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: 邓皓元

学　号: 200110618

专　业: 计算机科学与技术

日　期: 2022-10-30

1 实验目的与方法

实验总目的：实现一个目标平台是 RISC-V32 的编译器

实验方法与环境：JAVA 语言、IntelliJ IDEA Community、JDK17 及以上版本、RARS、编译工作台

1.1 词法分析器

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

1.2 语法分析

深入了解语法分析程序实现原理及方法

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

巩固对语义分析的基本功能和原理的认识

加深对自底下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

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

总体要求：读入文件存放在 data/in 目录下、输出文件存放在 data/out 目录下，输出文件内容参考存放在 data/std 目录下的文件

2.1 词法分析器

内容：编写一个词法分析程序，首先读取码点文件 coding_map.csv，生成符号表并打印成输出文件 old_symbol_table.txt，之后对文件内的类 C 语言程序段 input_code.txt 进行词法分析。 处理 C 语言源程序，过滤掉无用符号(空格、tab、换行等)，分解出正确的单词，以二元组形式存输出放在词法单元列表文件 token.txt 中。

要求：实现词法分析前的缓冲区、编写代码实现自动机进行词法分析的过程、以适当的数据结构保存词法分析的到的 token 列表

2.2 语法分析

内容：利用 LR(1)分析法，设计语法分析程序，结合编译工作台得到的 LR(1)文法文件 LR1_table.csv 对输入单词符号串(实验一的输出 token.txt)进行语法分析，输出推导过程中所用产生式序列并保存在输出文件 parser_list.txt 中

要求：实现实验一得到的词法单元列表的加载和存放、实现 LR 分析表的加载和使用方法、实现语法分析的驱动程序

2.3 典型语句的语义分析及中间代码生成

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(语义分析每次动作都通知语义分析器和中间代码生成器做相应的反应)，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 intermediate_code.txt(中间代码序列)和 new_symbol_table.txt(更新的符号表)中，中间代码使用三地址码的四元式表示。实现声明语句、简单赋值语句和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 验证生成的中间代码的正确性，将模拟器 IREmulator 验证生成的结果保存在 ir_emulate_resule 中。

要求：设计文法的翻译方案、设计各个观察者需要的用于语义分析和中间代码生成的数据结构（栈）、实现各观察者在分析过程中的动作处理、使用 IREmulator 对生成的中间代码进行验证。

2.4 目标代码生成

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

要求：加载前端提供的中间代码，视情况做预处理（预处理思路参考指导书），实现寄存器选择算法、实现目标代码生成算法、输出生成的目标代码到指定文件 assembly_language.asm 中、使用 Rars 运行目标代码，验证其正确性。

实验四不检查 out/assembly_language.asm 和 std/assembly_language.asm 是否一致， 而是验证通过 rars 运行汇编指令后得到的结果是否正确。

3 实验总体流程与函数功能描述

3.1 词法分析

3.1.1 编码表

词法分析器的输出是单词序列，单词分为关键字、运算符、分界符、标识符、常数五种单词种类，其中，关键字、运算符、分界符都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，而编码表就是可供查看单词类别和编码的表格。

3.1.2 正则文法

用 digit 表示数字：0,1,2,…,9;

