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: 吴梓滔

学　号: 220110430

专　业: 计算机科学与技术

日　期: 2024-11-20

1 实验目的与方法

实验目的：使用 Java 语言实现简易的类 C 语言的编译器，目标平台为支持 RV32M 指令集的 RISC-V 32 机器。

实验方法：使用的 Java 版本为 JDK 22，使用 rars 工具模拟 RISC-V 机器，以及编译工作台用于语法分析。

1.1 词法分析器

实验目的：

1) 加深对词法分析程序的功能及实现方法的理解；
2) 对类 C 语言的文法描述有更深的认识，理解有穷自动机、编码表和符号表在编译的整个过程中的应用；
3) 设计并编程实现一个词法分析程序，对类 C 语言源程序段进行词法分析，加深对高级语言的认识。

1.2 语法分析

实验目的：

1) 深入了解语法分析程序实现原理及方法。
2) 理解 LR(1)分析法是严格的从左向右扫描和自底向上的语法分析方法。

实验方法：

1) 使用编译工作台生成 LR(1)分析表并导入编译器。
2) 编写语法分析驱动过程。

1.3 典型语句的语义分析及中间代码生成

实验目的：

1) 加深对自底向上语法制导翻译技术的理解，掌握声明语句、赋值语句和算术运算语句的翻译方法。
2) 巩固语义分析的基本功能和原理的认识，理解中间代码生产的作用。

实验方法：

编写程序进行语义分析和中间代码生成，使用模拟 IR 计算验证结果正确性。

1.4 目标代码生成

实验目的：

1) 加深编译器总体结构的理解与掌握；
2) 掌握常见的 RISC-V 指令的使用方法；
3) 理解并掌握目标代码生成算法和寄存器选择算法。

2 实验内容及要求

每次实验室的实验内容和要求描述清楚。

2.1 词法分析器

使用 DFA 实现词法分析程序，读入源语言程序并转化为 Token 序列输出。在过程中自动维护符号表。

2.2 语法分析

1. 实验内容： 利用 LR(1)分析法，设计语法分析程序，对输入单词符号串进行语法分析；
2. 实验要求：
   1）输出推导过程中所用产生式序列并保存在输出文件中；
   2）较低完成要求：实验模板代码中支持变量申明、变量赋值、基本算术运算的文法；
   3）较优完成要求：自行设计文法并完成实验。
   4）要求：实验一的输出作为实验二的输入。

2.3 典型语句的语义分析及中间代码生成

实验内容:采用实验二中的文法,为产生式设计翻译方案,对所给程序段进行语法制导翻译,输出深 层的中间代码序列和更新后的符号表，并保存在相应文件中。

实验要求 ：

1) 利用该翻译方案,对所给程序段进行分析,输出生成的中间代码序列和更新后的符号表，并保存 在相应文件中。
2) 实现声明语句、简单赋值语句、算术表达式的语义分析与中间代码生成。
3) 使用框架中的模拟器验证生成的中间代码的正确性。

2.4 目标代码生成

1. 实验内容：
   1) 将实验三生成的中间代码转换为目标代码（汇编指令）；
   2) 运行生成的目标代码，验证结果的正确性。
2. 实验要求： 使用 RARS 运行由编译程序生成的目标代码，验证结果的正确性。

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

词法分析器的输出是单词序列，在 5 种单词种类中，关键字、运算符、分界符都是程序设计语言预先定义的，其数量是固定的。而标识符、常数则是由程序设计人员根据具体的需要按照程序设计语言的规定自行定义的，其数量可以是无穷多个。编译程序为了处理方便，通常需要按照一定的方式对单词进行分类和编码，在此基础上，将单词表示成二元组的形式（类别编码，单词值）。

在本实验处理的类 C 语言中，编码表在实验项目中的 coding_map.csv 文件。

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

其中，标识符 id 完全由字母组成，整数 IntConst 完全由数字组成。类别编码 1-11 的为程序预先定义的关键字、运算符。

3.1.2 正则文法

本实验处理的类 C 语言文法如下：

G=(V,T,P,S)，其中 V={S,A,B,Cdigit,no_0_digit,,char},T={任意符号}，P 定义如下:

约定： 用 digit 表示数字：0,1,2,…,9;

no_0_digit 表示数字：1,2,…,9;

用 letter 表示字母：A,B,…,Z,a,b,…,z,_

标识符：S→letter A A→letter A|digit A|ε

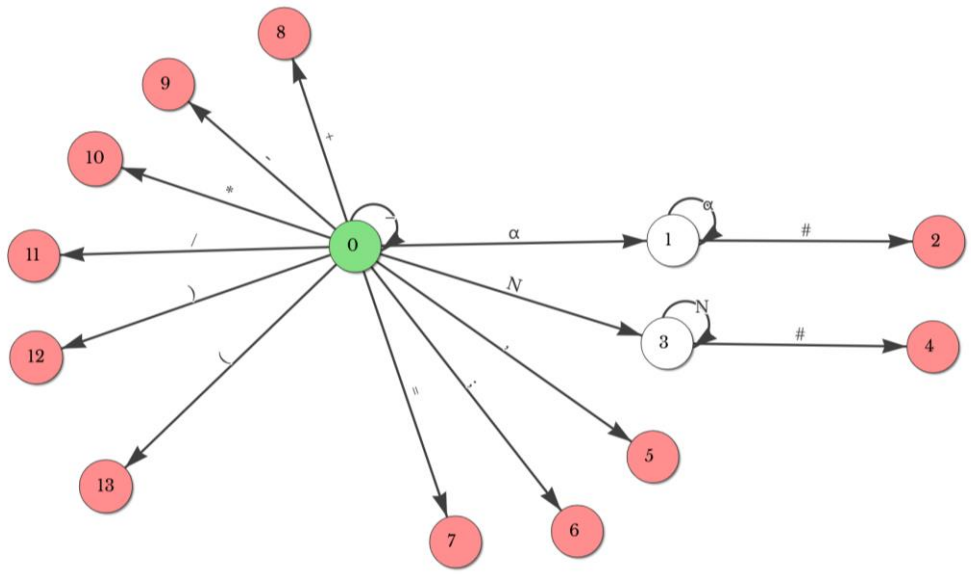
运算符、分隔符：S->B B->=|*|+|-|/|(|);

整常数：S ->no_0_digitB B -> digitB | ε

字符串常量：S -> “C”

