

# 第8章 优化处理

在实际开发编译器时，对于前端技术，近年来一直比较稳定，以语法制导、形式语言自动机为基础，更多强调的是语法制导技术的翻译与分析，到中间代码生成为止，编译器基础的算法已经介绍完毕。对于后端技术，发展十分迅猛，是现代编译器工程实现中相当重要的部分。后端是一个从中间代码到目标代码的生成系统，由两部分构成：一是**优化处理**，给定中间代码，加快代码执行效率，减小空间占用；二是**目标代码生成**，在优化处理的基础上，生成精简高效的目标代码。前端决定了编译器能否实现，后端则决定了编译器是否可用。

本章主要介绍优化处理，优化是编译器中一个重要环节，优化的设计在整个编译器设计过程中占据相当大的比重，目的是**产生更高效的目标代码**。优化处理中包括大量数据结构、算法的优化，本书不做详细展开，本章主要介绍优化的基本思想和几种优化方法。

## 8.1 优化的分类

优化处理可以在不同层面上进行，可分为在源代码或中间代码级上进行的与机器无关的优化，以及在目标代码级上进行的与机器有关的优化。

### 8.1.1 与机器无关的优化

不采用与目标机相关的知识，如果优化在各个机器上都可以进行，就是与机器无关的优化。由于每一条中间代码指令都是一个单操作，在中间代码级上优化比较容易。在源代码级的优化相对困难，可以在编写程序时实现。

- **全局优化**：针对整个源程序，能达到更好的优化效果，实现难度更大。
- **局部优化**：除全局优化以外，处理局部的函数和程序块，更易实现。

### 8.1.2 与机器有关的优化

需要考虑目标机的性质。目标代码生成是从四元式生成目标代码，如果不考虑优化，是一一对一的映射过程，是有模板的映射，而考虑优化则需要处理大量问题。

- **寄存器分配优化**：现代 CPU 的设计，包括操作系统和编译器，都要考虑目标机关于存储器

的分配。寄存器访问速度快，运算效率高，但资源有限，设计相关算法合理分配寄存器能够大大优化效率。

- **消除无用代码优化：**例如一些无用跳转，可以依据目标机进行。

以上这两类优化处理方法中，主要介绍局部优化（第八章）和寄存器分配优化（第九章）。

## 8.2 常见的几种局部优化方法

### 8.2.1 常值表达式节省（常数合并）

如： $a = 5 + 3; b = a + 1; \dots$

等号右边的表达式  $5 + 3$  和  $a + 1$  都是常值表达式， $a$  和  $b$  的值不需要在程序运行时进行计算，编译器会将其优化为  $a = 8; b = 9;$  的形式，这样的优化被称为**常值表达式节省**。这种方法看似简单，却十分关键，节省了大量无用计算，若没有这种技术，现代计算机会慢 10 倍，编写的程序中需要进行大量常数合并，是一种常见的优化形式。

注：若  $a = 5 + 3; \dots; a = x \dots; a = a + 1; a = 5 + 3$  中的  $a$  是常数，后  $a$  经过赋值， $a = a + 1$  中的  $a$  不再是常数， $a + 1$  不是常值表达式。

### 8.2.2 公共表达式节省（删除多余运算）

如： $a = b * d + 1; e = b * d - 2; \dots$

等号右边的两个表达式中都有  $b * d$ ，且  $b$  和  $d$  的值在使用前后没有被修改，则  $b * d$  是公共表达式，不用重复计算，可以优化为  $t = b * d; a = t + 1; e = t - 2;$  的形式，减少了一次乘法，增加了一次赋值，乘法的计算代价高于赋值，这样的形式达到了优化的目的。

注：若  $b = b * d + 1; e = b * d - 2;$  就不能再使用这样的方法，因为第一个赋值表达式中，修改了  $b$  的值，公共表达式要求表达式中的变量不变，所以此处的  $b * d$  不是公共表达式。

### 8.2.3 删除无用赋值

如： $a = b + c; x = d - e; y = b; a = e - h/5;$

看似没有问题，仔细观察在  $a$  的两次赋值之间，没有对  $a$  的应用，则前一次对  $a$  的赋值  $a = b + c$  为无用赋值。可以将第一个式子删除，优化为  $x = d - e; y = b; a = e - h/5;$  的形式。

