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:

学　号:

专　业: 计算机科学与技术

日　期: 2023-xx-xx

1 实验目的与方法

实验目的：利用 Java 语言实现 TXTv2 语言的编译器，主要包括词法分析器、语法分析器、语义分析与中间代码成器以及目标代码生成器四个组成部分。从实验 1 到实验 4，每次小实验均完成上述四个组成部分之一进行迭代。最终，生成能够在 RISC-V32 目标平台运行的汇编代码。

实验方法：主要编程语言是 Java,版本为 Java 21.0.1。开发工具是 VScode。此外，在语法分析器中还使用了编译工作台。在运行目标平台代码时，使用了 RARS。

1.1 词法分析器

实验目的：

1. 加深对词法分析程序的功能及实现方法的理解；
2. 对类 C 语言的文法描述有更深的认识，理解有穷自动机、编码表和符号表在 编译的整个过程中的应用；
3. 设计并编程实现一个词法分析程序，对类 C 语言源程序段进行词法分析，加 深对高级语言的认识。

1.2 语法分析

实验目的：

1. 深入了解语法分析程序实现原理及方法。
2. 理解 LR(1)分析法是严格的从左向右扫描和自底向上的语法分析方法。
3. 在利用 LR(1)分析法，对实验 1 产出的 tokens 进行语法分析。检查语法错误。

1.3 典型语句的语义分析及中间代码生成

实验目的：

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

1.4 目标代码生成

实验目的：

1. 加深编译器总体结构的理解与掌握；
2. 掌握常见的 RISC-V 指令的使用方法；

3. 理解并掌握目标代码生成算法和寄存器选择算法。

2 实验内容及要求

2.1 词法分析器

实验内容：
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。
实验要求：
1. 输入：以文件形式存放的放自定义的类 C 语言程序段;
2. 输出：以文件形式存放的 Token 串和简单符号表；
3. 输入的类 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。

2.2 语法分析

实验内容：
利用 LR(1)分析法，设计语法分析程序，对输入单词符号串进行语法分析。
实验要求：
1. 输入：实验 1 输出的 token 串；
2. 输出：利用 LR(1)分析法过程中所用的产生式序列并将其保存在文件中。

2.3 典型语句的语义分析及中间代码生成

实验内容：
设计单词串翻译方案，对单词串进行语法制导翻译。在翻译的过程中更新符号表并产生中间代码。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。并使用框架中的模拟器 IREmulator （中间代码模拟器)验证生成的中间代码的正确性。
实验要求：
1. 输入：实验 1 中生成的 token 串；
2. 输出：更新后的符号表与产生的中间代码并将二者以文件形式保存。

2.4 目标代码生成

实验内容：
将实验三生成的中间代码转换为目标代码（汇编指令)；利用 RARS 运行生成的目标代码，验证结果的正确性。
实验要求：
1. 输入：实验 3 中生成的中间代码；
2. 输出：能够在目标平台 RISCV32 上运行的汇编代码。使用 RARS 运行由编译程序生成的目标代码，验证结果的正确性。

3 实验总体流程与函数功能描述

3.1词法分析

3.1.1 编码表

词法分析器输出的 token 单词序列主要有五类：关键字、运算符、分界符、标识符、常数。前三者是程序设计语言预先定义的，其数量是固定的。而标识符、常数则是由程序 设计人员根据具体的需要按照程序设计语言的规定自行定义的，其数量可以是无 穷多个。编译程序为了处理方便，通常需要按照一定的方式对单词进行分类和编码。将 token 表示为(类别编码，单词值)这一二元组形式。对于关键字、运算符、分界符，单词值可以为空。对于标识符、常数单词值不能为空。

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

对应/data/in/coding_map.csv 中的编码。

3.1.2 正则文法

对单词进行形式化描述，采取正则文法。

G=(V,T,P,S)，其中 V={S,A,B,Cdigit,no_0_digit,char},T={任意符号}，P 定义如下

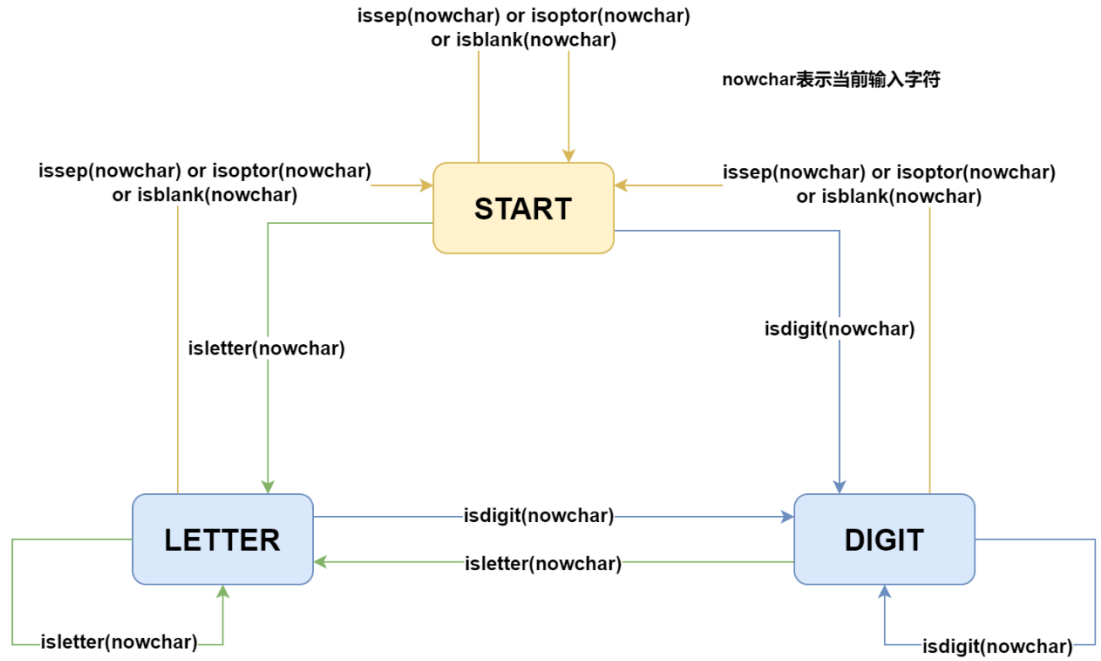
- 标识符：S→letter A　　A→letter A|digit A| ε
- 运算符、分隔符：S->B B->=|*|+|-|/|(|)|;
- 整常数：S ->no_0_digitB　 B -> digitB | ε
- 字符串常量：S -> “C”
- 字符常量：S->’D’

规定： 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,_

3.1.3 状态转换图

下图示意识别一行源代码语句的状态转换图



注：上图中仅体现状态转变。状态转变对应的操作略去。

3.1.4 词法分析程序设计思路和算法描述

1. 对 SymbolTable.java 的修改

（1）主要存储数据结构采取 HashMap。键类型为 String，值类型为 SymbolTableEntry。其中对于值类型，其有两个属性，分别为 text(String)和 type(SourceCodeType)。用于存储对应的符号的文本内容以及类型。在本次实验中，仅考虑整数类型，故 SourceCodeType 这一枚举类类型中仅有 Int 类。

```
public class SymbolTable {

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

TableEntry.java D:\document\study\Threeup\exp\编译原理\template\src\cn\edu\hitsz\compiler\symtab - 定义 (1)

