

## 1 实验目的与方法

实验目的：实现一个 Java 实现的 TXTv2 语言编译器，目标平台为 RISC-V 32(指令集 RV32M)；深入了解编译程序实现原理及方法。

### 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。

实验环境： Java 17； IntelliJ IDEA 2022.2.2

### 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。

实验环境： Java 17； IntelliJ IDEA 2022.2.2；编译工作台

### 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。

实验环境： Java 17； IntelliJ IDEA 2022.2.2

### 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。

实验环境： Java 17； IntelliJ IDEA 2022.2.2

## 2 实验内容及要求

### 2.1 词法分析器

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。

1. 输入：以文件形式存放的类 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。

### 2.2 语法分析

编写一个语法分析程序，读取实验一获取的 token 表，对文件内的类 C 语言程序段进行语法分析。

1. 利用自底向上的 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；
3. 支持变量声明、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

完成语义分析及中间代码生成的观察者模式代码，在实验二自底向上语法分析文法的基础上实现一个 S-SDD 的 SDT，为语法正确的单词串设计翻译方案，完成语法制导翻译。

1. 利用该翻译方案，对所给程序段进行分析，输出生成的中间代码序列和更新后的符号表，并保存在相应文件中，中间代码使用三地址码的四元式表示。
2. 实现声明语句、简单赋值语句、算术表达式的语义分析与中间代码生成。
3. 使用框架中的模拟器 IREmulator 验证生成的中间代码的正确性。

### 2.4 目标代码生成

完成目标代码生成程序。

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。

### 3 实验总体流程与函数功能描述

#### 3.1 词法分析

##### 2.1.1. 编码表

无更多功能,故采用了实验 template 项目中自带的`coding\_map`文件中的编码表,如下所示:

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

##### 2.1.2. 正则文法

$G=(V,T,P,S)$ , 其中  $V=\{S,A,B,Cdigit,no\_0\_digit,char\}$ ,  $T=\{\text{任意符号}\}$ ,  
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,\_

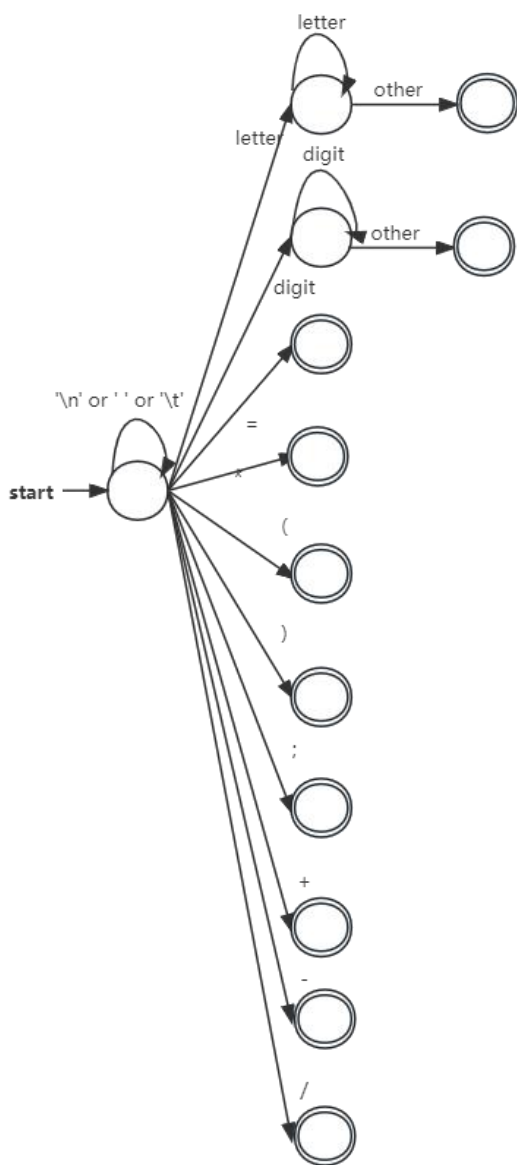
则 P 定义如下:

标识符:  $S \rightarrow \text{letter } A$ ;  $A \rightarrow \text{letter } A | \text{digit } A | \varepsilon$ ;

整常数:  $S \rightarrow no\_0\_digit B$ ;  $B \rightarrow \text{digit } B | \varepsilon$ ;

