

//打表

for (int row = 0; row < stateNum; row++) {

outfile << setiosflags(ios::left) << setw(5) << row << ": ";

auto pt = grammer.Terminals.begin();

auto pnt = grammer.NTterminals.begin();

for (int col = 0; col < colNum; col++) {

string symbol;

if (col < vtNum) {

symbol = \*pt;

pt++;

}

else {

symbol = \*pnt;

pnt++;

}

in = pair<int, string>(row, symbol);

if (LR1\_table.count(in) == 0) {//REJ

outfile << setiosflags(ios::left) << setw(10) << " ";

}

else {

if (LR1\_table[in].first != "null") {

string temp;

if (LR1\_table[in].first != "acc")

temp = LR1\_table[in].first + to\_string(LR1\_table[in].second);

else

temp = LR1\_table[in].first;

outfile << setiosflags(ios::left) << setw(10) << temp;

}

else

outfile << setiosflags(ios::left) << setw(10) << LR1\_table[in].second;

}

}

outfile << endl;

}

}

### **7.2.5 2\_Grammatical\_Output.cpp**

#include "2\_Grammatical\_Analysis.h"

#include "2\_Grammatical\_LR1Grammer.h"

#include <fstream>

#include <queue>

#define QUEUE\_TYPE pair<ThreadedList\*, int>

void addNode(ofstream& outfile, int cnt, ThreadedList\* Node);

void addEdge(ofstream& outfile, int start, int end);

void addIdentifier(ofstream& outfile, int cnt, QUEUE\_TYPE identifier);

string dotPath = "../outfiles/syntaxTree.dot";

string pngPath = "../outfiles/syntaxTree.png";

// 将符合dot脚本语言规范的归结过程输出到.dot文件中

// 再调用接口函数转换成归结树的图片

void generateTreeDot(ThreadedList\* Root) {

ofstream outfile(dotPath, ios::out);

outfile << "digraph all{" << endl;

outfile << " rankdir = LR" << endl;

outfile << " node[shape = ellipse, fontname = \"Times-Roman\"]" << endl;

outfile << " edge[arrow = 0.5]" << endl;

// 使用label属性，避免重复结点

// 方法：对整棵树进行层次遍历，为每个结点创建一个node

// 使用nullptr作为换层标识

int nodeCount = 0;

queue <QUEUE\_TYPE> threadedListQueue;

threadedListQueue.push(QUEUE\_TYPE(Root, 0));

addNode(outfile, nodeCount, Root); // 写入根结点

while (!threadedListQueue.empty())

{

// 每次弹出队首，进行扩展连接，再将压入队列

// 扩展时，先添加结点声明，再添加连线

QUEUE\_TYPE elem = threadedListQueue.front();

threadedListQueue.pop();

if (elem.first->IdIfExists != "")

{

++nodeCount; // 添加新的结点前，需要count自增

addIdentifier(outfile, nodeCount, elem);

}

for (int i = 0; i < elem.first->threads.size(); ++i)

{

++nodeCount; // 添加新的结点前，需要count自增

QUEUE\_TYPE newNode(elem.first->threads[i], nodeCount);

addNode(outfile, nodeCount, newNode.first);

addEdge(outfile, elem.second, nodeCount);

threadedListQueue.push(newNode);

}

}

outfile << "}" << endl;

outfile.close();

string command;

command += "dot -Tpng " + dotPath + " -o " + pngPath;

system(command.c\_str());

}

void addIdentifier(ofstream& outfile, int cnt, QUEUE\_TYPE identifier)

{

outfile << "node" << to\_string(cnt) << '[' << "label = " << '\"';

outfile << identifier.first->IdIfExists;

outfile << '\"' << ']' << ';' << endl;

outfile << "node" << identifier.second << "->" << "node" << cnt;

outfile << "[style = \"dotted\", arrowtail = \"none\", arrowhead = \"none\"]" << ';' << endl;

}

void addNode(ofstream &outfile, int cnt, ThreadedList\* Node)

{

outfile << "node" << to\_string(cnt) << '[' << "label = " << '\"';

if (Node->regulationStep)

outfile << '(' << Node->regulationStep << ')' << ' ';

outfile << Node->symbol;

outfile << '\"' << ']' << ';' << endl;

}

void addEdge(ofstream& outfile, int start, int end)

{

outfile << "node" << start << "->" << "node" << end << ';' << endl;

}

### **7.2.6 3\_Semantic\_Analysis.cpp**

#include "3\_Semantic\_Analysis.h"

#include <fstream>

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* SemanticSymbolTable \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// 构造函数

SemanticSymbolTable::SemanticSymbolTable(const TableType type, const string name)

{

this->table\_type = type;

this->table\_name = name;

}

// 根据名字寻找一个变量 没找到则返回-1，否则返回变量在表中的标号

int SemanticSymbolTable::FindSymbol(const string name)

{

int size = this->table.size();

for (int i = 0; i < size; i++) {

if (table[i].identifierName == name)

return i;

}

return -1;

}

// 加入一个变量，返回加入的位置

int SemanticSymbolTable::AddSymbol(const IdentifierInfo id)

{

// 表中没有则加入 并返回其在表中的位置。

if (FindSymbol(id.identifierName) == -1) {

table.push\_back(id);

return table.size() - 1;

}

// 已存在则返回-1

return -1;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* SemanticAnalysis \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// 构造函数

SemanticAnalysis::SemanticAnalysis()

{

// 为语义分析构造全局符号表

this->tables.push\_back(SemanticSymbolTable(SemanticSymbolTable::GlobalTable, "theGlobalTable"));

// 设置当前作用域为 全局作用域

this->currentTableStack.push\_back(0);

// 构造临时变量符号表

this->tables.push\_back(SemanticSymbolTable(SemanticSymbolTable::TempTable, "theTempTable"));

// 约定第0条四元式：(j,-,-,main\_function)

this->nextQuaternaryIdx = 1;

this->mainIdx = -1; // main函数标号初始值设置为-1

this->tmpVarCount = 0; // 临时变量计数为0

this->backpatchLevel = 0; // 起初不需要回填

}

void SemanticAnalysis::AddSymbol2List(const SemanticSymbol symbol)

{

// 将所有符号信息放入symbol\_list

this->symbolList.push\_back(symbol);

}

// 分析过程调用

// 根据不同类型，分不同情况进行语义分析

void SemanticAnalysis::Analysis(const string production\_left, const vector<string> production\_right){

if (production\_left == "Program")

Semantic\_Analysis\_Program(production\_left, production\_right);

else if (production\_left == "ExtDef")

Semantic\_Analysis\_ExtDef(production\_left, production\_right);

else if (production\_left == "VarSpecifier")

Semantic\_Analysis\_VarSpecifier(production\_left, production\_right);

else if (production\_left == "FunSpecifier")

Semantic\_Analysis\_FunSpecifier(production\_left, production\_right);

else if (production\_left == "FunDec")

Semantic\_Analysis\_FunDec(production\_left, production\_right);

else if (production\_left == "CreateFunTable\_m")

Semantic\_Analysis\_CreateFunTable\_m(production\_left, production\_right);

else if (production\_left == "ParamDec")

Semantic\_Analysis\_ParamDec(production\_left, production\_right);

else if (production\_left == "Block")

Semantic\_Analysis\_Block(production\_left, production\_right);

else if (production\_left == "Def")

Semantic\_Analysis\_Def(production\_left, production\_right);

else if (production\_left == "AssignStmt")

Semantic\_Analysis\_AssignStmt(production\_left, production\_right);

else if (production\_left == "Exp")

Semantic\_Analysis\_Exp(production\_left, production\_right);

else if (production\_left == "AddSubExp")

Semantic\_Analysis\_AddSubExp(production\_left, production\_right);

else if (production\_left == "Item")

Semantic\_Analysis\_Item(production\_left, production\_right);

else if (production\_left == "Factor")

Semantic\_Analysis\_Factor(production\_left, production\_right);

else if (production\_left == "CallStmt")

Semantic\_Analysis\_CallStmt(production\_left, production\_right);

else if (production\_left == "CallFunCheck")

Semantic\_Analysis\_CallFunCheck(production\_left, production\_right);

else if (production\_left == "Args")

Semantic\_Analysis\_Args(production\_left, production\_right);

else if (production\_left == "ReturnStmt")

Semantic\_Analysis\_ReturnStmt(production\_left, production\_right);

else if (production\_left == "Relop")

Semantic\_Analysis\_Relop(production\_left, production\_right);

else if (production\_left == "IfStmt")

Semantic\_Analysis\_IfStmt(production\_left, production\_right);

else if (production\_left == "IfStmt\_m1")

Semantic\_Analysis\_IfStmt\_m1(production\_left, production\_right);

else if (production\_left == "IfStmt\_m2")

Semantic\_Analysis\_IfStmt\_m2(production\_left, production\_right);

else if (production\_left == "IfNext")

