# 1 实验目的与方法

**实验目的为本次实验的实验目的，方法为所使用的语言，软件环境等。**

## 1.1 词法分析器

实验目的：设计并实现一个可以对类 C 语言源程序段进行词法分析的词法分析程序，加深对高级语言的认识和对词法分析程序的功能及实现的理解，加深对类 C 语言文法的认识，理解有穷自动机、编码表和符号表在整个编译过程中的应用。
语言：Java
软件：IntelliJ IDEA
环境：Windows 10

## 1.2 语法分析

实验目的：深入了解语法分析程序的实现原理及方法，理解 LR(1)分析法是严格的从左到右扫描和自底向上的语法分析方法。
语言：Java
软件：IntelliJ IDEA、编译工作台
环境：Windows 10

## 1.3 典型语句的语义分析及中间代码生成

实验目的：加深对自底向上的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。巩固对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。
语言：Java
软件：IntelliJ IDEA
环境：Windows 10

## 1.4 目标代码生成

实验目的：加深对编译器总体结构的理解和掌握，掌握常见的 RISC-V 指令的使用方法，理解并掌握目标代码生成算法和寄存器选择算法。
语言：Java
软件：IntelliJ IDEA、rars
环境：Windows 10

# 2 实验内容及要求

每次实验室的实验内容和要求描述清楚。

## 2.1 词法分析器

编写一个词法分析程序，读取代码文件，对文件内自定义的类 C 语言程序段进行词法分析。处理 C 语言源程序，过滤掉无用符号，分解出正确的单词，以二元组的形式输出存放在文件中。

1．词法分析程序输入：以文件形式存放的自定义的类 C 语言程序段;

2．词法分析程序输出：以文件形式存放的 Token 串和简单符号表；

3．词法分析程序输入单词类型要求：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. 利用 LR(1)分析法，设计语法分析程序，对输入的单词符号串进行语法分析；
2. 输出推导过程中所用产生式序列，并将其保存在输出文件中；
3. 要求：实现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；
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. 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。
4. 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. 实现目标代码生成算法和寄存器选择算法，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。

# 3 实验总体流程与函数功能描述

## 3.1词法分析

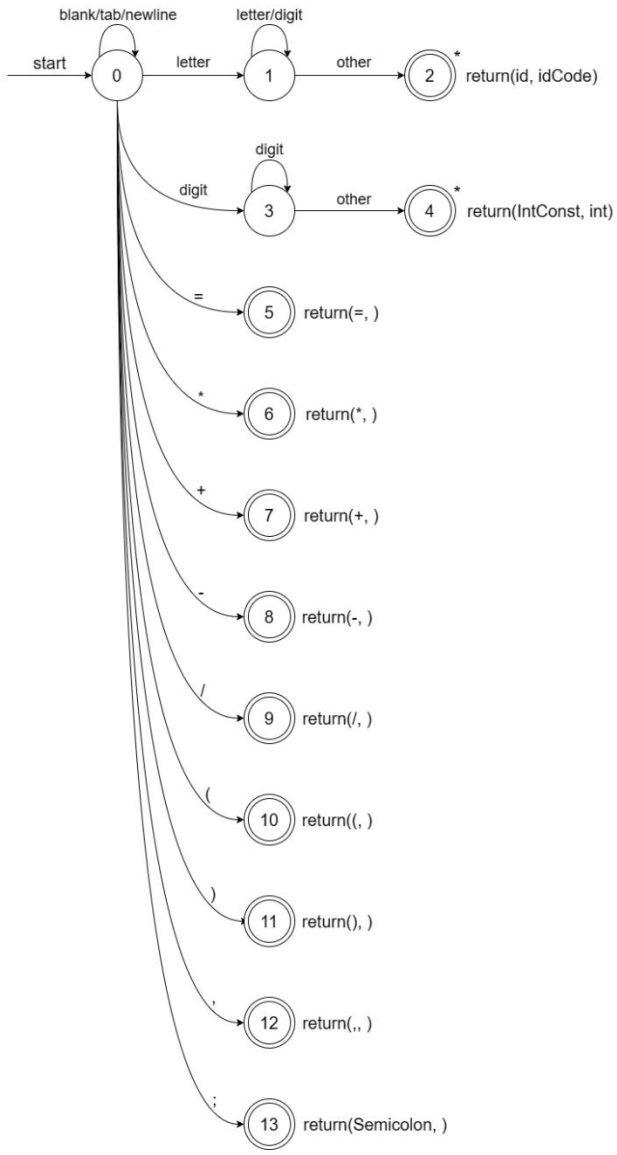
### 3.1.1 编码表

| | |
|---|---|
| 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

G=(V, T, P, S)，其中 V={S, A, B, C, digit, no_0_digit, letter}，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 表示字母及下划线：A, B, …, Z, a, b, …, z, _
标识符（及关键字）：S → letterA　A → letterA | digitA | ε
整常数：S → 0 | no_0_digitB　B → digitB | ε
运算符及分界符：S → C　C → = | * | + | - | / | ( | ) | , | ;

### 3.1.3 状态转换图




### 3.1.4 词法分析程序设计思路和算法描述

首先，利用 FileUtils 中提供的方法从指定路径中完整地读入源程序文件（用一个字符串接收读入的所有内容）。

```
content = FileUtils.readFile(path);
```

