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: 李春阳

学　号: 210110812

专　业: 计算机科学与技术

日　期: 2023-10-31

# 1 实验目的与方法

**实验目的：**利用 Java 实现 TXTv2 语言编译器, 包括词法分析器、语法分析器、语义分析与中间代码成器以及目标代码生成器四个组成部分，每次实验均在上一次实验的基础上进行扩展和完善。目标平台为 RISC-V32 （指令集 RV32M）。

**实验方法：**使用的编程语言为 Java，软件环境为 Java17，开发工具为 IntelliJ IDEA。此外，还使用了 RARS 及编译工作台。实验方法后续将不再赘述。

## 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。

## 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。

## 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

## 2.1 词法分析器

**1. 实验内容：**

编写一个词法分析程序，读取文件，对文件内自定义的类 C 语言程序段进行词法分析。

**2. 实验要求：**

1）输入：以文件形式存放自定义的类 C 语言程序段;

2）输出：以文件形式存放的 TOKEN 串和简单符号表;

3）要求：输入的 C 语言程序段包含常见的关键字，标识符，常数，运算符和分界符等

## 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

**1. 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。

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。

**2. 实验要求：**

使用框架中的模拟器 IREmulator 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，本次实验中单词种类可分为 5 类：关键字、运算符、分界符、标识符、常数。其中关键字、运算符、分界符都是程序设计语言预先定义的，数量固定；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，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），本次实验所用到的编码表如下：

| 单词名称 | 类别编码 | 单词值（若无单词值，则为“-”） |
|---|---|---|
| int | 1 | - |
| return | 2 | - |
| = | 3 | - |
| , | 4 | - |
| Semicolon | 5 | - |
| + | 6 | - |
| - | 7 | - |
| * | 8 | - |
| / | 9 | - |
| ( | 10 | - |
| ) | 11 | - |
| id | 51 | 内部字符串 |
| IntConst | 52 | 整数值 |

上述编码表对应着 data / in / coding_map.csv 文件中的内容。

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

### 3.1.2 正则文法

多数程序语言单词的词法都能用正则文法来描述，基于单词的这种形式化描述会给词法分析器的设计与实现带来很大的方便。本实验中正则文法表示如下：

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”

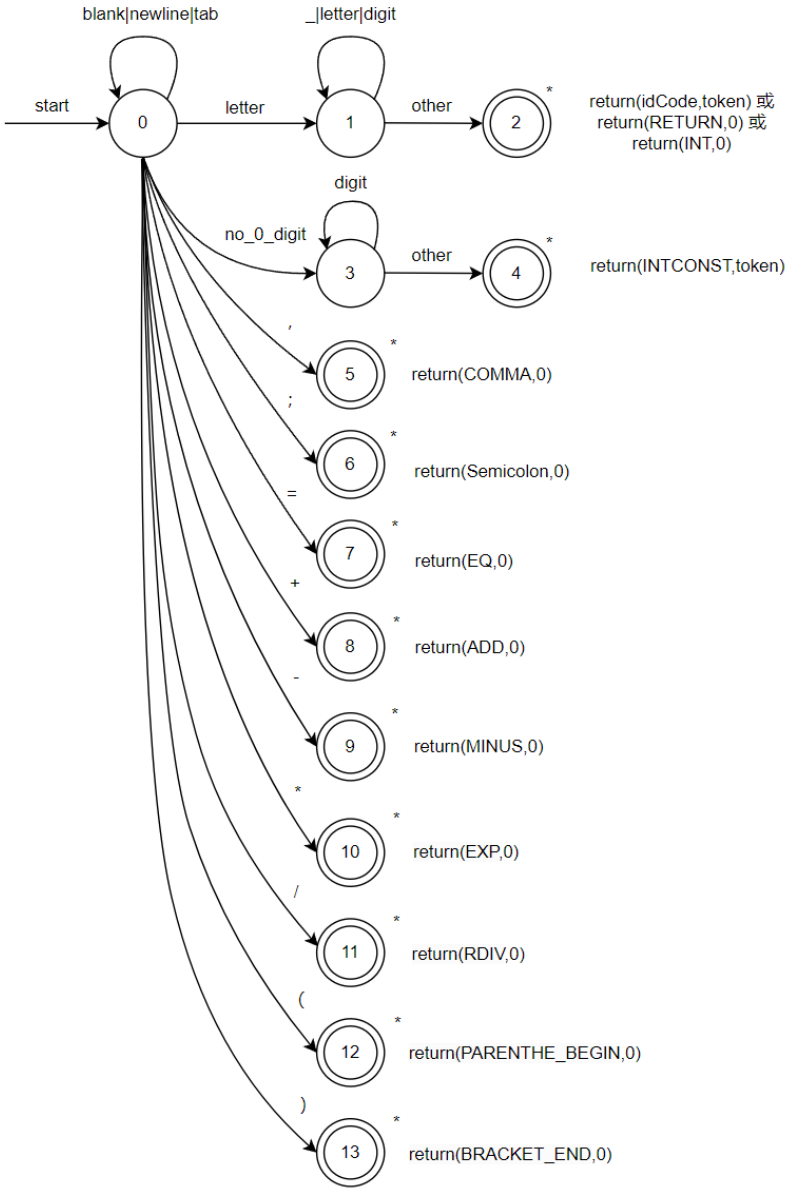
字符常量：S →’D’

### 3.1.3 状态转换图

具体要实现的词法规则如下:

| 单词名称 | 正则表达式 |
|---|---|
| int | int |
| return | return |
| = | = |
| , | , |
| Semicolon | ; |
| + | + |
| - | - |
| * | * |
| / | / |
| ( | ( |
| ) | ) |
| id | [a-zA-Z_][a-zA-Z]* |
| IntConst | [0-9]+ |

根据词法规则设计状态转移图如下图所示：




### 3.1.4 词法分析程序设计思路和算法描述

1） **修改 SymbolTable.java 文件**，即完善与符号表相关的数据结构及操作方法。在 SymbolTable.java 文件中，get、add、has、getAllEntries 函数分别用于从符号表中获得条目、向符号表中添加条目、判断符号表中是否存在某条目、获取符号表的所有条目。为实现各函数功能，首先需要建立一个将字符串 String 与符号表项 SymbolTableEntry 一一对应的 Map，便于根据给定的字符串查找对应的标识符。

```
public Map<String, SymbolTableEntry> symbolTable = new HashMap<>();
```

利用 HashMap 中的 containsKey 方法实现 has 方法：

```java
public boolean has(String text) {
    return symbolTable.containsKey(text);
    //throw new NotImplementedException();
}
```

