

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: 房煊梓
学 号: 210010101
专 业: 智能强基—计算机
日 期: 2023-11-09

1 实验目的与方法

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 语言单词符号的文法描述有更深入的认识，理解有限自动机、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。

实验方法：

- (1) 使用语言为 JAVA
- (2) 使用平台为 IntelliJ IDEA
- (3) 使用 JDK17

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) 使用语言为 JAVA
- (2) 使用平台为 IntelliJ IDEA
- (3) 使用 JDK17

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，理解中间代码的作用。

实验方法：

- (1) 使用语言为 JAVA
- (2) 使用平台为 IntelliJ IDEA
- (3) 使用 JDK17

1.4 目标代码生成

实验目的：

- (1) 加深对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。

实验方法：

- (1) 使用语言为 JAVA
- (2) 使用平台为 IntelliJ IDEA
- (3) 使用 JDK17
- (4) 使用 rars

2 实验内容及要求

2.1 词法分析器

实验内容：

编写一个词法分析程序，读取文件，对文件内的类 C 语言程序段进行词法分析。

输入：以文件形式存放的类 C 语言程序段。

输出：以文件形式存放的 TOKEN 串和简单符号表。

实验要求：

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。

2.2 语法分析

实验内容：

(1) 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) 输出推导过程中所用产生式序列并保存在输出文件中。

实验要求：

(1) 实验一的输出作为实验二的输入。

(2) 较低完成要求：实验模板代码中支持变量声明、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。

2.3 典型语句的语义分析及中间代码生成

实验内容：

(1) 采用实验二中的文法，为语法正确的单词串设计翻译方案，完成语法制导翻译。

(2) 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，中间代码使用三地址码的四元式表示。

(3) 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。

(4) 使用框架中的模拟器 IREmulator 验证生成的中间代码的正确性。

(5) 实验三无新增输入文件，intermediate_code.txt 可与 std 不一样，ir_emulate_result.txt 中内容正确即可。

实验要求：

(1) 设计文法的翻译方案；

(2) 设计各个观察者可能需要的数据结构；

(3) 实现各观察者。

2.4 目标代码生成

实验内容：

- (1) 将实验三生成的中间代码转换为目标代码（RISC-V 指令）；
- (2) 使用 RARS 运行生成的目标代码，验证结果的正确性。

实验要求：

- (1) 加载实验三生成的中间代码，视情况做预处理；
- (2) 实现寄存器选择算法和目标代码生成算法；
- (3) 输出生成的目标代码到指定文件中；
- (4) 使用 rars 运行目标代码，验证其正确性；
- (5) 使用 scripts 中的 check-result.py 检查实验四结果时，先要修改文件中的 rars_path。

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

本实验的编码表如下。

| coding_map.csv | |
|----------------|-------------|
| 1 | 1 int |
| 2 | 2 return |
| 3 | 3 = |
| 4 | 4 , |
| 5 | 5 Semicolon |
| 6 | 6 + |
| 7 | 7 - |
| 8 | 8 * |
| 9 | 9 / |
| 10 | 10 (|
| 11 | 11) |
| 12 | 51 id |
| 13 | 52 IntConst |

3.1.2. 正则文法

本实验的正则文法如下。

约定用 digit 表示数字: 0,1,2,...,9; 用 no_0_digit 表示数字: 1,2,...,9;

用 letter 表示字母: A,B,...,Z,a,b,...,z,_

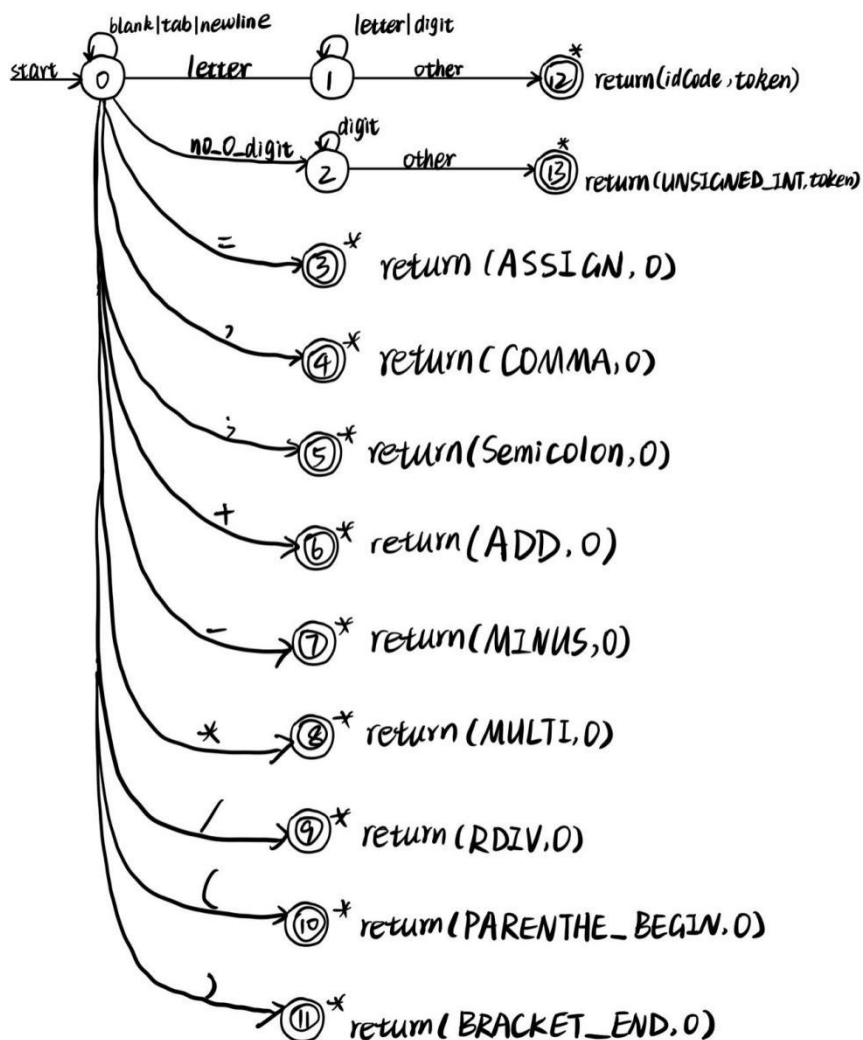
标识符: $S \rightarrow \text{blank } S \mid \text{tab } S \mid \text{newline } S \mid \text{letter } A$ $A \rightarrow \text{letter } A \mid \text{digit } A \mid \varepsilon$

运算符、分隔符: $S \rightarrow B$ $B \rightarrow = \mid ; \mid + \mid - \mid * \mid / \mid (\mid)$

整常数: $S \rightarrow \text{no_0_digit } B$ $B \rightarrow \text{digit } B \mid \varepsilon$

3.1.3. 状态转换图

状态转换图如下。



3.1.4. 词法分析程序设计思路和算法描述

词法分析程序的整体思路是结合当前输入和状态机进行转换，主要实现的文件为 LexicalAnalyzer.java。

1.主要成员为：符号表 symbolTable，读入的内容 text 和词法单元列表 Tokens。

```
3 usages
private final SymbolTable symbolTable;
5 usages
private String text;
15 usages
private List<Token> Tokens = new ArrayList<>();
```

2.主要方法有：loadFile，run，getTokens 和 dumpTokens。

(1)loadFile 方法：直接利用 FileUtils 实现词法分析前的缓冲区。

```
public void loadFile(String path) {
    // TODO: 词法分析前的缓冲区实现
    // 可自由实现各类缓冲区
    // 或直接采用完整读入方法

    //这里直接读入
    text=FileUtils.readFile(path);
    //System.out.println(text); //测试一下能否正常读入

    //throw new NotImplementedException();
}
```