```
public class SymbolTableEntry {
    public void setType(SourceCodeType type) {

        this.type = type;
    }

    private final String text;
    private SourceCodeType type;
}
```

（2）该类的 get、add、has 方法的实现

分别调用 HashMap 中 get、put、containsKey 方法来获取对应的返回值或进行对应的操作。

（3）通过 getAllEntries 方法获得对 symbolMap 的引用；通过 dumpTable 方法将符号表导出到指定文件中。

2. 对 LexicalAnalyzer.java 的修改

（1）词法分析的缓冲区实现

将源代码文件逐行读取，并存储到 sourceTestlist 中。此过程采用框架所提供的 FileUtils.readLines 函数。

（2）自动机实现词法分析过程 run()函数

对每一行源代码调用 LexicalOneLine 函数，逐行对源代码进行分析。在所有行都分析结束后添加 eof，表明 token 串的结束。

（3）核心函数 LexicalOneLine

- 对于关键字的识别，采取 HashSet 进行存储

```
private static final Set<String> keywords = new HashSet<>();
static {
    keywords.add(e:"int");
    keywords.add(e:"return");
}       You, 上个月 • single line token created
```

- 三个主要的变量：

  分别表示当前输入的字符的位置索引；当前状态机状态；用于存储标识符、常数等的缓冲区。

```
int currentPos = 0;
int currentState = START;
StringBuilder currentToken = new StringBuilder();       You, 上个月 • sing
```

- 基建函数：

  ·addLetterTotokens(StringBuilder currentToken)：

  将关键字和变量名（即 SymbolTable 中存储的标识符）加入到 tokens 中。并清空缓冲区。在加入 tokens 中，还需判断是否是关键字，如果不是，还需将其添加到 SymbolTable 中。

```
/**
 * 将关键字和变量名等字母组成的字符输出到tokens中，并将缓冲区清空
 *
 * @param currentToken
 */
private void addLetterTotokens(StringBuilder currentToken) {      You, 上个月 • single line token created
    String temp = currentToken.toString();
    if (keywords.contains(temp)) {// 是关键字
        tokens.add(Token.simple(temp));
    } else {// 不是关键字
        tokens.add(Token.normal(tokenKindId:"id", temp));
        addToSymbolTable(temp);
    }
    currentToken.setLength(newLength:0);// 清空字符串
}
```

  ·addNumberTotokens(StringBuilder currentToken)：

  将数字字符串输出到 tokens 中，并将缓冲区清空。

```
/**
 * 将数字字符串输出到tokens中，并将缓冲区清空       You, 上个月 • exp1 complet
 *
 * @param currentToken
 */
private void addNumberTotokens(StringBuilder currentToken) {
    String temp = currentToken.toString();
    tokens.add(Token.normal(tokenKindId:"IntConst", temp));
    currentToken.setLength(newLength:0);// 清空字符串
}
```

  ·addOperatorTotokens(*char op*)

将运算符输出到 tokens 中，包括四则运算、等于号

・addSepTotokens(*char sepsym*)

将分隔符输出到 tokens 中，包括(),;四个符号

- 状态机主要有三个状态，START、LETTER、DIGIT。分别表示初始状态，字母状态、数字状态。下面以 LETTER 状态进行说明。前文提及，当前输入的字符的位置索引为 currentPos。其对应的单个字符为 nowchar。

```
case LETTER:
    if (nowchar == ' ') {
        addLetterTotokens(currentToken);
        currentState = START;       You, 上个月 • single line token created
    } else if (Character.isLetter(nowchar)) {
        currentToken.append(nowchar);
        currentState = LETTER;
    } else if (Character.isDigit(nowchar)) {
        throw new RuntimeException(message:"在字母后紧跟数字");
    } else if (nowchar == '+' || nowchar == '-' || nowchar == '*' || nowchar == '/' || nowchar == '=') {
        addLetterTotokens(currentToken);
        addOperatorTotokens(nowchar);
        currentState = START;
    } else if (nowchar == ';' || nowchar == ',' || nowchar == '(' || nowchar == ')') {
        addLetterTotokens(currentToken);
        addSepTotokens(nowchar);
        currentState = START;
    } else {
        throw new RuntimeException(message:"未知字符");
    }
    currentPos++;
    break;
```

当前状态是 LETTER，说明 currentToken（下称 token 缓冲区）中至少已有一个字母。nowchar 表示当前输入字符。

・输入为空格：调用 addLetterTotokens 函数。并修改状态机状态为初始状态。

・输入为字母：加入 token 缓冲区中，状态机状态仍为 letter 状态。

・输入为数字：产生错误。因为关键字以及标识符不存在字母后面紧跟数字的情况。

・输入为四则运算符号与等号：调用 addLetterTotokens 函数。并将 nowchar 加入 tokens 中。修改状态机状态为初始状态。一个例子：abc+这个长度为 4 的字符串。当 abc 已在 token 缓冲区时，如果遇到+，应该将 token 缓冲区添加到 tokens 串中并清空 token 缓冲区。之后再将+号添加到 tokens 串中。

・输入为分界符：与上一种情况类似。

・其他情况：报错。

- 同时，未避免分析完源代码一行后 token 缓冲区中仍有值(虽然此种情况的出现往往时因为没有加分号)，应该对缓冲区中剩余的字符进行处理。

```
    // 为避免int a中因为没有出现;导致最后漏掉token。还需要检查builder中是否还有字符
    if (currentToken.length() != 0) {
        if (currentState == LETTER) {
            addLetterTotokens(currentToken);
        } else if (currentState == DIGIT) {       You, 上个月 • repair end withou
            addNumberTotokens(currentToken);
        }
    }
}
```

3.2 语法分析

3.2.1 拓展文法

对于某些文法，存在一些右部含有文法开始符号的产生式，在归约过程中需要分清楚是 否已经归约到文法的最初开始符。因此，需要对原有文法进行拓广。在本次实验中，拓展文法框架已经提供，以文件形式存储在/data/in/grammar.txt 中。

```
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

在进行 LR(1)语法分析时，需要根据 LR1 分析表进行操作。移入操作时，根据 Action 表进行状态的转移。规约操作时，根据 Goto 表选择对应的规约产生式并进行状态的转移。但无论是 Action 还是 Goto 操作，都依赖于状态栈和符号栈。

	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 Ser		16	2			3
8											reduce S -> D id							
9	shift 13	shift 14								shift 15			17				11	12
10				shift 18	shift 19						reduce S -> return E							
11				reduce E -	reduce E -	shift 20					reduce E -> A							
12				reduce A -	reduce A -	reduce A -> B					reduce A -> B							
13				reduce B -	reduce B -	reduce B -> id					reduce B -> id							
14	shift 24	shift 25								shift 26			21				22	23
15				reduce B -	reduce B -	reduce B -> IntConst					reduce B -> IntConst							
16												reduce S_list -> S Semicolon S_list						
17				shift 18	shift 19						reduce S -> id = E							
18	shift 13	shift 14								shift 15							27	12
19	shift 13	shift 14								shift 15							28	12
20	shift 13	shift 14								shift 15								29
21			shift 30	shift 31	shift 32													
22			reduce E -	reduce E -	reduce E -	shift 33												
23			reduce A -	reduce A -	reduce A -	reduce A -> B												
24			reduce B -	reduce B -	reduce B -	reduce B -> id												
25	shift 24	shift 25								shift 26			34				22	23
26			reduce B -	reduce B -	reduce B -	reduce B -> IntConst												
27				reduce E -	reduce E -	shift 20					reduce E -> E + A							