运算符:  $S \rightarrow C$ ;  $C \rightarrow = | * | + | - | /$

### 2.1.3. 状态转换图



### 2.1.4. 词法分析程序设计思路和算法描述

设计思路：根据状态转移图，通过状态机实现词法分析程序。

算法描述：

1. 设置初始状态为 0；
2. 读入文件一个字符，根据字符情况进行状态转移；
3. 若转移到终结状态，则向 `token` 表中新增一条符号记录，然后转移到状态 0；
4. 重复过程 2、3，直到读到文件终止字符。

部分代码如下：

```
public void run() {
    for (String line : file_lines) { // 读取文件
        State state = State.ZERO;
        StringBuilder sb = new StringBuilder();
        for (char c : line.toCharArray()) {
            switch (state) { // 状态转移
                case FORTEEN:
                    if ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z') || (c >= '0' && c <= '9')) {
                        state = State.FORTEEN;
                        sb.append(c);
                    } else { // 遇到终结符则进行符号表写入，并转移到状态 0
                        state = State.ZERO; // 转移到状态 0
                        if (sb.toString().equals("int")) { // 遇到终结符则进行符号表写入
                            tokens.add(Token.normal(TokenKind.fromString("int"), ""));
                        } else if (sb.toString().equals("return")){
                            tokens.add(Token.normal(TokenKind.fromString("return"), ""));
                        } else {
                            tokens.add(Token.normal(TokenKind.fromString("id"),
                                sb.toString()));
                            symbolTable.add(sb.toString());
                        }
                        sb = new StringBuilder();
                    }
                    break;
                ... // 省略
            }
        }
    }

    tokens.add(Token.normal(TokenKind.eof(), "")); // 添加符号表结束标志
}
```

## 3.2 语法分析

### 3.2.1 拓展文法

无更多功能，故采用了实验 template 项目中自带的`grammer.txt`文件中的文法，如下所示：

```
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

使用编译工作台按照指导书步骤对`grammar.txt`文件中的文法进行了分析表的构建与导出，最终存放在项目的`data\in\LR1\_table.csv`中。

### 3.2.3 状态栈和符号栈的数据结构和设计思路

#### 1. 符号栈

- ① 数据结构: `Stack<Term> symbolStack = new Stack<>();`
- ② 设计思路: 由于为栈结构, 并且里面存放的为终结符和非终极符, 故采用了 Java 的 Stack 类, 并且将泛型设置为终结符和非终极符类的父接口 Term。

#### 2. 状态栈

- ① 数据结构: `Stack<Status> statusStack = new Stack<>();`
- ② 设计思路: 由于为栈结构, 并且里面存放的为 Status 类, 故采用了 Java 的 Stack 类, 并且将泛型设置为 Status。

#### 3. 词法单元存储

- ① 数据结构: `List<Token> buffer = new LinkedList<>();`
- ② 设计思路: 由于 buffer 可以视为队列结构的条带, 在初始化时填充队列, 之后也仅需从队头获取元素, 并且无需遍历操作, 故而采用增删元素较为方便、性能较佳的 LinkedList 来存储数据。

### 3.2.4 LR 驱动程序设计思路和算法描述

设计思路: 通过 LR 分析表和 LR 文法的分析逻辑进行语法分析。

算法描述:

1. 每次读取状态栈顶和输入首部字母, 进行查表。
  1. 若查出 sx, 则弹出首部字母, 将其及其状态 x 入栈。

2. 若查出  $rx$ ，用第  $x$  个产生式归约符号栈（出栈符号及其对应状态，入栈归约结果），查询左部符号对应 GOTO 表，将 GOTO 表对应状态  $x$  入状态栈。

2. 重复 1，直到得到  $acc$ 。

部分代码如下：

```
public void run() {
    while(true) {
        Status status = statusStack.peek(); // 状态栈头元素
        Token symbol = buffer.get(0); // 输入队列头元素
        Action action = status.getAction(symbol); // 查询 LR 分析表
        switch (action.getKind()) { // 根据查询结果进行状态转换
            case Reduce: // rx
                Production production = action.getProduction(); // 获取产生式
                int length = production.body().size(); // 弹出状态栈和符号栈对应元素
                for (int i = 0; i < length; i++) {
                    symbolStack.pop();
                    statusStack.pop();
                }
                symbolStack.push(production.head()); // 将左部压入符号栈
                statusStack.push(statusStack.peek().getGoto(production.head())); // 将 GOTO
                // 表查询结果压入状态栈
                break;

