哈尔滨工业大学(深圳)

《编译原理》实验报告

学　院: 计算机科学与技术

姓　名: 刘睿

学　号: 220110720

专　业: 计算机科学与技术

日　期: 2024-11-24

1 实验目的与方法

实验目的总述

本次实验的主要目的是设计并实现一个用于支持 RISC-V 32M 指令集的 TXTv2 语言编译器。通过完成这一实验，可以加深对编译原理的理解，特别是词法分析、语法分析、语义分析以及目标代码生成的全过程。此外，本实验还旨在实践 RISC-V 指令集的应用，加深对计算机体系结构的认识，以及提高解决复杂问题的能力和实际编程技能。

实验方法总述

实验中，我们采用 Java 编程语言实现了编译器的各个模块，包括词法分析器、语法分析器、语义分析器以及目标代码生成器。目标平台为支持 RV32M 指令集的 RISC-V 32 模拟器，具体使用 RARS (RISC-V Assembler and Runtime Simulator) 进行目标代码的验证。在实现过程中，设计了基于 TXTv2 语言的测试用例，用以测试编译器的功能正确性，并通过 RARS 模拟执行这些生成的汇编代码，验证编译结果是否与预期一致。

1.1 词法分析器

实验目的

1. 理解词法分析在编译流程中的位置与功能，熟悉如何将源代码解析为一系列 Token。
2. 掌握有限自动机（DFA）在词法分析中的应用，学习如何通过编码表和符号表高效识别输入流中的关键字、标识符、常数、运算符和分界符。
3. 提升对类 C 语言语法规则的认识，加深对高级编程语言设计特点的理解。
4. 学会通过程序实现一个完整的词法分析流程，包括去除无用符号、分解正确单词并存储为二元组形式，最终为语法分析器的后续处理提供支持。

1.2 语法分析

实验目的

1. 学习和掌握 LR(1) 语法分析的原理，理解其作为自底向上、从左向右扫描的语法分析方法在编译器设计中的应用。
2. 掌握语法分析器中关键数据结构（如分析栈、状态表和动作表）的作用，并能够根据文法规则和输入字符串正确生成分析步骤。
3. 提升对上下文无关文法的理解，学会如何构造 LR(1) 分析表并通过代码实现自动化分析过程。
4. 通过实际编程练习，掌握如何将语法分析器与词法分析器结合，为后续的语义分析提供支持。

1.3 典型语句的语义分析及中间代码生成

实验目的

1. 理解自底向上语法制导翻译技术的原理，掌握如何对声明语句、赋值语句和算术运算语句进行正确的语义翻译。
2. 掌握语义分析的基本功能和实现原理，学会如何检查语义正确性（如类型匹配和作用域约束），确保语法分析后生成的结构符合语言规则。
3. 学习和实现中间代码的生成规则，理解中间代码在优化和目标代码生成过程中的重要作用，熟悉常用中间表示形式（如三地址码、抽象语法树等）。

1.4 目标代码生成

实验目的

1. 加深对编译器总体系结构的理解与掌握，明确目标代码生成在整个编译流程中的作用。
2. 学习和掌握常见 RISC-V 指令的使用方法，能够将中间代码正确翻译为 RISC-V 汇编代码，确保其符合目标平台的指令集规范。
3. 理解目标代码生成的算法，包括指令选择、寄存器分配与管理等，熟悉如何通过算法优化生成的目标代码，以提升代码运行效率。

2 实验内容及要求

2.1 词法分析器

实验内容及要求

本实验旨在实现一个可以解析类 C 语言程序的词法分析器，具体内容及要求如下：
1. 输入要求：词法分析器的输入是以文件形式存储的类 C 语言程序段，程序段需包含常见的关键字、标识符、常数、运算符和分界符等。
2. 处理流程：词法分析器需能够自动过滤源代码中的无用符号，例如空格和注释，并按照词法规则正确解析代码中的单词。
3. 输出要求：词法分析器的输出包括两个部分：
   - 一个 Token 序列，记录词法单元的类型及其对应的值；
   - 一个简单符号表，存储标识符及其相关信息。
4. 实现要求：
   - 使用有限状态自动机（DFA）来实现输入流的解析；
   - 支持类 C 语言的基本词法单元，包括关键字（如 if、while）、标识符、整数常量、运算符（如 +、-、*、/）、分界符（如 ;、{、}）等；

- 处理异常情况，确保程序的健壮性。例如，当输入中出现未识别的符号时，程序需要输出相应的错误提示。

2.2 语法分析

实验内容及要求

本实验的目标是实现一个基于 LR(1) 分析法的语法分析器，具体内容及要求如下：

1. 使用 LR(1) 分析法设计语法分析程序，对输入的单词符号串进行语法分析，确保输入符号符合预定义的语法规则。
2. 在分析过程中，输出推导过程中所用产生式序列，并将这些信息保存到指定的输出文件中，用于记录和验证分析过程。
3. 较低完成要求：实验模板代码需支持变量声明、变量赋值以及基本算术运算的文法。文法的基本功能需涵盖通用的语法结构，保证语法分析器能够处理基础用例。
4. 较高完成要求：学生需自行设计文法并完成实验，设计的文法需支持更复杂的语法结构，同时确保生成的分析表能够覆盖所有情况，保证语法分析器的正确性和完整性。
5. 实验要求：
   - 实验一的输出作为实验二的输入，输入的数据需为格式化的单词符号串；
   - 在复杂用例测试的情况下，学生需对复杂文法进行定义，并基于该文法自行生成 LR(1) 分析表，完成实验二的设计与验证。

2.3 典型语句的语义分析及中间代码生成

实验内容及要求

本实验的目标是实现典型语句的语义分析及中间代码生成，具体内容及要求如下：

1. 采用实验二中的文法，基于语法正确的单词符号串设计翻译方案，实现语法制导翻译，完成对输入程序段的语义分析。
2. 利用翻译方案，对所给的程序段进行逐步分析，生成对应的中间代码序列，同时更新和维护符号表，并将结果保存在指定的文件中。
3. 支持以下语义功能的实现和中间代码生成：
   - 声明语句：为变量分配内存空间并记录其相关信息；
   - 简单赋值语句：实现变量的赋值操作；
   - 算术表达式：解析表达式的语义并生成对应的计算指令。
4. 使用框架中的模拟器（IREmulator）对生成的中间代码进行验证，确保中间代码的正确性和执行的预期效果。

2.4 目标代码生成

实验内容及要求

本实验的目标是将实验三中生成的中间代码转换为目标代码（汇编指令）并验证其正确性。具体内容及要求如下：

1. 将实验三中生成的中间代码翻译为符合 RISC-V 指令集的目标代码（汇编指令）。翻译过程需严格遵循目标平台的指令规范，并确保生成的代码能够正确反映源程序的语义。
2. 使用 RARS（RISC-V Assembler and Runtime Simulator）运行生成的目标代码，并验证其运行结果的正确性。实验关注的是代码运行的实际结果是否符合预期，而非简单比较生成的汇编代码与参考标准文件的一致性。

3 实验总体流程与函数功能描述

3.1词法分析

3.1.1 编码表

编码表是词法分析器中的核心组件之一，用于对源代码中的不同词法单元（如关键字、运算符、标识符等）进行分类和编码。其作用是通过为每种词法单元分配一个唯一的编码值，方便后续的语法分析器对这些单元进行处理。

我们实验中预定义的编码表见 template/data/in/coding_map.csv（如图）

```
1 int
2 return
3 =
4 ,
5 Semicolon
6 +
7 -
8 *
9 /
10 (
11 )
51 id
52 IntConst
```

编码表为每个词法单元分配一个唯一的整数编号。例如，`int` 对应的编号是 1，`return` 对应的编号是 2，以此类推。这些编号在词法分析阶段用于标识源程序中的具体词法单元。