字符常量：S->’D’

3.1.3 状态转换图



注：

1) 0->0 的输入符号“_”表示空白字符“ ”，
2) 0->1 和 1->1 的输入符号“α”表示任何字母，
3) 1->2 的输入符号“#”表示任何非字母的字符，
4) 0->3 和 3->3 的输入符号“N”表示任何数字，
5) 3->4 的输入符号“#”表示任何非数字的字符，
6) 到达接收状态后，要构造相应的 Token，并返回状态 0。

3.1.4 词法分析程序设计思路和算法描述

词法分析程序（LexicalAnalyzer.java）的核心算法是根据已经设计好的状态转换图，通过维护一个状态机，每次读入一个缓冲区输入字符，自动机做出相应的处理。在实际设计中根据实际，对自动机状态做了一个简化。

（1）run 核心算法

状态机设计：自动机设置 3 个主要状态，START，INTCONST 和 ID。另有一个 ERROR 状态用于集中处理出错情况。语言中，‘,’‘;’‘+’‘-’‘=’等符号是长度一定为 1，读到即可直接确定一个 token，并且可以用 Token.Symple 方法创建 token。对于这些情况，不再单独设置状态，而是统一成为 START 状态下的一个转移，START 状态下读到 1 个字符能够确定的 token 时，直接创建这个 token，并仍然转移到 START 状态。

START 状态表示等待开始读入一个新字符。任何 token 的第一个字符，都是在 START 状态下读入。

INTCONST 状态用于读进一个常数，当识别到第一个数字类型，进入 INTCONST 状态，此后保持读入数字，直到输入不再是数字，说明完成了一整个数字的读入，将创建 token，并且回到 START 状态。

ID 状态用于读入一个标识符。由于该语言标识符一定完全由字母组成，所以读入第一个字母后，转移到 ID 状态。此后保持读入字母，直到第一个不是字母的输入，说明完成了一整个标识符的读入，将创建 token，并回到 START 状态。

词法分析程序的核心算法主要是状态机的维护，以及字符类型的判断。实现了一个辅助方法，用于判断一个字符输入是否是单个字符的 token。

```
Params: ch – run()过程中每次读取的一个字符 判断ch是否能够用 Token.simple() 方法构建Token
Returns: bool值
// 判断是否为单字符标点
1 usage   changdick
private boolean isSinglePunctuation(char ch) {
    return ch == ',' || ch == ';' || ch == '=' || ch == '+' || ch == '-' || ch == '*' || ch == '/' || ch == '(' || ch == ')';
}
```

在 run 方法中，初始设置状态 currentState 为 START，循环并每次读取输入串的第 i 个字符为 ch。通过 switch 语句，根据 currentState 的值做不同的动作。

```
while (i < sourceCode.length()) {
    char ch = word[i];

    // 状态转移
    switch (currentState) {
        case START -> {
            if (Character.isWhitespace(ch)) {
                //  若ch为空格，直接读下一个字符，保持为START状态
                i++;
            } else if (isSinglePunctuation(ch)) {
                  currentState = State.SINGLE_PUNCTUATION;
                // 如果是可以用simple方法直接构造的Token，直接把他构造了，读下一个字符，并
                // 能够用Token.simple()方法的情况，但是要判断';'传入'semicolon'
                if (ch == ';') {
                    tokens.add(Token.simple( tokenKindId: "Semicolon"));
                } else {
                    tokens.add(Token.simple(String.valueOf(ch)));
                }
                i++;

            } else if (Character.isLetter(ch)) {
                currentState = State.ID;
            } else if (Character.isDigit(ch)) {
                currentState = State.INTCONST;
            } else {
                currentState = State.ERROR;
            }
        }
```

如上图所示，在 START 状态，依次跳过空格，检查是否为单字符 token，如果是，将调用 simple 方法构造 token，并通过 i++读过这一个字符。若不是，继续检查是否为字母，或是数字，并跳转到响应的状态。

在读标识符或整型常数对应的状态内，核心操作是通过字符串的 substring(m, n)方法，截取字符串的[m:n-1]范围的字符。由于标志符一定完全由字母组成，整型常数一定完全由数字组成，都可以用简单的类型判断，来寻找 token 结束的位置。因此保存 token 的开始下标为 index，让 i 连续下移直到该 token 的结束位置停止，即可用 substring(index,i)来获取 token。如下图所示。

```
// 能够用Token.simple()方法的情况，但是要判断';'传入'semicolon'
case INTCONST -> {
    int index = i;
    for (; i < sourceCode.length() && Character.isLetter(word[i]); i++) {}
    String digit = sourceCode.substring(index, i);
    tokens.add(Token.normal( tokenKindId: "IntConst", digit));    // 将(index, i)的内容即数字字段读下创建TOken
    currentState = State.START;                                   // 回到start状态
}
case ID -> {
    int index = i;
    for (; i < sourceCode.length() && Character.isLetter(word[i]); i++) {}

    String key = sourceCode.substring(index, i);     // 将[a-zA-Z]+ 的字段(index,i)存下来
```

在实际构建标识符的 token 时，还会涉及查字母表的步骤。这一部分在符号表的维护中叙述。

（2）符号表、token 列表等数据结构的维护

符号表 symbolTable 用于储存符号。对于第一次遇到的用户符号，会存入符号表中，随后再次创建标识符时，可以通过查符号表，快速构建 token。

```
    // 如果已经在符号表里面的，如int和return 就可以直接用simple方法构建
    if (TokenKind.isAllowed(key)) {
        tokens.add(Token.simple(key));
    } else {
        tokens.add(Token.normal( tokenKindId: "id", key));
        if (!symbolTable.has(key)) {
            symbolTable.add(key);
        }
    }
    currentState = State.START;
}
```

创建的 Token 用一个 List<Token>列表存储，在词法分析程序执行过程中，每创建一个 token，通过 add 方法加入到列表中，最终直接返回整个列表对象。

词法分析的输入源码是从文件读入的。在词法分析程序内，调用 java 的文件 io 类，读取指定路径的源文件，并转换为字符串，存储到一个字符串对象中，在核心算法中，就是依次读入这个字符串的每个字符。