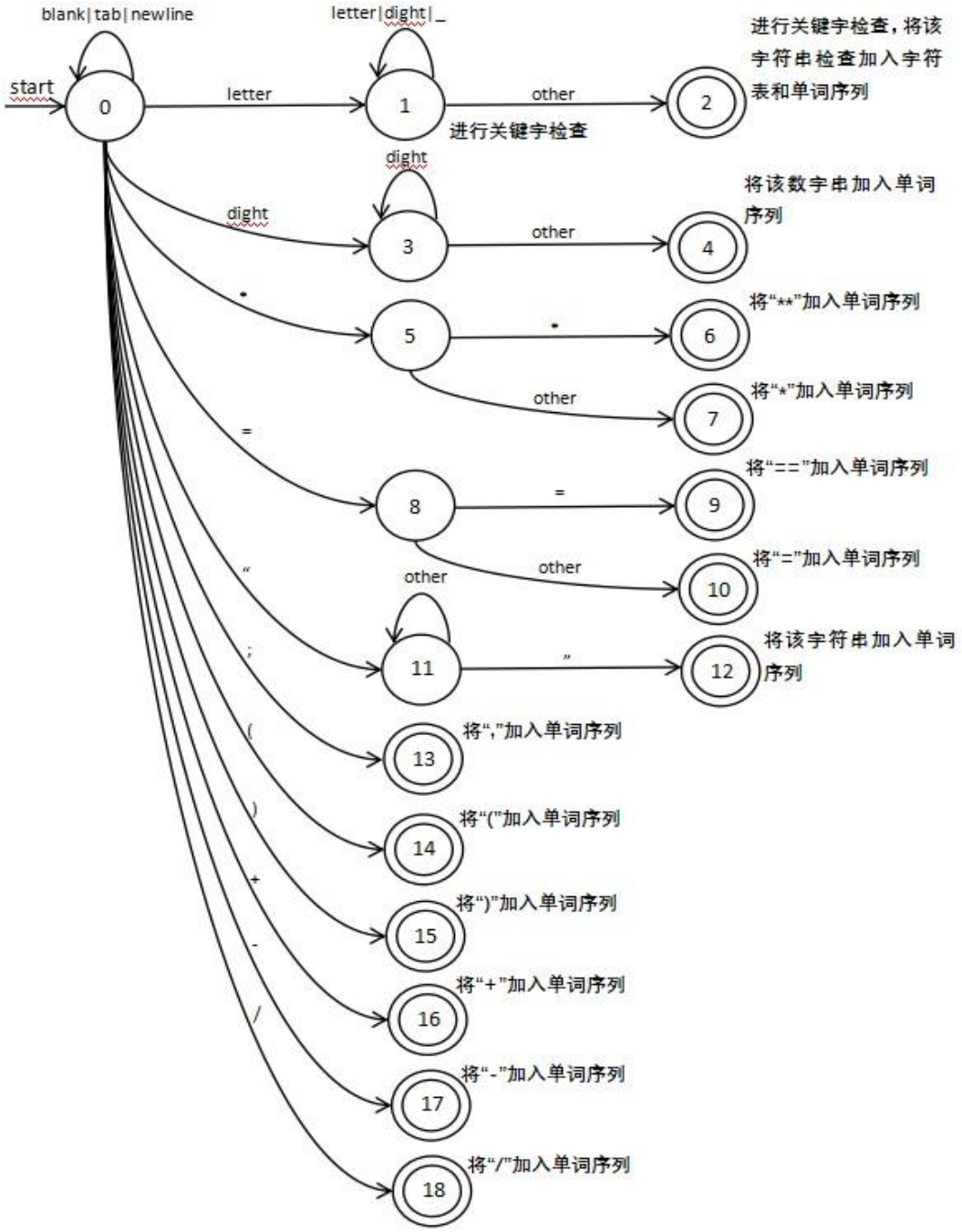
用 letter 表示字母：A,B,…,Z,a,b,…,z;

标识符：S→letter A，A→letter A|digit A|_A|ε

整常数：S ->digitB，B -> digitB | ε

运算符：S -> C，C -> =D|*E|+|-|/，D -> =| ε，E -> *| ε

3.1.3 状态转换图



3.1.4 主要函数流程

循环检测输入的字符串序列，对每个字符串从头开始检测每个字符，用 left 和 right 指针表示正在读入的字符串(每个字符串开始读取则将 left 和 right 赋值为 0)循环读入直到 left 为当前字符串长度。

假设当前正在检测的字符串为 str，长度为 length

0 状态(字符串序列中每个字符串开始读取的状态):

如果当前检测到的字符(str.charAt(left))为空格、换行或者 tab，跳过当前字符，进入状态 0(left++,right++)。

如果当前检测到的字符(str.charAt(left))为 letter，进入字符串检测状态，进入状态 1(right++)。

如果当前检测到的字符(str.charAt(left))为 dight，进入数字串检测状态，进入状态 2(right++)。

如果当前检测到的字符(str.charAt(left))为*，进入乘号检测状态，进入状态 3(right++)。

如果当前检测到的字符(str.charAt(left))为=，进入等号检测状态，进入状态 4(right++)。

如果当前检测到的字符(str.charAt(left))为“，进入左双引号检测状态，进入状态 5(right++)。

如果当前检测到的字符(str.charAt(left))为;，将 Semicolon 作为简单单词加入单词序列 tokenlist，回到状态 0(left++,right++)。

如果当前检测到的字符(str.charAt(left))为(，将(作为简单单词加入单词序列 tokenlist，进入状态 0(left++,right++)。

如果当前检测到的字符(str.charAt(left))为)，将)作为简单单词加入单词序列 tokenlist，进入状态 0(left++,right++)。

如果当前检测到的字符(str.charAt(left))为+，将+作为简单单词加入单词序列 tokenlist，进入状态 0(left++,right++)。

如果当前检测到的字符(str.charAt(left))为-，将-作为简单单词加入单词序列 tokenlist，进入状态 0(left++,right++)。

如果当前检测到的字符(str.charAt(left))为/，将/作为简单单词加入单词序列 tokenlist，进入状态 0(left++,right++)。

1 状态(字符串检测状态):

如果当前检测到的字符(str.charAt(right))为 letter|dight|_，保持字符串检测状态，进入状态 1(right++)。

如果当前检测到的字符为其他字符，先检测当前字符串(str.substring(left,right)，不包括当前检测到的字符)是否为关键字，若是关键字(Tokenkind.isAllowed(str.substring(left,right)))，则将该字符串作为简单单词(Token.simple(str.substring(left,right)))加入单词序列 tokenlist，否则将该字符串作为一般单词(Token.normal(“id”，str.substring(left,right)))加入单词序列 tokenlist，进入状态 0(left=right)。

2 状态(数字串检测状态):