(2)run 方法：是本次实验主要实现的方法，利用自动机实现词法分析过程。

主要思路为：利用 switch-case 实现状态机，根据当前的状态和读入的字符确定下一个状态以及是否形成词法单元加入列表。

状态 0 表示准备读入字符，针对读入字符的不同情况进行不同处理。下图表示当前字符为制表符、回车符、换行符、空格符、字母、下划线和非 0 的数字的情况。

```
//当前字符为制表符，回车符，换行符或空格符
if(ch=='\t' || ch=='\r' || ch=='\n' || ch==' '){
    //设置状态为0，跳过，去读下一个字符
    state=0;
    continue;
}

//当读到的字符为字母或下划线，则可能是标识符或关键字，将字符加入id串，设置状态为1，下次循环进入状态1
if(Character.isAlphabetic(ch) || ch=='_'){
    id+=ch;
    state=1;
}

//当读到的字符为非0的数字，则为IntConst，将字符加入num串，设置状态为2，下次循环进入状态2
else if(Character.isDigit(ch) && ch!='0'){
    num+=ch;
    state=2;
}else{
    //其他情况，根据读入的字符分别对应设置不同的状态，下次循环进行状态转移
```

其他情况较为简单，下图只展示其中一部分。

```
switch (ch){
    case '=':
        state = 3;
        break;
    case ',':
        state = 4;
        break;
    case ';':
        //测试一下是否读到了分号
        //System.out.println("read Semicolon");
        state = 5;
        break;
```

状态 1 表示标识符或关键字，根据当前符号判断“单词”是否已结束，采取相应行动。若未结束，则加入 id 串，下次循环进入状态 1；否则判断当前的 id 串是关键字还是标识符并加入词法单元，下次循环进入状态 0。

```
case 1:
    //状态为1，表示标识符或关键字

    //读取当前符号
    ch=text.charAt(pointer);
    //如果ch是字母或数字或下划线，则将其识别为id串的一部分，加入id串，并且设置状态为1，下次循环进入状态1
    if(Character.isAlphabetic(ch)||Character.isDigit(ch)||ch=='_'){
        id+=ch;
        state=1;
    }else{
        //如果读入的不是以上情况，则说明这个“单词”已经结束了，识别id串是关键字还是标识符，加入词法单元
        switch (id){
            case "return":
                token=Token.simple( tokenKindId: "return");
                Tokens.add(token);
                //System.out.println(id);
                break;
            case "int":
                token=Token.simple( tokenKindId: "int");
                Tokens.add(token);
                //System.out.println(id);
                break;
            default:
                token=Token.normal( tokenKindId: "id",id);
                Tokens.add(token);
                //检测符号表中是否已含有该标识符，若无向符号表加入该标识符即可
                if (!symbolTable.has(id)) {
                    symbolTable.add(id);
                    // System.out.println("new id: "+id);
                }
                //System.out.println("id: "+id);
                break;
        }
        //“单词”已经读完，重置id串和state
        id="";
        state=0;
        //为防止有符号被跳过，pointer回退
        pointer--;
    }
    break;
```

状态 2 的处理思路与状态 1 类似，根据当前字符采取相应行动。

```
case 2:
    //状态为2，表示IntConst

    //读入当前字符
    ch=text.charAt(pointer);
    //如果ch是数字，则将其识别为num串的一部分，加入num串，并且设置状态为2，下次循环进入状态2
    if(Character.isDigit(ch)){
        num+=ch;
        state=2;
    }else{
        //如果读入的不是以上情况，则“单词”已结束，加入词法单元
        token=Token.normal( tokenKindId: "IntConst",num);
        Tokens.add(token);
        //System.out.println(num);
        //“单词”已经读完，重置num串和state
        num="";
        state=0;
        //为防止有符号被跳过，pointer回退
        pointer--;
    }
    break;
```

其他情况较为简单，直接根据相应的符号加入词法单元并且重置状态即可。截图只展示一部分。

```
//其他情况较为简单，直接根据相应的符号加入词法单元并且重置状态即可
case 3:
    token=Token.simple( tokenKindId: "=");
    Tokens.add(token);
    state=0;
    break;
case 4:
    token=Token.simple( tokenKindId: ",");
    Tokens.add(token);
    state=0;
    break;
```

当所有字符读完后，加入 eof。

```
//已经读完所有字符
Tokens.add(Token.eof());
```

(3)getTokens 方法：直接返回列表 Tokens 即可。

```
public Iterable<Token> getTokens() {
    // TODO: 从词法分析过程中获取 Token 列表
    // 词法分析过程可以使用 Stream 或 Iterator 实现按需分析
    // 亦可以直接分析完整文件
    // 总之实现过程能转化为一列表即可

    //System.out.println("getTokens\n");
    return Tokens;

    //throw new NotImplementedException();
}
```

(4)dumpTokens 方法：利用 FileUtils 实现。

```
public void dumpTokens(String path) {
    //System.out.println("dump Tokens\n");
    FileUtils.writeLines(
        path,
        StreamSupport.stream(getTokens().spliterator(), parallel: false).map(Token::toString).toList()
    );
}
```


| ACTION | | | | | | | | | | | | | GOTO | | | | |
|--------|----------------|----------|---------------------|---------------------|---------------------|---------------------|---|---------|---------|----------|-----------|------------------------------------|------|--------|---|----|----|
| 状态 | id | (|) | + | - | * | = | int | return | IntConst | Semicolon | \$ | E | S_list | S | A | B |
| 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 | | | | | | reduce A → A * B | | | | | |
| 30 | | | reduce B → (E) | 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) | | | | | | | | | | | |

3.2.3 状态栈和符号栈的数据结构和设计思路

状态栈和符号栈的创建代码如下：

```
Stack<Status> statusStack = new Stack<>();
```

```
Stack<Symbol> symbolStack = new Stack<>();
```

1. 状态栈的数据结构

状态栈中存储的元素是 Status，一个状态由 action 和 goto 组成，它们的语义如下：

action: 在当前状态下遇到某个终结符需要采取什么动作。

goto: 在当前状态下遇到（规约到）某个非终结符需要转移到什么状态。

```
public record Status(int index, Map<TokenKind, Action> action, Map<NonTerminal, Status> goto_) {  
    /**  
     * 构造一个状态  
     *  
     * @param index 状态的索引/编号  
     * @return 构造出的状态  
     */  
    2 usages  
    public static Status create(int index) {  
        if (index < 0) {  
            throw new RuntimeException("Index of status can NOT smaller than zero");  
        }  
  
        return new Status(index);  
    }  
}
```

```
3 usages  
public Action getAction(TokenKind terminal) { return action.getOrDefault(terminal, Action.error()); }  
  
/**  
 * 当遇到该词法单元时，应该转移到哪个状态  
 *  
 * @param token 词法单元  
 * @return 应该转移到的状态  
 */  
no usages  
public Action getAction(Token token) { return getAction(token.getKind()); }  
  
/**  
 * 当规约到该非终结符时，应该转移到哪个状态  
 *  
 * @param nonTerminal 非终结符  
 * @return 应该转移到的状态  
 */  
2 usages  
public Status getGoto(NonTerminal nonTerminal) { return goto_.getOrDefault(nonTerminal, Status.error()); }
```

2. 符号栈的数据结构

符号栈中存储的元素是 Symbol，一个符号可能是终结符或非终结符，定义一个 Symbol 来实现 Union<Token, NonTerminal> 的功能，设计了相关的构造方法和其他的方法。

```

public class Symbol {
    3 usages
    Token token;
    3 usages
    NonTerminal nonTerminal;

    2 usages
    private Symbol(Token token, NonTerminal nonTerminal){
        this.token = token;
        this.nonTerminal = nonTerminal;
    }

    2 usages
    public Symbol(Token token) { this(token, nonTerminal: null); }

    1 usage
    public Symbol(NonTerminal nonTerminal) { this(token: null, nonTerminal); }

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

    no usages
    public boolean isNonTerminal() { return this.nonTerminal != null; }

    no usages
    public Token Symbol_getToken(){ return this.token;}

    1 usage
    public NonTerminal Symbol_getNonTerminal(){ return this.nonTerminal;}
}

```

3.2.4 LR 驱动程序设计思路和算法描述

LR 驱动程序的整体思路是在遇到 Shift/Reduce/Accept 时实现状态栈和符号栈的变化并执行相应的 call 函数，主要实现的文件为 SyntaxAnalyzer.java。

