本科生实验报告

实验课程: 编译原理

实验名称: 编译器实验

专业名称: 计科（人工智能与大数据方向）

学生姓名: 吕文禧

学生学号: 19335151

实验地点: D503

实验成绩:

报告时间: 2022/7/2

实验目的

> - **通过编译器相关子系统的设计，进一步加深对编译器构造的理解；**
> - **!培养学生独立分析问题、解决问题的能力，以及系统软件设计的能力；**
> - **提高程序设计能力、程序调试能力**

实验要求

> 编译器构造实验
>
> - 一个简单文法的编译器的设计与实现
>   - 一个简单文法的编译器前段的设计与实现
>   - 定义一个简单程序设计语言文法（包括变量说明语句、算术运算表达式、赋值语句；扩展包括逻辑运算表达式、If语句、While语句等）；
>   - 扫描器设计实现；
>   - 符号表系统的设计实现；
>   - 语法分析器设计实现；
>   - 中间代码设计；
>   - 中间代码生成器设计实现。
>   - 一个简单文法的编译器后段的设计与实现
>   - 中间代码的优化设计与实现（鼓励）；
>   - 目标代码的生成（使用汇编语言描述，指令集自选）；
>   - 目标代码的成功运行

实验设计

以下部分展示本人实现的编译器实现的设计思路，框架，所使用到的数据结构，以及一部分关键代码的分析。.

前端

符号系统的设计

由于实现的编译器支持语言的语法功能不多（变量说明语句、算术运算表达式、赋值语句；扩展包括逻辑运算表达式、If语句），暂时不支持数组、结构体、函数的声明与实现。所以符号表`ST`设计用于记录变量的名字、种类、类型（变量还是函数，此处都是变量）、活跃信息、地址信息。一张常数表`CT`，用于记录程序中出现的常数信息。一张长度表`type_len`，用于记录各种数据类型在内存地址中所占空间。

具体如何填入符号表则会在词法分析设计部分与语义分析设计中详细分析。

使用自定义的结构体`table_item`作为表项元素，记录变量的名字、种类、类型、活跃信息、地址信息。使用哈希表`unordered_map`建立变量名字与表项的映射，所以变量名字即为token的指针。

具体代码实现见`symbol_table.h`文件。

词法分析设计

由于在第一次实验中就实现了C语言的词法分析器。本次实验在此基础上修改，加入了符号表的相关填入步骤。

- 首先构造C语言中关键词表，建立对应token类别码的映射。
{"break","4"}	**{"case","5"}**	**{"char","6"}**	**{"const","7"}**	**{"continue","8"}**
,{"default","9"},	{"do","10"}	{"double","11"}	{"else","12"}	{"enum","13"}
{"extern","14"}	{"float","15"},	{"for","16"}	{"goto","17"}	{"if","18"}
{"int","19"}	{"long","20"}	{"register","21"}	{"return","22"}	{"short","23"}
{"signed","24"}	{"sizeof","25"}	{"static","26"}	{"struct","27"}	{"switch","28"}
{"typedef","29"},	{"union","30"}	{"unsigned","31"}	{"void","32"}	{"volatile","33"}
{"while","34"}				
- 构造C语言的界符表，此处考虑更多的运算符以及界符，将`++` `--` `+=`等纳入词表中，具体词表如下：
{">=","35"}	**{"<=","36"}**	**{"==","37"}**	**{"!=","38"}**	**{"=","39"}**
{">","40"}	{"<","41"}	{"%","42"}	{"+","43"}	{"+=","44"}
{"++","45"}	{"-","46"}	{"-=","47"}	{"--","48"}	{"*","49"}
{"*=","50"}	{"/","51"}	{"/=","52"}	{"(","53"}	{")","54"}
{"{","55"}	{"}","56"}	{",","57"}	{";","58"}	{"[","59"}
{"]","60"}	{"|“,”61"}	{"&","62"}	{"^","63"}	{"!","64"}
{"<<","65"}	{">>","66"}	{"->","67"}	{".","68"}	{"#","69"}
{"||","70"}	{"&&","71"}			