            case Shift: // sx
                symbolStack.push(symbol.getKind()); // 将输入队列头压入符号栈
                buffer.remove(0);
                statusStack.push(action.getStatus()); // 将 LR 表查询结果压入状态栈
                break;

            case Accept: // 语法分析成功
                System.out.println("Parser exits successfully!");
                return ;

            case Error: // 语法分析失败
                System.out.println("Parser exits with errors!");
                return ;

            default:
                System.out.println("Unknown action kind.");
                return ;
        }
    }
}
```

## 3.3 语义分析和中间代码生成

### 3.3.1 翻译方案

仅实现了符号表的更新完善，和对赋值语句、算术运算的中间代码生成，故采取了 PPT 中介绍的通过分析栈进行语义分析的方法。

#### 语义分析栈执行动作说明：

1、语义分析的结果是更新符号表，根据翻译方案，语义分析栈需要保存type属性；

2、自底向上的分析过程中Shift时将符号的type属性（从Token中获得）入栈；

3、当 $D \rightarrow int$ 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；

4、当 $S \rightarrow D \ id$ 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）；

### 3.3.2 语义分析和中间代码生成的数据结构

使用了单独的 type 栈和 IRValue 栈：

```
private Stack<IRValue> irStack = new Stack<>();  
private Stack<String> attributeStack = new Stack<>();
```

还有一个用于暂时存储中间代码的列表结构：

```
private List<Instruction> irs = new ArrayList<>();
```

### 3.3.3 语法分析程序设计思路和算法描述

设计思路：通过栈进行语义分析。

算法描述：由于我们使用基于自底向上语法分析+S-SDD 的语法制导翻译方案，故而只需在`whenShift`时将属性进栈，在归约（也即`whenReduce`）时根据产生式的类型进行符号表的更新 or 中间代码的生成。

在实验具体设计中，将符号表更新和中间代码生成拆开来实现，更具解耦性。

#### 1. 中间代码生成

由于我们只需关注对赋值语句和算术运算语句的实现，故而只需关注这几条语句的归约：



6	S -> id = E;
7	S -> return E;
8	E -> E + A;
9	E -> E - A;
10	E -> A;
11	A -> A * B;

## (1) case 6

此时是对赋值语句进行归约，需生成一个 MOV 中间代码，E 和 id 分别位于 stack[top]和 stack[top- 1]的位置：

```
IRValue irv = irStack.peek();
irStack.pop();
irs.add(Instruction.createMov(((IRVariable)irStack.peek()), irv));
irStack.pop();
break;
```

## (2) case 7

此时是对返回语句进行归约，需生成一个 Ret 中间代码，E 位于 stack[top]的位置：

```
irs.add(Instruction.createRet(irStack.peek()));
irStack.pop();
```

## (3) case 8

此时是对算术运算中的加法语句进行归约，需生成一个 ADD 中间代码，A 和 E 分别位于 stack[top]和 stack[top- 1]的位置，并且将归约结果入栈。注意，此时需要产生一个临时变量用于存储 A 和 E 的运算结果，并且需要再次将临时变量入栈：

```
IRValue lhs = irStack.peek();
irStack.pop();
IRValue rhs = irStack.peek();
irStack.pop();
IRVariable t = IRVariable.temp();
irs.add(Instruction.createAdd(t, rhs, lhs));
irStack.push(t);
break;
```

## (4) case 9

此时是对算术运算中的减法语句进行归约，需生成一个 SUB 中间代码，A 和 E 分别位于 stack[top]和 stack[top- 1]的位置，并且将归约结果入栈。注意，此时需要产生一个临时变量用于存储 A 和 E 的运算结果，并且需要再次将临时变量入栈：

```
IRValue lhs = irStack.peek();
```

```

    irStack.pop();
    IRValue rhs = irStack.peek();
    irStack.pop();
    IRVariable t = IRVariable.temp();
    irs.add(Instruction.createSub(t, rhs, lhs));
    irStack.push(t);
    break;

```

#### (5) case 11

此时是对算术运算中的乘法语句进行归约，需生成一个 MUL 中间代码，A 和 E 分别位于 stack[top]和 stack[top- 1]的位置，并且将归约结果入栈。注意，此时需要产生一个临时变量用于存储 A 和 E 的运算结果，并且需要再次将临时变量入栈：

