编译原理实验报告

实验题目：		学号：
日期：	班级：	姓名：
Email：		

实验目的：

本实验要实现的是一个可以把四元式翻译成 x86 目标代码的代码生成器。代码生成器可以求解待用信息、活跃信息、寄存器描述符和地址描述符等，根据他们分配寄存器，并逐条把四元式翻译成汇编代码。

实验环境介绍：

硬件环境：

Legion R7000P2021 AMD Ryzen 7 5800H with Radeon Graphics 3.20 GHz

软件环境：

Clion

解决问题的主要思路：

本实验中将符号表以及四元式表作为输入，并根据实验要求输出对应的目标代码，主要参考实验给出的伪代码来完成。

实验解决过程分以下几个步骤：

1. 针对给出的符号表和四元式表，将其划分成若干个基本块。

2. 针对每一个基本块中的变量求出其待用信息和活跃信息，根据这些信息以及基本块进一步求出目标代码。

3. 无论输入的程序是否正确（即使输入是 Syntax Error），最后输出目标代码即可。

实验步骤：

本实验在生成目标代码时需要考虑到寄存器的使用，此处我们有三个寄存器，分别为 R0、R1、R2，分配寄存器的过程也是先分配 R0，再分配 R1，后分配 R2。

本实验中目标代码的映射规则如下：

中间代码	目标代码	说明
(=,lhs/UINT/UFLOAT,-,dest)	mov R, lhs/UINT/UFLOAT	(1) R 是新分配给 dest 的寄存器 (2) 如果 lhs∈Rval(R)，则不生成目标代码
(j,-,-,dest)	jmp ?dest	无条件转移指令

(jnz,lhs,-,dest)	mov R, lhs
	cmp R, 0
	jne ?dest
		如果 lhs∈Rval(R)，则不生成第 1 条代码

(jθ,lhs,rhs,dest)	mov R, lhs
	cmp R, rhs
	jθ ?dest
		如果 lhs∈Rval(R)，则不生成第 1 条代码

(+,lhs,rhs,dest)	mov R, lhs add R, rhs	(1) R 是新分配给 dest 的寄存器 (2) 如果 lhs∈Rval(R)，则不生成第 1 条目标代码
(-,lhs,rhs,dest)	mov R, lhs sub R, rhs	(1) R 是新分配给 dest 的寄存器 (2) 如果 lhs∈Rval(R)，则不生成第 1 条目标代码
(*,lhs,rhs,dest)	mov R, lhs mul R, rhs	(1) R 是新分配给 dest 的寄存器，不考虑特殊寄存器 (2) 如果 lhs∈Rval(R)，则不生成第 1 条目标代码
(/,lhs,rhs,dest)	mov R, lhs div R, rhs	(1) R 是新分配给 dest 的寄存器，不考虑特殊寄存器 (2) 如果 lhs∈Rval(R)，则不生成第 1 条目标代码

(==,lhs,rhs,dest)	mov R, lhs
	cmp R, rhs
	sete R
		(1) R 是新分配给 dest 的寄存器
		(2) 如果 lhs∈Rval(R)，则不生成第 1 条目标代码

(!=,lhs,rhs,dest)	mov R, lhs
	cmp R, rhs
	setne R
		(1) R 是新分配给 dest 的寄存器
		(2) 如果 lhs∈Rval(R)，则不生成第 1 条目标代码

(<,lhs,rhs,dest)	mov R, lhs
	cmp R, rhs
	setl R
		(1) R 是新分配给 dest 的寄存器
		(2) 如果 lhs∈Rval(R)，则不生成第 1 条目标代码

(<=,lhs,rhs,dest)	mov R, lhs
	cmp R, rhs
	setle R
		(1) R 是新分配给 dest 的寄存器
		(2) 如果 lhs∈Rval(R)，则不生成第 1 条目标代码

(>,lhs,rhs,dest)	mov R, lhs
	cmp R, rhs
	setg R
		(1) R 是新分配给 dest 的寄存器
		(2) 如果 lhs∈Rval(R)，则不生成第 1 条目标代码

