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: 胡冰

学　号: 200111412

专　业: 计算机科学与技术

日　期: 2022-11-7

1 实验目的与方法

总体实验目的：利用 Java 实现 TXT 语言编译器，设计并实现编译器的词法分析器、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。该编译器支持将 TXT 语言（实验中的类 C 语言）转化为在支持 RV32M 指令集的 RISC-V 32 机器上可以成功运行的汇编代码。

总体实验方法：使用的语言为 Java，使用的软件环境为 Java17、RARS、编译工作台。

1.1 词法分析器

1） 实验目的：

使用自动机的写法实现一个简单、特定的词法分析器：根据从文件中读取源代码的代码文本，生成词法单元迭代器并正确地将源语言中的每个标识符插入到符号表中（不要求记录每个词法单元的行号、起始列号、结束列号，不要求处理注释）

- 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；
- 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；
- 巩固课堂学习的词法分析方法.

2） 实验方法：

- 使用的语言：Java
- 使用的软件环境：Java17

1.2 语法分析

1） 实验目的：

- 深入了解语法分析程序实现原理及方法；
- 理解 LR(1)分析法时严格的从左向右扫描和自底向上的语法分析方法；
- 巩固课程学习的自底向上语法分析方法。

2） 实验方法：

- 使用的语言：Java
- 使用的软件环境：Java17

1.3 典型语句的语义分析及中间代码生成

1） 实验目的：

- 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。
- 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；
- 加深对课堂学习的语义分析方法和中间代码生成方法的认识。

2） 实验方法：

- 使用的语言：Java

- 使用的软件环境：Java17

1.4 目标代码生成

1) 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) 实验方法：
   - 使用的语言：Java
   - 使用的软件环境：Java17、RARS、编译工作台

2 实验内容及要求

2.1 词法分析器

1) 实验内容：

   编写一个词法分析程序，读取文件，对文件内自定义的类 c 语言程序段进行词法分析。
2) 实验要求：
   - 输入：以文件形式存放自定义的类 C 语言程序段；
   - 输出：以文件形式存放的 TOKEN 串和简单符号表；
   - 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。输出的两个文件通过脚本检测文件。

2.2 语法分析

1) 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（实验一的输出）进行语法分析
2) 实验要求：
   - 输出推导过程中所用产生式序列并保存在输出文件中。

2.3 典型语句的语义分析及中间代码生成

1) 实验内容：
   - 采用实验二中的文法，为语法正确的单词串设计翻译方案，完成语法制导翻译；
   - 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；
   - 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；
   - 使用框架中的模拟器 IREmulator 验证生成的中间代码的正确性。
2) 实验要求：

   在前两个实验的基础上，实现语义分析和中间代码生成，并使用框架中的模拟器 IREmulator 验证生成的中间代码的正确性。

2.4 目标代码生成

1） 实验内容：

- 将实验三生成的中间代码转换为目标代码（RISC-V 汇编指令）；

2） 实验要求

- 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

编译器为了方便对源程序进行编译处理，需要按照一定的方式对单词进行分类和编码，所以需要定义一个编码表。

```
1 int
2 return
3 =
4 ,
5 Semicolon
6 +
7 -
8 *
9 /
10 (
11 )
12 ==
13 ++
14 --
15 **
51 id
52 IntConst
```

图表 1 定义编码表

3.1.2. 正则文法

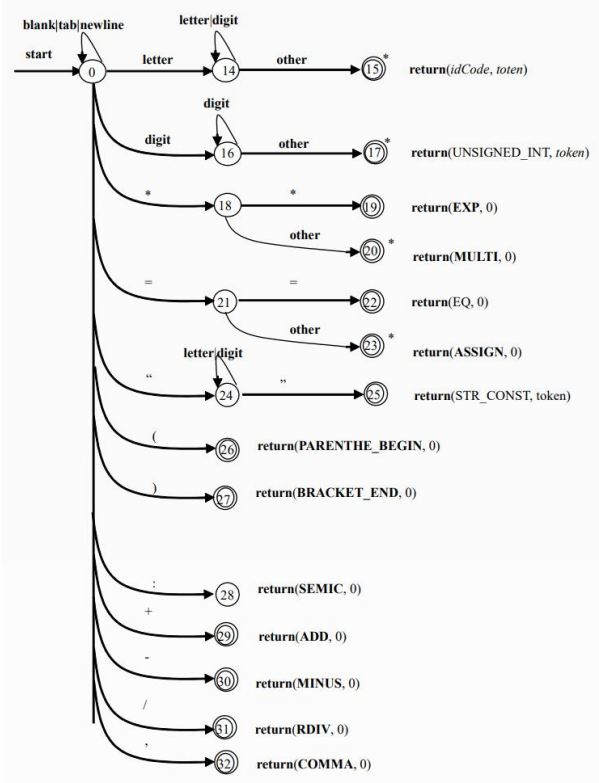
词法分析器需要通过正则文法来识别单词，因此需要定义正则文法。

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

图表 2 正则文法

3.1.3. 状态转换图

状态转换图时正则文法到编写程序的桥梁。有穷自动机和正则文法等价，考虑到状态转换图的直观性，因此可以从状态转换图出发来考虑词法分析器的设计。



图表 3 总体状态转换图

3.1.4. 主要函数流程

```
private final SymbolTable symbolTable;    //符号表

private List<Token> Tokens = new ArrayList<>();    //token序列

private String input_code ;    //存放文件内容
```