然后，借助指针 i 依次读入字符串中的每个字符：如果当前读入的字符为空格/换行符/tab 符，就直接令 i 前移，跳过该字符；

```
for (i = 0; i < content.length(); i++) {
    ch = content.charAt(i);
    if (ch == ' ' || ch == '\n' || ch == '\t') {
        continue;
    }
```

如果当前读入的字符为字母，就先将其记录下来，然后移动指针 i，将其后面的字符依次添加到记录该字符的字符串 str 的尾部，直到 i 指向的字符既不是字母或数字，也不是下划线——此时，需要将 i 回退一个字符（因为当前读入的无关字符其实是下一个单词的开始字符，且下一次读入字符前还会执行一次 i++），然后退出当前循环。循环结束后，我们还需要对在这个过程中生成的字符串 str 进行一次检查：如果 str 中存放的正好是“int”或“return”，就代表我们遇到的是一个关键字，那么直接将其对应的 token 添加到我们事先声明的用于存放 token 串的 tokenList 中去即可；反之，就代表我们遇到的是一个标识符，那么我们除了要把该标识符对应的 token（包含其种类“id”和其 id 值 str）添加到 tokenList 中去以外，还应把它的名字 str 添加到符号表中。

```
if (Character.isLetter(ch)) {
    str = String.valueOf(ch);
    for (i = i+1; i < content.length(); i++) {
        ch = content.charAt(i);
        if (!Character.isLetterOrDigit(ch) && ch != '_') {
            i--;
            break;
        } else {
            str += ch;
        }
    }
    if ("int".equals(str) || "return".equals(str)) {
        tokenList.add(Token.simple(str));
    } else {
        tokenList.add(Token.normal( tokenKindId: "id", str));
        if (!symbolTable.has(str)) {
            symbolTable.add(str);
        }
    }
```

如果当前读入的字符是数字，就也先将其记录下来，然后移动指针 i，将其后面的字符依次添加到记录该字符的字符串 str 的尾部，直到 i 指向的字符不是数字为止——此时，需要将 i 回退一个字符（因为当前读入的无关字符其实是下一个单词的开始字符，且下一次读入字符前还会执行一次 i++），然后退出当前循环。循环结束后，我们需要将在这个过程中生成的字符串 str 对应的 token（包含其种类“IntConst”和其具体数值）添加到集合 tokenList 中去。

```
} else if (Character.isDigit(ch)) {
    str = String.valueOf(ch);
    for (i = i+1; i < content.length(); i++) {
        ch = content.charAt(i);
        if (!Character.isDigit(ch)) {
            i--;
            break;
        } else {
            str += ch;
        }
    }
    tokenList.add(Token.normal( tokenKindId: "IntConst", str));
```

如果当前读入的字符不属于以上提到的所有情况（说明当前正在考察的字符只是一个普通的运算符或标点），就直接将其对应的 token 添加到集合 tokenList 中（注意：分号的种别码为“Semicolon”，而其他符号的种别码就是它们本身）。

```
} else if (ch == ';') {
    tokenList.add(Token.simple( tokenKindId: "Semicolon"));
} else {
    str = String.valueOf(ch);
    tokenList.add(Token.simple(str));
}
```

当所有字符都被读入后，我们还需要将文件结束符对应的 token 添加到集合 tokennList 中去，以示代码段的词法分析结束。

最后，为了方便已经生成的 token 串的获取和遍历，我们还需要实现 getToken 方法，通过调用 List 类中的 iterator 方法将 tokenList 转化成一个迭代器。

除此之外，本次实验中还实现了 SymbolTable 类中的 get、add、has 和 getAllEntries 方法，其中，get 方法和 has 方法的主要实现逻辑是遍历符号表中的所有条目，寻找其中符号名与给定字符串相同的条目；add 方法的主要逻辑是先借助 has 方法判断待添加的条目是否已经存在，若不存在，再向符号表中添加一个以给定文本为关键字的符号条目；getAllEntries 方法的主要逻辑是声明一个以符号名为键、以符号条目为值的哈希表，然后遍历符号表，将表中的每个条目及其关键字都添加到该哈希表中。

## 3.2 语法分析

### 3.2.1 拓展文法

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

### 3.2.2 LR1 分析表

| 状态 | ACTION | | | | | | | | | | | | GOTO | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Semicolon | id | int | = | return | + | - | * | ( | ) | IntConst | $ | S_list | S | D | E | 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 | reduce S -> return E | | | | | shift 18 | shift 19 | | | | | | | | | | | |
| 11 | reduce E -> A | | | | | reduce E -> A | reduce E -> A | shift 20 | | | | | | | | | | |
| 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 | reduce S -> id = E | | | | | shift 18 | shift 19 | | | | | | | | | | | |
| 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 | shift 33 | | reduce E -> A | | | | | | | | |
| 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 | reduce E -> E + A | shift 20 | | | | | | | | | | |
| 28 | reduce E -> E - A | | | | | reduce E -> E - A | reduce E -> E - A | shift 20 | | | | | | | | | | |
| 29 | reduce A -> A * B | | | | | reduce A -> A * B | reduce A -> A * B | reduce A -> A * B | | | | | | | | | | |
| 30 | | shift 24 | | | | | | | shift 25 | | shift 26 | | | | | | 35 | 23 |
| 31 | | shift 24 | | | | | | | shift 25 | | shift 26 | | | | | | 36 | 23 |
| 32 | reduce B -> ( E ) | | | | | reduce B -> ( E ) | reduce B -> ( E ) | reduce B -> ( E ) | | | | | | | | | | |
| 33 | | shift 24 | | | | | | | shift 25 | | shift 26 | | | | | | | 37 |
| 34 | | | | | | shift 30 | shift 31 | | | shift 38 | | | | | | | | |
| 35 | | | | | | reduce E -> E + A | reduce E -> E + A | shift 33 | | reduce E -> E + A | | | | | | | | |
| 36 | | | | | | reduce E -> E - A | reduce E -> E - A | shift 33 | | reduce E -> E - A | | | | | | | | |
| 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：声明一个专门用于存放状态类（Status）对象的栈 statusStack，通过调用 Stack 类中的 add、pop、peek 等方法来分别实现向状态栈中压入状态、从状态栈栈顶弹出状态、获取状态栈栈顶的状态等需求。

