

## 1 实验目的与方法

**实验整体目标：**本学期我们的编译原理实验总共有 4 次，分别是词法分析、语法分析、语义分析与 IR 生成、汇编生成。每次实验均是在上一次实验的基础上进行扩展和完善（增量开发），逐步实现一个目标平台为 RISC-V 32（指令集为 RV32M）的小型 TXT 语言编译器。

**实验方法（全部 4 次实验均相同）：**所使用的编程语言为 Java，其中 JDK 版本为 JDK17 及以上。开发工具为 JetBrains 公司旗下著名的 Java 语言集成开发环境 IntelliJ IDEA（版本号为 2021.3.3, Ultimate Edition）。在对生成的汇编代码进行运行时，使用软件 RARS 进行模拟。也就是说，可以认为目标机器是 RARS。

### 1.1 词法分析器

**实验目的：**

- 加深对词法分析程序的功能及实现方法的理解。
- 对类 C 语言单词符号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整个过程中的作用。
- 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。

**备注：**类 C 语言是指 C 语言子集或者自定义的其他类似 C 语言语法的编程语言。

### 1.2 语法分析

**实验目的：**

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

### 1.3 典型语句的语义分析及中间代码生成

**实验目的：**

- 巩固对语义分析的基本功能和原理的认识。
- 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。
- 理解中间代码的表示形式，掌握三地址码的实现方式。

### 1.4 目标代码生成

**实验目的：**

- 加深对编译器总体结构的理解与掌握。
- 掌握常见的 RISC-V 指令的使用方法。
- 理解并掌握目标代码生成算法和寄存器选择算法。

## 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，对文件内的类 C 语言程序段进行词

法分析。处理 C 语言源程序，过滤掉无用符号，分解出正确的单词，以二元组的形式输出并存放到文件中。

- 输入：以文件形式存放的类 C 语言程序段 `input_code.txt`（即 TXT 语言源程序）、码点文件 `coding_map.csv`；
- 输出：以文件形式存放的 Token 串 `token.txt`、生成的简单符号表 `old_symbol_table.txt`。

#### 其它实验要求：

1. 词法分析器需要使用自动机实现，并忽略掉词法单元之间可能有的空白字符。
2. 生成词法单元迭代器，正确地将源语言中的每个标识符插入到符号表当中。
3. 不要求记录每个词法单元的行号、起始列号、结束列号，也不要求处理注释。

## 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，结合文法对输入单词符号串（即实验一的输出）进行语法分析。

输出推导过程中所用的产生式序列并保存在输出文件中。

- 输入：以文件形式存放的类 C 语言程序段 `input_code.txt`（即 TXT 语言源程序）、码点文件 `coding_map.csv`、语法文件 `grammar.txt`、第三方工具生成的 LR 分析表 `LR1_table.csv`；
- 输出：以文件形式存放的 Token 串 `token.txt`、生成的简单符号表 `old_symbol_table.txt`、归约过程的产生式列表 `parser_list.txt`。

### 其它实验要求：

1. 实现一个通用的 LR 语法分析驱动程序，它可以读入词法单元类别、任意语法以及与之匹配的 LR 分析表，随后将词法单元流作为输入，根据语法和 LR 分析表执行移进、归约、接受、出错动作，并在前三种动作执行时调用注册到其上的观察者的对应方法。
2. 该驱动程序作为被观察者，在动作发生时通知各个观察者，并调用各个观察者中相应的处理方法。
3. 若正确实现了本实验的功能，其中一个注册的观察者会按归约顺序输出所有记录到的产生式。

## 2.3 典型语句的语义分析及中间代码生成

### 实验内容：

使用实验二生成的文法，为语法正确的单词串设计翻译方案，完成语法制导翻译。

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。（注：更新后的符号表只需要保存 id 和 type 两个属性。）

实现声明语句、简单赋值语句和算术表达式的语义分析与中间代码生成（参考教材 P260 声明语句的翻译、P267 赋值语句的翻译）。

使用框架中的模拟器 IREmulator 验证生成的中间代码的正确性。

- 输入：以文件形式存放的类 C 语言程序段 `input_code.txt`（即 TXT 语言源程序）、码点文件 `coding_map.csv`、语法文件 `grammar.txt`、第三方工具生成的 LR 分析表 `LR1_table.csv`；

- 输出：以文件形式存放的 Token 串 `token.txt`、生成的简单符号表 `old_symbol_table.txt`、归约过程的产生式列表 `parser_list.txt`、语义分析后生成的符号表 `new_symbol_table.txt`、中间表示的模拟执行结果 `ir_emulate_result.txt`、中间表示(即中间代码) `intermediate_code.txt`。

**其它实验要求：**

1. 完成语义分析与中间代码生成。这两个过程分别对应两个观察者，它们被注册到语法分析器中，其特定方法会在语法分析执行特定动作时被调用，同时获得动作的相关信息。
2. 对于语义分析，需要在遇到每次声明的时候，于符号表中记录每个标识符的类型信息。
3. 对于中间代码生成，需要在遇到不同语法产生式时执行不同动作，产生中间表示指令列表。

## 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（即汇编指令）。

使用 RARS 运行生成的目标代码，验证结果的正确性。

- 输入：以文件形式存放的类 C 语言程序段 `input_code.txt`（即 TXT 语言源程序）、码点文件 `coding_map.csv`、语法文件 `grammar.txt`、第三方工具生成的 LR 分析表 `LR1_table.csv`；
- 输出：以文件形式存放的 Token 串 `token.txt`、生成的简单符号表 `old_symbol_table.txt`、归约过程的产生式列表 `parser_list.txt`、语义分析后生成的符号表 `new_symbol_table.txt`、中间表示的模拟执行结果 `ir_emulate_result.txt`、中间表示(即中间代码) `intermediate_code.txt`、生成的汇编代码 `assembly_language.asm`。

**其它实验要求：**

1. 完成汇编代码生成与寄存器的分配，目标平台为支持 RV32M 的 `riscv32` 机器。
2. 根据 RISC-V 的函数调用规范，在代码生成时只使用 Caller 保存的 `t0` 到 `t6` 寄存器作为汇编代码中任意使用的寄存器，同时使用 `a0` 寄存器存放程序的返回值。此外，规定 `data` 的开始地址为 `0x0`。
3. 使用不完备的寄存器分配方法或完备的寄存器分配方法均可以。其中，不完备的寄存器分配方案是指在对变量做寄存器指派时，若有寄存器空闲，则指派任意空闲寄存器进行分配；若无寄存器空闲，判断当前是否有寄存器被不再使用的变量占用，若有则指派任意满足该条件的寄存器，若无则直接报错。这种分配方法不能处理更复杂的程序，因此是不完备的。完备的寄存器分配方法则可以处理上述情况。

### 3 实验总体流程与函数功能描述

#### 3.1. 词法分析

##### 3.1.1. 编码表

词法分析器的输出是单词序列，在本次实验中，所输出的单词可以划分为 5 类，即关键字、运算符、分界符、标识符、常数。前 3 类（关键字、运算符、分界符）都是程序设计语言预先定义的，数量固定；后 2 类（标识符、常数）则由程序设计人员根据具体需要自行定义，数量可以为任意多个。为了让编译程序处理方便，通常需要按照一定方式对单词进行分类和编码，并在此基础上将单词表示成二元组（类别编码、单词值）的形式。本次实验要用到的编码表如表 1 所示。

单词名称	类别编码	单词值（若无单词值，则为“-”）
int	1	-
return	2	-
=	3	-
,	4	-
Semicolon	5	-
+	6	-
-	7	-
*	8	-
/	9	-
(	10	-
)	11	-

id	51	内部字符串
IntConst	52	整数值

表 1 本实验要用到的编码表

### 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。词法分析器也是根据构造出的正则文法来识别各个单词。

正则文法表示:

