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: 杨行

学　号: 200111325

专　业: 计算机科学与技术

日　期: 2022-11-01

1 实验目的与方法

实验目的：

本次实验主要通过完成四个部分实现一个简单的 TXT 语言编译器，目标平台为 RISC-V 32 。基于这种实践加深对理论课程中编译语言过程的理解，以及细节设计的体会。

1.1 词法分析器

通过自主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。并加深对词法分析程序的功能及实现方法的理解，同时对类 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。在此基础上实现从源程序中分出各种单词的方法，加深对课堂教学的理解，提高词法分析方法的实践能力。

1.2 语法分析

深入了解语法分析程序实现原理及方法。学会应用编译工作台等工具生成 LR 分析表。并通过实验中对 LR 语义分析中 ACTION 表与 GOTO 表转化为 Java 语言的流程实现，进一步理解 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。

1.3 典型语句的语义分析及中间代码生成

通过对语义分析及中间代码生成的实现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。并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。

1.4 目标代码生成

通过实验加深对编译器总体结构的理解与掌握，同时进一步掌握常见的 RISC-V 指令的使用方法，并在实现简单的程序编译的基础上进一步理解并掌握目标代码生成算法和寄存器选择算法。

实现方法：

本次实验采用的语言为 java；
实验的操作系统环境为 windows10/11;
实验平台采用的软件环境为基于 jdk17 的 IntelliJ IDEA 2021.3.2，RARS.

2 实验内容及要求

2.1 词法分析器

实验内容：使用自动机的写法实现一个简单的，特定的词法分析器. 即通过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。读入的源

语言的代码文本，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。并将生成的词法单元和符号表分别输出到文件中。

实验要求： 1.输入：以文件形式存放自定义的类 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，运算符和分界符等。

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进行语法分析，并输出推导过程中所用产生式序列并保存在输出文件中。即实现一个通用的 LR (1)语法分析驱动程序. 它可以读入词法单元类别, 任意的语法以及与之匹配的任意的 LR(1) 分析表, 随后读入词法单元流, 依次根据分析表与语法执行移入、规约、接受、报错动作, 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.同时框架预先提供了一个在每次规约的时候记录规约到的产生式的观察者. 该观察者将会按规约顺序输出所有记录到的产生式到文件中,通过记录的产生式判断是否正确实现了本实验。

实验要求：输入的单词符号串为实验一的输出。

2.3 典型语句的语义分析及中间代码生成

实验内容：完成 SDT 风格的语义分析与 IR 生成.这两个过程已经作为观察者被注册到语法分析器中, 其特定方法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. 在语义分析中, 你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. 在 IR 生成中, 你需要在遇到不同语法产生式时执行不同动作, 产生 IR (中间表示) 指令列表.

实验要求：

1.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。
2.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。
3.实现对应的声明语句、简单赋值语句、算术表达式的语义分析与中间代码生成。
4. 使用框架中的模拟器 IREmulator 验证生成的中间代码的正确性。

2.4 目标代码生成

实验内容及要求：将实验三生成的中间代码转换为目标代码（汇编指令）；使用 RARS 运行生成的目标代码，验证结果的正确性。

3 实验总体流程与函数功能描述

3.1. 词法分析

3.1.1. 编码表

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

3.1.2. 正则文法

正则文法表示：

约定：用 digit 表示数字：0,1,2,…,9; no_0_digit 表示数字：1,2,…,9;用 letter 表示字母：A,B,…,Z,a,b,…,z,以及下划线_；