在词法分析阶段，当分析器读取到一个词法单元时，编码表用于快速查找该单元的类型和编号。编码后的词法单元以 Token 的形式输出，供语法分析阶段进一步处理。

通过编码表的设计，词法分析器能够高效地将源代码转换为一系列标准化的 Token，实现在词法单元的识别与分类，为后续的语法分析奠定基础。

3.1.2 正则文法

在多数编程语言中，单词的词法规则通常可以通过正则文法进行描述。基于正则文法的这种形式化描述为词法分析器的设计与实现提供了极大的便利。本实验中定义的正则文法如下：

1. 文法定义：
   文法由以下四部分组成：
   - 非终结符集合 V，包括 S, A, B, C, digit, no_0_digit, char；
   - 终结符集合 T，包括程序中使用的实际符号；
   - 生成规则集合 P；
   - 起始符号 S。
2. 符号约定：
   digit 表示数字，包括 0, 1, 2, ..., 9；
   no_0_digit 表示非零数字，包括 1, 2, ..., 9；
   letter 表示字母，包括 A, B, ..., Z, a, b, ..., z。
3. 生成规则：
   标识符：
   - S → letter A
   - A → letter A | digit A | ε

   运算符、分隔符：
   - S → B
   - B → + | - | * | / | (|)

   整数常数：
   - S → no_0_digit B
   - B → digit B | ε

   字符串常量：
   - S → "C"

   字符常量：
   - S → 'D'
4. 功能描述：
   正则文法定义了语言中的基本词法单元，包括标识符、运算符、分隔符、整数常数、字符串常量和字符常量。每种单词类型都具备清晰的生成规则，有助于词法分析器通过有限状态自动机（DFA）或其他方法高效识别这些单元。

3.1.3 状态转换图

状态转换图是词法分析器设计中的一个重要工具，用于描述有限状态自动机（DFA）的工作流程。通过状态转换图，可以清晰地表示输入字符与状态之间的关系，以及如何根据输入的字符流转换到不同的状态，从而实现对源代码中各类词法单元的识别。

本实验的状态转换图包括以下主要内容：
1. 起始状态：

所有的词法单元解析都从起始状态 `start` 开始。根据输入字符的类型（如字母、数字、空格或运算符等），转移到不同的状态。

2. 状态转换规则：

对于字母字符（`letter`），转移到状态 14，用于处理标识符；

对于数字字符（`digit`），转移到状态 16，用于处理无符号整数；

遇到 `=` 转移到状态 21，进一步判断是否为赋值操作或比较操作；

对于引号 `"`，转移到状态 24，处理字符串常量；

对于其他运算符（如 `+`, `-`, `*`, `/`），分别转移到各自的终止状态，用于返回对应的运算符 Token。

3. 终止状态：

每条路径的终点状态会返回一个对应的 `Token`，例如：

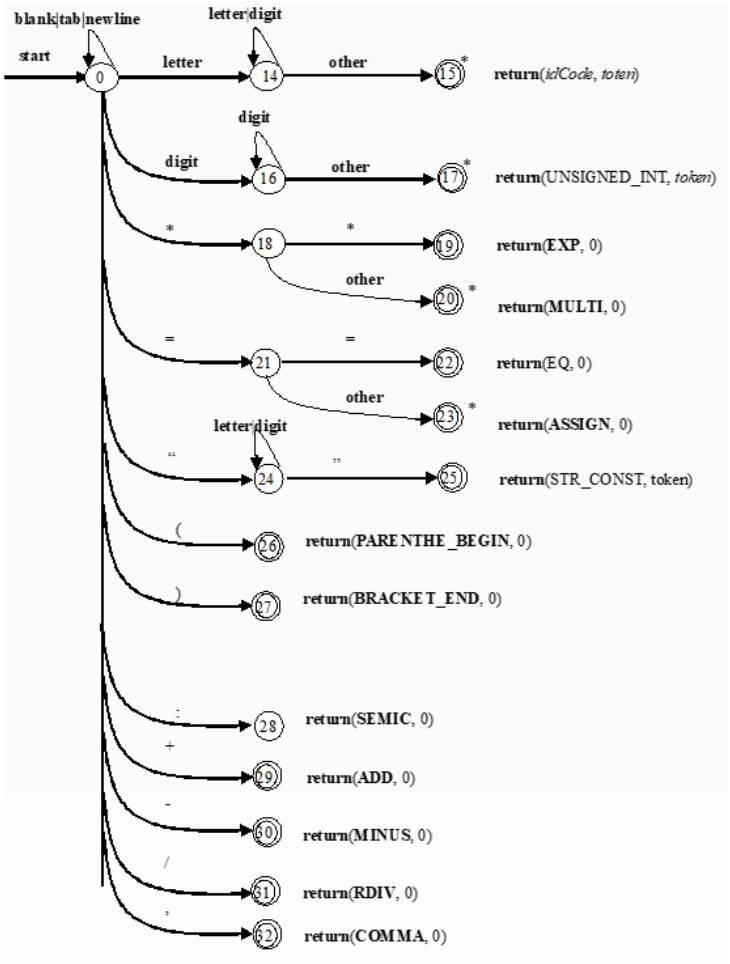
- 状态 15 返回标识符的 `idCode` 和标识符值；
- 状态 17 返回无符号整数 `Token`；
- 状态 18 返回指数符号 `EXP`；
- 状态 35 返回字符串常量 `Token`；
- 状态 28、29、30、31、32 等分别返回符号 `;`, `+`, `-`, `/`, `,` 的 `Token`。

4. 处理特殊输入：

空白字符（`blank`、`tab`、`newline`）会在起始状态被自动忽略，不会影响状态的转移。

输入无法匹配任何有效状态时，分析器会提示语法错误并停止解析。

具体转换图见指导书（如图）



3.1.4 词法分析程序设计思路和算法描述

① template/src/cn/edu/hitsz/compiler/lexer/LexicalAnalyzer.java

1. 文件加载

模块通过 `loadFile` 方法加载输入文件内容，将源代码存储为一个字符串 `sourceString`，为后续分析提供基础。该方法使用 `BufferedReader` 逐行读取文件内容并拼接，确保能够高效处理大文件。

```
38      public void loadFile(String path) {
39          // TODO: 词法分析前的缓冲区实现
40          // 可自由实现各类缓冲区
41          // 或直接采用完整读入方法
42          try (BufferedReader br = new BufferedReader(new FileReader(path))) {
43              String textLine = br.readLine();
44              while (textLine != null) {
45                  sourceString += textLine;
46                  textLine = br.readLine();
47              }
48
49          } catch (IOException e) {
50              throw new RuntimeException(e);
51          }
52      }
```

2. 词法分析流程

核心词法分析逻辑由 `run` 方法实现，逐字符扫描 `sourceString`，根据字符类型调用相应的处理方法

```
58      public void run() {
59          // 将源字符串转换为字符数组
60          char[] word = sourceString.toCharArray();
61          int i = 0;
62
63          while (i < sourceString.length()) {
64              char ch = word[i];
```

具体流程包括：

- 跳过空白字符（如空格、换行符等）。

```
66              // 跳过空白字符
67              if (isWhitespace(ch)) {
68                  i++;
69                  continue;
70              }
```

- 识别单字符标点符号（如 `;`、`+`、`-` 等），生成对应的 Token。

```
72              // 判断是否为单字符标点符号
73              if (isSinglePunctuation(ch)) {
74                  if (ch == ';') {
75                      tokenList.add(Token.simple("Semicolon"));
76                  } else {
77                      tokenList.add(Token.simple(String.valueOf(ch)));
78                  }
79                  i++;
80                  continue;
81              }
```

- 处理以字母开头的序列，区分关键字和标识符。

```
113         // 处理字母标识符
114         private int processLetter(int i, char[] word) {
115             int start = i;
116             while (i < word.length && Character.isLetter(word[i])) {
117                 i++;
118             }
119             String key = sourceString.substring(start, i);
120 
121             if (TokenKind.isAllowed(key)) {
122                 tokenList.add(Token.simple(key));
123             } else {
124                 tokenList.add(Token.normal("id", key));
125                 if (!symbolTable.has(key)) {
126                     symbolTable.add(key);
127                 }
128             }
129             return i;
130         }
```

- 处理以数字开头的序列，解析为整数常量。

```
132         // 处理数字常量
133         private int processDigit(int i, char[] word) {
134             int start = i;
135             while (i < word.length && Character.isDigit(word[i])) {
136                 i++;
137             }
138             String digit = sourceString.substring(start, i);
139             tokenList.add(Token.normal("IntConst", digit));
140             return i;
141         }
```

