

# 实验报告

## 编译原理课程实验

——模块五

组长	1950071	丁晨航
小组成员	1951976	李林飞
小组成员	1953281	王文炯
小组成员	1750798	张越

同济大学

Tongji University

## 目录

一、实验简介.....	3
二、实验原理.....	3
三、分析与实现.....	3
四、实验结果.....	4
五、总结与思考.....	4
1.实验小结.....	4
2.实验亮点.....	4
3.平台使用反馈.....	4
4.课程建议.....	4
5.小组分工与成绩权重.....	5

## 一、实验简介

- 局部优化方法
  - 算法 1：基本块划分（前提：代码是三地址代码）
  - 算法 2：基本快优化（基本块的 DAG 构造算法）
- 循环优化
  - 算法 3：查找循环的不变运算
  - 算法 4：代码外提
  - 算法 5：强度削弱和删除归纳变量（前提：指定归纳变量）

## 二、实验原理

常用的优化技术有：

1. 删除公共子表达式
2. 复写传播

3. 删除无用代码
4. 代码外提
5. 强度削弱
6. 删除归纳变量

## 三、分析与实现

### 1. 数据结构设计

#### 1.1 四元式结构体

C++

```
1  /*
2  * 四元式结构体
3  */
4  struct Quaternary
5  {
6      int id;           // 序号
7      string op;        // 操作符
8      string arg1;      // 操作数1
9      string arg2;      // 操作数2
10     string res;       // 结果
11     string label;     // 与四元式绑定的跳转标号
12 };
```

#### 1.2 基本块

C++

```
1  // 一个基本块类
2  struct BasicBlock
3  {
4      // 基本块id
5      int block_id;
6      // 基本块中含有的四元式的id集合
7      vector<int> quaList;
8      // 判断该基本块是否是循环体，辅助循环不变运算、代码外提和强度削弱
9      bool loop = false;
10     // 基本块中循环不变运算的集合
11     vector<int> unchangeOpList;
12 };
```

### 1.3 DAG图类

C++

```
1 //DAG图类
2 class DAGraph
3 {
4 private:
5     vector<Node*> nodeList;          //结点集
6     unordered_map<string, int> mappingTable;          //标识符与结点的对应表, 用
    于Node(A)
7
8     bool ifNodeDefine(string identifier);          //标识符的Node是否定义, 1为定
    义
9     int getNode(string identifier);          //求Node(), 若未定义则返回-1
10    int createLeafNode(string identifier);          //构造标记为identifier的叶结
    点, 返回新节点的下标
11    bool ifNodeConstLeaf(string identifier);          //标识符的Node是否是标记为
    常数的叶结点, 保证结点已存在, 1为是
12    bool ifNodeConstLeaf(int node_id);          //Node是否是标记为常数的叶结点,
    保证结点已存在, 1为是
13    int excuteOP_1(int nodeB, string op);          //执行op B, 即1类代码, 保证结
    点是常数叶结点, 返回结果结点的下标
14    int excuteOP_2(int nodeB, int nodeC, string op);
15    //执行B op C, 即2类代码, 保证结点是常数叶结点, 且id2为第二个操作数, 返回结果结点的
    下标
16    void removeNode();          //删除最新的结点, 并从mappingtable中移除
17    int FindNode(string op, int lc, int rc = -1);          //检查是否存在一结
    点, 其后继为lc, rc且标记为op, 若没有则返回-1
18    int createInnerNode(string op, int lc, int rc = -1);          //构造内部结
    点, 其label为op, 子节点为lc, rc, 返回新节点的下标
19    void resIdentifierAdd(string identifier, int n);          //对代码中A的操
    作, 即步骤4, 将A附加在结点n上
20
21 public:
22     DAGraph();
23     ~DAGraph();
24     void Clear();
25
26     void outputGraph();          //输出DAG图
27     void constructbyTac(Quaternary& quat);          //读入一条四元式, 构造DAG图
28     void outputQuaternary();          //生成DAG对应的四元式
29     void deleteUselessAssign();          //删除无用赋值, 假设T开头的标识符在基本块
    后不会被引用
30     void DAGOptimize(BasicBlock& block, vector<Quaternary>& qua);
    //DAG优化
31
32 };
```

## 1.4 DAG结点类

C++

```
1 //DAG结点类
2 class Node
3 {
4 private:
5     string label;           //标记
6     bool ifLeaf;           //是否是叶结点, 1为是
7     bool ifConst;          //是否是标记为常数的叶结点, 1为是
8     set<string> identifierSet; //附加标识符集
9     int leftChild;         //左子结点
10    int rightChild;        //右子结点
11
12 public:
13     Node(string l, bool ifl, bool ifc);
14     Node(string l, int lc, int rc = -1);
15     string getLabel() { return label; }
16     bool getIfLeaf() { return ifLeaf; }
17     bool getIfConst() { return ifConst; }
18     int getLeftChild() { return leftChild; }
19     int getRightChild() { return rightChild; }
20     set<string> getIdentifierSet() { return identifierSet; }
21
22     void identifierAdd(string identifier); //集合增加元素
23     void identifierRemove(string identifier); //集合删除元素
24
25     friend ostream& operator << (ostream& os, Node& node);
26 };
```

## 7. 数据预处理

本项目需要进行数据预处理,是因为我们输入的是三地址代码。但是实际上,我们在分析代码时,通过四元式更加适合我们提取代码中的信息。因此,如何将三地址代码转换成四元式是我们项目的一大重点。