如果当前检测到的字符(str.charAt(right))为 dight，保持数字串检测状态，进入状态 2(right++)。

如果当前检测到的字符为其他字符，则将该字符串(str.substring(left,right)作为一般单词(Token.normal(“IntConst”，str.substring(left,right)))加入单词序列 tokenlist，进入状态 0(left=right)。

3 状态(乘号检测状态):

如果当前检测到的字符(str.charAt(right))为*，将**作为简单单词（Token.simple(“**”))加入单词序列 tokenlist，进入状态 0(right++,left=right)。

如果当前检测到的字符为其他字符，将*作为简单单词（Token.simple(“*”))加入单词序列 tokenlist，进入状态 0(left=right)。

4 状态(等号检测状态):

如果当前检测到的字符(str.charAt(right))为=，将==作为简单单词（Token.simple(“==”))加入单词序列 tokenlist，进入状态 0(right++,left=right)。

如果当前检测到的字符为其他字符，将=作为简单单词（Token.simple(“=”))加入单词序列 tokenlist，进入状态 0(left=right)。

5 状态(左引号检测状态):

如果当前检测到的字符(str.charAt(right))为”，将该字符串(str.substring(left+1,right-1)作为一般单词(Token.normal(“StrConst”，str.substring(left+1,right-1)))加入单词序列 tokenlist，进入状态 0(right++,left=right)