$G = (V, T, P, S)$ , 其中  $V = \{S, A, B, C, \text{digit}, \text{no\_0\_digit}, \text{char}\}$ ,  $T = \{\text{任意符号}\}$ ,  $P$  为产生式集合,  $S$  为文法开始符号, 并约定用  $\text{digit}$  表示数字 0 至 9, 用  $\text{no\_0\_digit}$  表示数字 1 至 9, 用  $\text{letter}$  表示字母 A 至 Z 以及 a 至 z。不同类型的单词所对应产生式如表 2 所示。(注意: 关键字先按标识符进行识别, 然后再判断是否为关键字)

单词类型	对应的产生式
标识符	$S \rightarrow \text{letter}A \mid \_A; A \rightarrow \text{letter}A \mid \text{digit}A \mid \varepsilon$
整常数	$S \rightarrow \text{no\_0\_digit}B; B \rightarrow \text{digit}B \mid \varepsilon$
运算符或分界符	$S \rightarrow C; C \rightarrow = \mid * \mid + \mid - \mid / \mid ( \mid ) \mid , \mid ;$

表 2 不同类型单词对应的产生式

### 3.1.3. 状态转换图

根据所设计出的识别词法单元的正则文法,可以得到不同词法单元类别对应的正则表达式,如表 3 所示。

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

表 3 不同词法单元类型对应的正则表达式

基于上述正则文法和正则表达式,可以构造出词法分析器在识别词法单元时要用到的 DFA,其状态转换图如图 1 所示。

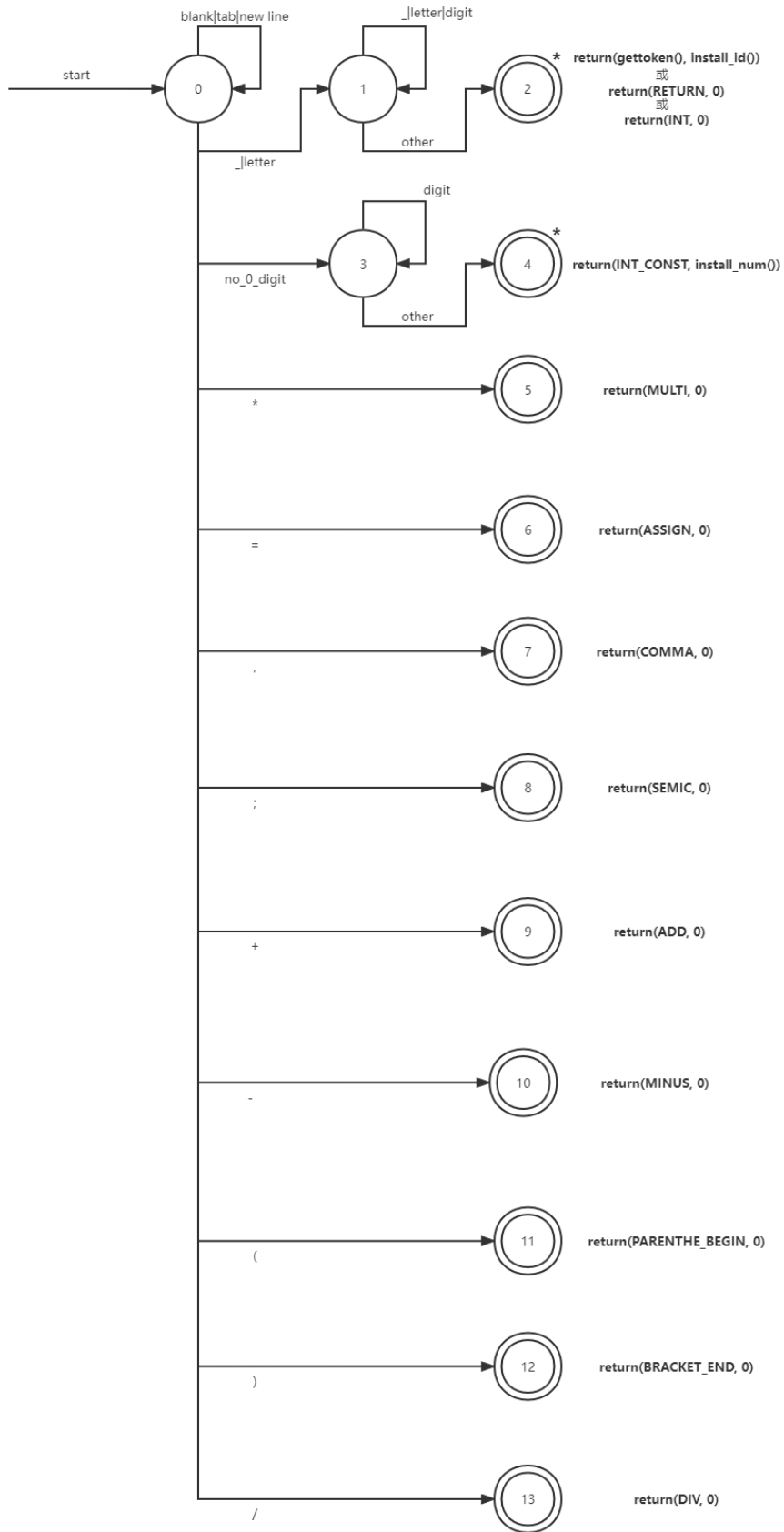


图 1 词法分析器识别词法单元的 DFA

关于此状态转换图的说明如下：

- 由两个圆圈包裹的状态都是接收状态。对于右上角带星号“\*”的接收状态，需要在读入 other 字符后回退一个字符；对于右上角不带星号“\*”的接收状态，不需要回退，也不需要通过 other 字符来判断当前读入是否应当结束，直接在需要识别的字符出现后就可以马上接收。
- 对于接收状态 2，除了需要接收用户自定义的标识符以外，还需要接受本次实验可能出现的 return 关键字以及 int 关键字。它们都是以 “\_” 或 letter 开头，并且可能在之后连接了 “\_”、letter 或 digit 字符。

#### 3.1.4. 主要函数流程

首先，修改 SymbolTable.java 文件，即完善与符号表相关的数据结构与操作方法。在 SymbolTable.java 文件中，get、add、has 函数分别用于从符号表中获得条目、向符号表中添加条目、判断符号表中是否存在某条目。另外，建立一个字符串 String 与符号表项 SymbolTableEntry 一一对应的 Map，便于根据给定的字符串查找对应的标识符。

随后，修改 LexicalAnalyzer.java 文件，完成词法分析程序的设计。先将待编译的源程序字符流读入并加载到缓冲区中，同时维护好单词开始指针、扫描指针这两种 Pointer。接着，源源不断地从缓冲区中读入字符，直至缓冲区的字符耗尽，并反复执行以下的判断或接收操作：

- 如果遇到空白字符（空格、制表符、换行符、回车符等），直接忽略它们，单词开始指针、扫描指针均向移动一个字符。



- 如果遇到字母或者下划线字符，单词开始指针保持不动，而扫描指针则继续向前移动（接下来只要遇到的是字符、数字或者下划线字符，扫描指针都可以继续向前扫描），直到遇到其它种类的字符。然后，获取单词开始指针、扫描指针之间的字符所形成的单词 `word`，并进行如下处理：
  - 如果 `word` 是“`int`”，说明识别出了关键字 `int`，对关键字 `int` 进行接收；
  - 如果 `word` 是“`return`”，说明识别出了关键字 `return`，对关键字 `return` 进行接收；
  - 如果 `word` 是其它字符串，说明识别出了标识符，对该标识符进行接收，并将其添加到符号表中。
  - 完成以上三种情况的判断和处理后，由于当前的扫描指针指向了 `other` 字符，需要调用 `retract` 函数将其回退一个字符。
- 如果遇到非 0 的数字字符，单词开始指针保持不动，而扫描指针则继续向前移动（接下来只要遇到的是数字字符，扫描指针都可以继续向前扫描），直到遇到其它种类的字符。然后，获取单词开始指针、扫描指针之间的字符所形成的单词 `word`，显然 `word` 就是识别出来的 `INT` 型常量，直接进行接收并将其添加到符号表中。同样，完成后需要调用 `retract` 函数让扫描指针回退一个字符。
- 如果遇到“`=`”、“`,`”、“`;`”、“`+`”、“`-`”、“`*`”、“`/`”、“`(`”、“`)`”等字符，则说明识别出了相对应的运算符或分界符，对它们进行接收。此外，由于扫描指针没有指向 `other` 字符，完成上述字符的接收后不需要进行回退。