在分析过程中,我们将三地址代码分成两个部分:跳转表达式和运算表达式所组成的代码。我们以此进行分类,再根据字符串匹配的思想进行三地址代码的解析,例如下图代码:

C++

```
1 vector<Quaternary> three2four(const vector<string> codes)
2 {
3     vector<Quaternary> res;
4     vector<BigLabel> Label;
5
```

```

6      int tac1Num = 0; // 公用四元式1的四元式个数
7      int length = codes.size();
8
9      int i = 0;
10     for (; i < length; i++)
11     {
12         string coding = codes[i];
13         // 退出语句
14         string::size_type idx_E = coding.find("E:");
15         if (idx_E != string::npos)
16         {
17             Quaternary tac;
18             tac.op = "halt";
19             tac.arg1 = "#";
20             tac.arg2 = "#";
21             tac.res = "#";
22             tac.label = "E:";
23             res.push_back(tac);
24             continue;
25         }
26         string label_temp = loadLabel(codes[i], coding);
27         // 跳转语句
28         string::size_type idx_goto = coding.find("goto");
29         if (idx_goto != string::npos)
30         {
31             // 分支跳转语句
32             string::size_type idx_if = coding.find("if");
33             if (idx_if != string::npos)
34             {
35                 int pos_start = (int)idx_if + 3; // if语句判断开始字段
36                 int pos_end = idx_goto; // 不包括该位字符
37                 int pos_num = pos_end - pos_start;
38                 string s_temp = coding.substr(pos_start, pos_num);
39                 .....
40                 // 运算语句
41                 // 数组特殊运算语句
42                 string::size_type idx_value = coding.find(":="); // 老师所给文件有错误，赋
值均应该有:=, 且一定会有
43                 string::size_type idx_arrleft = coding.find("[");
44                 string::size_type idx_arrright = coding.find("]");
45                 if (idx_arrleft != string::npos)
46                 {
47                     string arr_judge = "";
48                     int pos_num = 0;
49                     string s_temp = "";
50                     string::size_type idx_judge = coding.find("+");
51                     if (idx_judge != string::npos) arr_judge = "+";
52                     idx_judge = coding.find("-");
53                     if (idx_judge != string::npos) arr_judge = "-";

```

```

53         if (idx_judge != string::npos) arr_judge = "...";
54         idx_judge = coding.find("*");
55         if (idx_judge != string::npos) arr_judge = "*";
56         idx_judge = coding.find("/");
57         if (idx_judge != string::npos) arr_judge = "/";
58         idx_judge = coding.find("mod");
59         if (idx_judge != string::npos) arr_judge = "mod";
60         .....
61         return res;
62     }

```

在解析过程中，我们注意到一些比较重要的代码信息，如下进行展开说明：

① 标号 由于我们不确定每一条三地址代码的标号有没有标出、有没有在后续的分析中产生作用，因此，我们将标号作为四元式结构体中的一个属性进行解析。

但是，我们希望对这个解析进行更进一步。因为从所给示例中我们会发现特殊情况，例如  $T4[T3]=T9+1$ ，这样的三地址代码将会转换成不止一条四元式。因此最终，四元式的数量和三地址代码的数量不是一致的。所以，我们对于标号的解析，将会以四元式作为依据。即我们可以根据标号跳转到下一个四元式。

在实现过程中，我们利用了课堂中所学习到的回填的思想。当我们发现跳转语句的标号时，我们很可能不知道此时对应的四元式的标号，所以我们将这个标号和本四元式作为一个key（实际代码中为结构体）。当我们最终完成所有四元式的扫描，再进行回填，已经知道每一个标号对应的四元式标号了，我们就可以进行回填操作，代码如下图：



C++

```
1 // 通过key暂存
2 string s_temp_look = s_temp + ":";
3 string::size_type idx_lookfor;
4 for (int lookfor = 0; lookfor < codes.size(); lookfor++)
5 {
6     idx_lookfor = codes[lookfor].find(s_temp_look);
7     if (idx_lookfor != string::npos)
8     {
9         tac.res = "-1";
10        BigLabel label = { s_temp_look, res.size() };
11        Label.push_back(label);
12        break;
13    }
14 }
15
16 tac.label = label_temp;
17
18 // 回填
19 for (int i = 0; i < res.size(); i++)
20 {
21     res[i].id = i;
22     // 开始模拟回填
23     if (res[i].label != "")
24     {
25         simuBackpatch(res[i].label, i, Label, res);
26     }
27 }
```

② 取地址。本项目所给示例中，有类似addr(A)这样的代码。但是实际在优化过程中，我们并不需要解析A数组的内部信息，所以对于addr(A)我们可以视其为普通的变量T1。

③ 数组。数组在本项目三地址代码转换为四元式中可以说是一个比较难的部分，因为可能会额外生成一个新的四元式。我们先将除了数组之外的操作符之外的其他操作符进行正常的解析，于是只剩下数组，我们就可以按照下面的规则进行解析：