### 8.2.4 不变表达式外提（循环优化之一：把循环不变运算提到循环外）

如： $i = 1; \text{while}(i < 100) \ x = (k + a)/i; i ++;$

假设循环体中  $k$  和  $a$  的值都没有改变，则每一次循环中  $k + a$  的值都不变，将其称为循环不变表达式。如果每次循环中，都计算一遍  $k + a$ ，循环次数非常多时，计算代价将非常高。因

此，可以优化为  $i = 1; t = k + a; \text{while}(i < 100) \ x = t/i; \dots; i ++;$  的形式。

不变表达式外提也是一种常见的优化方法。如何通过算法实现找出不变表达式并外提，也是一个需要解决的问题。

### 8.2.5 消减运算强度（循环优化之二：把运算强度大的运算换算成强度小的运算）

如： $i = 1; \text{while}(i < 100) \ t = 4 * i; b = a \uparrow 2; i ++;$

幂运算强度大于乘除取模运算，乘除取模运算强度大于加减赋值运算。 $a \uparrow 2$  为  $a$  的平方，循环变量  $i$  每次加 1， $4 * i$  的结果为 4, 8, 12, ... 的等差数列，每次乘法的效果和每次加 4 的效果一致。因此，可以优化为  $i = 1; t = 4 * i; \text{while}(i < 100) \ t = t + 4; b = a * a; \dots; i ++;$  的形式。

## 8.3 局部优化算法探讨

通过以上例子，我们对优化有了一定感性的认识，接下来进一步系统地进行探讨。在编译器设计中，局部优化算法是以**基本块**为单位进行的，**基本块**也是目标代码生成的基本单位。

### 定义 8.1 基本块

程序中一段顺序执行的语句序列，其中只有一个入口和一个出口。

根据定义中的“顺序执行”、“一个入口，一个出口”，自然联想到，基本块的划分与条件判断以及跳转语句有关。执行一个基本块内的程序，只能从第一条顺序执行到最后一条，不存在其他入口或出口。满足这样要求的程序段称为一个独立的基本块。根据定义，只要找到基本块的入口和出口，就能找到基本块。

### 8.3.1 基本块划分算法

1. 找出基本块的入口语句：（以下为两种判断条件，满足其一即可）

- 程序的第一个语句或转向语句转移到的语句，四元式中的转向语句包括 goto、then、else (无条件跳)、do (循环体中)。
- 紧跟在转向语句后面的语句。

2. 对每一入口语句，构造其所属的基本块：

- 从该入口语句到另一入口语句之间的语句序列；
- 从该入口语句到另一转移语句（或停止语句）之间的语句序列。

**例 8.1** 条件语句四元式如下，给出基本块划分：

```

gt(E)
(then, res(E), __, __)
gt(S1)
(else, __, __, __)
gt(S2)
(ifend, __, __, __)

```

解：语句，无外乎逻辑以及逻辑下的具体操作。其中  $gt(S_1)$  和  $gt(S_2)$  均为完整语义块，若不考虑嵌套条件语句， $gt(S_1)$  和  $gt(S_2)$  均为顺序执行。因此给出的四元式中，包含  $gt(E)$  (then, res(E), \_\_, \_\_),  $gt(S_1)$  (else, \_\_, \_\_, \_\_) 和  $gt(S_2)$  (ifend, \_\_, \_\_, \_\_) 三个基本块，入口语句为第一个语句或转移语句的后一句，一直到另一转移语句结束。

**例 8.2** 设有源程序片段，给出基本块划分：

```

x = 1;
a : r = x * 5;
if(x < 10)
x = x + 1;
goto a;
r = 0;

```

解：`goto` 语句在编程中不建议大家使用，但有助于理解编译器。第一步，写出对应的四元式序列如图8.1，其中的赋值语句是以单操作形式，最左侧为结果单元。`goto` 语句对应的四元式，定义操作符为  $gt$ ，表示直接跳转。 $lb$  四元式表示跳转的目标，因此将它作为基本块的开始。 $if$  四元式等价于前面介绍的 (then, res(E), \_\_, \_\_)。 $ie$  四元式是转向语句转移到的语句，作为基本块的开始。