(>=,lhs,rhs,dest)	mov R, lhs
	cmp R, rhs
	setge R
		(1) R 是新分配给 dest 的寄存器
		(2) 如果 lhs∈Rval(R)，则不生成第 1 条目标代码

(&&,lhs,rhs,dest)	mov R, lhs and R, rhs	(1) R 是新分配给 dest 的寄存器，不考虑特殊寄存器 (2) 如果 lhs∈Rval(R)，则不生成第 1 条目标代码

(||,lhs,rhs,dest)	mov R, lhs or R, rhs	(1) R 是新分配给 dest 的寄存器，不考虑特殊寄存器 (2) 如果 lhs∈Rval(R)，则不生成第 1 条目标代码
(!,lhs,-,dest)	mov R, lhs not R	(1) R 是新分配给 dest 的寄存器，不考虑特殊寄存器 (2) 如果 lhs∈Rval(R)，则不生成第 1 条目标代码
(R,-,-,dest)	jmp ?read(dest)	视作无条件转移指令
(W,-,-,dest)	jmp ?write(dest)	视作无条件转移指令
(End,-,-,-)	halt	程序终止
Syntax Error	halt	程序异常终止

1. 本实验中涉及到符号表，而符号表中的符号有四个属性 name，type，value，offset，为此我们定义一个结构体来表示，如下所示：

```
struct symbol {
    string name;
    int type;
    string value;
    int offset;
};
```

同时在输入的时候还需要输入四元式表，同样定义一个类，如下所示：

```
struct quad {
    string op;
    string arg1;
    string arg2;
    string result;
};
```

2. 在得到所有四元式后，对其进行划分成若干个基本块，此处参考实验中给出的伪代码，如下所示：

```
void Genblock() {
    int size = Quad.size();
    vector<int> vec; // 基本块入口数组,值为1表示是基本块入口
    vec.resize(Quad.size(),0);
    vec[0] = 1; // 第一条四元式是入口
    for (int i = 0; i < size; i++) {
        auto t = Quad[i];
        if (t.op[0] == 'j' && t.op.size() > 1) {
            int tep = atoi(t.result.c_str());
            vec[tep] = 1;
            if (i < size - 1) vec[i + 1] = 1;
        }
        if (t.op == "j") {
            int tep = atoi(t.result.c_str());
            vec[tep] = 1;
        }
        if (t.op == "R") vec[i] = 1;
        if (t.op == "W") vec[i] = 1;
    }
```

```
    int i = 0;
    while (i < size) {
        auto t = Quad[i];
        if (vec[i]) { // i是基本块入口
            if (i == size - 1) {
                basicblock.push_back(make_pair(i,i));
                break;
            }
            for (int j = i + 1; j < size; j++) {
                if (vec[j]) {
                    basicblock.push_back(make_pair(i,j-1));
                    i = j;
                    break;
                }
                else if (Quad[j].op[0] == 'j' || Quad[j].op == "End") {
                    basicblock.push_back(make_pair(i,j));
                    i = j + 1;
                    break;
                }
            }
        } else i++;
    }
}
```

3. 划分完基本块后，进一步就是求出每个变量的待用信息和活跃信息。我们以基本块为单位，求出这些信息，如下所示：

```
void GenLive(int end, int start){
    vector<LiveInfor> SymbolInfor(symbolcnt, {-1, 1});
    vector<LiveInfor> TempInfor(tempcnt, {-1, 0});
    for (int i = end; i >= start; i--) { // 逆序求解
        array<string, 3> Infor = { Quad[i].arg1, Quad[i].arg2, Quad[i].result};
        for (int j = 2; j >= 0; j--) {
            if (Infor[j][0] == 'T') {
                int index = 0;
                if(Infor[j][1] == 'B') index = stoi(Infor[j].substr(2));
                else index = stoi(Infor[j].substr(1, Infor[j].find('_') - 1));
                auto &vars = Infor[j][1] == 'B' ? SymbolInfor : TempInfor;
                LiveTable[i][j] = vars[index];
                if (j == 2) vars[index] = { -1, 0 };
                else vars[index] = { i, 1 };
            }
        }
    }
}

void GetLive(){
    LiveTable.resize(Quad.size());
    for (auto item : basicblock) GenLive(item.second, item.first);
}
```