当缓冲区中的字符耗尽后，向 `tokens` 列表中添加标志着 `end of file` 的词法单元 `eof`，结束词法分析程序的执行。

### 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。在本实验中，产生式  $S\_list \rightarrow S \text{ Semicolon } S\_list$  和产生式  $S\_list \rightarrow S \text{ Semicolon}$  的右部均含有  $S\_list$ ，为了便于判断是否已经归约到最初的开始符，需要编写拓展文法。

得到的拓展文法如表 4 所示。

序号	产生式
0	$P \rightarrow S\_list$
1	$S\_list \rightarrow S \text{ Semicolon } S\_list$
2	$S\_list \rightarrow S \text{ Semicolon}$
3	$S \rightarrow D \text{ id}$
4	$D \rightarrow \text{int}$
5	$S \rightarrow \text{id} = E$
6	$S \rightarrow \text{return } E$
7	$E \rightarrow E + A$
8	$E \rightarrow E - A$
9	$E \rightarrow A$
10	$A \rightarrow A * B$

11	$A \rightarrow B$
12	$B \rightarrow ( E )$
13	$B \rightarrow id$
14	$B \rightarrow IntConst$

表 4 对文法进行拓展后得到的拓展文法

### 3.2.2. LR1 分析表

要完成语法分析程序的设计，还需要得到一个与扩展文法相对应的 LR(1) 分析表。可以使用第三方工具“编译工作台”构建一个包含了 ACTION 和 GOTO 的 LR(1) 分析表，如图 2 所示。

状态	ACTION id	( )	+	-	*	=	int shift 5	return shift 6	IntConst	Semicolon	\$	GOTO E	S_list 1	S 2	A 3	B 4	D 5
0	shift 4																
1											accept						
2										shift 7							
3	shift 8																
4						shift 9											
5	reduce D -> int																
6	shift 13	shift 14							shift 15			10			11	12	
7	shift 4						shift 5	shift 6		reduce S_list -> S Semicolon			16	2			3
8										reduce S -> D id							
9	shift 13	shift 14							shift 15			17			11	12	
10			shift 18	shift 19						reduce S -> return E							
11			reduce E -> A	reduce E -> A	shift 20					reduce E -> A							
12			reduce A -> B	reduce A -> B	reduce A -> B					reduce A -> B							
13			reduce B -> id	reduce B -> id	reduce B -> id					reduce B -> id							
14	shift 24	shift 25							shift 26			21			22	23	
15			reduce B -> IntConst	reduce B -> IntConst	reduce B -> IntConst					reduce B -> IntConst	reduce S_list -> S Semicolon S_list						
16																	
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	12	
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	reduce A -> A * B				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	reduce A -> A * B											
38			reduce B -> ( E )	reduce B -> ( E )	reduce B -> ( E )	reduce B -> ( E )											

图 2 使用第三方工具生成的 LR(1) 分析表

### 3.2.3. 状态栈和符号栈的数据结构

#### 状态栈的数据结构：

由于所给的代码框架中提供了关于 LR 分析表中状态的类 Status（定义在文件 Status.java 中），因此可以直接维护一个关于类 Status 的栈来实现关于状态栈的数据结构。故可以通过下面这行代码来定义状态栈。

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

#### 符号栈的数据结构：

在处理符号栈时，我们可能需要同时将 Token 和 NonTerminal 装在栈中，但是 Token 和 NonTerminal 除了 Object 之外没有共同祖先类，因此我们期望有一个类似 Union<Token, NonTerminal>的结构。可以自定义一个 Symbol 类来完成这个共用体的功能，这样就能把 Token 和 NonTerminal 同时装在栈中，实现符号栈的数据结构。

自定义 Symbol 类的代码如下所示。

```
class Symbol {
    private final Token token;
    private final NonTerminal nonTerminal;

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

    public Symbol(Token token) {
```

```
        this(token, null);
    }

    public Symbol(NonTerminal nonTerminal) {
        this(null, nonTerminal);
    }

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

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

    public Token getToken() {
        return this.token;
    }

    public NonTerminal getNonTerminal() {
        return this.nonTerminal;
    }
}
```

自定义 Symbol 类以后，定义符号栈也就不难了，代码如下所示。

```
/**
 * 符号栈
 */
private final Stack<Symbol> symbolStack = new Stack<>();
```

#### 3.2.4. LR 驱动程序流程描述

首先，加载词法分析程序分析得到的词法单元流、扩展文法对应的产生式列表、第三方工具生成的 LR(1) 分析表。其中，词法单元流被保存到输入缓冲区 inputBuffer 中。用 getInit 方法获取起始状态，压入状态栈 statusStack 中；将(new Symbol(Token.eof()))压入符号栈 symbolStack 中（相当于压入“#”）。

接着，当输入缓冲区 `inputBuffer` 非空的时候，反复执行以下步骤：

- 获取当前状态栈的栈顶元素、输入缓冲区的第一个元素（但是均不能弹出相应元素）`currentStatus`、`currentToken`。
- 调用加载的 LR(1) 表的 `getAction` 方法，根据 `currentStatus`、`currentToken` 确定接下来要执行的动作：
  - 如果是移进动作（即 `Shift`），调用 `callWhenInShift` 方法通知各个观察者，同时调用 `action.getStatus` 方法获得要转换到的状态 `shiftTo`，将 `shiftTo` 和 `(new Symbol(inputBuffer.pop()))` 分别压入状态栈、符号栈。
  - 如果是归约动作（即 `Reduce`），调用 `callWhenInReduce` 方法通知各个观察者，同时调用 `action.getProduction` 方法获取归约所用的产生式，然后调用 `getGoto` 方法（相当于查询 `Goto` 表）获取规约后要转移到的状态 `shiftTo`。先从状态栈、符号栈中弹出被归约的元素，然后再分别移入 `shiftTo`、`(new Symbol(production.head()))`。
  - 如果是接受动作（即 `Accept`），调用 `callWhenInAccept` 方法通知各个观察者，同时将标志位 `isAccept` 的值置为 `true`（为 `true` 时会提前结束循环，因为已经成功接收，语法分析可以结束）。
  - 如果是出错操作（即 `Error`），会抛出运行时异常，并提示语法分析出错。

反复执行以上步骤。当输入缓冲区中的 `Token` 被耗尽，或者标记位 `isAccept` 的值为 `true` 时，语法分析结束。将语法分析过程中得到的一系列产生式按其归约的先后顺序输出到文件中。

### 3.3. 语义分析和中间代码生成

#### 3.3.1. 翻译方案

本次实验涉及到的只有声明语句、简单赋值语句、算术运算语句这三种语句的语法制导翻译。也就是说，需要处理的都是综合属性。故可以使用 S-属性定义的自底向上翻译方案，只需要将每个语义规则放到相应产生式的末尾，在对产生式进行规约的时候执行相应的语义动作，就可以得到翻译模式。