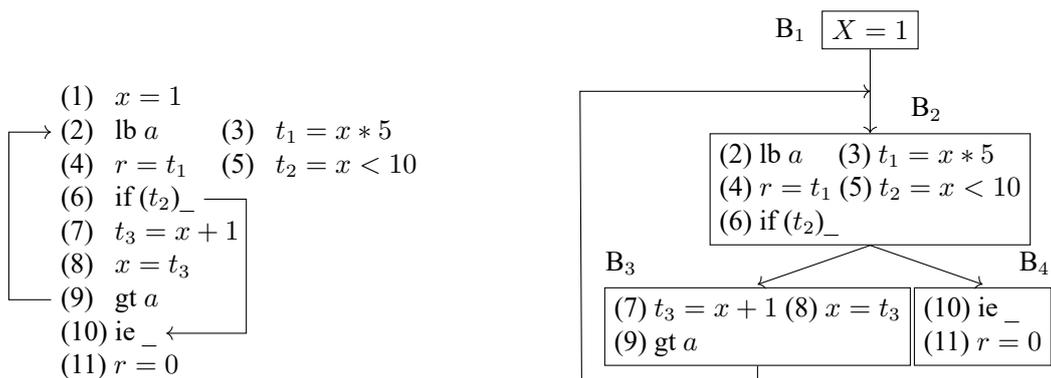


图 8.1: 对应的四元式序列

图 8.2: 以基本块为节点的程序流图

第二步，划分基本块。根据算法，找出所有入口，两入口之间的部分即为基本块。观察四元式序列，第 (1) 个四元式是程序入口，是基本块的入口语句；第 (2) 个四元式是第 (9) 个四元式跳转到的语句，可以作为基本块的入口；第 (3) 到第 (5) 个四元式，根据定义，都不是基本块入口；第 (6) 个四元式，是跳转语句，不是基本块入口，但跳转语句的下一条，即第 (7) 个四元

式是基本块入口；第(8)个四元式不是基本块入口；第(9)个四元式是跳转语句，不是基本块入口，下一个四元式(10)是基本块入口。从而划分得到如下四个基本块，如图8.2。

### 8.3.2 局部优化示例

**例 8.3** 基本块内设有语句片段：

$$B = 5; A = 2 * 3.14 / (R + r);$$

$$B = 2 * 3.14 / (R + r) * (R - r);$$

解：这是一个基本块，优化后结果如下。第一个式子删除对 B 的无用赋值；第二个式子可进行常值表达式节省， $2 * 3.14$  可优化为 6.28；第二个式子和第三个式子中都出现了  $2 * 3.14 / (R + r)$ ，可以进行公共表达式节省，第二个式子中替换为 A。

$$A = 6.28 / (R + r);$$

$$B = A * (R - r);$$

上述是人工优化的思考过程，下面系统地进行优化。

- 第一步，根据原语句片段，生成图8.3左侧四元式序列。
  - 第二步，逐个观察四元式。
    - 第(1)个四元式，没有可优化的。
    - 第(2)个四元式，是常值表达式，可以优化为  $t_1 = 6.28$ ，保存该值，不生成四元式。
    - 第(3)个四元式， $R + r$  必须计算，不能优化。
    - 第(4)个四元式，将  $t_1$  替换为 6.28。
    - 第(5)个四元式不能优化。
    - 第(6)个四元式，与第(2)个四元式相同，优化为  $t_4 = 6.28$  并保存。
    - 第(7)个四元式，与第(3)个四元式运算、运算对象相同，是公共表达式，保留  $t_5 = t_2$  的关系。
    - 第(8)个四元式， $t_4$  用 6.28 替代， $t_5$  用  $t_2$  替代，与第(4)个四元式有公共表达式，保留  $t_6 = t_3$  的关系。
    - 第(9)个四元式不能优化。
    - 第(10)个四元式中的  $t_6$  用  $t_3$  代替。
    - 第(11)个四元式赋值给 B。
- 得到结果如图8.3右侧，相较原四元式序列，省去了四条。
- 第三步：继续观察四元式。B 的两次赋值之间（第(2)到第(10)个四元式中），B 没有引用，因此删除第(1)个四元式中的无用赋值 B。