图表 4 LexicalAnalyzer 主要成员

```
public void loadFile(String path) {
    // TODO: 词法分析前的缓冲区实现
    // 可自由实现各类缓冲区
    // 或直接采用完整读入方法
    this.input_code = FileUtils.readFile(path);
}
```

图表 5 从给予的路径中读取并加载文件内容（源程序代码）

实现 run()函数，执行语法分析，准备好返回的 token 列表：先利用循环处理空格、换行符、制表符，再使用 switch 以状态机方式实现词法分析器（注意在处理标识符、常数等成分时需要超前搜索）。

```
while( ch == ' ' || ch == '\n' || ch == '\t' ){  //跳过空格、换行符、制表符
    if( i < len - 1 ){
        ch = this.input_code.charAt( ++i );
    }else{
        break;
    }
}
```

图表 6 处理空格、换行符、制表符

```
if(Character.isDigit(ch)){    //处理常数
    String temp = "";
    temp = temp + ch ;
    if( i == len -1 ) {
        break;
    }
    ch = this.input_code.charAt( ++i );
    while(Character.isDigit(ch)){
        temp += ch ;
        ch = this.input_code.charAt( ++i );
    }
    i -- ; //回退一个
    Token t = Token.normal( tokenKindId: "IntConst",temp);
    this.Tokens.add(t);
}
```

图表 7 处理常数（使用超前搜索）

```
else if(Character.isAlphabetic(ch)){    //处理字符串
    String temp = "";
    temp = temp + ch ;
    Token t;
    if( i == len -1 ) {
        break;
    }
    ch = this.input_code.charAt( ++i );
    while(Character.isAlphabetic(ch)||Character.isDigit(ch)){
        temp += ch ;
        ch = this.input_code.charAt( ++i );
    }
    i -- ; //回退一个
    if("return".equals(temp)){
        t = Token.simple("return");
        this.Tokens.add(t);
    }else if("int".equals(temp)){
        t = Token.simple("int");
        this.Tokens.add(t);
    }
    else{
        t = Token.normal( tokenKindId: "id",temp);
        this.Tokens.add(t);
        if(!symbolTable.has(temp)){
            symbolTable.add(temp);
        }
    }
}
```

图表 8 处理 return、int 和标识符

```
Token t;
switch(ch){
    case '=':
        ch = this.input_code.charAt( ++i );
        if(ch == '='){
            t = Token.simple("==");
            this.Tokens.add(t);
        }else {
            i--;
            t = Token.simple("=");
            this.Tokens.add(t);
        }
        break;
    case ',':
        t = Token.simple(",");
        this.Tokens.add(t);
        break;
    case ';':
        t = Token.simple("Semicolon");
        this.Tokens.add(t);
        break;
```

图表 9 处理个别特殊操作符

```
public Iterable<Token> getTokens() {
    // 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 this.Tokens;

}
```

图表 10 主函数中获取得到的 token 序列

3.2. 语法分析

3.2.1. 拓展文法

为了让构造的 LR1 语法分析方法的 DFA 只有一个初态和终态，所以需要将原文法编写为一个拓广文法（由于框架中已经提供）。

```
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;
```

图表 11 拓广文法

3.2.2. LR1 分析表

根据拓广文法结合 LR1 分析表的构造方法，可以构造 LR1 分析表，从而用于进行语法分析。LR 分析表是一张类似自动机的状态转移图,当在当前状态遇到某某符号时便执行动作并转移。规约时就根据"Action 表"里写的产生式生成非终结符,然后将生成的非终结符作为"输入符号"根据"Goto 表"来确定转移;移入时就直接根据"Action 表"转移到对应状态

	ACTION									GOTO						
	id	(	)	+	-	*	=	int	return	IntConst	Semicolon	$	E	S_list	S	A	B	D
0	shift 4							shift 5	shift 6						1	2		3
1												accept						
2											shift 7							
3	shift 8																	
4							shift 9											
5	reduce D -> int																	
6	shift 13	shift 14								shift 15			10				11	12
7	shift 4							shift 5	shift 6			reduce S_list -> S Sen		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													