3.2.3 状态栈和符号栈的数据结构和设计思路

1. 状态栈

Status 类在框架中已提供。在 SyntaxAnalyzer 类中，以此为基础建立状态栈。

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

2. 符号栈

在设计符号栈中，需要注意到符号栈中需要同时能够存储终结符（在代码中也就是 Token 类型）和非终结符类型（代码中即 NonTerminal 类型）。为此需要建立一个接口或者类进行统一。实现中，采取了后者。建立了 Symbol.java 类。使用方法重载完成了不同的类构造方法。代码如下：

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

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

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

    public Symbol(NonTerminal nonTerminal) {
        this(token:null, nonTerminal);      You, 2周前 • lab3:symbol type
    }

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

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

    @Override
    public String toString() {
        return (isToken() ? token.toString() : nonTerminal.toString());
    }
}
```

3.2.4 LR 驱动程序设计思路和算法描述

1. registerObserver

实验 2 中，由于需要输入规约产生式列表。在此采取观察者模式。将一个规约收集观察者注册到 SyntaxAnalyzer 类中的观察者列表当中。在分析过程中，如果采取了规约操作。只需要通知所有的观察者。同时也对于该观察者调用 setSymbolTable 方法。观察者可以选择是否需要持有一个对于符号表的引用。

2. 加载 tokens 串和 LR1 分析表

分别在函数 loadTokens 和 loadLRTable 中完成。

3. LR1 分析过程

（1）首先，初始化状态栈和符号栈。对于状态栈，调用 lrTable.getInit()方法进行初始化。

（2）重复以下过程，直至达到 accept 状态。

- 获取当前状态栈栈顶元素以及输入 token。根据 lrTable 选择对应的操作。

```
currentStatus = statusStack.peek();
currentToken = tokens.get(currentTokenIndex);
action = lrTable.getAction(currentStatus, currentToken);
```

- 是移入操作：
  - 通知所有观察者。向他们传递当前状态和当前输入 token。
  - 将 action 获取到的状态压入状态栈。
  - 根据 token 创建新的 Symbol 对象，并将其压入符号栈。
  - token 索引向后自增 1。
- 是规约操作：
  - 根据 action 获取规约产生式。
  - 通知所有者。向他们传递当前状态和规约产生式。
  - 获取产生式右部的长度。将状态栈和符号栈都弹出该长度。
  - 根据产生式的头部和当前状态选择 goto 的状态。
  - 往符号栈中压入产生式左部的非终结符，往状态栈中压入 goto 的状态。
- 在其他情况下，进行报错。

```
while (!success) {
    currentStatus = statusStack.peek();
    currentToken = tokens.get(currentTokenIndex);
    action = lrTable.getAction(currentStatus, currentToken);
    switch (action.getKind()) {
        case ActionKind.Accept:
            this.callWhenInAccept(currentStatus);
            success = true;
            break;
        case ActionKind.Shift:
            this.callWhenInShift(currentStatus, currentToken);
            statusStack.push(action.getStatus());
            symbolStack.push(new Symbol(currentToken));
            // 终结符入栈后，指针后移
            currentTokenIndex++;
            break;
        case ActionKind.Reduce:
            Production production = action.getProduction();
            this.callWhenInReduce(currentStatus, production);
            rightlen = production.body().size();
            // 两个栈弹出
            this.pop2Stacks(rightlen);
            // 获取非终结符
            notmal = production.head();
            // 往符号栈压入非终结符
            symbolStack.push(new Symbol(notmal));
            // 往状态栈中压入goto获取到的状态
            currentStatus = statusStack.peek();
            statusStack.push(currentStatus.getGoto(notmal));
            break;
        case ActionKind.Error:
            throw new RuntimeException(message:"error");
        default:
            throw new RuntimeException(message:"error");
    }
}
System.out.println(this.getClass() + ":syntax analysis finished");
```

3.3 语义分析和中间代码生成

3.3.1 翻译方案

采用 S-属性定义的自底向上翻译方案。具体方案如下：

P → S_list {P.val = S_list.val}
S_list → S Semicolon S_list {S_list.val = S_list1.val}
S_list → S Semicolon; {S_list.val = S.val }
S → D id {p=lookup(id.name); if p != nil then enter(id.name, D.type) else error}
S → return E; {S.value = E.value}
D → int {D.type = int;}
S → id = E {gencode(id.val = E.val);}
E → A {E.val = val;}
A → B {A.val = B.val;}
B → IntConst {B.val = IntConst.lexval;}
E → E1 + A {E.val = newtemp(); gencode(E.val = E1.val + A.val);}
E → E1 – A {E.val = newtemp(); gencode(E.val = E1.val - A.val);}
A → A1 * B{A.val = newtemp();gencode(A.val = A1.val * B.val);}
B → (E){B.val = E.val;}
B → id { p = lookup(id.name); if p != nil then B.val = id.val else error}
函数说明：
lookup(name)查询符号表，返回 name 对应的记录
gencode(code)生成三地址指令 code，即中间代码
enter 将变量的类型填入符号表，即更新符号表

3.3.2 语义分析和中间代码生成的数据结构

1. 语义分析数据结构

在语义分析中，主要目的是更新符号表。与实验 2 中相同，符号栈中此时存放的是终结符（在代码中也就是 Token 类型）和非终结符类型（代码中即 NonTerminal 类型）。与实验 2 略有不同，在语义分析中仅用 Token 类型作为符号栈的数据结构。只采用一个存储 Token 类型的栈。

- 对于 Token 类型本身可以直接入栈。
- 对于 NonTerminal 类型,我们将其转化为 Token 类型。非终结符的字符描述(比如 D->int 这个产生式非终结符 D 的字符描述就是“D”）存入 Token.text 中。而非终结符的属性（在更新符号表中是变量类型）则利用 Token.kind 进行保存与传递。

2. 中间代码生成

与语义分析类似，在实现中复用 Token 类型。只采用一个存储 Token 类型的栈。

- 如果一个非终结符最后生成的是立即数，那么在 Token.kind 字段保存的就是 IntConst 类型，Token.text 字段保存的就是该立即数的字符串形式。
- 如果一个非终结符不是直接最后产生立即数，那么在 Token.kind 字段保存的就是 eof 类型，Token.text 字段保存的就是该非终结符对应的中间临时变量的字符串形式。

同时，用 IntConst 和 eof 区分，也与 IRImmediate 和 IRVariable 这两类相对应。便于产生中间代码。

3.3.3 语发分析程序设计思路和算法描述

1. SemanticAnalyzer 语义分析观察者

观察者 SemanticAnalyzer 已经在语法分析阶段被注册到观察者列表 observers 当中，因

此会在语法分析程序执行移入、归约、接收等动作时，接受到通知以及相关的信息。仅使用一个栈 public Stack<Token> tokenStack = new Stack<>()。同时在注册过程中，也利用类变量 st_in_SA 持有了对符号表的一个引用（这一过程在 setSymbolTable 函数中实现）。

（1）在接受动作（whenAccept）中，语义分析器不需要执行任何操作。因为此时意味着语法分析和语义分析的结束。

（2）在移入动作(whenShift)中，tokenStack 将每一个被观察者传来的 Token 入栈。

（3）在规约动作(whenReduce)中，仅需处理 4 和 5 两条产生式。因为在本次实验中语义分析的主要任务是更新符号表。与标识符的类型有关的产生式只有 4、5 两条内容，分别为：S -> D id; D -> int;

- 对于第 4 条产生式 S -> D id
  - 使用第五条产生式时，栈顶符号一定是 id。且栈顶符号下一个 token 其 kind 字段存储着 id 的变量类型信息。
  - 连续两次弹栈分别保存在 idout 和 Dout 两个 Token 类型的变量之中。
  - 将构造的空 Token S 压入栈中。
  - 根据 idout.text 查找符号表，将其类型设为 Dout.kind 对应的类型。这一操作在 setSymbolType 函数中实现（见（4））。
- 对于第 5 条产生式 D -> int
  - 使用第五条产生式时，栈顶符号一定是 int。
  - 将 int 弹出栈，并获取 int 对应的 Token.kind 的字符串描述。
  - 构造新的 Token。kind 与 int 对应的 Token.kind 相同，text 属性设置为“nonterminal D”（text 属性不太重要）。
  - 将新构造的 token 入栈。