对于 add 方法，首先需要检测符号表中是否已含有该标识符，若无则向符号表添加该标识符；对于 get 方法，也需要检测符号表中是否已含有该标识符，若有则返回该符号在符号表中所处的条目。

```java
public SymbolTableEntry get(String text) {
    if(has(text)){
        return symbolTable.get(text);
    }else{
        throw new NotImplementedException();
    }
}
```

```java
public SymbolTableEntry add(String text) {
    if(!has(text)){
        return symbolTable.put(text,new SymbolTableEntry(text));
    }else{
        throw new NotImplementedException();
    }
}
```

对于 getAllEntries 函数，直接返回该 Map 即可。

```java
private Map<String, SymbolTableEntry> getAllEntries() {
    return symbolTable;
    //throw new NotImplementedException();
}
```

2） **修改 LexicalAnalyzer.java 文件**。将 data / in / input_code.txt 中的内容读入并加载到缓冲区中，并以字符串的形式储存道 sourceString 中。

```java
public void loadFile(String path) {
    // 词法分析前的缓冲区实现
    try (BufferedReader br = new BufferedReader(new FileReader(path))){
        String line;
        while ((line = br.readLine()) != null) {
            sourceString += line;
        }
        //System.out.println(sourceString);
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
}
```

接着设置词法单元列表，用于储存从文件中读出的词法单元。

```java
public List<Token> tokens = new ArrayList<>();
```

接着，利用 toCharArray 方法将 sourceString 中的内容转换为字符数组，并进行遍历。为了结构更清晰，我们将读入的字符进行分为特殊字符、单个字符（包括运算符及分界符等）、关键字及标识符和数字四类。然后根据分类进行各自的判断或接收操作。

```java
public void run() {
    //自动机实现的词法分析过程
    String status;
    int i = 0;
    char ch;
    char[] word = sourceString.toCharArray();
    while(i < sourceString.length()) {
        ch = word[i];

        //为了结构更清晰，我们将读入的字符进行分类
        if (ch == ' ' || ch == '\r' || ch == '\n' || ch == '\t') {
            //特殊字符
            status = "SKIP";
        } else if (ch == ',' || ch == ';' || ch == '=' || ch == '+' || ch == '-' || ch == '*' || ch == '/' || ch == '(' || ch == ')') {
            //仅判断一个字符便可以得出结果的情况
            status = "SINGLE-PUNCTUATION";
        } else if (Character.isLetter(ch)) {
            //关键字和标识符
            status = "LETTER";
        } else if (Character.isDigit(ch)) {
            //数字
            status = "DIGIT";
        } else {
            status = "ERROR";
        }
```

如果遇到空白字符（空格、制表符、换行符、回车符等），则直接忽略。如果遇到“=”、“,”、“;”、“+”、“-”、“*”、“/”、“(”、“)”等字符，则直接将其储存至词法单元列表。

```java
switch (status) {
    case "SKIP" -> ++i;
    case "SINGLE-PUNCTUATION" -> {
        switch (ch) {
            case ',' -> tokens.add(Token.simple( tokenKindId: ","));
            case ';' -> tokens.add(Token.simple( tokenKindId: "Semicolon"));
            case '=' -> tokens.add(Token.simple( tokenKindId: "="));
            case '+' -> tokens.add(Token.simple( tokenKindId: "+"));
            case '-' -> tokens.add(Token.simple( tokenKindId: "-"));
            case '*' -> tokens.add(Token.simple( tokenKindId: "*"));
            case '/' -> tokens.add(Token.simple( tokenKindId: "/"));
            case '(' -> tokens.add(Token.simple( tokenKindId: "("));
            case ')' -> tokens.add(Token.simple( tokenKindId: ")"));
            default -> {
            }
        }
        i++;
    }
```

如果遇到数字字符，则一直读取直到遇到其它种类的字符。该行为可以获取到一个单词 word，显然这就是识别出来的 int 型常量，直接将其类别“IntConst”及值储存至词法单元列表即可。

```java
case "DIGIT" -> {
    int index = i;
    while (Character.isDigit(ch) && (i+1 < sourceString.length())) {
        ch = word[++i];
    }
    String digit = sourceString.substring(index, i);
    tokens.add(Token.normal( tokenKindId: "IntConst", digit));
}
```

如果遇到字母，则一直读取直到遇到其它种类的字符。该行为可以获取到一个单词 word，该单词可能为关键字，也可能为标识符。通过 TokenKind.isAllowed(key)对该读取到的 word 进行判断，如果 word 为关键字（int/return），则直接对该关键字进行接收；如果 word 是其它字符串（即标识符），则对该标识符进行接收，并将其添加到符号表中。

```
case "LETTER" -> {
    int index = i;
    while (Character.isLetter(ch) && (i+1 < sourceString.length())) {
        ch = word[++i];
    }
    String key = sourceString.substring(index, i);

    //当前读入字符串为标识符
    if (TokenKind.isAllowed(key)) {
        tokens.add(Token.simple(key));
    } else {
        //当前读入字符串为关键字
        tokens.add(Token.normal( tokenKindId: "id",key));
        if (!symbolTable.has(key)) {
            symbolTable.add(key);
        }
    }
}
```

当字符全部读取完毕后，向 tokens 列表中添加标志 end of file 的词法单元 eof，结束词法分析程序的执行。

```
//插入结束符
tokens.add(Token.eof());
```

通过 getTokens 方法返回词法单元列表。

```
public Iterable<Token> getTokens() {
    // 从词法分析过程中获取 Token 列表
    return tokens;
}
```

## 3.2 语法分析

### 3.2.1 拓展文法

对于某些文法，存在一些右部含有文法开始符号的产生式，在归约过程中需要分清楚是否已经归约到文法的最初开始符。因此，需要对原有文法进行拓广。本次实验的拓广文法框架中已提供，并储存在 data / in / grammar.txt 文件中。

```
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;
```

### 3.2.2 LR1 分析表

根据已有的拓广文法，使用第三方工具“编译工作台”可以构造出 LR1 分析表，从而进行语法分析。LR 分析表实际上就是一张类似自动机的状态转移图，当在当前状态遇到某某符号时便执行动作并转移。归约时根据“Action 表”里写的产生式生成非终结符，然后将生成的非终结符作为“输入符号”根据“Goto 表”来确定转移；移入时直接根据“Action 表”转移到对应状态。