由此，可以得到高效的自底向上翻译方案，如表 5 所示。

序号	翻译方案
0	$P \rightarrow S\_list$
1	$S\_list \rightarrow S \text{ Semicolon } S\_list$
2	$S\_list \rightarrow S \text{ Semicolon}$
3	$S \rightarrow D \text{ id } \{p = \text{lookup}(\text{id.name}); \text{ if } p \neq \text{nil} \text{ then enter}(\text{id.name}, D.\text{type}) \text{ else error}\}$
4	$D \rightarrow \text{int } \{D.\text{type} = \text{int};\}$
5	$S \rightarrow \text{id} = E \{p = \text{lookup}(\text{id.name}); \text{ if } p == \text{nil} \text{ then error else gencode}(\text{id.val} = E.\text{val})\}$
6	$S \rightarrow \text{return } E \{\text{gencode}(\text{return } E.\text{val});\}$
7	$E \rightarrow E_1 + A \{E.\text{val} = \text{newtemp}(); \text{ gencode}(E.\text{val} = E_1.\text{val} + A.\text{val})\}$
8	$E \rightarrow E_1 - A \{E.\text{val} = \text{newtemp}(); \text{ gencode}(E.\text{val} = E_1.\text{val} - A.\text{val})\}$
9	$E \rightarrow A \{E.\text{val} = A.\text{val}\}$

10	$A \rightarrow A_1 * B \{A.val = newtemp(); \text{ gencode}(A.val = A_1.val * B.val)\}$
11	$A \rightarrow B \{A.val = B.val;\}$
12	$B \rightarrow ( E ) \{B.val = E.val;\}$
13	$B \rightarrow id \{p = \text{lookup}(id.name); \text{ if } p \neq \text{nil} \text{ then } B.val = id.val \text{ else error}\}$
14	$B \rightarrow \text{IntConst} \{B.val = \text{IntConst.lexval};\}$

表 5 S-属性定义的自底向上翻译方案

### 3.3.2. 语义分析和中间代码生成使用的数据结构

语义分析使用的数据结构：

在文件 SemanticAnalyzer.java 中，主要完成声明语句的翻译，因此翻译时对应的综合属性为源语言的变量类型 SourceCodeType。这就意味着在翻译过程中，语义分析栈需要同时保存 type 属性信息和词法单元 Token，我们又可以仿照实验二语法分析中的做法，定义一个类似于共用体的类完成这一功能，代码如下所示。

```
class SemanticStackEntry {
    private final Token token;
    private final SourceCodeType type;

    public SemanticStackEntry(Token token) {
        this.token = token;
        this.type = null;
    }

    public SemanticStackEntry(SourceCodeType type) {
        this.token = null;
        this.type = type;
    }
}
```



```
}

public SemanticStackEntry() {
    this.token = null;
    this.type = null;
}

public Token getToken() {
    if (this.token == null) {
        throw new RuntimeException("Not a token!");
    }
    return this.token;
}

public SourceCodeType getType() {
    if (this.type == null) {
        throw new RuntimeException("Not a type!");
    }
    return this.type;
}
}
```

同时，可以定义语义分析栈，如下所示。

```
/**
 * 语义分析栈
 */
private final Stack<SemanticStackEntry> semanticStack = new
Stack<>();
```

### 中间代码生成使用的数据结构：

在文件 IRGenerator.java 中，主要完成简单赋值语句、算术运算语句的翻译，因此翻译时对应的综合属性为中间代码指令中的变量或立即数（即 IRValue）。这就意味着在翻译过程中，进行中间代码生成的栈需要同时保存 IRValue 属性信息和词法单元 Token，我们又可以仿照实验二语法分析中的做法，定义一个类似于共用体的类完成这一功能，代码如下所示。

```
class IRGeneratorStackEntry {
    private final Token token;
    private final IRValue irValue;

    public IRGeneratorStackEntry(Token token) {
        this.token = token;
        this.irValue = null;
    }

    public IRGeneratorStackEntry(IRValue irValue) {
        this.token = null;
        this.irValue = irValue;
    }

    public IRGeneratorStackEntry() {
        this.token = null;
        this.irValue = null;
    }

    public Token getToken() {
        if (this.token == null) {
            throw new RuntimeException("Not a token!");
        }
        return this.token;
    }

    public IRValue getIrValue() {
        if (this.irValue == null) {
            throw new RuntimeException("Not an irValue!");
        }
        return this.irValue;
    }
}
```

由此，可以定义进行中间代码生成时所要用的栈、形成的中间代码指令列表，如下所示。

```
/**
 * 生成得到的 Instruction 列表
 */
private final List<Instruction> instructions = new ArrayList<>();
```

```
private final Stack<IRGeneratorStackEntry> irGeneratorStack = new  
Stack<>();
```

### 3.3.3. 主要流程描述（两个观察者的实现）

语义分析（对应观察者 SemanticAnalyzer）的实现：

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

被观察者调用 whenAccept 方法时，语义分析程序不需要执行任何动作（空实现），因为遇到 Accept 时语法分析已经可以结束。

被观察者调用 whenReduce 方法时，获取此次归约的产生式的索引，并根据索引判断归约用的是哪一种产生式。

对于产生式“ $S \rightarrow D \text{ id}$ ”，获取 id 对应的 token（进而获得 id 的 idText）和 D 对应的 type，再根据 idText 和 type 更新标识符 id 在符号表当中的信息（在本实验中则是更新标识符的变量类型）。弹出了 D 和 id 对应的元素后，需要向语义分析栈中压入空记录进行占位。对于产生式“ $D \rightarrow \text{int}$ ”，则需要将 D 符号的 type 属性压入栈。对于使用其它产生式进行归约的情况，由于语义分析程序只需要翻译声明语句，因此直接进行弹栈并压入空记录占位。

被观察者调用 whenShift 方法时，直接使用 SemanticStackEntry 类对语法分析程序传递过来的 currentToken 变量进行包装，将其压入语义分析栈中，以便归约时能再次弹出并获得有关信息。

## 中间代码生成（对应观察者 IRGenerator）的实现：

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

被观察者调用 whenAccept 方法时，中间代码生成程序不需要执行任何动作（空实现），因为遇到 Accept 时中间代码生成已经可以结束。

被观察者调用 whenShift 方法时，直接使用 IRGeneratorStackEntry 类对语法分析程序传递过来的 currentToken 变量进行包装，将其压入中间代码生成时所要用到的栈中，以便归约时能再次弹出并获得有关信息。

被观察者调用 whenReduce 方法时，获取此次归约的产生式的索引，并根据索引判断归约用的是哪一种产生式。

对于产生式 “ $S \rightarrow id = E$ ”，弹栈，并获得 id 和 E 对应的 IRValue 信息，由此可以调用 Instruction 类的 createMov 方法生成相应的中间代码指令，并将其添加到 instructions 列表中。最后，向 irGeneratorStack 中压入空记录以进行占位。

对于产生式 “ $S \rightarrow \text{return } E$ ”，弹栈，并获得 E 对应的 IRValue 信息，由此可以调用 Instruction 类的 createRet 方法生成相应的中间代码指令，并将其添加到 instructions 列表中。最后，向 irGeneratorStack 中压入空记录来占位。

对于产生式 “ $E \rightarrow E + A$ ”、“ $E \rightarrow E - A$ ”、“ $A \rightarrow A * B$ ”，均是先弹栈，

然后获得产生式右部符号对应的 IRValue 信息，然后再调用 IRVariable 的 temp 方法生成临时变量以存放运算结果。由此，可以调用 Instruction 类的 createAdd、createSub、createMul 方法生成相应的中间代码指令，并将其添加到 instructions 列表中。最后，将(new IRGeneratorStackEntry(result))压栈。

对于产生式 “E  $\rightarrow$  A”、“A  $\rightarrow$  B”、“B  $\rightarrow$  id”、“B  $\rightarrow$  IntConst”、“B  $\rightarrow$  ( E )”，均是先弹栈，获取右部符号对应的 IRValue 信息后，再使用 IRGeneratorStackEntry 类进行包装，重新压入栈中。

由于中间代码生成程序只需要完成简单赋值语句、算术运算语句的翻译，因此对于其它类型的产生式，不需要关心其语义动作，直接进行弹栈处理并压入空记录进行占位。

### 3.4. 目标代码生成

#### 3.4.1. 主要流程描述

自定义的数据结构：

类型	名称	主要成员变量	简单描述
enum	Register	无	寄存器
enum	Opcode	无	汇编指令的操作码
class	Operand	Register register; Integer immediate;	汇编指令的操作数
class	AsmInstruction	Opcode opcode; List<Operand> operands;	汇编指令
List<AsmInstruction>	asmInstructions	无	生成的汇编指令列表
List<Instruction>	instructions	无	存放经过预处理后的中

