

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: 吕弋卿
学 号: 210110315
专 业: 计算机科学与技术
日 期: 2023-11-19

1.实验目的与方法

总体实验目的：使用Java语言实现类C语言编译器，目标平台为RISC-V 32(指令集RV32M)

实验方法：

编程语言：Java

Jdk版本：Oracle OpenJDK version 19.0.2

集成环境：IntelliJ IDEA 2022.3.3 RC (Ultimate Edition)

其它：编译工作台、RARS

1.1.词法分析器

实验目的：使用自动机实现词法分析器，将输入的源程序文本转换为Token序列，并将发现的标识符加入符号表中。

- 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入的认识，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。

实验方法：Java

1.2.语法分析

实验目的：使用LR(1)分析法对词法分析输出的Token串竞选语法分析，并输出产生式序列。

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

实验方法：Java，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：对语法分析的输出结果设计翻译方案，完成语法制导翻译。

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

实验方法：Java

1.4.目标代码生成

实验目的：将中间代码转换为目标代码(RISC-V指令)

- 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。

实验方法：Java、RARS

2.实验内容及要求

2.1.词法分析器

实验内容：编写一个词法分析程序，读取码点文件和源程序文件，对类C语言程序进行词法分析，输出二元组序列，并将发现的标识符加入符号表中。

实验要求：

- 1.词法分析程序输入：以文件形式存放自定义的类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

实验内容：利用编译工作台生成LR(1)分析表，使用LR(1)分析法设计自底向上的语法分析程序，对词法分析器输出的Token序列进行语法分析，输出产生式序列至文件中。

实验要求：

完成对实验模板代码中支持变量声明、变量赋值、基本算术运算的文法的语法分析。

2.3.典型语句的语义分析及中间代码生成

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

实验要求：

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

2.4.目标代码生成

实验内容：将前面生成的中间代码转换为目标代码(RISC-V指令)至文件中，并运行以验证结果。

实验要求：

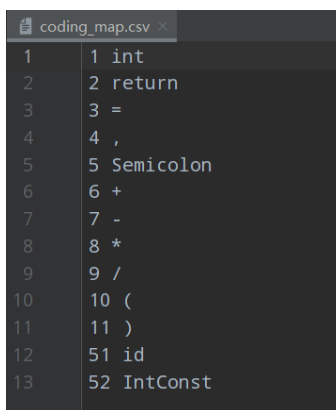
使用RARS运行out/assembly_language.asm, 验证返回结果是否正确。

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

在特定编程语言中，关键字、运算符、分界符都是语言预先定义的，标识符、常数则由程序设计人员按照一定规则定义。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）。因此需要编码表(coding_map.csv)来规定这种对应关系。



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

3.1.2.正则文法

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，正则文法为：

$G=(V,T,P,S)$,其中 $V=\{S,A,B,C,digit,no_0_digit,letter\}$, $T=\{\text{任意符号}\}$, P 定义如下：

digit表示数字0,1,...,9; no_0_digit表示1,2,...,9; letter表示A,B,...,Z,a,b,...,z,_

标识符 : $S \rightarrow letter A \quad A \rightarrow letter A | digit A | \epsilon$

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

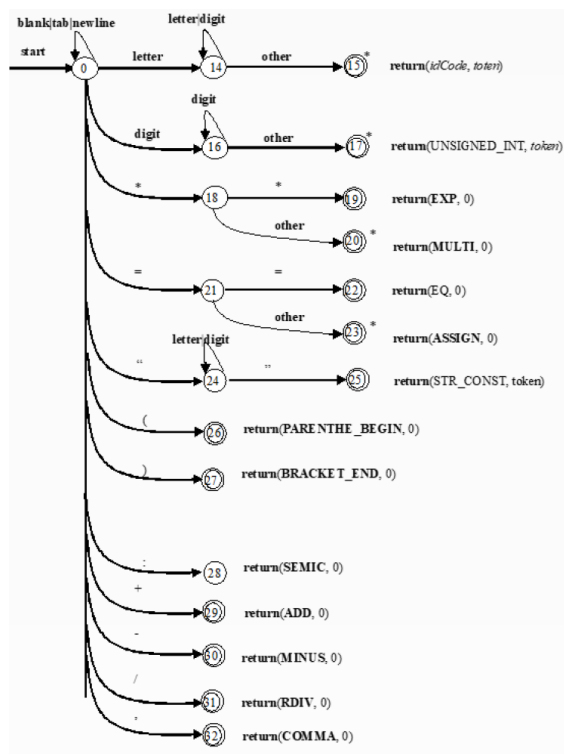
整常数 : $S \rightarrow no_0_digit B \quad B \rightarrow digit B | \epsilon$

字符串常量 : $S \rightarrow "C"$

字符常量 : $S \rightarrow 'D'$

3.1.3.状态转换图

词法分析器DFA的状态转化图如下：



3.1.4.词法分析程序设计思路和算法描述

```
// 符号表
private final SymbolTable symbolTable;
// token序列
private final List<Token> tokenList = new ArrayList<>();
// 输入源代码，初始为空，读取文件时追加
private String inputCode = "";
/**
 * 状态机状态，对应DFA状态转换图
 * Status: INITIAL      状态0
 * Status: IDENTIFIER   读入第一个字符为letter，对应状态14，预计为标识符
 * Status: INTEGER      读入第一个字符为digit，对应状态16，预计为整常数
 */
public enum IterateStatus{INITIAL, IDENTIFIER, INTEGER};
// 构造函数：初始化符号表
public LexicalAnalyzer(SymbolTable symbolTable) {
    this.symbolTable = symbolTable;
}
```