| 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 Se | | 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 - | reduce E - | shift 20 | | | | | reduce E -> A | | | | | | | |
| 12 | | | | reduce A - | reduce A - | reduce A -> B | | | | | reduce A -> B | | | | | | | |
| 13 | | | | reduce B - | reduce B - | reduce B -> id | | | | | reduce B -> id | | | | | | | |
| 14 | shift 24 | shift 25 | | | | | | | | shift 26 | | | 21 | | | 22 | 23 | |
| 15 | | | | reduce B - | reduce B - | 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 - | reduce E - | reduce E - | shift 33 | | | | | | | | | | | | |
| 23 | | | reduce A - | reduce A - | reduce A - | reduce A -> B | | | | | | | | | | | | |
| 24 | | | reduce B - | reduce B - | reduce B - | reduce B -> id | | | | | | | | | | | | |
| 25 | shift 24 | shift 25 | | | | | | | | shift 26 | | | 34 | | | 22 | 23 | |
| 26 | | | reduce B - | reduce B - | reduce B - | reduce B -> IntConst | | | | | | | | | | | | |
| 27 | | | | reduce E - | reduce E - | shift 20 | | | | | reduce E -> E + A | | | | | | | |
| 28 | | | | reduce E - | reduce E - | shift 20 | | | | | reduce E -> E - A | | | | | | | |
| 29 | | | | reduce A - | reduce A - | reduce A -> A * B | | | | | reduce A -> A * B | | | | | | | |
| 30 | | | | reduce B - | reduce B - | 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 - | reduce E - | reduce E - | shift 33 | | | | | | | | | | | | |
| 36 | | | reduce E - | reduce E - | reduce E - | shift 33 | | | | | | | | | | | | |
| 37 | | | reduce A - | reduce A - | reduce A - | reduce A -> A * B | | | | | | | | | | | | |
| 38 | | | reduce B - | reduce B - | reduce B - | reduce B -> ( E ) | | | | | | | | | | | | |

### 3.2.3 状态栈和符号栈的数据结构和设计思路

#### 1） 状态栈的数据结构

由于状态 Status 已定义成单独的类，故状态栈的数据结构如下：

```
private final Stack<Status> statusStack = new Stack<>();    //状态栈
```

#### 2） 符号栈的数据结构

在处理符号栈时，我们可能需要同时将终结符 Token 和非终极符 NonTerminal 装在栈中，但是 Token 和 NonTerminal 除了 Object 之外没有共同祖先类，因此我们可以自定义一个 Symbol 类来完成这个共用体的功能，这样就能把 Token 和 NonTerminal 同时装在栈中，实现符号栈的数据结构。

自定义 Symbol 类的代码如下：

```java
public class Symbol{
    3 个用法
    Token token;
    2 个用法
    NonTerminal nonTerminal;
    6 个用法
    SourceCodeType type = null;
    22 个用法
    IRValue value = null;

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

    4 个用法
    public Symbol(Token token) { this(token, nonTerminal: null); }
    6 个用法
    public Symbol(NonTerminal nonTerminal) { this( token: null, nonTerminal); }
    public boolean isToken() { return this.token != null; }
    public boolean isNonterminal() { return this.nonTerminal != null; }
}
```

符号栈的数据结构如下：

```java
private final Stack<Symbol> tokenStack = new Stack<>();    //符号栈
```

### 3.2.4 LR 驱动程序设计思路和算法描述

1） 加载词法分析程序分析得到的词法单元流、拓广文法对应的产生式列表及 LR(1)分析表，并用对应的数据结构储存。

```java
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) {
        tokenList.add(token);
    }
}
```

```java
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();
    this.lrtable=table;
}
```

**2） 填写语法分析过程 run 方法：**

首先，初始化符号栈及状态栈。对于状态栈，用 LR(1)分析表中的 getInit 方法获取起始状态，并压入 statusStack 中；对于符号栈，将(new Symbol(Token.eof()))压入 symbolStack 中（相当于压入“#”）。然后反复执行下列操作，直至达到 Accept 接收状态。

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

获取当前状态栈及符号栈的栈顶元素 currentStatus 和 currentToken。调用加载的 LR(1)表的 getAction 方法，根据 currentStatus、currentToken 确定接下来要执行的动作：

- 如果是移入动作（Shift）：调用 callWhenInShift 方法通知各个观察者，并调用当前 Action 的 getStatus 方法获取下一步转换到的状态、并将该状态压入状态栈，同时对应的 token 压入符号栈。
- 如果是归约动作（Reduce）：调用 callWhenInReduce 方法通知各个观察者，同时调用当前 Action 的 getProduction 方法获取归约所用的产生式，然后根据产生式长度，使符号栈和状态栈均弹出对应长度个符号。最后把产生式左侧的非终结符压入符号栈，并根据符号栈和状态栈栈顶状态获取 Goto 表的状态（调用 getGoto 方法），压入状态栈。
- 如果是接受动作（Accept）：调用 callWhenInAccept 方法通知各个观察者，同时将标志位 flag 的值置为 false，语法分析执行结束。
- 如果是出错动作（Error）：抛出异常，并提示语法分析出错，同时结束语法分析过程。

```
108         int i=0;
109         boolean flag = true;
110         do{
111             Token currentToken = tokenList.get(i);
112             Status currentStatus = statusStack.peek();
113             Action currentAction = lrtable.getAction(currentStatus, currentToken);
114
115             switch (currentAction.getKind()) {
116                 case Shift: {
117                     callWhenInShift(currentStatus, currentToken);
118                     statusStack.push(currentAction.getStatus());
119                     tokenStack.push(new Symbol(currentToken));
120                     i++;
121                     break;
122                 }
123                 case Reduce: {
124                     Production currentProduction = currentAction.getProduction();
125                     callWhenInReduce(currentStatus, currentProduction);
126                     for (int j=0; j<currentProduction.body().size(); j++) {
127                         statusStack.pop();
128                         tokenStack.pop();
129                     }
130                     statusStack.push(lrtable.getGoto(statusStack.peek(),currentProduction.head()));
131                     tokenStack.push(new Symbol(currentProduction.head()));
132                     break;
133                 }
134                 case Error:{
135                     currentAction.error();
136                     System.out.println("error!");
137                     flag = false;
138                     break;
```

```
                case Accept: {
                    callWhenInAccept(currentStatus);
                    flag = false;
                    break;
                }
                default: flag = false;
            }
        }while(flag);
    }
}
```

反复执行以上步骤，当达到接受状态时，ProductionCollector 观察者内部顺序记录归约所用到的产生式，语法分析结束输出到文件。

## 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。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}

注：lookup(name)：查询符号表，返回 name 对应的记录
gencode(code)：生成三地址指令 code
enter:将变量的类型填入符号表

### 3.3.2 语义分析和中间代码生成的数据结构

1）设置新的类用于存放**终结符和非终结符的集合**（部分同实验二），并在其中新增“SourceCodeType type”和“IRValue value”两个属性，分别记录类型及取值两个综合属性，并初始化为 null。具体定义如下：

```java
public class Symbol{
    3 个用法
    Token token;
    2 个用法
    NonTerminal nonTerminal;
    6 个用法
    SourceCodeType type = null;
    22 个用法
    IRValue value = null;

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

    4 个用法
    public Symbol(Token token) { this(token, nonTerminal: null); }
    6 个用法
    public Symbol(NonTerminal nonTerminal) { this( token: null, nonTerminal); }
    public boolean isToken() { return this.token != null; }
    public boolean isNonterminal() { return this.nonTerminal != null; }
}
```