如果当前检测到的字符为其他字符，保持左引号检测状态，进入状态 5(right++)。

3.2 语法分析

3.2.1 拓展文法

对文法进行拓广的目的是为了对某些右部含有开始符号的文法，在归约过程中能分清是否已归约到文法的最初开始符。而本次实验中的 grammer.txt 文法开始符为 P，而产生式右部不含 P，因此不需要进行拓展。

```
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

根据程序设计语言语法的文法构造 LR(1)分析表，实验中借助编译工作台完成，LR(1)分析表的构建根据每个非终结符的 First 集和 Follow 集以及根据状态栈栈顶状态以及符号栈栈顶符号进行状态转移和选择移入和归约操作。

3.2.3 状态栈和符号栈的数据结构

由于数据结构栈拥有的“后入先出”属性，状态栈和符号栈都采取栈的数据结构来进行状态和符号的保存。

3.2.4 LR 驱动程序流程描述

1.将单词序列 TokenList 存入队列(队列“先入先出”的特性适合单词序列的存入)tokensQueue<Token>。

2.保存 LR 分析表，创建并初始化状态栈 statusStack<Status>和符号栈 termsStack<Term>

3.首先将 LR 分析表的首状态压入状态栈

4.循环查询状态栈栈顶状态直到栈顶为空(错误)或者动作为错误(错误)或者动作为接受(执行接受)

循环体：

I.判断状态栈和单词队列是否为空，都不为空则记录状态栈栈顶状态 statu，记录单词队列队头单词 token，存在为空则抛出错误

II.检查当前状态栈栈顶状态 statu 根据单词队列队头单词 token 的动作 action：

若执行动作为 shift 转移：在执行 shift 动作时通知各个观察者，把转移动作的下一个状态压入状态栈，把单词队列队头单词 token 的 TokenKind 压入符号栈

若执行动作为 reduce 规约：在执行 reduce 动作时通知各个观察者，依次检查规约产生式右部 body 和符号栈处于队头的符号是否对应，对应则弹出一个符号栈和状态栈的符号和状态，将产生式左部 head 压入符号栈，否则抛出错误。判断符号栈非空且符号栈栈顶为非终止符且符号栈栈顶 goto 状态不是 Error,则执行 goto，将当前状态栈栈顶状态 statu 的 goto 状态压入状态栈。

若执行动作为 accept 接受：在执行 accept 动作时通知各个观察者

若执行动作为 error 错误：抛出错误

IV.判断执行动作是否为 accept，若为 accept 则弹出状态栈栈顶元素，跳出循环

5.执行完毕

3.3 语义分析和中间代码生成

3.3.1 翻译方案

语义分析：

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具有 type 属性的符号；

与语义分析相关的产生式有 S -> D id 和 D -> int；

自底向上的分析过程中 Shift 时将带有 type 属性(从 Token 中获得)的符号入栈；

自底向上的分析过程中 Reduce 时，先记录并出栈产生式右部符号

如果规约时使用的产生式是 D -> int,需要将带有 int 的 type 属性的 D 符号入栈；

如果规约时使用的产生式是 S -> D id,更新符号表中相应变量 id 的 type 信息，压入无 type 属性的 S 符号占位；

如果使用其他产生式规约，若产生式中拥有运算符号则检验运算式子的 type 信息是否符合，若符合则将对应的 type 赋给即将压入符号栈的产生式头部，否则直接压入无 type 属性的产生式头部符号占位。

自底向上的分析过程中 Accept 时，检查栈中是否只有一个元素。

中间代码生成：

中间代码生成的结果是生成中间代码序列 List<Instruction>，根据翻译方案，中间代码栈需要保存 val 属性；

与中间代码生成相关的产生式有 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；

自底向上的分析过程中 Shift 时将带有 val 属性(从 Token 中获得)的符号入栈；

自底向上的分析过程中 Reduce 时，先记录并出栈产生式右部符号

如果规约时使用的产生式是 S -> id = E,在中间代码序列中加入 Instruction:createMov(named(id.getText),E.getVal())，压入无 val 属性的 S 符号占位

如果规约时使用的产生式是 S -> return E,在中间代码序列中加入 createRet(E.getVal())，压入无 val 属性的 S 符号占位

如果规约时使用的产生式是 E -> E + A,temp=temp()，在中间代码序列中加入 createAdd(temp,E.getVal(),A.getVal())，压入 temp 属性的 E 符号占位

如果规约时使用的产生式是 E -> E - A,temp=temp()，在中间代码序列中加入 createSub(temp,E.getVal(),A.getVal())，压入 temp 属性的 E 符号占位

如果规约时使用的产生式是 E -> A,压入 val 属性为 A 的 val 属性的 E 符号占位

如果规约时使用的产生式是 A -> A * B,temp=temp()，在中间代码序列中加入 createMul(temp,A.getVal(),B.getVal())，压入 temp 属性的 A 符号占位

如果规约时使用的产生式是 A -> B,压入 val 属性为 B 的 val 属性的 A 符号占位

如果规约时使用的产生式是 B -> (E),压入 val 属性为 E 的 val 属性的 B 符号占位

如果规约时使用的产生式是 B -> id,压入 val 属性为 id 的 val 属性的 B 符号占位

如果规约时使用的产生式是 B -> IntConst,压入 val 属性为 IntConst 的 val 属性的 B 符号占位

如果使用其他产生式规约，直接压入无 val 属性的产生式头部符号占位。

自底向上的分析过程中 Accept 时，检查栈中是否只有一个元素。

3.3.2 语义分析和中间代码生成使用的数据结构

语义分析用了栈的数据结构，中间代码生成用了栈的数据结构

3.3.3 主要流程描述（两个观察者的实现）

语义分析：

初始化：

创建类内类 TermThis extends Term 继承父类 Term 属性和方法的同时拥有额外私有的 String text 和 SourceCode type 成员属性以及他们的公共设值和取值方法
创建私有的 Symboltable 来存储符号表
创建并初始化私有的符号栈 Stack<TermThis>来存取带有 type 属性的符号

观察到 shift 的反应：
定义一个用接收到的单词 token 名字创建的 TermThis 实例
若单词 token 类别码 code 为 Int 则将该实例的 type 设置为 Int
若单词 token 类别码 code 为 id 则将该实例的 text 设置为 token 的 text
若单词 token 类别码 code 为 IntConst 则将该实例的 type 设置为 Int
将该实例加入到符号栈

观察到 reduce 的反应：
先记录并出栈当前产生式右部数量的符号，再利用记录下来的符号的属性以及不同规约产生式做出不同反应(在翻译方案中有具体对应实施策略)，最后根据不同规约产生式压入带有不同属性的产生式左部符号。

观察到 accept 的反应：
检测符号栈是否只有一个符号

中间代码生成：

初始化：
创建类内类 TermThis extends Term 继承父类 Term 属性和方法的同时拥有额外私有的 String text 和 IRValue val 成员属性以及他们的公共设值和取值方法
创建私有的 Symboltable 来存储符号表
创建并初始化私有的符号栈 Stack<TermThis>来存取带有 val 属性的符号
创建并初始化私有的指令列表 List<Instruction>来存取中间代码生成指令 Instruction

观察到 shift 的反应：
定义一个用接收到的单词 token 名字创建的 TermThis 实例
若单词 token 类别码 code 为 id 则将该实例的 text 设置为单词 token 的 text，将该实例的 val 设置为单词 token 的 val
若单词 token 类别码 code 为 IntConst 则将该实例的 val 设置为单词 token 的 val
将该实例加入到符号栈

观察到 reduce 的反应：
先记录并出栈当前产生式右部数量的符号，再利用记录下来的符号的属性以及不同规约产生式做出不同反应(在翻译方案中有具体对应实施策略)，最后根据不同规约产生式压入带有不同属性的产生式左部符号。

观察到 accept 的反应：