4. 接下来就是生成目标代码，参考实验给出的伪代码，对于每一条四元式，先判断它的类型（即是运算类型还是跳转类型等），然后针对不同类型的四元式进行不同的操作。生成完毕后，将出口处是活跃变量且只在内存中的变量写入寄存器。如果基本块的最后一个四元式是跳转语句，那么根据不同的跳转类型生成代码即可。最后将寄存器描述符和地址描述符清空。
如果四元式是运算类型：

```
if(t.op[0] != 'j' && t.op != "W" && t.op != "R" && t.op != "End"){ // 说明它是运算类型的
            if (t.arg1[0] == 'T'){
                if(LiveTable[i][0].use == -1) Vari[t.arg1] = INF;
                else Vari[t.arg1] = LiveTable[i][0].use;
            }
            if (t.arg2[0] == 'T') {
                if(LiveTable[i][1].use == -1) Vari[t.arg2] = INF;
                else Vari[t.arg2] = LiveTable[i][1].use;

            }
            if (t.result[0] == 'T') {
                if(LiveTable[i][2].use == -1) Vari[t.result] = INF;
                else Vari[t.result] = LiveTable[i][2].use;
            }
            string Rz = getReg(t,i); // 分配寄存器
            string temp = GetReg(t.arg1);
            string temp1 = t.arg2;
            if(temp1 != "-") temp1 = GetReg(t.arg2);
            if (temp == Rz) {
                if (temp1 != "-") { // 一个操作数的情况
                    string val;
                    if (temp1 == t.arg2 && temp1[0] == 'T') val = Genadd(temp1);
                    else val = temp1;
                    GenInstr(t.op, Rz,  val, i);
                }
                if (t.op == "!") result[i].push_back("not " + temp);
                Aval[t.arg1].erase(Rz);
            }
```

```
            else {
                string x;
                if (temp == t.arg1 && temp[0] == 'T') x = Genadd(temp);
                else x = temp;
                result[i].push_back("mov " + Rz + ", " + x);
                if (temp1 != "-") {
                    string val;
                    if (temp1 == t.arg2 && temp1[0] == 'T') val = Genadd(temp1);
                    else val = temp1;//寄存器
                    GenInstr(t.op, Rz, val, i);
                }
            }
```

```
                if (temp1 == Rz) Aval[t.arg2].erase(Rz);
                Rval[Rz].clear();
                Rval[Rz].insert(t.result);
                Aval[t.result].clear();
                Aval[t.result].insert(Rz);
                ReaReg(t.arg1);
                ReaReg(t.arg2);
                if (LiveTable[i][2].use == -1) Vari[t.result] = INF;
                else Vari[t.result] = LiveTable[i][2].use;

            }
```

如果四元式是 R/W 类型：

```
            else if(t.op == "R") result[i].push_back("jmp ?read(" + Genadd(t.result) + ")");
            else if(t.op == "W") result[i].push_back("jmp ?write(" + Genadd(t.result) + ")");
```

5. 针对不同类型的四元式生成目标代码时，采用的基本思路都是先为四元式的左值分配寄存器，然后针对四元式中的 op1、op2 和左值的存储情况生成不同的 MOV 代码。例如，如果 op1 在寄存器中，且 op1 后续不活跃，那么将 op1 的寄存器分配给左值，此时就不需要生成额外的代码，若 op1 后续活跃，那么需要将 op1 中的内容 MOV 到新生成的寄存器中，以此类推。