2）语义分析栈：分别在 IRGenerator.java 和 SemanticAnalyzer.java 文件中定义语义分析栈的数据结构，采用 Java 中的 Stack 结构实现；并在 IRGenerator.java 文件中额外定义中间代码指令列表，采用 Java 中的 List 结构实现。

```java
// TODO: 实验三: 实现语义分析
2 个用法
public class SemanticAnalyzer implements ActionObserver {

    public SymbolTable table;
    8 个用法
    private final Stack<Symbol> tokenStack = new Stack<>();
```

```java
// TODO: 实验三: 实现 IR 生成
2 个用法
public class IRGenerator implements ActionObserver {

    public SymbolTable table;
    30 个用法
    private final Stack<Symbol> tokenStack = new Stack<>();
    6 个用法
    private List<Instruction> IRList = new ArrayList<>();
```

3）其余涉及到的数据结构：

- 语法分析类 SynatxAnalyzer
- 符号表类 SymbolTable
- IRValue、IRVariable、IRImmediate 类
- Instruction 类：中间表示的指令
- Production 类：产生式
- Term、NonTerminal、TokenKind 类：各类文法符号

### 3.3.3 语法分析程序设计思路和算法描述

#### 1. 语义分析观察者的实现

观察者 SemanticAnalyzer 已经在语法分析阶段被注册到观察者列表 observers 当中，因此会在语法分析程序执行移入、归约、接收等动作时，接受到通知以及相关的信息。

在接收动作（whenAccept）时，语义分析程序不需要执行任何动作，因为遇到 Accept 时语法分析已经可以结束。

在移入过程（whenShift）中，由于源语言中只有 int 变量，因此通过 Token 中的 getKindType()方法判断 token 的 type 属性是否为 int 类型，若是，则将该符号的 type 字段定义为 SourceCodeType.Int，否则设为 null，并将符号加入语义分析栈，以便归约时能再次弹出并获得相关信息。

```
56      @Override
57      public void whenShift(Status currentStatus, Token currentToken) {
58          // TODO: 该过程在遇到 shift 时要采取的代码动作
59          Symbol curSymbol = new Symbol(currentToken);
60
61          if(Objects.equals(currentToken.getKindId(), b: "int")){
62              curSymbol.type = SourceCodeType.Int;
63          }else{
64              curSymbol.type = null;
65          }
66
67          tokenStack.push(curSymbol);
68      }
```

在归约过程（whenReduce）中，仅需处理的产生式列表中的 4、5 两条内容（仅这两条涉及更新符号表）。

- 对于第 4 条 S→D id 产生式，分别取出 id 和 D 两个符号，并将 id 的 type 更新为 D 的 type，更新符号表中相应变量的 type 信息，最后压入空记录占位（D id 被归约为 S，S 不需要携带信息）。
- 对于第 5 条 D→int 产生式，把 int 这个 token 的 type 类型赋值给 D 的 type，并将非终结符 D 压入语义分析栈。
- 对于其他产生式，由于不涉及到更新符号表，则仅将产生式右部弹出、左部压入语义分析栈即可。在 Symbol 类中 type 字段已初始化为 null，故无需额外处理。

```java
public void whenReduce(Status currentStatus, Production production) {
    // TODO: 该过程在遇到 reduce production 时要采取的代码动作
    Symbol curToken1, curToken2;
    Symbol curNonTerminal;
    switch(production.index()){
        case 4:     //S -> D id;
            curToken1 = tokenStack.pop();   //弹出id
            curToken2 = tokenStack.pop();   //弹出D
            // 将符号表中id的type更新为D的type
            this.table.get(curToken1.token.getText()).setType(curToken2.type);
            curNonTerminal = new Symbol(production.head());
            curNonTerminal.type = null;
            tokenStack.push(curNonTerminal);
            break;
        case 5:     //D -> int;
            curToken1 = tokenStack.pop();
            curNonTerminal = new Symbol(production.head());
            curNonTerminal.type = curToken1.type;
            tokenStack.push(curNonTerminal);
            break;
        default:
            for(int i=0; i<production.body().size(); i++){
                tokenStack.pop();
            }
            tokenStack.push(new Symbol(production.head()));
    }
}
```

## 2. 中间代码生成观察者的实现

观察者 IRGenerator 已经在语法分析阶段被注册到观察者列表 observers 当中，因此会在语法分析程序执行移进、归约、接收等动作时，接受到通知以及相关的信息。在 IRGenerator 里需要实现的任务是：根据翻译方案，利用规约到的产生式编写不同的语义动作，并在语义翻译的过程中生成中间代码。

在接收动作（whenAccept）中，语义分析程序不需要执行任何动作，因为遇到 Accept 时语法分析已经可以结束。

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

在移入过程（whenShift）中，由于某符号的 IRValue 属性分为立即数（IRImmediate）和变量（IRVariable）两种，因此在移入时通过正则表达式判断当前读入的符号为 IRImmediate 还是 IRVariable，并将其赋值给该符号的 value 属性，然后将符号加入语义分析栈，以便归约时能再次弹出并获得相关信息。

```
public void whenShift(Status currentStatus, Token currentToken) {
    // TODO
    String number = "^[0-9]+$";
    Symbol curSymbol = new Symbol(currentToken);
    if(currentToken.getText().matches(number)){
        curSymbol.value = IRImmediate.of(Integer.parseInt(currentToken.getText()));
    }else{
        curSymbol.value = IRVariable.named(currentToken.getText());
    }
    tokenStack.push(curSymbol);
}
```

在归约过程（whenReduce）中，通过 Production 的 index 方法获取当前归约的产生式的索引，并根据索引判断归约使用的是哪条产生式，针对不同的产生式编写不同的语义动作如下：