上图为LexicalAnalyzer类中的属性以及构造函数，介绍见图中注释。

```
private boolean isLetter(char ch) {
    return (ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z');
}

private boolean isDigit(char ch) {
    return (ch >= '0' && ch <= '9');
}

private boolean isDelimiter(char ch){
    return (ch == '\n' || ch == ' ' || ch == '\t');
}
```

上图为三个自定义的功能函数，分别用来识别字母，数字，空格、换行等分界字符。

```

/**
 * 从给予的路径中读取并加载文件内容
 *
 * @param path 路径
 */
public void loadFile(String path) {
    // TODO: 词法分析前的缓冲区实现
    try (BufferedReader reader = new BufferedReader(new FileReader(path))) {
        String line;
        while ((line = reader.readLine()) != null) {
            inputCode += line;
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}

```

loadFile()方法，逐行从源文件中读入代码，并追加至字符串inputCode中。

```

/**
 * 执行词法分析，准备好用于返回的 token 列表 <br>
 * 需要维护实验一所需的符号表条目，而得在语法分析中才能确定的符号表条目的成员可以先设置为 null
 */
public void run() {
    // TODO: 自动机实现的词法分析过程
    IterateStatus currentStatus = IterateStatus.INITIAL;
    StringBuilder bufferedString = new StringBuilder();
    char ch;
    for(int i = 0; i < inputCode.length(); i++){
        ch = inputCode.charAt(i); // Character at the position i of the string
        System.out.println("-DEBUG- Current Status: " + currentStatus + ", Character = " + ch);
        switch (currentStatus){
            case INITIAL : {
                if(isDelimiter(ch)){ // Ignore delimiter
                    currentStatus = IterateStatus.INITIAL;
                }
            }
        }
    }
}

```

run()方法，此为前处理部分，每轮读取源代码中的一个字符送入DFA中，bufferedString用于记录当前读入的字符串。

```

System.out.println("-DEBUG- Current Status: " + currentStatus + ", Character = " + ch);
switch (currentStatus){
    case INITIAL : {
        if(isDelimiter(ch)){ // Ignore delimiter
            currentStatus = IterateStatus.INITIAL;
        }
        else if(isLetter(ch)){ // A token start with a letter
            bufferedString.append(ch);
            currentStatus = IterateStatus.IDENTIFIER;
        }
        else if(isDigit(ch)){ // A token start with a digit
            bufferedString.append(ch);
            currentStatus = IterateStatus.INTEGER;
        }
        else {
            switch (ch){
                case '=': tokenList.add(Token.normal(TokenKind.fromString("="), "")); break;
                case '(': tokenList.add(Token.normal(TokenKind.fromString("("), "")); break;
                case ')': tokenList.add(Token.normal(TokenKind.fromString(")"), "")); break;
                case '+': tokenList.add(Token.normal(TokenKind.fromString("+"), "")); break;
                case '-': tokenList.add(Token.normal(TokenKind.fromString("-"), "")); break;
                case '*': tokenList.add(Token.normal(TokenKind.fromString("*"), "")); break;
                case '/': tokenList.add(Token.normal(TokenKind.fromString("/"), "")); break;
                case ';': tokenList.add(Token.normal(TokenKind.fromString("Semicolon"), "")); break;
                default: break;
            }
            currentStatus = IterateStatus.INITIAL;
        }
    }
    break;
}
}

```

DFA：状态为INITIAL时，遇到delimiter则跳过，遇到字母则状态转移为IDENTIFIER，遇到数字则状态转移为INTEGER，遇到其他关键字、运算符则将其加入tokenList，并且状态仍为INITIAL。

```

case IDENTIFIER : {
    if(isLetter(ch) || isDigit(ch)){
        bufferedString.append(ch);
        currentStatus = IterateStatus.IDENTIFIER;
    }
    else {
        if (bufferedString.toString().equals("int")) {
            tokenList.add(Token.normal(TokenKind.fromString("int"), ""));
        }
        else if (bufferedString.toString().equals("return")){
            tokenList.add(Token.normal(TokenKind.fromString("return"), ""));
        }
        else {        // Identifier defined by user
            tokenList.add(Token.normal(TokenKind.fromString("id"), bufferedString.toString()));
            symbolTable.add(bufferedString.toString());
        }
        i--;        // Retrace the index for once
        bufferedString = new StringBuilder();
        currentStatus = IterateStatus.INITIAL;
    }
    break;
}
}

```

DFA：状态为IDENTIFIER时，若下个字符是字母或数字则追加至bufferedString状态仍为IDENTIFIER；若不是，则对bufferedString进行检查，若是关键字int或return则加入tokenList，若不是，则应为用户定义的标识符，将其加入符号表，在else块末尾需要进行三项操作：

- 1.由于向后读了一个字符，需要将输入指针i向前回溯一位。
- 2.将bufferedString清空
- 3.状态转移为INITIAL

```

case INTEGER : {
    if (isDigit(ch)) {
        currentStatus = IterateStatus.INTEGER;
        bufferedString.append(ch);
    }
    else {
        tokenList.add(Token.normal(TokenKind.fromString("IntConst"), bufferedString.toString()));
        i--;        // Retrace the index for once
        bufferedString = new StringBuilder();
        currentStatus = IterateStatus.INITIAL;
    }
    break;
}
default: break;
}

```

DFA：状态为INTEGER时，与IDENTIFIER的操作类似，不做赘述。

```

/**
 * 获得词法分析的结果，保证在调用了 run 方法之后调用
 *
 * @return Token 列表
 */
public Iterable<Token> getTokenList() {
    // TODO: 从词法分析过程中获取 Token 列表
    // 词法分析过程可以使用 Stream 或 Iterator 实现按需分析
    // 亦可以直接分析完整个文件
    // 总之实现过程能转化为一列表即可
    return tokenList;
}