1.主要成员为：词法单元列表 tokenList，初始状态 init_status，LR 分析表 lrTable 以及上述状态栈 statusStack 和符号栈 symbolStack。

```

5 usages
List<Token> tokenList = new ArrayList<>();

```

```

2 usages
Status init_status;
3 usages
LRTable lrTable;

```

2.主要方法有：loadTokens，loadLRTable 和 run。

(1)loadTokens 方法：加载词法单元，将输入的 tokens 用 tokenList 存起来。

```

5 usages
List<Token> tokenList = new ArrayList<>();
1 usage
public void loadTokens(Iterable<Token> tokens) {
    // TODO: 加载词法单元
    // 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
    // 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，
    // 在自行设计的时候请加以考虑此种情况
    //throw new NotImplementedException();

    for (Token token : tokens) {
        //System.out.println("token:"+token);
        tokenList.add(token);
    }
    //System.out.println("tokenList:"+tokenList);
}

```

(2)loadLRTable 方法：加载 LR 分析表，保存初始状态到 init_status，并将输入的 LR 分析表赋给 lrTable。

```

2 usages
Status init_status;
3 usages
LRTable lrTable;
1 usage
public void loadLRTable(LRTable table) {
    // TODO: 加载 LR 分析表
    // 你可以自行选择要如何使用该表格：
    // 是直接对 LRTable 调用 getAction/getGoto，抑或是直接将 initStatus 存起来使用
    // throw new NotImplementedException();

    // 将初始状态保存下来
    init_status = table.getInit();
    // System.out.println("init status:"+init_status);

    lrTable = table;
}

```

(3) run 方法：实现驱动程序。首先需要建立状态栈和符号栈并进行初始化，然后循环读入 token 并获取状态栈的栈顶元素，据此查 LR 分析表获取下一个执行动作。其中 flag 表示接受状态，每次循环时都检查 flag，当其为 1 时退出循环，即语法分析执行结束。

```

int i=0;
int flag=0;
while(i<tokenList.size()){
    if(flag==1){break;}
    //System.out.println("当前i:"+i);

    //获取待读入的下一个token
    Token token_now = tokenList.get(i);
    //System.out.println("待读入token:"+token_now);

    //获取状态栈栈顶元素
    Status status_now = statusStack.peek();
    //System.out.println("状态栈栈顶元素: "+status_now);

    //根据状态栈栈顶元素和待读入的下一个token查询判断下一个待执行动作
    Action action = lrTable.getAction(status_now, token_now);
}

```

再利用 switch-case 针对不同的动作采取不同的做法：

若为 shift，把 action 对应的状态压入状态栈，对应的 token 压入符号栈，调用相关 call 函数。

```

switch (action.getKind()) {
    case Shift -> {
        //如果是shift，把action对应的状态压入状态栈，对应的token压入符号栈
        final var shiftTo = action.getStatus();
        statusStack.push(shiftTo);
        Symbol shiftSymbol = new Symbol(token_now);
        symbolStack.push(shiftSymbol);
        //由于是移入动作，下次应当读下一个token了
        i++;
        //调用相关call
        callWhenInShift(status_now, token_now);
    }
}

```

若为 reduce，根据产生式长度，符号栈和状态栈均弹出对应长度个 token 和状态，再将产生式左侧的非终结符压入符号栈，根据符号栈和状态栈栈顶状态获取 goto 表的状态，压入状态栈，然后调用相关 call 函数。

```

case Reduce -> {
    //如果是reduce, 根据产生式长度, 符号栈和状态栈均弹出对应长度个token和状态
    final var production = action.getProduction();
    int length = production.body().size();
    for(int j=0;j<length;j++){
        symbolStack.pop();
        statusStack.pop();
    }
    //产生式左侧的非终结符压入符号栈
    Symbol reduceSymbol = new Symbol(production.head());
    symbolStack.push(reduceSymbol);
    //根据符号栈和状态栈栈顶状态获取goto表的状态, 压入状态栈
    Status reduceStatus = lrTable.getGoto(statusStack.peek(), symbolStack.peek().Symbol_getNonTerminal());
    statusStack.push(reduceStatus);
    //调用相关call
    callWhenInReduce(status_now, production);
}

```

若为 accept, 语法分析执行结束, 调用相关 call 函数并置 flag 为 1 即可。而若为 error, 不采取行动。

```

case Accept -> {
    //如果是accept, 语法分析执行结束
    //调用相关call
    callWhenInAccept(status_now);
    flag=1;
}
case Error -> {
    //如果是error
    //System.out.println("出错!");
}
default -> {
    //do nothing
}

```

3.3 语义分析和中间代码生成

3.3.1 翻译方案

采用 S-属性定义的自底向上翻译方案。翻译方案如下:

1. $P \rightarrow S_list; \{P.val = S_list.val;\}$
2. $S_list \rightarrow S \text{ Semicolon } S_list1; \{S_list.val = S_list1.val;\}$
3. $S_list \rightarrow S \text{ Semicolon}; \{S_list.val = S.val;\}$
4. $S \rightarrow D \text{ id}; \{p = \text{lookup}(\text{id.name}); \text{if } p \neq \text{nil} \text{ then enter}(\text{id.name}, D.type) \text{ else error}\}$
5. $D \rightarrow \text{int}; \{D.type = \text{int};\}$
6. $S \rightarrow \text{id} = E; \{\text{gencode}(\text{id.val} = E.val);\}$
7. $S \rightarrow \text{return } E; \{S.val = E.val;\}$
8. $E \rightarrow E + A; \{E.val = E1.val + A.val;\}$
9. $E \rightarrow E - A; \{E.val = E1.val - A.val;\}$
10. $E \rightarrow A; \{E.val = A.val;\}$
11. $A \rightarrow A * B; \{A.val = A1.val * B.val;\}$
12. $A \rightarrow B; \{A.val = B.val;\}$
13. $B \rightarrow (E); \{B.val = E.val;\}$
14. $B \rightarrow \text{id}; \{B.val = \text{id.val};\}$
15. $B \rightarrow \text{IntConst}; \{B.val = \text{IntConst.lexval};\}$

3.3.2 语义分析和中间代码生成的数据结构

由于有符号栈，故扩展符号栈的数据结构，增加数据类型 `SourceCodeType` 作为语义分析栈的数据结构；符号栈基础上，增加 `IRValue` 数据类型。修改之后的符号类如下，修改了一些成员和方法。

```
public class Symbol {
    7 usages
    Token token;
    7 usages
    NonTerminal nonTerminal;

    //增加数据类型SourceCodeType作为语义分析栈的数据结构
    5 usages
    SourceCodeType type;
    //语义栈基础上，增加IRValue数据类型
    18 usages
    IRValue value;

    2 usages
    private Symbol(Token token, NonTerminal nonTerminal){...}

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

    3 usages
    public Symbol(NonTerminal nonTerminal) { this(token: null, nonTerminal); }

    //增加相关构造方法
    1 usage
    public Symbol(Token token, NonTerminal nonTerminal, SourceCodeType type){...}
    1 usage
    public Symbol(NonTerminal nonTerminal, SourceCodeType type) { this(token: null, nonTerminal, type); }

    1 usage
    public Symbol(IRValue value){...}

    15 usages
    public Symbol(NonTerminal nonTerminal, IRValue value){...}

    2 usages
    public Symbol(Token token, SourceCodeType type) {...}

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

    public boolean isNonterminal() { return this.nonTerminal != null; }

    1 usage
    public Token Symbol_getToken(){ return this.token;}

    1 usage
    public NonTerminal Symbol_getNonTerminal(){ return this.nonTerminal;}
}
```

3.3.3 语法分析程序设计思路和算法描述

语法分析程序主要实现语义分析 `SemanticAnalyzer`（在自底向上的分析过程中更新符号表）和中间代码生成 `IRGenerator`（生成中间代码）。

1. SemanticAnalyzer

(1)主要成员为：符号表 `symbolTable` 和符号栈 `symbolStack`。

```
3 usages
SymbolTable symbolTable;

10 usages
Stack<Symbol> symbolStack = new Stack<>();
```