- 对于第 6 条产生式“S → id = E”：将产生式右部弹栈，分别获得 id 和 E 对应的 IRValue 属性信息，然后调用 Instruction 类的 createMov 方法生成相应的中间代码指令，并将其添加到中间代码指令列表 IRList 中。最后，向符号栈 tokenStack 中压入产生式左部的非终结符。
- 对于第 7 条产生式“S → return E”：将产生式右部弹栈，获得 E 对应的 IRValue 属性信息，然后调用 Instruction 类的 createRet 方法生成相应的中间代码指令，并将其添加到中间代码指令列表 IRList 中。最后，向符号栈 tokenStack 中压入产生式左部的非终结符。
- 对于第 8、9、11 条产生式“E→E + A”、“E→E - A”、“A→A * B”：将产生式右部弹栈，分别获得产生式右部符号（E、A 或者 A、B）对应的 IRValue 信息，然后调用 IRVariable 的 temp 方法生成临时变量以存放运算结果。调用 Instruction 类的 createAdd、createSub、createMul 方法生成相应的中间代码指令，并将其添加到中间代码指令列表 IRList 中。最后，将产生式左部的非终结符的 value 属性赋值为 IRVariable.temp()返回的值，并压入符号栈 tokenStack 中。
- 对于第 10、12、13、14、15 条产生式“E → A”、“A → B”、“B → ( E )”、“B → id”、“B → IntConst”：将产生式右部弹栈，获取右部符号对应的 IRValue 信息赋给产生式左部的非终结符的 value 属性，并将产生式左部的非终结符压入符号栈 tokenStack 中。
- 中间代码生成程序只需要完成简单赋值语句、算术运算语句的翻译，因此对于其它类型的产生式，直接将产生式右部全部弹出并压入产生式左部的非终结符即可。

```java
public void whenReduce(Status currentStatus, Production production) {
    // TODO
    Symbol lhs, rhs;
    Symbol curNonTerminal = new Symbol(production.head());
    IRVariable valueTemp;
    switch(production.index()){
        case 6:     //S -> id = E;
            rhs = tokenStack.pop();
            tokenStack.pop();
            lhs = tokenStack.pop();
            valueTemp = (IRVariable) lhs.value;
            curNonTerminal.value = null;
            IRList.add(Instruction.createMov(valueTemp, rhs.value));
            tokenStack.push(curNonTerminal);
            break;
        case 7:     //S -> return E;
            rhs = tokenStack.pop();
            tokenStack.pop();
            curNonTerminal.value = null;
            IRList.add(Instruction.createRet(rhs.value));
            tokenStack.push(curNonTerminal);
            break;
```

```java
case 8:     //E -> E + A;
    rhs = tokenStack.pop();
    tokenStack.pop();
    lhs = tokenStack.pop();
    valueTemp = IRVariable.temp();  //生成临时变量
    IRList.add(Instruction.createAdd(valueTemp, lhs.value, rhs.value));
    curNonTerminal.value = valueTemp;
    tokenStack.push(curNonTerminal);
    break;
case 9:     //E -> E - A;
    rhs = tokenStack.pop();
    tokenStack.pop();
    lhs = tokenStack.pop();
    valueTemp = IRVariable.temp();  //生成临时变量
    IRList.add(Instruction.createSub(valueTemp, lhs.value, rhs.value));
    curNonTerminal.value = valueTemp;
    tokenStack.push(curNonTerminal);
    break;
case 10:    //E -> A;
case 12:    //A -> B;
case 14:    //B -> id;
    curNonTerminal.value = tokenStack.pop().value;
    tokenStack.push(curNonTerminal);
    break;
case 11:    //A -> A * B;
    rhs = tokenStack.pop();
    tokenStack.pop();
    lhs = tokenStack.pop();
    valueTemp = IRVariable.temp();  //生成临时变量
    IRList.add(Instruction.createMul(valueTemp, lhs.value, rhs.value));
    curNonTerminal.value = valueTemp;
    tokenStack.push(curNonTerminal);
    break;
```

```
case 13:    //B -> ( E );
    tokenStack.pop();
    rhs = tokenStack.pop();
    tokenStack.pop();
    curNonTerminal.value = rhs.value;
    tokenStack.push(curNonTerminal);
    break;
case 15:    //B -> IntConst;
    rhs = tokenStack.pop();
    curNonTerminal.value = rhs.value;
    tokenStack.push(curNonTerminal);
    break;
default:
    for(int i=0; i<production.body().size(); i++){
        tokenStack.pop();
    }
    tokenStack.push(new Symbol(production.head()));
```

在 setSymbolTable 函数中，通过赋值操作将符号表 table 读取进来并储存。在 getIR 函数中，直接返回中间代码指令列表 IRList 即可。

```
public void setSymbolTable(SymbolTable table) {
    // TODO
    // throw new NotImplementedException();
    this.table = table;
}

2 个用法
public List<Instruction> getIR() {
    // TODO
    // throw new NotImplementedException();
    return IRList;
}
```

## 3.4 目标代码生成

### 3.4.1 设计思路和算法描述

1）**数据结构定义**：由于需要维护寄存器的占用信息，实现“给定寄存器号查找里面存储的变量”和“给定变量查找该变量存在的寄存器号”双向查找，因此我们定义一个双射的表结构用于定义寄存器分配表，代码如下：

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

    1 个用法
    public void removeByKey(K key) { VKmap.remove(KVmap.remove(key)); }

    1 个用法
    public void removeByValue(V value) {
        KVmap.remove(VKmap.remove(value));

    }

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

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

    2 个用法
    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);
    }

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

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

设置寄存器号为枚举变量，且不包括存放返回值的寄存器 a0。

```java
enum Register {
    t0,t1,t2,t3,t4,t5,t6
}
```

2） **预处理：**

对于立即数加减法，由于 IR 中并没有“只能有最多一个立即数”且“将立即数放在右手处”这个限制，将会导致在生成目标代码时带来繁琐的判断和指令插入。因此在生成汇编代码前我们需要对中间代码进行预处理，从而简化后续汇编指令生成过程。

首先，定义存放经过预处理后的中间指令列表 instructions。

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

然后，读入前端提供的中间代码。对于一元指令，无需进行预处理，直接存放至 instructions 列表中即可。如果这个一元指令是 Ret 指令，存入后可直接舍弃后续指令。

```java
public void loadIR(List<Instruction> originInstructions) {
    // TODO: 读入前端提供的中间代码并生成所需要的信息
    for (Instruction instruction : originInstructions){
        InstructionKind instructionKind = instruction.getKind();
        // 如果是RET指令
        if (instructionKind.isReturn()){
            // 遇到Ret指令后直接舍弃后续指令
            instructions.add(instruction);
            break;
        }
        //如果是一个操作数的指令
        if(instructionKind.isUnary()){
            instructions.add(instruction);
        }
```

对于二元指令，如果两个操作数都是立即数，则通过 instruction.getLHS()和 instruction.getRHS()方法获取两个操作数的值，并直接进行求值得到结果，然后替换成 MOV 指令。

```java
//如果是两个操作数的指令
else if(instructionKind.isBinary()){
    IRValue lhs = instruction.getLHS();
    IRValue rhs = instruction.getRHS();
    IRVariable result = instruction.getResult();
    // 如果两个操作数都是立即数：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 (lhs.isImmediate() && rhs.isImmediate()){
        int immediateResult = 0;
        switch (instructionKind){
            case ADD -> immediateResult = ((IRImmediate)lhs).getValue() + ((IRImmediate)rhs).getValue();
            case SUB -> immediateResult = ((IRImmediate)lhs).getValue() - ((IRImmediate)rhs).getValue();
            case MUL -> immediateResult = ((IRImmediate)lhs).getValue() * ((IRImmediate)rhs).getValue();
            default -> System.out.println("error");
        }
        instructions.add(Instruction.createMov(result, IRImmediate.of(immediateResult)));
    }
```