			间代码指令
Map<Register, IRVariable>	registerAllocTable	无	寄存器分配表
List<IRVariable>	variablesInUse	无	保存仍要被使用的 IRVariable，便于寄存器分配时进行判断

### 主要流程描述：

首先读入前端提供的中间代码并进行预处理，以简化后续汇编指令生成过程中的判断逻辑。对于二元指令，如果两个操作数都是立即数，直接进行求值得到结果，然后替换成 MOV 指令；如果左操作数是立即数，右操作数是变量，且运算类型是减法或者乘法，需要调用 IRVariable 的 temp 方法生成一个临时变量 tempResult，然后前插一条 MOV 指令，将立即数的值赋给 tempResult，然后用 tempResult 替换原立即数，将指令调整为无立即数指令；对于右操作数是立即数，左操作数是变量，且运算类型是乘法的情况，也同样要使用类似的方法进行调整，生成临时变量 tempResult 进行替换并前插一条 MOV 指令，具体不再赘述。其它类型中间代码指令则正常读入。

此外，预处理步骤还要保存全部中间指令用到的 IRVariable 到列表 variablesInUse 中，从而能够判断某个变量是否还需要在之后被使用，便于进行寄存器的分配。

本次实验所使用的寄存器分配方案是不完备的分配方案。具体来说，就是对于一个待分配寄存器的变量，首先查询当前是否还剩余有空闲的寄存器，如果有则直接分配。如果已经没有空闲寄存器，查询寄存器分配表 registerAllocTable，并根据 variablesInUse 列表判断是否已经有变量不再被使用，若有则直接分配该变量占有的寄存器；否则，将无法继续进行寄存器的分配并抛出异常（因此实现的是不完备的寄存器分配方法）。

生成汇编代码的主要过程如下：

遍历 instructions 列表，对于其中的中间代码指令，判断指令的操作类型，根据不同的运算类型来进行生成。

对于 ADD 指令，如果两个操作数都是变量，直接生成对应的 add 指令；如果

其中一个操作数是立即数，另一个操作数是变量，则可以生成对应的 `addi` 指令（其中左操作数为立即数时需要调换左、右操作数的顺序）。两个操作数都是立即数的情况已经在预处理阶段被排除。

对于 `SUB` 指令，如果两个操作数都是变量，直接生成对应的 `sub` 指令；如果左操作数是变量，右操作数是立即数，则可以生成对应的 `subi` 指令。左操作数为立即数、右操作数为变量的情况，以及两个操作数都是立即数的情况均已经在预处理阶段被排除。

对于 `MUL` 指令，可以直接生成对应的 `mul` 指令。这是因为所有存在操作数为立即数的情况均已经在预处理阶段被排除。

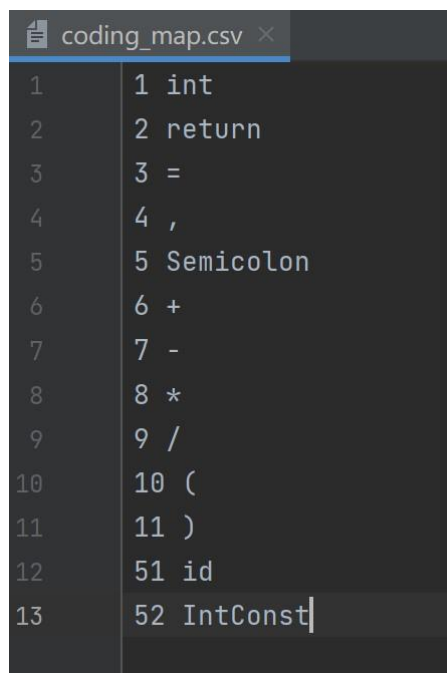
对于 `RET` 指令和 `MOV` 指令，两者生成的汇编指令都只有两个操作数，即源操作数和目的操作数（其中 `RET` 指令的目的操作数固定为存放返回值的寄存器 `a0`）。对于源操作数 `from`，如果它的类型为立即数，则直接生成 `li` 指令（对应“`li rd, imm`”）；如果它的类型为变量，则直接生成 `mv` 指令（对应“`mv rd, rsl`”）。

完成所有汇编指令的生成后，将它们输出到文件中。

#### 4 实验结果与分析

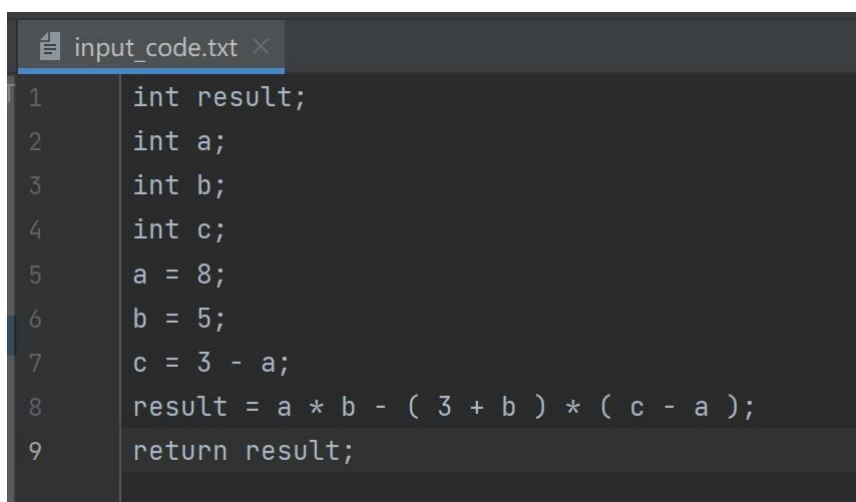
##### 词法分析

输入 1: 码点文件 coding\_map.csv



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

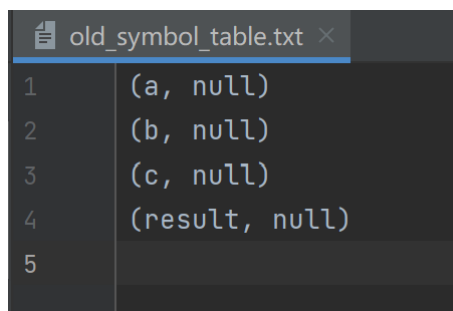
输入 2: 待编译的代码 input\_code.txt



```
1 int result;
2 int a;
3 int b;
4 int c;
5 a = 8;
6 b = 5;
7 c = 3 - a;
8 result = a * b - ( 3 + b ) * ( c - a );
9 return result;
```

