



第10章 代码优化



重庆大学 葛亮

知识点：基本块优化
循环优化

代码优化

- 10.1 代码优化概述
- 10.2 基本块优化
- 10.3 dag在基本块优化中的应用
- 10.4 循环优化
- 10.5 窥孔优化
- 小结

10.1 代码优化概述

■ 代码优化程序的任务

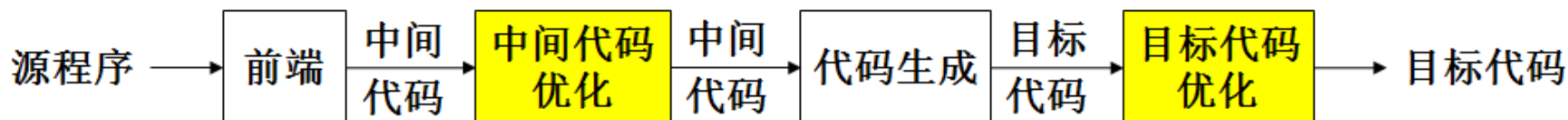
- ◆ 对中间代码或目标代码进行等价变换，使变换后的代码质量更高。

■ 对代码优化程序的要求

- ◆ 等价变换
- ◆ 提高目标代码的执行速度
- ◆ 减少目标代码占用的空间

■ 代码优化程序的位置

- ◆ 目标代码生成之前的中间代码优化
- ◆ 目标代码生成之后的目标代码优化



代码优化的主要种类

■ 中间代码优化

◆ 基本块优化

- 在基本块内进行的优化。
- 常数合并与传播、删除公共子表达式、复制传播、削弱计算强度、改变计算次序等。

◆ 循环优化

- 在循环语句所生成的中间代码序列上进行的优化。
- 循环展开、代码外提、削弱计算强度、删除归纳变量等。

◆ 全局优化

- 在非线性程序段上（含多个基本块）进行的优化。

■ 目标代码优化

◆ 窥孔优化

- 在目标代码上进行局部改进的优化。
- 删除冗余指令、控制流优化、代数化简等。

10.2 基本块优化

10.2.1 常数合并及常数传播

10.2.2 删除公共表达式

10.2.3 复制传播

10.2.4 削弱计算强度

10.2.5 改变计算次序

10.2.1 常数合并及常数传播

- 常数合并：将在编译时可计算出值的表达式用其值替代。

$x=2+3+y$ 可代之以： $x=5+y$

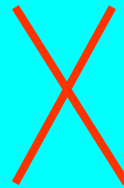
- 常数传播：用在编译时已知的变量值代替程序正文中对这些变量的引用。

$PI:=3.14;$


$D\text{-}to\text{-}R:= 0.01744$

```
i:=0
10: i:=i+1
...
if i<10 goto 10
```

```
i:=0
10: i:=0+1
...
if i<10 goto 10
```



```
...
a[i]:=9.0
...
a[j]:=3.0
b:=a[i]
```



常数合并的实现

■ 在符号表中增加两个信息域

- ◆ 标志域：指示当前该变量的值是否存在。
- ◆ 常数域：如果变量值存在，则该域存放的即是该变量的当前值。

■ 常数合并时，注意事项：

- ◆ 不能将结合律与交换律用于浮点表达式。
 - 浮点运算的精度有限，这两条定律并非是恒真的。
 - 如： $(11+2.8)+0.3$ vs. $11+(2.8+0.3)$
- ◆ 不应将任何附加的错误引入。

10.2.2 删除公共表达式

- 在一个基本块中，当第一次对表达式E求值之后，如果E中的运算对象都没有改变，再次对E求值，则除E的第一次出现之外，其余的都是冗余的公共表达式。
- 删除冗余的公共表达式，用第一次出现时的求值结果代替之。