Semantic\_Analysis\_IfNext(production\_left, production\_right);

else if (production\_left == "IfStmt\_next")

Semantic\_Analysis\_IfStmt\_next(production\_left, production\_right);

else if (production\_left == "WhileStmt")

Semantic\_Analysis\_WhileStmt(production\_left, production\_right);

else if (production\_left == "WhileStmt\_m1")

Semantic\_Analysis\_WhileStmt\_m1(production\_left, production\_right);

else if (production\_left == "WhileStmt\_m2")

Semantic\_Analysis\_WhileStmt\_m2(production\_left, production\_right);

else {

if (production\_right[0] != "@") {

int count = production\_right.size();

while (count--)

this->symbolList.pop\_back();

}

this->symbolList.push\_back({ production\_left,"",-1,-1,-1,-1 });

}

}

// 打印四元式表

void SemanticAnalysis::PrintQuaternary(const string file\_path)

{

ofstream fout;

fout.open(file\_path, ios::out);

if (!fout) {

cout << file\_path << "文件打开失败!" << endl;

return;

}

for (int i = 0; i < this->quaternary.size(); i++) {

fout << this->quaternary[i].idx << " (" << this->quaternary[i].operatorType << "," <<

this->quaternary[i].op1 << "," << this->quaternary[i].op2 << "," << this->quaternary[i].result << ")\n";

}

}

// 语义分析函数

// Program : ExtDefList

void SemanticAnalysis::Semantic\_Analysis\_Program(const string production\_left, const vector<string> production\_right)

{

// 如果没有定义main函数，则报错

if (mainIdx == -1) {

cout << "语义分析中发生错误：未定义main函数" << endl;

exit(SEMANTIC\_ERROR\_NO\_MAIN);

}

int count = production\_right.size();

while (count--) {

symbolList.pop\_back();

}

// 在最前面加入四元式

quaternary.insert(quaternary.begin(), { 0, "j","-" , "-", to\_string(mainIdx) });

symbolList.push\_back({ production\_left, "", -1, -1, -1,-1 });

}

// ExtDef : VarSpecifier identifier ; \ VarSpecifier FunDec Block

void SemanticAnalysis::Semantic\_Analysis\_ExtDef(const string production\_left, const vector<string> production\_right)

{

// 如果是定义变量

if (production\_right.size() == 3) {

// 目前symbolList的末尾是Specifier identifier ;，由此找到Specifier和identifier

SemanticSymbol specifier = symbolList[symbolList.size() - 3]; // 变量类型

SemanticSymbol identifier = symbolList[symbolList.size() - 2]; // 变量名

// 用于判断该变量是否在当前层已经定义

bool existed = false;

SemanticSymbolTable\* current\_table = &tables[currentTableStack.back()];

if (current\_table->FindSymbol(identifier.value) != -1) {

cout << "语义分析中发生错误：（" << identifier.row << "行，" << identifier.col << "列）变量" << identifier.value << "重定义" << endl;

exit(SEMANTIC\_ERROR\_REDEFINED);

}

// 将这一变量加入table

IdentifierInfo variable;

variable.identifierName = identifier.value; // 变量名

variable.idType = IdentifierInfo::VAR;

variable.specifierType = specifier.value; // 变量类型

// 加入table

current\_table->AddSymbol(variable);

// symbolList更新

int count = production\_right.size();

while (count--)

symbolList.pop\_back();

symbolList.push\_back({ production\_left,identifier.value,identifier.row,identifier.col,currentTableStack.back(), int(current\_table->table.size() - 1) });

}

// 如果是定义函数

else {

SemanticSymbol identifier = symbolList[symbolList.size() - 2];

// 需要退出作用域

currentTableStack.pop\_back();

// 更新symbolList

int count = production\_right.size();

while (count--)

symbolList.pop\_back();

symbolList.push\_back({ production\_left,identifier.value,identifier.row,identifier.col,identifier.tableIdx,identifier.idx });

}

}

// VarSpecifier : int \ void \ float \ double

void SemanticAnalysis::Semantic\_Analysis\_VarSpecifier(const string production\_left, const vector<string> production\_right)

{

// symbolList的最后一个是int

SemanticSymbol specifier = symbolList.back();

specifier.value = production\_right[0]; // 将变量类型设置为value加入symbolList

int count = production\_right.size();

while (count--)

symbolList.pop\_back();

symbolList.push\_back({ production\_left,specifier.value,specifier.row,specifier.col,-1,-1 });

}

// $ VarSpecifier : void \ int

void SemanticAnalysis::Semantic\_Analysis\_FunSpecifier(const string production\_left, const vector<string> production\_right)

{

// symbolList的最后一个是int或void

SemanticSymbol specifier = symbolList.back();

int count = production\_right.size();

while (count--)

symbolList.pop\_back();

symbolList.push\_back({ production\_left,specifier.value,specifier.row,specifier.col,-1,-1 });

}

// FunDec : identifier CreateFunTable\_m ( VarList )

void SemanticAnalysis::Semantic\_Analysis\_FunDec(const string production\_left, const vector<string> production\_right)

{

// symbolList的CreateFunTable\_m记录了table信息

SemanticSymbol specifier = symbolList[symbolList.size() - 4];

int count = production\_right.size();

while (count--)

symbolList.pop\_back();

symbolList.push\_back({ production\_left,specifier.value,specifier.row,specifier.col,specifier.tableIdx,specifier.idx });

}

// CreateFunTable\_m : @

void SemanticAnalysis::Semantic\_Analysis\_CreateFunTable\_m(const string production\_left, const vector<string> production\_right)

{

// 创建函数表

// 此时symbolList最后一个符号为函数名，倒数第二个为函数返回值

SemanticSymbol identifier = symbolList.back();

SemanticSymbol specifier = symbolList[symbolList.size() - 2];

// 首先在全局的table判断函数名是否重定义

if (tables[0].FindSymbol(identifier.value) != -1) {

cout << "语义分析中发生错误：（" << identifier.row << "行，" << identifier.col << "列）函数" << identifier.value << "重定义" << endl;

exit(SEMANTIC\_ERROR\_REDEFINED);

}

// 创建函数表

tables.push\_back(SemanticSymbolTable(SemanticSymbolTable::FunctionTable, identifier.value));

// 在全局符号表创建当前函数的符号项（这里参数个数和入口地址会进行回填）

tables[0].AddSymbol({ IdentifierInfo::FUN,specifier.value,identifier.value,0,0,int(tables.size() - 1) });

// 函数表压入栈

currentTableStack.push\_back(tables.size() - 1);

// 返回值

IdentifierInfo return\_value;

return\_value.idType = IdentifierInfo::RET;

return\_value.identifierName = tables.back().table\_name + "\_return\_value";

return\_value.specifierType = specifier.value;

// 如果为main函数，则进行记录

if (identifier.value == "main")

mainIdx = nextQuaternaryIdx;

// 加入四元式

quaternary.push\_back({ nextQuaternaryIdx++ , identifier.value,"-","-" ,"-" });

// 向函数表中加入返回变量

tables[currentTableStack.back()].AddSymbol(return\_value);

// 空串不需要pop

// 进行pushback

symbolList.push\_back({ production\_left,identifier.value,identifier.row,identifier.col,0,int(tables[0].table.size() - 1) });

}

// ParamDec : VarSpecifier identifier

void SemanticAnalysis::Semantic\_Analysis\_ParamDec(const string production\_left, const vector<string> production\_right)

{

// symbolList最后一个为变量名，倒数第二个为类型

SemanticSymbol identifier = symbolList.back();

SemanticSymbol specifier = symbolList[symbolList.size() - 2];

// 当前函数表

SemanticSymbolTable& function\_table = tables[currentTableStack.back()];

// 如果已经进行过定义

if (function\_table.FindSymbol(identifier.value) != -1) {

cout << "语义分析中发生错误：（" << identifier.row << "行，" << identifier.col << "列）函数参数" << identifier.value << "重定义" << endl;

exit(SEMANTIC\_ERROR\_REDEFINED);

}

// 函数表加入形参变量

int new\_position = function\_table.AddSymbol({ IdentifierInfo::VAR,specifier.value,identifier.value,-1,-1,-1 });

// 当前函数在全局符号中的索引

int table\_position = tables[0].FindSymbol(function\_table.table\_name);

// 形参个数++

tables[0].table[table\_position].funcParameterNum++;

// 加入四元式

quaternary.push\_back({ nextQuaternaryIdx++, "defpar","-" , "-", identifier.value });

// symbolList更新

int count = production\_right.size();

while (count--)

symbolList.pop\_back();

symbolList.push\_back({ production\_left,identifier.value,identifier.row,identifier.col,currentTableStack.back(),new\_position });

}

// Block : { DefList StmtList }

void SemanticAnalysis::Semantic\_Analysis\_Block(const string production\_left, const vector<string> production\_right)