则标识符：S→letter A　A→letter A|digit A| ε

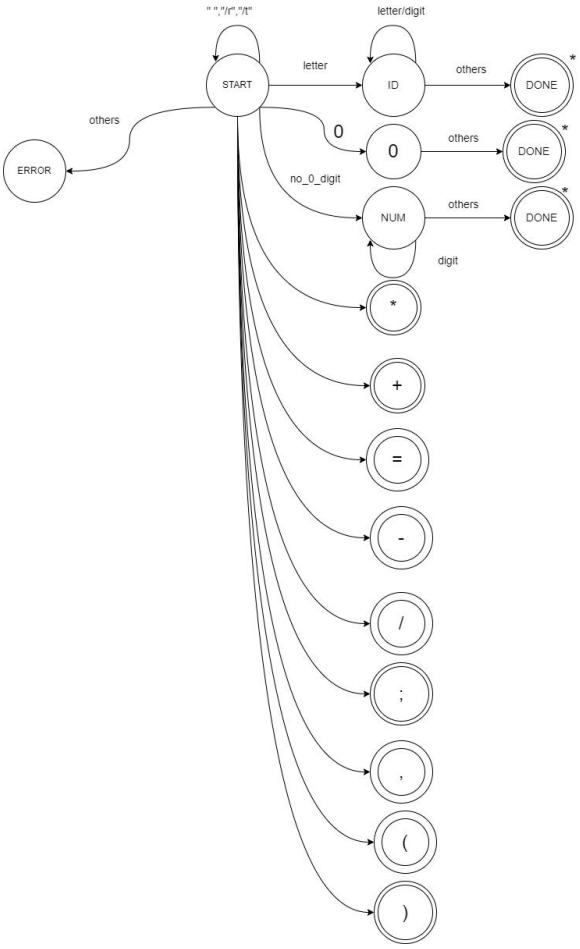
整常数：S ->no_0_digitB|0　　B -> digitB | ε

运算符：S -> C　　C -> =|*|+|-|/

3.1.3. 状态转换图

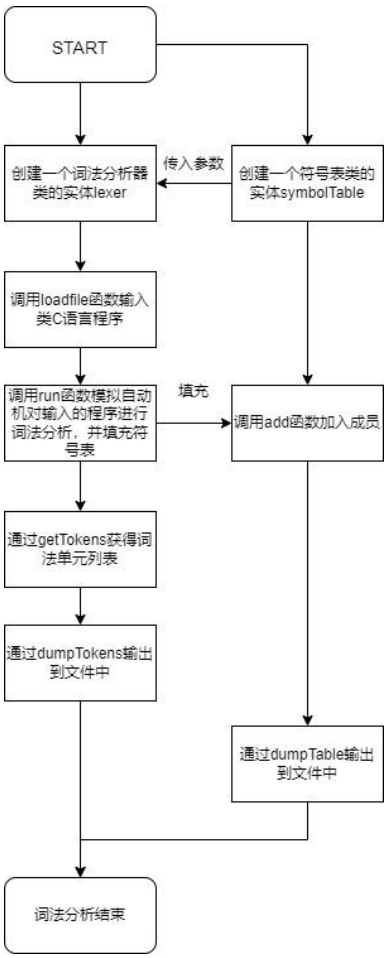
约定：用 digit 表示数字：0,1,2,…,9; no_0_digit 表示数字：1,2,…,9;用 letter 表示字母：A,B,…,Z,a,b,…,z,以及下划线_.

状态转换图如下：



3.1.4. 主要函数流程

词法分析主要函数流程图：



本次实验主要应用了词法分析器类 LexicalAnalyzer 以及符号表类 SymbolTable。

符号表类 SymbolTable 内部函数：

has（String text）：

输入：待判断符号的文本表示　　输出：该符号的条目是否位于符号表中的 bool 判断　功能：检测某符号是否存在于符号表中。

add(String text)：

输入：待加入符号表中的新符号的文本表示　输出：该符号在符号表中对应的新条目

功能：加入新符号于符号表中。

getAllEntries()：

输出：符号表的所有条目。

功能：获得符号表的所有条目以供 dumpTable 函数 使用。

get(String text)：

输入：符号的文本表示　　输出：该符号在符号表中的条目　功能：获取符号表中已有的条目。

dumpTable(String path)：

输入：输入文件的路径文本表示。 功能：按格式输出符号表中所有条目到指定文件中。

词法分析器类 LexicalAnalyzer 内部函数：

isNum(char ch)：功能：判断输入字符是否是数字。

isLetter (char ch)：功能：判断输入字符是否是字母。

isSign(char ch)：功能：判断输入字符是否是运算符与分隔符。

loadFile(String path)：功能：载入类 C 语言程序文件。

run() ：功能：模拟自动机完成词法单元的划分以及符号表的初步填充。

getTokens()：功能：获得词法分析的结果 Token 列表供 dumpTokens 使用。

dumpTokens(String path)：功能：按格式输出 Token 列表中所有条目到指定文件中。

3.2. 语法分析

3.2.1. 拓展文法

语法分析采用的拓展文法如下：

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

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 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												
38			reduce B -> (E)	reduce B -> (E)	reduce B -> (E)	reduce B -> (E)												

3.2.3. 状态栈和符号栈的数据结构

数据结构设计：

状态栈的数据结构：

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

状态栈的数据结构直接构建一个 Status 类的 Stack 型集合模拟状态栈。

符号栈的数据结构：

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

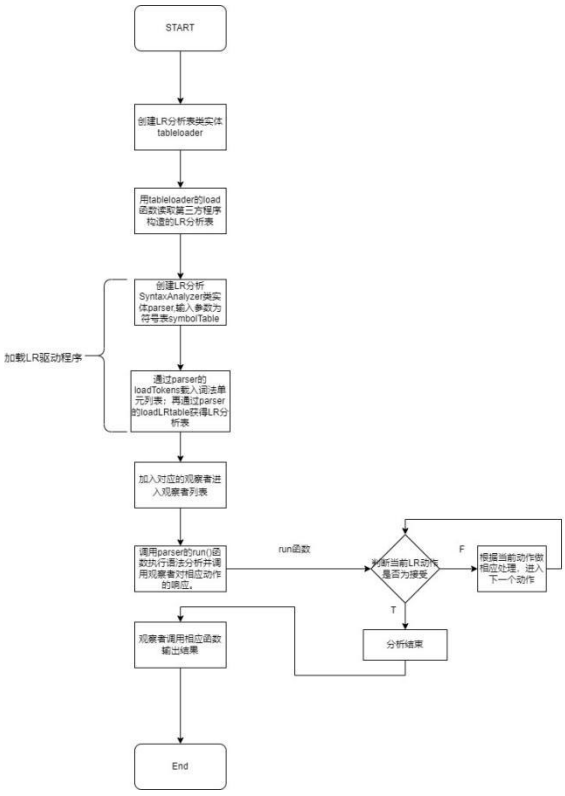
符号栈的数据结构直接构建一个 Symbol 类的 Stack 型集合模拟符号栈。

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

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



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

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

Symbol 类为 SyntaxAnalyzer 类的内部类，构建目的是词法单元 Token 与非终结符 NonTerminal 除了 Object 类之外没有公共的父类，分开构建栈过于繁琐，所以创建一个 union 的内部类 Symbol 类，其含有两个成员变量，词法单元类的 token 以及非终结符类的 nonTerminal.如果是词法单元 Token，nonTerminal 为 null;如果是词法单元非终结符 NonTerminal,token 为 null.达到一个 Symbol 栈同时储存词法单元和非终结符。

3.2.4. LR 驱动程序流程描述

LR 驱动程序流程图：



3.3. 语义分析和中间代码生成

3.3.1. 翻译方案

实验主要要求实现声明语句、简单赋值语句、算术表达式的语义分析与中间代码生成。

故翻译方案中涉及到的产生式只保留了声明语句与简单赋值语句、算术表达式的部分。

翻译方案如下：（temp 表示 IR 指令中的参数名,newtemp 生成新参数名）

S -> D id {p = lookup(id.name);if p!=null then enter(id.name,D.type) else error}(在符号表中找到 id 对应的条目，存在则将 id 对应条目的属性更改为 D.type.)