3.2 语法分析

3.2.1 拓展文法

我们需要借助编译工作台生成 LR（1）分析表，要求输入的文法只有一个起始状态和终结状态。对现有文法进行拓展，得到拓展文法 grammar.txt。

```
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

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
8											reduce S -> D id							
9	shift 13	shift 14								shift 15			17			11	12	
10				shift 18	shift 19						reduce S -> return E							
11				reduce E -> A	reduce E -> A	shift 20					reduce E -> A							
12				reduce A -> B	reduce A -> B	reduce A -> B					reduce A -> B							
13				reduce B -> id	reduce B -> id	reduce B -> id					reduce B -> id							
14	shift 24	shift 25								shift 26			21			22	23	
15				reduce B -> IntConst	reduce B -> IntConst	reduce B -> IntConst					reduce B -> IntConst							
16												reduce S_list -> S Semicolon S_list						
17				shift 18	shift 19						reduce S -> id = E							
18	shift 13	shift 14								shift 15						27	12	
19	shift 13	shift 14								shift 15						28	12	
20	shift 13	shift 14								shift 15							29	
21			shift 30	shift 31	shift 32													
22			reduce E -> A	reduce E -> A	reduce E -> A	shift 33												
23			reduce A -> B	reduce A -> B	reduce A -> B	reduce A -> B												
24			reduce B -> id	reduce B -> id	reduce B -> id	reduce B -> id												
25	shift 24	shift 25								shift 26			34			22	23	
26			reduce B -> IntConst	reduce B -> IntConst	reduce B -> IntConst	reduce B -> IntConst												
27				reduce E -> E + A	reduce E -> E + A	shift 20					reduce E -> E + A							
28				reduce E -> E - A	reduce E -> E - A	shift 20					reduce E -> E - A							
29				reduce A -> A * B	reduce A -> A * B	reduce A -> A * B					reduce A -> A * B							
30				reduce B -> (E)	reduce B -> (E)	reduce B -> (E)					reduce B -> (E)							
31	shift 24	shift 25								shift 26						35	23	
32	shift 24	shift 25								shift 26						36	23	
33	shift 24	shift 25								shift 26							37	
34			shift 38	shift 31	shift 32													
35			reduce E -> E + A	reduce E -> E + A	reduce E -> E + A	shift 33												
36			reduce E -> E - A	reduce E -> E - A	reduce E -> E - A	shift 33												
37			reduce A -> A * B	reduce A -> A * B	reduce A -> A * B	reduce A -> A * B												
38			reduce B -> (E)	reduce B -> (E)	reduce B -> (E)	reduce B -> (E)												

通过编译工作台自动生成的 LR1 分析表如图所示。

3.2.3 状态栈和符号栈的数据结构和设计思路

符号栈可能存储的是终结符，可能存储非终结符。有可能是来自于 token 串的终结符，也可能是原符号栈中元素规约得到的非终结符。因此创建一个类 VTSymbol 来作为符号栈的数据类。VTSymbol 设置数据字段 Token 或 NonTerminal，并且只能有其中一个，另一个为 Null，来实现对终结符和非终结符的简单封装。当符号栈要执行入栈操作时，用 Token 或者 NonTerminal 创建 VTSymbol 对象，再压入栈中。

状态栈和符号栈的定义：使用双端队列结构。

```
private final Deque<VTSymbol> SymbolStack = new ArrayDeque<>();
private final Deque<Status> statusStack = new ArrayDeque<>();
```

3.2.4 LR 驱动程序设计思路和算法描述

模板代码已经实现了 LR1 分析表对象 lrTable，以及产生式对应的类及方法。通过 lrTable 的一些方法，可以很方便地完成查表获取每一步的动作。LR 驱动程序设计主要就是维护符

号栈、状态栈、输入缓冲区，根据当前栈顶元素，调用 lrTable 提供的方法进行查表，得到新的操作，并更新栈。重复这个过程直到读完输入。

输入缓冲区使用一个双端队列实现，实际上作为队列来使用。定义为 private final Deque<Token> tokenQueue = new ArrayDeque<>(); 在 loadTokens 方法，将 token 串拷贝到 tokenQueue 队列中，就完成了输入缓冲区的初始化。

驱动程序的核心步骤是通过一个循环，每次根据状态栈顶元素和当前输入缓冲区的第一个元素值查分析表，获取 Action，并根据 Action，做对应操作。如图所示。

```
while (!tokenQueue.isEmpty()) {
    // peek()取头部，不移除  poll移除头部返回  add加尾部
    Status currentStatus = statusStack.peek();  // 查看当前状态
    Action action = lrTable.getAction(statusStack.peek(), tokenQueue.peek());  // 根据状态和当前 token 获取 action

    switch (action.getKind()) {
        case Shift:
            handleShift(action);
            break;
        case Reduce:
            handleReduce(action);
            break;
        case Accept:
            handleAccept(currentStatus);
            return;  // 分析结束
        default:
            throw new IllegalStateException("Unexpected action kind: " + action.getKind());
    }
}
```

通过 lrTable 的 getAction 方法，读入状态和输入，返回对应的 Action。Action 的 getKind() 方法能够查询该动作的类型，从而判断是移进、规约或是接受。进而可以做对应的操作。在具体实现中，将对应的操作提取成一个方法，增加代码的可读性。

处理移进时，需要做的是将输入缓冲区首个元素移出，加入符号栈栈顶；同时将新状态压入状态栈。通过 Action 的 getStatus 方法，可以查到下一个状态。此外还需要调用 callWhenInShift 方法，该方法与翻译有关。

```
1 usage   changdick *
private void handleShift(Action action) {
    // 执行 Shift 动作：状态入栈。token 出队，并压入符号栈

    callWhenInShift(statusStack.peek(), tokenQueue.peek()); // 看上去好像语义分析慢一步，要等符号表更新


    Status nextState = action.getStatus(); // action.getStatus只有当action是shift类型才能调用，action已经存储好了
    statusStack.push(nextState);   // 将新状态压入状态栈
    SymbolStack.push(new VTSymbol(tokenQueue.poll()));   // 读入的是token，因为语法分析只需要token，但是构建符号放
    // 调用poll方法出队，在返回的同时，将token出队
```

处理规约操作时，获取产生式，执行规约。规约后应该将状态栈出栈，再用栈顶查 goto 表，把新状态压入栈中。把符号栈中右部的也出栈，把左部压入符号栈。产生式可以用 action 类的 getProduction 方法获取，production 提供了 body 字段，可以查到产生式右部的符号数，用于判断弹栈操作的执行次数。状态栈的新元素，是用弹栈后的新栈顶，以及这一步规约得到的符号查 goto 表得到的。