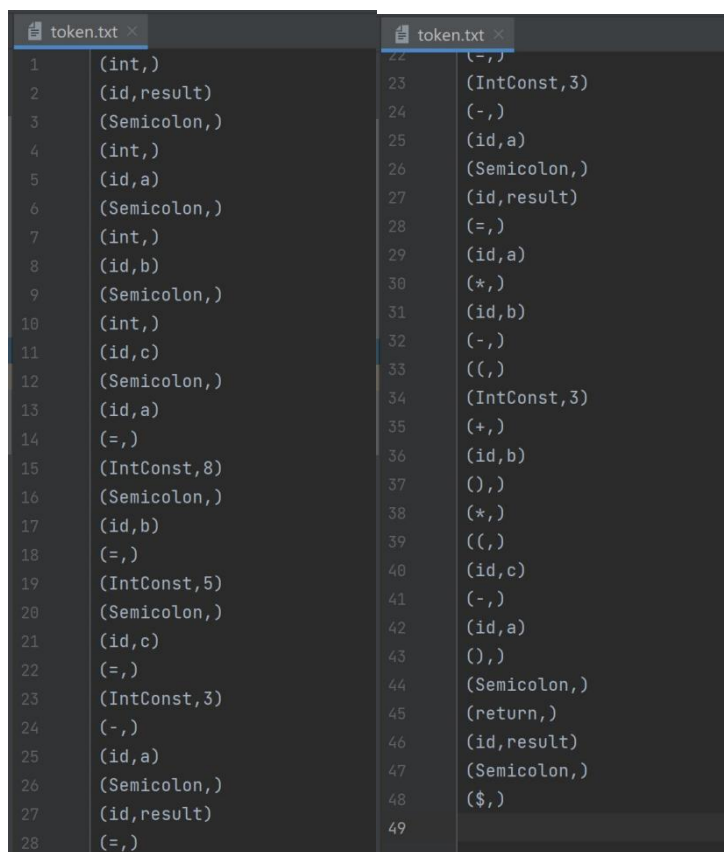


## 输出 1: 符号表 old\_symbol\_table.txt



1	(a, null)
2	(b, null)
3	(c, null)
4	(result, null)
5	

## 输出 2: 词法单元列表 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	(-,)
30	(IntConst,3)
31	(-,)
32	(id,a)
33	(=,)
34	(id,b)
35	(*,)
36	(id,b)
37	(-,)
38	((,)
39	(id,c)
40	(-,)
41	(id,a)
42	(-,)
43	(Semicolon,)
44	(return,)
45	(id,result)
46	(Semicolon,)
47	(\$,)
48	
49	

## 分析:

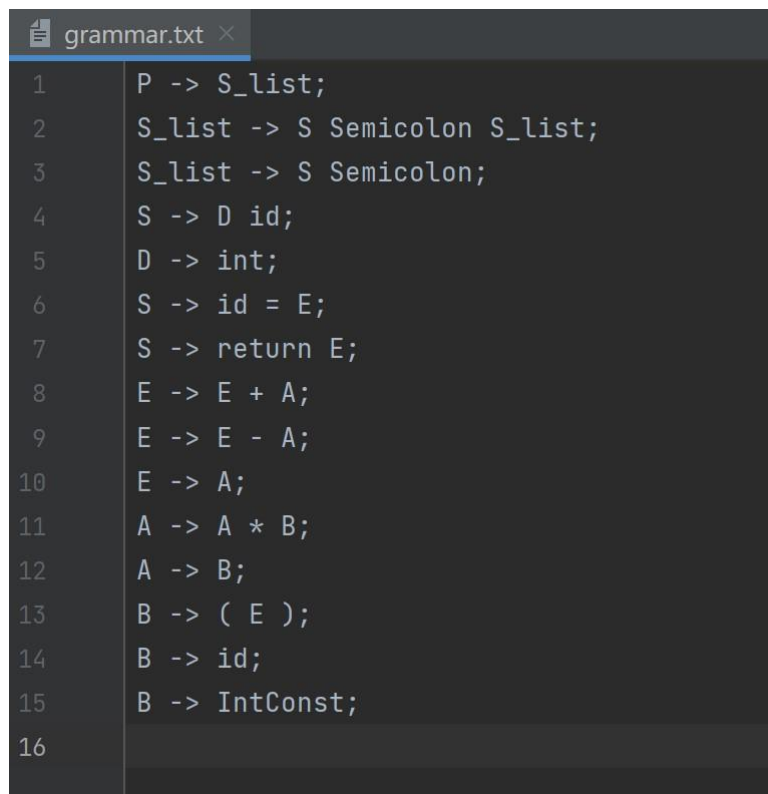
输入文件 coding\_map 规定了所有待识别的词法单元的分类, 以及它们各自对应的编码。输入文件 input\_code.txt 则是待编译的代码。所有输出文件均与 std 目录下的对应文件相一致, 可见正确完成了词法分析器的功能, 解析输入的源程序得到词法单元, 并生成了一个暂时不包含 type 类型信息的符号表。

## 语法分析

输入 1: 码点文件 coding\_map.csv (与词法分析相一致)

输入 2: 待编译的代码 input\_code.txt (与词法分析相一致)

输入 3: 语法文件 grammar.txt



```
grammar.txt x
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
```

## 输入 4: LR 分析表 LR1\_table.csv

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

输出 1: 符号表 old\_symbol\_table.txt (与词法分析相一致)

输出 2: 词法单元列表 token.txt (与词法分析相一致)

## 输出 3: 归约得到的产生式列表 parser\_list.txt

1	D -> int	27	A -> A * B	54	E -> E + A
2	S -> D id	28	E -> A	55	B -> ( E )
3	D -> int	29	B -> IntConst	56	A -> B
4	S -> D id	30	A -> B	57	B -> id
5	D -> int	31	E -> A	58	A -> B
6	S -> D id	32	B -> id	59	E -> A
7	D -> int	33	A -> B	60	B -> id
8	S -> D id	34	E -> E + A	61	A -> B
9	B -> IntConst	35	B -> ( E )	62	E -> E - A
10	A -> B	36	A -> B	63	B -> ( E )
11	E -> A	37	B -> id	64	A -> A * B
12	S -> id = E	38	A -> B	65	E -> E - A
13	B -> IntConst	39	E -> A	66	S -> id = E
14	A -> B	40	B -> id	67	B -> id
15	E -> A	41	A -> B	68	A -> B
16	S -> id = E	42	E -> E - A	69	E -> A
17	B -> IntConst	43	B -> ( E )	70	S -> return E
18	A -> B	44	A -> A * B	71	S_list -> S Semicolon
19	E -> A	45	E -> E - A	72	S_list -> S Semicolon S_list
20	B -> id	46	S -> id = E	73	S_list -> S Semicolon S_list
21	A -> B	47	B -> id	74	S_list -> S Semicolon S_list
22	E -> E - A	48	A -> B	75	S_list -> S Semicolon S_list
23	S -> id = E	49	E -> A	76	S_list -> S Semicolon S_list
24	B -> id	50	S -> return E	77	S_list -> S Semicolon S_list
25	A -> B	51	S_list -> S Semicolon	78	S_list -> S Semicolon S_list
26	B -> id	52	S_list -> S Semicolon S_list	79	S_list -> S Semicolon S_list
27	A -> A * B	53	S_list -> S Semicolon S_list	80	P -> S_list
28	E -> A	54	S_list -> S Semicolon S_list		

## 分析:

新增加的输入文件是语法文件以及使用第三方工具(编译工作台)生成的 LR 分析表。新增加的输出文件则是生成的产生式列表 parser\_list.txt。对比分析可知,该文件与 std 目录下的对应文件内容相一致,可见正确实现了语法分析的功能,按照归约的先后顺序输出产生式到文件中。

## 语义分析与中间代码生成

输入 1: 码点文件 coding\_map.csv (与词法分析相一致)

输入 2: 待编译的代码 input\_code.txt (与词法分析相一致)

输入 3: 语法文件 grammar.txt (与语法分析相一致)

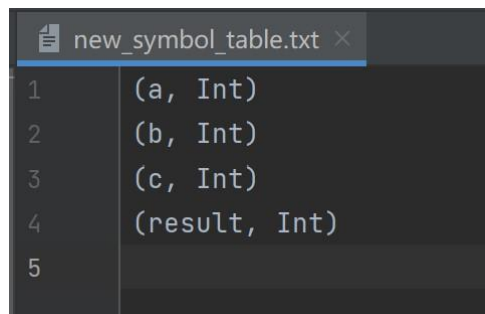
输入 4: LR 分析表 LR1\_table.csv (与语法分析相一致)

输出 1: 符号表 old\_symbol\_table.txt (与词法分析相一致)

输出 2: 待编译的代码 input\_code.txt (与词法分析相一致)