> 在预先建立好上述内部表后，词法分析器就能将关键词与标识符区分开来，同时可以自动识别界符并赋予token，便于后续语法分析的操作使用。

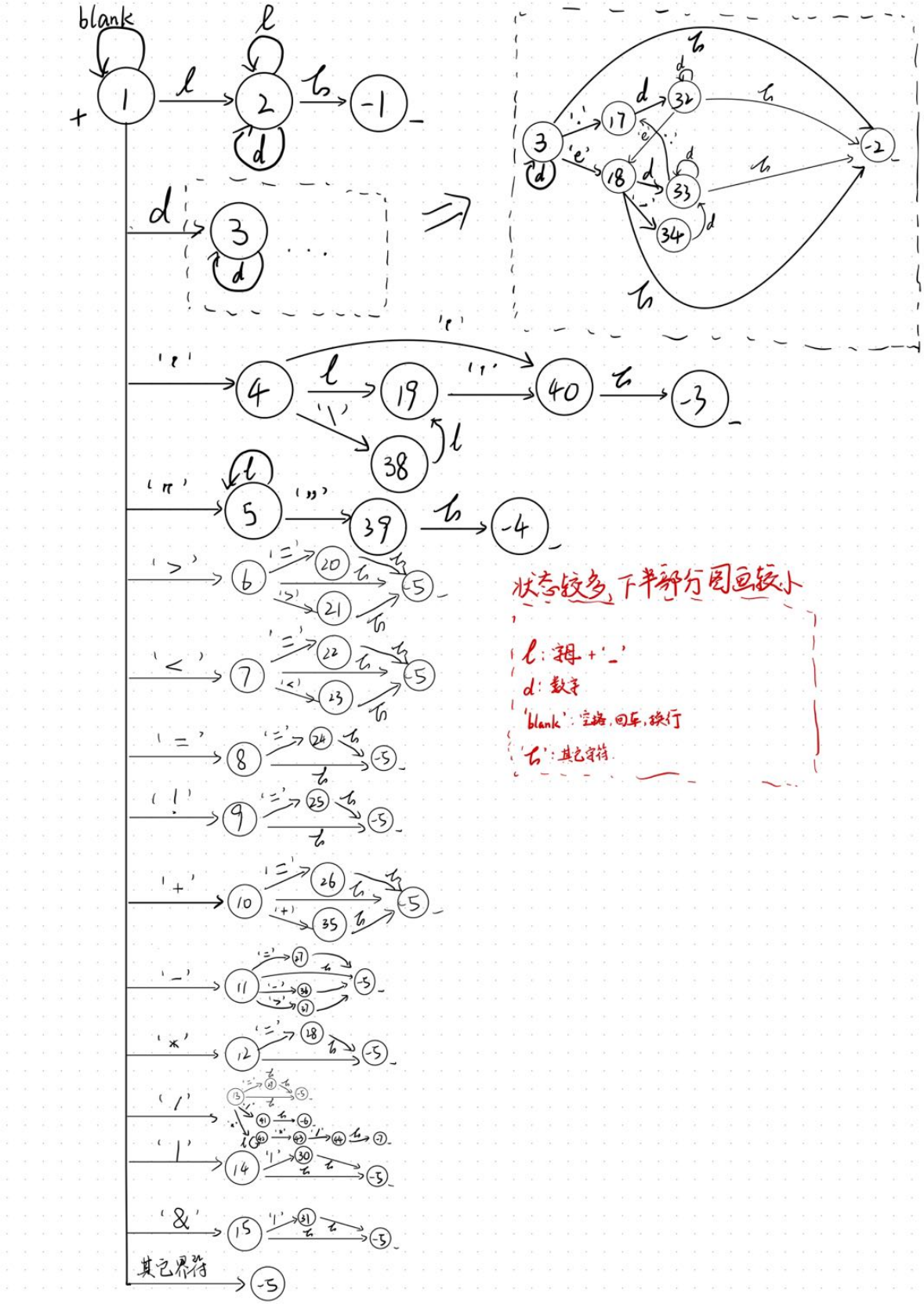
词法自动机设计

词法扫描器的DFA与第一次实验的DFA一致。

为了可以识别不同界符、区分所有标识符、常数变量、字符、字符串，设计DFA如下，在起始状态遇到不同字符时转变为不同状态。为了最终可以区分识别，我将终态设计为6个不同的值，0为异常，即遇到非法词如10. 、10.a$、‘da’等不符合C语言词法的时返回0状态（在图中未画出，异常）；-1为标识符或关键字；-2为常数；-3为字符；-4为字符串；-5为界符；-6为行末注释；-7为行内注释。

DFA总共有51种状态，可以识别绝大多数C语言中会出现的符号，全部合法的标识符，除auto以外的全部关键字，用科学记数法的常数或是普通常数以及字符与字符串。同时还可以做到简单识别一些不合C语言词法的代码，具有一定的可用性。

还需要考虑代码中存在注释的情况，对于/**/注释可能跨行，也可能在行内，也可能跨行，而//注释较为简单，其只能是行末且不能跨行。但识别到他们后都不需要对注释生成token_code. 得到终态后对应选择忽略即可。



token生成器

根据DFA最后返回的非正状态，可以实现分辨token对应的类别码，从而实现token类别码生成器：”-1“为异常，”00“为标识符，”01“为字符，”02“为字符串，”03“为常数，其他对应内部表中所映射的类别码。

token生成器与第一次实验不同的地方：

- token为< 类别码，源码 >
- 需要加入符号表的填入工作

- 当遇到token类别码为00的token时，识别为变量，需要初始化一个表项。但此时编译器只能获知出现了一个变量，不能确定其种类，类型以及地址信息。所以表项信息记录为改变量名称，地址置为-1，其他均为空。
- 当遇到token类别码为03的token时，识别为常数，需要填入常数表，直接填入即可，不需额外记录信息。

词法生成的具体代码见`scanner.h`文件。

文法设计

本次实验中我设计的编译器可以支持的语言功能有：变量说明语句、算术运算表达式、赋值语句；扩展包括逻辑运算表达式、If语句

所以设计文法如下：program符号作为开始符号，并且全部消除左递归，符合LL1文法。

program -> 变量声明 (1) | 语句 (2)

变量声明-> type id 变量声明‘ (3)

type -> void (4)| int (5)| char (6) | float (7)

变量声明‘-> ',' id 变量声明‘ (8)| ε (9)

语句-> A 语句’ (10)| if_语句 (11)

语句’-> '=' A 语句’ (12) | ε (13)

A -> EA' (14)

A‘ -> E A' (15) | ε (16)

$$E \to TE_1 \quad (17)$$
$$E_1 \to \omega_0 TE_1 \quad (18) | \ \varepsilon \ (19)$$
$$T \to FT_1 \quad (20)$$
$$T_1 \to \omega_1 FT_1 \quad (21) | \ \varepsilon \ (22)$$
$$F \to I \quad (23) | \ \ C \ (24) | \ (\text{语句}) \ (25)$$

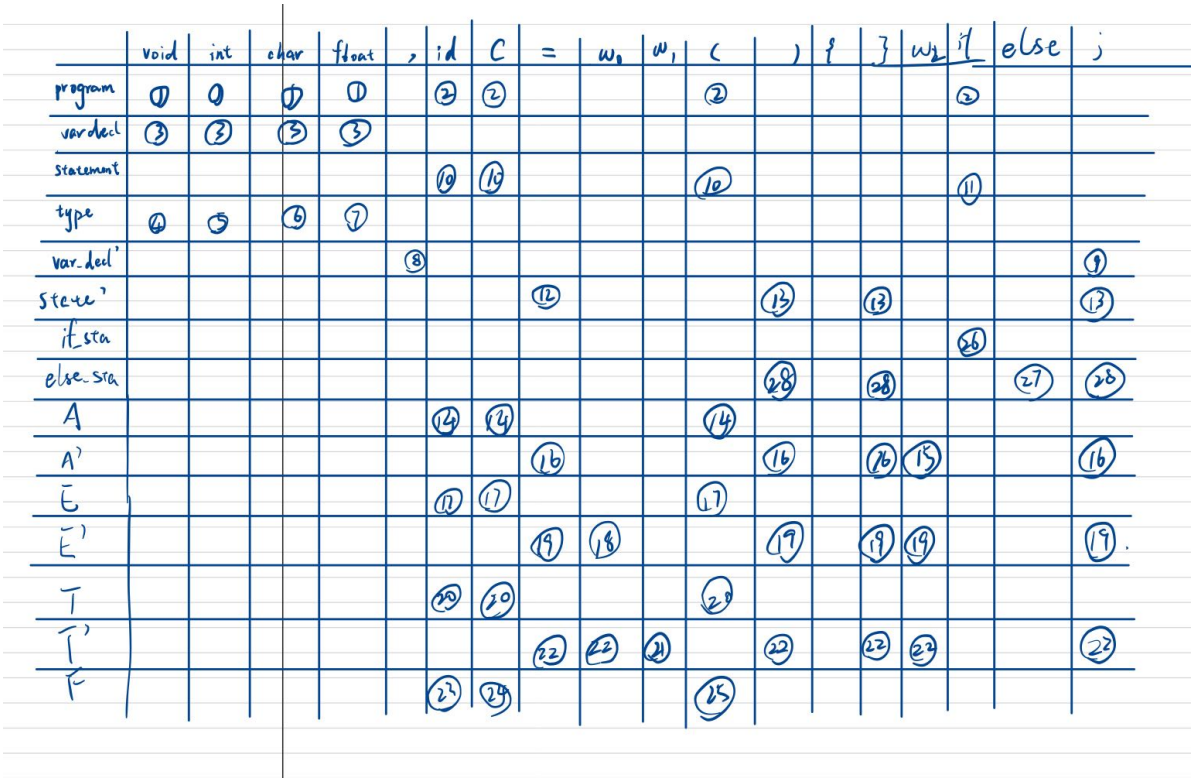
if_语句 -> if '(' A ')' ‘{’ program '}' else语句 (26)

else语句-> 'else' ‘{’ program '}' (27)| ε (28)