- 遇到文件结束时添加 `EOF` Token 标记。

3. 标识符和数字处理

标识符处理：通过 `processLetter` 方法实现，提取由字母和数字组成的标识符字符串。判断其是否为关键字（使用 `TokenKind.isAllowed` 方法），若为关键字则生成对应 Token；否则生成普通标识符 Token 并加入符号表。

```
113         // 处理字母标识符
114         private int processLetter(int i, char[] word) {
115             int start = i;
116             while (i < word.length && Character.isLetter(word[i])) {
117                 i++;
118             }
119             String key = sourceString.substring(start, i);
120 
121             if (TokenKind.isAllowed(key)) {
122                 tokenList.add(Token.simple(key));
123             } else {
124                 tokenList.add(Token.normal("id", key));
125                 if (!symbolTable.has(key)) {
126                     symbolTable.add(key);
127                 }
128             }
129             return i;
130         }
```

数字处理：通过 `processDigit` 方法实现，提取由连续数字组成的整数常量，并生成 `IntConst` 类型的 Token。

```
132         // 处理数字常量
133         private int processDigit(int i, char[] word) {
134             int start = i;
135             while (i < word.length && Character.isDigit(word[i])) {
136                 i++;
137             }
138             String digit = sourceString.substring(start, i);
139             tokenList.add(Token.normal("IntConst", digit));
140             return i;
141         }
```

4. 符号表管理

模块与 `SymbolTable` 紧密集成，用于管理标识符信息。当遇到新的标识符时，通过 `SymbolTable.add` 方法将其加入符号表；在检查标识符是否已存在时，调用 `SymbolTable.has` 方法。符号表的设计确保了标识符的唯一性和便捷管理。

5. Token 列表获取与输出

通过 `getTokens` 方法返回词法分析生成的 Token 列表。

使用 `dumpTokens` 方法将 Token 列表写入指定文件，便于调试和验证。

② template/src/cn/edu/hitsz/compiler/symtab/SymbolTable.java

1. 符号表的核心结构

符号表使用 HashMap 作为底层数据结构，键为标识符的字符串表示，值为 SymbolTableEntry 对象，存储标识符的详细信息。这种设计具备以下优势：

• 快速查找：HashMap 的时间复杂度为 O(1)，能够高效支持大规模符号管理。

• 灵活存储：通过 SymbolTableEntry 对象，可以扩展存储标识符的更多属性，如类型、作用域等。

```
20          // 用字典作为符号表
21          public Map<String, SymbolTableEntry> table = new HashMap<>();
```

2. 符号表的主要功能

模块提供以下核心功能：

新增符号（add 方法）：检查标识符是否已存在，若不存在则将其加入符号表，并返回对应的条目。若已存在，则抛出异常以避免重复添加。

```
45      public SymbolTableEntry add(String text) {
46          if (has(text)) {
47              throw new RuntimeException();
48          } else {
49              SymbolTableEntry tableEntry = new SymbolTableEntry(text);
50              table.put(text, tableEntry);
51              return tableEntry;
52          }
53      }
```

查询符号（get 方法）：根据标识符字符串查询对应的条目，若未找到则抛出异常。

```
30      public SymbolTableEntry get(String text) {
31          if (has(text)) {
32              return table.get(text);
33          } else {
34              throw new RuntimeException();
35          }
36      }
```

检查符号存在性（has 方法）：判断符号表中是否已包含指定标识符，返回布尔值。

```
61      public boolean has(String text) {
62          // 利用 Java 哈希表的方法能很简单完成
63          return table.containsKey(text);
64      }
```

获取所有条目（getAllEntries 方法）：返回符号表中所有条目，为后续输出和处理提供支持。

```
71      private Map<String, SymbolTableEntry> getAllEntries() {
72          return table;
73      }
```

3. 符号表内容输出

通过 dumpTable 方法，符号表内容可以以 (标识符, 类型) 的格式输出到指定文件路径。

③ template/src/cn/edu/hitsz/compiler/lexer/Token.java

（Token 类已给出完整代码，无需补充，简单介绍如下）

`Token` 类是词法分析模块的核心组件之一，用于表示词法分析的结果。词法分析器将源代码中的字符序列解析为结构化的词法单元，每个单元由 `Token` 类实例化

1. 核心功能

Token 表示源代码中的一个词法单元，由类型（TokenKind）和词素（lexeme）组成。
类型（TokenKind）描述了该单元的类别（如关键字、标识符、数字等）。
词素（text）是对应的字符内容，表示实际的源文本值（如标识符名称或数字值）。

2. 主要方法

eof()：生成表示文件结束的特殊 Token，用于标记词法分析的结束。
simple()：生成简单的 Token，如关键字或标点符号，不需要额外的文本信息。
normal()：生成带有额外文本信息的 Token，如标识符或数字字面量。
getKind()：获取 Token 的类型。
getText()：获取 Token 的词素文本。
toString()：返回 Token 的字符串表示，格式为 (类型, 文本)。

3.2 语法分析

3.2.1 拓展文法

对于某些文法，若在生成式的右部包含文法开始符号，则在归约过程中需要明确判断是否已经归约到文法的初始起点。为此，需要对原有文法进行拓展。本次实验的拓展文法框架已在文件 template/data/in/grammar.txt 中提供，具体内容如下：

```
P -> S_list;
S_list -> S Semicolon S_list;
S_list -> S Semicolon;
S -> D id;
D -> int;
S -> id = E;
S -> return E;
E -> E + A;
E -> E - A;
E -> A;
A -> A * B;
A -> B;
B -> ( E );
B -> id;
B -> IntConst;
```

3.2.2 LR1 分析表

LR(1) 分析表是用于指导语法分析器进行自底向上语法分析的核心工具，其由动作表（Action）和转移表（Goto）两部分组成：

1. 动作表（Action）
   指定当前状态下根据输入符号进行的操作，包括：
   - 移入（Shift）：将符号移入栈并转移到新状态。
   - 归约（Reduce）：依据产生式对栈内容进行归约。
   - 接受（Accept）：表示分析完成，输入合法。
   - 错误（Error）：输入不符合文法规则。
2. 转移表（Goto）
   指定在非终结符的情况下，状态栈如何进行转移，常用于归约后跳转到目标状态。

本次实验的 LR1 分析表已在文件 template/data/in/LR1_table.csv 中提供，部分内容如下：

状态	ACTION												GOTO				
	id	(	)	+	-	*	=	int	return	IntConst	Semicolon	$	E	S_list	S	A	B	D
0	shift 4							shift 5	shift 6					1	2			3
1												accept						
2											shift 7							
3	shift 8																	
4							shift 9											
5	reduce D -> int																	
6	shift 13	shift 14								shift 15			10			11	12	
7	shift 4							shift 5	shift 6			reduce S_list -> S Semicolon		16	2			3

此外，本次实验可以通过编译工作台自主构建自己的 LR1 分析表

3.2.3 状态栈和符号栈的数据结构和设计思路

① template/src/cn/edu/hitsz/compiler/parser/SyntaxAnalyzer.java

1. 数据结构设计

• 状态栈和符号栈:

```
27          private final Stack<Symbol> tokenStack = new Stack<>();   //符号栈
28          private final Stack<Status> statusStack = new Stack<>();   //状态栈
```

状态栈用于存储分析过程中的状态，符号栈用于存储当前处理的符号（Token 或非终结符）。这两者共同驱动语法分析过程。

• LR 分析表:

```
26          private LRTable lrtable;
```

LR 分析表保存了文法的 Action 和 Goto 信息，用于查找当前状态下的操作或状态转移。

• 词法单元序列:

```
25          private List<Token> tokenList = new ArrayList();
```

`tokenList` 存储所有输入的词法单元，通过索引 `tokenIndex` 实现按需读取和处理。

2. 加载数据

• 加载词法单元

```
79      public void loadTokens(Iterable<Token> tokens) {
80          // TODO: 加载词法单元
81          // 你可以自行选择要如何存储词法单元，譬如使用迭代器，或是栈，或是干脆使用一个 list 全存起来
82          // 需要注意的是，在实现驱动程序的过程中，你会需要面对只读取一个 token 而不能消耗它的情况，
83          // 在自行设计的时候请加以考虑此种情况
84          // throw new NotImplementedException();
85          tokens.forEach(tokenList::add);
86      }
```

该方法将词法单元序列加载到分析器中，为后续的语法分析做好准备。

• 加载 LR 分析表

```
88      public void loadLRTable(LRTable table) {
89          // TODO: 加载 LR 分析表
90          // 你可以自行选择要如何使用该表格:
91          // 是直接对 LRTable 调用 getAction/getGoto，抑或是直接将 initStatus 存起来使用
92          // throw new NotImplementedException();
93          this.lrtable = table;
94      }
```