如果左操作数是立即数、右操作数是变量，且运算类型是减法或者乘法，需要调用 IRVariable 的 temp 方法生成一个临时变量 temp，并前插一条 MOV 指令，将立即数的值赋给临时变量，然后用临时变量替换原立即数，从而将指令调整为无立即数指令。对于右操作数是立即数、左操作数是变量，且运算类型是乘法的情况，也需要使用同样的方法进行调整，具体不再赘述。

对于不存在立即数的指令，则正常读入并直接存放至 instructions 列表中即可。

```
// 左立即数修改指令
else if (lhs.isImmediate() && rhs.isIRVariable()){
    switch (instructionKind){
        // 加法交换左立即数至右边
        case ADD -> instructions.add(Instruction.createAdd(result, rhs, lhs));
        // 减法与乘法指令前添加 MOV temp imm
        case SUB -> {
            IRVariable temp = IRVariable.temp();
            instructions.add(Instruction.createMov(temp, lhs));
            instructions.add(Instruction.createSub(result, temp, rhs));
        }
        case MUL -> {
            IRVariable temp = IRVariable.temp();
            instructions.add(Instruction.createMov(temp, lhs));
            instructions.add(Instruction.createMul(result, temp, rhs));
        }
        default -> System.out.println("error");
    }
}
// 右立即数修改指令
else if (lhs.isIRVariable() && rhs.isImmediate()){
    switch (instructionKind){
        case ADD, SUB -> instructions.add(instruction);
        case MUL -> {
            IRVariable temp = IRVariable.temp();
            instructions.add(Instruction.createMov(temp, rhs));
            instructions.add(Instruction.createMul(result, lhs, temp));
        }
        default -> System.out.println("error");
    }
}
```

**3） 设计寄存器分配方案（不完备的分配方案）：**

首先定义双射的寄存器分配表如下：

```
BMap<IRValue, Register> registerMap = new BMap<>();
```

寄存器分配算法的逻辑较为清晰：如果传入的操作数为立即数，则无需分配寄存器；如果传入的操作数为变量，且已经储存在寄存器中，则无需另外分配寄存器；仅当传入的操作数为当前未存入任何寄存器的变量时，遍历整个寄存器分配表寻找空闲寄存器，如果存在，则利用 replace 方法将传入的变量储存在空闲寄存器中，否则寻找后续不再使用的变量占用的寄存器并为其分配。在寻找后续不再使用的寄存器时，我们采取如下操作：遍历当前指令的所有后续指令，并依次判断后续指令涉及到的所有变量是否存在对应的寄存器，如果存在则将该寄存器号从空闲寄存器列表中删除，遍历结束后，空闲寄存器列表中剩余的寄存器即为后续不再使用的寄存器。如果无法分配任何寄存器，则报错。

```java
public void VariableToRegister(IRValue operands, int index){
    // 立即数无需分配寄存器
    if(operands.isImmediate()){
        return;
    }
    // 当前变量已经存在寄存器中，则无需再分配寄存器
    if(registerMap.containsKey(operands)){
        return;
    }
    // 寻找空闲寄存器
    for(Register register: Register.values()){
        if(!registerMap.containsValue(register)){
            registerMap.replace(operands, register);
            return;
        }
    }
    // 若无空闲寄存器，则夺取不再使用的变量所占的寄存器
    Set<Register> notUseRegs = Arrays.stream(Register.values()).collect(Collectors.toSet());
    for(int i = index; i<instructions.size(); i++){
        Instruction instruction = instructions.get(i);
        // 遍历搜寻不再使用的变量
        for(IRValue irValue: instruction.getAllOperands()){
            notUseRegs.remove(registerMap.getByKey(irValue));
        }
    }
    if(!notUseRegs.isEmpty()){
        registerMap.replace(operands, notUseRegs.iterator().next());
        return;
    }

    // 否则将无法分配寄存器并报错
    throw new RuntimeException("No enough registers!");
}
```

**4） 生成汇编代码：**

首先，定义汇编代码生成列表 asmInstructions，并初始化第一行内容为“.text”

```java
private final List<String> asmInstructions = new ArrayList<>(List.of( e1: ".text"));
```

遍历 instructions 列表，对于其中预处理后的中间代码指令，判断指令的操作类型并作不同处理。如下：

- 对于 ADD 指令：如果两个操作数都是变量，则根据寄存器分配方案为两个变量分别分配寄存器，并生成相应的 add 指令；如果其中一个操作数是立即数，另一个操作数是变量，则根据寄存器分配方案为其中的变量分配寄存器，并生成相应的 addi 指令

```
public void run() {
    // TODO: 执行寄存器分配与代码生成
    int i=0;
    String asmCode = null;
    for(Instruction instruction: instructions){
        InstructionKind instructionKind = instruction.getKind();
        switch (instructionKind){
            case ADD -> {
                IRValue lhs = instruction.getLHS();
                IRValue rhs = instruction.getRHS();
                IRVariable result = instruction.getResult();
                VariableToRegister(lhs, i);
                VariableToRegister(rhs, i);
                VariableToRegister(result, i);
                Register lhsReg = registerMap.getByKey(lhs);
                Register rhsReg = registerMap.getByKey(rhs);
                Register resultReg = registerMap.getByKey(result);
                if(rhs.isImmediate()){
                    asmCode = String.format("\taddi %s, %s, %s", resultReg.toString(), lhsReg.toString(), rhs.toString());
                }else{
                    asmCode = String.format("\tadd %s, %s, %s", resultReg.toString(), lhsReg.toString(), rhsReg.toString());
                }
            }
```

- 对于 SUB 指令：如果两个操作数都是变量，则根据寄存器分配方案为两个变量分别分配寄存器，并生成相应的 sub 指令；如果其中一个操作数是立即数，另一个操作数是变量，则根据寄存器分配方案为其中的变量分配寄存器，并生成相应的 subi 指令
- 对于 MUL 指令：直接根据寄存器分配方案为两个变量分别分配寄存器，并生成相应的 mul 指令。

```
            case SUB -> {
                IRValue lhs = instruction.getLHS();
                IRValue rhs = instruction.getRHS();
                IRVariable result = instruction.getResult();
                VariableToRegister(lhs, i);
                VariableToRegister(rhs, i);
                VariableToRegister(result, i);
                Register lhsReg = registerMap.getByKey(lhs);
                Register rhsReg = registerMap.getByKey(rhs);
                Register resultReg = registerMap.getByKey(result);
                if(rhs.isImmediate()){
                    asmCode = String.format("\tsubi %s, %s, %s", resultReg.toString(), lhsReg.toString(), rhs.toString());
                }else{
                    asmCode = String.format("\tsub %s, %s, %s",resultReg.toString(), lhsReg.toString(), rhsReg.toString());
                }
            }

            case MUL -> {
                IRValue lhs = instruction.getLHS();
                IRValue rhs = instruction.getRHS();
                IRVariable result = instruction.getResult();
                VariableToRegister(lhs, i);
                VariableToRegister(rhs, i);
                VariableToRegister(result, i);
                Register lhsReg = registerMap.getByKey(lhs);
                Register rhsReg = registerMap.getByKey(rhs);
                Register resultReg = registerMap.getByKey(result);
                asmCode = String.format("\tmul %s, %s, %s", resultReg.toString(), lhsReg.toString(), rhsReg.toString());
            }
```