```
@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
    // System.out.println(tokenStack);

    Token top;
    switch (production.index()) {
        case 5: //D -> int;
            top = tokenStack.pop();
            var tokenD = Token.normal(top.getKindId(), text:"nonterminal D");
            tokenStack.push(tokenD);
            break;
        case 4: //S -> D id;
            var idout = tokenStack.pop();
            var Dout = tokenStack.pop();
            Token tokenS = null;
            tokenStack.push(tokenS);
            setSymbolType(idout.getTextString(), Dout.getKindString());
            break;
        default:
            break;
    }

}
```

（4）setSymbolType

该函数主要作用时在使用产生式 4 时，在符号表中根据 var_name 查找对应的 symbol_tb_entery。并利用 setType 函数将该 entery 的 type 字段更新为根据 var_type 选择的 SourceCodeType。采用此方式，也方便了其他 *SourceCodeType* 的添加和实现。

```
private void setSymbolType(String var_name, String var_type){
    // System.out.println("set variable type begin");
    if(st_in_SA.get(var_name).getType() != null){
        return;
    }
    var symbol_tb_entery = st_in_SA.get(var_name);
    switch (var_type) {
        case "int":
            // System.out.printf("set variable %s to IntConst\n", var_name);
            symbol_tb_entery.setType(SourceCodeType.Int);
            break;
        default:
            break;
    }
    st_in_SA.printSymbolTable();
    // System.out.println("set variable type end");
}
```

2. IRGenerator 中间代码生成器观察者的实现

观察者 IRGenerator 已经在语法分析阶段被注册到观察者列表 observers 当中，因此会在语法分析程序执行移入、归约、接收等动作时，接受到通知以及相关的信息。与 SemanticAnalyzer 语义分析不同，中间代码生成器不需要持有对于符号表的引用。因此 setSymbolTable 函数什么也不用做。

（1）在接受动作（whenAccept）中，中间代码生成器不需要执行任何操作。因为此时意味着语法分析和中间代码生成的结束。

（2）在移入动作(whenShift)中，tokens 将每一个被观察者传来的 Token 入栈。

（3）在规约动作（whenReduce）中，针对使用的不同的产生式，会产生不同的中间代码。

- 首先说明相关辅助函数的作用

  · *boolean* genImm(String *kindstring*)：根据传入的 token 的 kind 属性的字符串描述来判断该非终结符的类型（即是 IRvariable 还是 IRimmediate）。如果是 IRimmediate 则返回真。

  · *boolean* genId(String *kindstring*)：函数主要思路与 genImm 相同。

  · *IRValue* genL_R(Token *t*)：根据传入的 token，利用上面的两个判断函数判断是生成 IRvariable 还是 IRimmediate。并用二者的父类 IRValue 作为返回值的类型。

```
private IRValue genL_R(Token t) {
    if (genId(t.getKindString())) {// 是产生的变量
        return IRVariable.named(t.getTextString());
    } else if (genImm(t.getKindString())) {// 是立即数
        return IRImmediate.of(Integer.parseInt(t.getTextString()));
    } else {
        throw new RuntimeException(message:"genL_R出错");
    }
}
```

- 对于 8，9，11 三条产生式，即 E -> E + A; E -> E - A; A -> A * B

  实现思路大体相同。都能产生一条运算中间代码。当规约时候使用这三条产生式之一时，采取下面步骤。

  · 符号栈连续弹出三个符号。将第一个弹出的和第三个弹出的符号进行保存。

  · 利用 IRVarible.temp()方法产生一个临时变量。

  · 调用 genL_R()函数为第一个弹出的和第三个弹出的符号创建对应的 IRValue。（通用

父类是 IRValue，但实际类型是 IRvariable 还是 IRimmediate 是符号本身决定的）

・调用静态方法创建一条中间代码并将其添加到 instructions 列表中。

・利用产生式规约后，还需要根据产生式头部对应的临时变量构造一个新的 token，并将其压入符号栈。Token 的 kind 字段为 eof（因为规约之后的符号是一个 IRvariable），Token 的 text 字段对应的是临时变量的字符串。

下图为使用产生式 8：

```
case 8:// E -> E + A;
    a = tokens.pop();
    tokens.pop();
    e = tokens.pop();
    temp = IRVariable.temp();
    // 判断E和A类型并产生对应中间变量
    p1 = genL_R(e);
    p2 = genL_R(a);
    instructions.add(Instruction.createAdd(temp, p1, p2));
    tokens.push(Token.normal(TokenKind.eof(), temp.toString()));
    break;
```

- 对于产生式 10，12，13，即 E -> A；A->B；B -> (E)

这两条产生式不会产生新的中间代码。其主要作用是将右部的属性传递到头部。对于 10，是将 A 的属性传递给 E；对于 13 是将 E 的属性传递给 B。二者相似，下面以产生式 13 为例说明。

・将符号栈连续 3 次弹栈，获取第二个弹出的符号（即 E）。

・如果 E 是一个 IRvariable 类型的 token，即 genId()为真，则将 e 的 text 属性赋值给 B 的 text 属性，利用 eof 这一个 TokenKind 构造一个对应 IRvariable 类型的 Token，并将其入栈。

・如果 E 是一个 IRimmediate 类型的 token，即 genImm()为真，则将 e 的 text 属性赋值给 B 的 text 属性，利用 e 的 token.kind 属性构造一个对应 IRvariable 类型的 Token，并将其入栈。

如此一来，在规约过程中，无论是 IRvariable 还是 IRimmediate 都在不断地从语法分析数底部向上传递。