{

// 更新symbolList

int count = production\_right.size();

while (count--)

symbolList.pop\_back();

symbolList.push\_back({ production\_left,to\_string(nextQuaternaryIdx),-1,-1,-1,-1 });

}

// Def : VarSpecifier identifier ;

void SemanticAnalysis::Semantic\_Analysis\_Def(const string production\_left, const vector<string> production\_right)

{

// symbolList的倒数第二个、倒数第三个是变量名和类型

SemanticSymbol identifier = symbolList[symbolList.size() - 2];

SemanticSymbol specifier = symbolList[symbolList.size() - 3];

SemanticSymbolTable& current\_table = tables[currentTableStack.back()];

// 重定义则报错

if (current\_table.FindSymbol(identifier.value) != -1)

{

cout << "语义分析中发生错误：（" << identifier.row << "行，" << identifier.col << "列）变量" << identifier.value << "重定义" << endl;

exit(SEMANTIC\_ERROR\_REDEFINED);

}

current\_table.AddSymbol({ IdentifierInfo::VAR,specifier.value,identifier.value ,-1,-1,-1 });

int count = production\_right.size();

while (count--)

symbolList.pop\_back();

symbolList.push\_back({ production\_left, identifier.value ,identifier.row,identifier.col,currentTableStack.back(),int(tables[currentTableStack.back()].table.size() - 1) });

}

// AssignStmt : identifier = Exp

void SemanticAnalysis::Semantic\_Analysis\_AssignStmt(const string production\_left, const vector<string> production\_right)

{

// symbolList的倒数第一个、倒数第三个是Exp和变量名

SemanticSymbol identifier = symbolList[symbolList.size() - 3];

SemanticSymbol exp = symbolList.back();

// 检查id是否存在，不存在则报错

bool existed = false;

int tableIdx = -1, idx = -1;

// 从当前层开始向上遍历

for (int scope\_layer = currentTableStack.size() - 1; scope\_layer >= 0; scope\_layer--) {

auto current\_table = tables[currentTableStack[scope\_layer]];

if ((idx = current\_table.FindSymbol(identifier.value)) != -1) {

existed = true;

tableIdx = currentTableStack[scope\_layer];

break;

}

}

if (existed == false) {

cout << "语义分析中发生错误：（" << identifier.row << "行，" << identifier.col << "列）变量" << identifier.value << "未定义" << endl;

exit(SEMANTIC\_ERROR\_UNDEFINED);

}

quaternary.push\_back({ nextQuaternaryIdx++, "=", exp.value, "-", identifier.value });

// 更新symbolList

int count = production\_right.size();

while (count--)

symbolList.pop\_back();

symbolList.push\_back({ production\_left, identifier.value ,identifier.row,identifier.col,tableIdx,idx });

}

// Exp : AddSubExp \ Exp Relop AddSubExp

void SemanticAnalysis::Semantic\_Analysis\_Exp(const string production\_left, const vector<string> production\_right)

{

if (production\_right.size() == 1) {

SemanticSymbol exp = symbolList.back();

// 更新symbolList

int count = production\_right.size();

while (count--)

symbolList.pop\_back();

symbolList.push\_back({ production\_left, exp.value ,exp.row,exp.col,exp.tableIdx,exp.idx });

}

else {

SemanticSymbol sub\_exp1 = symbolList[symbolList.size() - 3];

SemanticSymbol op = symbolList[symbolList.size() - 2];

SemanticSymbol sub\_exp2 = symbolList[symbolList.size() - 1];

int next\_label\_num = nextQuaternaryIdx++;

string new\_tmp\_var = "T" + to\_string(tmpVarCount++);

quaternary.push\_back({ next\_label\_num, "j" + op.value, sub\_exp1.value, sub\_exp2.value, to\_string(next\_label\_num + 3) });

quaternary.push\_back({ nextQuaternaryIdx++, "=", "0", "-", new\_tmp\_var });

quaternary.push\_back({ nextQuaternaryIdx++, "j", "-", "-", to\_string(next\_label\_num + 4) });

quaternary.push\_back({ nextQuaternaryIdx++, "=", "1", "-", new\_tmp\_var });

int count = production\_right.size();

while (count--)

symbolList.pop\_back();

symbolList.push\_back({ production\_left, new\_tmp\_var ,-1,-1,-1,-1 });

}

}

// AddSubExp : Item \ Item + Item \ Item - Item

void SemanticAnalysis::Semantic\_Analysis\_AddSubExp(const string production\_left, const vector<string> production\_right)

{

if (production\_right.size() == 1) {

SemanticSymbol exp = symbolList.back();

// 更新symbolList

int count = production\_right.size();

while (count--)

symbolList.pop\_back();

symbolList.push\_back({ production\_left, exp.value ,exp.row,exp.col,exp.tableIdx,exp.idx });

}

else {

SemanticSymbol sub\_exp1 = symbolList[symbolList.size() - 3];

SemanticSymbol op = symbolList[symbolList.size() - 2];

SemanticSymbol sub\_exp2 = symbolList[symbolList.size() - 1];

string new\_tmp\_var = "T" + to\_string(tmpVarCount++);

quaternary.push\_back({ nextQuaternaryIdx++, op.token, sub\_exp1.value, sub\_exp2.value, new\_tmp\_var });

int count = production\_right.size();

while (count--)

symbolList.pop\_back();

symbolList.push\_back({ production\_left, new\_tmp\_var ,-1,-1,-1,-1 });

}

}

// Item : Factor \ Factor \* Factor \ Factor / Factor

void SemanticAnalysis::Semantic\_Analysis\_Item(const string production\_left, const vector<string> production\_right)

{

if (production\_right.size() == 1) {

SemanticSymbol exp = symbolList.back();

// 更新symbolList

int count = production\_right.size();

while (count--)

symbolList.pop\_back();

symbolList.push\_back({ production\_left, exp.value ,exp.row,exp.col,exp.tableIdx,exp.idx });

}

else {

SemanticSymbol sub\_exp1 = symbolList[symbolList.size() - 3];

SemanticSymbol op = symbolList[symbolList.size() - 2];

SemanticSymbol sub\_exp2 = symbolList[symbolList.size() - 1];

std::string new\_tmp\_var = "T" + to\_string(tmpVarCount++);

quaternary.push\_back({ nextQuaternaryIdx++, op.token, sub\_exp1.value, sub\_exp2.value, new\_tmp\_var });

int count = production\_right.size();

while (count--)

symbolList.pop\_back();

symbolList.push\_back({ production\_left, new\_tmp\_var ,-1,-1,-1,-1 });

}

}

// Factor : number \ ( Exp ) \ identifier \ CallStmt

void SemanticAnalysis::Semantic\_Analysis\_Factor(const string production\_left, const vector<string> production\_right)

{

if (production\_right.size() == 1) {

SemanticSymbol exp = symbolList.back();

// 如果是ID检查其是否进行过定义

if (production\_right[0] == "identifier") {

bool existed = false;

for (int scope\_layer = currentTableStack.size() - 1; scope\_layer >= 0; scope\_layer--) {

auto current\_table = tables[currentTableStack[scope\_layer]];

if (current\_table.FindSymbol(exp.value) != -1) {

existed = true;

break;

}

}

if (existed == false) {

cout << "语义分析中发生错误：（" << exp.row << "行，" << exp.col << "列）变量" << exp.value << "未定义" << endl;

exit(SEMANTIC\_ERROR\_UNDEFINED);

}

}

// 更新symbolList

int count = production\_right.size();

while (count--)

symbolList.pop\_back();

symbolList.push\_back({ production\_left, exp.value ,exp.row,exp.col,exp.tableIdx,exp.idx });

}

else {

SemanticSymbol exp = symbolList[symbolList.size() - 2];

// 更新symbolList

int count = production\_right.size();

while (count--)

symbolList.pop\_back();

symbolList.push\_back({ production\_left, exp.value ,exp.row,exp.col,exp.tableIdx,exp.idx });

}

}

// CallStmt : identifier ( CallFunCheck Args )

void SemanticAnalysis::Semantic\_Analysis\_CallStmt(const string production\_left, const vector<string> production\_right)

{

SemanticSymbol identifier = symbolList[symbolList.size() - 5];

SemanticSymbol check = symbolList[symbolList.size() - 3];

SemanticSymbol args = symbolList[symbolList.size() - 2];

// 检查函数是否定义（在CallFunCheck时已经检查）

// 检查参数个数

int para\_num = tables[check.tableIdx].table[check.idx].funcParameterNum;

if (para\_num > stoi(args.value)) {

cout << "语义分析中发生错误：（" << identifier.row << "行，" << identifier.col << "列）函数" << identifier.value << "调用时参数过少" << endl;

exit(SEMANTIC\_ERROR\_PARAMETER\_NUM);

}

else if (para\_num < stoi(args.value)) {

cout << "语义分析中发生错误：（" << identifier.row << "行，" << identifier.col << "列）函数" << identifier.value << "调用时参数过多" << endl;

exit(SEMANTIC\_ERROR\_PARAMETER\_NUM);