由于以上的文法的定义，本编译器要求if语句内容不能为空，而且if语句中嵌套的if语句需要在大括号后接`;`作为终结符（if视为一个独立的program），if后接else语句也需要加`;`作为终结符（if else语句视为一个语句），本文法支持if与else的内部嵌套语法，如：if(a > b) {if(a > b) {};} 。

LL1分析表

通过分析每个产生式的select集获得LL1分析表如下：

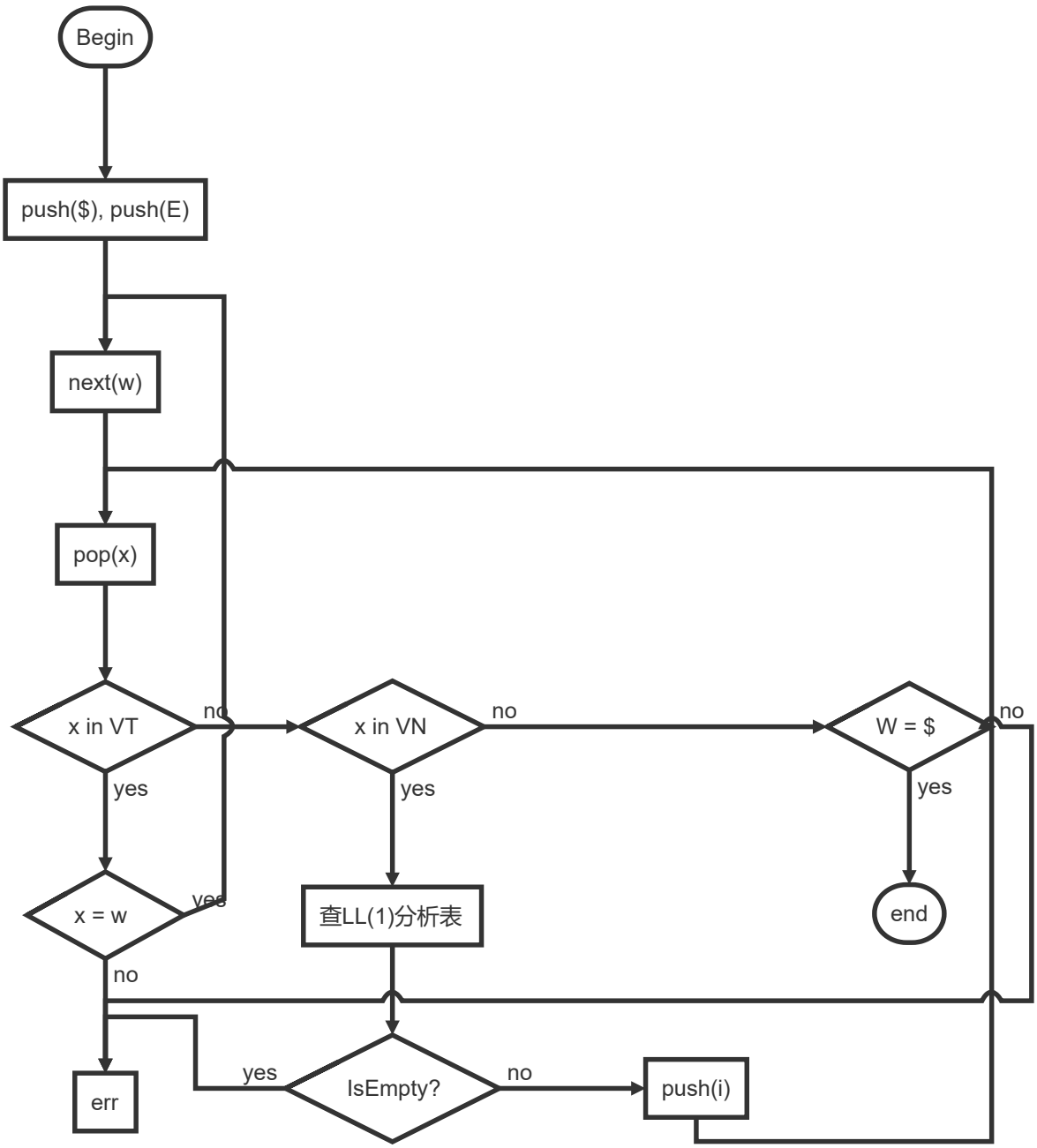


语法分析逻辑

本次实验语法分析器只是文法拓展了更多功能，产生了更多状态，分析逻辑与实验2中语法分析器的代码逻辑实现一致：

对于LL1的语法分析器，需要一个分析栈一张分析表，一开始向栈内压入#program作为起始状态。开始时读入一个token，检查栈顶元素，若栈顶元素为非终结符Vn，则从LL1分析表中查找对应产生式的右部，若存在则逆序压入栈中，继续查看下一个栈顶元素而不移进字符，若该表项为空则返回错误信息；若栈顶元素为终结符Vt，则与当前移进的字符检查匹配，若不匹配则返回错误信息，若匹配则继续移进字符。重复上述步骤直至出现错误或者匹配到终结符`#`。

语法分析器的程序逻辑如下：



具体代码实现见`parser.h`文件。

语义分析设计

本次实验语义分析器逻辑与实验3中语义分析器的分析代码逻辑实现一致，具体动作符号代表操作稍有修改，沿用的语义分析器逻辑如下：

要实现语义分析器，只需要基于语法分析器拓展，将表达式文法改写为翻译制导文法，加入动作符号指明在文法中执行语义翻译动作的时机以及需要执行的对应动作，同时使用一个语义栈辅助产生四元式。

在语法分析器中，将同优先级的运算符合并为一个集合（如：ω0指代 + - ...，ω1指代 / * , ω2指代逻辑运算 ...）。而语法分析器则需要对每个运算符指定不同的四元式产生操作（如：GEQ{+}, GEQ{-}, GEQ{*}...），如何在不增加产生式的前提下，即保留原有的语法分析器实现，又能够对不同的运算符产生不同的产生式呢？其实很简单，只要可以把运算符记录下来，在遇到动作符号时，将对应的运算符取

出即可。

那么另一个问题就来了，如何保证运算符的顺序与动作符号能够保持一致对应呢？而且，如何记录这些元素呢，可以观察到在遇到并执行动作符号之前，要被处理（如：压栈、产生四元式，填入符号表操作）的元素已经在匹配语法时已经被移进丢失了。