(1) **a:=b+c**

(2) **b:=a-d**

(3) **c:=b+c**

(4) **d:= b**

示例

B₄

(4) $t_1 := 4 * i$

(5) $t_2 := a - 4$

(6) $t_3 := 4 * i$

(7) $t_4 := a - 4$

(8) $t_5 := t_4[t_3]$

(9) $t_6 := 4 * i$

(10) $t_7 := b - 4$

(11) $t_8 := t_7[t_6]$

(12) $t_9 := t_5 + t_8$

(13) $t_2[t_1] := t_9$

(14) $t_{10} := i + 1$

(15) $i := t_{10}$

(16) goto B₂

(4) $t_1 := 4 * i$

(5) $t_2 := a - 4$

(6') $t_3 := t_1$

(7') $t_4 := t_2$

(8) $t_5 := t_4[t_3]$

(9') $t_6 := t_1$

(10) $t_7 := b - 4$

(11) $t_8 := t_7[t_6]$

(12) $t_9 := t_5 + t_8$

(13) $t_2[t_1] := t_9$

(14) $t_{10} := i + 1$

(15) $i := t_{10}$

(16) goto B₂

10.2.3 复制传播

- 在复制语句 $f:=g$ 之后，尽可能用 g 代替 f 。

```
(4)  $t_1:=4*i$   
(5)  $t_2:=a-4$   
(6')  $t_3:=t_1$   
(7')  $t_4:=t_2$   
(8)  $t_5:=t_4[t_3]$   
(9')  $t_6:=t_1$   
(10)  $t_7:=b-4$   
(11)  $t_8:=t_7[t_6]$   
(12)  $t_9:=t_5+t_8$   
(13)  $t_2[t_1]:=t_9$   
(14)  $t_{10}:=i+1$   
(15)  $i:=t_{10}$   
(16) goto  $B_2$ 
```

```
(4)  $t_1:=4*i$   
(5)  $t_2:=a-4$   
(6')  $t_3:=t_1$   
(7')  $t_4:=t_2$   
(8')  $t_5:=t_2[t_1]$   
(9')  $t_6:=t_1$   
(10)  $t_7:=b-4$   
(11')  $t_8:=t_7[t_1]$   
(12)  $t_9:=t_5+t_8$   
(13)  $t_2[t_1]:=t_9$   
(14)  $t_{10}:=i+1$   
(15)  $i:=t_{10}$   
(16) goto  $B_2$ 
```

```
(4)  $t_1:=4*i$   
(5)  $t_2:=a-4$   
  
(8')  $t_5:=t_2[t_1]$   
  
(10)  $t_7:=b-4$   
(11')  $t_8:=t_7[t_1]$   
(12)  $t_9:=t_5+t_8$   
(13)  $t_2[t_1]:=t_9$   
(14)  $t_{10}:=i+1$   
(15)  $i:=t_{10}$   
(16) goto  $B_2$ 
```

```
(4)  $t_1:=4*i$   
(5)  $t_2:=a-4$   
  
(8')  $t_5:=t_2[t_1]$   
  
(10)  $t_7:=b-4$   
(11')  $t_8:=t_7[t_1]$   
(12)  $t_9:=t_5+t_8$   
(13)  $t_2[t_1]:=t_9$   
  
(15')  $i:=i+1$   
(16) goto  $B_2$ 
```

删除死代码

- **死代码**：如果对一个变量 x 求值之后却不引用它的值，则称对 x 求值的代码为死代码。
- **死块**：控制流不可到达的块称为死块。
 - ◆ 如果一个基本块是在某一条件为真时进入执行的，经数据流分析的结果知该条件恒为假，则此块是死块。
 - ◆ 如果一个基本块是在某个条件为假时才进入执行，而该条件却恒为真，则这个块也是死块。
- 在确定一个基本块是死块之前，需要检查转移到该块的所有转移语句的条件。
- 死块的删除，可能使其后继块成为无控制转入的块，这样的块也成为死块，同样应该删除。