$A := B[C] \rightarrow \{ [=], B, C, A \}$

$D[C] := B \rightarrow \{ [=], B, D, C \}$

**模块效果图**

```

四元式集合读入测试:
0: := 3.14 # T0
1: * 2 T0 T1
2: + R r T2
3: * T1 T2 A
4: := A # B
5: * 2 T0 T3
6: + R r T4
7: * T3 T4 T5
8: - R r T6
9: * T5 T6 B
10: := 1 # I
11: j> I 10 25
12: * 2 J T1
13: * 10 I T2
14: + T2 T1 T3
15: - A 11 T4
16: * 2 J T5
17: * 10 I T6
18: + T6 T5 T7
19: - A 11 T8
20: =[] T8 T7 T9
21: + T9 1 X1
22: []= X1 T4 T3
23: + I 1 I
24: j # # 11
25: halt # # #

```

### 3. 基本块划分

预处理：将三地址代码形式的文件数据转换为四元式结构体数组

#### 基本块划分算法

8. 找出中间语言程序中各个基本块的入口语句：
  - a. 程序第一个语句
  - b. 能由条件转移语句或无条件转移语句到的语句
  - c. 紧跟在条件转移语句后面的语句
9. 对以上求出的每个入口语句，确定其所属的基本块。它是由该入口语句到下一个语句(不包括该入口语句)、或到一转移语句(包括该转移语句)、或一停语句(包括该停语句)之间的语句序列组成的。
10. 凡未被纳入某一基本块中的语句，可以从程序中删除。

C++

```

1  // 输入四元式集合，返回基本块的信息
2  vector<BasicBlock> computeBasicBlock(vector<Quaternary> qua)
3  {
4      void QuickSort(vector<int> &array, int low, int high);
5      int getStandard(vector<int> &array, int i, int j);
6      Quaternary code;
7      vector<int> basicBlockStart;
8      vector<vector<int>> basicBlock;
9
10     // 0是基本块第一个入口点
11     basicBlockStart.push_back(0);
12     for(int i = 0; i < qua.size(); i++)

```

```

13     {
14         // goto操作
15         if(qua[i].op.find("j") != qua[i].op.npos)
16         {
17             if(qua[i].op.size()>1)
18             {
19                 basicBlockStart.push_back(qua[i].id + 1);
20             }
21             basicBlockStart.push_back(stoi(qua[i].res));
22         }
23         if(qua[i].op.find("halt") != qua[i].op.npos)
24         {
25             basicBlockStart.push_back(qua[i].id);
26         }
27     }
28
29     //把qua.size()(假设无穷大)作为最后一个，防止容器越界
30     basicBlockStart.push_back(qua.size());
31
32     // 对基本块开始语句进行预排序，便于后续划分
33     QuickSort(basicBlockStart, 0, basicBlockStart.size() - 1);
34     // 去重排序
35     basicBlockStart.erase(unique(basicBlockStart.begin(),
36                                 basicBlockStart.end()), basicBlockStart.end());
37
38     int blockIndex = 0;
39     vector<BasicBlock> basic_blocks;
40     for(int i = 0; i < basicBlockStart.size() - 1; i++)
41     {
42         BasicBlock b;
43         b.block_id = blockIndex;
44
45         for (int k = basicBlockStart[i]; k < basicBlockStart[i + 1]; k++)
46         {
47             if(k < qua.size())
48                 b.quaList.push_back(k);
49             else
50                 break;
51         }
52         basic_blocks.push_back(b);
53         blockIndex++;
54     }
55
56     return basic_blocks;
57 }

```

## 模块效果图

## 基本块划分测试：

0: 0 1 2 3 4 5 6 7 8 9 10

1: 11

2: 12 13 14 15 16 17 18 19 20 21 22 23 24

3: 25

## 4. 基本块优化

### 基本块的 DAG 构造算法

#### 1. 准备操作数的结点

如果  $NODE(B)$  无定义，则构造一标记为  $B$  的叶结点并定义  $NODE(B)$  为这个结点；

- 如果当前四元式是 0 型则记  $NODE(B)$  的值为  $n$ ，转 4。
- 如果当前四元式是 1 型，则转 2(a)
- 如果当前四元式是 2 型，则
  - 如果  $NODE(C)$  无定义，则构造一标记为  $C$  的叶结点并定义  $NODE(C)$  为这个结点；
  - 转 2(b)。

#### 2. 合并已知量（如果操作数都是常数，完成计算）

- 如果  $NODE(B)$  是标记为常数的叶结点，则转 2(c)；否则，转 3(a)。
- 如果  $NODE(B)$  和  $NODE(C)$  都是标记为常数的叶结点，则转 2(d)；否则，转 3(b)。
- 执行  $op\ B$  (即合并已知量)。令得到的新常数为  $P$ 。如果  $NODE(B)$  是处理当前四元式时新构造出来的结点，则删除它。如果  $NODE(P)$  无定义，则构造一用  $P$  作标记的叶结点  $n$ 。置  $NODE(P)=n$ ，转 4。
- 执行  $B\ op\ C$  (即合并已知量)。令得到的新常数为  $P$ 。如果  $NODE(B)$  或  $NODE(C)$  是处理当前四元式时新构造出来的结点，则删除它。如果  $NODE(P)$  无定义，则构造一用  $P$  作标记的叶结点  $n$ 。置  $NODE(P)=n$ ，转 4。

#### 3. 寻找公共子表达式（操作数不全是常数，完成计算）

- 检查 DAG 中是否已有一结点，其唯一后继为  $NODE(B)$  且标记为  $op$  (即公共子表达式)。如果没有，则构造该结点  $n$ ，否则，把已有的结点作为它的结点并设该结点为  $n$ 。转 4。
- 检查 DAG 中是否已有一结点，其左后继为  $NODE(B)$ ，右后继为  $NODE(C)$ ，且标记为  $op$  (即公共子表达式)。如果没有，则构造该结点  $n$ ，否则，把已有的结点作为它的结点并设该结点为  $n$ 。转 4。