```
case 13:// B -> ( E );
    tokens.pop();
    e = tokens.pop();
    tokens.pop();
    if (genId(e.getKindString())) {
        tokens.push(Token.normal(TokenKind.eof(), e.getTextString()));
    } else if (genImm(e.getKindString())) {      You, 上周 • lab3 finish
        tokens.push(Token.normal(e.getKindString(), e.getTextString()));
    } else {
        throw new RuntimeException(message:"B -> ( E )出错");
    }
    break;
```

- 对于 14，15 产生式，即 B->id；B->IntConst

这两条产生式可以看作是语法分析数的“叶子”节点。

・当运用到产生式 14 时，对符号栈进行弹栈。利用 eof 和 id.text 字段构造一个新的 token 并入栈。

・当运用到产生式 15 时，对符号栈进行弹栈。利用弹出符号的 kind 和该 IntConst 的 text 字段构造一个新的 token 并入栈。

```
case 14:// B -> id;       You, 上周 • lab3 finish
    id = tokens.pop();
    tokens.push(Token.normal(TokenKind.eof(), id.getTextString()));
    break;
case 15:// B -> IntConst;
    intconst = tokens.pop();
    tokens.push(Token.normal(intconst.getKindString(), intconst.getTextString()));
    break;
default:
```

- 对于产生式 6，即 S -> id = E;

  该产生式对应的中间代码中的 mov 指令。我们连续三次弹栈。用变量 e 存储弹出的第一个符号，id 存储第三个弹出的符号。

  - 如果 e 是产生立即数的，即 genImm()为真，需要用 IRImmediate 类型存储立即数，立即数的大小从 e 的 text 字段中获得。
  - 如果 e 是产生中间变量的，即 genId()为真，需要用 IRvariable 类型存储该变量。
  - 最后根据 id 的 text 字段和产生的 IRvalue 构造 Mov 产生式。

```
case 6:// S -> id = E;
    e = tokens.pop();
    tokens.pop();
    id = tokens.pop();
    if (genId(e.getKindString())) {// 说明是E非立即数
        instructions.add(Instruction.createMov(IRVariable.named(id.getTextString()),
                IRVariable.named(e.getTextString())));
    } else if (genImm(e.getKindString())) {// 说明是E是立即数
        instructions.add(Instruction.createMov(IRVariable.named(id.getTextString()),
                IRImmediate.of(Integer.parseInt(e.getTextString()))));       You, 上周 • lab3 finish
    } else {
        throw new RuntimeException(message:"S -> id = E出错");
    }
    break;
```

- 对于产生式 7，即 S ->return E

  根据弹出的 e 利用 genL_R()函数判断是变量还是立即数并返回对应的对象。利用 createRet 静态方法创建 ret 中间代码语句。

```
case 7:// S -> return E;
    e = tokens.pop();
    ret = tokens.pop();
    p1 = genL_R(e);
    instructions.add(Instruction.createRet(p1));       You, 上周 • lab3 finish
    break;
```

3.4 目标代码生成

3.4.1 设计思路

1. 汇编代码数据结构：

- Asmsentence：一条汇编语句对应的数据结构
  - 属性主要有三部分组成，与框架提供的 IR 语句类类似。

```
public class Asmsentence {
    // 基本元素
    private final AsmKind kind;
    private final AsmvalReg result;
    private final List<Asmvalue> parts;
    //基建设施
```

  - 将构造方法设为私有，通过静态方法创建一个实例。在创建实例过程中进行数据类型的校验。下为 addi 汇编语句的创建代码。

```
public static Asmsentence createaddi(AsmKind kind, Asmvalue part1, Asmvalue part2, AsmvalReg result) {
    if (!(part1 instanceof AsmvalReg && part2 instanceof AsmvalImm)) {
        throw new IllegalArgumentException(s:"part1 must be AsmvalReg, part2 must be Asmint");
    }
    return new Asmsentence(kind, result, List.of(part1, part2));
}
```

addi 中第二个参与运算的数据结构一定是立即数。在此利用 instanceof 进行数据类型的校验。

- AsmKind：汇编语句的类型，采用 enum 枚举类实现。

```
public enum AsmKind {
    add, addi, sub, subi, mul, li, mv
}
```

- AsmvalReg:汇编语句的目的寄存器

  持有对 Reg 类型的寄存器的引用。可以作为一条汇编语句的源寄存器，因此实现了 Asmvalue 接口。也可以作为一条汇编语句的目的寄存器。

```
public class AsmvalReg implements Asmvalue{
    private Reg reg;
    public AsmvalReg(Reg reg){
        this.reg = reg;
    }
    @Override
    public String toString() {
        return reg.toString();
    }
    Run | Debug
```

- AsmvalImm:汇编语句中的立即数：

  该类型可以作为一条汇编语句中的操作数，比如 addi 等，因此实现了 Asmvalue 接口。

```
public class AsmvalImm implements Asmvalue{
    private int value;
    public AsmvalImm(int value){
        this.value = value;
    }
    @Override
    public String toString(){
        return String.valueOf(value);
    }
```

- Asmvalue：汇编语句中的操作数类型，为一个接口。实现该接口的类有 AsmvalReg 和 AsmvalImm 两类（即上面两个箭头对应的类）。

```
public interface Asmvalue {
    String toString();
}
```

- Reg：可用寄存器，采取枚举类型实现。

```
public enum Reg {
    t0, t1, t2, t3, t4, t5, t6, a0
}
```

2. BMap

此类的使用泛型。建立两个 hashmap，并对外提供接口。在增删操作时时钟保持两个 hashmap 的双射操作。在目标代码生成中，主要是为了查找中间代码中临时变量对应的 Reg 寄存器。

3.4.2 算法描述

1. 在 laodIR 中对中间代码进行优化

(1)辅助函数 isImm 说明

对于每一条中间代码，其内部数据结构中有 List<IRValue> operands。该函数主要根据 operands 中每一个对象的类返回 01 值，来表示是否为立即数。如果是对应的位置为 1，否则为 0。

```
private int isImm(List<IRValue> operands) {
    if (operands.size() == 1) {
        var p1 = operands.get(index:0);
        if (p1 instanceof IRImmediate) {
            return 1;
        } else if (p1 instanceof IRVariable) {
            return 0;
        }
    } else if (operands.size() == 2) {      You, 3天前 • reg only one use
        var p1 = operands.get(index:0);
        var p2 = operands.get(index:1);
        if (p1 instanceof IRImmediate && p2 instanceof IRImmediate) {
            return 11;
        } else if ((p1 instanceof IRImmediate) && !(p2 instanceof IRImmediate)) {
            return 10;
        } else if (!(p1 instanceof IRImmediate) && (p2 instanceof IRImmediate)) {
            return 1;
        } else if (!(p1 instanceof IRImmediate) && !(p2 instanceof IRImmediate)) {
            return 0;
        }
        throw new RuntimeException(message:"error");
    }
    return -1;
}
```

(2)优化策略

- 对于有两个操作数且操作数都是立即数的指令，直接进行求值得到结果，生成 MOV 汇编语句。