通过加载预先构造的 LR 分析表，提供分析器所需的所有转移和归约规则。

3.2.4 LR 驱动程序设计思路和算法描述

① template/src/cn/edu/hitsz/compiler/parser/SyntaxAnalyzer.java

语法分析核心逻辑

• 初始化

```
96      public void run() {
97          // TODO: 实现驱动程序
98          // 你需要根据上面的输入来实现 LR 语法分析的驱动程序
99          // 请分别在遇到 Shift, Reduce, Accept 的时候调用上面的 callWhenInShift, callWhenInReduce, callWhenInAccept
100         // 否则用于为实验二打分的产生式输出可能不会正常工作
101
102         // 初始化符号栏和状态栏
103         statusStack.push(lrtable.getInit());
104         tokenStack.push(new Symbol(Token.eof()));
105
106         int currentIndex = 0;
107         boolean running = true;
```

语法分析以状态栈初始化为 `0`、符号栈初始化为 `EOF` 符号开始。通过读取 `tokenList` 获取第一个词法单元。

• 主循环

```
109         while (running && currentIndex < tokenList.size()) {
110             Token currentToken = tokenList.get(currentIndex);
111             Status currentStatus = statusStack.peek();
112             Action currentAction = lrtable.getAction(currentStatus, currentToken);
113
114             switch (currentAction.getKind()) {
115                 case Shift -> {
116                     callWhenInShift(currentStatus, currentToken);
117                     statusStack.push(currentAction.getStatus());
118                     tokenStack.push(new Symbol(currentToken));
119                     currentIndex++;
120                 }
121                 case Reduce -> {
122                     Production currentProduction = currentAction.getProduction();
123                     callWhenInReduce(currentStatus, currentProduction);
124                     for (int j = 0; j < currentProduction.body().size(); j++) {
125                         statusStack.pop();
126                         tokenStack.pop();
127                     }
128                     Status gotoStatus = lrtable.getGoto(statusStack.peek(), currentProduction.head());
129                     statusStack.push(gotoStatus);
130                     tokenStack.push(new Symbol(currentProduction.head()));
131                 }
132                 case Error -> {
133                     System.err.println("Error occurred at token: " + currentToken);
134                     running = false;
135                 }
136                 case Accept -> {
137                     callWhenInAccept(currentStatus);
138                     running = false;
139                 }
140                 default -> running = false;
141             }
142         }
```

- 如果动作为 `Shift`（如 "S3"），则将当前 Token 压入符号栈，同时状态栈转移到新状态。
- 如果动作为 `Reduce`（如 "R2"），根据文法规则弹出符号栈和状态栈的对应元素，将产生式左部压入符号栈，并跳转到 Goto 表的目标状态。
- 如果动作为 `ACC`，表示语法分析完成。
- 如果动作不存在，打印语法错误信息。

② template/src/cn/edu/hitsz/compiler/parser/Symbol.java

在语法分析阶段，为了统一处理 Token 和非终结符（NonTerminal），Symbol 类被设计为一个抽象层，能够同时容纳 Token 和非终结符，同时便于管理它们的附加信息（如类型和值）。

具体设计如下：

```java
public class Symbol {
    private final Token token;
    private final NonTerminal nonTerminal;
    private SourceCodeType type = null;
    private IRValue value = null;

    private Symbol(Token token, NonTerminal nonTerminal) {
        if (token != null && nonTerminal != null) {
            throw new IllegalArgumentException("Symbol cannot be both a token and a non-terminal.");
        }
        this.token = token;
        this.nonTerminal = nonTerminal;
    }

    public Symbol(Token token) {
        this(token, null);
    }

    public Symbol(NonTerminal nonTerminal) {
        this(null, nonTerminal);
    }

    public Token getToken() {
        return token;
    }

    public NonTerminal getNonTerminal() {
        return nonTerminal;
    }

    public boolean isToken() {
        return token != null;
    }

    public boolean isNonTerminal() {
        return nonTerminal != null;
    }

    public SourceCodeType getType() {
        return type;
    }

    public void setType(SourceCodeType type) {
        this.type = type;
    }

    public IRValue getValue() {
        return value;
    }

    public void setValue(IRValue value) {
        this.value = value;
    }

    @Override
    public String toString() {
        if (isToken()) {
            return "Token: " + token.toString();
        } else if (isNonTerminal()) {
            return "NonTerminal: " + nonTerminal.toString();
        }
        return "Empty Symbol";
    }
}
```

3.3 语义分析和中间代码生成

3.3.1 翻译方案

翻译方案采用基于 S 属性的翻译模式，通过在语法规则中嵌入语义动作，实现对变量声明、赋值语句以及表达式的语义处理。

输入语句为：
int a;
a = 3;

在语法分析过程中，产生式的归约序列为：
S -> D id
D -> int
S -> id = E
E -> A
A -> B
B -> IntConst

对于每条产生式的语义动作：
1. S -> D id
检查变量 id 是否在符号表中，若不存在，则将变量及其类型（D.type）存入符号表；若已存在，则报错。
2. D -> int
将 D 的类型属性设为 int。
3. S -> id = E
生成三地址指令 id = E，表示赋值语句的中间代码。
4. E -> A
将表达式 E 的值属性继承自 A 的值属性。
5. A -> B
将 A 的值属性继承自 B 的值属性。
6. B -> IntConst
将常量 IntConst 的字面值赋给 B 的值属性。

3.3.2 语义分析和中间代码生成的数据结构

① template/src/cn/edu/hitsz/compiler/parser/Symbol.java

语法分析中设计的 Symbol 类中添加如下两个属性：

```
11          private SourceCodeType type = null;
12          private IRValue value = null;
```

1. SourceCodeType 属性

作用：用于记录符号的源代码类型信息。
具体功能：
当符号是一个变量或表达式时，`SourceCodeType` 保存该变量或表达式的类型（如 `int`、`float` 等）。
在语义分析阶段，`SourceCodeType` 可以帮助检查类型匹配，例如赋值语句中左右两侧类型是否一致，算术运算中操作数类型是否兼容。

2. IRValue 属性
作用：用于记录符号对应的中间代码值。
具体功能：
在中间代码生成阶段，每个符号可能会被分配一个临时变量地址或具体值，`IRValue` 保存这些信息。
对于变量符号，`IRValue` 通常表示其在中间代码中的存储位置。
对于常量符号，`IRValue` 表示其实际的字面值。

② template/src/cn/edu/hitsz/compiler/parser/SemanticAnalyzer.java

```
17          private SymbolTable symbolTable;
18          private final Stack<Symbol> tokenStack = new Stack<>();
```

1. SymbolTable 符号表
用于存储源代码中的标识符及其相关属性（如类型、值、作用域等），在语义分析阶段为类型检查、变量管理等提供支持。
标识符查找：通过标识符的名称，从符号表中检索其相关信息。
属性更新：在语义分析中，根据语法归约的结果更新符号的属性，如类型。

2. tokenStack 符号栈
用于在语法分析过程中存储正在处理的 Token 或非终结符，辅助归约操作和语义分析。
栈操作：提供典型的栈操作，包括压栈（push）、弹栈（pop）和访问栈顶元素。
存储类型：栈中元素类型为 Symbol，包含 Token 或非终结符，以及其相关的属性（如类型、值）。

③ template/src/cn/edu/hitsz/compiler/parser/IRGenerator.java

```
19          public SymbolTable symbolTable;
20          private final Stack<Symbol> tokenStack = new Stack<>();
21          private final List<Instruction> irList = new ArrayList<>();
```

1. 符号栈（tokenStack）
符号栈是语法分析的核心数据结构之一，用于在归约和移入操作中存储当前的符号（终结符或非终结符）。
在 whenShift 中，当前 Token 被包装成 Symbol 对象后压入栈，供后续的语义和 IR 生成阶段使用。
在 whenReduce 中，根据产生式的右部符号弹出对应数量的元素，并通过生成 IR