```
1 usage   changdick *
private void handleReduce(Action action) {
    // 执行 Reduce 动作：获取产生式，执行规约。规约后应该将状态栈出栈，再用栈顶查goto表，把新状态压入栈中。 把符号栈中右部的也出栈，

    Production production = action.getProduction();  // Action对象已经存储好了产生式，调用get方法获取
    callWhenInReduce(statusStack.peek(), production);

    //  注意Production是记录 record， 访问记录的字段是这么访问的
    /*访问字段的方式确实看起来像调用方法。
    这是因为记录类型会自动生成访问器方法（getter），用于访问记录中的各个字段。当你使用recordObject.fieldName()的形式时，实际上是在
    for (int i = 0; i < production.body().size(); i++) {
        statusStack.pop();  // 弹出与产生式右部相应数量的状态
        SymbolStack.pop();  // 弹出符号
    }
    // 将产生式左部符号入栈
    SymbolStack.push(new VTSymbol(production.head()));
    // 根据此时的两个栈顶查goto表，获得新状态入栈
    statusStack.push(lrTable.getGoto(statusStack.peek(), production.head()));
}
```

在接受时，驱动程序将通过 return 返回。

3.3 语义分析和中间代码生成

3.3.1 翻译方案

采用语法制导翻译，在自底向上进行语法分析的同时，进行 S-属性定义的自底向上翻译方案。

具体方案如下：

P → S_list {P.val = S_list.val}

S_list → S Semicolon S_list {S_list.val = S_list1.val}

S_list → S Semicolon; {S_list.val = S.val }

S → D id {p=lookup(id.name); if p != nil then enter(id.name, D.type) else error}

S → return E; {S.value = E.value}

D → int {D.type = int;}

S → id = E {gencode(id.val = E.val);}

E → A {E.val = val;}

A → B {A.val = B.val;}

B → IntConst {B.val = IntConst.lexval;}

E → E1 + A {E.val = newtemp(); gencode(E.val = E1.val + A.val);}

E → E1 – A {E.val = newtemp(); gencode(E.val = E1.val - A.val);}

A → A1 * B{A.val = newtemp();gencode(A.val = A1.val * B.val);}

B → (E){B.val = E.val;}

B → id { p = lookup(id.name); if p != nil then B.val = id.val else error}

3.3.2 语义分析和中间代码生成的数据结构

语义分析使用的符号数据类：VTSymbol 类，作为语法分析和语义分析栈的符号元素。比语法分析时增加了 SourceCodeType 字段，用于存放类型。

```
class VTSymbol {
    3 usages
    Token token;
    1 usage
    NonTerminal variable;

    2 usages
    SourceCodeType type;

    3 usages  changdick
    private VTSymbol(Token token, NonTerminal variable, SourceCodeType type){
        this.token = token;
        this.variable = variable;
        this.type = type;
    }
```

语义分析栈：

```
8 usages
private final Deque<VTSymbol> symbolStack = new ArrayDeque<>();
```

语义分析中的符号栈，使用 Java 双端链表实现。

IR 生成使用的数据结构：

```
32 usages
private final Deque<VTSymbol> symbolStack = new ArrayDeque<>();   //

32 usages
private final Stack<IRValue> irStack = new Stack<>();  // IR生成过程中

6 usages
private final List<Instruction> instructions = new ArrayList<>();
```

symbolStack：符号栈同语义分析栈，主要用于占位。

irStack：用于存储 IR 生成中值的栈。和 symbolStack 同步操作，用 Java 的 Stack 实现。

instructions：储存 IR 生成的每一条指令的列表。

3.3.3 语法分析程序设计思路和算法描述

分为静态语义分析 SemanticAnalyzer 和 IR 生成程序 IRGenerator 两个部分实现，均采用观察者模式，作为语法分析驱动程序中的观察者，在语法分析的过程中通知观察者，使语义分析和 IR 生成程序做出同步的动作。从而只需要分别实现两个程序部分的功能。

(1)语义分析程序

语义分析 SemanticAnalyzer 主要是完成静态类型的传递，以及相应符号表的更新。由于目标语言的设计很简单，只有简单的赋值语法，且类型只有 int 一个类型，语义分析只需要在规约和移进步骤中使符号栈与语法分析栈同步操作，并且在涉及到类型传递的规约发生时，对相应符号的类型、以及符号表的类型进行更新即可。

如果发生移进：只需要将输入符号移进即可。语法分析驱动程序在通知移进时，会将当前输入符号传递到语义分析子程序，只需将该符号移入符号栈即可。

如果发生规约：对于涉及到类型的 4、5 号产生式，要做特殊处理，其余只需与语法分析驱动程序同步地弹出符号栈。首先获取产生式编号判断，若为 5 号产生式 D -> int，只需在弹出 int 符号的同时，压入通用的表示某种类型的符号 D，将其 type 字段设置为“Int”。若为 4 号产生式 S -> D id，再弹栈压栈的同时，需要把符号表中“id”的类型更新为 D 符号所指的类型。D 符号所指的类型可以是多种的，但由于目标语言只设置了 int 类型，实际只

有一种选择。但这样设计，便于语言的扩展，例如语言增加了 float 类型，那么在 int 或者 float 规约成 D 的时候，D 的 type 字段可能是 int 或者 float，他们可以在产生式 4 规约时传递到 id。

```
// 规约时要看是哪一条产生式，4，5号产生式要做类型的传递
int productionIndex = production.index();  // 获取产生式号判断
if (productionIndex == 5) {
    // D -> int;
    symbolStack.pop();   // 弹出 int这个token
    symbolStack.push(new VTSymbol(production.head(), SourceCodeType.Int));
} else if (productionIndex == 4) {
    //S -> D id; 要更新符号表
    VTSymbol id = symbolStack.pop(); // 弹出栈顶的符号id
    VTSymbol D  = symbolStack.pop(); // 弹出栈顶的D
    symbolTable.get(id.getToken().getText()).setType(D.type);   //  从符号表
    symbolStack.push(new VTSymbol(production.head()));
```

对于其他的情况，都不涉及到实质上的操作，只需要同步地弹栈、入栈即可。

(2)IR 生成程序

IR 生成程序的核心是值的传递，和指令的生成两个内容。有些产生式，仅仅是将值传给另一个符号，有些产生式则需要确定一个操作指令。程序的基本框架与语义分析类似，在移进的时候，同步语法分析栈的操作；在规约的时候，在同步栈操作的基础上，要根据不同的产生式做一些专门的工作。但与语义分析不同，语义分析只需要对 4、5 号产生式做特殊处理，而 IR 生成几乎要对每一条产生式做专门的处理。

IR 生成程序中设置了符号栈和 IR 值栈，两个栈同步进行操作。但实际上，这两个栈并不总是所有的栈操作都是有意义的，但是程序选择和语法分析同步操作，有时压入无意义的符号进行占位。

如果发生移进：