#### 4. 删除无用赋值（完成赋值）

如果  $NODE(A)$  无定义，则把  $A$  附加在结点  $n$  上并令  $NODE(A)=n$ ；否则，先把  $A$  从  $NODE(A)$  结点上的附加标识符集中删除（注意，如果  $NODE(A)$  是叶结点，则其  $A$  标记不删除）。把  $A$  附加到新结点  $n$  上并置  $NODE(A)=n$ 。转处理下一四元式。

C++

```
1 void DAGGraph::constructbyTac(Quaternary& quat)
2 {
```

```

2
3     string op = quat.op;
4     string A = quat.res;
5     string B = quat.arg1;
6     string C = quat.arg2;
7     if (isalpha(op[0])) //不对这样的四元式进行处理
8         return;
9     int resNode; //最终结果的结点号
10    int argNode1, argNode2; //操作数的结点号
11    argNode1 = getNode(B);
12    if (argNode1 == -1) //若B未定义
13        argNode1 = createLeafNode(B); //构造标记为B的叶结点，并定义
14    NODE(B)
15    {
16        if (op != "!=") //1类
17        {
18            //2(1)
19            if (ifNodeConstLeaf(argNode1)) //NODE(B)为常数叶结
20                {
21                    //2(3)
22                    resNode = excuteOP_1(argNode1, op); //合
23                }
24                else
25                {
26                    //3(1)
27                    resNode = FindNode(op, argNode1); //找公
28                    if (resNode == -1) //没有找到
29                    {
30                        resNode = createInnerNode(op, argNode1);
31                    }
32                }
33            }
34            else //0类
35            {
36                resNode = argNode1;
37            }
38        }
39        else //2类
40        {
41            argNode2 = getNode(C);
42            if (argNode2 == -1) //若C未定义
43                argNode2 = createLeafNode(C); //构造标记为C的叶结
44            //2(2)
45            if (ifNodeConstLeaf(argNode2) && ifNodeConstLeaf(argNode1))

```

```

//NODE(B)和NODE(C)都是常数叶结点
46         {
47             //2(4)
48             resNode = excuteOP_2(argNode1, argNode2, op);
49         }
50     else
51     {
52         //3(2)
53         resNode = FindNode(op, argNode1, argNode2);           //找
        公共子表达式
54         if (resNode == -1)           //没有找到
55         {
56             resNode = createInnerNode(op, argNode1,
        公共子表达式
        argNode2);
57         }
58     }
59 }
60 //4
61 resIdentifierAdd(A, resNode);
62 }

```

## 由 DAG 输出四元式算法

1. 依次遍历DAG图中的每一个结点，生成对应的四元式
2. 若结点是叶结点，以“=”为操作码，结点标记为第一操作数，附加标识符集中的元素为目的操作数生成四元式。
3. 若结点是内部结点，以结点标记为操作码，结点的左右子结点为操作数，附加标识符集中的第一个元素为目的操作数生成四元式；若附加标识符集中还有其他元素，则以“=”为操作码，集合中的第一个元素为第一操作数，其他元素为目的操作数生成四元式。

## 模块效果图

```
E:\Project2019\DAG\Debug\DAG.exe
Input:
:= 3.14 # T0
* 2 T0 T1
+ R r T2
* T1 T2 A
:= A # B
* 2 T0 T3
+ R r T4
* T3 T4 T5
- R r T6
* T5 T6 B

result:
+ R r T4
* 6.28 T4 A
- R r T6
* A T6 B
请按任意键继续. . .
```

## 5. 查找循环的不变运算

### 查找循环中不变运算的算法

预处理：需要判定一个基本块是否是一个循环体

1. 依次查看基本块L的每个四元式，如果它的每个运算对象或为常数，或者定值点在L外，则将此四元式标记为“不变运算”；
2. 重复第3步直至没有新的四元式被标记为“不变运算”为止；
3. 依次查看尚未被标记为“不变运算”的四元式，如果它的每个运算对象或为常数，或定值点在L外，或只有一个到达定值点且该点上的四元式已被标记为“不变运算”，则把被查看的四元式标记为“不变运算”。