图表 12 LR1 分析表（部分）

```java
public class LRTable {
    /**
     * 根据当前状态与当前词法单元获取对应动作
     *
     * @param status 当前状态
     * @param token  当前词法单元
     * @return 应采取的动作
     */
    public Action getAction(Status status, Token token) {...}

    /**
     * 根据当前状态与规约到非终结符获得应转移到的状态
     *
     * @param status      当前状态
     * @param nonTerminal 规约出的非终结符
     * @return 应转移到的状态
     */
    public Status getGoto(Status status, NonTerminal nonTerminal) { return status.getGoto(nonTerminal); }

    /**
     * @return 起始状态
     */
    public Status getInit() { return statusInIndexOrder.get(0); }

    public void dumpTable(String path) {...}

    private String convertToGotoString(Status status) {...}

    LRTable(List<Status> statusInIndexOrder, List<TokenKind> terminals, List<NonTerminal> nonTerminals) {...}

    private final List<Status> statusInIndexOrder;
    private final List<TokenKind> terminals;
    private final List<NonTerminal> nonTerminals;
}
```

图表 13 LRtable 代码实现框架

3.2.3. 状态栈和符号栈的数据结构

```java
//状态栈
private Stack<Status> statusStack = new Stack<>();
//符号栈
private Stack<Symbol> symbolStack = new Stack<>();
```

图表 14 状态栈与符号栈的定义

- 状态栈的数据结构：

  使用栈存储状态

```java
public record Status(int index, Map<TokenKind, Action> action, Map<NonTerminal, Status> goto_) {
    /**
     * 构造一个状态
     *
     * @param index 状态的索引/编号
     * @return 构造出的状态
     */
    public static Status create(int index) {...}

    /**
     * @return 获得代表错误的状态
     */
    public static Status error() { return errorInstance; }

    public boolean isError() { return this == errorInstance; }

    @Override
    public String toString() { return Integer.toString(index); }

    @Override
    public boolean equals(Object obj) {...}

    @Override
    public int hashCode() { return index; }
```

图表 15 状态类部分实现代码截图

一个状态 Status 由 action 和 goto 组成：action 表示在当前状态下遇到某个终结符需要采取什么动作，而 goto 表示在当前状态下遇到（规约到）某个非终结符需要转移到什么状态。

参数介绍：

index：状态在 LR 表中的索引/编号

action：在该状态下遇到终结符后应该转移到哪个状态

goto_：在该状态下规约到非终结符后应该转移到哪个状态

```java
public Action getAction(TokenKind terminal) { return action.getOrDefault(terminal, Action.error()); }

/**
 * 当遇到该词法单元时，应该转移到哪个状态
 *
 * @param token 词法单元
 * @return 应该转移到的状态
 */
public Action getAction(Token token) { return getAction(token.getKind()); }

/**
 * 当规约到该非终结符时，应该转移到哪个状态
 *
 * @param nonTerminal 非终结符
 * @return 应该转移到的状态
 */
public Status getGoto(NonTerminal nonTerminal) { return goto_.getOrDefault(nonTerminal, Status.error()); }
```

图表 16 利用这些函数可以获取相应的信息

- 符号栈的数据结构

  使用栈存储符号

```java
class Symbol{
    Token token;
    NonTerminal nonTerminal;  //非终结符
//    SourceCodeType type;
//    IRValue vary;

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

    private Symbol(Token token, NonTerminal nonTerminal,SourceCodeType type){...}

    public Symbol(Token token){...}

    public Symbol(NonTerminal nonTerminal){...}

    public Symbol(SourceCodeType TYPE){...}

    public Symbol(IRValue vary){...}

    public Symbol(NonTerminal t,IRValue vary){...}

    public Token getToken() { return token; }

    public NonTerminal getNonTerminal() { return nonTerminal; }

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

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

图表 16 符号类实现代码截图

在处理符号栈时，我们希望将 Token 和 NonTerminal 同时装在栈中. 但 Token 和 NonTerminal 并没有共同祖先类(除了 Object)，因此可以简单定义一个 Symbol 来实现 Union<Token, NonTerminal>的功能，从而方便我们进行语法分析。

3.2.4. LR 驱动程序流程描述

1) 加载词法单元

```java
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
    for(Token token : tokens){
        tokenQueue.add(token);
    }
}
```

图表 17 加载来自主程序的词法单元

2) 加载 LR 分析表

```java
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 initStatus 存起来使用
    lrTable = table;
}
```

图表 18 加载来自主程序的 LRtable

3) 驱动程序实现