为了解决上述问题，我额外添加了一个日志栈log，用于在语法分析执行移进动作前记录元素（括号与终止符不记录）。为什么这个日志栈可以解决提到的问题，因为日志栈的压栈发生在移进操作之前，也就是语法分析最后一次逆序压栈，此时必然压入一个带有动作符号的产生式，同时被移进元素与产生式中的动作符号正好对应，所以既解决了记录元素的问题，使用栈的数据结构，与语法分析栈一致，保证操作顺序的一致。

中间语言设计

本次实验编译器选择四元式作为中间语言。

算术/逻辑运算四元式：（ω, b, c, a）

赋值表达式：（=, b, _, a ）

if表达式：（if, b, _, _ ）

el表达式：（el, _, _, _ ）

ifend表达式：（ie, _, _, _ ）

翻译文法的设计

为了可以正确获得四元式中间语言，设计翻译文法如下，通过插入动作符号的位置指明执行动作的时机，与LL1分析法一致保持逆序压栈，遇到动作符号不移进，执行对应动作。

其中变量声明时，可能出连续使用同种种类定义多个变量，因此需要一个变量名栈记录同种种类的变量，一次将相同种类的变量写入符号表。

program -> 变量声明 (1) | 语句 (2)

变量声明-> type {init()} id {name()} {ENT()} 变量声明‘ (3)

type -> void {typ(v)} (4)| int {typ(i)} (5)| char {typ(c)} (6) | float {typ(f)} (7)

变量声明‘-> ',' id {name()} {ENT()} 变量声明‘ (8)| ε (9)

语句-> A语句’ (10)| if_语句 (11)

语句’-> '=' A {assi(=)} 语句’ (12) | ε (13)

A -> EA' (14)

A‘ -> E {GEQ(w2)} A' (15) | ε (16)

$$E \to TE_1 \quad (17)$$
$$E_1 \to \omega_0 T\{GEQ(w0)\}E_1 \quad (18) | \ \varepsilon \ (19)$$
$$T \to FT_1 \quad (20)$$
$$T_1 \to \omega_1 F\{GEQ(w1)\}T_1 \quad (21) | \ \varepsilon \ (22)$$
$$F \to I\{push(i)\} \quad (23) | \quad C\{push(c)\} \ (24) | \ (\text{语句}) \ (25)$$

if_语句 -> if {IF(if)} '(' A ')' ‘{’ program '}' {IE(ie)} else语句 (26)

else语句-> 'else' {el(el)} '{' program '}' {IE(ie)}（27)| ε (28)

动作符号意义

动作符号	意义
{init()}	初始化变量，声明的变量活跃信息初始化为'y'，类型初始化为'v'
{name()}	记录变量名字（源码），压入变量名栈中
{ENT()}	填入变量名栈中变量名对应的符号表信息，同时查符号表有无发生重定义
{typ(v)}	将当前待填入符号表的变量种类设置为void
{typ(i)}	将当前待填入符号表的变量种类设置为int
{typ(c)}	将当前待填入符号表的变量种类设置为char
{typ(f)}	将当前待填入符号表的变量种类设置为float
{assi(=)}	语义栈pop, pop, 生成赋值四元式
{GEQ(w2)}	语义栈pop, pop，生成逻辑运算四元式，push结果，同时查符号表有无发生类型不匹配
{GEQ(w0)}	语义栈pop, pop，生成加减算术四元式，push结果，同时查符号表有无发生类型不匹配
{GEQ(w1)}	语义栈pop, pop，生成乘除算术四元式，push结果，同时查符号表有无发生类型不匹配
{push(i)}	语义栈压入变量
{push(c)}	语义栈压入常量
{IF(if)}	语义栈pop, 生成if四元式
{el(el)}	生成el四元式
{IE(ie)}	生成ie四元式

根据上述动作符号执行，可以产生正确四元式中间代码。

具体代码实现见`parser.h`文件。

后端

基本块划分设计

为了目标代码能够正确生成，必须对中间代码进行基本块的划分，否则if语句块中的中间代码可能会影响if语句块外的中间代码的活跃信息，导致出现寄存器分配时发生错误的强制释放而数据丢失的情况。

基本块划分的算法如下：

- 找到所有入口语句
    - 程序的第一个语句

    - 跳转语句转到的语句
    - 紧跟在跳转语句后的语句
- 基本块划分
    - 入口语句到下一各入口语句之间的语句。
    - 入口语句到跳转语句之间的语句

根据上述算法，可以处理基本块的划分。因为不支持函数的使用，本编译器需要划分的部分只有if语句块，其他都是直接的全局定义或运算。但是 if 语句需要支持嵌套的 if 语句，这就给基本块的划分带来一定的麻烦。如何正确划分嵌套的 if 语句，我参考了括号的匹配，在设计翻译文法时，我要求 if 语句块的开头必须是 if 四元式， 产生 ie 四元式结尾，那么在二者之间的就是 if 语句块的正文内容，同时 if 语句的判断条件依赖于其前一语句的计算结果，所以 if 语句的基本块划分为 if 四元式语句的前一语句到 ie 语句。**基于括号匹配的思想**，栈顶的 if 语句一定与最近的 ie 语句匹配，那么栈顶的 if 语句与最近的 ie 语句之间出现的即为该 if 基本块的内容。**对else语句的处理同理。**

具体代码实现见`generator.h`文件中的`get_block`函数，输入为原始的四元式，获得基本块的集合。

寄存器和变量分配

活跃信息记录

在得到基本块后，可以按照以下算法逆序扫描基本块中的四元式：

- 初值：基本块内各变量 SYMBL[X(L)]分别填写： 若 X为非临时变量 则置 X(y); 否则置 X(n)
- 逆序扫描基本块内各四元式(设为 q:(ω B C A)): 执行：
    - ① QT[q:A(L)]:= SYMBL[A(L)];
    - ② SYMBL[A(L)]:=(n);
    - ③ QT[q:B,C(L)]:= SYMBL[B,C(L)];
    - ④ SYMBL[B,C(L)]:=(q);

具体代码实现见`generator.h`文件中的`scan_QT`函数，输入为基本块的集合，获得带活跃信息的四元式。

变量初始分配

在MIPS汇编语言的仿真软件`Mars`中内存的基址为`0x10010000`，所以变量的内存地址为基址 + 符号表中地址偏移量。变量发生内存读写的地址都按照该规则计算得到。

寄存器分配设计

首先需要一个可以记录寄存器信息的表，记为RDL，实验中使用数组实现，RDL[i]表示第 i 个寄存器中记录的变量是什么，因为在MIPS汇编语言的仿真软件`Mars`中有16个可用的变量寄存器，使用RDL的大小为16。