最终得到优化后 6 个四元式序列，优化过程如图8.3所示。

根据例8.3总结优化的基本算法设计：

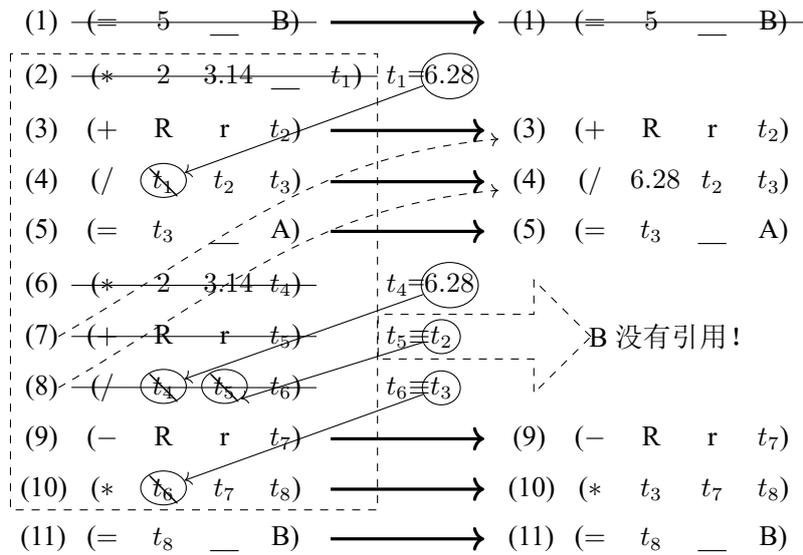


图 8.3: 四元式序列的优化过程

1. 对于常值表达式节省 (例中 (2)(6))
    - (1) 先进行常值计算;
    - (2) 取常值的变量以常值代之;
  2. 公共表达式节省 (例中 (7)(8))
    - (1) 找公共表达式, 建立结果变量等价关系;
    - (2) 等价变量以老变量代替新变量;
  3. 删除无用赋值 (例中 (1))
    - (1) 确认一个变量两个赋值点间无引用点;
    - (2) 则前一赋值点为无用赋值;
- .....

接下来, 进一步构造算法解决这些问题。

### 8.4 基于 DAG 的局部优化方法

DAG (Directed Acyclic Graph) 是指无环有向图, 或称有向无环图 (如图8.4), 对于计算机相关的系统开发及研究十分重要, 这里用来对基本块内的四元式序列进行优化。

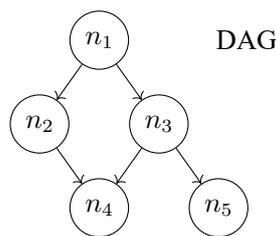


图 8.4: 有向无环图

### 8.4.1 四元式序列的 DAG 表示

如何把线性的四元式映射到图结构呢？下面图8.5给出 DAG 的结点内容及其表达。图中包含结点，因为是有向图，又包含前驱和后继。四元式中包含一个运算符，两个运算对象，一个结果单元。有向无环图的要素如何与四元式的要素对应起来，这是要研究的第一个问题。

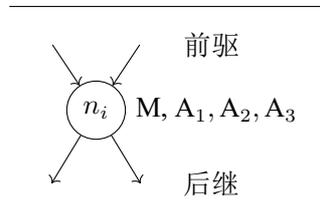


图 8.5: 结点的前驱和后继

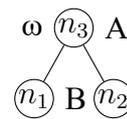
#### 1. DAG 的结点内容及其表示

- $n_i$ : 结点的编码;
- $\omega$ : 运算符, 置于结点左侧, 将该结点称为  $\omega$  运算结点, 代表  $\omega$  运算。 $\omega$  运算的运算对象为其后继节点, 结果单元标记在结点右侧;
- M: 主标记, 使用时作为代表优先被使用, 如果是叶子结点, 表示变量和常数的初值;
- $A_i$ : 附加标记, 为运算结果变量, 为便于描述等价关系, 可设置多个,  $i = 1, 2, 3, \dots$ 。如例8.3中  $t_2$  和  $t_5$  都表示  $R + r$  的结果, 可以  $t_2$  为主标记,  $t_5$  为附加标记, 从而便于描述等价关系。