(2)主要方法有：whenAccept, whenReduce, whenShift 和 setSymbolTable。

①whenAccept 方法：不采取动作。

```
@Override
public void whenAccept(Status currentStatus) {
    // TODO: 该过程在遇到 Accept 时要采取的代码动作
    //throw new NotImplementedException();
}
```

②whenReduce 方法：

当 $S \rightarrow D \text{ id}$ 这条产生式归约时，取出 D 的 type，这个 type 就是 id 的 type，更新符号表中相应变量的 type 信息，压入空记录占位($D \text{ id}$ 被归约为 S ， S 不需要携带信息)。

```
switch (production.index()){
    case 4 ->{ //S -> D id;
        //当S->D id这条产生式归约时，取出D的type，这个type就是id的type，更新符号表中相应变量的type信息，
        //压入空记录占位(D id被归约为S，S不需要携带信息);

        //lookup 获得id的名字，用于在符号表中查找
        String id_name = symbolStack.peek().Symbol_getToken().getText();
        //弹栈
        symbolStack.pop();
        //现在栈顶为D
        Symbol D = symbolStack.peek();
        //更新符号表，把id的类型更新为D的类型
        symbolTable.get(id_name).setType(D.type);
        //再把D弹栈
        symbolStack.pop();
        //压入空记录占位
        Symbol S = new Symbol(production.head());
        symbolStack.push(S);

        break;
    }
}
```

当 $D \rightarrow \text{int}$ 这条产生式归约时， int 这个 token 应该在语义分析栈中，把这个 token 的 type 类型赋值给 D 的 type。

```
case 5 ->{ //D -> int;
    //当D->int这条产生式归约时，int这个token应该在语义分析栈中，把这个token的type类型赋值给D的type;
    NonTerminal nonTerminal = production.head();
    //D.type = Int
    Symbol D = new Symbol(nonTerminal, SourceCodeType.Int);
    //弹出右部，压入左部
    symbolStack.pop();
    symbolStack.push(D);

    break;
}
```

其他情况压入空记录占位即可。

```
default -> { //其他情况压入空记录占位即可
    //先把产生式右部弹栈
    for(int i=0;i<production.body().size();i++){
        symbolStack.pop();
    }
    //再把产生式左部压栈
    Symbol S = new Symbol(production.head());
    symbolStack.push(S);

    break;
}
```

③whenShift 方法:

自底向上分析过程中遇到 Shift 时将符号的 type 属性（从 Token 中获得）入栈。

```
public void whenShift(Status currentStatus, Token currentToken) {
    // TODO: 该过程在遇到 shift 时要采取的代码动作
    //throw new NotImplementedException();
    //Shift时将符号的type属性（从Token中获得）入栈
    String currentName = currentToken.getText();
    Symbol currentSymbol;
    //currentSymbol = new Symbol(currentToken);
    if(currentName.length()==0){
        currentSymbol = new Symbol(currentToken);
    }else if(currentToken.getKind().getCode()==52){
        //System.out.println("text:"+currentName);
        currentSymbol = new Symbol(currentToken,SourceCodeType.Int);
    }else{
        //System.out.println("text:"+currentName+"    type:"+symbolTable.get(currentName).getType());
        currentSymbol = new Symbol(currentToken,symbolTable.get(currentName).getType());
    }
    /**if(currentToken.getKind().getCode()==52){
        currentSymbol = new Symbol(currentToken,SourceCodeType.Int);
    }else{
        currentSymbol = new Symbol(currentToken);
    }*/
    symbolStack.push(currentSymbol);
}
```

④setSymbolTable 方法: 将输入的 table 赋给 symbolTable 即可。

```
@Override
public void setSymbolTable(SymbolTable table) {
    // TODO: 设计你可能需要的符号表存储结构
    // 如果需要使用符号表的话, 可以把它或者它的一部分信息存起来, 比如使用一个成员变量存储
    //throw new NotImplementedException();
    symbolTable = table;
}
```

2.IRGenerator

(1)主要成员为: 指令列表 instructions 和符号栈 symbolStack。

```
6 usages
List<Instruction> instructions = new ArrayList<>();
57 usages
Stack<Symbol> symbolStack = new Stack<>();
```

(2)主要方法有: whenShift, whenReduce, whenAccept, setSymbolTable, getIR 和 dumpIR。

①whenShift 方法: 分析过程中遇到 Shift 时先判断当前的 token 的类型并据此确定其 IRValue, 然后将符号加入符号栈。

```
public void whenShift(Status currentStatus, Token currentToken) {
    // TODO
    //throw new NotImplementedException();
    //首先要判断token是立即数还是变量
    TokenKind currentKind = currentToken.getKind();
    int currentCode = currentKind.getCode();
    IRValue value;

    //由coding_map,IntConst的代码为52
    if(currentCode==52){
        //获取立即数的值作为value
        int val = Integer.parseInt(currentToken.getText());
        value = IRImmediate.of(val);
    }else{
        //获取变量的名字作为value
        String val = currentToken.getText();
        value = IRVariable.named(val);
    }
    //入栈
    Symbol symbol = new Symbol(value);
    symbolStack.push(symbol);
}
```


②whenReduce 方法：根据产生式索引进行符号栈的变化以及指令的生成。大致思路是在将产生式右部弹栈的过程中保存需要的信息，然后产生左部的符号进行压栈，并根据产生式选择是否生成指令和生成什么指令。

对产生式 1：处理产生式，将符号压入语义栈。

```
switch (production.index()){
    case 1 ->{ //P -> S_list;
        //符号栈变化
        //右部弹栈
        Symbol S_list = symbolStack.peek();
        symbolStack.pop();

        //将S_list的value给P
        Symbol P = new Symbol(production.head(),S_list.value);
        //左部压栈
        symbolStack.push(P);

        break;
    }
}
```

对产生式 2：处理产生式，将符号压入语义栈。

```
case 2 ->{ //S_list -> S Semicolon S_list;
    //符号栈变化
    //右部弹栈
    for(int i=0;i<production.body().size();i++){
        symbolStack.pop();
    }
    Symbol S_list = new Symbol(production.head(),IRVariable.named("S_list1"));
    //左部压栈
    symbolStack.push(S_list);

    break;
}
```

产生式 3, 4, 5 的情况与产生式 2 类似，只是 IRVariable 的名字不一样，这里不再展示。

对产生式 6：处理产生式，将符号压入语义栈，并且生成 MOV 指令，加入指令列表。

```
case 6 ->{ //S -> id = E;
    //符号栈变化
    //右部弹栈
    Symbol E = symbolStack.peek();
    symbolStack.pop();
    symbolStack.pop();
    Symbol id = symbolStack.peek();
    symbolStack.pop();

    //把id的value给S
    Symbol S = new Symbol(production.head(),id.value);
    //左部压栈
    symbolStack.push(S);

    //生成指令
    IRValue value = E.value;
    Instruction instruction = Instruction.createMov((IRVariable) id.value,value);
    instructions.add(instruction);

    break;
}
```

对产生式 7：处理产生式，将符号压入语义栈，并且生成 RET 指令，加入指令列表。

```
case 7 ->{ //S -> return E;
    //符号栈变化
    //右部弹栈
    Symbol E = symbolStack.peek();
    symbolStack.pop();
    symbolStack.pop();

    Symbol S = new Symbol(production.head(),IRVariable.named("S2"));
    //左部压栈
    symbolStack.push(S);

    //生成指令
    Instruction instruction = Instruction.createRet(E.value);
    instructions.add(instruction);

    break;
}
```

对产生式 8：处理产生式，将符号压入语义栈，并且生成 ADD 指令，加入指令列表。

```
case 8 ->{ //E -> E + A;
    //符号栈变化
    //右部弹栈
    Symbol A = symbolStack.peek();
    symbolStack.pop();
    symbolStack.pop();
    Symbol E1 = symbolStack.peek();
    symbolStack.pop();

    IRVariable res = IRVariable.temp();
    Symbol E = new Symbol(production.head(),res);
    //左部压栈
    symbolStack.push(E);

    //生成指令
    Instruction instruction = Instruction.createAdd(res, E1.value,A.value);
    instructions.add(instruction);

    break;
}
```