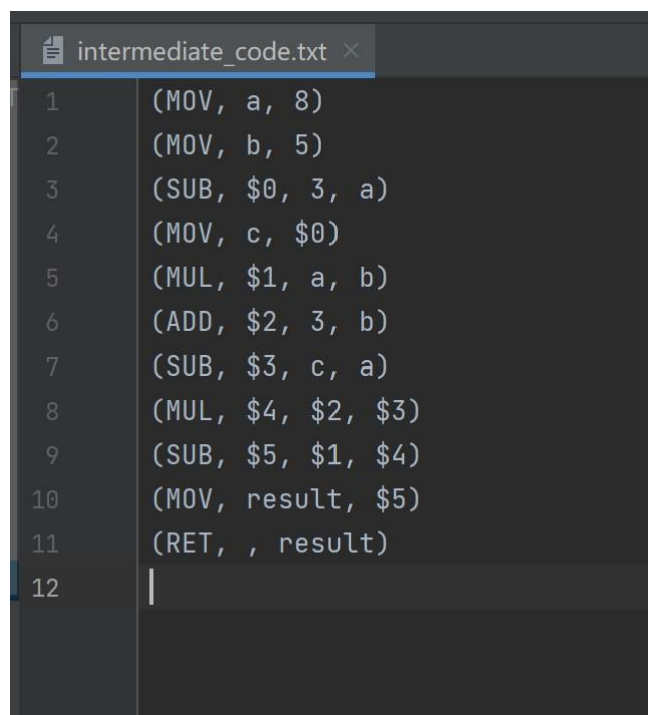
输出 3: 归约得到的产生式列表 parser\_list.txt(与语法分析一致)

输出 4: 语义分析后得到的新符号表 new\_symbol\_table.txt

A screenshot of a text editor window titled 'new\_symbol\_table.txt'. It contains a list of five entries, each on a new line, numbered 1 to 5 on the left. The entries are: (a, Int), (b, Int), (c, Int), (result, Int), and an empty line for entry 5.

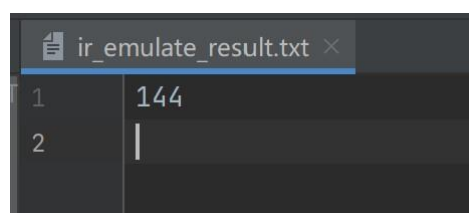
1	(a, Int)
2	(b, Int)
3	(c, Int)
4	(result, Int)
5	

输出 5: 生成的中间代码 intermediate\_code.txt

A screenshot of a text editor window titled 'intermediate\_code.txt'. It contains 12 lines of assembly-like code, numbered 1 to 12 on the left. The code is: (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), and an empty line for entry 12.

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 6: 中间代码模拟执行的结果 ir\_emulate\_result.txt

A screenshot of a text editor window titled 'ir\_emulate\_result.txt'. It contains two lines, numbered 1 and 2 on the left. Line 1 contains the value 144, and line 2 is empty.

1	144
2	

## 分析:

实验三没有新增的输入文件，所有的输入文件均与实验二相一致。新增加的输出文件包括有新符号表、生成的中间代码、中间代码模拟执行的结果。对比分析可知，所有输出文件均与 std 目录下的对应文件内容相一致，可见正确实现了语义分析与中间代码生成的功能。也就是说，程序能够准确进行语义分析，并更新符号表中关于变量类型 type 的信息。此外，观察文件 input\_code.txt 中的代码可知，返回值为  $8 \times 5 - (3 + 5) * (3 - 8 - 8) = 144$ ，而中间代码模拟运行的结果也是 144，确实符合要求。

## 目标代码生成

输入 1: 码点文件 coding\_map.csv (与词法分析相一致)

输入 2: 待编译的代码 input\_code.txt (与词法分析相一致)

输入 3: 语法文件 grammar.txt (与语法分析相一致)

输入 4: LR 分析表 LR1\_table.csv (与语法分析相一致)

输出 1: 符号表 old\_symbol\_table.txt (与词法分析相一致)

输出 2: 待编译的代码 input\_code.txt (与词法分析相一致)

输出 3: 归约得到的产生式列表 parser\_list.txt (与语法分析一致)

输出 4: 新符号表 new\_symbol\_table.txt (与实验三一致)

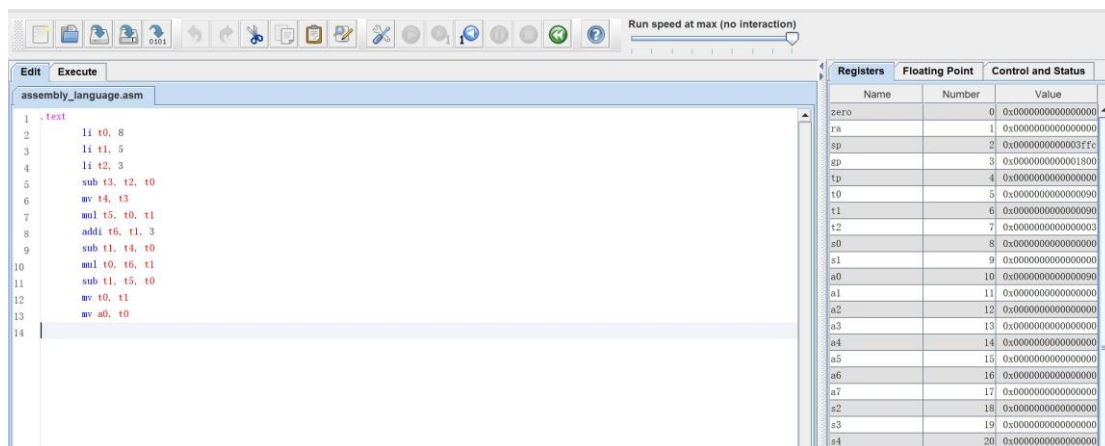
输出 5: 生成的中间代码 intermediate\_code.txt (与实验三一致)

输出 6: 中间代码模拟结果 ir\_emulate\_result.txt (与实验三一致)

输出 7: 生成的汇编代码 assembly\_language.asm

```
assembly_language.asm x
1      .text
2      li t4, 8      # (MOV, a, 8)
3      li t5, 5      # (MOV, b, 5)
4      li t6, 3      # (MOV, $6, 3)
5      sub t0, t6, t4  # (SUB, $0, $6, a)
6      mv t1, t0      # (MOV, c, $0)
7      mul t2, t4, t5  # (MUL, $1, a, b)
8      addi t3, t5, 3  # (ADD, $2, b, 3)
9      sub t6, t1, t4  # (SUB, $3, c, a)
10     mul t0, t3, t6  # (MUL, $4, $2, $3)
11     sub t5, t2, t0  # (SUB, $5, $1, $4)
12     mv t4, t5      # (MOV, result, $5)
13     mv a0, t4      # (RET, , result)
14
15
```

将生成的汇编指令在 RARS 上运行的结果：



分析：

实验四没有新增输入文件。新增的输出文件就是生成的汇编代码。将该汇编代码在 RARS 上运行后可以发现，寄存器 a0 的返回值为 0x90(对应十进制数 144)。由前面的分析可知，源程序段执行后的返回值也确实为 144，可见正确实现了汇编代码生成的功能。

## 使用 check-result.py 检查所有实验的输出

```
C:\Windows\System32\cmd.exe
Microsoft Windows [版本 10.0.22000.1219]
(c) Microsoft Corporation. 保留所有权利。

codes\scripts>python -u check-result.py 4 ../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.

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.

Program terminated by dropping off the bottom.
a0      144
```

观察可知，实验 1 至实验 3 的输出文件均与 std 目录下的标准文件相一致，实验 4 生成的汇编代码在 RARS 上运行后得到的返回值为 144，同样符合预期。也就是说，所设计出的编译器满足了本学期实验的所有要求。