指令将非终结符结果重新压入栈。

2. IR 指令列表（irList）

IR 指令列表是生成的中间代码的存储结构，用于存放程序的中间表示。

在 whenReduce 中，根据具体的归约规则，生成对应的 IR 指令并添加到列表中。

在代码生成完成后，可以通过 getIR 方法获取所有 IR 指令，或者通过 dumpIR 方法将指令输出到文件。

3. 符号表（symbolTable）

符号表存储程序中所有变量的类型和作用域信息，提供对标识符的查询和管理功能。

在 whenReduce 中，符号表用于设置标识符的类型（如变量声明的规则 S -> D id）。

提供全局访问标识符的能力，确保程序中变量的类型一致性。

3.3.3 语法分析程序设计思路和算法描述

① template/src/cn/edu/hitsz/compiler/parser/SemanticAnalyzer.java

语法分析程序采用自底向上的 LR 分析方法，通过状态栈和符号栈的配合，结合预先构造的 LR 分析表，完成对输入 Token 流的语法解析。其核心任务包括识别合法的语法结构并在适当的时机触发语义操作

1. 移入操作（Shift）

- 从输入流中读取下一个 Token，将其压入符号栈。
- 查找 Action 表，获取当前状态和当前符号的下一个状态，将该状态压入状态栈。

```
53      public void whenShift(Status currentStatus, Token currentToken) {
54          // TODO: 该过程在遇到 shift 时要采取的代码动作
55          Symbol curSymbol = new Symbol(currentToken);
56          if(Objects.equals(currentToken.getKindId(), "int")){
57              curSymbol.setType(SourceCodeType.Int);
58          }
59          tokenStack.push(curSymbol);
60      }
```

2. 归约操作（Reduce）

- 根据 Action 表中对应的归约规则，确定使用的产生式。
- 弹出符号栈和状态栈中与产生式右部匹配的符号和状态。
- 将产生式左部的非终结符压入符号栈。
- 使用 Goto 表获取下一状态，并压入状态栈。

```
27      public void whenReduce(Status currentStatus, Production production) {
28          // TODO: 该过程在遇到 reduce production 时要采取的代码动作
29          switch (production.index()) {
30              case 4 -> { // S -> D id
31                  Symbol idSymbol = tokenStack.pop();   // 弹出 id
32                  Symbol dSymbol = tokenStack.pop();    // 弹出 D
33                  // 更新符号表中 id 的类型
34                  symbolTable.get(idSymbol.getToken().getText()).setType(dSymbol.getType());
35                  tokenStack.push(new Symbol(production.head()));
36              }
37              case 5 -> { // D -> int
38                  tokenStack.pop();
39                  Symbol nonTerminal = new Symbol(production.head());
40                  nonTerminal.setType(SourceCodeType.Int);
41                  tokenStack.push(nonTerminal);
42              }
43              default -> {
44                  for (int i = 0; i < production.body().size(); i++) {
45                      tokenStack.pop();
46                  }
47                  tokenStack.push(new Symbol(production.head()));
48              }
49          }
50      }
```

3. 接受操作（Accept）

- 检查到 Action 表中的 `ACC` 项，表示语法分析完成，输入合法。

```
21      public void whenAccept(Status currentStatus) {
22          // TODO: 该过程在遇到 Accept 时要采取的代码动作
23          // Accept 状态下无需执行运行任何动作
24      }
```

② template/src/cn/edu/hitsz/compiler/parser/IRGenerator.java

IRGenerator 的设计目的是在语法分析阶段生成中间代码（IR），结合符号栈和符号表，依据产生式规则完成语义处理和 IR 指令的生成

1. 状态处理

• 移入操作（whenShift）：

将 Token 包装为 Symbol 并压入栈。

对于常量类型的 Token，生成立即数 IR 表示（IRImmediate）。

对于标识符类型的 Token，生成变量 IR 表示（IRVariable）。

```
24      public void whenShift(Status currentStatus, Token currentToken) {
25          // TODO
26          Symbol curSymbol = new Symbol(currentToken);
27          if (currentToken.getText().matches("^[0-9]+$")) {
28              curSymbol.setValue(IRImmediate.of(Integer.parseInt(currentToken.getText())));
29          } else {
30              curSymbol.setValue(IRVariable.named(currentToken.getText()));
31          }
32          tokenStack.push(curSymbol);
33      }
```

• 归约操作（whenReduce）：

根据生产式执行语义动作，生成对应的 IR 指令。

更新符号栈，将归约结果（非终结符）压入栈。

```
36      public void whenReduce(Status currentStatus, Production production) {
37          // TODO
38          // Reduce 时生成相应的 IR 指令
39          Symbol curNonTerminal = new Symbol(production.head());
40          IRVariable valueTemp;
41          Symbol lhs, rhs;
```

(IR 生成逻辑（whenReduce 方法）具体将在后面说明)

• 接受状态（whenAccept）：
在接受状态下，无需执行额外动作。

```
104         public void whenAccept(Status currentStatus) {
105             // TODO
106             // Accept 时不需要进行动作
107         }
```

2. IR 生成逻辑（whenReduce 方法）

whenReduce 根据当前生产式生成对应的 IR 指令。每个生产式通过 switch-case 结构处理，并执行特定的 IR 生成操作。

• 规则 S -> id = E：处理赋值语句，将表达式 E 的值赋给变量 id。

从符号栈中弹出 rhs（E 的结果）、= 符号，以及 lhs（目标变量 id）。

获取目标变量的中间表示（lhs.getValue()），并确保其类型为 IRVariable。
生成赋值指令（MOV），将 rhs 的值赋给 lhs，并加入 IR 列表。

```
44              case 6 -> { // S -> id = E
45                  rhs = tokenStack.pop();
46                  tokenStack.pop(); // 弹出 "="
47                  lhs = tokenStack.pop();
48                  valueTemp = (IRVariable) lhs.getValue();
49                  irList.add(Instruction.createMov(valueTemp, rhs.getValue()));
50              }
```

• 规则 S -> return E：处理返回语句，生成 RET 指令。

从符号栈中弹出 rhs（返回值）和 return 符号。

创建返回指令（RET），将 rhs 的值作为返回值。

```
51              case 7 -> { // S -> return E
52                  rhs = tokenStack.pop();
53                  tokenStack.pop(); // 弹出 "return"
54                  irList.add(Instruction.createRet(rhs.getValue()));
55              }
```

• 规则 E -> E + A 和 E -> E – A: 计算两个表达式 lhs 和 rhs 的加法或减法，结果存储在一个临时变量中。

从符号栈中弹出 rhs（右操作数）、运算符，以及 lhs（左操作数）。

创建一个临时变量（IRVariable.temp()）存储结果。

生成加法或减法指令（ADD 或 SUB），将结果存储到临时变量中。
将临时变量设置为非终结符的值属性。

```
56              case 8 -> { // E -> E + A
57                  rhs = tokenStack.pop();
58                  tokenStack.pop(); // 弹出 "+"
59                  lhs = tokenStack.pop();
60                  valueTemp = IRVariable.temp();
61                  irList.add(Instruction.createAdd(valueTemp, lhs.getValue(), rhs.getValue()));
62                  curNonTerminal.setValue(valueTemp);
63              }
64              case 9 -> { // E -> E - A
65                  rhs = tokenStack.pop();
66                  tokenStack.pop(); // 弹出 "-"
67                  lhs = tokenStack.pop();
68                  valueTemp = IRVariable.temp();
69                  irList.add(Instruction.createSub(valueTemp, lhs.getValue(), rhs.getValue()));
70                  curNonTerminal.setValue(valueTemp);
71              }
```

- 规则 A -> A * B：计算两个操作数的乘法，结果存储在临时变量中。

从符号栈中弹出右操作数（rhs）、乘号，以及左操作数（lhs）。创建临时变量存储乘法结果。生成乘法指令（MUL），将结果存储到临时变量中。设置非终结符的值属性为临时变量。

```
72              case 11 -> { // A -> A * B
73                  rhs = tokenStack.pop();
74                  tokenStack.pop(); // 弹出 "*"
75                  lhs = tokenStack.pop();
76                  valueTemp = IRVariable.temp();
77                  irList.add(Instruction.createMul(valueTemp, lhs.getValue(), rhs.getValue()));
78                  curNonTerminal.setValue(valueTemp);
79              }
```

- 规则 E -> A, A -> B, B -> id：直接传递子符号的值到非终结符。

弹出栈顶符号，将其值传递给非终结符。

```
80              case 10, 12, 14 -> { // E -> A, A -> B, B -> id
81                  curNonTerminal.setValue(tokenStack.pop().getValue());
82              }
```