D -> int {D.type = int;}

S -> id = E; {gencode(id.val = E.val);createMov (id,E.temp) }

S -> return E;{createRet(E.temp);}

E -> E + A;{createAdd(newtemp,E.temp,A.temp); E.val = E.val + A.val}

E -> E - A;{createSub(newtemp,E.temp,A.temp); E.val = E.val - A.val}

E -> A {E.temp = A.temp;}

A -> A*B; {createMul(newtemp,A.temp,B.temp); A.val = A.val * B.val}

A -> B; {A.temp = B.temp;}

B -> (E); {B.temp = E.temp;}

B -> id; {B.temp = id.name}

B -> IntConst {B.val = IntConst.lexval;B.temp = IntConst}

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：

符号表：

```
private  SymbolTable symbolTable =new SymbolTable();
```

符号表中含有一个图类型的数据结构存储初始的符号信息，语义分析需要调用进行修改。

符号栈和语义栈：

```
private Stack<Symbol> symbols = new Stack<>();
```

构建一个 Symbol 类的 Stack 型集合模拟。

```
class Symbol{
    Token token;
    NonTerminal nonTerminal;
    SourceCodeType sourceCodeType;

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



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

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

    public void setSourceCodeType(SourceCodeType sourceCodeType) { this.sourceCodeType = sourceCodeType; }
}
```

Symbol 类为 SemanticAnalyzer 类的内部类，构建目的是词法单元 Token 与非终结符 NonTerminal 除了 Object 类之外没有公共的父类，分开构建栈过于繁琐，所以创建一个 union 的内部类 Symbol 类，同时为了记录符号对应的语义，将 Symbol 设置含有 3 个成员变量，词法单元类的 token，非终结符类的 nonTerminal 以及表示符号语义类型的 sourceCodeType.如果是词法单元 Token，nonTerminal 为 null;如果是词法单元非终结符 NonTerminal,token 为 null.同时将符号的语义类型初始值设为 null，通过 set 函数在需要时设置为相应的类型，从而模拟语义以及符号出入栈。

中间代码生成使用的数据结构：

中间代码指令集：

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

创建一个 List 存储 Instruction 类的中间代码。

符号栈：

```
private Stack<Symbol> symbols = new Stack<>();
```

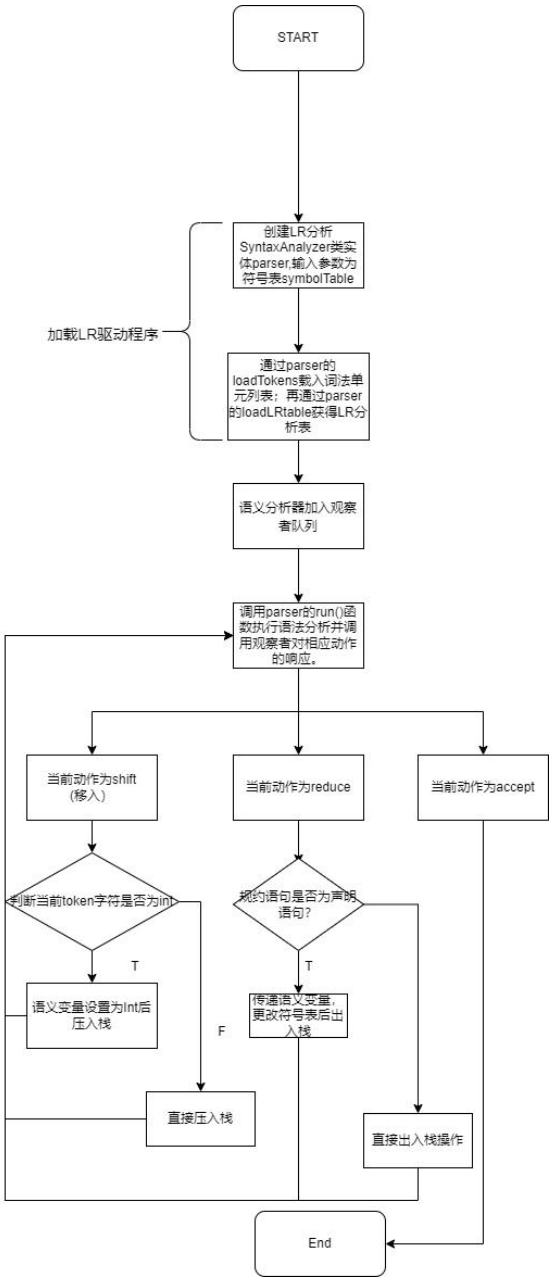
构建一个 Symbol 类的 Stack 型集合模拟。

```
class Symbol{
    Token token;
    NonTerminal nonTerminal;
    private IRValue from;

    private Symbol(Token token, NonTerminal nonTerminal){
        this.token = token;
        this.nonTerminal = nonTerminal;
       // this.val = Integer.parseInt(null);
        this.from = null;
    }



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

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

    public void setFrom(IRValue from) { this.from = from; }
}
```