```

getTokenList()方法：直接返回tokenList即可

3.2.语法分析

3.2.1.拓展文法

要使语法分析DFA中只有一个起始状态和终结状态，需要对文法进行拓展，在模板的grammar.txt中已经实现。

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

3.2.2.LR(1)分析表

将给出的grammar.txt按一定格式输入至编译工作台，得到LR(1)分析表如下：

状态	Action											goto							
	id	()	+	-	*	=	int	return	IntConst	Semicolon	\$	E	S_list	S	A	B	D	
0	shift 4							shift 5	shift 6			accept		1	2			3	
1											shift 7								
2																			
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											shift 15	reduce S => D id				11	12		
9	shift 13	shift 14											17						
10				shift 18	shift 19						shift 15	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	reduce S_list => S Semicolon S_list							
16												reduce S => id = E							
17				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	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	reduce E => E + A	shift 20					reduce E => E + A								
28			reduce E => E - A	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	shift 33					reduce A => A * B								
30			reduce B => (E)	reduce B => (E)	reduce B => (E)	reduce B => (E)					reduce B => (E)								
31	shift 24	shift 25								shift 26						35	23		
32	shift 24	shift 25								shift 26						36	23		
33	shift 24	shift 25								shift 26							37		
34			shift 38	shift 31	shift 32														
35			reduce E => E + A	reduce E => E + A	reduce E => E + A	shift 33													
36			reduce E => E - A	reduce E => E - A	reduce E => E - A	shift 33													
37			reduce A => A * B	reduce A => A * B	reduce A => A * B	shift 33													
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：给出的代码模板中的Status类已经实现了状态栈泛型类Stack<Status>的设计，这里不做赘述。

符号栈：由于符号栈中，既可以有终结符，也可以有非终结符，因此需要设计一个类(SymbolEntry)来作为Stack的泛型类型，这个类可以同时容纳Terminal和Nonterminal


```

1 package cn.edu.hitsz.compiler.parser;
2 import cn.edu.hitsz.compiler.lexer.Token;
3 import cn.edu.hitsz.compiler.parser.table.NonTerminal;
4
5 class SymbolEntry {
6     Token token;
7     NonTerminal variable;
8
9     private SymbolEntry(Token token, NonTerminal variable){
10         this.token = token;
11         this.variable = variable;
12     }
13
14     public SymbolEntry(Token token){
15         this(token, null);
16     }
17
18     public SymbolEntry(NonTerminal variable){
19         this(null, variable);
20     }
21
22     public Token getToken() {
23         return token;
24     }
25
26     public boolean isToken(){
27         return token != null;
28     }
29 }
30

```

下为状态栈和符号栈的定义：

```

1 private final Stack<Status> statusStack = new Stack<>();
2 private final Stack<SymbolEntry> symbolEntryStack = new Stack<>();

```

3.2.4.LR驱动程序设计思路和算法描述

```

private final SymbolTable symbolTable;
private final List<ActionObserver> observers = new ArrayList<>();
private final Queue<Token> tokenQueue = new LinkedTransferQueue<>();
private final Stack<Status> statusStack = new Stack<>();
private final Stack<SymbolEntry> symbolEntryStack = new Stack<>();
private LRTable lrTable;
public SyntaxAnalyzer(SymbolTable symbolTable) {
    this.symbolTable = symbolTable;
}

```

上图为SyntaxAnalyzer中的属性及构造函数，其中，tokenQueue为LR(1)分析过程中的输入缓冲区。

```

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
    Iterator<Token> tokenIterator = tokens.iterator();
    while(tokenIterator.hasNext()){
        Token token = tokenIterator.next();
        tokenQueue.add(token);
    }
    symbolEntryStack.push(new SymbolEntry(Token.eof()));
}

```

loadToken()方法：使用迭代器将词法分析得到的Token序列中的token逐个加入输入缓冲区tokenQueue。

```

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 initState 存起来使用
    lrTable = table;
    statusStack.push(lrTable.getInit());
}

```

loadLRTable()方法：初始化LR(1)分析表lrTable，并将初始状态压入状态栈statusStack中。

```
public void run() {
    // TODO: 实现驱动程序
    // 你需要根据上面的输入来实现 LR 语法分析的驱动程序
    // 请分别在遇到 Shift, Reduce, Accept 的时候调用上面的 callWhenInShift, callWhenInReduce, callWhenInAccept
    // 否则用于为实验二打分的产生式输出可能不会正常工作
    int actionTicker = 0;
    while (!tokenQueue.isEmpty()) { // 队列不为空
        actionTicker++;
        Token bufferedToken = tokenQueue.element();
        Status bufferedStatus = statusStack.peek();
        Action bufferedAction = lrTable.getAction(bufferedStatus, bufferedToken);
        switch (bufferdAction.getKind()){
            case Shift : {
```

run()方法：此为前处理部分，每次从输入缓冲中读取一个token，并读取当前状态栈顶的状态，调用LRTable中的getAction方法，获取当前应进行的动作并送入switch()语句。

```
        Action bufferedAction = lrTable.getAction(bufferedStatus, bufferedToken);
        switch (bufferdAction.getKind()){
            case Shift : {
                callWhenInShift(bufferedStatus, bufferedToken);
                statusStack.push(bufferdAction.getStatus());
                symbolEntryStack.push(new SymbolEntry(bufferedToken));
                tokenQueue.remove();
                System.out.println("-DEBUG- ACTION " + actionTicker + ": SHIFT, Rest Token = " + tokenQueue.size());
                break;
            }
```