检测符号栈是否只有一个符号

3.4 目标代码生成

3.4.1 主要流程描述

数据结构：

Reg 寄存器类：

private final int num 成员：表示该寄存器号，在寄存器建立时利用参数定义

private IRValue val 成员：表示该寄存器存放的变量

公共创建方法：public Reg(int num){ this.num = num; }

重写了 toString 方法：返回 String：t + num

拥有对 num 属性的 get 方法和 val 属性的 get 和 set 方法

判断是否空闲方法(该寄存器是否保存变量)：public boolean isSpare(){ return val == null; }

判断是否活跃方法(该寄存器保存的变量是否仍需使用，利用先前记录完毕的变量使用次数哈希表)：public boolean isActive(){ return val != null && usedTimeMap.get(val) != 0; }

BMap<K, V>双向映射的哈希表：

包含两个 KEY 和 VALUE 对立的哈希表，哈希表的添加和删除操作都是双向的，也能分别根据 KEY 和 VALUE 进行查找

记录当前变量和寄存器对应关系的双射表

BMap<IRValue,Reg> regMap

记录当前变量和内存位置对应关系的双射表

BMap<IRValue,Integer> locationMap

记录变量使用次数的哈希表

HashMap<IRValue,Integer> usedTimeMap

存放当前寄存器的栈(用队列表示)

Deque<Reg> regDeque

1.预处理：

为什么要进行预处理：

在 RISC-V 的汇编中，对于整数加法，我们考虑 add 和 addi 指令格式，其要求加法指令只能有最多一个立即数，且立即数需要放在右手处，即 addi rs, rd, imm. 然而在 IR 中并没有这个限制(且这个限制在前面的实验中就不应该存在)，这会为我们在生成目标代码时带来繁琐的判断和指令插入. 通过预处理可以简化目标代码的生成。

预处理思路：

对于 BinaryOp(两个操作数的指令):

将操作两个立即数的 BinaryOp 直接进行求值得到结果，然后替换成 MOV 指令

将操作一个立即数的指令 (除了乘法和左立即数减法) 进行调整，使之满足 a := b op imm 的格式

将操作一个立即数的乘法和左立即数减法调整，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。

对于 UnaryOp(一个操作数的指令):

根据语言规定，当遇到 Ret 指令后直接舍弃后续指令。

在原来 instrution 列表的基础上进行预处理，节省空间提高性能。

2.计算每个变量的使用次数：

用数据结构哈希表 HashMap<IRValue,Integer> usedTimeMap 进行存储，对预处理后的 instrucion 列表遍历，将出现在每个 instrucion 右部的 LHS、RHS、FROM 变量存入哈希表 usedTimeMap 中并存储对应的使用次数(每个变量每次出现在 instrucion 右部则认作使用一次)，得到记录完毕的变量使用次数哈希表。

3.初始化寄存器队列：

寄存器队列：Deque<Reg> regDeque

初始化，向队列中依次加入寄存器号为 0-6 的寄存器(regNumMax=7)，则 0 号寄存器处于队头，6 号寄存器处于队尾

与寄存器选择算法配合使用

4.寄存器选择算法：public Reg getReg(IRValue val,int regLeft, int regRight)

第一个参数为要存入寄存器的变量，第二个参数为第一个不能抢占的寄存器号，第三个参数为第二个不能抢占的寄存器号

1.寻找是否有寄存器保存当前变量：

在记录当前变量和寄存器对应关系的双射表 BMap<IRValue,Reg> regMap 中利用 containsKey 判断当前变量 IRValue val 是否有寄存器保存，若有则将该寄存器从寄存器队列 Deque<Reg> regDeque 中弹出，再加入寄存器队列队尾(表示其最近使用频率最高)，返回该寄存器。

2.按照使用频率在寄存器队列中寻找空闲寄存器

遍历寄存器队列 Deque<Reg> regDeque(由于队列数据结构的特殊性，从队头遍历至队尾，即是按照寄存器使用频率低到高顺序遍历)，若寻找到空闲的寄存器(判断寄存器空闲的方法)，则将得到的空闲寄存器弹出队列，将该空闲寄存器的变量改为传入的变量 IRValue val，再加入寄存器队列队尾(表示其最近使用频率最高)，判断该变量是否保存于内存栈中

(利用记录当前变量和内存位置对应关系的双射表 BMap<IRValue,Integer> locationMap)，若是则在目标生成代码中加入 lw(“lw reg, 内存位置(sp)”)目标代码来得到保存在内存的变量，返回该寄存器。

3.按照使用频率在寄存器队列中寻找不活跃寄存器：

遍历寄存器队列 Deque<Reg> regDeque(由于队列数据结构的特殊性，从队头遍历至队尾，即是按照寄存器使用频率低到高顺序遍历)，若寻找到不活跃的寄存器(判断寄存器不活跃的方法)，则将得到的不活跃寄存器弹出队列，将该不活跃寄存器的变量改为传入的变量 IRValue val，再加入寄存器队列队尾(表示其最近使用频率最高)，判断该变量是否保存于内存栈中

(利用记录当前变量和内存位置对应关系的双射表 BMap<IRValue,Integer> locationMap)，若是则在目标生成代码中加入 lw(“lw reg, 内存位置(sp)”)目标代码来得到保存在内存的变量，返回该寄存器。

4.按照使用频率在寄存器队列中寻找被抢夺的寄存器：

遍历寄存器队列 Deque<Reg> regDeque(由于队列数据结构的特殊性，从队头遍历至队尾，即是按照寄存器使用频率低到高顺序遍历)，若得到的寄存器号与两个参数的寄存器号不相等，则将得到的寄存器弹出队列，计算有无空闲内存

(利用记录当前变量和内存位置对应关系的双射表 BMap<IRValue,Integer> locationMap 和记录变量使用次数的哈希表 HashMap<IRValue,Integer> usedTimeMap，从低到高遍历内存栈，若得到的变量需要使用次数为 0 则将被抢夺寄存器原本保存的变量存入该内存位置，否则将内存指针+4，将被抢夺寄存器原本保存的变量存入内存栈栈顶)

在目标生成代码中加入 sw(“sw reg, 内存位置(sp)”)目标代码来保存被抢夺寄存器原本保存的变量

将该寄存器的变量改为传入的变量 IRValue val，再加入寄存器队列队尾(表示其最近使用频率最高)，判断该变量是否保存于内存栈中(利用记录当前变量和内存位置对应关系的双射表 BMap<IRValue,Integer> locationMap)，若是则在目标生成代码中加入 lw(“lw reg, 内存位置(sp)”)目标代码来得到保存在内存的变量，返回该寄存器。

5.目标代码生成：

遍历预处理后的 Instruction 列表，对不同的 InstructionKind 有不同的应对措施：

RET：

如果返回值 ReturnValue 是立即数，则加入 li(“li a0, ReturnValue”)目标代码

如果返回值 ReturnValue 是变量，则加入 mv(“mv a0, getReg(ReturnValue,regMaxNum,regMaxNum)”)目标代码，利用寄存器选择算法，后面两个参数为 regMaxNum=7 表示不需要考虑不能被夺取的寄存去

MOV：

如果赋值 From 是立即数，则加入 li(“li getReg(From,regMaxNum,regMaxNum), From”)目标代码

如果赋值 From 是变量，则加入

mv(“mv getReg(Result,From,regMaxNum),
getReg(From,regMaxNum,regMaxNum)”)目标代码
……

4 实验结果与分析

对实验的输入输出结果进行展示与分析。注意：要求给出编译器各阶段（词法分析、语法分析、中间代码生成、目标代码生成）的输入输出并进行分析说明。

input_code.txt

词法分析

实现词法分析前的缓冲区、编写代码实现自动机进行词法分析的过程、以适当的数据结构保存词法分析的到的 token 列表

生成符号表 old_symbol_table.txt

将缓冲区前的字符串按照语法状态机划分成不同的单词，将不存在于关键词表的单词存入符号表 symbolTable 中

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

生成单元单词列表 token.txt

将缓冲区前的字符串按照语法状态机划分成不同的单词，根据状态机不同终止状态存入不同属性和类别的单词到单词列表 tokenList 中