Symbol 类为 SemanticAnalyzer 类的内部类，构建目的是词法单元 Token 与非终结符 NonTerminal 除了 Object 类之外没有公共的父类，分开构建栈过于繁琐，所以创建一个 union 的内部类 Symbol 类，同时为了记录符号在中间代码中对应的操作变量，将 Symbol 设置含有 3 个成员变量，词法单元类的 token，非终结符类的 nonTerminal 以及表示操作变量的 IRvalue 类型的 from.如果是词法单元 Token，nonTerminal 为 null; 如果是词法单元非终结符 NonTerminal,token 为 null.同时将符号表示操作变量的初始值设为 null，通过 set 函数在需要时设置为相应的类型，从而模拟操作变量携带数据信息在中间代码中转移以及符号出入栈。

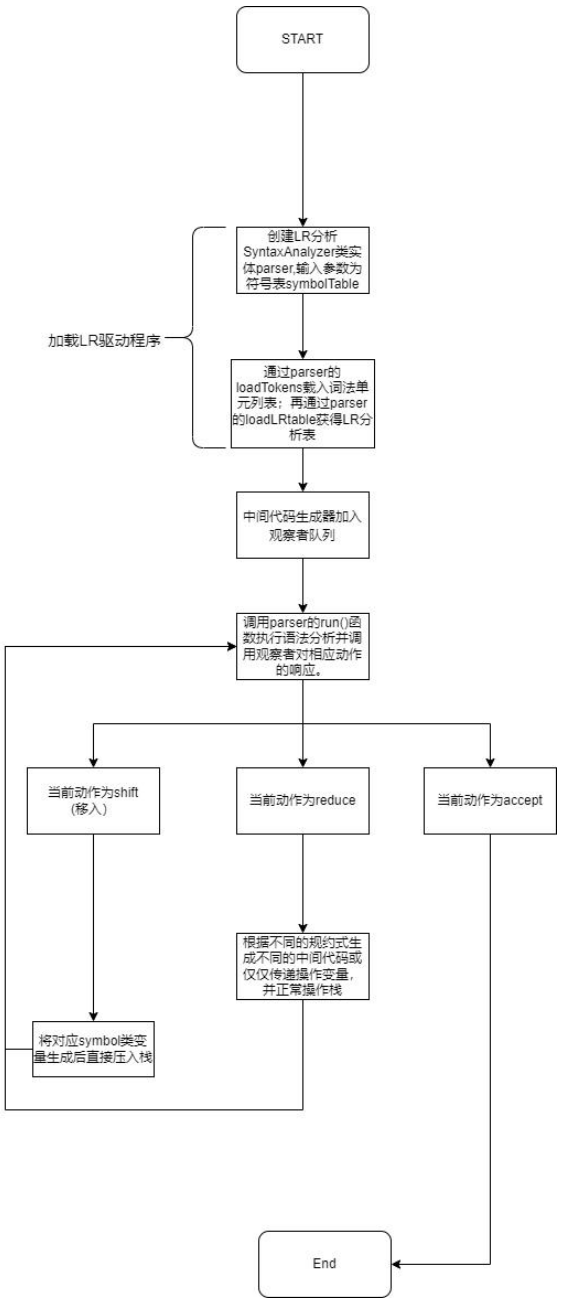
3.3.3. 主要流程描述（两个观察者的实现）

语义分析的流程图如下：



简要说明：通过将语义分析器加入观察者列表，语义分析器通过观察 parser 的动作，作出相应反应，如果当前动作为移入，且移入的 token 的字符是 int，则将其转变为 symbol 类实体后，sourceCodeType 设置为 Int 后压入栈。当前动作为规约时，则判断规约式类型为 D->int 时,则将 D 转变为 symbol 类实体后的 sourceCodeType 设置为 Int 后压入栈，同时正常出入栈。为 S -> D id 时，则在旧符号表中遍历，将 id 的 name 对应的符号语义类型更改为 D 的 sourceCodeType.其他情况正常出入栈。动作为接受，表示分析结束。