在得到带活跃信息的四元式后，寄存器的分配规则设计如下，例 q (ω, b, c, a)：

- 主动释放，如果存在RDL[i] == b
    - 如果b的活跃信息为 ‘n’，说明b接下来在被重新赋值前都不会被应用，该值可以被覆盖，所以可以将该寄存器直接分配给a。
    - 如果b的活跃信息为 ‘y’，说明b接下来在被重新赋值前会被应用，该值不可以被安全覆盖，需要写回内存再分配给a，但MIPS汇编语言是三寄存器语言，即运算中可以有三个寄存器参与，

所以这种情况下，优先寻找空闲寄存器。

- 寻找空寄存器
  - 寻找空闲寄存器分配。
- 强制释放
  - 如果没有空闲寄存器，且寄存器信息表中不存在当前变量信息，则选择一个在使用的寄存器，对于活跃信息不同情况处理逻辑与主动释放的处理一致。

具体代码实现见`generator.h`文件的`alloc_reg`函数。

目标代码生成设计

目标代码指令集选择

本次实验中实现的编译器使用MIPS指令集作为目标语言，MIPS指令集是一种精简指令。考虑到报告的篇幅，只介绍代码类型：

代码类型	功能
R型指令	通过三寄存器参与有func码指明的运算 e.g. add $8,$9, $10
I型指令	通过两寄存器以及立即数参与有func码指明的运算 e.g. add $8,$9, 100
J型指令	跳转指令，跳转到指令中地址指明的指令 e.g. j 0x00400000

指令集参考博客：https://blog.csdn.net/qq_39559641/article/details/89608132

目标代码生成

目标代码的生成主要依据四元式的op，由于生成的逻辑相同，只是生成运算指令不同，下面介绍运算顺序可逆的运算、不可逆的运算、赋值运算、if与else语句的目标代码生成设计。 PS：由于MIPS中浮点数运算十分复杂，本次实验中所有运算都视为int运算，float参与的运算会被降级为int。

常数之间运算

若参与运算的两个对象都是常数，直接将其计算得到常数结果。产生目标代码：

```
//清空写入目标寄存器，设置为0
code1 = "sub " + reg_a + ", " + reg_a + ", " + reg_a;
//将提前运算得到常数结果写入目标寄存器
code2 = "add " + reg_a + ", " + reg_a + ", " + 常数结果;
```

顺序可逆的运算（+，*，==，!= ...）

这里以乘法为例，以代码分析说明：

```
//如果op为*
else if(op == "*"){
    //QT_B_L为" "说明B为常数
    //QT_C_L为" "说明C为常数
    if(QT_B_L == " " && QT_C_L == " "){
        //计算得到常数结果
```

```
        reg_c = to_string(stoi(B) * stoi(C));
        reg_b = reg_a;
        //清空写入目标寄存器，设置为0
        string code = "sub " + reg_a + ", " + reg_a + ", " + reg_a;
        obj.push_back(code);
        //将提前运算得到常数结果写入目标寄存器
        code = "add " + reg_a + ", " + reg_b + ", " + reg_c;
        obj.push_back(code);
    }
    else{
        //QT_B_L为" "说明B为常数
        if(QT_B_L == " "){
            //交换运算位置
            reg_b = reg_c;
            //执行立即数运算
            reg_c = B;
        }
        //QT_C_L为" "说明C为常数
        if(QT_C_L == " "){
            //执行立即数运算
            reg_c = C;
        }
    }
    //目标代码生成
    string code = "mul " + reg_a + ", " + reg_b + ", " + reg_c;
    obj.push_back(code);
}
```

顺序不可逆的运算（-，/，>=，<= ...）

这类运算处理逻辑会比较复杂，存在三种情况，常数运算，B为常数，C为常数。

常数运算的处理逻辑在一开始以及描述过了，此处不再赘述。

B为常数情况最为复杂，其代表着 5-a 这类运算，而MIPS的立即数运算要求立即数在次寄存器位置，即C的位置，因此需要将常数的值写入一个寄存器中，为常数分配一个寄存器，则分配后该寄存器一定是不活跃的。

C为常数则生成立即数的运算代码。

这里以减法为例，以代码分析说明：

```
else if(op == "-"){
    //QT_B_L为" "说明B为常数
    //QT_C_L为" "说明C为常数
    if(QT_B_L == " " && QT_C_L == " "){
        reg_c = to_string(stoi(B) - stoi(C));
        reg_b = reg_a;
        string code = "sub " + reg_a + ", " + reg_a + ", " + reg_a;
        obj.push_back(code);
        code = "add " + reg_a + ", " + reg_b + ", " + reg_c;
        obj.push_back(code);
    }
    else{
        //QT_B_L为" "说明B为常数
        if(QT_B_L == " "){
```

```
            int tmp_reg = get_empty_reg();
            //不存在空闲寄存器
            if(tmp_reg == -1){
                //轮询强制释放
                int reg = full_index % 16;
                full_index++;
                //将被释放寄存器的值写回内存
                string code_1 = "sw $" + to_string(reg_basic_index + reg) + ", "
+ to_string(mem_basic_addr + STM.ST[RDL[reg].first].addr);
                obj.push_back(code_1);
                //该寄存器被记录为可用
                RDL[reg] = make_pair("", "n");
                reg_b = "$" + to_string(reg_basic_index + reg);
                //向分配的寄存器写入该常数的值
                string code_2 = "sub " + reg_b + ", " + reg_b + ", " + reg_b;

                obj.push_back(code_2);
                string code_3 = "add " + reg_b + ", " + reg_b + ", " + B;
                obj.push_back(code_3);
            }
            //存在空闲寄存器
            else{
                //向分配的寄存器写入该常数的值
                reg_b = "$" + to_string(reg_basic_index + tmp_reg);
                string code_2 = "sub " + reg_b + ", " + reg_b + ", " + reg_b;

                obj.push_back(code_2);
                string code_3 = "add " + reg_b + ", " + reg_b + ", " + B;
                obj.push_back(code_3);
            }
            string code = "sub " + reg_a + ", " + reg_b + ", " + reg_c;
            obj.push_back(code);
        }
        //QT_C_L为" "说明C为常数
        else if(QT_C_L == " "){
            reg_c = C;
            string code = "sub " + reg_a + ", " + reg_b + ", " + reg_c;
            obj.push_back(code);
        }
        //正常运算
        else{
            string code = "sub " + reg_a + ", " + reg_b + ", " + reg_c;
            obj.push_back(code);
        }
    }
}
```

赋值运算

赋值也有三种情况，一个是自赋值，一个是赋常数值，一个是赋变量值

代码分析说明如下：

```
else if(op == "="){
    //自赋值
    if(reg_a == reg_b){
```

```
        continue;
    }
    //清空写入目标寄存器，设置为0
    string code_1 =  "sub " + reg_a + ", " + reg_a + ", " + reg_a;
    obj.push_back(code_1);
    //B为常数，生成立即数加指令
    if(QT_B_L == " "){
        reg_b = B;
        string code_2 = "add " + reg_a + ", " + reg_a + ", " + reg_b;
        obj.push_back(code_2);
    }
    //变量赋值
    else{
        string code_2 = "add " + reg_a + ", " + reg_a + ", " + reg_b;
        obj.push_back(code_2);
    }
}
```

if与else语句

MIPS支持label跳转，由于 if 语句，else语句与 ie 语句都有括号匹配一般的关系，同时if语句 else 语句有可能存在嵌套的情况，这里也使用了跳转栈这个工具，根据四元式的op产生新的label或是新的跳转语句。