```
this.statusStack.push(lrTable.getInit());   //初始化
this.symbolStack.push(new Symbol(Token.eof()));   //符号栈中压入eof
while (!tokenQueue.isEmpty()){  //队列不为空
    Token tokenTemp = tokenQueue.peek();
    Status statusTemp = statusStack.peek();
    Action actionTemp = lrTable.getAction(statusTemp,tokenTemp);
    switch (actionTemp.getKind()){
        case Shift -> {
            Status shiftTo = actionTemp.getStatus();
            this.statusStack.push(shiftTo);
            this.symbolStack.push(new Symbol(tokenTemp));
            this.tokenQueue.remove();
            this.callWhenInShift(this.statusStack.peek(),tokenTemp);
            break;
        }
        case Reduce -> {
            Production production = actionTemp.getProduction();
            for(Term body : production.body()){
                this.statusStack.pop();
                this.symbolStack.pop();
            }
            this.statusStack.push(lrTable.getGoto(statusStack.peek(),production.head()));
            this.symbolStack.push(new Symbol(production.head()));
            this.callWhenInReduce(this.statusStack.peek(),production);
            break;
        }
        case Accept -> {
            this.callWhenInAccept(statusTemp);
            this.tokenQueue.remove();
            break;
        }
    }
```

图表 19 驱动程序代码实现框架

驱动程序实现：首先，建立符号栈与状态栈，并初始化栈。然后根据状态栈栈顶的元素和待读入的下一个 token 查询判断下一个待执行动作：如果是 shift，把 action 的状态压入状态栈，对应的 token 压入符号栈，通知对应的观察者；如果是 reduce，根据产生式的长度，符号栈和状态栈都弹出相应长度个 token 和状态，然后将产生式左边的符号压入符号栈中,根据当前符号栈栈顶的元素和状态栈栈顶的元素获取 goto 表的信息，将得到的状态压入状态栈，并且通知对应的观察者；如果是 accept，则语法分析结束，并通知相应的观察者。与此同时，ProductionCollector 观察者内部顺序记录归约所用的产生式，语法分析结束时会将其输出到文件中。

3.3. 语义分析和中间代码生成

3.3.1. 翻译方案

采用 S-属性定义的自底向上翻译方案。具体方案如下：

注：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

enter:将变量的类型填入符号表

- P -> S_list {P.value = S_list.value}

- S_list -> S Semicolon S_list {S_list.value = S_list1.value}
- S_list -> S Semicolon;{ S_list.value = S.value }
- 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 -> E + A {E.value = E1.value+A.value;}
- E -> E - A {E.value = E1.value-A.value;}
- A -> A * B{A.value = A1.value*B.value;}
- B -> (E){B.val = E.val;}
- B -> id {}

3.3.2. 语义分析和中间代码生成使用的数据结构

1) 语义分析使用的数据结构

在实验二的基础上拓广符号栈数据结构（增加数据类型 SourceCodeType）作为语义分析栈的数据结构。

2) 中间代码生成使用的数据结构

在语义栈的基础上，添加 IRValue 数据类型

```
class Symbol{
    Token token;
    NonTerminal nonTerminal;  //非终结符
    SourceCodeType type;
    IRValue vary;

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

    private Symbol(Token token, NonTerminal nonTerminal,SourceCodeType type){...}

    public Symbol(Token token){...}

    public Symbol(NonTerminal nonTerminal){...}

    public Symbol(SourceCodeType TYPE){...}

    public Symbol(IRValue vary){...}

    public Symbol(NonTerminal t,IRValue vary){...}

    public Token getToken() { return token; }

    public NonTerminal getNonTerminal() { return nonTerminal; }

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

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

图表 20 修改后的符号类

3.3.3. 主要流程描述（两个观察者的实现）

1） 语义分析观察者实现

- 语义分析的结果是更新符号表，根据翻译方案语义分析栈需要保存 type 属性；
- 自底向上的分析过程中 Shift 时将符号的 type 属性(从 Token 中获得）加入语义分析栈；
- 当 D->int 这条产生式发生归约时，int 这个 token 应该在语义分析栈中，把这个 token 的 tpye 类型赋值给 D 的 type；
- 当 S->D id 这条产生式归约时，取出 D 的 type，这个 type 是 id 的 type，更新符号表中相应变量的 type 信息，压入空记录占位；
- 对于其他产生式发生规约时，直接压入空记录占位即可.

```
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 case
    switch (production.index()){
        case 4 ->{   //S -> D id;
            String name = this.symbolStack.peek().getToken().getText();
            this.symbolStack.pop();
            SymbolTableEntry temp = symbolTable.get(name);
            SourceCodeType type = this.symbolStack.peek().type;
            symbolTable.get(name).setType(type);
            this.symbolStack.pop();
            symbolStack.push(new Symbol(production.head()));
            break;
        }
        case 5 ->{   //D -> int;
            symbolStack.pop();
            symbolStack.push(new Symbol(SourceCodeType.Int));
            break;
        }
        default -> {  //占位即可
            for(Term body:production.body()){
                symbolStack.pop();
            }
            symbolStack.push(new Symbol(production.head()));
            break;
        }
    }
}
```

图表 21 语义分析实现代码

2） 中间代码生成观察者实现

根据翻译方案，利用规约到的产生式编写不同的语义动作，在语义翻译的过程中生成中间代码。

```
case 1->{
    Symbol temp = symbolStack.peek();  //获取栈顶元素
    symbolStack.pop();
    symbolStack.push(new Symbol(production.head(),temp.vary));
    break;
}
```