```
if (nowp.getKind().isBinary() && isImm(nowp.getOperands()) == 11) {
    var param1 = ((IRImmediate) nowp.getOperands().get(index:0)).getValue();
    var param2 = ((IRImmediate) nowp.getOperands().get(index:1)).getValue();
    var result = nowp.getResult();
    switch (nowp.getKind()) {
        case InstructionKind.ADD:
            instructions.add(Instruction.createMov(result, IRImmediate.of(param1 + param2)));      You, 3天前 • reg only one u
            break;
        case InstructionKind.SUB:
            instructions.add(Instruction.createMov(result, IRImmediate.of(param1 - param2)));
            break;
        case InstructionKind.MUL:
            instructions.add(Instruction.createMov(result, IRImmediate.of(param1 * param2)));
            break;
        default:
            break;
    }
}
```

- 对于 add 操作且为立即数+非立即数的操操作：

  交换两个操作数的位置，使其优化后的中间代码能够直接转为 addi 汇编语句。

```
// add 立即数+非立即数
else if (nowp.getKind() == InstructionKind.ADD && isImm(nowp.getOperands()) == 10) {
    var param1 = ((IRImmediate) nowp.getOperands().get(index:0));
    var param2 = ((IRVariable) nowp.getOperands().get(index:1));
    instructions.add(Instruction.createAdd(nowp.getResult(), param2, param1));
}
```

- 对于中间代码的 sub 指令，且为非立即数-立即数

  将这一条中间代码等价为两条中间代码。先将立即数移入一个中间变量中，在执行减法操作。

```
else if (nowp.getKind() == InstructionKind.SUB && isImm(nowp.getOperands()) == 10) {
    var param1 = ((IRImmediate) nowp.getOperands().get(index:0));
    var param2 = ((IRVariable) nowp.getOperands().get(index:1));
    var newvar = IRVariable.named(getcnt());
    instructions.add(Instruction.createMov(newvar, param1));
    instructions.add(Instruction.createSub(nowp.getResult(), newvar, param2));
```

- 对于两个操作数有一个立即数的乘法操作也类似

```
} else if (nowp.getKind() == InstructionKind.MUL &&
        (isImm(nowp.getOperands()) == 10 || isImm(nowp.getOperands()) == 1)) {
    if (isImm(nowp.getOperands()) == 10) {
        var param1 = ((IRImmediate) nowp.getOperands().get(index:0));
        var param2 = ((IRVariable) nowp.getOperands().get(index:1));
        var newvar = IRVariable.named(getcnt());
        instructions.add(Instruction.createMov(newvar, param1));
        instructions.add(Instruction.createMul(nowp.getResult(), newvar, param2));
    } else {
        var param1 = ((IRImmediate) nowp.getOperands().get(index:1));
        var param2 = ((IRVariable) nowp.getOperands().get(index:0));
        var newvar = IRVariable.named(getcnt());
        instructions.add(Instruction.createMov(newvar, param1));
        instructions.add(Instruction.createMul(nowp.getResult(), param2, newvar));
    }
```

- 其他情况下，直接将原始是的中间代码句子加入优化后的中间代码列表中。

2. 中间代码转为汇编代码 run()方法

(1) 辅助函数 ir2asm

主要用于将一个 IRvalue 类型转为 Asmvalue 类型。其中涉及到了寄存器分配问题，这一点在 3 中会进行说明（寄存器分配对应的算法封装在 getReg 函数中）。

```
public Asmvalue ir2asm(IRValue irval) {
    if (irval instanceof IRVariable) {
        return new AsmvalReg(getReg(irval));
    } else if (irval instanceof IRImmediate) {
        return new AsmvalImm(((IRImmediate) irval).getValue());
    }
    throw new RuntimeException(message:"error in ir2asm");
}
```

(2) 主要过程：

主要思路是对中间代码语句的类型采取 case 语句进行分支，然后根据操作数的数据类型选择产生对应的汇编代码。

- 以 *InstructionKind*.ADD 为例
  - 如果操作数 1 不是立即数，操作数 2 是立即数，则生成 addi 汇编语句
  - 如果两个操作数都不是立即数，生成 add 汇编语句
  - 注意：不可能两个操作数都是立即数，因为预处理中已对此进行优化。

```
switch (nowp.getKind()) {
    case InstructionKind.ADD:       You, 3天前 • reg only one use
        if (isImm(nowp.getOperands()) == 01) {
            // System.out.println(AsmKind.addi);
            asm.add(Asmsentence.createaddi(AsmKind.addi,
                    ir2asm(nowp.getOperands().get(index:0)),
                    ir2asm(nowp.getOperands().get(index:1)),
                    (AsmvalReg) ir2asm(nowp.getResult())));
        } else {
            // System.out.println(AsmKind.add);
            asm.add(Asmsentence.createadd(AsmKind.add,
                    ir2asm(nowp.getOperands().get(index:0)),
                    ir2asm(nowp.getOperands().get(index:1)),
                    (AsmvalReg) ir2asm(nowp.getResult())));
        }
        break;
```

- 其他的运算与之类似，除了 InstructionKind.MOV 情况。中间代码的 MOV 类型可用对应汇编代码中的 li 和 mv 类型。比如(MOV,a,8)转为汇编后是 li t0, 8；(MOV,c,$0)转为汇编后是 mv t1, t0。区别在于源对象是一个变量还是寄存器，因此需要进行分类。

```
case InstructionKind.MOV:
    if (isImm(nowp.getOperands()) == 1) {
        // System.out.println(AsmKind.li);
        asm.add(Asmsentence.createli(AsmKind.li,
                ir2asm(nowp.getOperands().get(index:0)),
                (AsmvalReg) ir2asm(nowp.getResult())));
    } else if (isImm(nowp.getOperands()) == 0) {
        // System.out.println(AsmKind.mv);      You, 3天前 • reg only one use
        asm.add(Asmsentence.createmv(AsmKind.mv,
                ir2asm(nowp.getOperands().get(index:0)),
                (AsmvalReg) ir2asm(nowp.getResult())));
    } else {
        throw new RuntimeException(message:"error");
    }
    break;
```

3. 寄存器分配策略 getReg()函数

(1) 变量说明：

- BMap<Reg, String> bMap = new BMap<>()：已用寄存器和中间代码已分配寄存器的变量的双向映射。
- nowIRindex:当前分析到的中间代码的序号，在分析当前 IR 语句时，用于查找在当前中间代码语句之后不会再用到的中间变量（应对需要重新分配寄存器的情况）。
- resindex：在一条语句中，可能会有多个中间变量需要重新分配寄存器。该变量指明可用寄存器中的已用索引。
- reslist：存储当前中间代码发生冲突时可用使用寄存器列表。与 resindex 一同使用。

(2) 主要过程：

- 首先在 bMap 中寻找，查看该变量是否已经分配寄存器，如果找到则返回
- 变量所有的寄存器（用于存储返回值的 a0 除外），查看是否已分配。如未分配，将其分配给变量。同时往 bMap 中添加双射。返回分配的寄存器。（前两步代码截图如下）