对产生式 9：处理产生式，将符号压入语义栈，并且生成 SUB 指令，加入指令列表。

```
case 9 ->{ //E -> E - A;
    //符号栈变化
    //右部弹栈
    Symbol A = symbolStack.peek();
    symbolStack.pop();
    symbolStack.pop();
    Symbol E1 = symbolStack.peek();
    symbolStack.pop();

    IRVariable res = IRVariable.temp();
    Symbol E = new Symbol(production.head(),res);
    //左部压栈
    symbolStack.push(E);

    //生成指令
    Instruction instruction = Instruction.createSub(res, E1.value,A.value);
    instructions.add(instruction);

    break;
}
```

对产生式 10: 处理产生式, 将符号压入语义栈。

```
case 10 ->{ //E -> A;
    //符号栈变化
    //右部弹栈
    Symbol A = symbolStack.peek();
    symbolStack.pop();

    //把A的value给E
    Symbol E = new Symbol(production.head(),A.value);
    //左部压栈
    symbolStack.push(E);

    break;
}
```

对产生式 11: 处理产生式, 将符号压入语义栈, 并且生成 MUL 指令, 加入指令列表。

```
case 11 ->{ //A -> A * B;
    //符号栈变化
    //右部弹栈
    Symbol B = symbolStack.peek();
    symbolStack.pop();
    symbolStack.pop();
    Symbol A1 = symbolStack.peek();
    symbolStack.pop();

    IRVariable res = IRVariable.temp();
    Symbol A = new Symbol(production.head(),res);
    //左部压栈
    symbolStack.push(A);

    //生成指令
    Instruction instruction = Instruction.createMul(res, A1.value,B.value);
    instructions.add(instruction);

    break;
}
```

对产生式 12: 处理产生式, 将符号压入语义栈。

```
case 12 ->{ //A -> B;
    //符号栈变化
    //右部弹栈
    Symbol B = symbolStack.peek();
    symbolStack.pop();

    //把B的value给A
    Symbol A = new Symbol(production.head(),B.value);
    //左部压栈
    symbolStack.push(A);

    break;
}
```

对产生式 13：处理产生式，将符号压入语义栈。

```
case 13 ->{ //B -> ( E );
    //符号栈变化
    //右部弹栈
    symbolStack.pop();
    Symbol E = symbolStack.peek();
    symbolStack.pop();
    symbolStack.pop();

    //把E的value给B
    Symbol B = new Symbol(production.head(),E.value);
    //左部压栈
    symbolStack.push(B);

    break;
}
```

对产生式 14：处理产生式，将符号压入语义栈。

```
case 14 ->{ //B -> id;
    //符号栈变化
    //右部弹栈
    Symbol id = symbolStack.peek();
    symbolStack.pop();

    //把id的value给B
    Symbol B = new Symbol(production.head(),id.value);
    //左部压栈
    symbolStack.push(B);

    break;
}
```

对产生式 15：处理产生式，将符号压入语义栈。

```
case 15 ->{ //B -> IntConst;
    //符号栈变化
    //右部弹栈
    Symbol IntConst = symbolStack.peek();
    symbolStack.pop();

    //把IntConst的value给B
    Symbol B = new Symbol(production.head(),IntConst.value);
    //左部压栈
    symbolStack.push(B);

    break;
}
```

③whenAccept 方法：不采取行动。

```
@Override
public void whenAccept(Status currentStatus) {
    // TODO
    //throw new NotImplementedException();
}
```

④setSymbolTable 方法：不采取行动。

```
@Override
public void setSymbolTable(SymbolTable table) {
    // TODO
    //throw new NotImplementedException();
}
```

⑤getIR 方法：直接返回指令列表 instructions 即可。

```
public List<Instruction> getIR() {
    // TODO
    //throw new NotImplementedException();
    return instructions;
}
```

⑥dumpIR 方法：利用 FileUtils 实现。

```
public void dumpIR(String path) {
    FileUtils.writeLines(path, getIR().stream().map(Instruction::toString).toList());
}
```

3.4 目标代码生成

3.4.1 设计思路和算法描述

目标代码生成程序主要实现 AssemblyGenerator.java。

1.主要成员为：预处理后的指令列表 preprocessedInstructions，变量与寄存器的双射 map，记录变量使用计数的 usageMap，空闲寄存器列表 available_Regs 和生成的 riscv 指令列表 riscvInstructions。

```
/**保存预处理后的指令*/
9 usages
List<Instruction> preprocessedInstructions = new ArrayList<>();

/**双射*/
6 usages
BMap<IRVariable,String> map = new BMap<>();

/**usageMap, 用于记录变量的使用*/
8 usages
Map<IRVariable,Integer> usageMap = new HashMap<>();

/**空闲寄存器列表*/
5 usages
List<String> available_Regs = new ArrayList<>{
    Arrays.asList("t0","t1","t2","t3","t4","t5","t6")
};

/**生成的riscv指令列表*/
4 usages
List<String> riscvInstructions = new ArrayList<>{
    Arrays.asList(".text")
};
```

其中，根据实验指导书，双射 BMap 的代码如下：

```
package cn.edu.hitsz.compiler.asm;

import java.util.HashMap;
import java.util.Map;

// 双射
// 2 usages
public class BMap<K, V> {
    // 5 usages
    private final Map<K, V> KVmap = new HashMap<>();
    // 5 usages
    private final Map<V, K> VKmap = new HashMap<>();

    // 1 usage
    public void removeByKey(K key) { KVmap.remove(KVmap.remove(key)); }

    // 1 usage
    public void removeByValue(V value) {
        KVmap.remove(VKmap.remove(value));
    }

    // 1 usage
    public boolean containsKey(K key) { return KVmap.containsKey(key); }

    // no usages
    public boolean containsValue(V value) { return VKmap.containsKey(value); }

    public void replace(K key, V value) {
        // 对于双射关系，将会删除交叉项
        removeByKey(key);
        removeByValue(value);
        KVmap.put(key, value);
        VKmap.put(value, key);
    }

    // 3 usages
    public V getByKey(K key) { return KVmap.get(key); }

    // no usages
    public K getByValue(V value) { return VKmap.get(value); }
}
```

2.主要方法有：loadIR，usageMap_Add，Reg_Allocate，run 和 dump。

(1)loadIR 方法：加载实验三生成的中间代码，遍历每一条原始指令，视情况做预处理。

①对于 BinaryOp(两个操作数的指令)：

将操作两个立即数的 BinaryOp 直接进行求值得到结果，然后替换成 MOV 指令。

```
if(kind.isBinary()){
    //对于 BinaryOp(两个操作数的指令):
    IRValue LHS = originInstruction.getLHS();
    IRValue RHS = originInstruction.getRHS();
    if(LHS.isImmediate() && RHS.isImmediate()){
        //将操作两个立即数的 BinaryOp 直接进行求值得到结果，然后替换成 MOV 指令
        IRImmediate LHS_Immediate = (IRImmediate)LHS;
        IRImmediate RHS_Immediate = (IRImmediate)RHS;

        //直接求值
        int res=0;
        if(kind==InstructionKind.ADD){
            res = LHS_Immediate.getValue() + RHS_Immediate.getValue();
        }else if(kind==InstructionKind.SUB){
            res = LHS_Immediate.getValue() - RHS_Immediate.getValue();
        }else if(kind==InstructionKind.MUL){
            res = LHS_Immediate.getValue() * RHS_Immediate.getValue();
        }

        //创建MOV指令，加入列表
        IRVariable variable = IRVariable.temp();
        IRImmediate immediate = IRImmediate.of(res);
        Instruction newInstruction = Instruction.createMov(variable,immediate);
        preprocessedInstructions.add(newInstruction);
    }
}
```