switch()语句：当动作为Shift(移进)时，有以下动作：

- 1.调用callWhenShift()通知各观察者
- 2.将移进后应当转移到的状态压入状态栈statusStack中
- 3.将当前移进的符号压入符号栈symbolEntryStack中
- 4.从输入缓冲区tokenQueue中删去首字符

```
            case Reduce : {
                Production bufferedProduction = bufferdAction.getProduction();
                callWhenInReduce(bufferedStatus, bufferedProduction);
                Iterator<Term> bodyIterator = bufferedProduction.body().iterator();
                while (bodyIterator.hasNext()){
                    bodyIterator.next();
                    statusStack.pop();
                    symbolEntryStack.pop();
                }
                symbolEntryStack.push(new SymbolEntry(bufferedProduction.head()));
                statusStack.push(lrTable.getGoto(statusStack.peek(), bufferedProduction.head()));
                System.out.println("-DEBUG- ACTION " + actionTicker + ": REDUCE, Rest Token = " + tokenQueue.size());
                break;
            }
        }
```

switch()语句：当动作为Reduce(归约)时，有以下动作：

- 1.调用callWhenReduce()通知各观察者
- 2.获取当前用于归约的产生式bufferedProduction，并对其进行迭代，产生式右部有多少个字符，就要从状态栈和符号栈中弹出多少个元素。
- 3.将当前归约产生式的左部压入符号栈symbolEntryStack中
- 4.从LRTable中，根据当前状态栈顶和符号栈顶元素获取需要goto的状态，并将其压入状态栈statusStack中。

```
            case Accept : {
                callWhenInAccept(bufferedStatus);
                tokenQueue.remove();
                System.out.println("-DEBUG- ACTION " + actionTicker + ": ACCEPT, Rest Token = " + tokenQueue.size());
                break;
            }
            default: {
                break;
            }
        }
```

switch()语句：当动作为Accept(接收)时，调用callWhenAccept()通知各观察者即可。

3.3.语义分析和中间代码生成

3.3.1.翻译方案

模板中的产生式的语法制导翻译方案如下：

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

3.3.2.语义分析和中间代码生成的数据结构

语义分析：数据结构如下：其中symbolStack为符号栈，idTypeStack为语义栈

```
1 private final Stack<SymbolEntry> symbolStack = new Stack<>();
2 private final Stack<SourceCodeType> idTypeStack = new Stack<>();
```

中间代码生成：数据结构如下：symbolStack为符号栈，IRStack为语义栈，instructionList为生成的中间代码序列

```
1 private final Stack<SymbolEntry> symbolStack = new Stack<>();
2 private final Stack<IRValue> IRStack = new Stack<>();
3 private final List<Instruction> instructionList = new LinkedList<>();
```

3.3.3.语义分析程序设计思路和算法描述

语义分析：主要用于更新符号表SymbolTable

```
@Override
public void whenAccept(Status currentStatus) {
    // TODO: 该过程在遇到 Accept 时要采取的代码动作
    // Do nothing 遇到Accept时语义分析结束，无操作
}
```

whenAccept()方法：接收时语义分析无操作

```

@Override
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
    switch (production.index()) {
        case 4 -> { // S -> D id
            SymbolEntry id = symbolStack.pop();
            symbolStack.pop();
            idTypeStack.pop();
            symbolTable.get(id.token.getText()).setType(idTypeStack.pop()); // 将栈顶id的type设为与D对应的type
            idTypeStack.push(null); // 压入null占位
        }
        case 5 -> { // D -> int
            idTypeStack.pop();
            String varType = symbolStack.pop().getToken().getKindId();
            if(Objects.equals(varType, "int")){ // 如果栈顶token为int则向idTypeStack中压入int
                idTypeStack.push(SourceCodeType.Int);
            }
        }
        default -> { // 其他种类产生式，弹栈并压入null占位
            multipop(symbolStack, production.body().size());
            multipop(idTypeStack, production.body().size());
            idTypeStack.push(null);
        }
    }
    symbolStack.push(new SymbolEntry(production.head()));
}

```

whenReduce()方法：当语法分析遇到规约动作时，语义分析进行如下操作：

对使用产生式进行检查：

- 1.若为 $S \rightarrow D \text{ id}$ ，符号栈顶即为id，根据翻译方案，采取动作
 $\{p := \text{lookup}(\text{id.name}); \text{if } p \neq \text{nil then enter}(\text{id.name}, D.\text{type}) \text{ else error}\}$
- 2.若为 $D \rightarrow \text{int}$ ，根据翻译方案，采取动作 $\{D.\text{type} := \text{int}\}$
- 3.若为其他产生式，无需修改符号表，只需将符号栈、语义栈弹出相应个元素即可，并且由于符号栈中要压入产生式左部变量，因此语义栈中也要压入<null>来占位。

```

@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.push(new SymbolEntry(currentToken));
    idTypeStack.push(null);
}

```

whenShift()方法：遇到移进动作时，符号栈中压入符号，语义栈中压入<null>占位

```

@Override
public void setSymbolTable(SymbolTable table) {
    // TODO: 设计你可能需要的符号表存储结构
    this.symbolTable = table;
}

```

setSymbolTable()方法：直接赋值即可

中间代码生成：

```

@Override
public void whenShift(Status currentStatus, Token currentToken) {
    // TODO
    IRStack.push(null);
    symbolStack.push(new SymbolEntry(currentToken));
}

```

whenShift()方法：遇到移进动作时，符号栈中压入符号，语义栈中压入<null>占位

```

@Override
public void whenAccept(Status currentStatus) {
    // TODO
    // Do nothing 遇到Accept时中间代码生成结束，无操作
}

@Override
public void setSymbolTable(SymbolTable table) {
    // TODO
    this.symbolTable = table;
}

public List<Instruction> getIR() {
    // TODO
    return instructionList;
}

public void multipop(Stack stack, int times){
    for(int i=0; i<times; i++){
        stack.pop();
    }
}