中间代码生成的流程图如下：

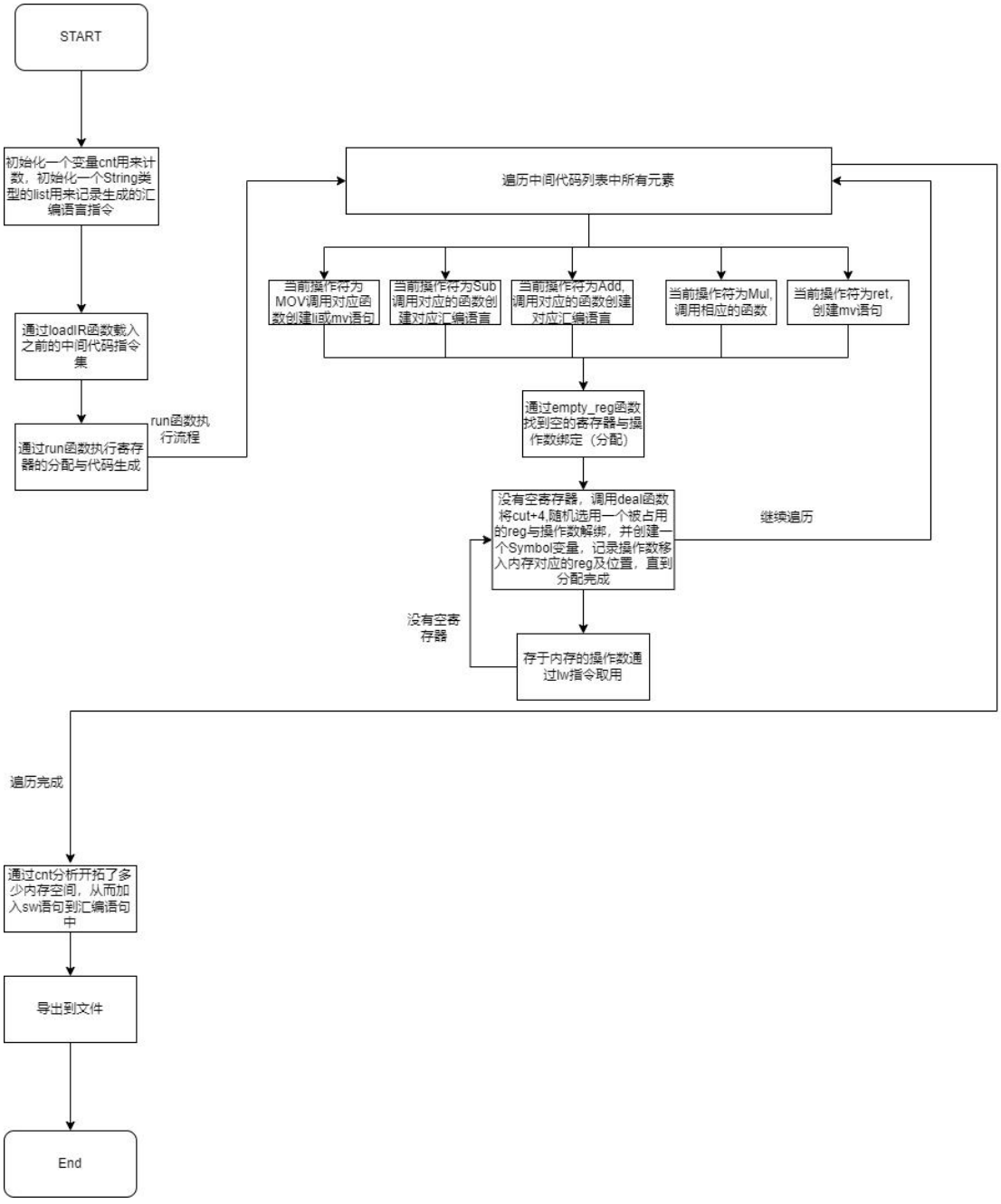


简要说明：通过将中间代码器加入观察者列表，中间代码器器通过观察 parser 的动作，作出相应反应，如果当前动作为移入，则将其转变为 symbol 类实体后压入栈。当前动作为规约时，则判断当前规约式类型判断应用的操作，例如 E -> E - A 便是利用 createSub 生成减法对于生成的中间表达式要加入储存的链表，B -> id，B -> IntConst，E -> A 一类仅仅只是传递表示操作变量的 IRvalue 类型的 from 值。动作为接受，表示分析结束。

3.4. 目标代码生成

3.4.1. 主要流程描述

流程图如下：



简要流程及关键函数，数据结构说明：

数据结构的初始化：

```
private List<Instruction> instructions = new ArrayList<>();
private BMap<IRValue,Reg> bMap = new BMap<>();
private List<Symbol> symbols = new ArrayList<>();
private List<Active> actives = new Stack<>();
private List<String> ASM = new ArrayList<>();
```

首先创建一个 Instruction 类型的 List 集合存放读取的中间代码，创建一个 BMap 图建立寄存器与操作数 IRValue 的绑定联系，创建一个 Symbol 类型的 List 集合存放存于内存中的数据，创建一个 actives 存储存活变量，ASM 集合存放生成的汇编语言。

```
class Symbol{
    IRValue irValue;
    Reg reg;
    int num;

    private Symbol(IRValue irValue,Reg reg,int num){
        this.irValue = irValue;
        this.reg = reg;
        this.num = num;
    }

}
```

Symbol 类为内部类，存放操作数 irValue，占用内存所处的寄存器基地址通过 reg 表示，以及地址偏移量 num。即建立操作数和内存地址的关系。

```
class Active {
    IRValue irValue;
    int num;

    private Active(IRValue irValue){
        this.irValue = irValue;
        this.num = 0;
    }

    public void up(){
        num++;
    }

    public void down(){
        num--;
    }
```

Active 类为内部类，存放变量名以及 irValue，活跃度用 num 表示，指令执行完将 num=0 即不存活的变量移除。

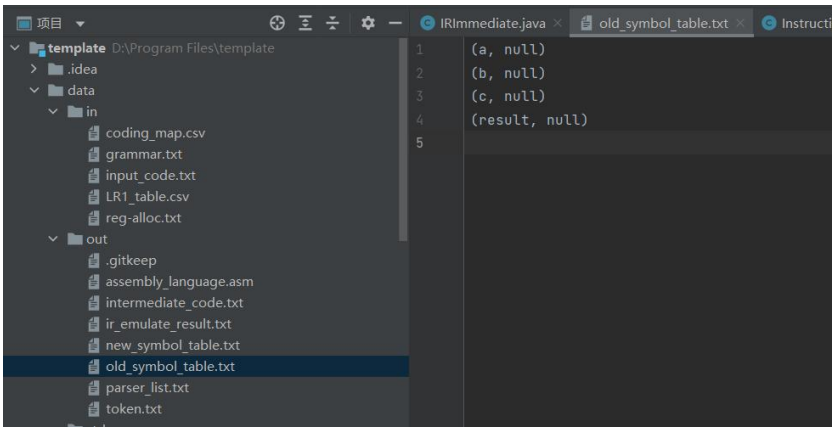
通过 loadIR 函数载入中间代码，并进行预处理，之后执行 run 函数。run 函数进行中间代码的遍历，通过判断当前中间代码的操作符，进行对应函数的调用，生成汇编代码。例如 MOV 调用 dealMOV 根据操作数的类型生成 li 或 mv 类型汇编。SUB 调用 dealSUB 根据操作数的类型，生成 sub 类型汇编。调用的函数存在类似的操作，即将操作数与寄存器相关连起来，并采用 BMap（内含两个图）维护双射关系。