}

// 生成函数调用四元式

string returnType = tables[0].table[tables[0].FindSymbol(identifier.value)].specifierType;

string new\_tmp\_var = returnType == "void" ? "-" : ("T" + to\_string(tmpVarCount++));

quaternary.push\_back({ nextQuaternaryIdx++, "call", identifier.value, "-", new\_tmp\_var });

int count = production\_right.size();

while (count--)

symbolList.pop\_back();

// 新的exp的value为临时变量名

symbolList.push\_back({ production\_left, new\_tmp\_var ,-1,-1,-1,-1 });

}

// CallFunCheck : @

void SemanticAnalysis::Semantic\_Analysis\_CallFunCheck(const string production\_left, const vector<string> production\_right)

{

SemanticSymbol fun\_id = symbolList[symbolList.size() - 2];

int fun\_id\_pos = tables[0].FindSymbol(fun\_id.value);

if (-1 == fun\_id\_pos) {

cout << "语义分析中发生错误：（" << fun\_id.row << "行，" << fun\_id.col << "列）函数" << fun\_id.value << "调用未定义" << endl;

exit(SEMANTIC\_ERROR\_UNDEFINED);

}

if (tables[0].table[fun\_id\_pos].idType != IdentifierInfo::FUN) {

cout << "语义分析中发生错误：（" << fun\_id.row << "行，" << fun\_id.col << "列）函数" << fun\_id.value << "调用未定义" << endl;

exit(SEMANTIC\_ERROR\_UNDEFINED);

}

symbolList.push\_back({ production\_left, fun\_id.value,fun\_id.row,fun\_id.col, 0, fun\_id\_pos });

}

// Args : Exp , Args \ Exp \ @

void SemanticAnalysis::Semantic\_Analysis\_Args(const string production\_left, const vector<string> production\_right)

{

if (production\_right.size() == 3) {

SemanticSymbol exp = symbolList[symbolList.size() - 3];

quaternary.push\_back({ nextQuaternaryIdx++, "param", exp.value, "-", "-" });

int aru\_num = stoi(symbolList.back().value) + 1;

int count = production\_right.size();

while (count--)

symbolList.pop\_back();

symbolList.push\_back({ production\_left, to\_string(aru\_num),-1,-1,-1,-1 });

}

else if (production\_right[0] == "Exp") {

SemanticSymbol exp = symbolList.back();

quaternary.push\_back({ nextQuaternaryIdx++, "param", exp.value, "-", "-" });

int count = production\_right.size();

while (count--)

symbolList.pop\_back();

symbolList.push\_back({ production\_left, "1",-1,-1,-1,-1 });

}

else if (production\_right[0] == "@") {

symbolList.push\_back({ production\_left, "0",-1,-1,-1,-1 });

}

}

// ReturnStmt : return Exp \ return

void SemanticAnalysis::Semantic\_Analysis\_ReturnStmt(const string production\_left, const vector<string> production\_right)

{

if (production\_right.size() == 2) {

// 返回值

SemanticSymbol return\_exp = symbolList.back();

// 函数表

SemanticSymbolTable function\_table = tables[currentTableStack.back()];

// 添加四元式

quaternary.push\_back({ nextQuaternaryIdx++,"=",return\_exp.value,"-",function\_table.table[0].identifierName });

// 添加四元式

quaternary.push\_back({ nextQuaternaryIdx++ ,"return",function\_table.table[0].identifierName,"-",function\_table.table\_name });

// 更新symbolList

int count = production\_right.size();

while (count--)

symbolList.pop\_back();

symbolList.push\_back({ production\_left, return\_exp.value,-1,-1,-1,-1 });

}

else {

// 函数表

SemanticSymbolTable function\_table = tables[currentTableStack.back()];

// 检查函数的返回值是否为void

if (tables[0].table[tables[0].FindSymbol(function\_table.table\_name)].specifierType != "void") {

cout << "语义分析中发生错误：（" << symbolList.back().row << "行，" << symbolList.back().col + sizeof("return") << "列）函数" << function\_table.table\_name << "必须有返回值" << endl;

exit(SEMANTIC\_ERROR\_NO\_RETURN);

}

// 添加四元式

quaternary.push\_back({ nextQuaternaryIdx++ ,"return","-","-",function\_table.table\_name });

// 更新symbolList

int count = production\_right.size();

while (count--)

symbolList.pop\_back();

symbolList.push\_back({ production\_left, "",-1,-1,-1,-1 });

}

}

// Relop : > \ < \ >= \ <= \ == \ !=

void SemanticAnalysis::Semantic\_Analysis\_Relop(const string production\_left, const vector<string> production\_right)

{

SemanticSymbol op = symbolList.back();

int count = production\_right.size();

while (count--) {

symbolList.pop\_back();

}

symbolList.push\_back({ production\_left, op.token ,-1,-1,-1,-1 });

}

// IfStmt : if IfStmt\_m1 ( Exp ) IfStmt\_m2 Block IfNext

void SemanticAnalysis::Semantic\_Analysis\_IfStmt(const string production\_left, const vector<string> production\_right)

{

SemanticSymbol ifstmt\_m2 = symbolList[symbolList.size() - 3];

SemanticSymbol ifnext = symbolList[symbolList.size() - 1];

if (ifnext.value.empty()) {

// 只有if没有else

// 真出口

quaternary[backpatchList.back()].result = ifstmt\_m2.value;

backpatchList.pop\_back();

// 假出口

quaternary[backpatchList.back()].result = to\_string(nextQuaternaryIdx);

backpatchList.pop\_back();

}

else {

// if块出口

quaternary[backpatchList.back()].result = to\_string(nextQuaternaryIdx);

backpatchList.pop\_back();

// if真出口

quaternary[backpatchList.back()].result = ifstmt\_m2.value;

backpatchList.pop\_back();

// if假出口

quaternary[backpatchList.back()].result = ifnext.value;

backpatchList.pop\_back();

}

backpatchLevel--;

// popback

int count = production\_right.size();

while (count--)

symbolList.pop\_back();

symbolList.push\_back({ production\_left,"",-1,-1,-1,-1 });

}

// IfStmt\_m1 ::= @

void SemanticAnalysis::Semantic\_Analysis\_IfStmt\_m1(const string production\_left, const vector<string> production\_right)

{

backpatchLevel++;

symbolList.push\_back({ production\_left,to\_string(nextQuaternaryIdx),-1,-1,-1,-1 });

}

// IfStmt\_m2 : @

void SemanticAnalysis::Semantic\_Analysis\_IfStmt\_m2(const string production\_left, const vector<string> production\_right)

{

SemanticSymbol if\_exp = symbolList[symbolList.size() - 2];

// 待回填四元式 : 假出口

quaternary.push\_back({ nextQuaternaryIdx++,"j=",if\_exp.value,"0","" });

backpatchList.push\_back(quaternary.size() - 1);

// 待回填四元式 : 真出口

quaternary.push\_back({ nextQuaternaryIdx++,"j=","-","-","" });

backpatchList.push\_back(quaternary.size() - 1);

symbolList.push\_back({ production\_left,to\_string(nextQuaternaryIdx),-1,-1,-1,-1 });

}

// IfNext : @ \ IfStmt\_next else Block

void SemanticAnalysis::Semantic\_Analysis\_IfNext(const string production\_left, const vector<string> production\_right)

{

SemanticSymbol if\_stmt\_next = symbolList[symbolList.size() - 3];

int count = production\_right.size();

while (count--)

symbolList.pop\_back();

symbolList.push\_back({ production\_left,if\_stmt\_next.value,-1,-1,-1,-1 });

}

// IfStmt\_next : @

void SemanticAnalysis::Semantic\_Analysis\_IfStmt\_next(const string production\_left, const vector<string> production\_right)

{

// if 的跳出语句(else 之前)(待回填)

quaternary.push\_back({ nextQuaternaryIdx++,"j","-","-","" });

backpatchList.push\_back(quaternary.size() - 1);

symbolList.push\_back({ production\_left,to\_string(nextQuaternaryIdx),-1,-1,-1,-1 });

}

// WhileStmt : while WhileStmt\_m1 ( Exp ) WhileStmt\_m2 Block

void SemanticAnalysis::Semantic\_Analysis\_WhileStmt(const string production\_left, const vector<string> production\_right)

{

SemanticSymbol whilestmt\_m1 = symbolList[symbolList.size() - 6];

SemanticSymbol whilestmt\_m2 = symbolList[symbolList.size() - 2];

// 无条件跳转到 while 的条件判断语句处

quaternary.push\_back({ nextQuaternaryIdx++,"j","-","-" ,whilestmt\_m1.value });

// 回填真出口

quaternary[backpatchList.back()].result = whilestmt\_m2.value;

backpatchList.pop\_back();