## 5 实验中遇到的困难与解决办法

### 遇到的困难及解决办法

**困难 1：**实验所提供的框架代码量较大，定义了较多类、方法，刚上手实验时对代码框架不够熟悉，需要花费较多时间阅读代码、理解作者意图，容易不知道从何下手，进展缓慢。

**解决办法：**反复阅读代码框架，结合编译器各个组成部分（词法分析、语法分析、语义分析、汇编生成等）的运作机制，弄清楚各个类、方法的功能作用以及相互之间的调用依赖关系。同时充分利用好 IDEA 的高亮提示、字符串搜索功能，弄明白哪些函数已经被调用过了；哪些函数虽然已经被实现但还没有被调用、可能是由我们学生补充代码时调用的；哪些部分是已经被实现好、不希望被我们修改的；哪些部分是还没有被完善好，要求我们进行补充。在充分阅读并理解前人代码的基础上，我们再开始补充自己的代码，能大大提高实验效率。

**困难 2：**刚上手实验时对理论课所学知识不够熟悉（存在部分内容遗忘），没有能够很迅速地把理论课所学知识与实验课的实践要求结合起来。

**解决办法：**反复阅读实验指导书、实验课 PPT 以及理论课 PPT，重新复习巩固理论课所讲知识，重温编译器各个关键组成部分的工作流程，自顶向下地把握所给



框架的运行机理。比如，对于实验二 LR(1) 分析表的分析过程，我们可以结合理论课 PPT 上的例子，先在草稿纸上用笔推算一遍，完成移进、归约、接收等操作，并画出每一轮结束后符号栈、状态栈的变化，体验一次完整的语法分析流程。完成这一工作后，我们对于语法分析这个步骤到底要干什么、有什么细节之处需要注意就有了更清晰的认识，剩下要做的就是看懂框架代码所提供的 API 接口，用好前人提供的“轮子”，将刚刚在纸上推演的过程重新用代码实现，完成编码。

**困难 3：**对实验的部分细节之处把握不到位，存在部分代码漏洞导致程序无法跑通，或运行后无法出现预期现象。调试工作（寻找代码错误）耗费较多时间。

**解决办法：**代码调试是开发中必不可少的环节，遇到程序错误我们只能静下心来仔细分析，反思代码。本学期的 4 次实验我主要使用控制台打印、断点调试的方法来进行 debug，并发现自己的程序会出错确实是因为对部分细节的把握不到位。例如，在实验四中，我就遭遇了 RuntimeException，它提示当前变量还没有被分配寄存器（即无法找到当前变量占用的寄存器）。在反复分析后，我发现是自己编写的查询条件“if (registerAllocTable.get(key) == irVariable)”有误，如图 3 所示。这个查询条件要求查询的 IRVariable 和寄存器分配表中的 IRVariable 必须是同一个对象，否则都会报错。但是，同一个变量可能在中间代码中反复多次出现，程序有可能会创建一个新的对象来保存该变量，上述判断条件实在是过于严苛。因此应该改为判断变量的名字是否相同，而不是去判断是否为同一个对象。如图 4 所示，将查询条件修改为判断变量名是否相同，能成功解决报错问题并顺利通过测试。

```
/**
 * 用于根据 IRVariable 从 registerAllocTable 查找出分配的寄存器
 */
private Register getRegisterWithIRVariable(IRVariable irVariable) {
    for (var key : registerAllocTable.keySet()) {
        System.out.println(registerAllocTable.get(key));
        if (registerAllocTable.get(key) == irVariable) {
            return key;
        }
    }
    // 若没找到，说明当前变量还没有被分配寄存器，报错
    throw new RuntimeException("The variable hasn't been allocated register yet!");
}

/**
```

图 3 修改前的代码（查询条件为要求是同一个对象）

```
/**
 * 用于根据 IRVariable 从 registerAllocTable 查找出分配的寄存器
 */
private Register getRegisterWithIRVariable(IRVariable irVariable) {
    for (var key : registerAllocTable.keySet()) {
        if (Objects.equals(registerAllocTable.get(key).getName(), irVariable.getName())) {
            return key;
        }
    }
    // 若没找到, 说明当前变量还没有被分配寄存器, 报错
    throw new RuntimeException("The variable hasn't been allocated register yet!");
}
```

图 4 修改后的代码（查询条件为要求变量名相同）

## 收获与建议

本学期的《编译原理》课程实验，让我体验了一次编译器开发的完整流程，既巩固了理论课所学的有关词法分析、语法分析、语义分析与中间代码生成、汇编生成等部分的内容，也对编译器的工作机制有了更深入的理解，同时进一步熟悉 Java 语言和 IDEA 开发工具，深刻锻炼了编程调试能力、分析设计能力、文档代码阅读能力，为后续课程学习乃至日后科研、工作奠定坚实的基础。

实验所提供的代码框架比较详细，配套的注释说明也非常到位，对于大部分此前没有接触过编译器的同学来说是非常好的入门学习材料。通过使用 Git 进行追踪，可以发现每次实验我改动的代码量大约在 200 行左右，最多的一次（实验四）也只有约 300 多行，总共改动代码量在 1000 行以内。也就是说，四次实验需要我们写的代码并不算很多，更关键之处在于充分阅读并理解前人写的代码，使用好已经实现的类、方法、接口，避免重复造轮子。我们在日后工作中，也往往很少需要从零开始编写代码，而是在已有代码的基础上进行补充完善。所以，阅读他人代码以及注释、文档的能力就显得尤为重要。完成本学期的实验则有效锻炼了这一能力。

代码框架中有很多值得学习的编程技巧。我在为汇编指令的生成设计数据结构时，就参考了模板代码为 IR 指令所设计的数据结构。又例如，在实验二中，NonTerminal 和 Token 没有公共父类，但符号栈却同时需要存放这两种元素，因此可以定义一个类似共用体的 Symbol 类来包装 Token 和 NonTerminal，从而达到把 NonTerminal 和 Token 同时存入符号栈中的目的。这种处理方式非常巧妙，我在实验三、实验四中都仿照这种写法进行设计，自定义的 SemanticStackEntry、

IRGeneratorStackEntry、Operand 等类都是将两个原本没有公共父类的元素进行包装，从而达到统一处理的目的。

完成实验的过程也促进了我理论课的学习。在进行实验之前，我对于理论课介绍的词法分析、语法分析、语义分析、汇编生成等过程只有一个抽象模糊的认识。但完成实验后，我在草稿纸上推演并编程实现了自动机识别词法单元、根据 LR 分析表进行移进归约等操作的过程，对它们有了更透彻的认识。我还自行设计翻译方案，充分体会到 S-属性定义的自底向上翻译模式的便捷高效性。汇编生成的实验我们只要求完成不完备的寄存器分配，这也让我明白为什么理论课上讲到，选择最优的寄存器指派方案是一个 NP 完全问题。

在实验中，我免不了会遭遇代码无法跑通、程序报错、程序运行无法出现预期现象的情况，查找 bug 的过程既是一场对人心性与定力的考验，也是一次难能可贵的历练机会。通过控制台打印、断点调试等方式，我最终顺利地解决了遇到的各类问题，更获得了能力上的提升。实验结束后实验报告的撰写，既有利于我养成严谨认真的科学态度，同时也便于我日后反思、回顾本学期的编译原理实验。撰写报告虽然耗时较长，但却在无形之中加深了我对每次实验任务的理解，让我重新审视自己的解决方案到底是什么、编译器各个组成部分到底是如何分工协作的、核心组件的数据结构应该怎么设计等等，令人受益匪浅。

至于建议，我认为可以适当增加编译器的功能，使其能够翻译更复杂的程序。我们最终设计出的编译器，只能翻译声明语句、简单赋值语句、算术逻辑语句。理论课上传授的有关常见控制结构翻译、回填技术的知识，缺乏相应的实验内容加以巩固。如果能让我们编程实现更多类型语句的翻译（并不超纲，理论课也讲了），可能会让同学们收获更多技能，也会更有成就感。此外，实验二的 LR 分析表可以使用第三方工具生成，我们在做实验时不需要关心它的具体生成细节。但事实上，如何生成这张 LR 分析表同样是理论课讲授的重点之一，所以如果能让我们编程构建这张 LR 分析表，也可以让同学们对底层技术有更深入的了解。

总的来说，本学期《编译原理》课程实验还是让我收获较大。