图表 22 对于产生式 P-> S_list 根据翻译方案编写语义动作，将 S_list 的 value 赋值给 P 的 value

```
case 2,3,4,5->{
    for(Term body:production.body()){
        symbolStack.pop();
    }
    symbolStack.push(new Symbol(production.head(),IRVariable.named("$")));
    break;
}
```

图表 23 对 index 分别为 2，3，4，5 的产生式进行处理，在语义栈中压入符号占位

```
case 6->{     //E-> id=E
    Symbol from = this.symbolStack.peek();
    this.symbolStack.pop();   // E
    this.symbolStack.pop();   // =
    Symbol temp = this.symbolStack.peek();
    this.symbolStack.pop();   //id
    IRVariable result = (IRVariable)temp.vary;
    Instruction instruction = Instruction.createMov(result,from.vary);  //MOV中间指令生成
    symbolStack.push(new Symbol(production.head(),IRVariable.named("$")));
    IR_list.add(instruction);   //加入中间指令集合
    break;
}
```

图表 24 对产生式 E-> id=E 处理，产生 MOV 类型中间指令，并将其并加入中间指令集

```
case 7->{
    Symbol temp = symbolStack.peek();
    symbolStack.pop();  //E
    symbolStack.pop();  //return
    Instruction instruction = Instruction.createRet(temp.vary); //产生RET中间指令
    IR_list.add(instruction);     //加入中间指令集合
    symbolStack.push(new Symbol(production.head(),IRVariable.named("$")));
    break;
}
```

图表 25 对产生式 S-> return E 处理，生成 RET 类型中间指令，并将其加入中间指令集

```
case 8->{   //E -> E + A;
    Symbol rhs = symbolStack.peek();
    symbolStack.pop();
    symbolStack.pop();
    Symbol lhs = symbolStack.peek();
    symbolStack.pop();
    IRVariable temp = IRVariable.temp();
    Instruction instruction = Instruction.createAdd(temp,lhs.vary,rhs.vary);  //产生ADD中间指令
    IR_list.add(instruction);   //加入指令集合
    symbolStack.push(new Symbol(production.head(),temp));
    break;
}
```

图表 26 对产生式 E-> E + A 处理，生成 ADD 类型中间代码，并将其加入中间代码集

```
case 9->{  //E -> E - A;
    Symbol rhs = symbolStack.peek();
    symbolStack.pop();
    symbolStack.pop();
    Symbol lhs = symbolStack.peek();
    symbolStack.pop();
    IRVariable temp = IRVariable.temp();
    Instruction instruction = Instruction.createSub(temp,lhs.vary,rhs.vary);  //产生SUB中间代码
    IR_list.add(instruction);  //加入指令集合
    symbolStack.push(new Symbol(production.head(),temp));
    break;
}
```

图表 27 对产生式 E->E-A 处理，生成 SUB 类型中间代码，并将其加入中间代码集

```
case 10,12,14->{
    Symbol temp = symbolStack.peek();
    symbolStack.pop();
    symbolStack.push(new Symbol(production.head(),temp.vary));
    break;
}
```

图表 28 对 index 为 10、12、14 的产生式进行处理，在语义栈中加入符号占位

```
case 11->{  //A -> A * B;
    Symbol rhs = symbolStack.peek();
    symbolStack.pop();
    symbolStack.pop();
    Symbol lhs = symbolStack.peek();
    symbolStack.pop();
    IRVariable temp = IRVariable.temp();
    Instruction instruction = Instruction.createMul(temp,lhs.vary,rhs.vary);
    IR_list.add(instruction);
    symbolStack.push(new Symbol(production.head(),temp));
    break;
}
```

图表 29 对产生式 A->A*B 进行处理，生成 MUL 类型中间指令，将其加入中间指令集合

```
case 13->{  //B -> ( E );
    symbolStack.pop();
    Symbol symbol = symbolStack.peek();
    symbolStack.pop();
    symbolStack.pop();
    symbolStack.push(new Symbol(production.head(),symbol.vary));
    break;
}
```

图表 30 对产生式 B->(E)处理，并在语义栈中压入符号占位

```
case 15->{   //B -> IntConst;
    Symbol temp = symbolStack.peek();
    symbolStack.pop();
    Symbol head = new Symbol(production.head(),temp.vary);
    symbolStack.push(head);
    break;
}
```