右图中值为  $\omega$  运算结果超过一个变量, M,  $A_1, A_2, A_3$  都取  $\omega$  运算结果值, 它们都应放在结点右侧。在 M,  $A_1, A_2, A_3$  中选择 M 作为主标记, 主标记之外的运算结果变量称为附加标记,  $A_1, A_2, A_3$  均与 M 等价。

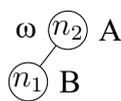
#### 2. 四元式的 DAG 表示

- 赋值: 赋值四元式 ( $=B \_ A$ ) 或  $A=B$ , 将 B 的值赋给 A, 表示 A 和 B 是等价的。DAG 表示为  $\omega(n_i)B|A$ , 省去了赋值运算符, B 为主标记, A 为附加标记。
- 双目运算: 双目运算四元式 ( $\omega B C A$ ) 或  $A = B\omega C$ , 将  $B\omega C$  赋值给 A, 其中  $\omega$  可以

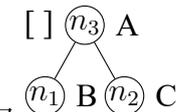


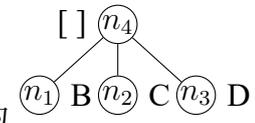
是算术运算、关系运算、逻辑运算等。DAG 表示为  $\omega(n_3)B|A$ ,  $\omega$  运算符放在结点  $n_3$  左侧, 结果单元 A 放在结点  $n_3$  右侧, 两个运算对象 B 和 C 放在下面结点  $n_1$  和  $n_2$  的右侧, 作为  $n_3$  的后继结点, 通过  $\omega$  运算赋值给 A。

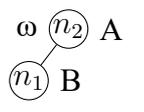
- 单目运算: 有单目运算四元式 ( $\omega B \_ A$ ) 或  $A = \omega B$ 。与双目运算类似, 从原来的二



叉变成“一叉”, DAG 表示为  $\omega(n_2)B|A$ 。结点  $n_1$  表示 B, 通过  $\omega$  运算赋值给 A。

- **下标变量赋值运算**：有下标变量赋值运算四元式  $A=B[C]$ ，DAG 表示为 。A 有两个后继 B 和 C，[] 表示按变量 C 为偏移量，取变量 B 对应数组的元素，结果

用 A 表示。将变量赋值给数组元素  $B[C]=D$ ，DAG 表示为 。这种情况比较特殊，[] 操作对象不是 B、C、D 中任何一个，而是以 B 指针开始，以 C 对应的变量为偏移，并以 D 赋值，因此结点  $n_4$  右侧没有出现结果单元，而是用这个结构表示赋值的过程。

- **转向**：有转向四元式  $(\omega [B] \_ A)$ ，B 为可选单元，DAG 表示如下：第一个表示无条件跳转到  $A \omega (n_i) A$ ，第二个根据 B 进行跳转 。

有了这样的表示形式，就能够完成对四元式序列的 DAG 表示。

**例 8.4** 求下述语句片段的 DAG 表示：

$B = 5; A = 2 * 3.14 * (R + r); B = 2 * 3.14 * (R + r) / (R - r);$

解：

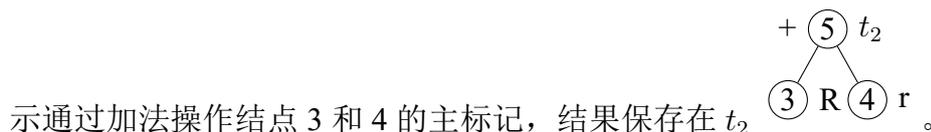
- 第一步，生成对应的四元式序列。
  - (1)  $B = 5$       (2)  $t_1 = 2 * 3.14$     (3)  $t_2 = R + r$
  - (4)  $t_3 = t_1 * t_2$     (5)  $A = t_3$             (6)  $t_4 = 2 * 3.14$
  - (7)  $t_5 = R + r$     (8)  $t_6 = t_4 * t_5$     (9)  $t_7 = R - r$
  - (10)  $t_8 = t_6 / t_7$     (11)  $B = t_8$