```
public void whenShift(Status currentStatus, Token currentToken) {
    // TODO
      throw new NotImplementedException();
    // 移位时，ir生成只需要把token存进来就行
    symbolStack.push(new VTSymbol(currentToken));
    irStack.push( item: null);                          // ir占位 ，  只有
}
```

移进的时候，直接将相应的输入符号存入符号表，并赋予一个 IR 值存入 IR 值栈中占位，但实际上，ir 值并不在此时确定，而是在规约的时候确定。

如果发生规约：当发生规约时，IR 生成的处理逻辑较为复杂，是根据规约产生式的不同进行处理的。

15 号产生式 B -> IntConst：

这一产生式是真正确定一个 IR 立即数值。需要将 IR 值的栈弹出占位元素，并获取常数值压入栈中。

```
case 15 -> {
      B -> IntConst; 值出现 此情况弹出栈顶，将数字值压回 数字值从token获取
    Token token = symbolStack.pop().getToken();
    IRImmediate immediate = IRImmediate.of(Integer.parseInt(token.getText()));
    irStack.pop();
    irStack.push(immediate);  // 把栈顶占位的弹了，压回ir数字

    symbolStack.push(new VTSymbol(production.head()));

}
```

14 号产生式 B -> id：

这一产生式是确定一个 IR 变量值。需要将 IR 值得栈弹出占位元素，并获取标识符名，作为 IR 变量值压入栈中。

```
case 14 -> {
        B -> id;  只需要把id的名字传给B
    Token token = symbolStack.pop().getToken();  //弹出栈顶，得到的是id
    IRVariable variable = IRVariable.named(token.getText());
    irStack.pop();
    irStack.push(variable);  // 结构和产生式15基本一样

    symbolStack.push(new VTSymbol(production.head()));
}
```

13 号产生式 B -> (E)：

这一产生式规约，主要需要将 E 的 IR 值传递到 B。通过弹栈，将(和)这类无意义的符号弹出。对于 10 号、12 号产生式 E -> A 和 A -> B，也是一个值传递到另一个值，用简单的出栈、入栈完成。

```
case 13 -> {
        B -> ( E );  和14、15基本1一样，但是E 不一定是IRImmediate或者IRVaria
    symbolStack.pop();symbolStack.pop();symbolStack.pop();  // 符号栈
    irStack.pop();
    IRValue value = irStack.pop();   // 第二个其实就是E的值，E.value要存
    irStack.pop();
    irStack.push(value);

    symbolStack.push(new VTSymbol(production.head()));

}
```

11 号产生式 A -> A * B：

这一产生式是定义一个乘法运算，这样的操作就需要生成对应指令。通过连续弹栈，获取 A 和 B 的 IR 值，并获取一个临时 IR 变量，创建对应的乘法指令。

```
case 11 -> {
      A -> A * B;
    symbolStack.pop();symbolStack.pop();symbolStack.pop();  //三个符号可以全部
    IRValue bValue = irStack.pop();  // B.value
    irStack.pop();         // 这个是 *
    IRValue aValue = irStack.pop();   // A.value
    IRVariable newValue = IRVariable.temp();
    irStack.push(newValue);  //压入栈中

    instructions.add(Instruction.createMul(newValue, aValue, bValue));  //

    symbolStack.push(new VTSymbol(production.head()));
```

其余 9 号产生式 E -> E – A，8 号产生式 E -> E + A，都是类似的二元运算，其处理方式和乘法类似，只在创建指令时用的方法不一样，可以用类似的方法完成处理。

6 号产生式 S -> id = E：

这一产生式定义一个赋值的语句，对应的指令是一个 MOV 指令。通过弹栈取出两个 IR 值，构造对应的 MOV 指令。同时，两个栈的操作要与语法分析栈保持一致。

```
case 6 -> {
      S -> id = E;
    symbolStack.pop();  // pop E
    symbolStack.pop();  // pop =
    Token token = symbolStack.pop().getToken();
    IRValue eValue = irStack.pop(); //ir栈第一次弹出是E.value
    irStack.pop(); //第二次弹出是=
    irStack.pop(); // 第三次弹出的是id占位用的

    IRVariable variable = IRVariable.named(token.getText());  //通过token获得varia
    instructions.add(Instruction.createMov(variable, eValue));  //构造赋值指令

    irStack.push( item: null);  // S只需要占位置

    symbolStack.push(new VTSymbol(production.head()));
}
```

7 号产生式 S -> return E：

这一产生式定义一个返回语句，对应的时 RET 指令。只需要弹出栈顶 IR 值，创建对应的指令即可。

```
case 7 -> {
      S -> return E;
    symbolStack.pop();symbolStack.pop();  //符号栈无含义，直接弹出2

    IRValue value = irStack.pop();
    irStack.pop();
    irStack.push( item: null);

    instructions.add(Instruction.createRet(value));

    symbolStack.push(new VTSymbol(production.head()));