符号栈：先创建一个整合了 Token 类型和 NonTerminal 类型的 Symbol 类，为 Token 类和 NonTerminal 类各写一个构造方法（以输入参数作为类中对应类型属性的值，同时将类中对应另一个类型的属性置为 null）及判断方法，然后在语法分析驱动程序中声明一个专门用来存放 Symbol 类对象的栈 symbolStack，最终通过调用 Stack 类中的 add、pop、peek 等方法来分别实现向符号栈中压入符号（token 或非终结符）、从符号栈栈顶弹出符号、获取符号栈栈顶的符号等需求。

### 3.2.4 LR 驱动程序设计思路和算法描述

首先，实现 loadTokens 方法：用迭代的方式将词法单元全部添加到语法分析类的私有集合 tokenList 中去。

```
Iterator<Token> it = tokens.iterator();
while (it.hasNext()) {
    tokenList.add(it.next());
}
```

然后，实现 loadLRTable 方法：将 LR(1)分析表中的初始状态存储起来，以完成对 LR(1)分析

表的加载。

```
initStatus = table.getInit();
```

最后，实现驱动程序：首先，声明一个状态栈和一个符号栈，并将从 LR 分析表中得到的初始状态压入状态栈，同时将文件结束符压入符号栈，以完成对两个栈的初始化。

```
statusStack.add(initStatus);
symbolStack.add(new Symbol(Token.eof()));
```

然后，遍历从词法分析过程中得到的词法单元串，用 action 记录下在当前正处于状态栈栈顶的状态下读入当前正好遍历到的 token 时应该执行的动作。如果当前应该执行的动作是 shift（即移进），就把要移进状态栈的状态压入状态栈中，同时把当前正在遍历的 token 压入符号栈中，然后通知各个观察者；

```
for (i = 0; i < tokenList.size(); i++) {
    action = statusStack.peek().getAction(tokenList.get(i));
    if (action.getKind() == Action.ActionKind.Shift) {
        statusStack.add(action.getStatus());
        symbolStack.add(new Symbol(tokenList.get(i)));
        callWhenInShift(statusStack.peek(), tokenList.get(i));
```

如果当前应该执行的动作是 reduce（即归约），就先根据要归约的产生式的右部的长度，从状态栈和符号栈中分别弹出相应数量的元素，然后再把待归约的产生式左部的非终结符压入符号栈中，同时根据当前状态栈栈顶的状态以及压入符号栈的非终结符从 goto 表中找到相应的状态，将其压入状态栈中，最后通知各个观察者即可。此外，因为在归约动作下，当前读入的 token 不应被消耗，所以我们还需要令 i--，以确保下一次被读入的还是当前这个 token。

```
} else if (action.getKind() == Action.ActionKind.Reduce) {
    length = action.getProduction().body().size();
    while (length > 0) {
        statusStack.pop();
        symbolStack.pop();
        length--;
    }
    symbolStack.add(new Symbol(action.getProduction().head()));
    statusStack.add(statusStack.peek().getGoto(action.getProduction().head()));
    callWhenInReduce(statusStack.peek(), action.getProduction());
    i--;
```

如果当前应该执行的动作是 accept（即接受），就说明语法分析结束，那么我们在通知完各个观察者后直接结束循环即可。

```
} else if (action.getKind() == Action.ActionKind.Accept) {
    callWhenInAccept(statusStack.peek());
    break;
```

如果当前应该执行的动作是 error，就说明语法分析过程出错，那么我们也要结束遍历。

```
} else {
    System.out.println("ERROR! ");
    break;
}
```

## 3.3 语义分析和中间代码生成

### 3.3.1 翻译方案

P → S_list;
S_list → S Semicolon S_list;
S_list → S Semicolon;
S → D id; {p = lookup(id.name); if p != nil then enter(id.name, D.type) else error}
D → int; {D.type = int;}
S → id = E; {gencode(id.val = E.val);}
S → return E; {gencode(return E.val);}
E → E + A; {gencode(newtemp = E.val + A.val); E.val = newtemp;}
E → E - A; {gencode(newtemp = E.val - A.val); E.val = newtemp;}
E → A; {E.val = A.val;}
A → A * B; {gencode(newtemp = A.val * B.val); A.val = newtemp;}
A → B; {A.val = B.val;}
B → ( E ); {B.val = E.val;}
B → id; {B.val = id.val;}
B → IntConst; {B.val = IntConst.lexval;}