- 对于 MOV 指令：由于只有两个操作数（源操作数和目的操作数），因此需要考虑其右操作数 form 是否为立即数。如果右操作数为变量，则生成相应的 mv 指令（即 mv rd, rs1）；如果右操作数为立即数，则生成相应的 li 指令（即 li rd, rs1）。
- 对于 RET 指令，其目的操作数固定存放在返回值寄存器 a0 中，因此通过 getReturnValue() 方法获取返回值后直接生成相应的 mv 语句即可（即 mv a0, rs1）。

最后，将汇编指令对应的中间代码作为注释输出，并将生成的汇编指令添加至 asmInstructions 列表中。

```
        case MOV -> {
            IRValue form = instruction.getFrom();
            IRVariable result = instruction.getResult();
            VariableToRegister(form, i);
            VariableToRegister(result, i);
            Register formReg = registerMap.getByKey(form);
            Register resultReg = registerMap.getByKey(result);
            if(form.isImmediate()){
                asmCode = String.format("\tli %s, %s", resultReg.toString(), form.toString());
            }else {
                asmCode = String.format("\tmv %s, %s", resultReg.toString(), formReg.toString());
            }
        }

        case RET -> {
            IRValue returnValue = instruction.getReturnValue();
            Register returnValueReg = registerMap.getByKey(returnValue);
            asmCode = String.format("\tmv a0, %s", returnValueReg.toString());
        }

        default -> System.out.println("error asm!");
    }
    asmCode += "\t\t#  %s".formatted(instruction.toString());
    asmInstructions.add(asmCode);
    i++;

    if(instructionKind == InstructionKind.RET){
        break;
    }
```

5） 至此，汇编代码已生成完毕，输出至文件中。

```
public void dump(String path) {
    // TODO: 输出汇编代码到文件
    FileUtils.writeLines(path, asmInstructions);
}
```

# 4 实验结果与分析

## 4.1 词法分析

输入 1：码点文件（编码表） coding_map.csv

规定了所有词法单元的类别以及各自对应的编码

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

输入 2：待编译的代码 input_code.txt

```
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：符号表 old_symbol_table.txt

此时还未进行语义分析，因此符号表中的符号 type 属性均为 null

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

输出 2：词法单元列表 token.txt

```
(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,)
($,)
```

**分析：**

解析待编译的代码后得到了词法单元列表和不包含 type 属性信息的符号表。所有输出文件均与 std 目录下的对应文件一致，正确完成了词法分析器的功能。

## 4.2 语法分析

输入 1：码点文件 coding_map.csv（与词法分析相一致）

输入 2：待编译的代码 input_code.txt（与词法分析相一致）

输入 3：语法文件 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;
```

输入 4：LR1 分析表 LR1_table.csv

| 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 Se | | 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 - | reduce E - | shift 20 | | | | | reduce E -> A | | | | | | | |
| 12 | | | | reduce A - | reduce A - | reduce A -> B | | | | | reduce A -> B | | | | | | | |
| 13 | | | | reduce B - | reduce B - | reduce B -> id | | | | | reduce B -> id | | | | | | | |
| 14 | shift 24 | shift 25 | | | | | | | | shift 26 | | | 21 | | | 22 | 23 | |
| 15 | | | | reduce B - | reduce B - | 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 - | reduce E - | reduce E - | shift 33 | | | | | | | | | | | | |
| 23 | | | reduce A - | reduce A - | reduce A - | reduce A -> B | | | | | | | | | | | | |
| 24 | | | reduce B - | reduce B - | reduce B - | reduce B -> id | | | | | | | | | | | | |
| 25 | shift 24 | shift 25 | | | | | | | | shift 26 | | | 34 | | | 22 | 23 | |
| 26 | | | reduce B - | reduce B - | reduce B - | reduce B -> IntConst | | | | | | | | | | | | |
| 27 | | | | reduce E - | reduce E - | shift 20 | | | | | reduce E -> E + A | | | | | | | |
| 28 | | | | reduce E - | reduce E - | shift 20 | | | | | reduce E -> E - A | | | | | | | |
| 29 | | | | reduce A - | reduce A - | reduce A -> A * B | | | | | reduce A -> A * B | | | | | | | |
| 30 | | | | reduce B - | reduce B - | 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 - | reduce E - | reduce E - | shift 33 | | | | | | | | | | | | |
| 36 | | | reduce E - | reduce E - | reduce E - | shift 33 | | | | | | | | | | | | |
| 37 | | | reduce A - | reduce A - | reduce A - | reduce A -> A * B | | | | | | | | | | | | |
| 38 | | | reduce B - | reduce B - | reduce B - | reduce B -> ( E ) | | | | | | | | | | | | |

输出 1：符号表 old_symbol_table.txt（与词法分析相一致）

输出 2：词法单元列表 token.txt（与词法分析相一致）

输出 3：归约得到的产生式列表 parser_list.txt

按照归约的先后顺序输出产生式到文件中

```
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
E -> A
B -> id
A -> B
E -> 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
```

**分析：**

新增的输入文件是语法文件以及使用第三方工具（编译工作台）生成的 LR1 分析表。新增的输出文件是生成的产生式列表，即按照归约的先后顺序输出的产生式。该文件与 std 目录下的对应文件内容一致，正确实现了语法分析的功能。

## 4.3 中间代码生成

输入 1：码点文件 coding_map.csv（与词法分析相一致）
输入 2：待编译的代码 input_code.txt（与词法分析相一致）
输入 3：语法文件 grammar.txt（与语法分析相一致）
输入 4：LR 分析表 LR1_table.csv（与语法分析相一致）
输出 1：语义分析前的符号表 old_symbol_table.txt（与词法分析相一致）
输出 2：词法单元列表 token.txt（与词法分析相一致）
输出 3：归约得到的产生式列表 parser_list.txt（与语法分析相一致）
输出 4：语义分析后的符号表 new_symbol_table.txt
更新符号表中关于变量的 type 属性信息

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

输出 5：中间表示 intermediate_code.txt

```
(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 6：中间表示的模拟执行结果 ir_emulate_result.txt