```
public Reg getReg(IRValue IRval) {
    if (!(IRval instanceof IRVariable)) {
        throw new RuntimeException(message:"寄存器分配错误");
    }
    IRval = (IRVariable) IRval;
    // 能够在Map中找到
    if (bMap.containsV(IRval.toString())) {
        return bMap.getK(IRval.toString());
    }
    // 不能
    for (var reg : Reg.values()) {
        if (reg == Reg.a0) {
            continue;
        }
        if (!(bMap.containsK(reg))) {
            bMap.put(reg, IRval.toString());
            return reg;
        }
    }
```

- 如果上述两步都不能找到空闲寄存器，则将之后中间代码中不再使用且目前占有就寄存器的变量进行指派。具体操作如下：

  ・当前执行的中间代码索引为 nowIRindex，在 getafterindexString()函数中从 nowIRindex+1 条开始往后寻找，获取每一条中间代码的 operands，并最后以集合 set(String)的形式返回，集合中就是之后会用到的中间代码变量。

  ・如果 resindex 为 0，遍历目前已经分配的寄存器对应的变量，如果该变量不在集合中，说明之后不会再用到该变量，可以对该变量对应的寄存器进行重新分配，将该寄存器添加到 reslist 中。

  ・在 reslist 中不为空后，根据 resindex 获取对应的可分配的寄存器。删除原有的键值对，添加新的键值对。将对应的新分配的寄存器返回。同时 resindex 自增 1。

  ・当上述操作依然找不到可用寄存器时，抛出错误。

```
if (resindex == 0) {
    // 生成可用re数组g
    reslist.clear();
    var s_set = getafterindexString();
    for (var reg : bMap.getusedKeys()) {      You, 3天前 • compiler finish
        var s = bMap.getV(reg);
        if (!s_set.contains(s)) {
            reslist.add(reg);
        }
        if (reslist.size() >= 3) {
            break;
        }
    }
}

if (resindex < reslist.size()) {
    bMap.removebyK(reslist.get(resindex));
    bMap.put(reslist.get(resindex), IRval.toString());
    return reslist.get(resindex++);
}
throw new RuntimeException(message:"寄存器已满");
```

4 实验结果与分析

1. 词法分析

在词法分析中,解析待编译的代码得到了不包含 type 属性的符号表以及源代码对应的 token 列表。正确完成了词法分析器功能。

（1） 输入

输入 1：码点文件（编码表）coding_map.csv	输入 2：待编译的源代码 input_code.txt

1 int
2 return
3 =
4 ,
5 Semicolon
6 +
7 -
8 *
9 /
10 (
11)
12 id
13 IntConst
	int result;
	int a;
	int b;
	int c;
	a = 8;
	b = 5;
	c = 3 - a;
	result = a * b - (3 + b) * (c - a);
	return result;

（2） 输出

➢ 输出 1 old_symbol_table.txt

由于此时还未进行语义分析，因此符号表的 type 属性均为空 null 值

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

➢ 输出 2 token.txt

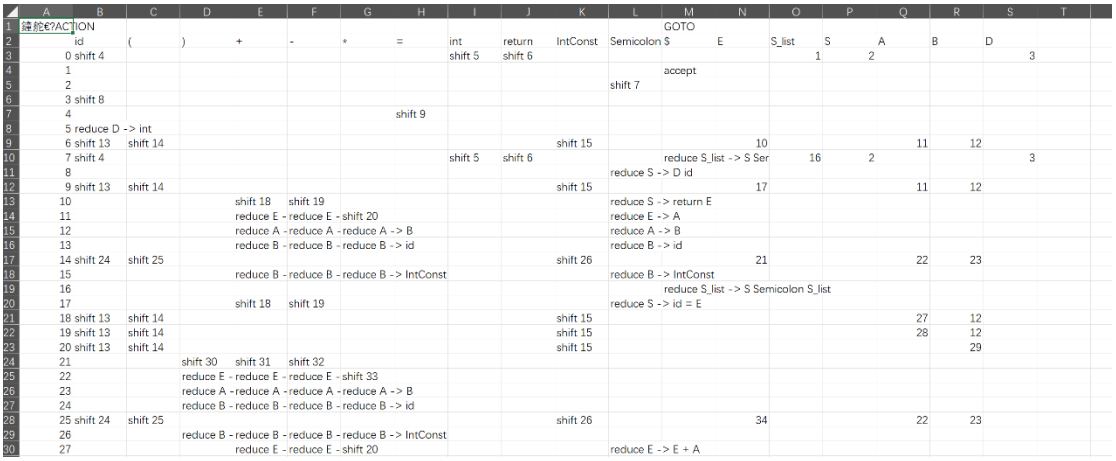
```
(int,)
(id,result)
(Semicolon,)
(int,)
(id,a)
(Semicolon,)
(int,)
(id,b)
(Semicolon,)
(int,)
(id,c)
(Semicolon,)
(id,a)
(=,)
(IntConst,8)
(Semicolon,)
(id,b)
(=,)
(IntConst,5)
(Semicolon,)
(id,c)
(=,)
(IntConst,3)
(-,)
(id,a)
(Semicolon,)
(id,result)
(=,)
(id,a)
(*,)
(id,b)
(-,)
((,)
(IntConst,3)
(+,)
(id,b)
(),)
(*,)
((,)
(id,c)
(-,)
(id,a)
(),)
(Semicolon,)
(return,)
(id,result)
(Semicolon,)
($,)
```

2. 语法分析

LR1_table.csv 是由第三方工具编译工作台生成的 LR1 分析表。新增的输出文件是规约使用的产生式列表。实际输出与标准输出一致，正确实现了自底向上的 LR1 语法分析的功能。

(1)输入

- 输入 1：词法分析中生成的 token.txt。当然，实现并不用重新读取文件来获得 token 序列，而是在语法分析中持有对存储 token 串的对象的引用。
- 输入 2：LR1_table.csv



编号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 Ser		16	2			3
8											reduce S -> D id							
9	shift 13	shift 14								shift 15			17			11	12	
10				shift 18	shift 19						reduce S -> return E							
11				reduce E -	reduce E -	shift 20					reduce E -> A							
12				reduce A -	reduce A -	reduce A -> B					reduce A -> B							
13				reduce B -	reduce B -	reduce B -> id					reduce B -> id							
14	shift 24	shift 25								shift 26			21			22	23	
15				reduce B -	reduce B -	reduce B -> IntConst					reduce B -> IntConst							
16												reduce S_list -> S Semicolon S_list						
17				shift 18	shift 19						reduce S -> id = E							
18	shift 13	shift 14								shift 15						27	12	
19	shift 13	shift 14								shift 15						28	12	
20	shift 13	shift 14								shift 15							29	
21			shift 30	shift 31	shift 32													
22			reduce E -	reduce E -	reduce E -	shift 33												
23			reduce A -	reduce A -	reduce A -	reduce A -> B												
24			reduce B -	reduce B -	reduce B -	reduce B -> id												
25	shift 24	shift 25								shift 26			34			22	23	
26			reduce B -	reduce B -	reduce B -	reduce B -> IntConst												
27				reduce E -	reduce E -	shift 20					reduce E -> E + A							

（2）输出

规约中按照先后顺序使用的产生式列表 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
35 B -> ( E )
36 A -> B
37 B -> id
38 A -> B
39 E -> A
40 B -> id
```

```
41 A -> B
42 E -> E - A
43 B -> ( E )
44 A -> A * B
45 E -> E - A
46 S -> id = E
47 B -> id
48 A -> B
49 E -> A
50 S -> return E
51 S_list -> S Semicolon
52 S_list -> S Semicolon S_list
53 S_list -> S Semicolon S_list
54 S_list -> S Semicolon S_list
55 S_list -> S Semicolon S_list
56 S_list -> S Semicolon S_list
57 S_list -> S Semicolon S_list
58 S_list -> S Semicolon S_list
59 S_list -> S Semicolon S_list
60 P -> S_list
```