左值、op1、op2 依次判断并分配寄存器

```
                if (t.arg1[0] == 'T'){
                    if(LiveTable[i][0].use == -1) Vari[t.arg1] = INF;
                    else Vari[t.arg1] = LiveTable[i][0].use;
                }
                if (t.arg2[0] == 'T') {
                    if(LiveTable[i][1].use == -1) Vari[t.arg2] = INF;
                    else Vari[t.arg2] = LiveTable[i][1].use;

                }
                if (t.result[0] == 'T') {
                    if(LiveTable[i][2].use == -1) Vari[t.result] = INF;
                    else Vari[t.result] = LiveTable[i][2].use;
                }
                string Rz = getReg(t,i); // 分配寄存器
                string temp = GetReg(t.arg1);
                string temp1 = t.arg2;
                if(temp1 != "-") temp1 = GetReg(t.arg2);
```

6. 针对不同类型的跳转四元式，我们先定义一个 vector<pair<string,string>> Tran 来映射，然后计算得出要跳转到标号即可。
Tran 如下：

```
vector<pair<string,string>> Tran = {
        {"j==", "je"}, {"j!=", "jne"}, {"j<",  "jl"}, {"j<=", "jle"},
        {"j>",  "jg"}, {"j>=", "jge"}
};
```

计算标号后，即可生成跳转指令的目标代码，如下：

```
void GenJ(quad tep, int Index){
    string Op1 = tep.arg1;
    string Op2 = tep.arg2;
    for (auto str: Aval[Op1])
        if (str[0] == 'R') {
            Op1 = str;
            break;
        }
    for (auto str: Aval[Op2])
        if (str[0] == 'R') {
            Op2 = str;
            break;
        }
    if (Op1 == tep.arg1) {
        Op1 = getReg(tep, Index);
        result[Index].push_back("mov " + Op1 + ", " + Genadd(tep.arg1));
    }
    if (Op2[0] == 'T') result[Index].push_back("cmp " + Op1 + ", " + Genadd(Op2));
    else result[Index].push_back("cmp " + Op1 + ", " + Op2);
    for(auto t : Tran)
        if(t.first == tep.op){
            result[Index].push_back(t.second + " ?" + tep.result);
            break;
        }
    int islabel = stoi(tep.result);
    IsLabel[islabel] = true;
}
```

实验结果展示及分析：

1. 对于如下输入的符号表以及四元式表：

```
3
a 0 null 0
b 0 null 4
c 0 null 8
11
23
0: (R,-,-,TB0)
1: (=,0,-,T0_i)
2: (=,T0_i,-,TB1)
3: (=,0,-,T1_i)
4: (j>=,TB0,T1_i,6)
5: (j,-,-,21)
6: (=,1,-,T2_i)
7: (-,TB0,T2_i,T3_i)
```

```
8: (=,T3_i,-,TB0)
9: (=,2,-,T4_i)
10: (/,TB0,T4_i,T5_i)
11: (=,2,-,T6_i)
12: (*,T5_i,T6_i,T7_i)
13: (-,TB0,T7_i,T8_i)
14: (=,T8_i,-,TB2)
15: (=,0,-,T9_i)
16: (j!=,TB2,T9_i,18)
17: (j,-,-,3)
18: (+,TB1,TB0,T10_i)
19: (=,T10_i,-,TB1)
20: (j,-,-,3)
21: (W,-,-,TB1)
22: (End,-,-,-)
```

该程序不存在语法错误，产生的目标代码输出如下所示：

```
jmp ?read([ebp-0])
mov R0, 0
mov [ebp-4], R0
?3:
mov R0, 0
mov R1, [ebp-0]
cmp R1, R0
jge ?6
jmp ?21
?6:
mov R0, 1
mov R1, [ebp-0]
sub R1, R0
mov R0, 2
mov R2, R1
div R2, R0
mov R0, 2
mul R2, R0
mov R0, R1
sub R0, R2
mov R2, 0
mov [ebp-0], R1
mov [ebp-8], R0
cmp R0, R2
jne ?18
jmp ?3
?18:
mov R0, [ebp-4]
add R0, [ebp-0]
mov [ebp-4], R0
```

```
jmp ?3
?21:
jmp ?write([ebp-4])
halt
```

2. 如果程序存在语法错误，即输入的是 Syntax Error，则直接输出 halt 即可，如下所示