- 规则 B -> (E)：解析括号表达式，直接传递内部表达式的值。

从符号栈中弹出右括号、表达式 E 以及左括号。将表达式 E 的值赋给非终结符。

```
83              case 13 -> { // B -> ( E )
84                  tokenStack.pop(); // 弹出 ")"
85                  rhs = tokenStack.pop(); // 弹出 E
86                  tokenStack.pop(); // 弹出 "("
87                  curNonTerminal.setValue(rhs.getValue());
88              }
```

- 规则 B -> IntConst: 处理常量值，直接将常量的值属性传递给非终结符。

弹出栈顶符号（常量），将其值传递给非终结符。

```
89              case 15 -> { // B -> IntConst
90                  rhs = tokenStack.pop();
91                  curNonTerminal.setValue(rhs.getValue());
92              }
```

- 其他规则（default）：对于不需要特殊处理的规则，仅进行栈操作。

根据产生式右部符号的数量，弹出栈顶对应数量的符号。

```
93              default -> {
94                  for (int i = 0; i < production.body().size(); i++) {
95                      tokenStack.pop();
96                  }
97              }
```

- 归约结果入栈：作用：将归约产生的新非终结符压入栈，供后续语法分析和 IR 生成使用。

```
99              tokenStack.push(curNonTerminal);
```

3.4 目标代码生成

3.4.1 设计思路和算法描述

① template/src/cn/edu/hitsz/compiler/asm/AssemblyGenerator.java

`AssemblyGenerator` 的主要功能是将中间代码（Intermediate Representation, IR）翻译成 RISC-V 汇编代码，同时完成寄存器的高效分配。以下融合设计思路和算法分析，对其各个部分逐一深入剖析。

1. 中间代码加载与预处理

方法 loadIR 遍历输入的中间代码 originInstructions。

根据指令类型（InstructionKind）对指令分类：

- isReturn：直接添加到队列并终止加载（函数返回的特殊处理）。
- IsUnary：单操作数指令无需修改，直接加入队列。
- isBinary：二元操作指令调用 processBinaryInstruction 进一步处理。

```
59      public void loadIR(List<Instruction> originInstructions) {
60          // TODO: 读入前端提供的中间代码并生成所需要的信息
61          // 遍历中间代码
62          for (Instruction instruction : originInstructions) {
63              InstructionKind instructionKind = instruction.getKind();
64
65              if (instructionKind.isReturn()) {
66                  // 如果是RET指令，添加并退出处理
67                  instructions.add(instruction);
68                  break;
69              } else if (instructionKind.isUnary()) {
70                  // 如果是单操作数指令，直接添加
71                  instructions.add(instruction);
72              } else if (instructionKind.isBinary()) {
73                  processBinaryInstruction(instruction);
74              }
75          }
76      }
```

2. 二元操作指令处理

```java
// 处理 Binary 类型的指令
private void processBinaryInstruction(Instruction instruction) {
    IRValue lhs = instruction.getLHS();
    IRValue rhs = instruction.getRHS();
    IRVariable result = instruction.getResult();
    InstructionKind kind = instruction.getKind();

    if (lhs.isImmediate() && rhs.isImmediate()) {
        // 两个操作数都是立即数
        int immediateResult = calculateImmediateResult(kind, (IRImmediate) lhs, (IRImmediate) rhs);
        instructions.add(Instruction.createMov(result, IRImmediate.of(immediateResult)));
    } else if (lhs.isImmediate() && rhs.isIRVariable()) {
        // 左立即数和右变量
        handleLeftImmediate(kind, lhs, rhs, result);
    } else if (lhs.isIRVariable() && rhs.isImmediate()) {
        // 左变量和右立即数
        handleRightImmediate(kind, lhs, rhs, result);
    } else {
        // 两个操作数均为变量
        instructions.add(instruction);
    }
}
```

对于二元操作指令，按左右操作数的类型分为四种情况处理：

•两个立即数：调用 calculateImmediateResult 方法计算结果，并将其转为 MOV 指令。

```java
// 计算两个立即数的结果
private int calculateImmediateResult(InstructionKind kind, IRImmediate lhs, IRImmediate rhs) {
    return switch (kind) {
        case ADD -> lhs.getValue() + rhs.getValue();
        case SUB -> lhs.getValue() - rhs.getValue();
        case MUL -> lhs.getValue() * rhs.getValue();
        default -> throw new IllegalArgumentException("Unsupported binary operation: " + kind);
    };
}
```

- 左操作数为立即数，右操作数为变量：调用 handleLeftImmediate 方法处理。

```java
// 处理左操作数为立即数的指令
private void handleLeftImmediate(InstructionKind kind, IRValue lhs, IRValue rhs, IRVariable result) {
    switch (kind) {
        case ADD -> instructions.add(Instruction.createAdd(result, rhs, lhs));
        case SUB, MUL -> {
            IRVariable temp = IRVariable.temp();
            instructions.add(Instruction.createMov(temp, lhs));
            if (kind == InstructionKind.SUB) {
                instructions.add(Instruction.createSub(result, temp, rhs));
            } else {
                instructions.add(Instruction.createMul(result, temp, rhs));
            }
        }
        default -> throw new IllegalArgumentException("Unsupported binary operation: " + kind);
    }
}
```

- 左操作数为变量，右操作数为立即数：调用 handleRightImmediate 方法处理。

```java
// 处理右操作数为立即数的指令
private void handleRightImmediate(InstructionKind kind, IRValue lhs, IRValue rhs, IRVariable result) {
    switch (kind) {
        case ADD, SUB -> instructions.add(Instruction.createAdd(result, lhs, rhs));
        case MUL -> {
            IRVariable temp = IRVariable.temp();
            instructions.add(Instruction.createMov(temp, rhs));
            instructions.add(Instruction.createMul(result, lhs, temp));
        }
        default -> throw new IllegalArgumentException("Unsupported binary operation: " + kind);
    }
}
```

- 两个操作数均为变量：直接加入队列。

3. 寄存器分配

VariableToRegister：

为变量分配寄存器：

- 优先检查是否已分配寄存器。
- 如果有空闲寄存器，直接分配。
- 若无空闲寄存器，调用 findReusableRegister 回收不再使用的寄存器。
- 如果无法分配，抛出异常。

```
145      public void VariableToRegister(IRValue operand, int currentIndex) {
146          // 立即数无需分配寄存器
147          if (operand.isImmediate()) {
148              return;
149          }
150
151          // 如果变量已经有寄存器分配，直接返回
152          if (registerMap.containsKey(operand)) {
153              return;
154          }
155
156          // 尝试分配空闲寄存器
157          for (int i = Register.values().length - 1; i >= 0; i--) {
158              Register register = Register.values()[i];
159              if (!registerMap.containsValue(register)) {
160                  registerMap.put(operand, register); // 分配空闲寄存器
161                  return;
162              }
163          }
164
165          // 无空闲寄存器，寻找不再使用的变量的寄存器
166          Register reusedRegister = findReusableRegister(currentIndex);
167          if (reusedRegister != null) {
168              registerMap.replace(operand, reusedRegister);
169          } else {
170              // 无可用寄存器时，抛出异常
171              throw new RuntimeException("No enough registers!");
172          }
173      }
```

findReusableRegister：

- 活跃变量分析：遍历后续指令，移除占用寄存器的活跃变量。
- 返回第一个空闲寄存器，若无空闲寄存器返回 null。

```
175      private Register findReusableRegister(int currentIndex) {
176          Set<Register> unusedRegisters = new HashSet<>(Arrays.asList(Register.values()));
177
178          // 遍历剩余指令，移除活跃变量的寄存器
179          for (int i = currentIndex; i < instructions.size(); i++) {
180              Instruction instruction = instructions.get(i);
181              for (IRValue operand : instruction.getAllOperands()) {
182                  Register register = registerMap.getByKey(operand);
183                  unusedRegisters.remove(register);
184              }
185          }
186
187          // 如果有未使用的寄存器，返回其中一个
188          return unusedRegisters.isEmpty() ? null : unusedRegisters.iterator().next();
189      }
```

4. 汇编指令生成

run 方法是 AssemblyGenerator 类的核心，它负责将预处理后的中间代码 (instructions) 转换为 RISC-V 汇编代码。其辅助方法（generateBinaryOperation、generateMovOperation 和 generateReturnOperation）完成具体的指令转换任务。