C++

```
1
2 // 循环不变运算
3 vector<BasicBlock> getUnchangeOperation(vector<Quaternary> all_qua , vector<BasicBlockStandbyActiveTable> basic_block_standby_active_tables, vector<BasicBlock> & basic_blocks)
4 {
5     // 寻找循环的基本块
6     vector<BasicBlock> res_basic_blocks = findLoop(all_qua, basic_blocks);
7
8     // 计算每一个基本块中的循环不变运算
9     for (int i = 0; i < res_basic_blocks.size(); i++)
10     {
11         // 对循环的基本块中寻找不变运算
12         if (res_basic_blocks[i].loop == true)
13         {
```

```

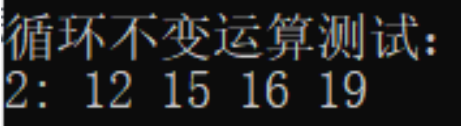
14
15 // 获取基本块的四元式集合以及非基本块内的四元式集合
16 vector<Quaternary> b_qua;
17 for(int j = 0; j < res_basic_blocks[i].quaList.size(); j
++))
18 {
19     b_qua.push_back(all_qua[res_basic_blocks[i].qual
ist[j]]);
20 }
21 // 获取基本内部的定值点集合
22 vector<string> b_node;
23 for (int i = 0; i < b_qua.size(); i++)
24 {
25     b_node.push_back(b_qua[i].res);
26 }
27 // 判断循环不变运算
28 for(int j = 0; j < b_qua.size(); j ++))
29 {
30     Quaternary q = b_qua[j];
31
32     if(q.op == "j")
33         continue;
34
35     // 两个操作数都是常数，并且在内部没有定值点(被重新复
制)
36     if(isdigit(q.arg1[0]) && isdigit(q.arg2[0]) && i
sInVector(q.res, b_node) == 1)
37     {
38         res_basic_blocks[i].unchangeOpList.push_
back(q.id);
39     }
40     // 一个是常数，另一个没有被重新定值
41     else if((isdigit(q.arg1[0]) && isInVector(q.arg
2, b_node) == 0 && isInVector(q.res, b_node) == 1) || (isdigit(q.arg2[0]) && is
InVector(q.arg1, b_node) == 0 && isInVector(q.res, b_node) == 1))
42     {
43         res_basic_blocks[i].unchangeOpList.push_
back(q.id);
44     }
45     // 两个操作数都没有被重新定值
46     else if(isInVector(q.arg1, b_node) == 0 && isInv
ector(q.arg2, b_node) == 0 && isInVector(q.res, b_node) == 1 && q.res != q.arg
1 && q.res != q.arg2)
47     {
48         res_basic_blocks[i].unchangeOpList.push_
back(q.id);
49     }
50 }

```



```
51         }  
52     }  
53  
54     return res_basic_blocks;  
55 }
```

### 模块效果图



循环不变运算测试:  
2: 12 15 16 19

## 6. 代码外提

代码外提主要是根据上述不变运算的基础上，对所有的不变运算进行筛选，选出那些真正可以被外提的代码，于是我们根据若干个条件对不变运算进行判定。

我们可以对所有条件进行概括：运算s的代码 $A := B \text{ op } C$ 如果可以外提总共要满足的条件一共有四个（前提条件该运算是不变运算）：

- ① s所在结点是L所有出口结点的必经结点；
- ② A在 L中的其他地方并未定值；
- ③ L中所有A的引用点中只有s中A的定值才能够到达；
- ④ A在离开L之后不再是活跃的

在这四个条件中，②和③是一定要满足的，①和④必须至少要满足其中一个。

判定条件①的代码为：

## OpenGL Shading Language

```
1  int MustNode(int round, const vector<BasicBlock>& BB)
2  {
3      // 先找循环启示位置
4      // 再找循环终止位置
5      int end_loop = round + 1;
6      for (int start_not_loop=round+1;; start_not_loop++)
7      {
8          if (BB[start_not_loop].loop == false)
9          {
10             end_loop = start_not_loop;
11             break;
12          }
13      }
14      if (end_loop != round + 1)
15      {
16          return -1;
17      }
18      return 1;
19  }
```

判定条件②的代码为：

## C++

```
1  int OnlyValue(int qNum, const vector<BasicBlock>& BB, const vector<Quaternary>&
   Q, int round)
2  {
3      string res_target = Q[qNum].res;
4
5      for (int i=0;i<BB[i].quaList.size();i++)
6      {
7          string res_temp = Q[BB[i].quaList[i]].res;
8          if (BB[i].quaList[i]!=qNum && res_temp==res_target)
9          {
10             return -1;
11          }
12      }
13
14      // 先找循环启示位置
15      int start_loop = round - 1;
16      for (int start_to_loop = round - 1;; start_to_loop--)
17      {
18          if (BB[start_to_loop].loop == false)
19          {
20             start_loop = start_to_loop + 1; // 循环终止即可，无需+1
21             break;
22          }
23      }
```

```

21         break;
22     }
23 }
24
25 // 再找循环终止位置
26 int end_loop = round + 1;
27 for (int start_not_loop = round + 1;; start_not_loop++)
28 {
29     if (BB[start_not_loop].loop == false)
30     {
31         end_loop = start_not_loop - 1; // 循环终止即可，无需+1
32         break;
33     }
34 }
35
36 // 查看循环体之间，即L内的情况
37 for (int i=0;i<BB.size();i++)
38 {
39     if (BB[i].block_id>=start_loop && BB[i].block_id<=end_loop)
40     {
41         vector<int> quaList_temp = BB[i].quaList;
42         for (int j = 0; j < quaList_temp.size(); j++)
43         {
44             string res_temp = Q[quaList_temp[j]].res;
45             if (quaList_temp[j] != qNum && res_temp ==
res_target)
46             {
47                 return -1;
48             }
49         }
50     }
51 }
52
53 return 1;
54 }

```

判定条件③的代码为：

## OpenGL Shading Language

```
1 // 条件1-3
2 int OnlyAccess(int qNum, const vector<BasicBlock>& BB, const vector<Quaternary>&
  Q, int round)
3 {
4     string res_target = Q[qNum].res;
5     int block_target = round;
6
7     // 先找循环启示位置
8     int start_loop = round - 1;
9     for (int start_to_loop = round - 1;; start_to_loop--)
10    {
11        if (BB[start_to_loop].loop == false)
12        {
13            start_loop = start_to_loop + 1; // 循环终止即可, 无需+1
14            break;
15        }
16    }
17
18    // 再找循环终止位置
19    int end_loop = round + 1;
20    for (int start_not_loop = round + 1;; start_not_loop++)
21    {
22        if (BB[start_not_loop].loop == false)
23        {
24            end_loop = start_not_loop - 1; // 循环终止即可, 无需+1
25            break;
26        }
27    }
28    .....
```