### 3.3.2 语义分析和中间代码生成的数据结构

语义分析：首先，向实验二创建的 Symbol 类中添加一个 SourceCodeType 类型的私有成员变量，同时仿照之前写的构造方法与判断方法，写一个用来构造 SourceCodeType 类实例的构造方法（在该构造方法中其他类型的成员变量都应置为 null）和一个用来判断某对象是否为 SourceCodeType 类型的判断方法。然后，在语义分析类中声明一个专门用来存放 Symbol 对象的符号栈 symbolStack，以及一个符号表 symbolTable，作为语义分析阶段的数据结构。

```
8 usages
private final Stack<Symbol> symbolStack = new Stack<>();
2 usages
private SymbolTable symbolTable;
```

中间代码生成：首先，向实验二创建的 Symbol 类中再添加一个 IRValue 类型的私有成员变量，同时仿照之前写的构造方法与判断方法，写一个用来构造 IRValue 实例的构造方法（在该构造方法中其他类型的成员变量都应置为 null）和一个用来判断某对象是否为 IRValue 类型的判断方法。然后，在中间代码生成类中声明一个专门用来存放 Symbol 对象的符号栈 symbolStack，以及一个用来存放中间指令 Instruction 的集合 instructions，作为中间代码生成阶段的数据结构。

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

此外，为了满足语义分析和中间代码生成的过程中存在的一些获取符号的具体类型以及调用

某个具体类中的方法的需求，我们还需要在 Symbol 类中创建几个 get 方法，以实现将某个 Symbol 对象转变成其对应的具体类的对象的功能。

```
public Token getToken() { return token; }

1 usage
public SourceCodeType getSourceCodeType() { return sourceCodeType; }

9 usages
public IRValue getIrValue() { return irValue; }
```

### 3.3.3 语法分析程序设计思路和算法描述

语义分析（SemanticAnalyzer）：

首先，实现 setSymbolTable 方法，将符号表存储起来。

```
@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，可以将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
    symbolTable = table;
}
```

然后，实现 whenShift 方法：只要遇到移进动作，就把当前的 token 压入符号栈中。

```
@Override
public void whenShift(Status currentStatus, Token currentToken) {
    // TODO: 该过程在遇到 shift 时要采取的代码动作
    symbolStack.add(new Symbol(currentToken));
}
```

最后，实现 whenReduce 方法：（考虑到语义分析阶段我们要做的只是收集标识符的 type 属性，更新符号表，所以这里我只对与标识符类型相关的产生式进行了特殊处理）

如果当前要归约的产生式是 D → int，即产生式的索引号为 5，就说明此时符号栈的栈顶元素应该是一个类型为 int 的 token，那么我们只需要将其弹出，然后按照该产生式对应的翻译方案，将 int 类型赋值给 D 的 type 属性，同时把 D 的 type 属性压入栈中即可。

```
if (production.index() == 5) {
    symbolStack.pop();
    symbolStack.add(new Symbol(SourceCodeType.Int));
```

如果当前要归约的产生式是 S → D id，即产生式的索引号为 4，就说明此时位于符号栈栈顶的是一个与某标识符对应的 token 以及 D 的 type 属性（因为之前归约 D 的时候我们只将 D 的 type 属性压入了栈中），那么我们只需要从符号表中找出符号栈栈顶 token 的属性值对应的符号条目，然后将该条目的 type 属性设置成弹出 token 后正位于符号栈栈顶的 D.type（同时将其弹出）即可，最后，我们还应将该产生式左部的非终结符 S 压入符号栈中占位（S 不需要携带信息）。

```
} else if (production.index() == 4) {
    final var entry = symbolTable.get(symbolStack.pop().getToken().getText());
    entry.setType(symbolStack.pop().getSourceCodeType());
    symbolStack.add(new Symbol(production.head()));
```

如果当前要归约的产生式不是上述两条，就说明该产生式与标识符的类型无关，那么我们直接按照语法分析中的 reduce 流程，从符号栈中弹出该产生式的右部，然后再将该产生式的左部压入栈中占位即可。

```
} else {
    int length = production.body().size();
    while (length > 0) {
        symbolStack.pop();
        length--;
    }
    symbolStack.add(new Symbol(production.head()));
}
```

中间代码生成（IRGenerator）：

首先，实现 whenShift 方法：只要遇到移进动作，就把当前的 token 压入符号栈中。

```
@Override
public void whenShift(Status currentStatus, Token currentToken) {
    // TODO
    symbolStack.add(new Symbol(currentToken));
}
```

然后，实现 whenReduce 方法：（考虑到中间代码生成阶段我们要做的只是传值和生成中间代码，所以对于前五条与中间代码生成无关的产生式，我们这里不作翻译）

如果当前要归约的产生式是 S → id = E（即产生式索引号为 6），就依次取出位于符号栈栈顶的 E.val、=对应的 token 以及与某标识符相对应的 token，然后利用 Instruction 类中的 createMov 方法生成一条形如(MOV, id, val)的中间指令，并把它添加到中间指令集合 instrutions 中，最后将产生式左部的 S 压入符号栈中占位（S 不需要携带信息）。

```
switch (production.index()) {
    case 6 :
        val = symbolStack.pop().getIrValue();
        symbolStack.pop();
        instructions.add(Instruction.createMov(IRVariable.named(symbolStack.pop().getToken().getText()), val));
        symbolStack.add(new Symbol(production.head()));
        break;
```