代码分析如下：

```
//若是if四元式
if(op == "if"){
    //产生等于0跳转目标代码，跳转地址待定
    string code = "beqz " + reg_b + ", fj" + to_string(f_cnt);
    f_cnt++;
    //压入跳转栈
    j_st.push(code);
    obj.push_back(code);
}
//el四元式
else if(op == "el"){
    //产生无条件跳转，因为if语句后若接else语句，执行完if语句块中内容要跳转到else尾，跳转地址待定
    string code = "b tj" + to_string(t_cnt);
    //压入跳转栈
    j_st.push(code);
    t_cnt++;
    obj.push_back(code);
    //交换位置，使无条件跳转出现在if语句块后，if的跳转地址出现在else语句前
    string tmp = obj[obj.size() - 1];
    obj[obj.size() - 1] = obj[obj.size() - 2];
    obj[obj.size() - 2] = tmp;
}
//if或el语句结束
else if(op == "ie"){
    string j_state = j_st.top();
    j_st.pop();
    //若和if匹配，产生if的跳转标签
    if(j_state.substr(0,2) == "be"){
```

```
            string code = "";
            for(int k = j_state.size() - 1 ; k >= 0 ; --k){
                if(j_state[k] == ' '){
                    break;
                }
                code += j_state[k];
            }
            reverse(code.begin(), code.end());
            code += ": ";
            obj.push_back(code);
        }
        //若和el匹配，产生el的跳转标签
        else if(j_state.substr(0,2) == "b "){
            string code = "";
            for(int k = j_state.size() - 1 ; k >= 0 ; --k){
                if(j_state[k] == ' '){
                    break;
                }
                code += j_state[k];
            }
            reverse(code.begin(), code.end());
            code += ": ";
            obj.push_back(code);
        }
    }
```

其他具体代码实现见`generator.h`文件的`generate`函数，输入四元式，输出目标代码。

实验结果

下面展示编译器实验的使用结果。

词法分析器

正确结果输出（覆盖全部支持c词法）

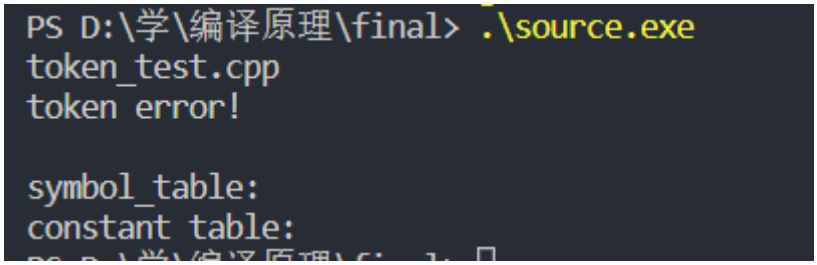
这里的测试样例单纯检验词法，不涉及文法，所以测试文件没有语义上意义。

```
PS D:\学\编译原理\final> .\source.exe
token_test.cpp
<break,04><case,05><char,06><const,07><continue,08><default,09><do,10><double,11><else,12><enum,13><extern,14><float,15><for,16><goto,17><if,18><int,19><long,20><register,21><return,22><short,23><signed,24><sizeof,25><static,26><struct,27><switch,28><typedef,29><union,30><unsigned,31><void,32><volatile,33><while,34><>=,35><<=,36><==,37><!=,38><=,39><>,40><<,41><%,42><+,43><+=,44><++,45><-,46><-=,47><--,48><*,49><*=,50></,51></=,52><(,53><),54><{,55><},56><,,57><;,58><|,61><&,62><^,63><!,64><<<,65><>>,66><->,67><.,68><#,69><variable,00><5,03><"abc",02><'c',01>
symbol_table:
variable variable    -1
constant table:
5 5
```