图表 31 对产生式 B->IntConst 处理，并在语义栈中压入符号占位

3.4. 目标代码生成

3.4.1. 主要流程描述

1） 在加载中间代码时对中间代码进行预处理

对于具有两个操作数的指令（如：ADD,SUB,MUL）：

- 如果指令的两个操作数都为立即数类型，则将这两个立即数直接按照对应的运算操作求值得到结果，然后替换成 MOV 指令。

```
if(LHSisImmediate && RHSisImmediate){   //两个操作数都是立即数 转化为MOV指令
    IRImmediate LHSimmediate = (IRImmediate) LHS;
    IRImmediate RHSimmediate = (IRImmediate) RHS;
    switch (kind){
        case ADD -> {
            int temp = LHSimmediate.getValue() + RHSimmediate.getValue();
            IRImmediate tempImmediate = IRImmediate.of(temp);
            IRVariable tempIRvariable = IRVariable.temp();
            Instruction tempInstruction = Instruction.createMov(tempIRvariable,tempImmediate);
            this.processedInstruction.add(tempInstruction);
            break;
        }
        case SUB -> {
            int temp = LHSimmediate.getValue() - RHSimmediate.getValue();
            IRImmediate tempImmediate = IRImmediate.of(temp);
            IRVariable tempIRvariable = IRVariable.temp();
            Instruction tempInstruction = Instruction.createMov(tempIRvariable,tempImmediate);
            this.processedInstruction.add(tempInstruction);
            break;
        }
        case MUL -> {
            int temp = LHSimmediate.getValue() * RHSimmediate.getValue();
            IRImmediate tempImmediate = IRImmediate.of(temp);
            IRVariable tempIRvariable = IRVariable.temp();
            Instruction tempInstruction = Instruction.createMov(tempIRvariable,tempImmediate);
            this.processedInstruction.add(tempInstruction);
            break;
        }
        default -> {
            break;
        }
    }
```

图表 32 两个操作数都为立即数，将其转化为 MOV 指令

- 如果指令只有一个操作数为立即数类型（除了左立即数减法），则将该指令进行调整，使之满足“a:=b op imm”的格式。

```
else if(LHSisImmediate && !RHSisImmediate){  //左操作数是立即数
    switch (kind){
        case ADD -> {
            IRVariable tempResult = instruction.getResult();
            Instruction tempInstruction = Instruction.createAdd(tempResult,RHS,LHS);
            this.processedInstruction.add(tempInstruction);
            break;
        }
        case MUL -> {
            IRVariable tempResult = instruction.getResult();
            Instruction tempInstruction = Instruction.createMul(tempResult,RHS,LHS);
            this.processedInstruction.add(tempInstruction);
            break;
        }
```

图表 33 左操作数为立即数，将其转化为 a=b op imm 模式

- 如果指令为左立即数减法，则前插一条“MOV a，imm”，用 a 替换原立即数，将指令调整为无立即数指令。

```
case SUB -> {
    IRVariable tempVariable = IRVariable.temp();
    Instruction tempMovInstruction = Instruction.createMov(tempVariable,LHS);
    this.processedInstruction.add(tempMovInstruction);
    IRVariable tempResult = instruction.getResult();
    Instruction tempInstruction = Instruction.createSub(tempResult,tempVariable,RHS);
    this.processedInstruction.add(tempInstruction);
    break;
}
```

图表 34 左立即数减法指令处理

2） 维护双射映射

利用双射映射可以方便地实现地址描述符和寄存器描述符。

```
public class BMap<K, V> {
    private final Map<K, V> KVmap = new HashMap<>();
    private final Map<V, K> VKmap = new HashMap<>();

    public void removeByKey(K key) { VKmap.remove(KVmap.remove(key)); }

    public void removeByValue(V value) {
        KVmap.remove(VKmap.remove(value));

    }

    public boolean containsKey(K key) { return KVmap.containsKey(key); }

    public boolean containsValue(V value) { return VKmap.containsKey(value); }

    public void replace(K key, V value) {
        // 对于双射关系，将会删除交叉项
        removeByKey(key);
        removeByValue(value);
        KVmap.put(key, value);
        VKmap.put(value, key);
    }

    public V getByKey(K key) { return KVmap.get(key); }

    public K getByValue(V value) {
        return VKmap.get(value);
    }
}
```

图表 35 双射映射实现

3） 实现寄存器分配算法

采用了不完备的寄存器分配，分配策略为：当要对一个变量做寄存器指派时，有寄存器空闲，则指派任意空闲寄存器分配；当无寄存器空闲时，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.

```java
public String allocateRegister(IRValue value){
    String register = "";
    if(this.bMap.containsValue(value)){  //
        register = this.bMap.getByValue(value);
        int tempNum = this.counterMap.get(value);
        this.counterMap.remove(value);
        this.counterMap.put(value,tempNum-1);   //使用过一次后减一
    }else{
        if(this.registerAlloccationList.isEmpty()){
            register = reallocateRegister();
            this.bMap.replace(register,value);
            int tempNum = this.counterMap.get(value);  //使用过一次后减一
            this.counterMap.remove(value);
            this.counterMap.put(value,tempNum-1);
        }
        else {
            register = this.registerAlloccationList.get(0);
            this.registerAlloccationList.remove( index: 0);
            this.bMap.replace(register,value);
            int tempNum = this.counterMap.get(value);  //使用过一次后减一
            this.counterMap.remove(value);
            this.counterMap.put(value,tempNum-1);
        }
    }
    return register;
}
```