10.2.4 削弱计算强度

- 对基本块的代数变换：对表达式中的求值计算用代数上等价的形式替换，以便使复杂的运算变换成为简单的运算。

$x := y ** 2$ （乘方需要调用函数来计算）

可以用代数上等价的乘式（如： $x := y * y$ ）代替

- $x := x + 0$ 和 $x := x * 1$
 - ◆ 执行的运算没有任何意义
 - ◆ 应将这样的语句从基本块中删除。

10.2.5 改变计算次序

- 考虑语句序列：

$t_1 := b + c$

$t_2 := x + y$

- 如果这两个语句是互不依赖的，即 x 、 y 均不为 t_1 ， b 、 c 均不为 t_2 ，则交换这两个语句的位置不影响基本块的执行结果。

- 对基本块中的临时变量重新命名不会改变基本块的执行结果。

如：语句 $t := b + c$

改成语句 $u := b + c$

把块中出现的所有 t 都改成 u ，不改变基本块的值。

10.3 dag在基本块优化中的应用

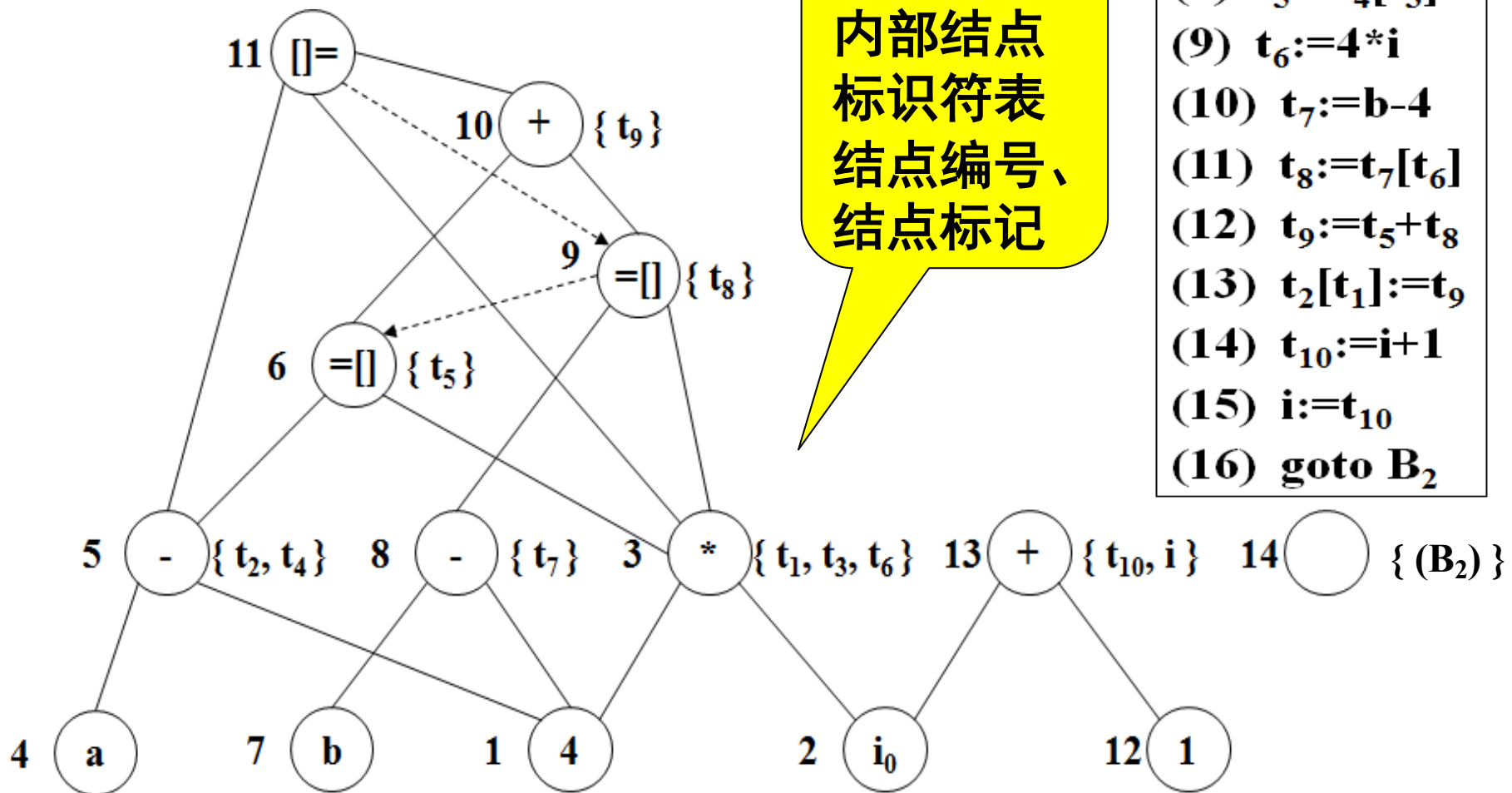
- dag是实现基本块等价变换的一种有效的数据结构。
- 一个基本块的dag是一种在其结点上带有下述标记的有向非循环图：
 - ◆ 图的叶结点由变量名或常量标记。
 - 根据作用到一个名字上的算符，可以决定需要的是名字的左值还是右值。
 - 大多数叶结点代表右值（叶结点代表名字的初始值），因此，通常将其标识符加上脚标0，以区别于指示名字的当前值的标识符。
 - ◆ 图的内部结点由一个运算符号标记，每个内部结点均代表应用其运算符对其子结点所代表的值进行运算的结果。
 - ◆ 图中每个结点都有一个标识符表，其中可有零个或多个标识符。这些标识符都具有该结点所代表的值。

10.3.1 基本块的dag表示

B₄

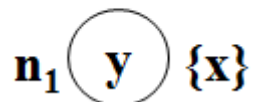
- (4) $t_1 := 4 * i$
- (5) $t_2 := a - 4$
- (6) $t_3 := 4 * i$
- (7) $t_4 := a - 4$
- (8) $t_5 := t_4[t_3]$
- (9) $t_6 := 4 * i$
- (10) $t_7 := b - 4$
- (11) $t_8 := t_7[t_6]$
- (12) $t_9 := t_5 + t_8$
- (13) $t_2[t_1] := t_9$
- (14) $t_{10} := i + 1$
- (15) $i := t_{10}$
- (16) **goto** B₂