```

    IRValue lhs = irStack.peek();
    irStack.pop();
    IRValue rhs = irStack.peek();
    irStack.pop();
    IRVariable t = IRVariable.temp();
    irs.add(Instruction.createMul(t, rhs, lhs));
    irStack.push(t);
    break;

```

## 2. 符号表更新

只需关注这几条语句的归约：

4	S -> D id;
5	D -> int;

#### (1) case 4

此时是对声明语句进行归约，需将符号表中 id 的类型更新为 D，简化实现起见默认为 INT 类型，并且出栈：

```

    if (symbolTable.has(attributeStack.peek())) {
        symbolTable.get(attributeStack.peek()).setType(SourceCodeType.Int);
    }
    attributeStack.pop();

```

#### (2) case 5

此时是对类型进行归约，简化实现起见默认为 INT 类型，所以直接出栈：

```

    attributeStack.pop();

```

## 3.4 目标代码生成

### 3.4.1 设计思路和算法描述

设计思路：将上述实验的结果整合出来的中间代码转化为 riscv 代码。仅实现了不完备的寄存器选择。

算法描述：

#### 1. 预处理

在转化中间代码之前，我进行了两个预处理操作：

##### (1) 调整指令中对操作寄存器的操作次序

根据 riscv 指令集的语法规则，对中间代码进行调整。调整目标：

指令	指令标准格式
MOV	MOV reg1, reg2/imm
ADD	ADD reg1, reg2/imm
SUB	ADD reg1, reg2/imm
MUL	MUL reg1, reg2
RET	RET reg1/imm

其中 MOV 和 RET 指令无需特别处理，因而我们重点在对 ADD、SUB 以及 MUL 指令的处理。

首先，如果两个操作数都是立即数，那我们就直接计算出计算结果，并将该指令转化为一个 MOV 指令：

```
if (itr.getLHS().isImmediate() && itr.getRHS().isImmediate()) {
    itr.add(Instruction.createMov(itr.getResult(),
        IRImmediate.of(Integer.parseInt(itr.getRHS().toString())
            * Integer.parseInt(itr.getLHS().toString()))));
}
```

除了上面这个情况外的其他情况，我们需要针对这三个指令进行不同的处理：

#### ① ADD

若第一个操作数（lhs）为立即数，我们只需调换两个操作数位置即可：

```
else if (itr.getLHS().isImmediate() || itr.getRHS().isImmediate()) {
    IRValue var = itr.getLHS().isImmediate() ? itr.getRHS() : itr.getLHS();
    IRValue imm = itr.getLHS().isImmediate() ? itr.getLHS() : itr.getRHS();
    itr.add(Instruction.createAdd(itr.getResult(), var, imm));
}
```

#### ② SUB

若第一个操作数（lhs）为立即数，我们需要新增一个临时变量，用于存放该立即数，将 SUB 指令转化为对两个寄存器的操作：

```
else if (itr.getLHS().isImmediate()) {
    // change to a two-register calculation
    IRVariable t = IRVariable.temp();
    itr.add(Instruction.createMov(t, itr.getLHS()));
```

```
itr.add(Instruction.createSub(itr.getResult(), t, itr.getRHS()));
}
```

### ③ MUL

riscv 不提供含立即数的 mul 指令，故而如果两个操作数中如果有立即数，就都需要通过增加 MOV 指令和临时变量的方法将 MUL 指令转化为三寄存器运算：

```
else if (itr.getLHS().isImmediate() || itr.getRHS().isImmediate()) {
    // change to a three-register calculation
    if (itr.getLHS().isImmediate()) {
        IRVariable t = IRVariable.temp();
        itr.add(Instruction.createMov(t, itr.getLHS()));
        itr.add(Instruction.createMul(itr.getResult(), t, itr.getRHS()));
    } else {
        IRVariable t = IRVariable.temp();
        itr.add(Instruction.createMov(t, itr.getRHS()));
        itr.add(Instruction.createMul(itr.getResult(), itr.getLHS(), t));
    }
}
```

### (2) 记录变量的使用信息，以供之后转化使用

在处理完中间代码后，就需要记录所有变量的 refcnt 等信息，以供之后寄存器选择淘汰算法利用。

为了方便查询修改，我设计了如下数据结构：

```
class RegisterInfo {
    public int getRegisterNo();
    public void setRegisterNo(int registerNo);

    public void IncreaseRefcnt();
    public void DecreaseRefcnt();
    public int getRefcnt();
};