}
```

对于其他产生式，没有实质上的 IR 值传递或指令的创建，就只需要根据规约的产生式，弹栈、压栈即可。

此外，IR 生成程序不使用符号表。

3.4 目标代码生成

3.4.1 设计思路和算法描述

目标代码生成模块 AssemblyGenerator，需要读入翻译过程生成的中间指令列表，输出对应的 RISC-V 汇编代码，且实验要求使用临时寄存器 t0-t6。关键的算法包括空闲寄存器的管理，寄存器的分配策略，以及汇编语句的生成。

目标代码生成的基本步骤是对中间指令逐句生成对应的汇编代码，每生成一句汇编代码后，自动检查所有变量，释放不再使用的变量的寄存器。

所有的寄存器最终是输出到汇编语句中呈现，因此，寄存器直接采用字符串类型来表示和存储。关键的数据结构是一个列表 freeRegisters 和变量名与寄存器的哈希表 registerMap。freeRegisters 用于存储所有的空闲寄存器。当一个寄存器空闲可用的时候，就应该是该列表中的元素。当需要分配该寄存器取储存变量，则应从该列表中移除；当该寄存器释放时，应

该将其加入到 freeRegister 中。

registerMap 用于存储所有正在使用的寄存器的变量和寄存器。当一个寄存器分配到变量，这个变量和寄存器就应加入该表。

空闲寄存器的管理基于上述数据结构，并且在每句汇编代码生成后有自动的检查和释放寄存器。寄存器的分配策略同样基于上述两个数据结构，在获取中间指令的变量后，会先检查改变量是否已经使用过并分配好寄存器，若是首次出现，将从空闲寄存器中取寄存器分配给该变量；若不是首次出现，将直接返回已经分配好的寄存器。汇编语句的生成中，除了获取操作数，构造汇编语句外，还需考虑立即数位置的交换等问题。

3.4.2 RISC-V 汇编操作数的获取方法实现

在进行汇编语句生成的时候，使用 getOperand 方法，读入一个 IRValue 类的立即数或变量，将直接返回可用于构造汇编语句的 String 型操作数，可以是寄存器的表示如“t0”，也可以是 RISC-V 立即数如“3”。具体使用中，只需要获取中间指令的操作数，作为该方法的参数即可。具体实现由 getOperand 方法完成。如 String src1 = getOperand(instruction.getLHS())。

getOperand 方法读入的是 IRValue，将具体判断该值是立即数还是变量。若为立即数，则直接返回其值的字符串。若为变量，在 RISC-V 中是用寄存器存储表示的，将直接返回寄存器表示，具体还需要判断改变量是首次出现，分配新寄存器，或是已经出现过，可以返回对应寄存器。对于两种情况，不在 getOperand 中考虑，而是调用 getRegister 方法，返回其寄存器。

```
6 usages   changdick
private String getOperand(IRValue value) {
    // 返回操作数的字符串表示
    // 假设有方法返回 IRValue 的字符串表示
    // 如果操作数是立即数，直接返回其值
    if (value.isImmediate()) {
        return value.toString();  // 假设 toString 返回的是立即数的值
    }
    // 有可能是存在寄存器里的，其他情况直接返回存它的寄存器。存它的寄存器可能是

    // 如果是其他类型的操作数，我们分配一个新的寄存器
    return getRegister(value.toString());  // 为该操作数分配一个新的寄存器

}
```

具体的 getRegister 方法读入一个变量名，返回一个寄存器用于操作该变量。在 getRegister 内，需要通过 registerMap 表判断该变量是否已经有寄存器，若已经有，则可以从 RegisterMap 查找并直接返回寄存器。若该变量是首次出现，还没有分配过寄存器，则需要通过 allocateRegister 方法取一个空闲寄存器分配给这个变量，并将该变量和寄存器的映射关系加入到 registerMap 中，并返回寄存器。

```
1 usage   changdick
private String getRegister(String variable) {
    if (registerMap.containsKey(variable)) {
        return registerMap.get(variable);
    } else {
        String reg = allocateRegister();
        registerMap.put(variable, reg);
        return reg;
    }

}
```

allocateRegister 方法从空闲寄存器列表中取出一个空闲寄存器，返回给上层。当空闲寄存器列表没有可用寄存器时，将抛出异常。

```
private String allocateRegister() {
    if (freeRegisters.isEmpty()) {
        throw new RuntimeException("No available registers");
    }
    // 从空闲寄存器池中取出一个寄存器
    String reg = freeRegisters.removeLast(); // 从列表末尾取出
    usedRegisters.add(reg);
    return reg;
}
```

通过这一系列方法，可以实现寄存器的分配功能。在生成汇编语言的关键步骤中，只需要调用 getOperand 即可从中间指令操作数得到 RISC V 汇编操作数，而不必关心是立即数还是需要寄存器的变量。getOperand 内部调用的方法会自动做好新旧变量的寄存器的分配。

3.4.3 寄存器空闲检查和自动释放算法

每生成一句汇编语言后，程序将自动检查当前已分配寄存器的变量，在后续指令中是否还会使用。如果不会使用，那么对应的寄存器就已经是空闲的，应该进行释放，使后面的变量可以正常分配寄存器。该算法的原理是，直接遍历一遍后面的所有中间指令，将其中出现的变量全部用集合记录下来；再遍历 registerMap 中的变量，检查是否出现在后续指令中，若没有出现，则该变量不会再使用，其分配到的寄存器应该释放。

释放寄存器，即将该寄存器加入到 freeRegisters 中。这一操作封装为 freeRegister 方法。在实际的代码实现中，其实还是用了 usedRegisters 记录已经使用的寄存器，但是实际上 usedRegisters 完全删去，也不影响该程序的功能。所以只介绍 freeRegisters 列表。

```
3 usages   changdick
private void freeRegister(String reg) {
    usedRegisters.remove(reg);
    freeRegisters.add(reg);
}
```

自动检查寄存器空闲并释放的方法是 releaseUnusedRegisters 方法，该方法每次运行，将初始化一个集合，并遍历当前指令后面的所有指令，将未来会使用的变量加入集合。

```
Set<String> usedInFuture = new HashSet<>();

// 遍历当前指令之后的每条指令，检查变量的使用情况
for (int i = instructions.indexOf(currentInstruction) + 1; i < instructions.size(); i++) {
    Instruction instruction = instructions.get(i);
    InstructionKind kind = instruction.getKind();

    // 根据指令种类获取使用的变量
    if (kind.isBinary()) {
        usedInFuture.add(instruction.getLHS().toString());
        usedInFuture.add(instruction.getRHS().toString());
        usedInFuture.add(instruction.getResult().toString());
    } else if (kind.isUnary()) {
        usedInFuture.add(instruction.getFrom().toString());
        usedInFuture.add(instruction.getResult().toString());
    } else if (kind.isReturn()) {
        if (instruction.getReturnValue() != null) {
            usedInFuture.add(instruction.getReturnValue().toString());
        }
    }
}
```

之后将检查 registerMap 中的变量是否不在 usedInFuture 集合中。如果不在，说明该变量不会再次使用，即可释放其寄存器，且将其从 registerMap 中删去。