- 第二步，逐个扫描四元式，构建优化 DAG 图。

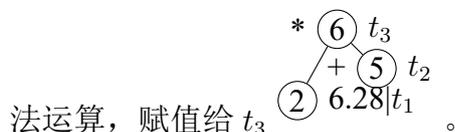
第 (1) 个赋值四元式，创建结点 1  $\textcircled{1} 5|B$ ，主标记为 5，附加标记为 B，表示  $B = 5$ 。

第 (2) 个常值运算四元式，将  $2*3.14$  化简为 6.28，创建结点 2  $\textcircled{2} 6.28|t_1$ ，与结点 1 类似，主标记为 6.28，附加标记为  $t_1$ 。

第 (3) 个四元式，R 和 r 在 DAG 中均未出现，创建结点 3、4 表示 R 和 r，再创建结点 5，表



第 (4) 个四元式， $t_1$  在结点 2， $t_2$  在结点 5，创建结点 6，操作结点 2 和 5 的主标记进行乘

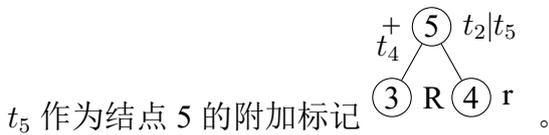


第 (5) 个四元式，将  $t_3$  赋值给 A， $t_3$  在 DAG 中已存在，因此 A 作为结点 6 的附加标

记 \*  $\textcircled{6} t_3|A$ 。注意，因为主标记  $t_3$  为临时变量，而  $A$  为用户定义变量，这里要将  $A$  和  $t_3$  进行互换，表示为 \*  $\textcircled{6} A|t_3$ 。

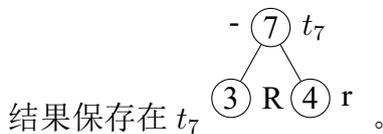
第 (6) 个四元式，将 6.28 赋值给  $t_4$ ，6.28 在 DAG 中已存在，因此  $t_4$  作为结点 2 的附加标记  $\textcircled{6} 6.28|t_1, t_4$ 。

第 (7) 个四元式，将  $R + r$  赋值给  $t_5$ ， $R$  和  $r$  以及  $R + r$  的结果在 DAG 中也已存在，因此将

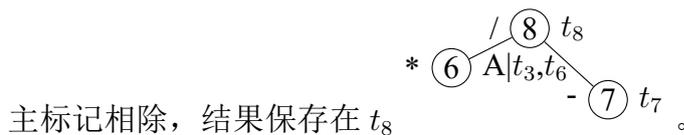


第 (8) 个四元式， $t_4$  在结点 2， $t_5$  在结点 5，结点 2 和结点 5 的乘法结果在 DAG 中已存在，因此将  $t_6$  作为结点 6 的附加标记  $\textcircled{6} A|t_3, t_6$ 。

第 (9) 个四元式， $R$  在结点 3， $r$  在结点 4， $R - r$  的结果不存在，因此创建结点 7， $R - r$  的



第 (10) 个四元式， $t_6$  在结点 6， $t_7$  在结点 7，创建结点 8，表示结点 6 的主标记和结点 7 的



第 (11) 个四元式，将  $t_8$  赋值给  $B$ ，在结点 1 中删去  $\textcircled{1} 5|B$ ，将  $B$  作为结点 8 的附加标记  $\textcircled{8} B|t_8$ ，同样需要调换  $B$  和  $t_8$  的顺序  $\textcircled{8} B|t_8$ 。

得到最终的 DAG 表示如下：

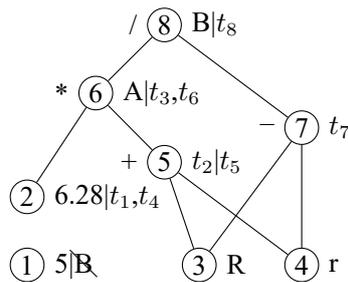


图 8.6: 最终的 DAG 表示

- 第三步，根据优化的 DAG 重组四元式。

结点 1，没有后继结点，即没有运算四元式，附加标记的  $B$  被删去，若存在附加标记，表示存在赋值四元式。

结点 2，没有后继结点，没有运算四元式，但存在等价关系，理论上生成  $t_1 = 6.28$  和  $t_4 = 6.28$  两个赋值四元式，但  $t_1$  和  $t_4$  是临时变量，是编译器定义的，与用户无关，临时变量不会在后续被使用，且  $t_1$ 、 $t_4$  与 6.28 等价，因此可以用 6.28 代替  $t_1$  和  $t_4$ ，附加标记中放的都是临时变量，这时的赋值四元式可以省去。