① run 方法：遍历所有中间代码指令，逐一将其翻译为对应的汇编代码。遇到 RET 指令时终止代码生成。

• 初始化 currentIndex 为 0，表示当前处理的指令位置，用于寄存器分配中的活跃变量分析。

• 遍历 instructions：

根据指令类型调用不同的辅助方法生成汇编代码：

二元操作指令（ADD, SUB, MUL）：调用 generateBinaryOperation。

单操作数指令（MOV）：调用 generateMovOperation。

返回指令（RET）：调用 generateReturnOperation。

生成的汇编代码存入 asmInstructions，同时附加注释。

• 遇到 RET 指令直接返回，避免处理无效代码。

```
201     public void run() {
202         int currentIndex = 0;
203 
204         for (Instruction instruction : instructions) {
205             InstructionKind kind = instruction.getKind();
206             String asmCode = null;
207 
208             switch (kind) {
209                 case ADD, SUB, MUL -> asmCode = generateBinaryOperation(instruction, kind, currentIndex);
210                 case MOV -> asmCode = generateMovOperation(instruction, currentIndex);
211                 case RET -> {
212                     asmCode = generateReturnOperation(instruction);
213                     asmInstructions.add(asmCode); // 添加 RET 指令的汇编代码
214                     return; // 直接退出，避免多余处理
215                 }
216                 default -> throw new IllegalArgumentException("Unsupported instruction kind: " + kind);
217             }
218 
219             if (asmCode != null) {
220                 asmCode += String.format("\t\t# %s", instruction); // 添加注释
221                 asmInstructions.add(asmCode);
222             }
223 
224             currentIndex++;
225         }
226     }
```

② generateBinaryOperation 方法：根据二元操作指令（ADD, SUB, MUL）生成 RISC-V 汇编代码。

```
229     // 生成二元操作的汇编代码
230     private String generateBinaryOperation(Instruction instruction, InstructionKind kind, int currentIndex) {
231         IRValue lhs = instruction.getLHS();
232         IRValue rhs = instruction.getRHS();
233         IRVariable result = instruction.getResult();
234 
235         // 为操作数和结果分配寄存器
236         VariableToRegister(lhs, currentIndex);
237         VariableToRegister(rhs, currentIndex);
238         VariableToRegister(result, currentIndex);
239 
240         Register lhsReg = registerMap.getByKey(lhs);
241         Register rhsReg = rhs.isImmediate() ? null : registerMap.getByKey(rhs);
242         Register resultReg = registerMap.getByKey(result);
243 
244         // 根据操作类型生成汇编代码
245         return switch (kind) {
246             case ADD -> rhs.isImmediate()
247                     ? String.format("\taddi %s, %s, %s", resultReg, lhsReg, rhs)
248                     : String.format("\tadd %s, %s, %s", resultReg, lhsReg, rhsReg);
249             case SUB -> rhs.isImmediate()
250                     ? String.format("\tsubi %s, %s, %s", resultReg, lhsReg, rhs)
251                     : String.format("\tsub %s, %s, %s", resultReg, lhsReg, rhsReg);
252             case MUL -> String.format("\tmul %s, %s, %s", resultReg, lhsReg, rhsReg);
253             default -> throw new IllegalArgumentException("Unsupported binary operation: " + kind);
254         };
255     }
```

③ generateMovOperation 方法：将单操作数 MOV 指令（数据传送）转换为 RISC-V 汇编代码。

```
257         // 生成MOV指令的汇编代码
258         private String generateMovOperation(Instruction instruction, int currentIndex) {
259             IRValue source = instruction.getFrom();
260             IRVariable result = instruction.getResult();
261 
262             // 为操作数和结果分配寄存器
263             VariableToRegister(source, currentIndex);
264             VariableToRegister(result, currentIndex);
265 
266             Register sourceReg = source.isImmediate() ? null : registerMap.getByKey(source);
267             Register resultReg = registerMap.getByKey(result);
268 
269             // 根据是否为立即数生成汇编代码
270             return source.isImmediate()
271                     ? String.format("\tli %s, %s", resultReg, source)
272                     : String.format("\tmv %s, %s", resultReg, sourceReg);
273         }
```

④ generateReturnOperation 方法：将返回指令（RET）转换为 RISC-V 汇编代码。

```
275         // 生成 RET 指令的汇编代码
276         private String generateReturnOperation(Instruction instruction) {
277             IRValue returnValue = instruction.getReturnValue();
278 
279             // 分配寄存器
280             VariableToRegister(returnValue, instructions.size() - 1);
281             Register returnValueReg = registerMap.getByKey(returnValue);
282 
283             // 生成RET指令并附加注释
284             return String.format("\tmv a0, %s\t\t# %s", returnValueReg, instruction);
285         }
```

② template/src/cn/edu/hitsz/compiler/asm/BMap.java

在编译器的寄存器分配过程中，双向映射的需求来源于变量与寄存器之间的双向关联关系。在寄存器分配时，变量（如中间代码中的 `IRVariable`）需要映射到具体的寄存器（如 RISC-V 的 `t0`,`t1` 等），因此需要能够快速通过变量找到对应的寄存器。同时，为了释放寄存器或检查寄存器的使用状态，也需要从寄存器反向找到其绑定的变量。这种双向查询需求在编译器中非常常见。传统的单向映射（如使用一个 `Map<K, V>`）虽然能够通过键快速找到值，但反向查找（即从值找到键）需要遍历整个映射，时间复杂度为 O(n)，在频繁查询的场景中效率较低。因此，为了优化查询效率，使用了双向映射的设计，通过两个 `Map` 分别维护正向（变量到寄存器）和反向（寄存器到变量）的映射，使得从任意一方查找另一方的时间复杂度都为 O(1)，极大提升了寄存器分配的性能和灵活性。

具体的设计如下：

```java
public class BMap<K, V> {
    private final Map<K, V> KVmap = new HashMap<>();
    private final Map<V, K> VKmap = new HashMap<>();

    public void put(K key, V value) {
        // 复用 replace 方法来添加映射
        replace(key, value);
    }

    public void removeByKey(K key) {
        if (KVmap.containsKey(key)) {
            V value = KVmap.remove(key);
            VKmap.remove(value);
        }
    }

    public void removeByValue(V value) {
        if (VKmap.containsKey(value)) {
            K key = VKmap.remove(value);
            KVmap.remove(key);
        }
    }

    public boolean containsKey(K key) {
        return KVmap.containsKey(key);
    }

    public boolean containsValue(V value) {
        return VKmap.containsKey(value);
    }


    public void replace(K key, V value) {
        // 如果 key 或 value 已存在，则移除旧的映射
        if (containsKey(key)) {
            removeByKey(key);
        }
        if (containsValue(value)) {
            removeByValue(value);
        }
        // 添加新的映射
        KVmap.put(key, value);
        VKmap.put(value, key);
    }

    public V getByKey(K key) {
        return KVmap.get(key);
    }

    public K getByValue(V value) {
        return VKmap.get(value);
    }

    public void clear() {
        KVmap.clear();
        VKmap.clear();
    }

    public int size() {
        return KVmap.size(); // KVmap 和 VKmap 的大小始终一致
    }

    public boolean isEmpty() {
        return KVmap.isEmpty(); // KVmap 和 VKmap 的状态始终一致
    }
}
```

4 实验结果与分析

对实验的输入输出结果进行展示与分析。注意：要求给出编译器各阶段（词法分析、语法分析、中间代码生成、目标代码生成）的输入输出并进行分析说明。

Lab1. 词法分析

输入 1：coding_map.csv

规定所有此法单元的类别及编码

```
1 int
2 return
3 =
4 ,
5 Semicolon
6 +
7 -
8 *
9 /
10 (
11 )
51 id
52 IntConst
```

输入 2：input_code.txt

待编译的代码

```
int result;
int a;
int b;
int c;
a = 8;
b = 5;
c = 3 - a;
result = a * b - ( 3 + b ) * ( c - a );
return result;
```

输出 1：old_symbol_table.txt

实验一只进行词法分析，未进行语义分析，故 type 属性均为 null

```
(a, null)
(b, null)
(c, null)
(result, null)
```