叶结点
内部结点
标识符表
结点编号、
结点标记

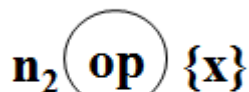


常用三地址语句的dag结点形式

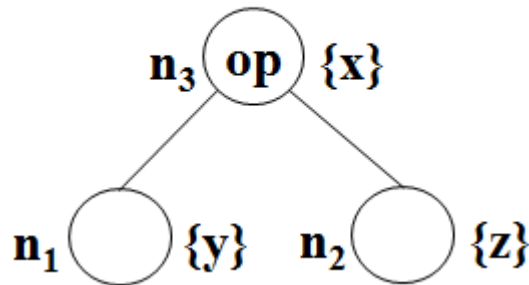
(0) $x:=y$



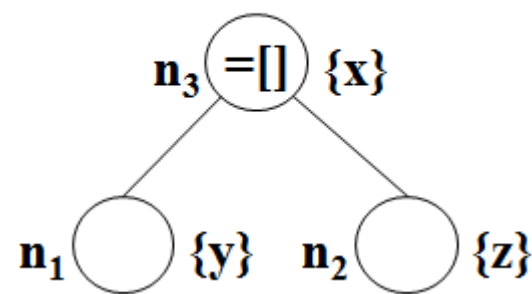
(2) $x:=op\ y$



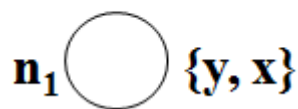
(3) $x:=y\ op\ z$



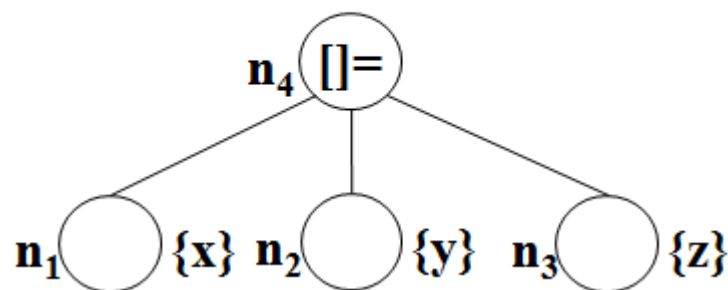
(4) $x:=y[z]$



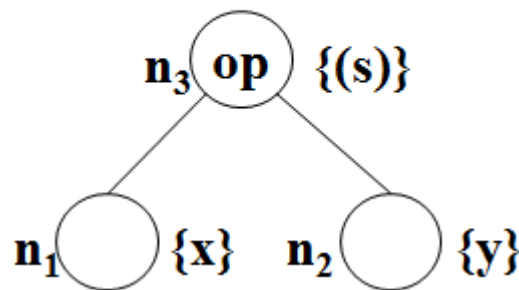
(1) $x:=y$



(5) $x[y]:=z$



(6) if $x\ op\ y$ goto (s)



(7) goto (s)



10.3.2 基本块的dag构造算法

- 输入：一个基本块。
- 输出：该基本块的dag，其中包括如下的信息。
 - ◆ 每个结点都有一个标记，叶结点的标记是一个名字或者常数，内部结点的标记是一个运算符号。
 - ◆ 在每个结点上有一个附加的标识符表，表中可以有零或多个名字。
- 算法用到的主要数据结构：
 - ◆ 保存dag的数据结构（如数组、链表等），其中存储各结点的信息以及结点之间的关系。
 - ◆ 保存结点附加信息的数据结构，需要记录结点的编号、标记、以及与结点相关的名字列表或常数。

算法用到的函数

- $n := \text{lookupnode}(\text{id}, \text{child}, n1, n2, n3)$: 根据所给参数查找dag结点。
 - ◆ 找到, 则返回该结点的编号 n ;
 - ◆ 否则, 返回-1。
 - ◆ 参数说明:
 - 若 id 是常数, 则查找以此常数标记的叶子结点。
 - 若 id 是名字, 则查找以此名字标记的叶子结点、或者标识符表中有名字 id 的内部结点。
 - 若 id 是运算符, 则查找以此运算符标记的内部结点, 且该结点有 child 个子结点。
 - 若 $\text{child}=1$, 则子结点的编号应为 $n1$;
 - 若 $\text{child}=2$, 则子结点的编号应依次为 $n1$ 和 $n2$;
 - 若 $\text{child}=3$, 则子结点的编号应依次为 $n1$ 、 $n2$ 和 $n3$ 。

算法用到的函数（续1）

- $n := \text{makenode}(\text{id}, \text{child}, n1, n2, n3)$: 建立一个标记为id的结点，初始化其标识符表为空，并返回新建结点编号n。

◆ 参数说明

- 若 $\text{child} = 0$ ，则建立一个标记为id的叶子结点。
此时，id可以是一个名字或者常数。
- 若 $\text{child} \neq 0$ ，则建立一个标记为id的内部结点。
此时，id是一个运算符。
 - 若 $\text{child} = 1$ ，则新建结点以编号为n1的结点为子结点；
 - 若 $\text{child} = 2$ ，则新建结点以编号为n1和n2的结点为左右子结点；
 - 若 $\text{child} = 3$ ，则新建结点依次以编号为n1、n2和n3的结点为左中右子结点。