```

whenAccept()方法：遇到接收动作，无操作

setSymbolTable()方法：直接赋值给symbolTable即可

getIR()方法：返回中间代码序列instructionList

multipop()方法：在stack栈中执行times次pop()操作

```

@Override
public void whenReduce(Status currentStatus, Production production) {
    // TODO
    switch (production.index()) {
        case 6 -> { // S -> id = E;
            multipop(symbolStack, 2);
            String identifier = symbolStack.pop().token.getText();
            IRValue variableE = IRStack.pop();
            IRVariable idName = IRVariable.named(identifier); // 获取id实体
            Instruction instruction = Instruction.createMov(idName, variableE); // 创建指令MOV
            instructionList.add(instruction);
            multipop(IRStack, 2);
            IRStack.push(null); // 压入null占位
        }
        case 7 -> { // S -> return E;
            multipop(symbolStack, 2);
            Instruction instruction = Instruction.createRet(IRStack.pop()); // 创建指令RET
            instructionList.add(instruction);
            IRStack.pop();
            IRStack.push(null); // 压入null占位
        }
    }
}

```

```

case 8 -> { // E -> E + A
    multipop(symbolStack, 3);
    IRValue variableA = IRStack.pop();
    IRStack.pop();
    IRValue variableE = IRStack.pop();
    IRVariable variable$ = IRVariable.temp();
    IRStack.push(variable$);
    Instruction instruction = Instruction.createAdd(variable$, variableE, variableA); // 创建指令ADD
    instructionList.add(instruction);
}
case 9 -> { // E -> E - A;
    multipop(symbolStack, 3);
    IRValue variableA = IRStack.pop();
    IRStack.pop();
    IRValue variableE = IRStack.pop();
    IRVariable variable$ = IRVariable.temp();
    IRStack.push(variable$);
    Instruction instruction = Instruction.createSub(variable$, variableE, variableA); // 创建指令SUB
    instructionList.add(instruction);
}
case 10, 12 -> { // E -> A, A -> B;
    symbolStack.pop();
    IRStack.push(IRStack.pop()); // 弹一压一，无新指令
}
case 11 -> { // A -> A * B;
    multipop(symbolStack, 3);
    IRValue variableB = IRStack.pop();
    IRStack.pop();
    IRValue variableA = IRStack.pop();
    IRVariable variable$ = IRVariable.temp();
    IRStack.push(variable$);
    Instruction instruction = Instruction.createMul(variable$, variableA, variableB); // 创建指令MUL
    instructionList.add(instruction);
}
}

```

```

case 13 -> { // B -> ( E );
    multipop(symbolStack, 3);
    IRStack.pop();
    IRValue variableE = IRStack.pop();
    IRStack.pop();
    IRStack.push(variableE); // 无新指令
}
case 14 -> { // B -> id;
    String identifier = symbolStack.pop().token.getText();
    IRStack.pop();
    IRVariable idName = IRVariable.named(identifier);
    IRStack.push(idName); // 无新指令
}
case 15 -> { // B -> IntConst;
    String immediateText = symbolStack.pop().token.getText();
    IRStack.pop();
    IRImmediate immediate = IRImmediate.of(Integer.parseInt(immediateText));
    IRStack.push(immediate); // 压入立即数，无新指令
}
default -> { // 未定义指令，弹出
    multipop(symbolStack, production.body().size());
    multipop(IRStack, production.body().size());
    IRStack.push(null);
}
}
symbolStack.push(new SymbolEntry(production.head()));

```

`whenReduce()`方法：遇到规约动作时，将产生式送入`switch()`

每次`switch()`结束时，将规约式左部变量压入符号栈`symbolStack`中(公共操作，提到`switch()`外)

产生式 $S \rightarrow id = E$ ：赋值语句，从栈中获取`id`和`E`的信息，调用静态方法`createMov()`生成一条`Mov`指令。

产生式 $S \rightarrow \text{return } E$ ：返回语句，调用静态方法`createRet()`生成一条`Ret`指令

产生式 $E \rightarrow E + A$ ：从语义栈`IRStack`中获取`A.val`和`E.val`，申请临时变量`variable$`，生成一条`Add`指令。

产生式 $E \rightarrow E - A$ ：从语义栈`IRStack`中获取`A.val`和`E.val`，申请临时变量`variable$`，生成一条`Sub`指令。

产生式 $E \rightarrow A$ 和 $A \rightarrow B$: 符号栈弹栈顶, 由于语法规则为 $E.val := A.val / A.val := B.val$, 语义栈顶值不变, 压入弹出的值即可

产生式 $A \rightarrow A * B$: 从语义栈IRStack中获取A.val和B.val, 申请临时变量variable\$, 生成一条Mul指令。

产生式 $B \rightarrow (E)$: 语义栈先弹一次(弹'('), 此时栈顶为E.val, 将其保存, 再弹一次(弹')'), 再将E.val压入栈, 即为B.val

产生式 $B \rightarrow id$: 符号栈弹栈, 获取id的信息, 再去查询id.val, 将其作为B.val压入语义栈

产生式 $B \rightarrow IntConst$: 符号栈弹栈, 获取IntConst的信息, 再去查询IntConst.lexval, 将其作为B.val压入语义栈

其他产生式, 语义栈、符号栈弹出与产生式长度相应的次数, 再向语义栈压入<null>占位即可

3.4.目标代码生成

3.4.1.设计思路和算法描述

```
2
3 public enum RegisterList {
4     t0, t1, t2, t3, t4, t5, t6;
5 }
6
```

可用的临时寄存器名的枚举

下为自定义的基础设施: RegisterAllocator类, 用于进行寄存器的分配

```
public class RegisterAllocator {

    public List<String> assemblyCode = new LinkedList<>();
    public List<Instruction> intermediateCode = new LinkedList<>();
    private final List<IRVariable> variableList = new LinkedList<>();
    private final List<RegisterList> existingRegisters = new LinkedList<>();

    RegisterAllocator(){
        assemblyCode.add(".text");
        for(RegisterList r: RegisterList.values()){
            variableList.add(null);
            existingRegisters.add(r);
        }
    }
}
```