如果当前要归约的产生式是 S → return E（即产生式索引号为 7），就依次取出位于符号栈栈顶的 E.val 和与关键字 return 相对应的 token，然后利用 Instruction 类中的 createRet 方法生成一条形如(RET, , val)的中间指令，并把它添加到中间指令集合 instructions 中，最后将将产生式左部的 S 压入符号栈中占位（S 不需要携带信息）。

```
case 7 :
    val = symbolStack.pop().getIrValue();
    symbolStack.pop();
    instructions.add(Instruction.createRet(val));
    symbolStack.add(new Symbol(production.head()));
    break;
```

如果当前要归约的产生式是 E → E + A（即产生式索引号为 8），就依次取出位于符号栈栈顶的 A.val、+对应的 token 以及 E.val，然后设置一个临时变量 temp，利用 Instruction 类中的 createAdd 方法生成一条 ADD 指令（以 temp 作为存放结果的变量），并把它添加到中间指令集合 instructions 中去，最后，将临时变量 temp 作为非终结符 E 的最新值压入栈中。

```
case 8 :
    temp = IRVariable.temp();
    val = symbolStack.pop().getIrValue();
    symbolStack.pop();
    instructions.add(Instruction.createAdd(temp, symbolStack.pop().getIrValue(), val));
    symbolStack.add(new Symbol(temp));
    break;
```

同理，如果当前要归约的产生式是 E → E - A 或 A → A * B（即索引号为 9 或 11），就仿照以上的步骤生成一条 SUB 指令或 MUL 指令（均以一个临时变量 temp 作为存放结果的变量），

然后将用到的临时变量作为产生式左部非终结符的新值压入栈中。（代码相似，不再展示）
如果当前要归约的产生式是 E → A 或 A → B（即索引号为 10 或 12），因为其翻译方案就是把产生式右部非终结符的值传给产生式左部的非终结符，体现在对符号栈的操作上就是先弹出产生式右部非终结符的 val，然后再将其作为产生式左部非终结符的 val 压入栈中，实际上翻译前后符号栈并没有改变，所以我们此时直接 break 即可，不需要执行额外的操作。
如果当前要归约的产生式是 B → ( E )（即产生式索引号为 13），那么根据翻译方案，应该先把位于符号栈栈顶的括号和 E.val 一起弹出，然后再把 E.val 作为 B.val 重新压入栈中，以完成值的传递和括号的去除。

```
case 13 :
    symbolStack.pop();
    val = symbolStack.pop().getIrValue();
    symbolStack.pop();
    symbolStack.add(new Symbol(val));
    break;
```

如果当前要归约的产生式是 B → id（即产生式索引号为 14），就取出当前位于符号栈栈顶的标识符 token，然后创建一个以该标识符的 id 名命名的变量 val，并将该变量作为 B.val 压入符号栈中。

```
case 14 :
    val = IRVariable.named(symbolStack.pop().getToken().getText());
    symbolStack.add(new Symbol(val));
    break;
```

如果当前要归约的产生式是 B → IntConst（即产生式索引号为 15），就取出当前位于符号栈栈顶的整型数 token，然后创建一个与其具体数值相对应的立即数 val，并将该立即数作为 B.val 压入符号栈中。

```
case 15 :
    val = IRImmediate.of(Integer.parseInt(symbolStack.pop().getToken().getText()));
    symbolStack.add(new Symbol(val));
    break;
```

如果当前要归约的产生式不属于以上任意一种情况，就说明它与传值以及中间指令生成无关，那么我们直接按照语法分析中的 reduce 流程，从符号栈中弹出该产生式的右部，然后再将该产生式的左部压入栈中占位即可。

```
default :
    int length = production.body().size();
    while (length > 0) {
        symbolStack.pop();
        length--;
    }
    symbolStack.add(new Symbol(production.head()));
    break;
```

## 3.4 目标代码生成

### 3.4.1 设计思路和算法描述

首先，定义一个枚举型变量 Reg，用于表示本次实验中用到的临时寄存器。

```
public enum Reg {
    //本实验中使用的临时寄存器
    no usages
    t0, t1, t2, t3, t4, t5, t6
}
```

然后，借助 Map 类定义一个双射关系，同时利用 Map 类中的 remove、containsKey、put 以及 get 方法为该双射类实现一系列功能方法，以便后续维护寄存器的占用信息，实现寄存器和变量的双向查找。

```
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<>();
```

最后，实现 AssemblyGenerator 类，完成寄存器分配与目标代码生成：
AssemblyGenerator 类的数据结构如下图所示：

```
private final List<Instruction> instructions = new ArrayList<>();
23 usages
private final HashMap<Integer, IRVariable> irVariables = new HashMap<>();
11 usages
private final BMap<IRVariable, Reg> regAlloc = new BMap<>();
3 usages
private final List<String> assemblyCode = new ArrayList<>();
```

