

一.主要功能

利用 DAG 来优化中间代码，即四元式，去除只定义而未使用的变量以及语句块中能够合并的赋值语句。

二.代码实现

主要添加了 infixOpt.cpp 文件，对于中间代码四元式部分进行优化处理

重要数据结构：

```
struct DAGNode {
    int number;//节点的唯一标识，与lastNum有关
    string content;//操作
    std::set<DAGNode*> parents;//父节点塞入
    DAGNode* leftChild;
    DAGNode* rightChild;
};
```

主要函数：

```
void optimizeInfixes() {
    splitBlocks();获得 crossing variable， 在多个块中总共出现不止一次的变量
    划分基本块，在基本块中绘制 DAG 树并执行优化
    重置 infix table 中的变量
    输出到文件
}
```

// Export optimized codes from DAG & varNodeTable

```
void exportCodesFromDAG() {
    获得 root node
    节点 node 记录到 calculationQueue，把 Node 从子节点中的 parentNode 中删去，并对左右
    子节点检测是否是 root，进行相同操作
    Output var with initials e.g. a0 = a，varNodetable 中的变量和 varsWithInitial 的变量比较，将
    变化了的变量 a +0,并 Insert a0 to static table，并插入新的 infixtable 中，即
    insertNewInfix(), 改变 allnodes[]中的变量名
    比较 calculationQueue 和 varNodetable 中 node 的值，用 varNodes 记录 varNode 中与
    calculation 中值一致的变量名
    建立两个 set<string> varsTostay 和 varsToleave，将 crossing variable 与 varNode
    比较，将相同变量的变量名添加到 varsTostay 中，
    如果一个没有，即 varsToStay.size() == 0 取 varNode 第一个 var 到 varsTostay,并添加
    infixtable，Insert new infix notation
    否则，选择第一个元素为运算的结果并使用 assign 赋值}
}
```

三.测试结果

Input code	中间代码（四元式）	优化后的中间代码
int main(){ int T0,T1,T2,T3,T4,T5,T6; int A,B; int R,r; R=2; r=3; T0=5; T1=2*T0;	int main() int T0 int T1 int T2 int T3 int T4 int T5 int T6	int main() int T0 int T1 int T2 int T3 int T4 int T5 int T6

<pre> T2=R+r; A=T1*T2; B=A; T3=2*T0; T4=R+r; T5=T3*T4; T6=R-r; B=T5*T6; printf(B); return; } </pre>	<pre> int A int B int R int r R = 2 r = 3 T0 = 5 #t0 = 2 * T0 T1 = #t0 #t1 = R + r T2 = #t1 #t2 = T1 * T2 A = #t2 B = A #t3 = 2 * T0 T3 = #t3 #t4 = R + r T4 = #t4 #t5 = T3 * T4 T5 = #t5 #t6 = R - r T6 = #t6 #t7 = T5 * T6 B = #t7 print B return </pre>	<pre> int A int B int R int r r = 3 R = 2 #t6 = R - r #t1 = R + r T0 = 5 #t0 = 2 * T0 #t2 = #t0 * #t1 B = #t2 * #t6 print B return </pre>
---	--	---

四. Debug 以及 C++知识点

1.DAG 原理

<https://max.book118.com/html/2016/0712/48008065.shtml>

1. 基本块的划分

入口语句的定义如下：

- ① 程序的第一个语句；或者，
- ② 条件转移语句或无条件转移语句的转移目标语句；
- ③ 紧跟在条件转移语句后面的语句。

有了入口语句的概念之后，就可以给出划分中间代码（四元式程序）为基本块的算法，其步骤如下：

① 求出四元式程序中各个基本块的入口语句。

② 对每一入口语句，构造其所属的基本块。它是由该入口语句到下一入口语句（不包括下一入口语句），或到一转移语句（包括该转移语句），或到一停语句（包括该停语句）之间的语句序列组成的。

③ 凡未被纳入某一基本块的语句、都是程序中控制流程无法到达的语句，因而也是不会被执行到的语句，可以把它们删除。

2. 基本块的优化手段

由于基本块内的逻辑清晰，故而要做的优化手段都是较为直接浅层次的。目前基本块内的常见的块内优化手段有：


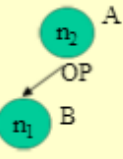
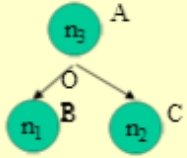
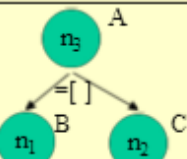
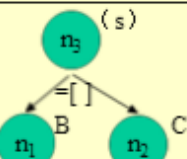
1. 删除公共子表达式

2. 删除无用代码

3. 重新命名临时变量（一般是用来应对创建过多临时变量的，如 $t2 := t1 + 3$ 如果后续并没有对 $t1$ 的引用，则可以 $t1 := t1 + 3$ 来节省一个临时变量的创建）