```
144
```

**分析：**

语义分析及中间代码生成过程没有新增的输入文件。新增的输出文件包括语义分析后的符号表、中间表示和中间表示的模拟执行结果。所有输出文件均与 std 目录下的对应文件内容一致，正确的实现了语义分析与中间代码生成的功能。

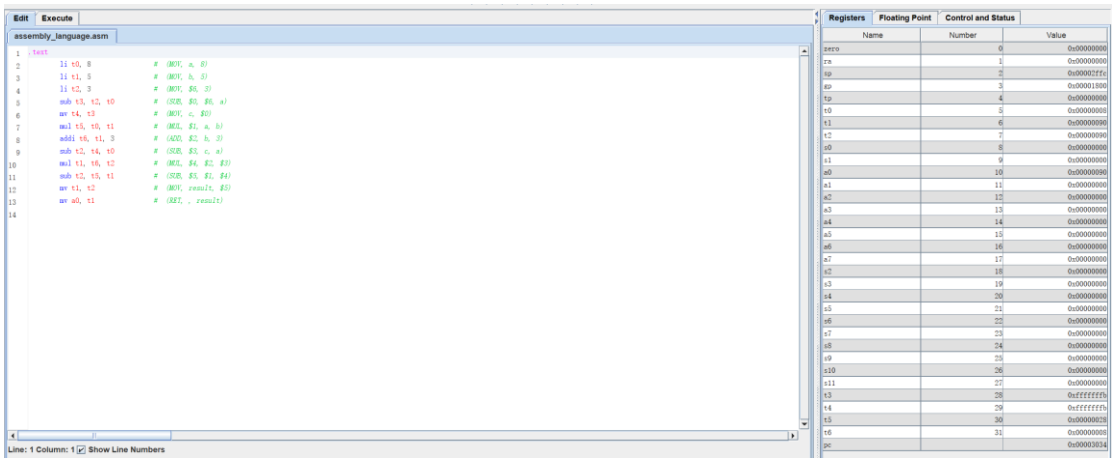
观察及分析可知，程序更新了符号表中变量类型 type 属性的信息。同时，input_code.txt 源代码返回值 8×5 - (3 + 5) * (3 - 8 - 8) = 144 与中间代码模拟运行的结果一致，可见程序能够准确的进行语义分析。

## 4.4 目标代码生成

输入 1：码点文件 coding_map.csv（与词法分析相一致）
输入 2：待编译的代码 input_code.txt（与词法分析相一致）
输入 3：语法文件 grammar.txt（与语法分析相一致）
输入 4：LR 分析表 LR1_table.csv（与语法分析相一致）
输出 1：符号表 old_symbol_table.txt（与词法分析相一致）
输出 2：词法单元列表 token.txt（与词法分析相一致）
输出 3：归约得到的产生式列表 parser_list.txt（与语法分析相一致）
输出 4：语义分析后得到的新符号表 new_symbol_table.txt
输出 5：生成的中间代码 intermediate_code.txt
输出 6：中间代码模拟执行的结果 ir_emulate_result.txt
输出 7：生成的汇编代码 assembly_language.asm

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

生成的汇编指令在 RARS 上运行的结果:




Registers | Floating Point | Control and Status

| Name | Number | Value |
|---|---|---|
| zero | 0 | 0x00000000 |
| ra | 1 | 0x00000000 |
| sp | 2 | 0x00002ffc |
| gp | 3 | 0x00001800 |
| tp | 4 | 0x00000000 |
| t0 | 5 | 0x00000008 |
| t1 | 6 | 0x00000090 |
| t2 | 7 | 0x00000090 |
| s0 | 8 | 0x00000000 |
| s1 | 9 | 0x00000000 |
| a0 | 10 | 0x00000090 |
| a1 | 11 | 0x00000000 |
| a2 | 12 | 0x00000000 |
| a3 | 13 | 0x00000000 |
| a4 | 14 | 0x00000000 |
| a5 | 15 | 0x00000000 |
| a6 | 16 | 0x00000000 |
| a7 | 17 | 0x00000000 |
| s2 | 18 | 0x00000000 |
| s3 | 19 | 0x00000000 |
| s4 | 20 | 0x00000000 |
| s5 | 21 | 0x00000000 |
| s6 | 22 | 0x00000000 |
| s7 | 23 | 0x00000000 |
| s8 | 24 | 0x00000000 |
| s9 | 25 | 0x00000000 |
| s10 | 26 | 0x00000000 |
| s11 | 27 | 0x00000000 |
| t3 | 28 | 0xfffffffb |
| t4 | 29 | 0xfffffffb |
| t5 | 30 | 0x00000028 |
| t6 | 31 | 0x00000008 |
| pc | | 0x00003034 |

**分析：**

目标代码生成过程没有新增的输入文件。新增的输出文件为生成的汇编代码。将生成的汇编代码在 RARS 上运行，寄存器 a0 的返回值为 0x90（即十进制数 144）。可见正确实现了目标代码生成的功能。

# 5 实验中遇到的困难与解决办法

## 5.1 遇到的困难

1. 本次实验已提供框架，但由于框架代码量较大、类及方法的定义较多，因此在刚上

手实验的时候需要花费大量时间理解框架中的代码及逻辑。除此之外，由于每次实验间隔时间较长，也会偶尔忘记之前实验中所调用的框架方法，导致实现进度缓慢。

2. 对理论课所学知识不够熟悉，导致不能很好地把理论课所学知识与实验课的实验要求结合起来。

## 5.2 解决办法

反复阅读指导书，真的很重要！（比如实验二中我就漏掉了关于 Symbol 类的定义的部分，后来还是询问了身边同学才意识到）。同时充分理解框架中已给出的注释，弄清哪些函数在哪里被调用过、哪些函数已经被实现但还未被调用、哪些代码不需要修改、哪些部分需要我们进行补充等等。同时，在迷茫的时候积极向老师及同学求助，往往可以加深对本次实验任务的理解（比如本次实验三，在最初我完全不知道从何下手，后来是在老师的指导下才充分理解需要做什么、大致怎么去做等等）。在理解后再着手写代码，往往会起到事半功倍的效果，从而大大提高实验效率。

## 5.3 收获和建议

本学期的编译原理实验，让我全面体验了一个小型编译器开发的完整流程，既巩固了我在理论课上所学的词法分析、语法分析、语义分析、中间代码生成和汇编生成等理论知识，还使我对编译器的工作机制有了更深入的理解。编程的过程并不复杂，但也使我进一步了解了 Java 语言（比如之前我不知道 Java 中已经定义了 Stack 这个数据结构），锻炼了编程能力以及代码阅读能力。

同时，代码框架中也有很多值得学习的技巧。比如在实验二中，NonTerminal 和 Token 没有公共父类，但符号栈却同时需要存放这两种元素，因此可以定义一个类似共用体的 Symbol 类来包装 Token 和 NonTerminal，从而达到把 NonTerminal 和 Token 同时存入符号栈中的目的，而我在实验三中也借鉴了这种方式。实验四中的双射表 BMap 的设计也同样巧妙。

关于建议，建议可以再细化指导书，添加对每次任务流程的描述及实验框架的描述，同时为指导书添加目录（每次翻的时候真的很麻烦）。

最后，感谢帮助过我的老师及同学，祝愿本课程越办越好！