3. 语义分析与中间代码生成

采取语法制导翻译，在语法分析的过程中，利用观察者模式进行语义分析和中间代码生成。

・在语义分析中，主要的任务是更新符号表。在 new_symbol_table.txt 中，每一个符号的 type 值都得到了更新。

・在中间代码生成中，得到了 intermediate_code.txt 中的文件。

・最后通过框架提供的 IR 模拟器模拟执行中间代码，得到了最终 return 的结果 144。

实际输出与标准输出一致。

(1) 输入

输入与语法分析输入相同。

(2) 输出

➢ 输出 1：new_symbol_table.txt

```
1  (a, Int)        You, 2周前 via
2  (b, Int)
3  (c, Int)
4  (result, Int)
5
```

➢ 输入 2：intermediate_code.txt

```
1  (MOV, a, 8)
2  (MOV, b, 5)
3  (SUB, $0, 3, a)
4  (MOV, c, $0)
5  (MUL, $1, a, b)
6  (ADD, $2, 3, b)
7  (SUB, $3, c, a)
8  (MUL, $4, $2, $3)
9  (SUB, $5, $1, $4)
10 (MOV, result, $5)
11 (RET, , result)
12
```

➢ 输出 3：ir_emulate_result.txt

```
ata > out > ir_emulate_result.txt
1   144
2
```

4. 目标代码生成

这一部分主要是编译器的后端，即将中间代码转为目标代码。对中间代码进行优化，并根据优化后的中间代码生成对应的目标代码。最终，生成的 asm 汇编代码能够在 RARS 平台上成功运行，并且将最终的结果，也就是 return 的值，保存在 a0 寄存器中，成功实现了目标代码生成功能。

(1) 输入：

中间代码生成中产生的 intermediate_code.txt。在目标代码生成中，引用了存储中间代码

的变量。

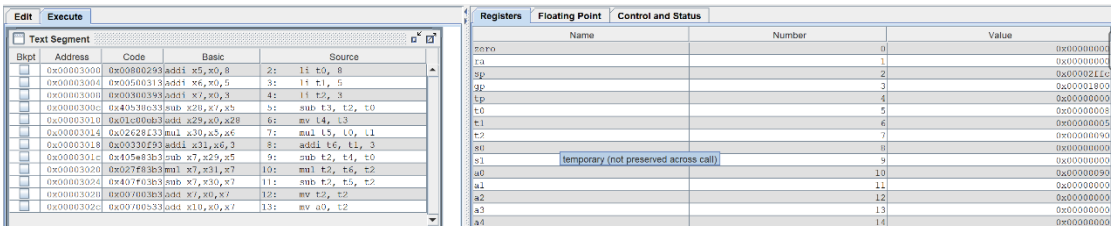
(2)输出：

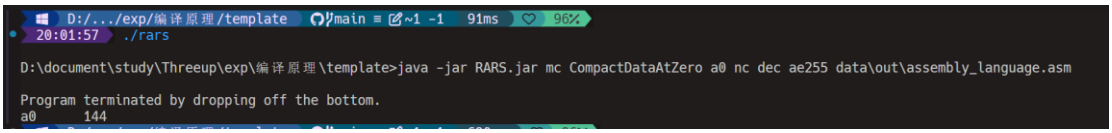
➢ 代码输出：assembly_language.asm

```
.text
    li t0, 8
    li t1, 5
    li t2, 3
    sub t3, t2, t0
    mv t4, t3
    mul t5, t0, t1
    addi t6, t1, 3
    sub t2, t4, t0
    mul t2, t6, t2
    sub t2, t5, t2
    mv t2, t2
    mv a0, t2

```

➢ RARS 输出：





5 实验中遇到的困难与解决办法

1. 问题 1 与解决方法

（1）问题描述：

在目标代码生成中。有类似情况：假设此时中间代码序号为 index，要将中间代码 ADD($1,$0,a)转为目标代码。此时$0,a 两个源变量已有对应的寄存器，且寄存器已经全部分配出去。$0,a 两个源变量在 index（即从 index+1 开始到结束）之后的中间代码中也不再使用。如果按照先给目标变量$1 寻找可用寄存器，再获得$0,a 两个变量对应寄存器的顺序来进行目标代码生成，可能会造成$1 寻找到的可用的寄存器是$0 或 a 对应的寄存器，然后修改 BMap 双射。从而造成寻找$0 或 a 的对应的寄存器发生错误。

（2）解决方案：

上述问题产生的根本原因是为目标变量寻找目标寄存器的过程中，修改了源变量与源寄存器之间的映射。解决此问题的关键在于目标变量以及原变量获取寄存器的顺序。可以先获得$0,a 两个源变量对应寄存器，再对目标变量$1 寻找可用寄存器。用这种方式，可以修正上述错误。在代码实现中，具体而言就是先对源变量调用 getReg()函数，再对目标变量调用 ir2ams()函数进行寄存器的分配（例子如下）。

```
asm.add(Asmsentence.createadd(AsmKind.add,
        ir2asm(nowp.getOperands().get(index:0)),
        ir2asm(nowp.getOperands().get(index:1)),
        (AsmvalReg) ir2asm(nowp.getResult()))));
```

修改过后，意味着目标寄存器和源寄存器可以是相同的，提高了寄存器的使用率（在生成的汇编语言文件中就有 sub t2, t4, t0；mul t2, t6, t2；sub t2, t5, t2 这样的语句序列）。

2. 问题 2 与解决方法

(1)问题描述：

在实验之初，框架代码量较大，类较多，在刚上手实验时需要花大量时间阅读代码，理解各类的具体作用、嵌套关系、继承关系、实现关系等。同时，进入实验 2 和实验 3，代码量虽然不大，但是实验内容十分抽象，不知道从何下手。

(2)解决方案：

对于框架代码，反复查看实验指导书中的 uml 图，梳理清楚各类之间的关系。在阅读实验指导书的过程中，对于各实验之间的关系也有了进一步的认识。对于实验 2 和实验 3，这两部分是对理论课知识的充分实践。深刻的理解理论课的知识是完成这两个实验的基础。在反复阅览理论课 ppt 以及实验指导书后，我大致有了实现的思路，还算顺利的完成了编码。

3. 收获和建议

(1) 一些收获：

- 使用 Java 语言完成这一个小的编译器，自己再一次感受到了面向对象思想以及设计模式的精妙之处。在 Instruction 类中，框架代码将构造方法私有，对外提供公开的静态方法构造实例，使得代码十分规整简洁。在目标代码生成的 Asmsentence 类中，我也采取了类似的设计；在实验 2 与实验 3 中，通过为语法分析器注册语义分析观察者和中间代码生成观察者，实现了语法制导翻译。通过观察者模式充分体现了语法制导翻译的定义(即翻译过程与语法分析紧密结合，通常通过在文法规则中附加语义动作来实现)。
- 将理论知识进行实践，从词法分析、语法分析、语义分析、中间代码生成再到目标代码生成，加深了自己对于编译器的工作过程的理解。

(2) 建议：

可以在第一次实验课上借一个小例子(比如一个简单的语句)，带领同学们大体认识整个大实验的编译器流程以及框架代码中各类的主要作用。

最后，感谢老师们的辛勤付出！