图表 36 分配算法

对于查找是否存在有寄存器被不再使用的变量占用，采用如下思路：首先，在生成目标代码之前对中间代码中所要用到的所有变量的个数进行计数(利用 hash 表实现由变量名到该变量名被使用次数的计数)，当使用某一变量一次时，就对对应的使用次数减一；当发生寄存器申请且空闲寄存器不足时，则在 hash 表中查找对应 value 值为 0 的变量名（表示该变量在后续程序中将不再被使用），然后利用维护的双射映射找到对应该变量的寄存器，将这个寄存器分配给发出申请的变量。

```java
public String reallocateRegister(){
    String register = "";
    IRValue tempValue = null;
    for(IRValue value:this.counterMap.keySet()){
        if(this.counterMap.get(value)==0) {
            tempValue = value;
            if(this.bMap.getByValue(tempValue) != null ){
                register = this.bMap.getByValue(tempValue);
                break;
            }
        }
    }
    return register;
}
```

图表 37 再分配算法

4） 目标代码生成

根据每条 IR 的类型，参数类型确定对应的代码生成。

```
// 生成指令
for(Instruction instruction:this.processedInstruction){
    InstructionKind kind = instruction.getKind();
    switch (kind){
        case ADD,SUB,MUL -> {
            IRVariable tempResult = instruction.getResult();
            IRValue tempLHS = instruction.getLHS();
            IRValue tempRHS = instruction.getRHS();
            String lhs = "" ;
            String rhs = "" ;
            String result = "";
            String op = "";
            if(tempRHS.isImmediate()){
                rhs = rhs + (((IRImmediate) tempRHS).getValue());
                lhs = allocateRegister(tempLHS);
                result = allocateRegister(tempResult);
                switch (kind){
                    case ADD -> {
                        op = "addi";
                        break;
                    }
                    case MUL-> {
                        op = "muli";
                        break;
                    }
                    default -> {
                        break;
                    }
                }
            }
            else{...}
            String riscvInstruction = "    " + op + " " + result + " " + lhs + " " + rhs;
            this.targetInstuction.add(riscvInstruction);
        }
```

图表 38 实现目标代码生成代码（部分）

5） 利用 RARS 工具对生成的目标代码进行正确性检测

```
1   .text
2       li t0 8
3       li t1 5
4       li t2 3
5       sub t3 t2 t0
6       mv t4 t3
7       mul t5 t0 t1
8       addi t6 t1 3
9       sub t0 t4 t0
10      mul t2 t6 t0
11      sub t2 t5 t2
12      mv t2 t2
13      mv a0 t2
14
```

图表 39 生成的目标代码

Name	Number	Value
zero	0	0x0000000000000000
ra	1	0x0000000000000000
sp	2	0x0000000000003ffc
gp	3	0x0000000000001800
tp	4	0x0000000000000000
t0	5	0xfffffffffffffff3
t1	6	0x0000000000000005
t2	7	0x0000000000000090
s0	8	0x0000000000000000
s1	9	0x0000000000000000
a0	10	0x0000000000000090
a1	11	0x0000000000000000
a2	12	0x0000000000000000
a3	13	0x0000000000000000
a4	14	0x0000000000000000
a5	15	0x0000000000000000
a6	16	0x0000000000000000
a7	17	0x0000000000000000
s2	18	0x0000000000000000
s3	19	0x0000000000000000
s4	20	0x0000000000000000
s5	21	0x0000000000000000
s6	22	0x0000000000000000
s7	23	0x0000000000000000
s8	24	0x0000000000000000
s9	25	0x0000000000000000
s10	26	0x0000000000000000
s11	27	0x0000000000000000
t3	28	0xfffffffffffffffb
t4	29	0xfffffffffffffffb
t5	30	0x0000000000000028
t6	31	0x0000000000000008
pc		0x0000000000000034

图表 40 RARS 输出结果

4 实验结果与分析

4.1 词法分析

输入：码点文件 coding_map.csv 和输入的代码 input_code.txt

```
1 int
2 return
3 =
4 ,
5 Semicolon
6 +
7 -
8 *
9 /
10 (
11 )
12 ==
13 ++
14 --
15 **
51 id
52 IntConst
```

图表 41 码点文件

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

图表 42 输入代码

输出：符号表 old_symbol_table.txt 和词法单元列表 token.txt

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

```

图表 43 符号表

```
(int,)
(id,result)
(Semicolon,)
(int,)
(id,a)
(Semicolon,)
(int,)
(id,b)
(Semicolon,)
```

图表 44 词法单元列表（部分）

结果分析：利用脚本检测文件将输出的 old_symbol_table.txt、token.txt 与标准输出进行比较，成功通过检测，表明词法分析单元可以生成正确的词法单元序列。

4.2 语法分析

输入：码点文件 coding_map.csv、语法文件 grammar.txt、输入代码 input_code.txt、IR 分析表 LR1_table.csv