上为类中的属性字段以及构造函数的定义。assemblyCode为生成的目标代码序列; intermediateCode为从上一步中读取的中间代码序列; variableList为程序中调用的各种变量的列表; existingRegisters为可用的寄存器列表。当调用目标代码生成程序时, 首先创建一个RegisterAllocator对象进行寄存器管理, 并初始化变量、寄存器列表。同时, 向汇编代码序列中写入".text"。

```
public void assignNewReg(RegisterList newReg, IRVariable newVal){
    for(int i=0; i<RegisterList.values().length; i++){
        if(existingRegisters.get(i).equals(newReg)){
            variableList.set(i, newVal);
        }
    }
}
```

assignNewReg()方法: 申请一个新的寄存器newReg并向其中写入值newVal

```

public boolean isAssigned(IRVariable destVar){
    for(int i=0; i<RegisterList.values().length; i++){
        if(destVar.equals(variableList.get(i))){
            return true;
        }
    }
    return false;
}

```

isAssigned()方法：检查目标变量destVar的值是否已在寄存器中，是则返回true，否则返回false

```

private void findAvailableReg(int initInst) {
    for(int i=0; i<RegisterList.values().length; i++) {
        boolean foundFlag = true, terminateFlag = true;
        RegisterList curReg = existingRegisters.get(i);
        for(int j=initInst; j<intermediateCode.size(); j++) {
            Instruction curInst = intermediateCode.get(j);
            if(curInst.getKind()==RET || !foundFlag || !terminateFlag) { break; }
            else if (curInst.getKind() == MOV){
                if(isAssigned(curInst.getResult())){
                    if(curReg.equals(getCorrespondReg(curInst.getResult())) { terminateFlag = false; }
                }
                if(curInst.getFrom().isImmediate()) { continue; }
                if(isAssigned((IRVariable) curInst.getFrom())) {
                    if(curReg.equals(getCorrespondReg((IRVariable) curInst.getFrom())) { foundFlag=false; }
                }
            } else {
                if(isAssigned(curInst.getResult())) {
                    if(curReg.equals(getCorrespondReg(curInst.getResult())) { terminateFlag=false; }
                }
                if(curInst.getLHS().isImmediate()) { continue; }
                if(isAssigned((IRVariable) curInst.getLHS())) {
                    if(curReg.equals(getCorrespondReg((IRVariable) curInst.getLHS())) { foundFlag=false; }
                }
                if(curInst.getRHS().isImmediate()) { continue; }
                if(isAssigned((IRVariable) curInst.getRHS())) {
                    if(curReg.equals(getCorrespondReg((IRVariable) curInst.getRHS())) { foundFlag=false; }
                }
            }
        }
        if(foundFlag) {
            variableList.set(i, null);
        }
    }
}

```

findAvailableReg()方法：寻找是否有后续不再使用的值在寄存器中。方案如下：initInst表示调用此方法时，还未完成从中间代码到目标代码的迁移的第一条中间代码（简单说即后续中间代码），对每个寄存器进行遍历：首先设置一个foundFlag且初始设为true，对后续中间代码其进行嵌套遍历，并检查其中涉及到的各个变量，如果发现有某个变量与当前寄存器中的变量相同，则将foundFlag设为false；对每个寄存器的遍历结束前，检查foundFlag，若为true，则将此寄存器的值设为<null>（释放该寄存器）。

```

public RegisterList getNewReg(int curInst) {
    findAvailableReg(curInst);
    for(int i=0; i<RegisterList.values().length; i++) {
        if(variableList.get(i) == null) {
            return existingRegisters.get(i);
        }
    }
    return null;
}

```

getNewReg()方法：先调用findAvailableReg（此时若有空闲寄存器，其值会被设为<null>），再遍历各个寄存器，若有空闲（寄存器值为<null>）则返回；若所有寄存器都被占用则返回<null>，在其调用处报错。

```

public RegisterList getCorrespondReg(IRVariable destVar) {
    for(int i=0; i<RegisterList.values().length; i++) {
        if(destVar.equals(variableList.get(i))) {
            return existingRegisters.get(i);
        }
    }
    return null;
}

```


getCorrespondReg: 遍历各寄存器，若找到目标值则返回寄存器编号，若找不到则返回<null>

下为目标代码生成具体实现类AssemblyGenerator:

```
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
    regAlc.intermediateCode = originInstructions;
    for(int i=0; i<regAlc.intermediateCode.size(); i++){
        Instruction curInst = regAlc.intermediateCode.get(i);
        if(curInst.getKind()==ADD || curInst.getKind()==SUB || curInst.getKind()==MUL){
            IRValue leftOperand = curInst.getLHS();
            if(leftOperand.isImmediate()) {
                IRVariable newTemp = IRVariable.temp();
                IRVariable result = curInst.getResult();
                IRValue rhs = curInst.getRHS();
                Instruction newInst = switch (curInst.getKind()) {
                    case ADD -> Instruction.createAdd(result, newTemp, rhs);
                    case SUB -> Instruction.createSub(result, newTemp, rhs);
                    case MUL -> Instruction.createMul(result, newTemp, rhs);
                    default -> null;
                };
                regAlc.intermediateCode.set(i, Instruction.createMov(newTemp, leftOperand));
                regAlc.intermediateCode.add(i+1, newInst);
            }
        }
    }
}
```