将操作一个立即数的乘法和左立即数减法调整，前插一条 MOV a, imm，用 a 替换原立即数，将指令调整为无立即数指令。

```

}else if(kind==InstructionKind.MUL&&(LHS.isImmediate()||RHS.isImmediate())
||kind==InstructionKind.SUB&&LHS.isImmediate()){
    //将操作一个立即数的乘法和左立即数减法调整
    IRImmediate immediate =null;
    IRVariable a = IRVariable.temp();

    if(LHS.isImmediate()){
        immediate =(IRImmediate) (originInstruction.getLHS());
    }else if(RHS.isImmediate()){
        immediate =(IRImmediate) (originInstruction.getRHS());
    }

    //前插一条MOV a,imm
    Instruction newInstruction1 = Instruction.createMov(a,immediate);
    preprocessedInstructions.add(newInstruction1);

    //用a替换原立即数，将指令调整为无立即数指令
    Instruction newInstruction2=null;
    IRVariable result = originInstruction.getResult();
    if(LHS.isImmediate()){
        if(kind==InstructionKind.SUB){
            newInstruction2=Instruction.createSub(result,a,RHS);
        }else{
            newInstruction2=Instruction.createMul(result,a,RHS);
        }
    }else if(RHS.isImmediate()){
        newInstruction2=Instruction.createMul(result,LHS,a);
    }
    preprocessedInstructions.add(newInstruction2);

```

将操作一个立即数的指令(除了乘法和左立即数减法)进行调整，使之满足 a:=b op imm 格式。

```

}else if(LHS.isImmediate()||RHS.isImmediate()){
    //将操作一个立即数的指令（除了乘法和左立即数减法）进行调整，使之满足 a := b op imm 的格式
    Instruction newInstruction;
    IRVariable result = originInstruction.getResult();
    if(kind==InstructionKind.ADD&&LHS.isImmediate()){
        newInstruction=Instruction.createAdd(result,RHS,LHS);
    }else{
        newInstruction=originInstruction;
    }
    preprocessedInstructions.add(newInstruction);

```

其余情况不做处理。

```

}else{
    preprocessedInstructions.add(originInstruction);
}

```

②对于 UnaryOp(一个操作数的指令):

```

}else if(kind.isUnary()){
    preprocessedInstructions.add(originInstruction);
}

```

③根据语言规定，当遇到 Ret 指令后直接舍弃后续指令。

```
}else if(kind.isReturn()){
    //根据语言规定，当遇到 Ret 指令后直接舍弃后续指令。
    preprocessedInstructions.add(originInstruction);
    return;
}
```

(2)usageMap_Add 方法：输入变量，当 usageMap 中已经含有该变量时，将其对应的值（即该变量对应的计数）加 1，否则将该变量添加到 usageMap 中，值为 1。

```
void usageMap_Add(IRVariable variable){
    if(usageMap.containsKey(variable)){
        //如果已经有了，则值加1
        usageMap.put(variable,usageMap.get(variable)+1);
    }else{
        //否则添加到usageMap中，并且将值置为1
        usageMap.put(variable,1);
    }
}
```

(3)Reg_Allocate 方法：根据输入的变量进行寄存器分配。

若该变量已在寄存器中，则直接使用当前的寄存器。

```
if(map.containsKey(variable)){
    //如果已经在寄存器中，使用当前寄存器
    reg=map.getByKey(variable);
}
```

否则，若还有空闲寄存器，则分配一个空闲寄存器；若空闲寄存器列表为空，即没有空闲寄存器了，则需要夺取不再使用的变量所占的寄存器。

```
}else{
    if(!available_Regs.isEmpty()){
        //若还有空闲寄存器，分配一个（这里选择列表最后一个）
        reg=available_Regs.get(available_Regs.size()-1);
        map.replace(variable,reg);
        //并且将分配的寄存器从空闲寄存器列表中移出
        available_Regs.remove(index: available_Regs.size()-1);
    }else{
        //若空闲寄存器列表为空，即没有空闲寄存器了，则需要夺取不再使用的变量所占的寄存器
        for (IRVariable var : usageMap.keySet()) {
            //看usageMap，当寄存器中有该变量且对应的值为0，说明不再使用，则夺取该寄存器
            if(map.getByKey(var)!=null&&usageMap.get(var)==0){
                reg=map.getByKey(var);
                break;
            }
        }
        //将变量存入寄存器
        map.replace(variable,reg);
    }
}
```


分配好之后需要将计数减 1。

```
//分配之后，计数减1
usageMap.put(variable, usageMap.get(variable)-1);
```

(4)run 方法:

①首先，循环对每一条预处理后的指令处理相应的 usageMap。

对于 BinaryOp(两个操作数的指令)，检查 LHS，RHS 和 Result，若为变量，则需要调用 usageMap_Add 方法。

```
for (Instruction preprocessedInstruction:preprocessedInstructions) {
    InstructionKind kind = preprocessedInstruction.getKind();
    if(kind.isBinary()){
        //对于 BinaryOp(两个操作数的指令):
        IRValue LHS = preprocessedInstruction.getLHS();
        IRValue RHS = preprocessedInstruction.getRHS();
        IRValue result = preprocessedInstruction.getResult();
        //若LHS是变量
        if(LHS.isIRVariable()){
            usageMap_Add((IRVariable) LHS);
        }
        //若RHS是变量
        if(RHS.isIRVariable()){
            usageMap_Add((IRVariable) RHS);
        }
        //若result是变量
        if(result.isIRVariable()){
            usageMap_Add((IRVariable) result);
        }
    }
}
```

对于 UnaryOp(一个操作数的指令)，检查 Result 和 From，若为变量，则调用 usageMap_Add 方法。

```
}else if(kind.isUnary()){
    IRValue result = preprocessedInstruction.getResult();
    IRValue from = preprocessedInstruction.getFrom();

    //若result是变量
    if(result.isIRVariable()){
        usageMap_Add((IRVariable) result);
    }
    //若from是变量
    if(from.isIRVariable()){
        usageMap_Add((IRVariable) from);
    }
}
```

对于 Ret 指令，检查 returnValue，若为变量，则调用 usageMap_Add 方法。

```
}else if(kind.isReturn()){
    IRValue returnValue = preprocessedInstruction.getReturnValue();
    //若returnValue是变量
    if(returnValue.isIRVariable()){
        usageMap_Add((IRVariable) returnValue);
    }
}
```

②循环对每一条预处理后的指令分配寄存器并生成相应的汇编指令。

对于 BinaryOp(两个操作数的指令), 首先使用寄存器选择算法为 result 选择寄存器, 再检查 LHS, 当 LHS 为变量时, 使用寄存器分配算法为其分配寄存器; 当 LHS 为立即数时, 直接获取, RHS 同理。

```
for (Instruction preprocessedInstruction:preprocessedInstructions){
    InstructionKind kind = preprocessedInstruction.getKind();
    String riscvInstruction;

    if(kind.isBinary()){
        IRVariable result = preprocessedInstruction.getResult();
        IRValue LHS = preprocessedInstruction.getLHS();
        IRValue RHS = preprocessedInstruction.getRHS();

        String lhs = "";
        String rhs = "";
        String result = "";
        String op = "";

        //使用寄存器选择算法为result选择寄存器
        result = Reg_Allocate(Result);

        //当LHS为变量时, 使用寄存器分配算法为其分配寄存器; 当LHS为立即数时, 直接获取
        if(LHS.isIRVariable()){
            lhs = Reg_Allocate((IRVariable) LHS);
        }else if(LHS.isImmediate()){
            lhs = lhs + (((IRImmediate) LHS).getValue());
        }

        //RHS同理
        if(RHS.isIRVariable()){
            rhs = Reg_Allocate((IRVariable) RHS);
        }else if(RHS.isImmediate()){
            rhs = rhs + (((IRImmediate) RHS).getValue());
        }
    }
}
```