```
// 遍历 registerMap 中的变量，检查是否仍在使用
Iterator<Map.Entry<String, String>> iterator = registerMap.entrySet().iterator();
while (iterator.hasNext()) {
    Map.Entry<String, String> entry = iterator.next();
    String variable = entry.getKey();
    String register = entry.getValue();

    if (!usedInFuture.contains(variable)) {
        // 如果变量不再使用，释放寄存器
        iterator.remove(); // 从 registerMap 中移除该变量
        freeRegister(register); // 释放寄存器
    }
}
```

目标代码生成运行的是 run 方法。该方法会依次对每一条中间指令生成汇编语句。自动释放寄存器的方法就在生成一条汇编语句之后运行。这样可以保证每一条汇编语句生成后，空闲寄存器列表都是完全的。

```
2 usages   changdick *
public void run() {
    // TODO: 执行寄存器分配与代码生成
      throw new NotImplementedException();
    // 依次处理每一条指令
    for (Instruction instruction : instructions) {
        generateAssembly(instruction);
        releaseUnusedRegisters(instruction);
    }
}
```

3.4.4 汇编语句生成算法

汇编语言生成 generateAssembly 方法，用于给一条中间指令生成指定的汇编语句。首先

获得指令类型，根据指令类型获取操作数。在该方法中，只需要获取中间指令的操作数，用 getOperand 方法即可得到对应的汇编语言中的操作数。

```
// 根据指令的种类来决定如何获取操作数，取数的get跟指令的种类有关，
if (kind.isBinary()) {
    // 对于二元操作，获取 LHS 和 RHS
    src1 = getOperand(instruction.getLHS());
    src2 = getOperand(instruction.getRHS());
    target = getOperand(instruction.getResult());
    lhsImmediate = instruction.getLHS().isImmediate();
    rhsImmediate = instruction.getRHS().isImmediate();
} else if (kind.isUnary()) {
    // 对于 MOV 指令，只获取从操作数
    src2 = getOperand(instruction.getFrom());
    target = getOperand(instruction.getResult());
    rhsImmediate = instruction.getFrom().isImmediate();
} else if (kind.isReturn()) {
    // 对于 RET 指令，直接获取返回值
    src1 = getOperand(instruction.getReturnValue());
}
```

在获取到操作数的同时，还获取操作数是否为立即数的布尔值，用于后续一些判断的需要。

由于中间指令中，如(ADD, $2, 3, b)，左侧操作数是立即数 3，而右侧操作数是变量 b，不符合 RISC V 汇编语言的习惯。由于 RISC-V 汇编语言中，立即数必须是第 2 个操作数，否则无法生成正确的 addi 指令。因此，对于 ADD 类型指令，需要检查操作数是否为立即数，且是否左右相反。如果是 ADD 指令且左侧操作数是立即数，那么应该对调左右侧操作数。

此后，可以根据中间指令的类型，用不同的规则生成汇编语句。汇编语句全是字符串表示，用一个列表储存，只需要用操作数的拼接即可构造出来。对于加法，检查右侧操作数，如果为立即数，则构造一条 addi 指令，加入输出语句的字符串列表中。否则构造响应的 add 指令。

对于减法，如果右侧操作数为立即数，则构造一条 addi 指令，在第二个操作数前增加一个“-”。如果左侧操作数为立即数，先构造一条 li 指令，将立即数直接加载到存储减法结果的寄存器 target 上，在构造 sub 指令，源操作数 1 为 target，源操作数 2 为 src2，结果存入的寄存器仍为 target，就不通过中间寄存器，也完成了减法。若不涉及立即数，则直接构造 sub 指令。对于乘法，也是类似的。

中间指令的 MOV 型指令，实际上对应汇编语言的两类指令。当中间指令的 MOV 指令两个操作数都是变量，对应的是汇编语言的 mv 指令，而对于 MOV 指令中，把立即数赋值给变量的类型，对应汇编语言的 li 伪指令。对于中间指令的返回指令，由于汇编语言指定以 a0 寄存器存储返回值，只需要构造一条 mv 指令，将获取到返回值的寄存器值赋值到 a0 即可。

```
switch (kind) {
    case ADD:
        if (rhsImmediate) {
            assemblyInstructions.add("addi " + target + ", " + src1 + ", " + src2);
        } else {
            assemblyInstructions.add("add " + target + ", " + src1 + ", " + src2);
        }
        break;

    case SUB:
        if (rhsImmediate) {
            // 使用 addi 指令，并将立即数取负
            assemblyInstructions.add("addi " + target + ", " + src1 + ", -" + src2);
        } else if (lhsImmediate) {
            // 如果 src1 是立即数，用 li 加载立即数到寄存器，再用 sub
            // 此时，target已经取了寄存器，其实不必申请临时寄存器，而是直接把立即数加载到target中，
            // 然后用target减去减数，存target，一样完成减法，不需要临时寄存器。
              String tempReg = allocateRegister();
            assemblyInstructions.add("li " + target + ", " + src1);
            assemblyInstructions.add("sub " + target + ", " + target + ", " + src2);
              freeRegister(tempReg);  // 释放临时寄存器
        } else {
            assemblyInstructions.add("sub " + target + ", " + src1 + ", " + src2);
        }
        break;
```

```
case MUL:
    if (rhsImmediate) {
        String tempReg = allocateRegister();
        assemblyInstructions.add("li " + tempReg + ", " + src2);
        assemblyInstructions.add("mul " + target + ", " + src1 + ", " + tempReg);
        freeRegister(tempReg);  // 释放临时寄存器
    } else {
        assemblyInstructions.add("mul " + target + ", " + src1 + ", " + src2);
    }
    break;

case MOV:
    if (rhsImmediate) {
        // 如果是立即数，则使用 li 指令
        assemblyInstructions.add("li " + target + ", " + src2);
    } else {
        assemblyInstructions.add("mv " + target + ", " + src2);
    }
    break;

case RET:

    assemblyInstructions.add("mv a0, " + src1);
    break;
```

3.4.5 汇编代码输出

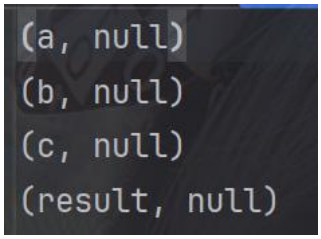
在上述 run 方法运行时，会对每一个中间指令调用 generateAssembly 方法，每次构造的汇编语句加入到 assemblyInstructions 列表中。run 执行结束后，汇编语言代码完整储存在该列表中，调用 Java 的 FileWriter 类逐行输出到文件，就完成了汇编代码的输出。

4 实验结果与分析

4.1 实验一分析

实验一输入为编码表 coding_map.csv 和源语言程序 input_code.txt。

输出为词法分析完成后的符号表 old_symbol_table.txt，以及 token 串 token.txt。



符号表 old_symbol_table.txt 中显示，a，b，c 和 result 都存入符号表中，并且类型字段均为 null。