```
(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,)
($,)
```

语法分析

实现实验一得到的词法单元列表的加载和存放、实现 LR 分析表的加载和使用方法、实现语法分析的驱动程序

生成产生式序列 parser_list.txt

利用 LR(1)分析法，设计语法分析程序，结合编译工作台得到的 LR(1)文法文件 LR1_table.csv 对输入单词符号串(实验一的输出 token.txt)进行语法分析，输出推导过程中所用产生式序列

```
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 -> 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
```

中间代码生成

设计文法的翻译方案、设计各个观察者需要的用于语义分析和中间代码生成的数据结构（栈）、实现各观察者在分析过程中的动作处理、使用 IREmulator 对生成的中间代码进行验证。

生成更新的符号表 new_symbol_table.txt

设置观察者，对实验二语法翻译的每次转移、规约、接受动作进行相应的反应动作，检验运算的正确性(例如需要都为数字型类别才能进行计算或者进行数字类别的转换，本次实验中只有 int 类型，因此只做了需要都为 int 类型才能进行运算的检验，否则报错)，对与声明语句有关的产生式则需要更新符号表

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

生成中间代码 ir_emulate_result.txt

设置观察者，对实验二语法翻译的每次转移、规约、接受动作进行相应的反应动作，对与赋值语句、运算语句和返回语句有关的产生式则需要向中间代

码列表中加入对应的中间代码

```
(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)
```

使用 IREmulator 对生成的中间代码进行验证 ir_emulate_result.txt

利用 IREmulator 对生成的中间代码进行计算验证得到的返回值结果

```
144
```

目标代码生成

加载前端提供的中间代码，视情况做预处理（预处理思路参考指导书），实现寄存器选择算法、实现目标代码生成算法、输出生成的目标代码到指定文件 assembly_language.asm 中、使用 Rars 运行目标代码，验证其正确性

预处理：将中间代码转化成适合 RISC-V 的中间代码(RET 中间代码之后的代码不需要执行、ADD 中间代码中产生式的 LHS 和 RHS 都可以是立即数，而 RISC-V 指令中只有 RHS 可以是立即数,MUL 和 SUB 中间代码 LHS 和 RHS 都不能是立即数)

寄存器选择算法：先判断当前变量是否被寄存器栈中的寄存器保存，
若有则直接使用保存其值的寄存器，
否则在空闲寄存器中寻找一个寄存器保存当前变量，
若找不到空闲寄存器则在不活跃寄存器中寻找一个寄存器保存当前变量，
若找不到不活跃寄存器则需要抢夺寄存器，抢夺寄存器顺序为按照使用频率从低到高，
被抢夺的寄存器保存的变量需要保存到内存栈中，添加 sw 指令
(判断内存栈是否有不再使用的变量，若有则将变量保存在该位置，否则需要开辟 4 个字节的内存空间(由于只有 int 整型，用一个全局变量记录需要开辟的地址空间，在目标代码生成最后一步加载到目标代码开头))
最后检测变量是否存在内存栈中，若是则需要添加 lw 指令，返回得到的寄存器

目标代码生成：初始化 t0-t6 寄存器栈，先记录每个变量需要被使用的次数(每次作为 LHS 和 RHS 出现则代表使用一次)在目标代码生成时可以用来判断

该变量是否还需要再次使用及寄存器是否活跃等条件的判断依据，根据不同的指令利用寄存器选择算法生成不同的目标代码(其中 sub 指令若有产生式右部立即数需要转化为 addi)，最后若需要使用内存栈，将内存栈初始化目标代码加入目标代码开头，将内存栈还原目标代码加入目标代码返回值之前。