重点在于寄存器分配，首先分配空闲的寄存器，如果寄存器都满则向内存储存腾出空闲寄存器。每次 run 中一个中间代码分析完成，就通过 release 函数将不存活的变量移除出内存和寄存器，腾出空闲空间。

最后通过 dumpfile 函数输出到文件中。

4 实验结果与分析

词法分析：
输入：coding_map.csv input_code.txt
输出：old_symbol_table.txt token.txt



old_symbol_table.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)
(-,)
((,)
```

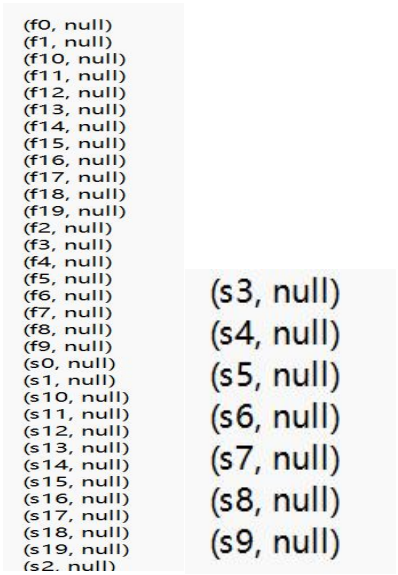
```
(IntConst,3)
(+,)
(id,b)
(),)
(*,)
((,)
(id,c)
(-,)
(id,a)
(),)
(Semicolon,)
(return,)
(id,result)
(Semicolon,)
($,)
```

Token.txt

输入：coding_map.csv reg-alloc.txt
输出：old_symbol_table.txt token.txt

```
(int,)
(id,f0)
(Semicolon,)
(int,)
(id,f1)
(Semicolon,)
(int,)
(id,f2)
(Semicolon,)
(int,)
(id,f3)
(Semicolon,)
(int,)
(id,f4)
(Semicolon,)
(int,)
(id,f5)
(Semicolon,)
(int,)
(id,f6)
(Semicolon,)
(int,)
(id,f7)
(Semicolon,)
(int,)
(id,f8)
(=,)
(id,s15)
(+,)
(id,f16)
(Semicolon,)
(id,s17)
(=,)
(id,s16)
(+,)
(id,f17)
(Semicolon,)
(id,s18)
(=,)
(id,s17)
(+,)
(id,f18)
(Semicolon,)
(id,s19)
(=,)
(id,s18)
(+,)
(id,f19)
(Semicolon,)
(return,)
(id,s19)
(Semicolon,)
($,)
```

token.txt



old_symbol_table.txt

两个文件的词法分析符合期望。

语法分析：

文件输入：

```
├──── coding_map.csv        # 码点文件
├──── grammar.txt           # 语法文件
├──── input_code.txt        # 输入代码
└──── LR1_table.csv         # 第三方工具生成的 IR 分析表
```

输出：

```
├──── parser_list.txt           # 规约过程的产生式列表
├──── old_symbol_table.txt      # 语义分析前的符号表
└──── token.txt                 # 词法单元流
```

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

```
S_list -> S Semicolon
S_list -> S Semicolon S_list
S_list -> S Semicolon S_list
S_list -> S Semicolon S_list
S_list -> S Semicolon S_list
S_list -> S Semicolon S_list
S_list -> S Semicolon S_list
S_list -> S Semicolon S_list
S_list -> S Semicolon S_list
P -> S_list
```

输入：
├──── coding_map.csv   # 码点文件
├──── grammar.txt   # 语法文件
├──── reg-alloc.txt   # 输入代码
└──── LR1_table.csv   # 第三方工具生成的 IR 分析表
主要输出：
└────parser_list.txt   # 规约过程的产生式列表

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

```
S_list -> S Semicolon S_list
S_list -> S Semicolon S_list
S_list -> S Semicolon S_list
S_list -> S Semicolon S_list
S_list -> S Semicolon S_list
S_list -> S Semicolon S_list
S_list -> S Semicolon S_list
S_list -> S Semicolon S_list
S_list -> S Semicolon S_list
S_list -> S Semicolon S_list
S_list -> S Semicolon S_list
S_list -> S Semicolon S_list
S_list -> S Semicolon S_list
S_list -> S Semicolon S_list
S_list -> S Semicolon S_list
S_list -> S Semicolon S_list
S_list -> S Semicolon S_list
S_list -> S Semicolon S_list
S_list -> S Semicolon S_list
S_list -> S Semicolon S_list
S_list -> S Semicolon S_list
S_list -> S Semicolon S_list
S_list -> S Semicolon S_list
S_list -> S Semicolon S_list
S_list -> S Semicolon S_list
P -> S_list
```

parser_list.txt 部分结果截图

结果符合预期

语义分析以及 IR 生成：

输入：
├── coding_map.csv   # 码点文件
├── grammar.txt   # 语法文件
├── input_code.txt   # 输入代码

└── LR1_table.csv　　# 第三方工具生成的 IR 分析表

输出：

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