（instructions：经过预处理后的中间指令的集合；irVariables：存储中间代码中各变量被引用的次序信息；regAlloc：存储寄存器的占用/分配信息；assemblyCode：存储生成的目标代码，一个字符串对应一条指令/一行代码）
首先，我们应加载前端提供的中间代码，对其进行预处理、存储以及信息提取。具体流程为：遍历前端提供的原始中间代码，根据操作数个数、指令类型以及立即数情况的不同，将中间指令划分成不同的类别，并对每一类指令进行不同的处理：
对于二元指令（有返回值，且有两个参数），我们需要对其操作数进行进一步的考察——如果其左右操作数都是立即数，就要根据指令的类型对这两个操作数直接进行求值，然后利用得到的结果和该指令本身用来接收结果的变量生成一条 MOV 指令，并把其添加到指令集合 instructions 中；除此之外，对于出现在该条 MOV 指令中的 result 变量，我们还应以变量计数器 i 的值作为其次序编号，然后把该变量连带着它的次序一起添加到哈希表 irVariables 中，为后续进行寄存器分配做准备。

```
if (instruction.getKind().isBinary()) {
    if (instruction.getLHS().isImmediate() && instruction.getRHS().isImmediate()) {
        IRImmediate immediate1 = (IRImmediate) instruction.getLHS();
        IRImmediate immediate2 = (IRImmediate) instruction.getRHS();
        int value;
        switch (instruction.getKind()) {
            case ADD :
                value = immediate1.getValue() + immediate2.getValue();
                break;
            case SUB :
                value = immediate1.getValue() - immediate2.getValue();
                break;
            default :
                value = immediate1.getValue() * immediate2.getValue();
                break;
        }
        instructions.add(Instruction.createMov(instruction.getResult(), IRImmediate.of(value)));
        irVariables.put(i++, instruction.getResult());
```

如果其左右操作数中只有一个是立即数，那么需要我们进行预处理的情况有以下几种：
（只有 ADD/SUB 指令+右操作数是立即数的情况不需要预处理<其对应的是汇编中的 addi/subi 指令>，其他情况都需要预处理<汇编中没有 muli 指令、汇编中不存在立即数在左的指令>）
一、该指令为 MUL 指令且其左操作数为立即数，那么我们就需要生成一条以其左操作数为数据来源的 MOV 指令，并新声明一个临时变量作为该 MOV 指令的结果变量，然后用该临时变量去替换原 MUL 指令中的左操作数，最后把这两条指令依次添加到指令集合中。此外，对于这两条指令中涉及的变量，我们也要依次为其设置一个序号，并把生成的序号-变量对依次添加到 irVariables 中（注意将结果变量排在操作数变量之前，这样结果变量就永远不会去抢夺与它同属一条指令的操作数变量的寄存器，从而避免后续目标代码生成时操作数变量因为寄存器被抢而找不到自身对应的寄存器）。

```
} else if (instruction.getKind() == InstructionKind.MUL && instruction.getLHS().isImmediate()){
    IRImmediate immediate = (IRImmediate) instruction.getLHS();
    IRVariable temp = IRVariable.temp();
    instructions.add(Instruction.createMov(temp, immediate));
    irVariables.put(i++, temp);
    instructions.add(Instruction.createMul(instruction.getResult(), temp, instruction.getRHS()));
    irVariables.put(i++, instruction.getResult());
    irVariables.put(i++, temp);
    irVariables.put(i++, (IRVariable) instruction.getRHS());
```

二、该指令为 MUL 指令且其右操作数为立即数，那么与上一种情况同理，我们需要生成一条以其右操作数为数据来源的 MOV 指令，并新声明一个临时变量作为该 MOV 指令的结果变量，然后用该临时变量去替换原 MUL 指令中的右操作数，最后把这两条指令依次添加到指令集合中。此外，对于这两条指令中涉及的变量，我们也要依次为其设置一个序号，并把生成的序号-变量对依次添加到 irVariables 中。（代码类似）
三、该指令为 SUB 指令且其左操作数（也就是被减数）为立即数，那么我们就需要生成一条以其左操作数为数据来源的 MOV 指令，并新声明一个临时变量作为该 MOV 指令的结果变量，然后用该临时变量去替换原 SUB 指令中的左操作数，最后把这两条指令依次添加到指令集合中。此外，对于这两条指令中涉及的变量，我们也要依次为其设置一个序号，并把生成的序号-变量对依次添加到 irVariables 中。

```
} else if (instruction.getKind() == InstructionKind.SUB && instruction.getLHS().isImmediate()) {
    IRImmediate immediate = (IRImmediate) instruction.getLHS();
    IRVariable temp = IRVariable.temp();
    instructions.add(Instruction.createMov(temp, immediate));
    irVariables.put(i++, temp);
    instructions.add(Instruction.createSub(instruction.getResult(), temp, instruction.getRHS()));
    irVariables.put(i++, instruction.getResult());
    irVariables.put(i++, temp);
    irVariables.put(i++, (IRVariable) instruction.getRHS());
```

四、该指令的左操作数为立即数且其既不是 MUL 指令也不是 SUB 指令（那在本实验中它只可能是 ADD 指令），那么我们只需要交换其左操作数与右操作数的位置即可（将立即数换到右操作数的位置上），同时，对于这条指令中的结果变量和原右操作数位置上的变量，我们也应依次为其设置一个序号，然后把生成的序号-变量对依次添加到 irVariables 中。

```
} else if (instruction.getLHS().isImmediate()) {
    //只可能是ADD指令
    instructions.add(Instruction.createAdd(instruction.getResult(), instruction.getRHS(), instruction.getLHS()));
    irVariables.put(i++,instruction.getResult());
    irVariables.put(i++, (IRVariable) instruction.getRHS());
```

如果以上情况都不符合，就说明该指令要么是 ADD/SUB 指令且只有右操作数是立即数，要么不含立即数，显然，这两种情况都不需要进行预处理，那么我们直接把该指令添加进指令集合即可，当然，对于指令中引用的变量，我们还是要依次为其设置一个序号，然后把生成