结点 3 和结点 4，没有后继结点，没有附加标记，不需要进行操作。

结点 5，存在后继结点，生成运算四元式  $t_2 = R + r$ ，附加标记上的临时变量不用生成赋值语句，因此具有等价关系的  $t_5$  被省去。

结点 6，存在后继结点，生成运算四元式  $A = 6.28 * t_2$ ，附加标记  $t_3$ 、 $t_6$  均为临时变量，省去。

结点 7，存在后继结点，生成运算四元式  $t_7 = R - r$ 。

结点 8，存在后继结点，生成运算四元式  $B = A/t_7$ 。

最终得到重组四元式如下，四元式数量从原来的 11 减少为 4。

$$(1) t_2 = R + r$$

$$(2) A = 6.28 * t_2$$

$$(3) t_7 = R - r$$

$$(4) B = A/t_7$$

介绍到这里，第二步中对于结点 6 和结点 8，为什么要进行位置交换？其实目的是减少四元式数量，优化中间代码。当结点右部存在若干个等价的值（包括常值、变量），在这若干个中选择一个作为主标记，优先关系为：常值（第一级）、用户定义变量（第二级）、临时变量（第三级）。如果附加标记中都是临时变量，赋值四元式可以省去；如果附加标记中存在用户定义变量，赋值四元式不可省去，因为在后续可能会被用户使用。结点 1 中 B 在附加标记上，且被重新赋值，原来的 B 可被主标记值替代，因此被省去，若 B 在主标记上，则不可省去。

### 8.4.2 基于 DAG 的局部优化算法

例 8.4 是一个典型的 DAG 优化算法实例，下面对基于 DAG 的局部优化算法进行系统总结。

#### 1. 构造基本块内优化的 DAG

假设：①  $n_i$  为已知结点号， $n$  为新结点号；② 访问各节点信息时，按结点号逆序进行；具体原因后面在例题中说明。

• 开始：

① DAG 置空；依次读取一四元式  $A = B \omega C$ 。

② 分别定义 B, C 结点，若已定义过，可以直接使用。

(1) 若赋值四元式  $A = B$

① 把 A 附加于 B 上，即  $\textcircled{n_1} \dots B \dots A$ ，表示 A 和 B 等价。

② 若 A 在  $n_2$  已定义过，且 A 在附加标记时，需要删去，即  $\textcircled{n_2} \dots | \dots A$ ；A 在主标记时，则无需删去。

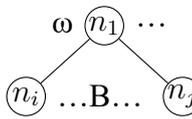
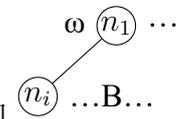
(2) 若常值表达式  $A = C_1 \omega C_2$  或  $A = C$

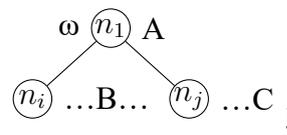
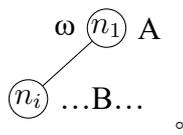
① 计算常值  $C = C_1 \omega C_2$ 。

②若 C 在  $n_1$  已定义过, 则  $\omega(n_1) C | \dots, A$ , 如上例中结点 2 的主标记为 6.28, 表示  $t_4 = 6.28$  时, 只需将  $t_4$  作为这个结点的附加标记; 否则申请新结点, 且  $\omega(n) C | A$ , C 为主标记, A 为附加标记。

③若 A 在  $n_2$  已定义过, 且 A 在附加标记, 删去, 即  $\omega(n_2) \dots | \dots A \dots$ 。

(3) 若其他表达式  $A = B \omega C$  或  $A = \omega B$

①若在  $n_1$  存在公共表达式  $B \omega C$  或  $\omega B$ , 分别表示为  $\omega(n_1) \dots$   和 , 则把 A 附加在  $n_1$  上, 即  $\omega(n_1) \dots, A$ 。