```

new_symbol_table.txt　　# 语义分析后的符号表

```
144
```

ir_emulate_result.txt　　# 中间表示的模拟执行的结果

```
(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)
```

intermediate_code.txt　　# 中间表示

输入：reg-alloc.txt

```
(f0, Int)
(f1, Int)
(f10, Int)
(f11, Int)
(f12, Int)
(f13, Int)
(f14, Int)
(f15, Int)
(f16, Int)
(f17, Int)
(f18, Int)
(f19, Int)
(f2, Int)
(f3, Int)
(f4, Int)
(f5, Int)
(f6, Int)
(f7, Int)
(f8, Int)
(f9, Int)
(s0, Int)
(s1, Int)
(s10, Int)
```

new_symbol_table.txt　　# 语义分析后的符号表

```
10945
```

ir_emulate_result.txt　　# 中间表示的模拟执行的结果

```
(MOV, f0, 0)
(MOV, f1, 1)
(ADD, $0, f1, f0)
(MOV, f2, $0)
(ADD, $1, f2, f1)
(MOV, f3, $1)
(ADD, $2, f3, f2)
(MOV, f4, $2)
(ADD, $3, f4, f3)
(MOV, f5, $3)
(ADD, $4, f5, f4)
(MOV, f6, $4)
(ADD, $5, f6, f5)
(MOV, f7, $5)
```

intermediate_code.txt　　# 中间表示部分截图

汇编生成：

输入：input_code.txt

输出：

```
.text
    li t0, 8          #  (MOV, a, 8)
    li t1, 5          #  (MOV, b, 5)
    li t2, 3          #  (MOV, $6, 3)
    sub t3, t2, t0      #  (SUB, $0, $6, a)
    mv t2, t3         #  (MOV, c, $0)
    mul t3, t0, t1      #  (MUL, $1, a, b)
    addi t4, t1, 3      #  (ADD, $2, b, 3)
    sub t1, t2, t0      #  (SUB, $3, c, a)
    mul t0, t4, t1      #  (MUL, $4, $2, $3)
    sub t1, t3, t0      #  (SUB, $5, $1, $4)
    mv t0, t1         #  (MOV, result, $5)
    mv a0, t0         #  (RET, , result)
```

assembly_language.asm　　# 汇编代码

从第 6 行可以看出 t2 原与$6 互相绑定，当第 5 句执行完$6 不存活释放，故后来的可以占用其寄存器。

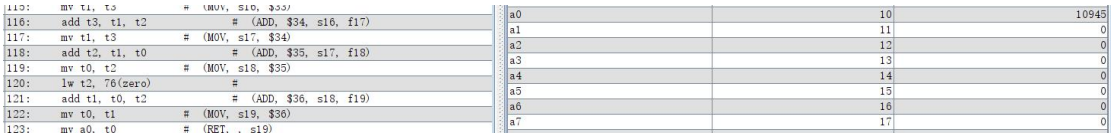
t2	7	-5
s0	8	0
s1	9	0
a0	10	144

执行结果

输入：reg-alloc.txt

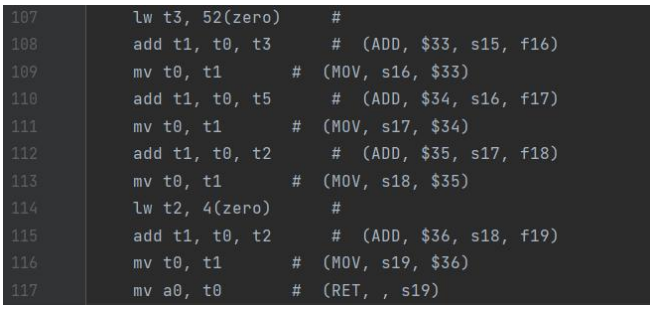
输出：asm 汇编文件

```
    add t2, t1, t0      #  (ADD, $35, s17, f18)
    mv t0, t2         #  (MOV, s18, $35)
    lw t2, 76(zero)     #
    add t1, t0, t2      #  (ADD, $36, s18, f19)
    mv t0, t1         #  (MOV, s19, $36)
    mv a0, t0         #  (RET, , s19)
```

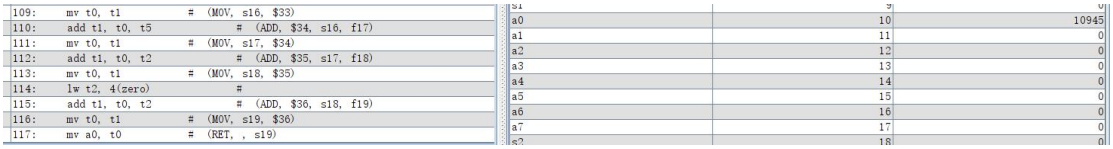


```
116:    add t3, t1, t2          #  (ADD, $34, s16, f17)
117:    mv t1, t3               #  (MOV, s17, $34)
118:    add t2, t1, t0          #  (ADD, $35, s17, f18)
119:    mv t0, t2               #  (MOV, s18, $35)
120:    lw t2, 76(zero)         #
121:    add t1, t0, t2          #  (ADD, $36, s18, f19)
122:    mv t0, t1               #  (MOV, s19, $36)
123:    mv a0, t0               #  (RET, , s19)
```

a0	10	10945
a1	11	0
a2	12	0
a3	13	0
a4	14	0
a5	15	0
a6	16	0
a7	17	0

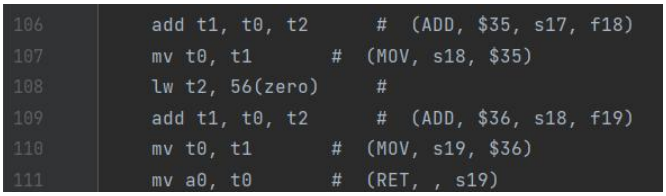


```
107        lw t3, 52(zero)     #
108        add t1, t0, t3      #  (ADD, $33, s15, f16)
109        mv t0, t1        #  (MOV, s16, $33)
110        add t1, t0, t5      #  (ADD, $34, s16, f17)
111        mv t0, t1        #  (MOV, s17, $34)
112        add t1, t0, t2      #  (ADD, $35, s17, f18)
113        mv t0, t1        #  (MOV, s18, $35)
114        lw t2, 4(zero)      #
115        add t1, t0, t2      #  (ADD, $36, s18, f19)
116        mv t0, t1        #  (MOV, s19, $36)
117        mv a0, t0        #  (RET, , s19)
```