```
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;
```

图表 45 语法文件

	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 Sen		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												

图表 46 LR(1)分析表

输出：规约过程的产生式列表 parser_list.txt、语义分析前的符号表 old_symbol_table.txt、词法单元列表 token.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
```

图表 47 规约过程的产生式列表（部分）

结果分析：利用脚本检测文件将输出的 parser_list.txt 与标准输出进行比较，成功通过检测，表明语法分析单元可以生成正确的规约过程的产生式列表。

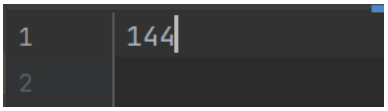
4.3 中间代码生成

输入：码点文件 coding_map.csv、语法文件 grammar.txt、输入代码 input_code.txt、IR 分析表 LR1_table.csv

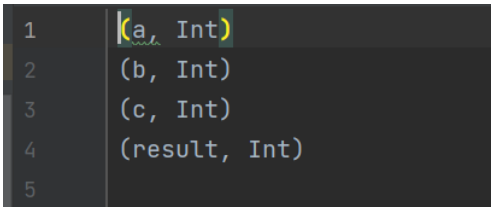
输出：中间表示 intermediate_code.txt、中间表示的模拟执行的结果 ir_emulate_result.txt、规约过程的产生式列表 parser_list.txt、语义分析后的符号表 new_symbol_table.txt、语义分析前的符号表 old_symbol_table.txt、词法单元列表 token.txt

```
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
```

图表 48 中间代码



图表 49 中间表示的模拟执行的结果



图表 50 语义分析后的符号表

结果分析：将中间表示的模拟执行结果与标准输出结果进行对比，两者均为 **144**，同时语义分析后的符号表也与标准文件相同，说明实现的语义分析和中间代码生成单元能够正常地进行工作。

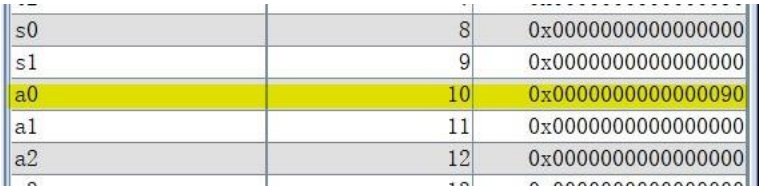
4.4 目标代码生成

输入：码点文件 coding_map.csv、语法文件 grammar.txt、输入代码 input_code.txt、IR 分析表 LR1_table.csv

输出：汇编代码 assembly_language.asm、中间表示 intermediate_code.txt、中间表示的模拟执行的结果 ir_emulate_result.txt、规约过程的产生式列表 parser_list.txt、语义分析后的符号表 new_symbol_table.txt、语义分析前的符号表 old_symbol_table.txt、词法单元列表 token.txt

```
.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 t0 t4 t0
    mul t2 t6 t0
    sub t2 t5 t2
    mv t2 t2
    mv a0 t2
```

图表 51 汇编代码



s0	8	0x0000000000000000
s1	9	0x0000000000000000
a0	10	0x0000000000000090
a1	11	0x0000000000000000
a2	12	0x0000000000000000

图表 52 汇编代码执行结果

结果分析：将生成的目标代码转移至 RARS 上执行，最终目标寄存器 a0 存放的值为 0x90，即 144，为代码正确执行的结果，说明本实验实现的编译器能够正确地生成目标代码

5 实验中遇到的困难与解决办法

5.1 实验中遇到的困难

- 完成实验需要使用大量陌生的框架接口，在实现编译器的过程中需要反复查看。有的时候还会忘记使用已有的 API，导致实现过程变得困难。
- 在将课本内容进行转化并实践时，对课本的内容存在理解上的困难，在实现编译器时走了很多弯路。

5.2 解决方法

- 在实验之前，多遍阅读、熟悉提供的框架接口，熟悉代码框架对于提高后续的编码效率具有很大的帮助。
- 在实验过程中，发现对课本所学知识存在困惑时，及时查看课本，对对应的知识点进行复习，这样可以提高实现编译器的效率。

5.3 对实验的建议

- 细化实验指导书（如：增加对部分需要使用的框架接口介绍）

5.4 收获

通过本次实验课程，完成了一个简单编译器的实现。在四个实验中，我分别完成了词法分析器、语法分析器、语义分析和中间代码生成以及目标代码生成四个部分的设计，加深了我对课本对应知识点的理解，在实践中对课堂所学内容进行了充分的复习与巩固。同时实验课程也极大程度上提高了我利用编程解决问题能力。

在实验过程中遇到的许多问题，很大程度上都是受限于编程基础的薄弱和对给定代码框架的不熟悉，因此在后续学习中还要进一步提高自身的编程能力，在充分掌握课堂知识的基础上，多多尝试对课堂知识的实现。