loadIR()方法: 将中间代码载入，并遍历一遍，若发现算数指令中有立即数，则将其分解为一条li和一条算术指令。

```
public void generateArithInstruction(Instruction curInst, int regPos, String operator) {
    IRValue source1 = curInst.getLHS();
    IRValue source2 = curInst.getRHS();
    IRVariable destination = curInst.getResult();
    RegisterList destReg, sourceReg = regAlc.getCorrespondReg((IRVariable) source1);
    if(regAlc.isAssigned(destination)) {
        destReg= regAlc.getCorrespondReg(destination);
    } else {
        destReg= regAlc.getNewReg(regPos);
        regAlc.assignNewReg(destReg,destination);
    }
    if(source2.isImmediate()) {
        regAlc.assemblyCode.add("\t" + operator + destReg + "," + sourceReg + "," + ((IRImmediate)source2).getValue());
    } else {
        RegisterList regR= regAlc.getCorrespondReg((IRVariable) source2);
        regAlc.assemblyCode.add("\t" + operator + destReg + "," + sourceReg + "," + regR);
    }
}
```

generateArithInstruction()方法: 生成算数指令Add/Sub/Mov,因操作雷同，因此单独作一个方法。

```
public void run() {
    // TODO: 执行寄存器分配与代码生成
    for(int i=0; i<regAlc.intermediateCode.size(); i++){
        Instruction curInst = regAlc.intermediateCode.get(i);
        switch(curInst.getKind()) {
            case ADD -> {
                generateArithInstruction(curInst, i, "add ");
            }
            case SUB -> {
                generateArithInstruction(curInst, i, "sub ");
            }
            case MUL -> {
                generateArithInstruction(curInst, i, "mul ");
            }
            case MOV -> {
                IRValue initial = curInst.getFrom();
            }
        }
    }
}
```

run()方法: 遍历中间代码并送入switch(), 若为ADD/SUB/MUL则调用generateArithInstruction()生成算术指令。

```

}
case MOV -> {
    IRValue initial = curInst.getFrom();
    IRVariable terminal = curInst.getResult();
    RegisterList destReg;
    if(regAlc.isAssigned(terminal)) {
        destReg= regAlc.getCorrespondReg(terminal);
    } else {
        destReg= regAlc.getNewReg(i);
        regAlc.assignNewReg(destReg,terminal);
    }
    if(initial.isImmediate()) {
        regAlc.assemblyCode.add("\tli " + destReg + "," + ((IRImmediate)initial).getValue());
    } else {
        RegisterList regFrom = regAlc.getCorrespondReg((IRVariable)initial);
        regAlc.assemblyCode.add("\tmv " + destReg + "," + regFrom);
    }
    regAlc.assignNewReg(destReg, terminal);
}
case RET -> {
    IRValue irv = curInst.getReturnValue();
    RegisterList r= regAlc.getCorrespondReg((IRVariable) irv);
    regAlc.assemblyCode.add("\tmv a0," + r);
}
}

```

run()方法：若为MOV，则生成Mov；若为RET，则生成一条Mov指令，将结果保存至返回值寄存器a0中。

4.实验结果与分析

4.1.实验1：词法分析器

4.1.1.out/token.txt:

```

1      (int,)
2      (id,result)
3      (Semicolon,)
4      (int,)
5      (id,a)
6      (Semicolon,)
7      (int,)
8      (id,b)
9      (Semicolon,)
10     (int,)
11     (id,c)
12     (Semicolon,)
13     (id,a)
14     (=,)
15     (IntConst,8)
16     (Semicolon,)
17     (id,b)
18     (=,)
19     (IntConst,5)
20     (Semicolon,)
21     (id,c)
22     (=,)
23     (IntConst,3)
24     (-,)
25     (id,a)
26     (Semicolon,)
27     (id,result)
28     (=,)
29     (id,a)
30     (*,)
31     (id,b)

```

```

28 (.)
29 (id,a)
30 (*,)
31 (id,b)
32 (-,)
33 ((.)
34 (IntConst,3)
35 (+,)
36 (id,b)
37 (.)
38 (*,)
39 ((.)
40 (id,c)
41 (-,)
42 (id,a)
43 (.)
44 (Semicolon,)
45 (return,)
46 (id,result)
47 (Semicolon,)
48 ($.)
49

```

4.1.2.out/old_symbol_table.txt:

```

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

```

4.1.3.python脚本验证:

```

PS C:\Users\De0Faustum\Desktop\Compilation\Lab\Exp3-Intermediate-Code\template\scripts> python check-result.py 1 ../data/std ../data/out
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.

PS C:\Users\De0Faustum\Desktop\Compilation\Lab\Exp3-Intermediate-Code\template\scripts>

```

4.2.实验二：语法分析器

4.2.1.out/parser_list.txt

```

1 D -> int
2 S -> D id
3 D -> int
4 S -> D id
5 D -> int
6 S -> D id
7 D -> int
8 S -> D id
9 B -> IntConst
10 A -> B
11 E -> A
12 S -> id = E
13 B -> IntConst
14 A -> B
15 E -> A
16 S -> id = E
17 B -> IntConst
18 A -> B
19 E -> A
20 B -> id
21 A -> B
22 E -> E - A
23 S -> id = E
24 B -> id
25 A -> B
26 B -> id
27 A -> A * B
28 E -> A
29 B -> IntConst
30 A -> B
31 E -> A
32 B -> id
33 A -> B
34 E -> E + A

```

4.2.2.python脚本验证:

```

PS C:\Users\De0Faustrum\Desktop\Compilation\Lab\Exp3-Intermediate-Code\template\scripts> python check-result.py 2 ../data/std ../data/out
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.

PS C:\Users\De0Faustrum\Desktop\Compilation\Lab\Exp3-Intermediate-Code\template\scripts> 

```

4.3.实验3: 语义分析和中间代码生成器

4.3.1.out/new_symbol_table.txt

```

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