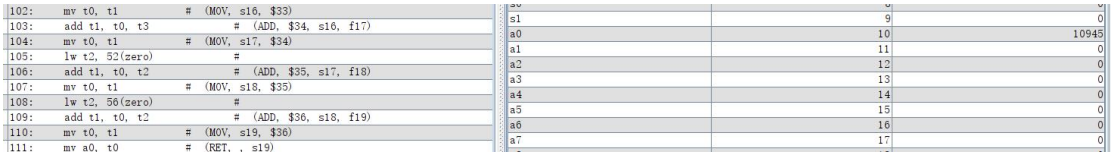


```
109:    mv t0, t1               #  (MOV, s16, $33)
110:    add t1, t0, t5          #  (ADD, $34, s16, f17)
111:    mv t0, t1               #  (MOV, s17, $34)
112:    add t1, t0, t2          #  (ADD, $35, s17, f18)
113:    mv t0, t1               #  (MOV, s18, $35)
114:    lw t2, 4(zero)          #
115:    add t1, t0, t2          #  (ADD, $36, s18, f19)
116:    mv t0, t1               #  (MOV, s19, $36)
117:    mv a0, t0               #  (RET, , s19)
```

a0	10	10945
a1	11	0
a2	12	0
a3	13	0
a4	14	0
a5	15	0
a6	16	0
a7	17	0



```
106        add t1, t0, t2      #  (ADD, $35, s17, f18)
107        mv t0, t1        #  (MOV, s18, $35)
108        lw t2, 56(zero)     #
109        add t1, t0, t2      #  (ADD, $36, s18, f19)
110        mv t0, t1        #  (MOV, s19, $36)
111        mv a0, t0        #  (RET, , s19)
```



```
102:    mv t0, t1               #  (MOV, s16, $33)
103:    add t1, t0, t3          #  (ADD, $34, s16, f17)
104:    mv t0, t1               #  (MOV, s17, $34)
105:    lw t2, 52(zero)         #
106:    add t1, t0, t2          #  (ADD, $35, s17, f18)
107:    mv t0, t1               #  (MOV, s18, $35)
108:    lw t2, 56(zero)         #
109:    add t1, t0, t2          #  (ADD, $36, s18, f19)
110:    mv t0, t1               #  (MOV, s19, $36)
111:    mv a0, t0               #  (RET, , s19)
```

s1	9	0
a0	10	10945
a1	11	0
a2	12	0
a3	13	0
a4	14	0
a5	15	0
a6	16	0
a7	17	0

执行 3 次可以发现由于寄存器占用满时分配，是随机选取一个寄存器将对应数据输入到内存，由于随机性的所以汇编语言的长度并不相同。但执行结果与模拟结果一致。符合预期。

5 实验中遇到的困难与解决办法

词法分析中遇到的问题主要来自对自动机模拟时回溯的判断，通过详细分析代码，复习理论知识，先作出自动机，在进行模拟。

语法分析的主要目的是对规约式的记录，遇到的问题主要来自 goto 时弹栈有错误，通过插桩打印元素到终端解决这个问题。

语义分析和 IR 代码生成的困难在于非终结符不能存储欣喜，所以不能直接将翻译方案直接照搬，需要合理设计数据结构模拟动作以及属性传递，语义分析时通过构建内部类 Symbol，对其构建栈来模拟语义栈和符号栈。IR 代码生成时同样通过构建内部类 Symbol，对其构建栈来模拟属性栈和符号栈。

汇编生成遇到的困难比较多，主要来自寄存器的分配以及栈的分配中存活变量的记录，解决方法是生成中间代码时同时记录变量，出现一次加入变量列表，初始 num 记为 0，再次出现 num+1. 当执行时，通过控制 num 加减，移

除 num 为 0 变量与寄存器及内存栈的关系达到目的。

收获：通过本次实验我通过自主设计 java 语言对类 C 语言源程序段进行分析，加深对高级语言的认识。并加深对词法分析程序的功能及实现方法的理解，同时对类 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。并且学会了简单应用编译工作台等工具生成 LR 分析表。在通过实验进一步理解 LR(1)分析法的执行流程与性质的同时加深了对自底向上语法制导翻译技术的理解，提高掌握声明语句、赋值语句和算术运算语句的翻译方法的应用程度。最后通过对汇编生成的设计，进一步理解语言设计内存分配的设计思路，进一步掌握汇编语言的工作流程。

不足：本次实验勉强实现了跑过样例 reg-alloc.txt ，但仍有许多不足，寄存器分配的策略采用随机的方案导致汇编语言的长度有时长有时短，不能维持在一个期望的较短值。维护栈空间的方式仅仅只是清除不存活变量对栈的占用，没有定期清理或移动达到维护效果，对于基于 x0 的偏移量超过 lw, sw 指令立即数范围的访存如何解决没有很好的思路。并且实验 4 实现的代码有许多冗杂的地方，由于种种问题没有得到优化。

建议：增加对整体框架的介绍，便于快速上手以及进一步理解一个完整编译器的模块设计思路。实验报告可以一开始就发布，使得每完成一次实验可以相应的进行记录设计思路，记录实验结果，便于与后续内容更新时比对。