②若不存在公共表达式, 则申请新结点 n,  或 。

③若 A 在  $n_2$  已定义过, 且 A 在附加标记, 则删除, 即  $\omega(n_2) \dots | \dots A \dots$ ; 若 A 为主标记, 则免删。

• 结束: 调整结果单元结点的主标记、附加标记的顺序。

★ 主标记优先顺序为: 常量, 非临时变量, 临时变量。

### 2. 根据基本块内优化的 DAG, 重组四元式

假设: ①临时变量的作用域是基本块内; ②非临时变量的作用域也可以是基本块外。

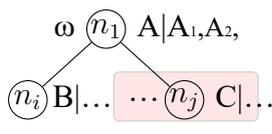
两条假设的目的是在最终优化后的代码中, 必须保证所有非临时变量的值是最新的, 临时变量的值若在后续没有使用, 就不需要赋值。

• 开始: 按结点编码顺序, 依次读取每一结点  $n_1$  信息:

(1) 若  $n_1$  为带有附加标记的叶结点, 即  $\omega(n_1) B | A_1, A_2, \dots$ , 表示  $A_i$  与 B 等价

①若  $A_i$  为非临时变量, 则生成  $q_1: A_i = B (i = 1, 2, \dots)$ 。

②若  $A_i$  是临时变量, 则不需要生成。临时变量在基本块外不使用, 而生成四元式时使用的是主标记 B。

(2) 若  $n_1$  为带有附加标记的非叶结点, 即 。

①生成  $q_1: A = B \omega C$  或  $A = \omega B$ , 用主标记进行计算。

②若  $A_i$  为非临时变量, 则生成  $q_2: A_i = A (i = 1, 2, \dots)$ 。保证非临时变量在基本块出口时, 值是正确的。若  $A_i$  是临时变量, 则不需要生成四元式。

★ 注意: 以主标记参加运算。

例 8.5 求下述语句片段的 DAG 表示:

$$A = 2 * 3 + B / C$$

$$B = 2 * 3 + B / C$$

$$C = 2 * 3 + B / C$$

解:

- 第一步, 生成四元式序列 (不能“自动”优化)

(1) (\*, 2, 3,  $t_1$ )

(2) (/, B, C,  $t_2$ )

(3) (+,  $t_1$ ,  $t_2$ ,  $t_3$ )

(4) (=,  $t_3$ , \_\_, A)

(5) (\*, 2, 3,  $t_4$ )

(6) (/, B, C,  $t_5$ )

(7) (+,  $t_4$ ,  $t_5$ ,  $t_6$ )

(8) (=,  $t_6$ , \_\_, B)

(9) (\*, 2, 3,  $t_7$ )

(10) (/, B, C,  $t_8$ )

(11) (+,  $t_7$ ,  $t_8$ ,  $t_9$ )

(12) (=,  $t_9$ , \_\_, C)

- 第二步, 构造优化的 DAG

依次读取四元式, 根据算法构造 DAG。需要注意, 访问各结点信息时, 按结点号逆序进行。例如读取第 10 个四元式后, 从结点 5 开始找最新的 B 和 C, 从而建立新的结点, 旧的 B 参与运算, 且是主标记, 要保留。前两个表达式中的 B/C 是公共表达式, 而第三个式子中不是, B 的值已经被更新。类似地, 读取各个四元式, 依据算法构建 DAG, 最终优化后的 DAG 图如图 8.7。

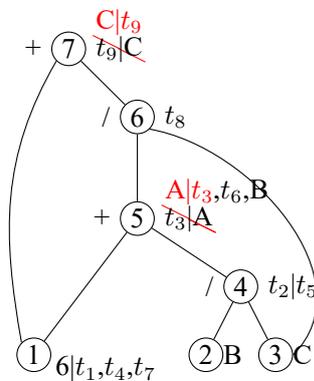


图 8.7: 最终优化后的 DAG 表示

- 第三步, 优化后的四元式序列。按结点编码顺序, 依次读取每一结点  $n_1$  信息, 生成相应四元式序列如下。

- (1) ( $/$ , B, C,  $t_2$ )
- (2) (+, 6,  $t_2$ , A)
- (3) (=, A,  $\_$ , B)
- (4) ( $/$ , A, C,  $t_8$ )
- (5) (+, 6,  $t_8$ , C)