判定条件④的代码为：

## C++

```
1 int NotActive(int qNum, const vector<BasicBlock>& BB, const vector<Quaternary>&
  Q, int round)
2 {
3     string res_target = Q[qNum].res;
4
5     int start_notloop = round + 1; // 默认出口基本块块号
6
7     for (int find_notloop=round+1;;find_notloop++)
8     {
9         if (BB[find_notloop].loop==false)
10        {
11            start_notloop = find_notloop;
```

```

11         start_notloop = find_notloop;
12         break;
13     }
14 }
15
16 int lookfor = start_notloop; // 查看的基本块编号，根据lookfor进行循环
17 for (;;)
18 {
19     vector<int> quaList_lookfor = BB[lookfor].quaList;
20     for (int i=0;i<quaList_lookfor.size();i++)
21     {
22         if (Q[quaList_lookfor[i]].arg1 == res_target ||
23             Q[quaList_lookfor[i]].arg2 == res_target)
24         {
25             return -1;
26         }
27     }
28     Quaternary end_qua = Q[BB[lookfor].quaList.back()];
29     string::size_type idx_jump = end_qua.op.find("j");
30     if (idx_jump != string::npos)
31     {
32         int lookfor_q = stoi(end_qua.res);
33         lookfor = findBasic(lookfor_q,BB);
34         continue;
35     }
36     else
37     {
38         lookfor += 1;
39         if ((lookfor+1)>BB.size())
40         {
41             return 1;
42         }
43         continue;
44     }
45 }
46 }

```

## 模块效果图

代码外提运算测试：

```

0: 0 1 2 3 4 5 6 7 8 9 10
2: 11
3: 12 13 14 15 16 17 18 19 20 21 22 23 24
4: 25
1: 12 15 16 19

```

## 7. 强度削弱

### 强度削弱的算法

预处理：需要判定一个基本块是否是一个循环体

1. 遍历所有基本块，判断该基本块是否是一个循环体，若是，则转2，否则进行下一次判断；
2. 遍历基本块中的四元式，根据循环不变量的信息去寻找基本归纳变量，找到后做以下两步操作：
  - a. 将找到的基本归纳变量用一个hash\_set存下来，便于后面查找；
  - b. 将该基本归纳变量及其对应的四元式用一个hash\_set存下来，便于后续进行线性关系的变换；
3. 判断是否有新的基本归纳变量产生（由2或3中的强度削弱产生），若是，进行如下循环：
  - a. 遍历基本块，根据基本归纳变量的信息去寻找归纳变量，对找到的归纳变量通过如下操作进行强度削弱：
    - i. 根据线性关系，获取新的操作数和运算符；
    - ii. 生成强度削弱后的新四元式；
    - iii. 将新的四元式放到本基本块的倒数第二个位置上；
    - iv. 将原来的四元式移到第n-2个基本块的末尾（设当前基本块为n）；
    - v. 更新基本归纳变量信息：用新生成的基本归纳变量与其四元式对应关系替换掉原来hash\_set与hash\_map中的内容，并将标志位置为true，表示还要再次进行基本块遍历；

C++

```
1 void strengthWeakening(vector<BasicBlock>& all_block, vector<Quaternary>& qua) {
2     // 判断是否是数字的函数
3     bool isnum(string s);
4     // 存放循环不变量
5     unordered_set<string> inVariables;
6     // 存放基本归纳变量
7     unordered_set<string> basIndVariables;
8     // 存放新的基本归纳变量
9     unordered_set<string> newBasIndVariables;
10    // 存放归纳变量
11    unordered_set<string> indVariables;
12    // 存放归纳变量与四元式对应关系
13    unordered_map<string, Quaternary> varToCode;
14    // 存放新归纳变量与四元式对应关系
15    unordered_map<string, Quaternary> newVarToCode;
16    // 判断是否还要继续循环强度削弱
17    bool isContinue = true;
18    //判断一个基本块是否是循环基本块
19    for (int i = 0; i < all_block.size(); i++)
20    {
21        if (all_block[i].loop)
22        {
23            // 保存基本块内部四元式id
24            vector<int> codeIndex = all_block[i].quaternary;
```

```

24         vector<int> codeIndex = all_block[i].qualist;
25
26
27         // 将该循环体内部的循环不变量放到一个hash_set中，便于后续判断
28         for (int j = 0; j < all_block[i].unchangeOpList.size();
j++)
29         {
30             int codeId = all_block[i].unchangeOpList[j];
31             inVariables.insert(qua[codeId].res);
32         }
33
34
35         // 判断一个变量是否是基本归纳变量
36         for (int k = 0; k < codeIndex.size(); k++)
37         {
38             Quaternary code = qua[codeIndex[k]];
39
40             // 判断该变量是否是基本归纳变量：第一操作数等于目的操
作数 && 第二操作数是循环不变量 && 运算符为+/-
41             if (code.arg1 == code.res && (isnum(code.arg2) |
| (inVariables.find(code.arg2) != inVariables.end())) && (code.op == "+" || cod
e.op == "-")) {
42
43                 // 若是，则加入基本归纳变量数组中，并把对应的
基本归纳变量-四元式关系放入hash_map中
44                 basIndVariables.insert(code.res);
45                 varToCode[code.res] = code;
46             }
47         }
48         // 遍历一次基本块后，由于进行强度虚弱后会产生新的基本归纳变量，
且原来的基本归纳变量不能在使用，所以需要根据是否产生新的基本归纳变量判断是否要再次遍历基本
块
49         while (isContinue == true) {
50             isContinue = false;
51             // 判断一个变量是否是归纳变量，若是，则对其进行强度削
弱，强度削弱后的形式为： $T:=T+/-C$ ，C为循环不变量
52             for (int k = 0; k < codeIndex.size(); k++)
53             {
54                 Quaternary code = qua[codeIndex[k]];
55                 bool judge = false;
56                 // 定义新的第二操作数
57                 string newArg2;
58                 // 获取基本归纳变量所属四元式部分信息
59                 string preArg2;
60                 string preOp;
61                 string newOp;
62                 // 情况一： $T:=K*I$ ，K为循环不变量，I为基本归纳
变量
63                 if (code.op == "*" && (basIndVariables.f