// 回填假出口

quaternary[backpatchList.back()].result = to\_string(nextQuaternaryIdx);

backpatchList.pop\_back();

backpatchLevel--;

int count = production\_right.size();

while (count--)

symbolList.pop\_back();

symbolList.push\_back({ production\_left,"",-1,-1,-1,-1 });

}

// WhileStmt\_m1 : @

void SemanticAnalysis::Semantic\_Analysis\_WhileStmt\_m1(const string production\_left, const vector<string> production\_right)

{

backpatchLevel++;

symbolList.push\_back({ production\_left,to\_string(nextQuaternaryIdx),-1,-1,-1,-1 });

}

// WhileStmt\_m2 : @

void SemanticAnalysis::Semantic\_Analysis\_WhileStmt\_m2(const string production\_left, const vector<string> production\_right)

{

SemanticSymbol while\_exp = symbolList[symbolList.size() - 2];

// 假出口

quaternary.push\_back({ nextQuaternaryIdx++,"j=",while\_exp.value,"0","" });

backpatchList.push\_back(quaternary.size() - 1);

// 真出口

quaternary.push\_back({ nextQuaternaryIdx++ ,"j","-","-" ,"" });

backpatchList.push\_back(quaternary.size() - 1);

symbolList.push\_back({ production\_left,to\_string(nextQuaternaryIdx),-1,-1,-1,-1 });

}

### **7.2.7 4\_Assembly\_Generator.cpp**

#include "4\_Assembly\_Generator.h"

// mips结构下，一共有32个寄存器

vector<string> registers = {

"$zero", // $0 常量0

"$at", // $1 由汇编器使用

"$v0","$v1", // $2-$3 用来存放子程序的返回值

"$a0","$a1","$a2","$a3", // $4-$7 用来传递子程序的前四个参数

"$t0","$t1","$t2","$t3","$t4","$t5","$t6","$t7", // $8-$15 暂存器，用来存放子程序计算过程中的临时变量

"$s0","$s1","$s2","$s3","$s4","$s5","$s6","$s7", // $16-$23 存放子程序调用过程中需要保持不变的值

"$t8","$t9", // $24-$25 保留

"$k0","k1", // $26-$27 操作系统／异常处理保留

"$gp", // $28 全局指针 (Global Pointer)

"$sp", // $29 堆栈指针 (Stack Pointer) 指向栈顶上一个-4

"$fp", // $30 帧指针 (Frame Pointer)

"$ra" // $31 返回地址 (Return Address)

};

bool is\_digit(string str)

{

if (isdigit(str[0])) // 是正数

return true;

else if (str[0] == '-' && isdigit(str[1])) // 是负数

return true;

return false;

}

Assembly\_Generator::Assembly\_Generator(string out\_path, const vector<Quatenary> input\_quatenary)

{

// 1. 打开输出文件

Asm\_stream.open(out\_path, ios::out);

if (!Asm\_stream.is\_open())

{

cout << "ERROR: cannot open the assembly output file!" << endl;

exit(EXIT\_FAILURE);

}

// 2. 存入生成的四元式

quatenary = input\_quatenary;

// 3. 存储标签对应地址

for (Quatenary cur : quatenary)

if (cur.operatorType[0] == 'j') // j j> j< j= ...

J\_DestinationAddress.insert(cur.result);

// 4. 初始化寄存器信息以及寄存器值

for (int i = 0; i < REGS\_NUMBER; i++)

{

RegisterInfo[i] = { i, registers[i], 0 }; // {序号，名字，未使用时间}

RVALUE[i] = i ? "" : "0"; // 0号寄存器赋值为0，其余不赋值

}

}

// 汇编代码生成核心模块

// 根据四元式的类型执行不同的生成函数调用

void Assembly\_Generator::generate()

{

// 程序开始前，需要将main函数相关指针赋值，堆栈指针$sp赋栈顶esp，帧指针$fp赋返回地址ebp

// main参数：argc argv

Asm\_stream << "addi $sp,$sp," << to\_string(STACK\_SEG\_ADDR - 4 \* paramsNumber - 8) << endl;

Asm\_stream << "addi $fp,$fp," << to\_string(STACK\_SEG\_ADDR - 4 \* paramsNumber) << endl;

// int j = 0;

for (Quatenary quat : quatenary)

{

// cout << j++ << endl;

// 对于每一条四元式，各个寄存器未使用时间加一

for (int i = 0; i < REGS\_NUMBER; ++i)

{

Reg\* reg = &RegisterInfo[i];

if (reg->unusedTime < INT32\_MAX)

++reg->unusedTime;

}

// 如果需要输出标签，则输出之（所有标签都按照"Label\_ + 地址"的形式命名）

if (J\_DestinationAddress.find(to\_string(quat.idx)) != J\_DestinationAddress.end()) {

if (quat.op1 == "-" && quat.op2 == "-" && quat.result == "-") // 如果是过程块定义，则不需要

;

else

Asm\_stream << "Label\_" << to\_string(quat.idx) << " :" << endl;

}

// 对于每一条四元式，根据操作的不同，调用不同的处理函数

string opType = quat.operatorType;

if (opType == "call") {

callTypeHandler(quat);

}

else if (opType == "return") {

returnTypeHandler(quat);

}

else if (opType == "+" || opType == "-" || opType == "\*" || opType == "/") {

//if (AVALUE.find(quat.result) == AVALUE.end())

// AVALUE.insert({ quat.result, {} }); // 如果该变量不在AVALUE中，则插入

computeTypeHandler(quat);

}

else if (opType == "=") {

//if (AVALUE.find(quat.result) == AVALUE.end())

// AVALUE.insert({ quat.result, {} }); // 如果该变量不在AVALUE中，则插入

assignTypeHandler(quat);

}

else if (opType == "param") {

paramTypeHandler(quat);

}

else if (opType == "defpar") {

defparTypeHandler(quat);

}

else if (opType[0] == 'j') {

jTypeHandler(quat);

}

else if (quat.op1 == "-" && quat.op2 == "-" && quat.result == "-") {

defprocTypeHandler(quat);

}

}

}

// 处理函数调用的汇编代码生成

void Assembly\_Generator::callTypeHandler(Quatenary quat) // 函数调用

{

// 1. 调用时更新栈帧 2. 返回后撤销剩余的栈帧（一部分在return处被撤销）

// 栈帧的结构： fp↓ sp↓

// | .. | 0 -4 -8 -12 -16 -20 -24 -28 -32 -36 -40

// | params | t0 t1 t2 t3 t4 t5 t6 t7 fp ra

// (old)

// 类似UNIX系统的栈帧结构

// 函数调用时，需要保存寄存器现场（不考虑硬件现场），参数已经在前面的param四元式压栈

// mips中没有push / pop指令，因此使用sw / lw形式实现

{

Asm\_stream << "sw $t0,($sp) " << endl; // 寄存器压栈

Asm\_stream << "sw $t1,-4($sp) " << endl;

Asm\_stream << "sw $t2,-8($sp) " << endl;

Asm\_stream << "sw $t3,-12($sp)" << endl;

Asm\_stream << "sw $t4,-16($sp)" << endl;

Asm\_stream << "sw $t5,-20($sp)" << endl;

Asm\_stream << "sw $t6,-24($sp)" << endl;

Asm\_stream << "sw $t7,-28($sp)" << endl;

}

Asm\_stream << "subi $sp,$sp,32 " << endl; // 1. 分配栈帧

Asm\_stream << "sw $ra,-4($sp)" << endl; // 2. 保存返回地址

Asm\_stream << "sw $fp,($sp) " << endl; // 3. 保存堆栈指针

Asm\_stream << "move $sp,$fp " << endl; // 4. 更新堆栈指针

Asm\_stream << "subi $sp,$sp,8 " << endl; // 5. 更新栈帧

Asm\_stream << "jal "<< quat.op1 << endl;

// 跳转返回后的指令

Asm\_stream << "addiu $sp,$sp,8 " << endl;

{

Asm\_stream << "lw $t7,4($sp) " << endl; // 寄存器出栈，恢复现场

Asm\_stream << "lw $t6,8($sp) " << endl;

Asm\_stream << "lw $t5,12($sp)" << endl;

Asm\_stream << "lw $t4,16($sp)" << endl;

Asm\_stream << "lw $t3,20($sp)" << endl;

Asm\_stream << "lw $t2,24($sp)" << endl;

Asm\_stream << "lw $t1,28($sp)" << endl;

Asm\_stream << "lw $t0,32($sp)" << endl;

}

// 恢复sp到上一帧的栈顶，其中32是八个寄存器的偏移，剩下的是传入参数的偏移

Asm\_stream << "lw $sp," << to\_string(32 + paramsNumber \* 4) << "($fp) " << endl;

Asm\_stream << "lw $fp,($fp) " << endl; // 恢复fp到上一帧的old fp位置（类似UNIX中的old ebp）

if (quat.result != "-") // 如果调用时设置了变量接收返回值（考虑到可能有void函数不会接收返回值）

{

int returnReg = AVALUE[quat.result];