输出 2：token.txt

通过词法分析得到的 token 序列，对比 std 可见得到了正确的输出

```
(int,)
(id,result)
(Semicolon,)
(int,)
(id,a)
(Semicolon,)
(int,)
(id,b)
(Semicolon,)
(int,)
(id,c)
(Semicolon,)
(id,a)
(=,)
(IntConst,8)
(Semicolon,)
(id,b)
(=,)
(IntConst,5)
(Semicolon,)
(id,c)
(=,)
(IntConst,3)
(-,)
(id,a)
(Semicolon,)
(id,result)
(=,)
(id,a)
(*,)
(id,b)
(-,)
((,)
(IntConst,3)
(+,)
(id,b)
(),)
(*,)
((,)
(id,c)
(-,)
(id,a)
(),)
(Semicolon,)
(return,)
(id,result)
(Semicolon,)
($,)
```

Lab2. 语法分析

输入 1：grammar.txt

描述一个简单的编程语言的语法结构，为编译器的语法分析（如构造语法树或进行语法检查）提供了基础

```
P -> S_list;
S_list -> S Semicolon S_list;
S_list -> S Semicolon;
S -> D id;
D -> int;
S -> id = E;
S -> return E;
E -> E + A;
E -> E - A;
E -> A;
A -> A * B;
A -> B;
B -> ( E );
B -> id;
B -> IntConst;
```

输入 2：LR1_table.csv

LR1 分析表，辅助进行自底向上的语法分析

状态	ACTION												GOTO					
	id	(	)	+	-	*	=	int	return	IntConst	Semicolon	$	E	S_list	S	A	B	D
0	shift 4							shift 5	shift 6					1	2			3
1												accept						
2											shift 7							
3	shift 8																	
4							shift 9											
5	reduce D -> int																	
6	shift 13	shift 14								shift 15			10			11	12	
7	shift 4							shift 5	shift 6			reduce S_list -> S Semicolon		16	2			3

输出 1：parser_list.txt

按照分析表规约后得到的产生式列表，对比 std 可见得到了正确的输出

```
D -> int
S -> D id
D -> int
S -> D id
D -> int
S -> D id
D -> int
S -> D id
B -> IntConst
A -> B
E -> A
S -> id = E
B -> IntConst
A -> B
E -> A
S -> id = E
B -> IntConst
A -> B
E -> A
B -> id
A -> B
E -> E - A
S -> id = E
B -> id
A -> B
B -> id
A -> A * B
E -> A
B -> IntConst
A -> B
A -> B
E -> A
B -> id
A -> B
E -> A
B -> ( E )
A -> B
B -> id
A -> B
E -> A
B -> id
A -> B
E -> E - A
B -> ( E )
A -> A * B
E -> E - A
S -> id = E
B -> id
A -> B
E -> A
S -> return E
S_list -> S Semicolon
S_list -> S Semicolon S_list
S_list -> S Semicolon S_list
S_list -> S Semicolon S_list
S_list -> S Semicolon S_list
S_list -> S Semicolon S_list
S_list -> S Semicolon S_list
S_list -> S Semicolon S_list
S_list -> S Semicolon S_list
P -> S_list
```

Lab3. 中间代码生成

（利用前两个实验的输出进行中间代码生成，无额外输入）

输出 1：new_symbol_table.txt

语义分析后的符号表，更新了之前 old_symbol_table.txt 中缺失的 type 属性信息，与 std 对比或者直接脑测得知正确

```
(a, Int)
(b, Int)
(c, Int)
(result, Int)
```

输出 2：intermediate_code.txt

中间表示的三地址码，对比 std 可知正确

```
(MOV, a, 8)
(MOV, b, 5)
(SUB, $0, 3, a)
(MOV, c, $0)
(MUL, $1, a, b)
(ADD, $2, 3, b)
(SUB, $3, c, a)
(MUL, $4, $2, $3)
(SUB, $5, $1, $4)
(MOV, result, $5)
(RET, , result)
```

输出 3：ir_emulate_result.txt

中间表示的模拟执行结果 144，对比 std 可知正确。同时与源代码理论的返回值相对应

```
1       144
```

Lab4. 目标代码生成

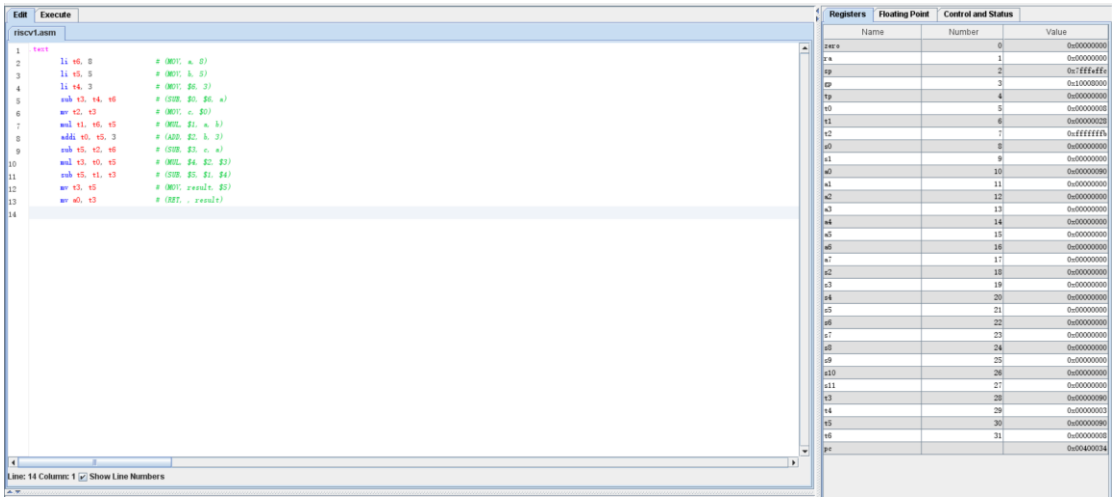
（主要利用 Lab3 得到的中间代码，无额外输入）

输出 1：assembly_language.asm

生成的汇编代码

```
1       .text
2               li t6, 8                # (MOV, a, 8)
3               li t5, 5                # (MOV, b, 5)
4               li t4, 3                # (MOV, $6, 3)
5               sub t3, t4, t6              # (SUB, $0, $6, a)
6               mv t2, t3               # (MOV, c, $0)
7               mul t1, t6, t5              # (MUL, $1, a, b)
8               addi t0, t5, 3                  # (ADD, $2, b, 3)
9               sub t5, t2, t6              # (SUB, $3, c, a)
10              mul t3, t0, t5              # (MUL, $4, $2, $3)
11              sub t5, t1, t3              # (SUB, $5, $1, $4)
12              mv t3, t5               # (MOV, result, $5)
13              mv a0, t3               # (RET, , result)
```

汇编代码在 RARS 上的运行结果



注意到用于储存返回值的寄存器 a0 的最后结果为 0x90（114），可见我们生成的汇编代码有正确实现功能

5 实验中遇到的困难与解决办法

在本次实验中，我遇到的主要困难在于前期汇编理论课的进度尚未完全覆盖实验的需求，导致对于实验的整体框架不够清晰，尤其是各个模块的功能和它们之间的关系比较模糊。同时，实验的框架代码量较大，其中许多模块的实现已被提供，这对我们在自行设计代码时提

出了较高的要求。如果不能对框架代码有较深的理解，便容易在设计中与框架代码产生冲突，或者遗漏一些功能。

针对这些困难，我采取了一些解决办法。首先，通过多次阅读实验指导书，加深对框架的整体理解。同时，我尝试在框架代码和自己编写的代码中加入 `logger` 记录和异常值信息的报错功能，以便更好地定位问题。其次，与同学交流，向他们请教疑惑，并参考他们的建议和思路，这些都为我的实验进展提供了很大的帮助。

通过本次实验，我不仅加深了对 Java 编程语言的理解，也巩固了相关的理论知识，更加清晰地掌握了编译器的工作流程。此外，通过亲手实现并调试代码，我进一步加深了对自底向上分析方法的理解。

一些建议可以为今后的实验优化提供参考。希望实验框架代码能够在实验课上进行统一讲解，帮助大家更好地理解整体框架的设计。同时，建议理论课和实验课能够适当错开时间，使同学们在对编译器工作步骤有了整体认识后再进行具体的迭代开发任务，这样可以提升实验效率。

最后，感谢编译原理老师们的辛勤付出！