Map<IRVariable, RegisterInfo> variables = new HashMap<>();
Map<Integer, Boolean> registers = new HashMap<>();
```

'RegisterInfo'用于代表某一变量的寄存器信息，包括其目前占用的寄存器，以及其 refcnt 两个信息。通过一个 Map<IRVariable, RegisterInfo>来组织该信息。

同时，为了在寄存器选择算法中方便获取空闲寄存器，我设计了 Map<Integer, Boolean> registers 来记录寄存器的信息。

在编写代码中，需要保证这两个字段的一致性。

在预处理过程中，我们需要遍历所有预处理完之后的指令，记录指令中涉及到变量的引用信息（在此仅以指令中 result 位变量的记录为例）：

```
if (variables.containsKey(itr.getResult())) {
    variables.get(itr.getResult()).IncreaseRefcnt();
} else {
    variables.put(itr.getResult(), new RegisterInfo());
}
```

## 2. 寄存器选择

由于为非完备寄存器选择，故而实现思路较为简单。我们提供一两个对外接口，`processInvalid` 和 `getRegister`。

### ① `processInvalid`

在每次转化完一条指令都对其进行调用，用于集中处理那些引用次数为 0 的变量，释放其寄存器。

```
private void processInvalid() {
    for (Map.Entry<IRVariable, RegisterInfo> en
        : variables.entrySet()) {
        if (en.getValue().getRefcnt() == 0
            && en.getValue().getRegisterNo() != Integer.MAX_VALUE) {
            registers.put(en.getValue().registerNo, false); // release
            en.getValue().setRegisterNo(Integer.MAX_VALUE);
        }
    }
}
```

### ② `getRegister`

首先试图返回该变量已经占用的寄存器；若没有，寻找一个空闲寄存器并占用。

```
private int getRegister(IRValue result) {
    int registerNo = variables.get(result).getRegisterNo();
    if (registerNo != Integer.MAX_VALUE) {
        variables.get(result).DecreaseRefcnt();
        return registerNo;
    }

    // a steal
    for (Map.Entry<Integer, Boolean> en : registers.entrySet()) {
        if (en.getValue()) continue; // is occupied
        registerNo = en.getKey();
        variables.get(result).setRegisterNo(en.getKey());
        break;
    }

    if (registerNo == Integer.MAX_VALUE)
        throw new RuntimeException();
    registers.put(registerNo, true);
    variables.get(result).DecreaseRefcnt();
    return registerNo;
}
```

## 3. 代码生成

经过预处理之后，中间代码的语义已经十分明确。生成过程需要做的就是根据指令的类型进行简单的字符串操作、调用上面两个寄存器选择函数即可，不做赘述。

## 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。

### 4.1 词法分析

输入：`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：符号表和 token list。形式化打印后如下。

token list:

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

符号表：（还没进行语义分析，所以类型字段为 **null**）

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

## 4.2 语法分析

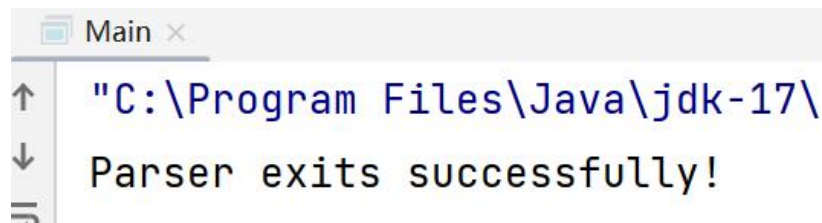
输入：`input\_code.txt` 【PPT 示例】

```
int a;  
int c;  
c = 3-a;
```

输出：语法分析过程使用的产生式列表，以及语法分析成功的控制台输出。形式化打印后如下。

```
D -> int  
S -> D id
```

```
D -> int
S -> D id
B -> IntConst
A -> B
E -> A
B -> id
A -> B
E -> E - A
S -> id = E
S_list -> S Semicolon
S_list -> S Semicolon S_list
S_list -> S Semicolon S_list
P -> S_list
```



```
↑ "C:\Program Files\Java\jdk-17\  
↓ Parser exits successfully!
```

### 4.3 语义分析

输入：语义分析通过观察者模式，绑定在了语法分析的实现过程中，故而输入与语法分析相近，外加一个符号表。

输出：更新后的符号表和生成的中间代码，还有通过 IREmulator 的代码执行结果。

符号表：（经过语义分析，已经填充了类型字段）

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

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)
```



中间代码执行结果:

144

## 4.4 目标代码生成

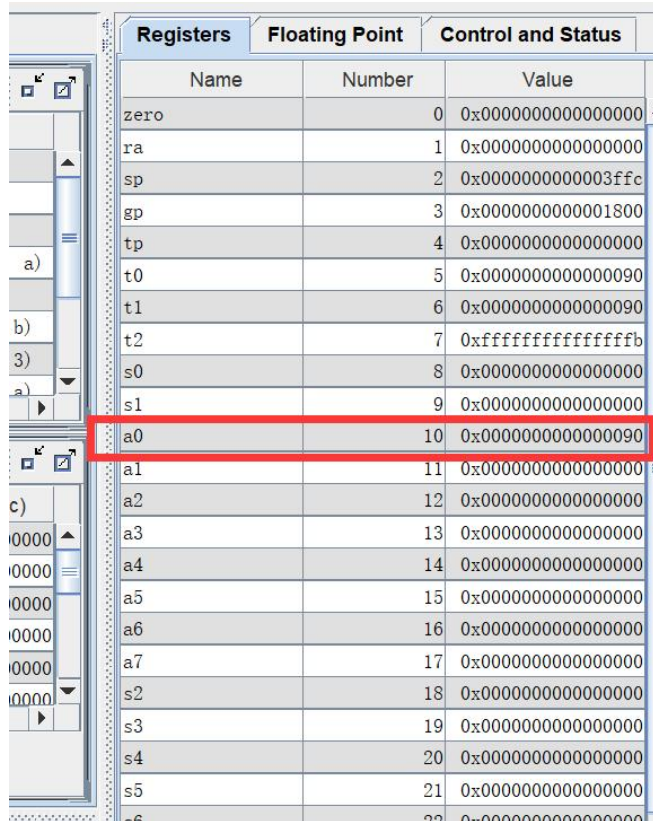
输入: 中间代码

输出: riscv 代码, 以及 RARS 的运行结果

riscv 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 t2, t3      # (MOV, c, $0)
    mul t3, t0, t1 # (MUL, $1, a, b)
    addi t4, t1, 3 # (ADD, $2, b, 3)
    sub t1, t2, t0 # (SUB, $3, c, a)
    mul t0, t4, t1 # (MUL, $4, $2, $3)
    sub t1, t3, t0 # (SUB, $5, $1, $4)
    mv t0, t1      # (MOV, result, $5)
    mv a0, t0      # (RET, , result)
```

RARS 运行结果:



Registers			Floating Point	Control and Status
Name	Number	Value		
zero	0	0x0000000000000000		
ra	1	0x0000000000000000		
sp	2	0x0000000000003ffc		
gp	3	0x0000000000001800		
tp	4	0x0000000000000000		
t0	5	0x0000000000000090		
t1	6	0x0000000000000090		
t2	7	0xfffffffffffffffffb		
s0	8	0x0000000000000000		
s1	9	0x0000000000000000		
a0	10	0x0000000000000090		
a1	11	0x0000000000000000		
a2	12	0x0000000000000000		
a3	13	0x0000000000000000		
a4	14	0x0000000000000000		
a5	15	0x0000000000000000		
a6	16	0x0000000000000000		
a7	17	0x0000000000000000		
s2	18	0x0000000000000000		
s3	19	0x0000000000000000		
s4	20	0x0000000000000000		
s5	21	0x0000000000000000		
s6	22	0x0000000000000000		

## 5 实验中遇到的困难与解决办法

描述实验中遇到的困难与解决办法，对实验的意见与建议或收获。

### 5.1 困难与解决办法

整个实验过程没有太难以解决的困难，PPT 和授课都很清楚，不清楚的地方也可以通过学习理论课补充。

每个实验具体耗时如下：

lab1	3h
lab2	2h
lab3	4h
lab4	2h

### 5.2 收获与建议

实验设计很好很现代，让我亲自动手从零开始实现了一个简易的编译器，与理论课程结合十分紧密，让我掌握了编译器内部实现的大体面貌，也拉起了我对编译原理的学习兴趣。

美中不足的是感觉课时安排有点奇怪，比如实验二内容偏简单，但课时太长了，导致中间很长一段时间没碰编译原理，再做实验三的时候全都忘光了就学了很久（）

还有一点小小建议，感觉指导书可以再加点拓展阅读内容，大概介绍一下当前前沿的技术是怎么处理该阶段的，权当一个小科普。