```
.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 t4, t3       #  (MOV, c, $0)
    mul t5, t0, t1        #  (MUL, $1, a, b)
    addi t6, t1, 3        #  (ADD, $2, b, 3)
    sub t2, t4, t0        #  (SUB, $3, c, a)
    mul t3, t6, t2        #  (MUL, $4, $2, $3)
    sub t1, t5, t3        #  (SUB, $5, $1, $4)
    mv t4, t1       #  (MOV, result, $5)
    mv a0, t4       #  (RET, , result)
```

在 RARS 上得到 a0 保存的值为 90(144)，正确

a0	10	0x0000000000000090

reg-alloc.txt
得到的目标代码

```
.text addi
sp, sp, -60
li t0, 0        #  (MOV, f0, 0)
li t1, 1        #  (MOV, f1, 1)
add t2, t1, t0        #  (ADD, $0, f1, f0)
mv t3, t2       #  (MOV, f2, $0)
add t4, t3, t1        #  (ADD, $1, f2, f1)
mv t5, t4       #  (MOV, f3, $1)
add t6, t5, t3        #  (ADD, $2, f3, f2)
mv t2, t6       #  (MOV, f4, $2)
add t4, t2, t5        #  (ADD, $3, f4, f3)
mv t6, t4       #  (MOV, f5, $3)
add t4, t6, t2        #  (ADD, $4, f5, f4)
sw  t0, 0(sp)
mv t0, t4       #  (MOV, f6, $4)
add t4, t0, t6        #  (ADD, $5, f6, f5)
sw  t1, 4(sp)
mv t1, t4       #  (MOV, f7, $5)
add t4, t1, t0        #  (ADD, $6, f7, f6)
sw  t3, 8(sp)
```

```
mv t3, t4        #  (MOV, f8, $6)
add t4, t3, t1       #  (ADD, $7, f8, f7)
sw  t5, 12(sp)
mv t5, t4        #  (MOV, f9, $7)
add t4, t5, t3       #  (ADD, $8, f9, f8)
sw  t2, 16(sp)
mv t2, t4        #  (MOV, f10, $8)
add t4, t2, t5       #  (ADD, $9, f10, f9)
sw  t6, 20(sp)
mv t6, t4        #  (MOV, f11, $9)
add t4, t6, t2       #  (ADD, $10, f11, f10)
sw  t0, 24(sp)
mv t0, t4        #  (MOV, f12, $10)
add t4, t0, t6       #  (ADD, $11, f12, f11)
sw  t1, 28(sp)
mv t1, t4        #  (MOV, f13, $11)
add t4, t1, t0       #  (ADD, $12, f13, f12)
sw  t3, 32(sp)
mv t3, t4        #  (MOV, f14, $12)
add t4, t3, t1       #  (ADD, $13, f14, f13)
sw  t5, 36(sp)
mv t5, t4        #  (MOV, f15, $13)
add t4, t5, t3       #  (ADD, $14, f15, f14)
sw  t2, 40(sp)
mv t2, t4        #  (MOV, f16, $14)
add t4, t2, t5       #  (ADD, $15, f16, f15)
sw  t6, 44(sp)
mv t6, t4        #  (MOV, f17, $15)
add t4, t6, t2       #  (ADD, $16, f17, f16)
sw  t0, 48(sp)
mv t0, t4        #  (MOV, f18, $16)
add t4, t0, t6       #  (ADD, $17, f18, f17)
sw  t1, 52(sp)
mv t1, t4        #  (MOV, f19, $17)
lw  t4, 0(sp)
sw  t3, 56(sp)
mv t3, t4        #  (MOV, s0, f0)
lw  t4, 4(sp)
sw  t5, 0(sp)
add t5, t3, t4       #  (ADD, $18, s0, f1)
mv t3, t5        #  (MOV, s1, $18)
lw  t4, 8(sp)
add t5, t3, t4       #  (ADD, $19, s1, f2)
mv t3, t5        #  (MOV, s2, $19)
```