再根据指令类型生成相应的汇编指令。

```
//相应的操作op
if(RHS.isImmediate()){
    if(kind==InstructionKind.ADD){
        op="addi";
    }else if(kind==InstructionKind.MUL){
        op="mul";
    }
}
}else{
    if(kind==InstructionKind.ADD){
        op="add";
    }else if(kind==InstructionKind.MUL){
        op="mul";
    }else if(kind==InstructionKind.SUB){
        op="sub";
    }
}

//生成汇编指令
riscvInstruction= "    " + op + "    " + result + "    " + lhs + "    " + rhs;
riscvInstructions.add(riscvInstruction);
```

对于 UnaryOp(一个操作数的指令), 首先使用寄存器选择算法为 result 选择寄存器, 再检查 From, 当 From 为变量时, 使用寄存器分配算法为其分配寄存器; 当 From 为立即数时, 直接获取。

```

}else if(kind.isUnary()){
    IRVariable Result = preprocessedInstruction.getResult();
    IRValue From = preprocessedInstruction.getFrom();

    String from = "";
    String result = "";
    String op = " ";

    //使用寄存器选择算法为result选择寄存器
    result = Reg_Allocate(Result);

    //当From为变量时, 使用寄存器分配算法为其分配寄存器; 当From为立即数时, 直接获取
    if(From.isIRVariable()){
        from = Reg_Allocate((IRVariable) From);
    }else if(From.isImmediate()){
        from = from + ((IRImmediate) From).getValue();
    }
}

```

再根据指令类型生成相应的汇编指令。

```

//相应的操作op
if(From.isIRVariable()){
    op="mv";
}else if(From.isImmediate()){
    op="li";
}

//生成汇编指令
riscvInstruction = "    " + op + "    " + result + "    " + from ;
riscvInstructions.add(riscvInstruction);

```

对于 Ret 指令, 分配寄存器并生成汇编指令, 最终结果写入 a0。

```

}else if(kind.isReturn()){
    //return, 分配寄存器并生成汇编指令, 最终结果写入a0
    IRValue ReturnValue = preprocessedInstruction.getReturnValue();
    String returnValue = Reg_Allocate((IRVariable) ReturnValue);
    riscvInstruction = "    "+"mv"+"    "+"a0"+"    "+returnValue;
    riscvInstructions.add(riscvInstruction);
}

```

(5)dump 方法: 利用 FileUtils 输出汇编代码到文件。

```

public void dump(String path) {
    // TODO: 输出汇编代码到文件
    /**for (String riscvInstruction: riscvInstructions) {
        System.out.println(riscvInstruction);
    }*/
    FileUtils.writeLines(path, riscvInstructions.stream().toList());
    //throw new NotImplementedException();
}

```

4 实验结果与分析

4.1 词法分析

输入：以文件形式存放的类 C 语言程序段，即码点文件 `coding_map.csv` 和输入的代码 `input_code.txt`。

| coding_map.csv × | |
|------------------|-------------|
| 1 | 1 int |
| 2 | 2 return |
| 3 | 3 = |
| 4 | 4 , |
| 5 | 5 Semicolon |
| 6 | 6 + |
| 7 | 7 - |
| 8 | 8 * |
| 9 | 9 / |
| 10 | 10 (|
| 11 | 11) |
| 12 | 51 id |
| 13 | 52 IntConst |

```
input_code.txt ×
1  int result;
2  int a;
3  int b;
4  int c;
5  a = 8;
6  b = 5;
7  c = 3 - a;
8  result = a * b - ( 3 + b ) * ( c - a );
9  return result;
```

输出：以文件形式存放的 TOKEN 串和简单符号表，即符号表 `old_symbol_table.txt` 和词法单元列表 `token.txt`。

| old_symbol_table.txt × | |
|------------------------|----------------|
| 1 | (a, null) |
| 2 | (b, null) |
| 3 | (c, null) |
| 4 | (result, null) |
| 5 | |

```
token.txt x
1 (int,)
2 (id,result)
3 (Semicolon,)
4 (int,)
5 (id,a)
6 (Semicolon,)
7 (int,)
8 (id,b)
9 (Semicolon,)
10 (int,)
11 (id,c)
12 (Semicolon,)
13 (id,a)
14 (=,)
15 (IntConst,8)
16 (Semicolon,)
17 (id,b)
18 (=,)
19 (IntConst,5)
20 (Semicolon,)
21 (id,c)
22 (=,)
23 (IntConst,3)
24 (-,)
25 (id,a)
26 (Semicolon,)
27 (id,result)
28 (=,)
29 (id,a)
30 (*,)
31 (id,b)
32 (-,)
33 ((,)
34 (IntConst,3)
35 (+,)
36 (id,b)
37 (,),)
38 (*,)
39 ((,)
40 (id,c)
41 (-,)
42 (id,a)
43 (,),)
44 (Semicolon,)
45 (return,)
46 (id,result)
47 (Semicolon,)
48 ($,)
49
```

分析：比较标准输出与输出的 old_symbol_table.txt、token.txt，成功通过脚本检测，说明词法分析结果正确。

4.2 语法分析

输入：码点文件 coding_map.csv，语法文件 grammar.txt，输入代码 input_code.txt 和第三方工具生成的 LR 分析表 LR1_table.csv。

（其中 coding_map.csv 和 input_code.txt 与之前实验相同，这里不再截图。）

```

grammar.txt
1  P -> S_list;
2  S_list -> S Semicolon S_list;
3  S_list -> S Semicolon;
4  S -> D id;
5  D -> int;
6  S -> id = E;
7  S -> return E;
8  E -> E + A;
9  E -> E - A;
10 E -> A;
11 A -> A * B;
12 A -> B;
13 B -> ( E );
14 B -> id;
15 B -> IntConst;
16

```

```

LR1_table.csv
1  状态,ACTION,,,,,,,,GOTO,,,,
2  ,id,(,)+,-,*,=,int,return,IntConst,Semicolon,$,E,S_list,S,A,B,D
3  0,shift 4,,,,,,,,,shift 5,shift 6,,,,,1,2,,3
4  1,,,,,,,,,accept,,,,
5  2,,,,,,,,,shift 7,,,,
6  3,shift 8,,,,,,,,
7  4,,,,,,,,,shift 9,,,,
8  5,reduce D -> int,,,,,,,,
9  6,shift 13,shift 14,,,,,,,,,shift 15,,,10,,,11,12,
10 7,shift 4,,,,,,,,,shift 5,shift 6,,reduce S_list -> S Semicolon,,16,2,,3
11 8,,,,,,,,,reduce S -> D id,,,,
12 9,shift 13,shift 14,,,,,,,,,shift 15,,,17,,,11,12,
13 10,,,,,shift 18,shift 19,,,,,reduce S -> return E,,,,
14 11,,,,,reduce E -> A,reduce E -> A,shift 20,,,,,reduce E -> A,,,,
15 12,,,,,reduce A -> B,reduce A -> B,reduce A -> B,,,,,reduce A -> B,,,,
16 13,,,,,reduce B -> id,reduce B -> id,reduce B -> id,,,,,reduce B -> id,,,,
17 14,shift 24,shift 25,,,,,,shift 26,,,21,,,22,23,
18 15,,,,,reduce B -> IntConst,reduce B -> IntConst,reduce B -> IntConst,,,,,reduce B -> IntConst,,,,
19 16,,,,,,,,,reduce S_list -> S Semicolon S_list,,,,
20 17,,,,,shift 18,shift 19,,,,,reduce S -> id = E,,,,
21 18,shift 13,shift 14,,,,,,,,,shift 15,,,27,12,
22 19,shift 13,shift 14,,,,,,,,,shift 15,,,28,12,
23 20,shift 13,shift 14,,,,,,,,,shift 15,,,29,
24 21,,,,,shift 30,shift 31,shift 32,,,,
25 22,,,,,reduce E -> A,reduce E -> A,reduce E -> A,shift 33,,,,
26 23,,,,,reduce A -> B,reduce A -> B,reduce A -> B,reduce A -> B,,,,
27 24,,,,,reduce B -> id,reduce B -> id,reduce B -> id,reduce B -> id,,,,
28 25,shift 24,shift 25,,,,,,shift 26,,,34,,,22,23,
29 26,,,,,reduce B -> IntConst,reduce B -> IntConst,reduce B -> IntConst,reduce B -> IntConst,,,,
30 27,,,,,reduce E -> E + A,reduce E -> E + A,shift 20,,,,,reduce E -> E + A,,,,
31 28,,,,,reduce E -> E - A,reduce E -> E - A,shift 20,,,,,reduce E -> E - A,,,,
32 29,,,,,reduce A -> A * B,reduce A -> A * B,reduce A -> A * B,,,,,reduce A -> A * B,,,,
33 30,,,,,reduce B -> ( E ),reduce B -> ( E ),reduce B -> ( E ),,,,,,reduce B -> ( E ),,,,
34 31,shift 24,shift 25,,,,,,shift 26,,,35,23,
35 32,shift 24,shift 25,,,,,,shift 26,,,36,23,
36 33,shift 24,shift 25,,,,,,shift 26,,,37,
37 34,,,,,shift 30,shift 31,shift 32,,,,
38 35,,,,,reduce E -> E + A,reduce E -> E + A,reduce E -> E + A,shift 33,,,,
39 36,,,,,reduce E -> E - A,reduce E -> E - A,reduce E -> E - A,shift 33,,,,
40 37,,,,,reduce A -> A * B,reduce A -> A * B,reduce A -> A * B,reduce A -> A * B,,,,
41 38,,,,,reduce B -> ( E ),reduce B -> ( E ),reduce B -> ( E ),reduce B -> ( E ),,,,
42