// 大于零，说明在寄存器中，放到寄存器里即可

if (returnReg <= 0) {

// 为零，说明没有声明过，需要分配一个寄存器（由于下标访问不存在的话会直接插入，所以无需再次插入之）

// 为-1，说明放到了内存 —— 但是没关系！反正也要覆盖，直接分配寄存器就好了！

returnReg = allocateReg(quat.result, "-", "-");

}

Asm\_stream << "move " + registers[returnReg] + ", $v1" << endl;

paramsNumber = 0;

}

}

void Assembly\_Generator::returnTypeHandler(Quatenary quat) // 返回

{

// 1. 保存返回值 2. 撤销部分栈帧

if (quat.op1 != "-") // 如果有返回值，将返回值压栈，否则不需要压栈

Asm\_stream << "move $v1," + registers[AVALUE[quat.op1]] << endl;

// 释放局部变量占用的寄存器，修改AVALUE与RVALUE

for (auto localVar = localVariantAddressTable.begin(); localVar != localVariantAddressTable.end(); ++localVar)

{

int reg = AVALUE[localVar->first];

if (reg > 0) // 将还在占用的寄存器清除

RVALUE[reg] = "";

AVALUE.erase(localVar->first);

}

Asm\_stream << "move $sp,$fp " << endl;

Asm\_stream << "subi $sp,$sp,8 " << endl;

Asm\_stream << "lw $ra,-4($fp)" << endl; // 返回地址赋给$ra

// 如果不是main函数则跳转

if (procedureName != "main")

Asm\_stream << "jr $ra" << endl;

else

Asm\_stream << "break" << endl;

localVarOffset = LOCAL\_VAR\_OFFSET;

localVariantAddressTable.clear();

procedureName = "";

}

void Assembly\_Generator::jTypeHandler(Quatenary quat) // 跳转（包括j j< j> j= j<= j>=）

{

if (quat.operatorType == "j")

{

//Asm\_stream << "j ";

//// 四元式结果项首字符非数字，说明为函数名，否则为标签地址（对于后续的条件跳转，不会有跳转函数的情况发生）

//if (!isdigit(quat.result[0])) {

// Asm\_stream << functionAddressTable[quat.result] << endl;

//}

//else {

// Asm\_stream << "Label\_" << quat.result << endl;

//}

Quatenary dst = quatenary[atoi(quat.result.c\_str())];

if (dst.op1 == "-" && dst.op2 == "-" && dst.result == "-")

Asm\_stream << "j " << dst.operatorType << endl;

else

Asm\_stream << "j Label\_" << quat.result << endl;

}

else

{

int A\_Reg = -1, B\_Reg = -1;

// 如果A是立即数，把A放在$t8，否则找到A所在的寄存器或为A分配寄存器

if (isdigit(quat.op1[0])) {

Asm\_stream << "addi $t8,$zero," << quat.op1 << endl;

A\_Reg = 24;

}

else {

A\_Reg = AVALUE[quat.op1];

if (A\_Reg == -1) // 说明需要将参数从内存换入

A\_Reg = loadVar(quat.op1);

else if (A\_Reg == 0)

A\_Reg = allocateReg(quat.op1);

}

// B同理

if (isdigit(quat.op2[0])) {

Asm\_stream << "addi $t9,$zero," << quat.op2 << endl;

B\_Reg = 25;

}

// 否则找到A所在的寄存器 / 为A抢占寄存器

else {

B\_Reg = allocateReg(quat.op2);

}

if (quat.operatorType == "j>") // J if A > B

Asm\_stream << "bgt ";

else if (quat.operatorType == "j=") // J if A = B

Asm\_stream << "beq ";

else if (quat.operatorType == "j<") // J if A < B

Asm\_stream << "blt ";

else if (quat.operatorType == "j>=") // J if A >= B

Asm\_stream << "bge ";

else if (quat.operatorType == "j<=") // J if A <= B

Asm\_stream << "ble ";

Asm\_stream << registers[A\_Reg] << "," << registers[B\_Reg] << ",Label\_" << quat.result << endl;

}

}

// 计算（包括+ - \* /）的汇编代码生成

void Assembly\_Generator::computeTypeHandler(Quatenary quat)

{

// 1. 分配寄存器，将B和C从内存换入

// A := B +-\*/ C

// 对于AVALUE为-1的变量B/C，考察内存是否有这个变量（应该是有的），需要lw指令存进寄存器

int A\_Reg = AVALUE[quat.result];

int B\_Reg = AVALUE[quat.op1];

int C\_Reg = -1; // C可能是立即数，要先判断（但B不会是立即数）

if (B\_Reg == -1) // 说明B的值在内存，需要换入

B\_Reg = loadVar(quat.op1, A\_Reg, C\_Reg);

else if (B\_Reg == 0) // 说明没有出现过B，这种情况下B可能是全局变量，按照全局变量处理

{

// 先将全局变量B注册到表中

globalVariantAddressTable.insert({ quat.op1, localVarOffset });

globalVarAddr += 4; // 向下增长

// 为B申请一个寄存器

B\_Reg = allocateReg(quat.op1, (A\_Reg > 0) ? quat.result : "-");

}

if (!is\_digit(quat.op2)) // C不是立即数，则需要考虑寄存器问题

{

C\_Reg = AVALUE[quat.op2];

if (C\_Reg == -1) // 说明C的值在内存，需要换入

C\_Reg = loadVar(quat.op2, A\_Reg, B\_Reg);

RegisterInfo[C\_Reg].unusedTime = 0;

}

//if (A\_Reg == -1) // 如果在内存，需要换到寄存器中

// A\_Reg = loadVar(quat.result, B\_Reg, C\_Reg);

if (A\_Reg <= 0) // 第一次使用，需要分配寄存器

{

if(is\_digit(quat.op2))

A\_Reg = allocateReg(quat.result, quat.op1);

else

A\_Reg = allocateReg(quat.result, quat.op1, quat.op2);

}

// 2. 编写加减乘除的汇编

if (quat.operatorType == "+")

{

if (is\_digit(quat.op2))

Asm\_stream << "addi " << registers[A\_Reg] << "," << registers[B\_Reg] << "," << quat.op2 << endl;

else

Asm\_stream << "add " << registers[A\_Reg] << "," << registers[B\_Reg] << "," << registers[C\_Reg] << endl;

}

else if (quat.operatorType == "-")

{

if (is\_digit(quat.op2))

Asm\_stream << "subi " << registers[A\_Reg] << "," << registers[B\_Reg] << "," << quat.op2 << endl;

else

Asm\_stream << "sub " << registers[A\_Reg] << "," << registers[B\_Reg] << "," << registers[C\_Reg] << endl;

}

else if (quat.operatorType == "\*") // mips中mult指令先将结果保存到{hi，lo}，而后使用mfhi&mflo取出，此处使用mul指令简化

{

if (is\_digit(quat.op2)) // MIPS中没有立即数乘法，必须先把立即数C赋值到一个临时寄存器

{

Asm\_stream << "addi $t8,$zero," << quat.op2 << endl;

C\_Reg = 24;

}

Asm\_stream << "mul " << registers[A\_Reg] << "," << registers[B\_Reg] << "," << registers[C\_Reg] << endl;

}

else if (quat.operatorType == "/")

{

if (is\_digit(quat.op2)) // MIPS中没有立即数乘法，必须先把立即数C赋值到一个临时寄存器

{

Asm\_stream << "addi $t8,$zero," << quat.op2 << endl;

C\_Reg = 24;

}

Asm\_stream << "div " << registers[B\_Reg] << "," << registers[C\_Reg] << endl;

Asm\_stream << "mov " << registers[A\_Reg] << ",$lo" << endl;

}

RegisterInfo[B\_Reg].unusedTime = 0;

}

void Assembly\_Generator::assignTypeHandler(Quatenary quat) // 赋值

{

// A := B

// 赋值操作，需要考虑立即数和变量两种情况；对于变量，则还需要将变量从内存换入

int A\_Reg = AVALUE[quat.result];

int B\_Reg = -1;

if (!is\_digit(quat.op1)) // 如果不是立即数，需要考虑寄存器分配问题

B\_Reg = AVALUE[quat.op1]; // 说明B的值在内存，但不需要换入，使用lw指令即可

if (A\_Reg <= 0) // 为零，说明是新变量，分配一个寄存器，但不能和B冲突

{

if (is\_digit(quat.op1))

A\_Reg = allocateReg(quat.result);

else

A\_Reg = allocateReg(quat.result, quat.op1);

}

//else if (A\_Reg == -1) // 为-1，说明值在内存，需要换入（实际上不需要换入，因为即将覆盖，需要分配一个寄存器）

// A\_Reg = loadVar(quat.result, B\_Reg);

// 进行赋值操作汇编的生成

if (is\_digit(quat.op1))

Asm\_stream << "addi " << registers[A\_Reg] << ",$zero," << quat.op1 << endl;

else if (B\_Reg != -1) // 如果B在寄存器中，使用move指令

Asm\_stream << "move " << registers[A\_Reg] << "," << registers[B\_Reg] << endl;

else // 如果B在内存中，使用lw指令

{

if (localVariantAddressTable.find(quat.op1) != localVariantAddressTable.end()) {

int offset = localVariantAddressTable[quat.op1];

Asm\_stream << "lw " << registers[A\_Reg] << "," << to\_string(offset) << "($fp)" << endl;

}

else if (globalVariantAddressTable.find(quat.op1) != globalVariantAddressTable.end()) {

int addr = globalVariantAddressTable[quat.op1];

Asm\_stream << "lw " << registers[A\_Reg] << "," << to\_string(addr) << "($zero)" << endl;

}

}

if (B\_Reg != -1)

RegisterInfo[B\_Reg].unusedTime = 0;

}

void Assembly\_Generator::paramTypeHandler(Quatenary quat) // 实参声明

{

// 实参声明时，将需要作为参数的值压入栈帧

// 栈帧的结构： sp↓

// | .. | 0 -4 -8 -12

// | locals | p1 p2 p3 ...