4. 交换语句顺序

5. 在结果不变的前提下，更换代数操作（如 $x := y^{**}2$ 是需要根据 $**$ 运算符重载指数函数的，这是挺耗时的操作，故而可以用强度更低的 $x := y*y$ 来代替）

类型	四元式	DAG结点	说明
0型	$(=, B, A)$		把B赋给变量A，即A、B具有同样的值。无条件转向语句也可这样表示。
1型	(OP, B, A)		OP是单目运算符，与0型类似
2型	(OP, B, C, A)		B、C为两个叶结点，OP为运算符，运算结果赋给内部结点右边的标记A
3型	$(=[], B, C, A)$		B是数组，C是数组下标地址， $=[]$ 表示对数组B中下标变量地址为C的元素进行运算，结果赋给变量A
4型	$(JROP, B, C, (s))$		运算的结果将转向内部结点右边标出的语句(s)

对基本块的每一四元式，依次执行：

1. 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. (1)。

如果当前四元式是 2 型，则：(I) 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，(II) 转 2. (2)。

2.

(1) 如果 NODE (B) 是标记为常数的叶结点，则转 2. (3)，否则转 3. (1)。

(2) 如果 NODE (B) 和 NODE (C) 都是标记为常数的叶结点，则转 2. (4)，否则转 3. (2)。

(3) 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 做标记的叶结点 n。置 NODE (P) = n，转 4。

(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 做标记的叶结点 n。置 NODE (P) = n，转 4。

3.

(1) 检查 DAG 中是否已有一结点，其唯一后继为 NODE (B)，且标记为 op (即找公共子表达式)。如果没有，则构造该结点 n，否则就把已有的结点作为它的结点并设该结

点为 n ，转 4。

(2) 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 (即找公共子表达式)。如果没有，则构造该结点 n ，否则就把已有的结点作为它的结点并设该结点为 n 。转 4。

4 .

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，则其标记 A 不删除)，把 A 附加到新结点 n 上并令 $NODE (A) = n$ 。转处理下一四元式。

(1) $T_0 := 3.14$

(2) $T_1 := 2 * T_0$

(3) $T_2 := R + r$

(4) $A := T_1 * T_2$

(5) $B := A$

(6) $T_3 := 2 * T_0$

(7) $T_4 := R + r$

(8) $T_5 := T_3 * T_4$

(9) $T_6 := R - r$

(10) $B := T_5 * T_6$

1) $(=, 3.14, T_1)$

2) $(*, 2, T_1, T_2)$

3) $(+, R_1, R_2, T_3)$

4) $(*, T_2, T_3, A)$

5) $(=, A, B)$

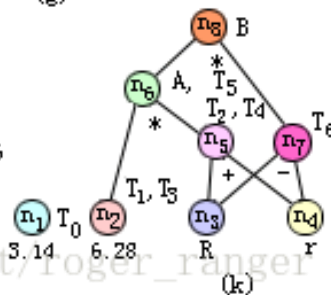
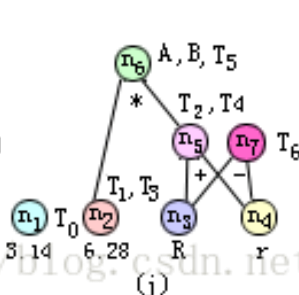
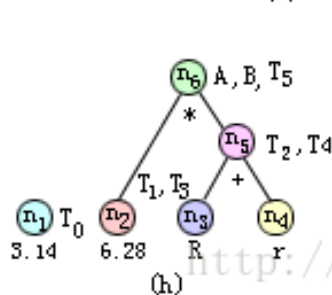
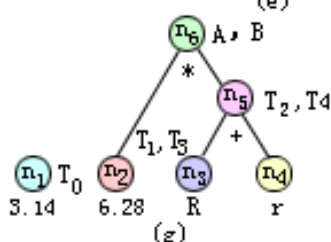
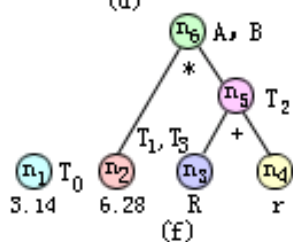
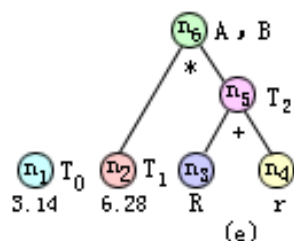
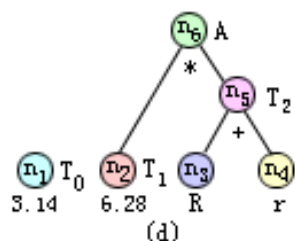
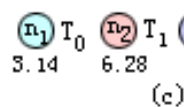
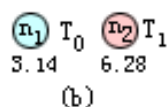
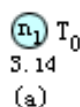
6) $(*, 2, T_1, T_4)$