可见确实支持多种c语言界符与关键词，以及变量、常数、字符串、字符识别。

错误结果报错

对于`1.d`的token错误情况，报错如下：



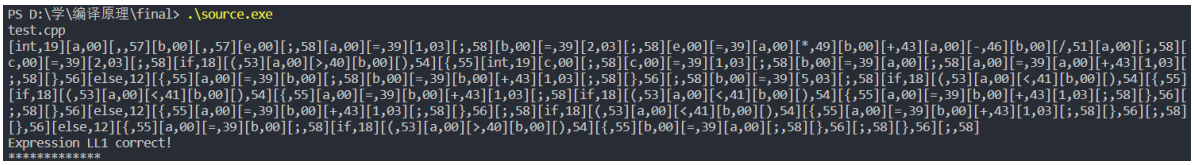
此时符号表、常数表为空，输出token error的错误。

语法分析器

正确结果出自测试文件为test.cpp, 范围覆盖了变量说明语句、算术运算表达式、赋值语句；扩展包括逻辑运算表达式、If语句。

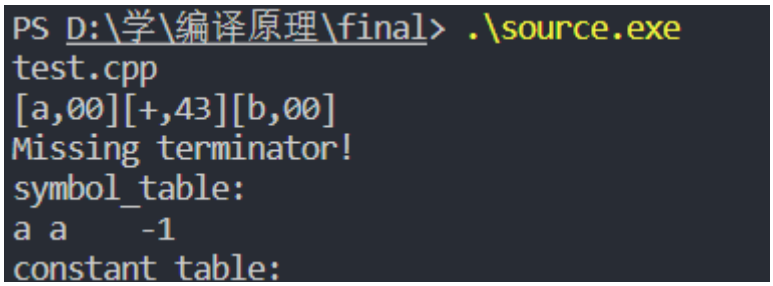
正确结果输出

上面的序列为token_seq，当测试文件没有语法错误时，输出Expression LL1 correct！字样

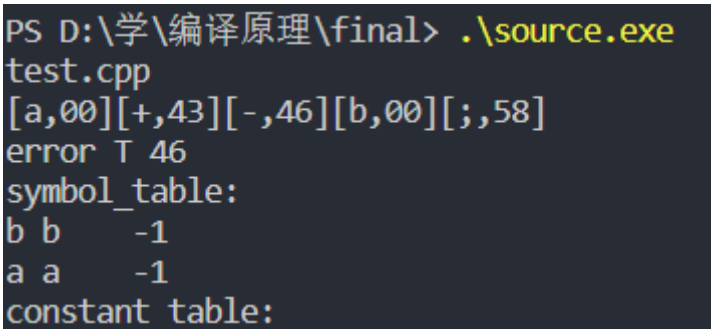


错误结果报错

对`a+b`，可以产生缺失终结符的错误。



对`a +- b;`，可以产生缺失操作数的错误。

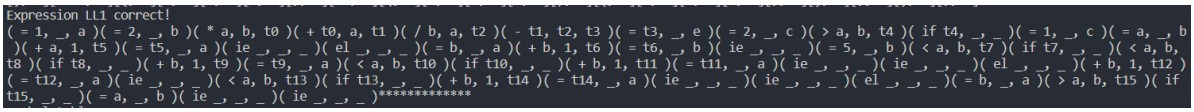


语义分析器

正确结果出自测试文件为test.cpp, 范围覆盖了变量说明语句、算术运算表达式、赋值语句；扩展包括逻辑运算表达式、If语句。

正确结果输出

在语义分析正确的情况下，可以输出正确四元式序列：



符号表更新如下：临时变量活跃信息为'n'，全局变量为'y'，正确更新种类、类型、内存地址偏移量。

```
symbol_table:
t14 t14 int v n 72
t12 t12 int v n 64
a a int v y 0
t0 t0 int v n 12
t13 t13 int v n 68
t10 t10 int v n 56
t7 t7 int v n 44
e e int v y 8
c c int v y 32
t1 t1 int v n 16
t5 t5 int v n 36
t6 t6 int v n 40
t15 t15 int v n 76
t2 t2 int v n 20
t9 t9 int v n 52
b b int v y 4
t3 t3 int v n 24
t4 t4 int v n 28
t11 t11 int v n 60
t8 t8 int v n 48
constant table:
2 2
5 5
1 1
```

错误结果报错

对于`int a; int a;`语句可以识别出重定义：

```
PS D:\学\编译原理\final> .\source.exe
test.cpp
[int,19][a,00][;,58][int,19][a,00][;,58]
a is redefined!
```

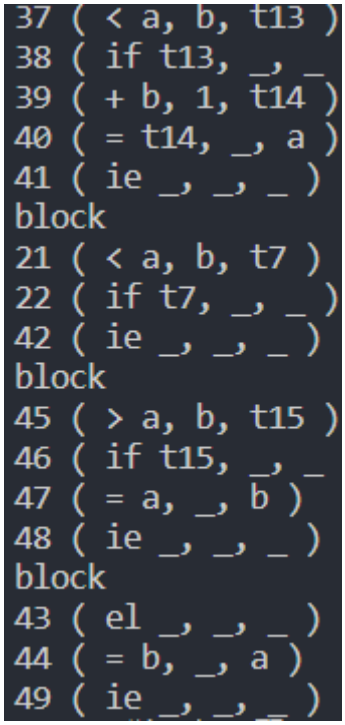
对于`char a; a = 1 + 2;`语句可以识别出类型不匹配：

```
PS D:\学\编译原理\final> .\source.exe
test.cpp
[char,06][a,00][;,58][a,00][=,39][1,03][+,43][2,03][;,58]
a is not a number!
```

基本块划分

划分结果

```
************
block
0 ( = 1, _, a )
1 ( = 2, _, b )
2 ( * a, b, t0 )
3 ( + t0, a, t1 )
4 ( / b, a, t2 )
5 ( - t1, t2, t3 )
6 ( = t3, _, e )
7 ( = 2, _, c )
block
8 ( > a, b, t4 )
9 ( if t4, _, _ )
10 ( = 1, _, c )
11 ( = a, _, b )
12 ( + a, 1, t5 )
13 ( = t5, _, a )
14 ( ie _, _, _ )
block
15 ( el _, _, _ )
16 ( = b, _, a )
17 ( + b, 1, t6 )
18 ( = t6, _, b )
19 ( ie _, _, _ )
block
20 ( = 5, _, b )
21 ( < a, b, t7 )
block
27 ( < a, b, t10 )
28 ( if t10, _, _ )
29 ( + b, 1, t11 )
30 ( = t11, _, a )
31 ( ie _, _, _ )
block
23 ( < a, b, t8 )
24 ( if t8, _, _ )
25 ( + b, 1, t9 )
26 ( = t9, _, a )
32 ( ie _, _, _ )
block
33 ( el _, _, _ )
34 ( + b, 1, t12 )
35 ( = t12, _, a )
36 ( ie _, _, _ )
block
```



产生block与测试文件`test.cpp`情况一致。测试文件中 if 语句块有6个，else语句块有3个，全局的语句块有2个，划分的block也有11个，所以基本块划分的功能正确。

中间代码生成

结果出自测试文件为test.cpp, 范围覆盖了变量说明语句、算术运算表达式、赋值语句；扩展包括逻辑运算表达式、If语句。

带活跃变量结果输出

```
(= 1( )_( )a(2))
(= 2( )_( )b(2))
(* a(3)b(4)t0(3))
(+ t0(n)a(4)t1(5))
(/ b(y)a(y)t2(5))
(- t1(n)t2(n)t3(6))
(= t3(n)_( )e(y))
(= 2( )_( )c(y))
(> a(3)b(n)t4(1))
(if t4(n)_( )_( ))
(= 1( )_( )c(y))
(= a(4)_( )b(y))
(+ a(n)1( )t5(5))
(= t5(n)_( )a(y))
(ie _( )_( )_( ))
(el _( )_( )_( ))
(= b(2)_( )a(y))
(+ b(n)1( )t6(3))
(= t6(n)_( )b(y))
(ie _( )_( )_( ))
(= 5( )_( )b(1))
(< a(y)b(y)t7(n))
(< a(y)b(y)t7(1))
(if t7(n)_( )_( ))
(< a(n)b(2)t8(1))
(if t8(n)_( )_( ))
(+ b(y)1( )t9(3))
(= t9(n)_( )a(y))
(< a(n)b(2)t10(1))
(if t10(n)_( )_( ))
(+ b(y)1( )t11(3))
(= t11(n)_( )a(y))
(ie _( )_( )_( ))
(ie _( )_( )_( ))
(el _( )_( )_( ))
(+ b(y)1( )t12(2))
(= t12(n)_( )a(y))
(ie _( )_( )_( ))
(< a(n)b(2)t13(1))
(if t13(n)_( )_( ))
(+ b(y)1( )t14(3))
(= t14(n)_( )a(y))
(ie _( )_( )_( ))
(ie _( )_( )_( ))
```