算法用到的函数（续2）

- **attachnode(n, x):** 将名字x附加到结点n上，即加入结点n的标识符表中。
- **detachnode(n, x):** 将名字x从结点n的标识符表中删除，若结点n的标识符表中没有名字x，则没有影响。

构造方法

(1) $x:=y$ (2) $x:=op\ y$ (3) $x:=y\ op\ z$
从入口语句开始，依次处理每一条三地址语句

- (1) for (基本块中的每一条三地址语句) {
- (2) switch 当前处理的三地址语句 {
- (3) case 形如 $x:=y$ 的赋值语句:
- (4) $n:=lookupnode(y, 0, 0, 0, 0);$
- (5) if ($n==-1$) // 所查结点不存在
- (6) $n:=makenode(y, 0, 0, 0, 0);$ // 建立一个标记为 y 的叶子结点
- (7) $m:=lookupnode(x, 0, 0, 0, 0);$
- (8) if ($m!=-1$) // 所查结点已经存在
- (9) $detachnode(m, x);$
- (10) $attachnode(n, x);$
- (11) break;

构造方法（续1）

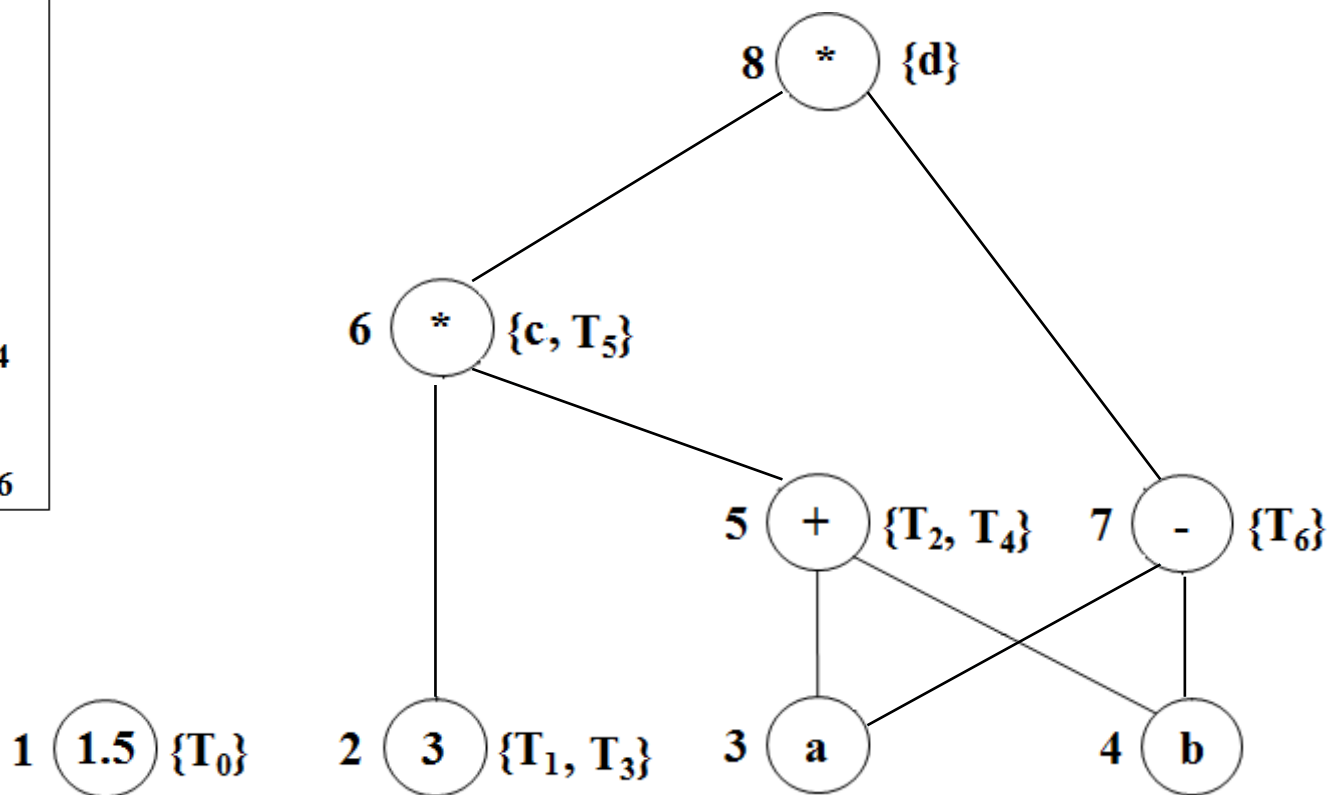
```
(12) case 形如  $x:=op\ y$  的赋值语句:
(13)     if (y是常数) {           // 常数合并
(14)          $p:=op\ y$ ;           // 计算出op y的值p
(15)          $n:=lookupnode(p, 0, 0, 0, 0)$ ;
(16)         if ( $n==-1$ )
(17)              $n:=makenode(p, 0, 0, 0, 0)$ ;
(18)         };
(19)     else {                   // y不是常数
(20)          $k:=lookupnode(y, 0, 0, 0, 0)$ ;
(21)         if ( $k==-1$ ) {
(22)              $k:=makenode(y, 0, 0, 0, 0)$ ;
(23)              $n:=makenode(op, 1, k, 0, 0)$ 
(24)         }
(25)     else {
(26)          $n:=lookupnode(op, 1, k, 0, 0)$ ;
(27)         if ( $n==-1$ )
(28)              $n:=makenode(op, 1, k, 0, 0)$ 
(29)         }
(30)     }
(31)      $m:=lookupnode(x, 0, 0, 0, 0)$ ;
(32)     if ( $m!=-1$ )           // 所查结点已经存在
(33)          $detachnode(m, x)$ ;
(34)      $attachnode(n, x)$ ;
(35)     break;
```