的序号-变量对依次添加到 irVariables 中。

```
} else {
    instructions.add(instruction);
    irVariables.put(i++, instruction.getResult());
    irVariables.put(i++, (IRVariable) instruction.getLHS());
    if (instruction.getRHS().isIRVariable()) {
        irVariables.put(i++, (IRVariable) instruction.getRHS());
    }
}
```

对于返回指令，我们除了要把它添加进指令集合、把可能出现的变量添加到次序表中以外，还应在此处直接终止对中间指令的遍历，以体现程序的返回。

```
} else if (instruction.getKind().isReturn()) {
    instructions.add(instruction);
    if (instruction.getReturnValue().isIRVariable()) {
        irVariables.put(i++, (IRVariable) instruction.getReturnValue());
    }
    break;
```

对于一元指令（有返回值且有一个参数），我们只需要把添加进指令集合、把它里面可能出现的变量依次添加到次序表中即可。

```
} else {
    instructions.add(instruction);
    irVariables.put(i++, instruction.getResult());
    if (instruction.getFrom().isIRVariable()) {
        irVariables.put(i++, (IRVariable) instruction.getFrom());
    }
}
```

加载并处理完中间代码后，我们就要开始执行寄存器分配与目标代码生成。对此，我们需要遍历在上一步中已经准备好的中间指令集合，然后根据每条指令的类型，在其对应的目标代码字符串 code 中率先加入关于其对应的汇编指令类型的描述符；

```
for (final var instruction : instructions) {
    code = "    ";
    switch (instruction.getKind()) {
        case MOV :
            if (instruction.getFrom().isImmediate()) {
                code += "li";
            } else {
                code += "mv";
            }
            break;
```

（以 MOV 指令为例，如果其操作数为立即数，则它对应的应该是汇编中的 li 指令；如果其操作数为变量，那么它对应的就是汇编中的 mv 指令）

之后，我们需要确定汇编指令中的目标寄存器：对于二元指令和一元指令，如果其结果变量不在 regAlloc 中，也就是说它还没有对应的寄存器，那么我们就需要为它分配一个寄存器：首先，遍历所有的临时寄存器，如果有还未被占用的寄存器，就将其分配给上述变量，并把该寄存器与变量一起添加到 regAlloc 中，形成一组双向映射，同时，将标志变量 flag 置为 1，然后退出遍历；

```
if (instruction.getKind().isBinary() || instruction.getKind().isUnary()) {
    if (!regAlloc.containsKey(instruction.getResult())) {
        flag = 0;
        for (final var reg : regs) {
            if (!regAlloc.containsValue(reg)) {
                regAlloc.replace(instruction.getResult(), reg);
                flag = 1;
                break;
            }
        }
```

如果遍历完所有的临时寄存器都没找到一个尚未被占用的寄存器（即 flag=0），就说明此时已经没有空闲寄存器了，那么我们就需要为该变量抢夺一个后续不再使用的变量占用的寄存器，具体来说，就是要遍历所有的临时寄存器，找到占用各个寄存器的变量，然后在存储各变量被引用的次序信息的哈希表 irVariables 中，从当前指令的结果变量对应的序号开始，寻找后续被引用的变量中有没有占用当前寄存器的那个变量，如果有，就说明该变量后续还会被引用，那么我们就置使用标志 use 为 1，然后利用 use 跳过当前正在考察的这个寄存器，继续对下个寄存器的占用变量进行考察；如果没有，就说明该变量后续不再被引用，那么我们直接把它占用的寄存器重新分配给当前指令的结果变量即可。

```
if (flag == 0) {
    //没有空闲寄存器
    for (final var reg : regs) {
        use = 0;
        IRVariable irVariable = regAlloc.getByValue(reg);
        for (int j = i; j < irVariables.size(); j++) {
            if (irVariables.get(j).equals(irVariable)) {
                use = 1;
                break;
            }
        }
        if (use == 0) {
            //该变量后续不再使用
            regAlloc.replace(instruction.getResult(), regAlloc.getByKey(irVariable));
            break;
        }
    }
}
```

分配完寄存器后，将结果变量对应的寄存器的名字作为汇编指令的目标寄存器添加到目标代码中，同时令变量计数器加一，以示我们此时已经使用/处理了 i 次变量（设置 i 就是为了确定在当前指令之后还有哪些变量会被引用）。

```
code += regAlloc.getByKey(instruction.getResult()).name();
i++;
```

对于返回指令，我们直接把目标寄存器设置为 a0 即可。

```
} else {
    code += "a0";
}
```

确定了目标寄存器之后，我们就要对操作数进行处理：对于二元指令，因为经过预处理后，其左操作数必定是一个变量，且对于这些作为操作数的变量，我们必然已经为其分配过寄存器了，所以我们直接找到指令左操作数对应的寄存器，将其添加到目标代码中即可（同时记得令变量计数器加一）。至于指令的右操作数，我们需要先对其类型进行判断：如果它是立即数，就直接将该立即数添加到目标代码中；反之，就将它占用的寄存器的名字添加到目标代码中，同时令变量计数器加一。

```
if (instruction.getKind().isBinary()) {
    IRVariable irVariable = (IRVariable) instruction.getLHS();
    code += regAlloc.getByKey(irVariable).name();
    i++;
    code += ", ";
    if (instruction.getRHS().isImmediate()) {
        IRImmediate immediate = (IRImmediate) instruction.getRHS();
        code += immediate.toString();
    } else {
        irVariable = (IRVariable) instruction.getRHS();
        code += regAlloc.getByKey(irVariable).name();
        i++;
    }
```