该活跃信息产生符合代码逻辑。

目标代码生成

结果出自测试文件为test.cpp, 范围覆盖了变量说明语句、算术运算表达式、赋值语句；扩展包括逻辑运算表达式、If语句。

汇编代码与c语言的对照分析在代码段的注释中

```
.text
// a = 1;
sub $8, $8, $8
add $8, $8, 1
b = 2;
sub $9, $9, $9
add $9, $9, 2
// e = a * b + a - b / a;
mul $10, $8, $9
add $11, $10, $8
div $10, $9, $8
sub $10, $11, $10
sub $11, $11, $11
add $11, $11, $10
// c = 2;
sub $10, $10, $10
add $10, $10, 2
// if (a > b)
sgt $9, $8, $9
beqz $9, fj0
// c = 1;
sub $10, $10, $10
add $10, $10, 1
// b = a;
sub $9, $9, $9
add $9, $9, $8
// a = a + 1;
add $12, $8, 1
sub $8, $8, $8
add $8, $8, $12
b tj0
// else
fj0:
// a = b;
sub $8, $8, $8
add $8, $8, $9
// b = b + 1;
add $12, $9, 1
sub $9, $9, $9
add $9, $9, $12
// else end
tj0:
// b = 5;
sub $9, $9, $9
add $9, $9, 5
// if(a < b){
slt $12, $8, $9
beqz $12, fj1
//  if(a < b){
slt $12, $8, $9
```

```
beqz $12, fj2
// a = b + 1;
add $8, $9, 1
sub $12, $12, $12
add $12, $12, $8
//    if(a < b){
slt $8, $12, $9
beqz $8, fj3
// a = b + 1;
add $8, $9, 1
sub $12, $12, $12
add $12, $12, $8
fj3:
b tj1
//  else{
fj2:
// a = b + 1;
add $8, $9, 1
sub $12, $12, $12
add $12, $12, $8
tj1:
// if(a < b){
slt $8, $12, $9
beqz $8, fj4
// a = b + 1;
add $8, $9, 1
sub $12, $12, $12
add $12, $12, $8
fj4:
b tj2
//else{
fj1:
// a = b;
sub $12, $12, $12
add $12, $12, $9
// if(a > b)
sgt $8, $12, $9
beqz $8, fj5
sub $9, $9, $9
add $9, $9, $12
fj5:
tj2:
//写回内存
sw $9, 268500996
sw $10, 268501024
sw $11, 268501000
sw $12, 268500992
```

MIPS目标代码生成如上，可见程序可以handle嵌套的if语句块，生成正确标签，同时生成正确的目标代码。正确性在运行结果检验。

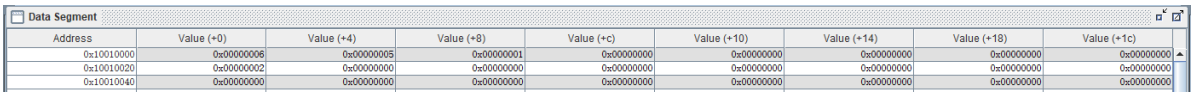
目标代码运行结果

先对程序的最终结果进行分析，并对此对比目标运行结果分析正确性。（基于测试文件为test.cpp分析，单步分析过于冗长，这里直接讨论最终结果的一致性，单步结果实验中也是符合的。）

```
int a, b, e;
a = 1;
b = 2;
// e = 1
e = a * b + a - b / a;
c = 2;
//此处为false
if (a > b){
    int c;
    c = 1;
    b = a;
    a = a + 1;
}
else{
    //此时a = 2, b = 3
    a = b;
    b = b + 1;
};
//b = 5;
b = 5;
//true
if(a < b){
    //true
    if(a < b){
        // a = 6
        a = b + 1;
        //false
        if(a < b){
            a = b + 1;
        };
    }
    //false
    else{
        a = b + 1;
    };
    //false
    if(a < b){
        a = b + 1;
    };
}
else{
    a = b;
    if(a > b){
        b = a;
    };
};
// a = 6, b = 5, c = 2, e = 1
```

在Mars仿真环境中运行结果如下：结合符号表的地址偏移量分析，写回内存的值 a = 6, b = 5, e = 1, c = 2。与分析一致，说明生成的目标代码正确。（仿真结果需要放大才能看到）

```
symbol_table:
t14 t14 int v n 72
t12 t12 int v n 64
a a int v y 0
t0 t0 int v n 12
t13 t13 int v n 68
t10 t10 int v n 56
t7 t7 int v n 44
e e int v y 8
c c int v y 32
t1 t1 int v n 16
t5 t5 int v n 36
t6 t6 int v n 40
t15 t15 int v n 76
t2 t2 int v n 20
t9 t9 int v n 52
b b int v y 4
t3 t3 int v n 24
t4 t4 int v n 28
t11 t11 int v n 60
t8 t8 int v n 48
constant table:
2 2
5 5
1 1
```



若想检验复现结果，可以将编译得到的`.txt`文件在Mars中打开单步运行检查寄存器中结果与预期是否一致。

总结

本次实验是这个学期编译原理课程的大作业。综合了前三次实验，还需要考虑到后端的实现，是一次极具挑战性，综合性极强的实验，给我提出了极大的挑战，考验了我的编程能力，对编译原理的掌握程度，代码纠错能力，我从中也有不少收获，也因此有所成长。

在词法分析器阶段，由于之前lab1就实现了一个c语言词法扫描器，打下了良好基础，只需在其基础上稍作修改即可适配到编译器的开发之中，为编译器开发节省了大量的前端设计开发时间，可以集中精力用于编译器后续阶段的开发。

在语法分析阶段，需要重新设计文法，由于并不是重新实现一个C语言，而是部分的C语言语句，所以我从C语言文法中抽离简化其中的部分语句，使用LL1分析法，获得了LL1文法的分析表，虽然其中也有一些瑕疵，但是也能够在规则内正常运行，通过一些LL1分析法之外的补丁代码修复一些bug，是语法分析可以正常运行，在这一部分不算遇到特别大的困难。

在语义分析阶段，需要改造语法分析器，在合适的位置插入动作符号才能得到正确的四元式，在这一部分，我也花了不少时间检查实验动作符号的插入位置是否合适，如果不合适应该插入到什么地方，如果当前翻译文法不能满足四元式的生成，如何进一步修改扩充文法，都是这一阶段需要解决的问题。不过，通过实验的开发，我还是解决了这些问题，成功生成了正确的四元式。

在编译器后端开发阶段，这部分由于在前期未有实验铺垫，这一部分也是编译器开发中遇到困难最多，开发用时最久的部分。首先是对基本块的忽视，一开始考虑只实现简单的一些文法，所以所有变量均为全局变量，但在拓展if语句后，没有基本块的划分的情况下，活跃信息基本无法正确分析，因此重新加入了基本块，导致代码进行了一轮重构；再者是拓展了if语句后，没有考虑到if语句的嵌套情况，思维都是线性的开发思维，考虑及其不周到，但代码也已经开发了大半，也要硬着头皮将代码重新构造。引入了栈来解决嵌套if的问题，这一部分是我思考最多的部分，因为可以参考的资料不多。

最终看到目标代码能够正确生成并运行得到正确结果，确实使我十分有成就感。但本次实验仍存在许多不足，如未对中间代码进行优化，语法十分简单，远远未达到可用的程度。希望在日后，能够有机会，有动力来完善这一项目，继续进步。