```

```

ind(code.arg2) != basIndVariables.end()) && ((inVariables.find(code.arg1) != inV
ariables.end()) || isnum(code.arg1)))
64         {
65             preArg2 = varToCode[code.arg2].a
rg2;
66             preOp = varToCode[code.arg2].op;
67             judge = true;
68             // 计算新的第二操作数，分常数和基本
归纳变量两种情况
69             if (isnum(code.arg1) && isnum(pr
eArg2)) {
70                 newArg2 = to_string(stoi
(code.arg1) * stoi(preArg2));
71             }
72             else {
73                 newArg2 = code.arg1 +
"*" + preArg2;
74             }
75             newOp = preOp;
76
77         }
78         //情况二:  $T:=I+/-C$ ,  $C$ 为循环不变量,  $I$ 为基本归
纳变量
79         else if ((code.op == "+" || code.op ==
"-") && (basIndVariables.find(code.arg1) != basIndVariables.end()) && ((inVariab
les.find(code.arg2) != inVariables.end()) || isnum(code.arg2)) && (basIndVariabl
es.find(code.res)==basIndVariables.end())) {
80             preArg2 = varToCode[code.arg1].a
rg2;
81             preOp = varToCode[code.arg1].op;
82             judge = true;
83             // 判断操作数是否是数字
84             if (isnum(preArg2)) {
85                 // 计算新的第二操作数
86                 switch (preOp[0])
87                 {
88                     case '+': {
89                         newArg2 = to_str
ing(0 + stoi(preArg2));
90                         break;
91                     }
92                     case '-': {
93                         newArg2 = to_str
ing(0 - stoi(preArg2));
94                         break;
95                     }
96                 }
97                 if (stoi(newArg2) >= 0)

```



```

98         {
99             newOp = code.op;
100         }
101         else {
102             newOp = code.op
103         }
104     }
105     else {
106         newArg2 = preArg2;
107         newOp = preOp;
108     }
109 }
110 if (judge == true) {
111     isContinue = true;
112     // 产生强度优化后的代码
113     int newCodeId = qua.size();
114     Quaternary newCode = { newCodeI
115 d,newOp,code.res,newArg2,code.res };
116     // 将强度优化后的代码加入到qua的尾
117     qua.push_back(newCode);
118     // 删除本基本块中该四元式的信息
119     all_block[i].quaList.erase(find
120 (all_block[i].quaList.begin(), all_block[i].quaList.end(), codeIndex[k]));
121     // 将原来的四元式提到第n-2个基本块
122     中
123     if (i - 2 >= 0) {
124         for (int l = 0; l < all_
125             block.size(); l++) {
126                 if (all_block
127                     [l].block_id == i - 2)
128                     {
129                         all_bloc
130                         k[l].quaList.push_back(codeIndex[k]);
131                     }
132             }
133         else {
134             for (int l = 0; l < all_
135                 block.size(); l++) {
136                     all_block[l].blo
137                     ck_id++;
138                 }
139             vector<int> newQuaList;

```

```

136                                     newQualist.push_back(cod
eIndex[k]);
137                                     BasicBlock b = { 0,newQu
aList,false };
138                                     all_block.push_back(b);
139                                     }
140
141                                     //将新生成的代码加入本基本块末尾
142                                     int lastCode = all_block[i].quaL
ist[all_block[i].qualist.size()-1];
143                                     all_block[i].qualist[all_block
[i].qualist.size()-1] = newCodeId;
144                                     all_block[i].qualist.push_back(l
astCode);
145
146                                     //将新生成的基本归纳变量加入到新基本
归纳变量数组
147                                     newBasIndVariables.insert(code.r
es);
148                                     newVarToCode[code.res] = newCod
e;
149                                     }
150                                     }
151                                     //替换归纳变量
152                                     basIndVariables = newBasIndVariables;
153                                     varToCode = newVarToCode;
154                                     }
155                                     }
156                                     }
157     }

```

## 模块效果图

## 强度削弱测试

```
0: := 3.14 # T0
1: * 2 T0 T1
2: + R r T2
3: * T1 T2 A
4: := A # B
5: * 2 T0 T3
6: + R r T4
7: * T3 T4 T5
8: - R r T6
9: * T5 T6 B
10: := 1 # I
11: j> I 10 25
12: * 2 J T1
13: * 10 I T2
14: + T2 T1 T3
15: - A 11 T4
16: * 2 J T5
```