// 类似UNIX系统的栈帧结构

// 1. 准备工作：确保参数在寄存器中

int reg = AVALUE[quat.op1];

if (reg == -1) // 参数不在寄存器中，则将其从内存换至寄存器中

{

reg = loadVar(quat.op1);

if (localVariantAddressTable.find(quat.op1) != localVariantAddressTable.end()) {

int offset = localVariantAddressTable[quat.op1];

Asm\_stream << "lw " << registers[reg] << "," << to\_string(offset) << "($fp)" << endl;

}

else if (globalVariantAddressTable.find(quat.op1) != globalVariantAddressTable.end()) {

int addr = globalVariantAddressTable[quat.op1];

Asm\_stream << "lw " << registers[reg] << "," << to\_string(addr) << "($zero)" << endl;

}

}

// 2. 生成压栈汇编代码

Asm\_stream << "sw " << registers[reg] << ",0($sp)" << endl;

Asm\_stream << "subi $sp,$sp,4" << endl;

RegisterInfo[reg].unusedTime = 0;

++paramsNumber; // 为后续call函数处理时，对于指针的恢复移位提供偏移

}

void Assembly\_Generator::defparTypeHandler(Quatenary quat) // 形参声明

{

// 栈帧的结构： fp↓ sp↓

// | .. | 0 -4 -8 -12 -16 -20 -24 -28 -32 -36 -40

// | params | t0 t1 t2 t3 t4 t5 t6 t7 fp ra

// (old)

// 类似UNIX系统的栈帧结构

// 为形参找到对应的实参地址匹配

localVariantAddressTable.insert({ quat.result, 4 \* (paramsNumber + 1 + 8) }); // +8 指八个寄存器现场

++paramsNumber;

AVALUE.insert({quat.result, -1}); // 形参都在内存中，因此插入-1

}

void Assembly\_Generator::defprocTypeHandler(Quatenary quat) // 过程块声明

{

paramsNumber = 0;

procedureName = quat.operatorType;

localVariantAddressTable.clear();

localVarOffset = LOCAL\_VAR\_OFFSET;

//过程标号

Asm\_stream << quat.operatorType << " :" << endl;

}

int Assembly\_Generator::allocateReg(string A, string B, string C)

{

int A\_Reg = AVALUE[A];

if (A\_Reg > 0)

{

RegisterInfo[A\_Reg].unusedTime = 0;

return A\_Reg;

}

// 如果B的现行值在某个寄存器Ri中，RVALUE[Ri]中只包含B，

// 此外，或者B与A是同一个标识符，或者B的现行值在执行四元式A:=B op C之后不会再引用，

// 则选取Ri为所需要的寄存器R

int B\_Reg = 0, C\_Reg = 0;

if (B != "-")

B\_Reg = AVALUE[B];

if (C != "-")

C\_Reg = AVALUE[C];

if (B\_Reg > 0 && A == B)

{

RegisterInfo[B\_Reg].unusedTime = 0;

return B\_Reg;

}

// 如果有尚未分配的寄存器，则从中选取一个Ri为所需要的寄存器R

for (int i = 8; i <= 15; i++)

{

if (RVALUE[i].empty()) // 如果为空，说明空闲

{

AVALUE[A] = i;

RVALUE[i] = A; // 分配寄存器，修改RVALUE和AVALUE

RegisterInfo[i].unusedTime = 0;

return i;

}

}

int seized\_reg\_index = LRU\_GetRegister(B\_Reg, C\_Reg); // 如果没有B和C，则对应的reg为-1，不影响这里换寄存器

string M = RVALUE[seized\_reg\_index]; // 获得被抢占的变量名称

if (M == A && M != B && M != C) // 如果M是A，且不是B或C，不需要存数

;

else

saveVar(M);

// 更新AVALUE和RVALUE

AVALUE[M] = -1; // -1则说明已经被换到内存中

AVALUE[A] = seized\_reg\_index;

RVALUE[seized\_reg\_index] = A;

RegisterInfo[seized\_reg\_index].unusedTime = 0;

return seized\_reg\_index;

}

void Assembly\_Generator::saveVar(string var)

{

int reg\_index = AVALUE[var];

// 若var不在内存中，则需要先分配内存空间

if (localVariantAddressTable.find(var) == localVariantAddressTable.end() &&

globalVariantAddressTable.find(var) == globalVariantAddressTable.end())

{

// 过程块名称为""，说明是全局变量，否则是局部变量

if (procedureName != "")

{

localVariantAddressTable.insert({ var, localVarOffset });

localVarOffset -= 4; // 向上增长

Asm\_stream << "subi $sp,$sp,4" << endl; // 局部变量放在栈区

}

else

{

globalVariantAddressTable.insert({ var, globalVarAddr });

globalVarAddr += 4; // 向下增长

}

}

if (localVariantAddressTable.find(var) != localVariantAddressTable.end()) {

int offset = localVariantAddressTable[var];

Asm\_stream << "sw " << registers[reg\_index] << "," << to\_string(offset) << "($fp)" << endl;

}

else if (globalVariantAddressTable.find(var) != globalVariantAddressTable.end()) {

int addr = globalVariantAddressTable[var];

Asm\_stream << "sw " << registers[reg\_index] << "," << to\_string(addr) << "($zero)" << endl;

}

}

int Assembly\_Generator::loadVar(string var, int except1, int except2)

{

// 1. 查找可以换的寄存器，将寄存器中原本的内容（如果有）换到内存

int reg\_index = 0; // 下面选一个寄存器，序号存在这里

// 如果有尚未分配的寄存器，则从中选取一个Ri为所需要的寄存器R

for (int i = 8; i <= 15; i++)

{

if (RVALUE[i].empty()) // 如果为空，说明空闲

{

reg\_index = i;

break;

}

}

if (!reg\_index) // 没有空闲寄存器，则抢占一个

{

reg\_index = LRU\_GetRegister(except1, except2);

string M = RVALUE[reg\_index]; // 获得被抢占的变量名称

saveVar(M); // 将被抢占的变量存到内存

AVALUE[M] = -1; // -1则说明已经被换到内存中

}

// 2. 判断是全局变量还是局部变量，添加lw取数指令，将其真正换到寄存器内

AVALUE[var] = reg\_index;

RVALUE[reg\_index] = var; // 分配寄存器，修改RVALUE和AVALUE

RegisterInfo[reg\_index].unusedTime = 0;

if (localVariantAddressTable.find(var) != localVariantAddressTable.end()) {

int offset = localVariantAddressTable[var];

Asm\_stream << "lw " << registers[reg\_index] << "," << to\_string(offset) << "($fp)" << endl;

}

else if (globalVariantAddressTable.find(var) != globalVariantAddressTable.end()) {

int addr = globalVariantAddressTable[var];

Asm\_stream << "lw " << registers[reg\_index] << "," << to\_string(addr) << "($zero)" << endl;

}

return reg\_index;

}

int Assembly\_Generator::LRU\_GetRegister(int except1 = -1, int except2 = -1)

{

int seized\_reg\_index = -1;

{

int max\_unusedTime = -1; // 通过LRU算法，得到将要抢占的寄存器

for (int i = 8; i <= 15; i++) {

if (RegisterInfo[i].unusedTime > max\_unusedTime && except1 != i && except2 != i)

{

seized\_reg\_index = i;

max\_unusedTime = RegisterInfo[i].unusedTime;

}

}

}

return seized\_reg\_index;

}

void Assembly\_Generator::Print\_Assembly(string path)

{

ifstream AssemblyFile;

AssemblyFile.open(path, ios::in);

if (!AssemblyFile.is\_open())

{

cout << "没有找到汇编文件" << endl;

exit(EXIT\_FAILURE);