```

4.3.2.out/new_symbol_table.txt

```
intermediate_code.txt × new_symbol_table.txt ×
\Lab\Exp3-Intermediate-Code\template
2      (MOV, b, 5)
3      (SUB, $0, 3, a)
4      (MOV, c, $0)
5      (MUL, $1, a, b)
6      (ADD, $2, 3, b)
7      (SUB, $3, c, a)
8      (MUL, $4, $2, $3)
9      (SUB, $5, $1, $4)
10     (MOV, result, $5)
11     (RET, , result)
12
```

4.3.3.out/new_symbol_table.txt

```
ir_emulate_result.txt ×
1      144
2
```

4.3.4.python脚本验证:

```
PS C:\Users\De0Fastrum\Desktop\Compilation\Lab\Exp3-Intermediate-Code\template\scripts> python check-result.py 2 ../data/std ../data/out
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 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.

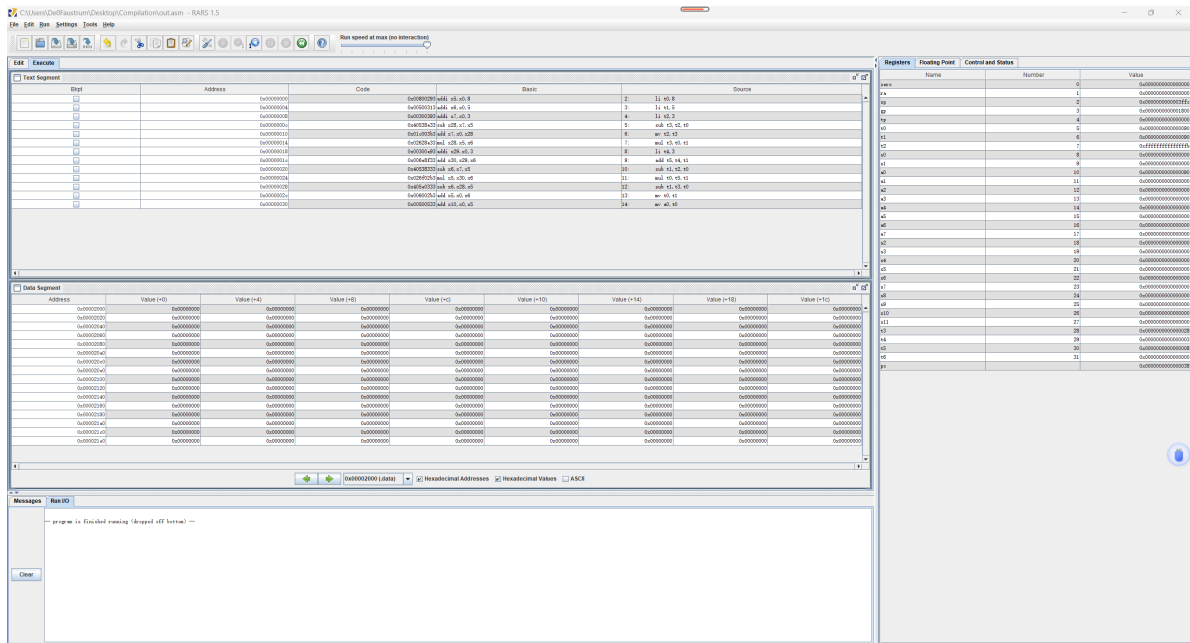
PS C:\Users\De0Fastrum\Desktop\Compilation\Lab\Exp3-Intermediate-Code\template\scripts>
```

4.4.实验4：目标代码生成

4.4.1.out/assembly_language.asm

```
assembly_language.asm ×
Clion 支持 *.asm 文件
1      .text
2      li t0,8
3      li t1,5
4      li t2,3
5      sub t3,t2,t0
6      mv t2,t3
7      mul t3,t0,t1
8      li t4,3
9      add t5,t4,t1
10     sub t1,t2,t0
11     mul t0,t5,t1
12     sub t1,t3,t0
13     mv t0,t1
14     mv a0,t0
15
```

4.4.2.RARS汇编运行结果：



Registers Floating Point Control and Status			
Name	Number	Value	
zero	0	0x0000000000000000	
ra	1	0x0000000000000000	
sp	2	0x0000000000003ffe	
gp	3	0x0000000000001800	
tp	4	0x0000000000000000	
t0	5	0x0000000000000090	
t1	6	0x0000000000000090	
t2	7	0xfffffffffffffffb	
s0	8	0x0000000000000000	
s1	9	0x0000000000000000	
s2	10	0x0000000000000090	
s3	11	0x0000000000000000	
s4	12	0x0000000000000000	
s5	13	0x0000000000000000	
s6	14	0x0000000000000000	
s7	15	0x0000000000000000	
s8	16	0x0000000000000000	
s9	17	0x0000000000000000	
s10	18	0x0000000000000000	
s11	19	0x0000000000000000	
t3	20	0x0000000000000000	
t4	21	0x0000000000000000	
t5	22	0x0000000000000000	
t6	23	0x0000000000000000	
pc	24	0x0000000000000038	

如上图所示，运行后返回值寄存器 a_0 中的值为0x90,即为十进制144，符合预期结果。

5.实验中遇到的困难与解决办法

实验1：刚接触模板代码时，各类接口数量很多，一时不能完全看懂，导致写的进度较慢，后花了两个小时研究模板代码并且画出类图之后才厘清。

实验2：设计符号栈时，因为符号栈中既能有终结符也能有非终结符，因此在定义符号栈时， $Stack<>$ 的泛型类型难以确定，后自主设计了新的数据结构SymbolEntry类得到解决。

实验4：寄存器分配算法的实现较为困难，尤其是查找后续不再使用的寄存器，后仔细研究了指导书，设计了空闲寄存器回收算法得以解决。