## 四、实验结果

测试样例输出结果：

```
读入文件完毕
四元式集合读入测试：
0: := 3.14 # T0
1: * 2 T0 T1
2: + R r T2
3: * T1 T2 A
4: := A # B
5: * 2 T0 T3
6: + R r T4
7: * T3 T4 T5
8: - R r T6
9: * T5 T6 B
10: := 1 # I
11: j> I 10 25
12: * 2 J T1
13: * 10 I T2
14: + T2 T1 T3
15: - A 11 T4
16: * 2 J T5
17: * 10 I T6
18: + T6 T5 T7
19: - A 11 T8
20: =[] T8 T7 T9
21: + T9 1 X1
22: []= X1 T4 T3
23: + I 1 I
24: j # # 11
25: halt # # #
```

基本块划分测试：

0: 0 1 2 3 4 5 6 7 8 9 10

1: 11

2: 12 13 14 15 16 17 18 19 20 21 22 23 24

3: 25

四元式集合的待用信息和活跃信息表：

0: (1, 1) (-1, 0) (4, 1)

1: (3, 1) (-1, 0) (5, 1)

2: (3, 1) (6, 1) (6, 1)

3: (4, 1) (-1, 0) (-1, 0)

4: (-1, 0) (15, 1) (10, 1)

5: (7, 1) (-1, 0) (-1, 0)

6: (7, 1) (8, 1) (8, 1)

7: (9, 1) (-1, 0) (-1, 0)

8: (9, 1) (-1, 0) (-1, 0)

9: (-1, 0) (-1, 0) (-1, 0)

10: (11, 1) (-1, 0) (24, 1)

11: (-1, 0) (13, 1) (-1, 0)

12: (14, 1) (-1, 0) (16, 1)

13: (14, 1) (-1, 0) (17, 1)

14: (-1, 0) (-1, 0) (-1, 0)

15: (22, 1) (19, 1) (-1, 0)

16: (18, 1) (-1, 0) (-1, 0)

17: (18, 1) (-1, 0) (23, 1)

18: (20, 1) (-1, 0) (-1, 0)

19: (20, 1) (-1, 0) (-1, 0)

20: (21, 1) (-1, 0) (-1, 0)

21: (22, 1) (-1, 0) (-1, 0)

22: (-1, 0) (-1, 0) (-1, 0)

23: (-1, 0) (-1, 0) (-1, 0)

24: (-1, 0) (25, 1) (24, 1)

25: (-1, 0) (-1, 0) (25, 1)

## 五、总结与思考

### 1.实验小结

通过本实验，本小组得以将课程所学知识与动手实践相结合，既深入理解了优化部分的原理，又提升了自己的编码能力，收获良多。这次实验还让我们了解到了编译器的部分工作过程，加深了本小组成员对编码过程的

### 2.实验亮点

## 2.1 系统化设计

本实验根据已给的示例，完成了输入三地址代码、解析四元式、基本块划分、局部优化、循环优化等等一系列的操作，在操作中流程分的非常清楚，代码思路也非常明了。

## 2.2 数据结构

本实验针对四元式、DAG图、DAG结点等部分设计了非常详细的数据结构，具体内容已在上述说明完毕。

## 2.2 细节部分

本实验从示例出发，对实际三地址代码等内容的很多细节部分都做了解析，同样也在上述进行展示完成。

## 2.2 拓展内容

本实验除了老师所要求的部分之外，我们还额外完成了四元式集合的代用信息和活跃信息表，如下图：

四元式集合的待用信息和活跃信息表：

```
0: (1, 1) (-1, 0) (4, 1)
1: (3, 1) (-1, 0) (5, 1)
2: (3, 1) (6, 1) (6, 1)
3: (4, 1) (-1, 0) (-1, 0)
4: (-1, 0) (15, 1) (10, 1)
5: (7, 1) (-1, 0) (-1, 0)
6: (7, 1) (8, 1) (8, 1)
7: (9, 1) (-1, 0) (-1, 0)
8: (9, 1) (-1, 0) (-1, 0)
9: (-1, 0) (-1, 0) (-1, 0)
10: (11, 1) (-1, 0) (24, 1)
11: (-1, 0) (13, 1) (-1, 0)
12: (14, 1) (-1, 0) (16, 1)
13: (14, 1) (-1, 0) (17, 1)
14: (-1, 0) (-1, 0) (-1, 0)
15: (22, 1) (19, 1) (-1, 0)
16: (18, 1) (-1, 0) (-1, 0)
17: (18, 1) (-1, 0) (23, 1)
18: (20, 1) (-1, 0) (-1, 0)
19: (20, 1) (-1, 0) (-1, 0)
20: (21, 1) (-1, 0) (-1, 0)
21: (22, 1) (-1, 0) (-1, 0)
22: (-1, 0) (-1, 0) (-1, 0)
23: (-1, 0) (-1, 0) (-1, 0)
24: (-1, 0) (25, 1) (24, 1)
25: (-1, 0) (-1, 0) (25, 1)
```

## 3.平台使用反馈

本项目暂未使用阿里云平台。

## 4.课程建议

本实验项目对我们巩固课程内容是有非常大的帮助的，因为我们在代码实现过程中一定对于原理知识有非常多的理解