```
PS C:\Users\83923\Desktop\BYYL\template> python3 .\scripts\diff.py .\data\out\token.txt .\data\std\token.txt
The src file is the same as std file.
```

token.txt 包含了词法分析后的 token 流。使用比对脚本将其与标准 token 串对照，是相同的，说明词法分析结果是正确的。

4.2 实验二分析

实验二额外输入语法规则文件 grammar.txt,以及 LR1 分析表 LR1_table.csv。

输出：较词法分析多输出 parser_list.txt，记录语法分析中按照规约顺序使用的产生式。

```
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
```

```
PS C:\Users\83923\Desktop\BYYL\template> python3 .\scripts\diff.py .\data\out\parser_list.txt .\data\std\parser_list.txt
The src file is the same as std file.
```

经过自动比对脚本验证，该文件和标准答案一致。

4.3 实验三分析

实验三不读入新的文件。输出文件为语义分析后新的符号表 new_symbol_table.txt，以及 IR 生成的中间表示指令 intermediate_code.txt，以及模拟运行中间指令的执行结果 ir_emulate_result.txt。

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

新符号表中，类型字段均从 null 变为 Int。语义分析之后，各个变量均确定了类型。

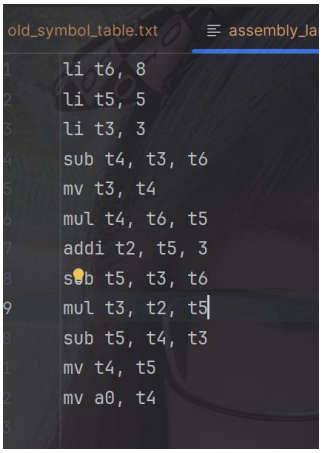
old_symbol_table.txt　intermediate_co

```
(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)
```

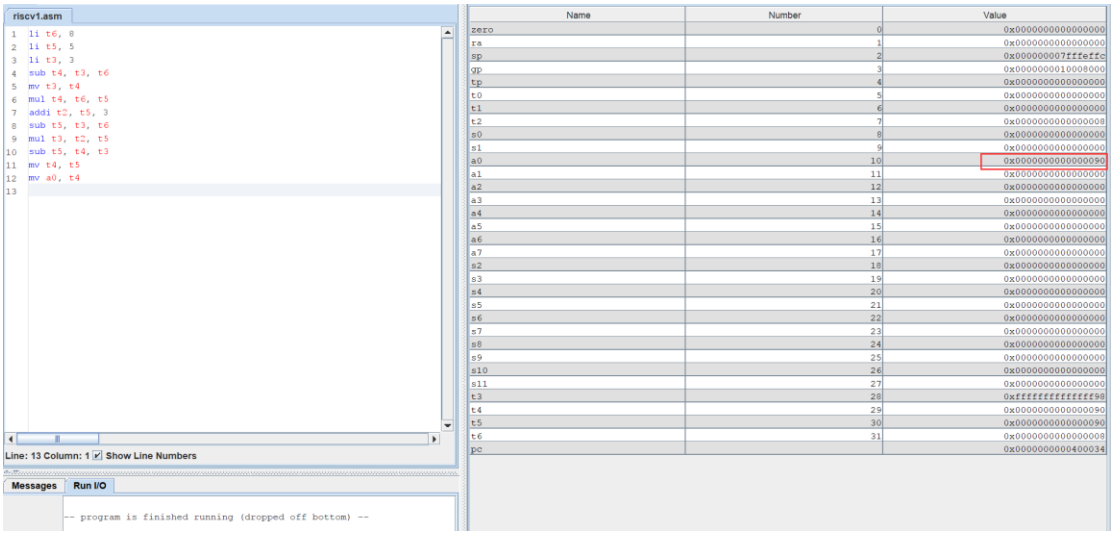
生成的中间表示指令。这些指令模拟运行的结果是 144，与标准答案一致。

4.4 实验四分析

实验四输出为源语言编译后的 RISC-V 汇编代码 assembly_language.asm。其内容如下。



可以看出，这段汇编代码只使用了 t0 到 t6 寄存器，并将返回值存储在 a0 返回给上层。将其复制到 rars.jar 中运行，比较结果。



如图，运行结束后，a0 寄存器的值为十六进制 90，即十进制 144，和预期的值相等。说明汇编代码和源语言代码的功能是一致的。

5 实验中遇到的困难与解决办法

实验中在熟悉框架代码花了很多时间。实验指导书对于框架代码讲的很破碎，代码里面的注释太深奥了，我和他的思维不在一个层面，看不懂他在说什么。缺乏一个完整的文档来介绍实验框架。建议给框架代码做一个完善的文档，精确到需要什么时可以调用哪个方法获得，有些框架代码让人不明白其意图在哪里。有几次自己实现了一个功能，花了很大篇幅代码，后来发现框架代码是有相应方法的，藏在某个类里面。感觉主要的难度还是来源于对框架代码的不熟悉，而不是设计编译器本身。比如实验四全让自己设计，反而做的比较简单。实验二和实验三去了解指令类、IR 值类、LR 分析表类以及相应方法，其实本来框架做的已经特别完全了，但是因为没梳理清楚框架代码的结构，写起来特别费劲。后来花时间慢慢了

解了几趟，大概设计出来了。

实验四没有按照指导书设计的双向映射表做，采用的结构偏简单，直接用一个空闲寄存器列表来管理空闲寄存器，用哈希表来存储变量和已经使用的寄存器，其实只有变量向寄存器的映射，就足够完成算法了。寄存器也是直接用字符串类型来存储和表示，感觉简单一点。