7) $(+, R_1, R_2, T_5)$

8) $(*, T_4, T_5, T_6)$

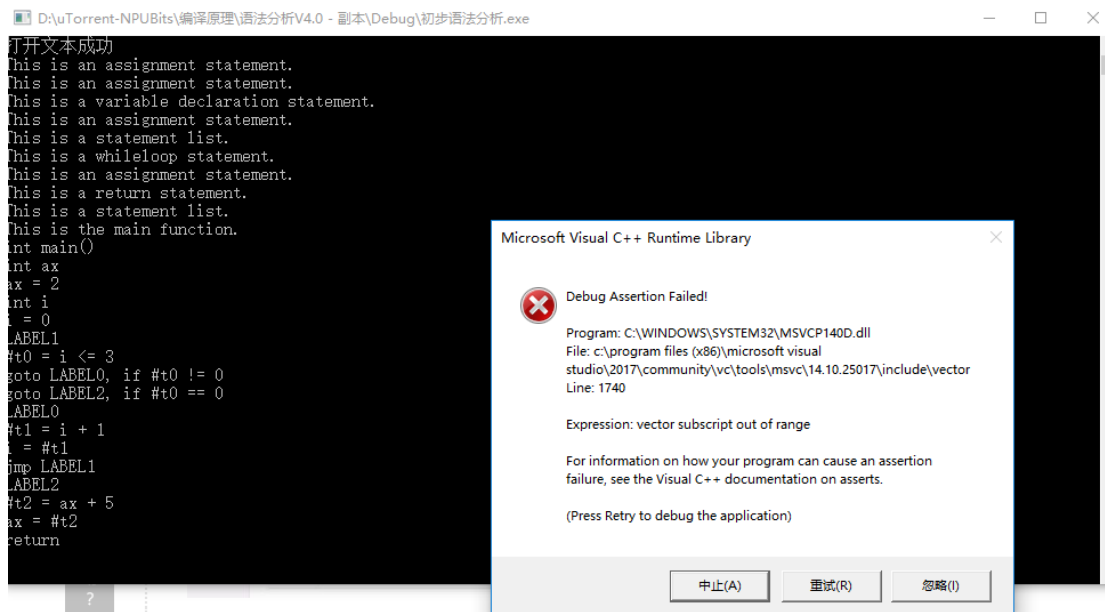
9) $(-, R_1, R_2, T_7)$

10) $(*, T_5, T_7, B)$

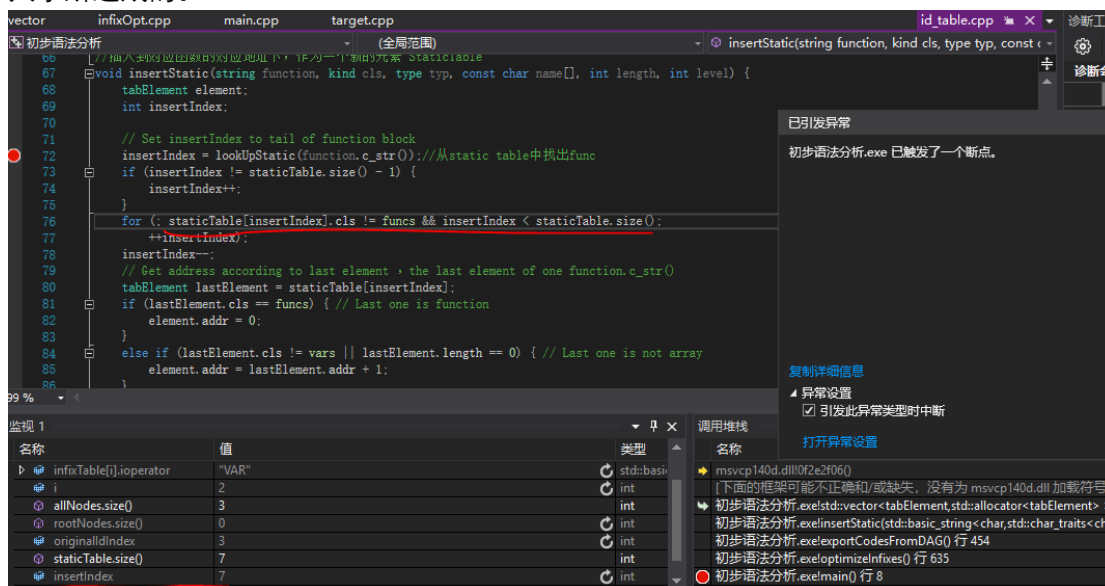


2.野指针错误

原程序中的中间代码优化添加之后，运行程序崩溃



总结来说这种错误存在两种情况，其一就是野指针，另一种情况就是内存泄露。通过一步一步对程序进行 DEBUG，发现这是由 vector 存放的数据超出了 vector 的大小所造成的。



该异常是由于 for 循环中的条件语句前后顺序不对，交换位置即可。如果不交换可能导致 insertIndex 超过 staticIndex.size () 的大小，在进行判断之前便在 vector 中查找，引起序号超出了 vector 的大小的异常。