}

string s;

while (getline(AssemblyFile, s))

{

cout << s << endl;

}

AssemblyFile.close(); //关闭文件输入流

}

Assembly\_Generator::~Assembly\_Generator()

{

Asm\_stream.close();

}

### **7.2.8 main.cpp**

#include <iostream>

#include <iomanip>

#include <sstream>

#include "1\_Lexical\_PreProcess.h"

#include "2\_Grammatical\_Analysis.h"

#include "2\_Grammatical\_LR1Grammer.h"

#include "4\_Assembly\_Generator.h"

using namespace std;

extern void lexAnalysis(const string path); // 词法分析接口函数

extern vector<TOKEN>lexAnswer; // 1、词法分析处理结果

map<pair<int, string>, pair<string, int>> LR1Table; // 2、LR1分析表

std::vector<productItem>productionItems; // 3、产生式

SemanticAnalysis semantic; // 4、语义分析

string START\_SIGN = "Program";// 原文法的起始符

// 打印系统头部信息

void printInfo\_start() {

system("cls");

cout << endl;

cout << " ██████╗ ██████╗ ██████╗ ███╗ ███╗██████╗ ██╗██╗ ███████╗██████╗ " << endl;

cout << " ██╔════╝ ██╔════╝██╔═══██╗████╗ ████║██╔══██╗██║██║ ██╔════╝██╔══██╗" << endl;

cout << " ██║ ██║ ██║ ██║██╔████╔██║██████╔╝██║██║ █████╗ ██████╔╝" << endl;

cout << " ██║ ██║ ██║ ██║██║╚██╔╝██║██╔═══╝ ██║██║ ██╔══╝ ██╔══██╗" << endl;

cout << " ╚██████╗ ╚██████╗╚██████╔╝██║ ╚═╝ ██║██║ ██║███████╗███████╗██║ ██║" << endl;

cout << " ╚═════╝ ╚═════╝ ╚═════╝ ╚═╝ ╚═╝╚═╝ ╚═╝╚══════╝╚══════╝╚═╝ ╚═╝" << endl;

cout << endl << " 类C语言编译器 -- 2151127 华洲琦 " << endl << endl;

cout << "------------------------------- 输 入 文 件 -------------------------------" << endl;

cout << " " << endl;

cout << "高级语言程序：sourceProgram.c " << endl;

cout << "语法文件： grammer.txt " << endl;

cout << " " << endl;

cout << "------------------------------- 输 出 文 件 -------------------------------" << endl;

cout << " " << endl;

cout << "词法分析结果：1\_Lexical\_AnalysisAnswer.txt " << endl;

cout << "语法分析文法：2\_Grammatical\_Grammer.txt " << endl;

cout << "LR1分析表： 2\_Grammatical\_LR1Table.txt " << endl;

cout << "语法树脚本： syntaxTree.dot " << endl;

cout << "语法树图片： syntaxTree.png " << endl;

cout << "中间四元式： 3\_Semantic\_Quaternary.txt " << endl;

cout << " " << endl;

cout << "---------------------------------------------------------------------------" << endl;

cout << " " << endl;

system("pause");

}

// 词法分析打印二元组函数

void lexAnalysis\_printRes(string ch) {

if (ch == "y" || ch == "Y") {

for (TOKEN tempPair : lexAnswer) { // 打印词法分析得到的文法符号序列

cout << setw(15) << tempPair.token\_key << " " << setw(15) << tempPair.value <<

setw(10) << " ROW:" << setw(3) << tempPair.row << setw(10) << " COL:" << setw(3) << tempPair.col << endl;

}

}

}

// 词法分析控制函数

void lexAnalysis\_controller() {

system("cls");

cout << "------------------- STEP1 词法分析 --------------------" << endl;

cout << "正在进行词法分析……" << endl;

lexAnalysis("../infiles/sourceProgram.c"); // 词法分析

string ch;

cout << "词法分析成功！" << endl;

cout << "是否需要打印文法符号序列到屏幕？ y/Y - 是 ";

cin >> ch;

lexAnalysis\_printRes(ch);

system("pause");

}

// 语法分析控制函数

void GrammaticalAnalysis\_controller() {

system("cls");

cout << "----------------- STEP2 语法分析 ------------------" << endl;

LR1\_grammer g;

cout << "正在读取类C语言文法……" << endl;

g.grammer.readGrammer("../infiles/grammer.txt"); // 读类C语言文法文法

g.grammer.printGrammer("../outfiles/2\_Grammatical\_Grammer.txt"); // 打印文法（输出到文件）

cout << "正在生成FIRST集……" << endl;

g.generateFirst(); // 生成所有符号的first集

cout << "正在生成项目集族……" << endl;

g.generateClosureFamily(); // 生成项目集族（过程中会生成LR1分析表的移进部分和GoTo表部分）

g.fillGuiYueTable(); // 填写LR1分析表的归约部分

LR1Table = g.LR1\_table;

productionItems = g.grammer.productItems;

cout << "正在生成LR1分析表……" << endl;

g.printLR1Table("../outfiles/2\_Grammatical\_LR1Table.txt"); // 打印LR1分析表

// 语法分析（执行规约）+ 语法树绘制

ThreadedList\* Root = grammaticalAnalysis(lexAnswer, LR1Table, productionItems, semantic);

if (!Root) {

cout << "ERROR: 规约失败!" << endl;

exit(EXIT\_FAILURE);

}

else {

cout << "规约成功!" << endl;

}

cout << "正在绘制语法树……" << endl;

generateTreeDot(Root); // 绘制语法树

cout << "语法分析成功！" << endl;

system("pause");

}

// 语义分析控制函数

void SemanticAnalysis\_controller() {

system("cls");

cout << "------------------- STEP3 语义分析 -------------------" << endl;

//在语法分析中已经执行完毕了语义分析，此处负责显示结果

cout << "语义分析成功！" << endl;

cout << "中间代码生成成功！" << endl;

semantic.PrintQuaternary("../outfiles/3\_Semantic\_Quaternary.txt");

cout << "是否打印四元式结果到屏幕？ y/Y - 是 ";

string ch;

cin >> ch;

if (ch == "y" || ch == "Y") {

for (int i = 0; i < semantic.quaternary.size(); i++) {

cout << semantic.quaternary[i].idx << " (" << semantic.quaternary[i].operatorType << "," <<

semantic.quaternary[i].op1 << "," << semantic.quaternary[i].op2 << "," << semantic.quaternary[i].result << ")\n";

}

}

system("pause");

}

// 汇编代码生成

void AssemblyGeneration\_controller() {

system("cls");

cout << "------------------ STEP4 Assembly Generation ------------------" << endl;

cout << "正在进行汇编代码生成" << endl;

ifstream quat\_stream;

quat\_stream.open("../outfiles/3\_Semantic\_Quaternary.txt", ios::in);

Assembly\_Generator generator("../outfiles/4\_Assembly.asm", semantic.quaternary);

// Assembly\_Generator generator("../outfiles/4\_Assembly.asm", debug);

generator.generate();

cout << "汇编代码生成成功！" << endl;

cout << "---------------------------------------------------------------" << endl;

generator.Print\_Assembly("../outfiles/4\_Assembly.asm");

cout << "---------------------------------------------------------------" << endl;

system("pause");

}

// 编译成功后返回主页打印信息

void printInfo\_end() {

system("cls");

cout << endl;

cout << " \_\_\_ \_\_\_ \_\_\_ " << endl;

cout << " /\\ \\ |\\\_\_\\ /\\ \\ " << endl;

cout << " /::\\ \\ |:| | /::\\ \\ " << endl;

cout << " /:/\\:\\ \\ |:| | /:/\\:\\ \\ " << endl;

cout << " /::\\~\\:\\\_\_\\ |:|\_\_|\_\_ /::\\~\\:\\ \\" << endl;

cout << " /:/\\:\\ \\:|\_\_| /::::\\\_\_\\ /:/\\:\\ \\:\\\_\_\\ " << endl;

cout << " \\:\\~\\:\\/:/ / /:/~~/~ \\:\\~\\:\\ \\/\_\_/" << endl;

cout << " \\:\\ \\::/ / /:/ / \\:\\ \\:\\\_\_\\ " << endl;

cout << " \\:\\/:/ / \\/\_\_/ \\:\\ \\/\_\_/ " << endl;

cout << " \\::/\_\_/ \\:\\\_\_\\ " << endl;

cout << " ~~ \\/\_\_/ " << endl;

cout << endl << endl << "编译成功" << endl;

system("pause");

}

int main()

{

// 打印系统信息

printInfo\_start();

// 词法分析

lexAnalysis\_controller();

// 语法分析

GrammaticalAnalysis\_controller();

// 语义分析

SemanticAnalysis\_controller();

// 汇编代码生成

AssemblyGeneration\_controller();

// 返回主页

printInfo\_end();

return 0;

}

**END 感谢观看**