对于一元指令（即 MOV 指令），因为我们无法确定其操作数的类型，所以也要进行一次判断：如果这个操作数是立即数，就直接将该立即数添加到目标代码中；反之，就将它占用的寄存器的名字添加到目标代码中，同时令变量计数器加一。

```
} else if (instruction.getKind().isUnary()) {
    if (instruction.getFrom().isImmediate()) {
        IRImmediate immediate = (IRImmediate) instruction.getFrom();
        code += immediate.toString();
    } else {
        IRVariable irVariable = (IRVariable) instruction.getFrom();
        code += regAlloc.getByKey(irVariable).name();
        i++;
    }
```

同样，对于返回指令，因为我们也无法确定其要返回的数的类型，所以也要进行一次判断：如果它要返回的数是立即数，就直接将该立即数添加到目标代码中；反之，就将这个返回值占用的寄存器的名字添加到目标代码中，同时令变量计数器加一。

```
} else {
    if (instruction.getReturnValue().isImmediate()) {
        IRImmediate immediate = (IRImmediate) instruction.getReturnValue();
        code += immediate.toString();
    } else {
        IRVariable irVariable = (IRVariable) instruction.getReturnValue();
        code += regAlloc.getByKey(irVariable).name();
        i++;
    }
}
```

处理完操作数后，在每一行目标代码的最后加上关于其对应的中间指令的注释，然后把用来存目标代码的字符串 code 添加到集合 assemblyCode 中去即可。

最后，实现 dump 方法，利用 FileUtils 类中的方法将生成的目标代码（存储在字符串集合 assemblyCode 中）输出到指定文件。

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

# 4 实验结果与分析

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

**词法分析：**

输入：

源代码：

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

输出：

token 串：

```
(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,)
($,)

```

原始符号表：

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

```

说明：词法分析阶段的输入是一段 C 语言代码，输出则是分析这段代码得到的一系列 token，以及一个用来存放这段代码中声明的变量但还未确定各变量的具体类型的原始符号表。显然，该 token 串对应了源代码中的每一个单词，该符号表也对应了源代码中声明的四个变量 a、b、c、result。

**语法分析：**

输入：

词法分析阶段生成的 token 串：

```
(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,)
($,)

```

拓广文法：

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

LR(1)分析表：（由拓广文法生成）

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

输出：

归约时使用的产生式序列：

```
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

```

说明：语法分析阶段的输入是词法分析阶段生成的 token 串、拓广文法以及根据拓广文法生成的 LR(1)分析表，输出是语法分析器进行归约时使用的产生式的序

列。显然，将输出的产生式序列逆过来就是源程序段在拓展文法下对应的最右推导，符合预期。

**语义分析：**

输入：（同语法分析阶段，因为语法分析和语义分析是同时进行的）

词法分析阶段生成的 token 串：

```
(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,)
($,)

```

拓广文法：

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

LR(1)分析表：（由拓广文法生成）

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

输出：

更新后的符号表：

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

```

说明：语义分析阶段的输入是词法分析阶段生成的 token 串、拓广文法以及根据拓广文法生成的 LR(1)分析表，输出是更新了标识符类型后的符号表。语义分析后，各标识符符号的 type 属性值都由 null 变成了它们原本的类型 Int，符合预期。

**中间代码生成：**

输入：（同语法分析阶段，因为语法分析、语义分析以及中间代码生成都是同时进行的）

词法分析阶段生成的 token 串：

```
(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,)
($,)

```

拓广文法：

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

LR(1)分析表：（由拓广文法生成）

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

输出：

中间代码：

```
(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)

```

模拟执行中间代码后得到的结果：

```
144

```

说明：中间代码生成阶段的输入是词法分析阶段生成的 token 串、拓广文法以及根据拓广文法生成的 LR(1)分析表，输出是生成的中间代码，以及模拟执行中间代码后得到的结果。经计算，可以发现源程序最终返回的变量 result 的值就等于我们模拟执行中间代码后得到的运算结果 144（result = a * b - ( 3 + b ) * ( c - a )，其中，a = 8，b = 5，c = 3 - a = -5），符合预期。

前三次实验的正确性验证：

```
Diffing lab1 output:
Diffing file token.txt:
The src file is the same as std file.
Diffing file old_symbol_table.txt:
The src file is the same as std file.

Diffing lab2 output:
Diffing file parser_list.txt:
The src file is the same as std file.

Diffing lab3 output:
Diffing file ir_emulate_result.txt:
The src file is the same as std file.
Diffing file new_symbol_table.txt:
The src file is the same as std file.
```

**目标代码生成：**

输入：

前一阶段生成的中间代码：

```
(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：

汇编代码：

```
.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)
```

说明：目标代码生成阶段的输入是中间代码生成阶段生成的中间代码，输出是用汇编语言编写的目标代码。这段汇编代码在 rars 中的运行结果如下图所示，可以看到，整个程序运行完后，寄存器 a0 中存的值是 0x00000090，即 144，说明该汇编程序的输出（返回值）就是 144，符合预期。