```

输出：规约过程的产生式列表 parser_list.txt，语义分析前的符号表 old_symbol_table.txt 和词法单元流 token.txt。

（其中 old_symbol_table.txt 和 token.txt 与之前实验相同，这里不再截图。）

```
parser_list.txt x
1  D -> int
2  S -> D id
3  D -> int
4  S -> D id
5  D -> int
6  S -> D id
7  D -> int
8  S -> D id
9  B -> IntConst
10 A -> B
11 E -> A
12 S -> id = E
13 B -> IntConst
14 A -> B
15 E -> A
16 S -> id = E
17 B -> IntConst
18 A -> B
19 E -> A
20 B -> id
21 A -> B
22 E -> E - A
23 S -> id = E
24 B -> id
25 A -> B
26 B -> id
27 A -> A * B
28 E -> A
29 B -> IntConst
30 A -> B
31 E -> A
32 B -> id
33 A -> B
34 E -> E + A
35 B -> ( E )
36 A -> B
37 B -> id
38 A -> B
39 E -> A
40 B -> id
41 A -> B
42 E -> E - A
43 B -> ( E )
44 A -> A * B
45 E -> E - A
46 S -> id = E
47 B -> id
48 A -> B
49 E -> A
50 S -> return E
51 S_list -> S Semicolon
52 S_list -> S Semicolon S_list
53 S_list -> S Semicolon S_list
54 S_list -> S Semicolon S_list
55 S_list -> S Semicolon S_list
56 S_list -> S Semicolon S_list
57 S_list -> S Semicolon S_list
58 S_list -> S Semicolon S_list
59 S_list -> S Semicolon S_list
60 P -> S_list
61
```

分析：比较标准输出与输出的 parser_list.txt，成功通过脚本检测，说明语法分析结果正确。

4.3 语义分析和中间代码生成

输入：码点文件 coding_map.csv，语法文件 grammar.txt，输入代码 input_code.txt 和第三方工具生成的 IR 分析表 LR1_table.csv。

（其中 coding_map.csv，input_code.txt，grammar.txt 和 LR1_table.csv 与之前实验相同，这里不再截图。）

输出：中间表示 intermediate_code.txt，中间表示的模拟执行的结果 ir_emulate_result.txt，规约过程的产生式列表 parser_list.txt，语义分析后的符号表 new_symbol_table.txt，语义分析前的符号表 old_symbol_table.txt 和词法单元流 token.txt。

（其中 old_symbol_table.txt，token.txt，parser_list.txt 与之前实验相同，这里不再截图。）

| intermediate_code.txt | |
|-----------------------|----------------------|
| 1 | (MOV, a, 8) |
| 2 | (MOV, b, 5) |
| 3 | (SUB, \$0, 3, a) |
| 4 | (MOV, c, \$0) |
| 5 | (MUL, \$1, a, b) |
| 6 | (ADD, \$2, 3, b) |
| 7 | (SUB, \$3, c, a) |
| 8 | (MUL, \$4, \$2, \$3) |
| 9 | (SUB, \$5, \$1, \$4) |
| 10 | (MOV, result, \$5) |
| 11 | (RET, , result) |
| 12 | |

| ir_emulate_result.txt | |
|-----------------------|-----|
| 1 | 144 |
| 2 | |

| new_symbol_table.txt | |
|----------------------|---------------|
| 1 | (a, Int) |
| 2 | (b, Int) |
| 3 | (c, Int) |
| 4 | (result, Int) |
| 5 | |

分析：比较标准输出与输出的 ir_emulate_result.txt，结果均为 144；比较标准输出与输出的 new_symbol_table.txt，结果一致，说明语义分析和中间代码生成结果正确。

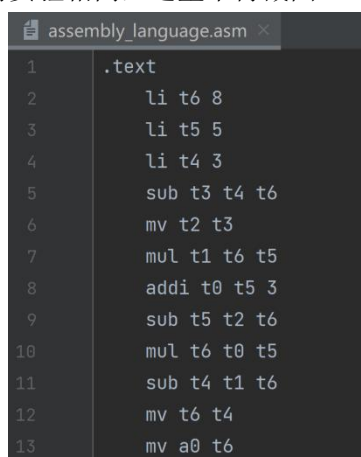
4.4 目标代码生成

输入：码点文件 coding_map.csv，语法文件 grammar.txt，输入代码 input_code.txt，(可选)第三方工具生成的 IR 分析表 LR1_table.csv。

(其中 coding_map.csv, input_code.txt, grammar.txt 和 LR1_table.csv 与之前实验相同，这里不再截图。)

输出：汇编代码 assembly_language.asm，中间表示 intermediate_code.txt，中间表示的模拟执行的结果 ir_emulate_result.txt，规约过程的产生式列表 parser_list.txt，语义分析后的符号表 new_symbol_table.txt，语义分析前的符号表 old_symbol_table.txt 和词法单元流 token.txt。

(其中 old_symbol_table.txt, token.txt, parser_list.txt, intermediate_code.txt, ir_emulate_result.txt 和 new_symbol_table.txt 与之前实验相同，这里不再截图。)



```
1 .text
2     li t6 8
3     li t5 5
4     li t4 3
5     sub t3 t4 t6
6     mv t2 t3
7     mul t1 t6 t5
8     addi t0 t5 3
9     sub t5 t2 t6
10    mul t6 t0 t5
11    sub t4 t1 t6
12    mv t6 t4
13    mv a0 t6
```

| | | |
|----|----|--------------------|
| a0 | 10 | 0x0000000000000090 |
|----|----|--------------------|

分析：在 rars 上执行生成的目标代码，执行结束后，寄存器 a0 存放的值为 0x90，即十进制的 144，即为正确执行的结果，说明生成的目标代码正确。

5 实验中遇到的困难与解决办法

在四次实验中，我遇到的困难主要有两个方面：一是对于实验框架接口代码不够熟悉，导致需要使用时可能想不到或者使用有困难；二是对于实验需要的算法不够熟悉，导致实现时思路混乱。

对于以上两个方面，我在实验的过程中逐渐探索了相应的解决办法：对于实验框架接口，阅读在线网站的指导和解释有助于理解和运用；对于实验需要实现的算法，仔细阅读和理解实验给出的 ppt 和指导书，其中其实已经说明了很多具体的思路，按照其思路来编写代码会更清晰。

对于本实验，我的建议是可以对框架接口的使用多举一些具体例子来帮助理解。

通过实现四次实验，我有很大的收获。每次实验实现编译器的不同部分，在实践中加深了我对编译器词法分析、语法分析、语义分析和中间代码生成、目标代码生成的理解；体会到了一步步实现一个编译器的过程，其中不断遇到问题并且想办法解决，增强了代码能力。