```
lw  t4, 12(sp)
add t5, t3, t4        #  (ADD, $20, s2, f3)
mv t3, t5         #  (MOV, s3, $20)
lw  t4, 16(sp)
add t5, t3, t4        #  (ADD, $21, s3, f4)
mv t3, t5         #  (MOV, s4, $21)
lw  t4, 20(sp)
add t5, t3, t4        #  (ADD, $22, s4, f5)
mv t3, t5         #  (MOV, s5, $22)
lw  t4, 24(sp)
add t5, t3, t4        #  (ADD, $23, s5, f6)
mv t3, t5         #  (MOV, s6, $23)
lw  t4, 28(sp)
add t5, t3, t4        #  (ADD, $24, s6, f7)
mv t3, t5         #  (MOV, s7, $24)
lw  t4, 32(sp)
add t5, t3, t4        #  (ADD, $25, s7, f8)
mv t3, t5         #  (MOV, s8, $25)
lw  t4, 36(sp)
add t5, t3, t4        #  (ADD, $26, s8, f9)
mv t3, t5         #  (MOV, s9, $26)
lw  t4, 40(sp)
add t5, t3, t4        #  (ADD, $27, s9, f10)
mv t3, t5         #  (MOV, s10, $27)
lw  t4, 44(sp)
add t5, t3, t4        #  (ADD, $28, s10, f11)
mv t3, t5         #  (MOV, s11, $28)
lw  t4, 48(sp)
add t5, t3, t4        #  (ADD, $29, s11, f12)
mv t3, t5         #  (MOV, s12, $29)
lw  t4, 52(sp)
add t5, t3, t4        #  (ADD, $30, s12, f13)
mv t3, t5         #  (MOV, s13, $30)
lw  t4, 56(sp)
add t5, t3, t4        #  (ADD, $31, s13, f14)
mv t3, t5         #  (MOV, s14, $31)
lw  t4, 0(sp)
add t5, t3, t4        #  (ADD, $32, s14, f15)
mv t3, t5         #  (MOV, s15, $32)
add t4, t3, t2        #  (ADD, $33, s15, f16)
mv t5, t4         #  (MOV, s16, $33)
add t3, t5, t6        #  (ADD, $34, s16, f17)
mv t2, t3         #  (MOV, s17, $34)
add t4, t2, t0        #  (ADD, $35, s17, f18)
```

```
mv t5, t4          #  (MOV, s18, $35)
add t6, t5, t1          #  (ADD, $36, s18, f19)
mv t3, t6          #  (MOV, s19, $36)
addi sp, sp, 60
mv a0, t3          #  (RET, , s19)
```

在 RARS 上得到 a0 保存的值为 2ac1(10945)，正确

a0	10	0x0000000000002ac1

5 实验中遇到的困难与解决办法

描述实验中遇到的困难与解决办法，对实验的意见与建议或收获。

困难与解决方法：

在实验四中，对于寄存器选择算法需要使用的数据结构和实现以及寄存器类 Reg 的成员属性和成员方法的考虑引发了我较多的思考，

首先对于寄存器类别，需要拥有它存储的变量的成员属性，要拥有判断寄存器空闲、活跃的方法

寄存器选择算法需要先判断变量是否有寄存器已经存放了该变量，加入了<寄存器，变量>双射哈希表来存取寄存器和变量对应关系

判断寄存器空闲可以利用寄存器的变量成员属性是否为空，

判断寄存器活跃就需要先遍历指令列表来提前得知每个变量的使用情况，因此加入了<变量，使用次数>哈希表来存取每个变量对应的使用情况。

夺取寄存器需要根据每个寄存器的使用频率来进行选择且不能夺取一个指令内其他变量使用的寄存器，为此寄存器选择算法需要加入两个寄存器号参数来避免指令内冲突，寄存器使用频率则利用队列的特性进行实现，将 t0-t6 寄存器加入寄存器队列<寄存器>队列，每次访问一个寄存器则将该寄存器放入寄存器队列队尾，夺取寄存器时从队头开始寻找，保证了寄存器按照使用频率夺取

夺取寄存器还需要将寄存器本来存放的变量存入内存栈，将内存栈变量存入寄存去，不仅需要根据内存栈位置找到变量，而且需要根据变量找到相对应内存栈存放位置，因此加入一个<变量，位置>双射哈希表来存取变量和位置的对应关系

意见与建议：

编译原理实验为学生提供了许多先前搭建好的框架和文法以及输入输出，许多好的框架让学生的精力可以完全投放在实验的完成上，例如一些输

入输出文件、一些预先写好的类和关系等、例如 LR(1)文法的编译工作台便于学生直接使用生成好的 LR(1)文法，但 LR(1)文法的分析表生成对于学生是十分重要的能力，希望可以尝试实现 LR(1)分析表的构建，并生成一些对应的检验方法来检测学生生成的分析表的正确性。

收获：

通过这次实验，我对于编译原理有了更加完善的认知，通过代码编程的实现方法更是让我对编译的各个过程有了充足的印象，让我深刻的意识到为什么编译原理的一整个过程需要词法分析、语法分析、预处理等等，若是需要编译的语言系统更加庞大和复杂，一个完善、周到的编译器定然需要分成多个精细化的模块对源语言进行编译实现，让我深刻意识到了编译过程对于精细的要求。