构造方法（续2）

```
(36) case形如  $x:=y \text{ op } z$  的赋值语句:
(37)   if ( $y$ 是常数 &&  $z$ 是常数) { // 常数合并
(38)        $p:=y \text{ op } z$ ; // 计算出 $y \text{ op } z$ 的值 $p$ 
(39)        $n:=lookupnode(p, 0, 0, 0, 0)$ ;
(40)       if ( $n==-1$ )
(41)            $n:=makenode(p, 0, 0, 0, 0)$ ;
(42)       }
(43)   else { //  $y$ 和 $z$ 中至少有一个不是常数
(44)        $k:=lookupnode(y, 0, 0, 0, 0)$ ;
(45)       if ( $k==-1$ )
(46)            $k:=makenode(y, 0, 0, 0, 0)$ ;
(47)        $l:=lookupnode(z, 0, 0, 0, 0)$ ;
(48)       if ( $l==-1$ )
(49)            $l:=makenode(z, 0, 0, 0, 0)$ ;
(50)        $n:=lookupnode(op, 2, k, l, 0)$ ;
(51)       if ( $n==-1$ )
(52)            $n:=makenode(op, 2, k, l, 0)$ ;
(53)   }
(54)    $m:=lookupnode(x, 0, 0, 0, 0)$ ;
(55)   if ( $m!=-1$ ) // 所查结点已经存在
(56)        $detachnode(m, x)$ ;
(57)    $attachnode(n, x)$ ;
(58)   break;
(59) }; // end switch
(60) }; // end for
```

算法应用示例

-
- (1) $T_0 := 1.5$
 - (2) $T_1 := 2 * T_0$
 - (3) $T_2 := a + b$
 - (4) $c := T_1 * T_2$
 - (5) $d := c$
 - (6) $T_3 := 2 * T_0$
 - (7) $T_4 := a + b$
 - (8) $T_5 := T_3 * T_4$
 - (9) $T_6 := a - b$
 - (10) $d := T_5 * T_6$



10.3.3 dag的应用

- 通过构造dag，可以获得一些十分有用的信息。
 - ◆ 首先，可以检测出公共子表达式。
 - ◆ 其次，可以确定出哪些名字的值在前驱块中计算而在本块内被引用。
即，dag中叶子结点对应的名字。
 - ◆ 再次，可以确定出哪些名字的值在本块中计算而可以在后继块中被引用。
即，在dag构造的结尾仍存在于结点的标识符表中的那些名字。
- dag应用
 - ◆ 简化基本块
 - ◆ 重排基本块的计算顺序

利用dag简化基本块

- 重新生成原来基本块的一个简化的三地址语句序列。
 - ◆ 公共表达式被删除
 - ◆ 复制语句被删除
- 内部结点的计算可以按dag的拓扑排序所得的任意次序进行。
- 在计算一个结点 n 时，把它的值赋给标识符表中的一个名字 x 。
 - ◆ 应优先选择其值在块外仍需要的名字 x 。
 - ◆ 如果结点 n 的标识符表中还有其它的名字 y_1 、 y_2 、 \dots 、 y_k ，它们的值在块外也使用，则可以用语句 $y_1:=x$ 、 $y_2:=x$ 、 \dots 、 $y_k:=x$ 对它们赋值。
 - ◆ 如果某内部结点 n 的标识符表为空，那么建立新的临时变量保存 n 的值。

示例

重新生成:

(1) $T_0 := 1.5$

(2) $T_1 := 3$

(3) $T_3 := 3$

(4) $T_2 := a + b$

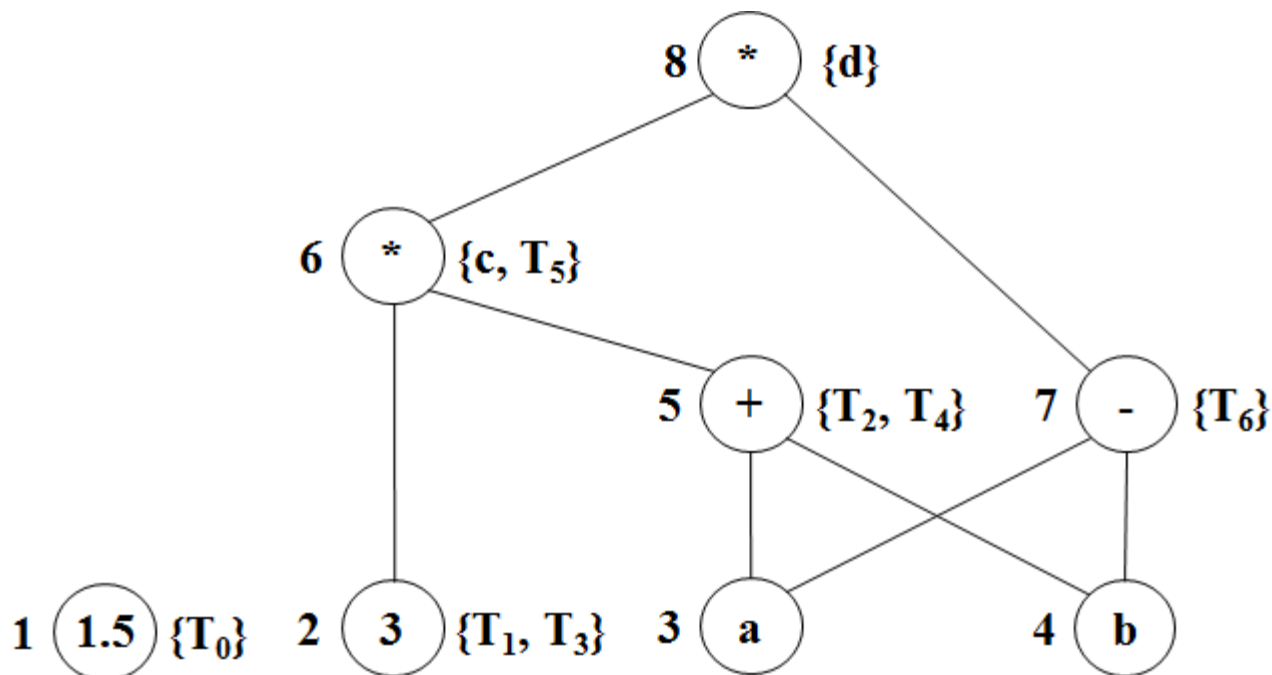
(5) $T_4 := T_2$

(6) $c := 3 * T_2$

(7) $T_5 := c$

(8) $T_6 := a - b$

(9) $d := c * T_6$



(1) $T_2 := a + b$

(2) $c := 3 * T_2$

(3) $T_6 := a - b$

(4) $d := c * T_6$

利用dag重排基本块的计算顺序

■ 基本块:

s:=a+b
t:=c+d
u:=e-t
v:=s-u

t:=c+d
u:=e-t
s:=a+b
v:=s-u

(1) MOV R₀, a
(2) ADD R₀, b
(3) MOV R₁, c
(4) ADD R₁, d
(5) MOV s, R₀
(6) MOV R₀, e
(7) SUB R₀, R₁
(8) MOV R₁, s
(9) SUB R₁, R₀
(10) MOV v, R₁

(1) MOV R₀, c
(2) ADD R₀, d
(3) MOV R₁, e
(4) SUB R₁, R₀
(5) MOV R₀, a
(6) ADD R₀, b
(7) SUB R₀, R₁
(8) MOV v, R₀

启发式排序算法

输入：基本块的dag。

输出：结点的计算顺序。

方法：利用一个栈结构保存各结点，开始时栈为空。

初始化栈顶指针；

while (存在未入栈的内部结点) {

 选取一个未入栈的、但其父结点均已入栈的结点n；

 将n压入栈顶；

 while (n的最左子结点m不是叶结点，
 并且 m的 所有父结点均已入栈) {

 将m入栈；

 n=m；

 }

}

从栈顶依次弹出结点，则得到dag的一个拓扑排序。

示例

■ 结点入栈顺序:

9

3

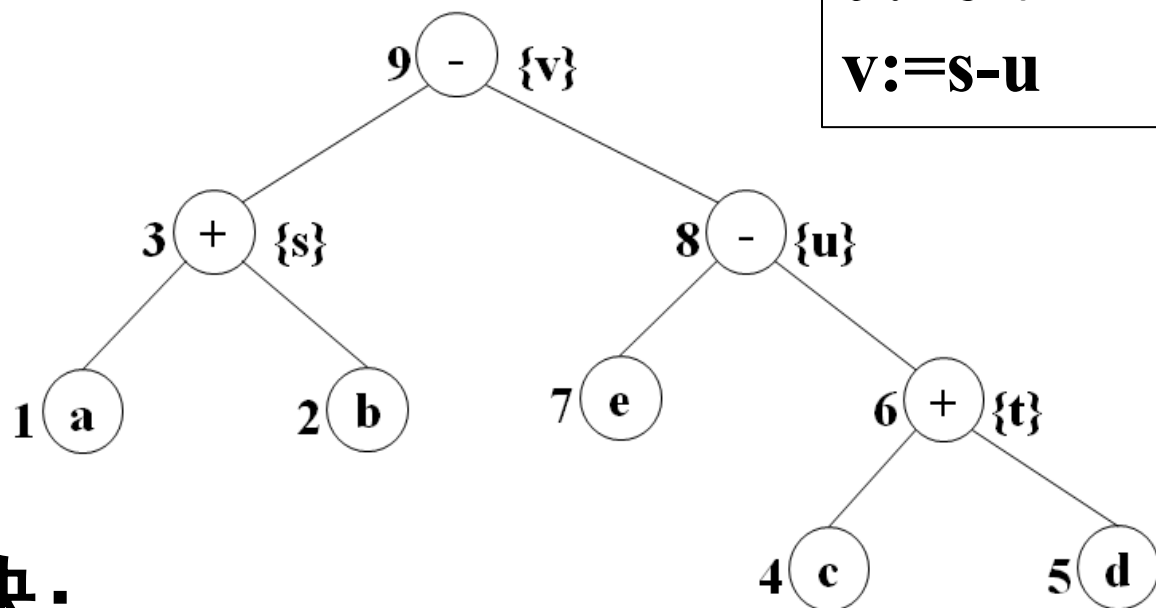
8

6

■ 重新组织基本块:

t:=c+d
u:=e-t
s:=a+b
v:=s-u

s:=a+b
t:=c+d
u:=e-t
v:=s-u



10.3.4 dag构造算法的进一步讨论

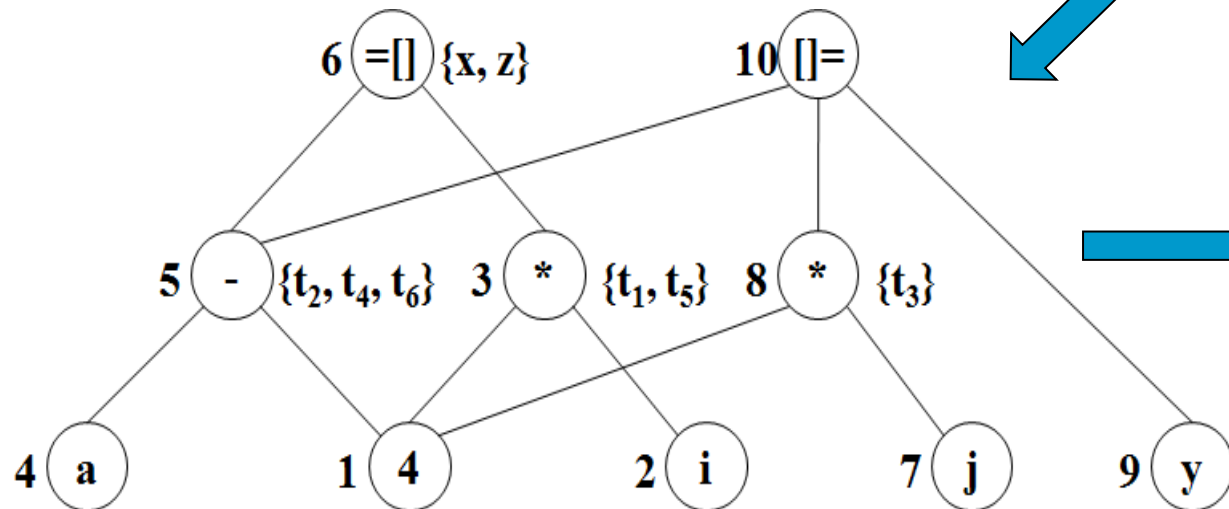
■ 考虑为数组元素赋值，如： $a[i] := x$

■ 如下程序片段：

$x = a[i];$

$a[j] = y;$

$z = a[i];$



(1)	$t_1 := 4 * i$
(2)	$t_2 := a - 4$
(3)	$x := t_2[t_1]$
(4)	$t_3 := 4 * j$
(5)	$t_4 := a - 4$
(6)	$t_4[t_3] := y$
(7)	$t_5 := 4 * i$
(8)	$t_6 := a - 4$
(9)	$z := t_6[t_5]$

?

(1)	$t_1 := 4 * i$
(2)	$t_2 := a - 4$
(3)	$x := t_2[t_1]$
(4)	$z := x$
(5)	$t_3 := 4 * j$
(6)	$t_2[t_3] := y$

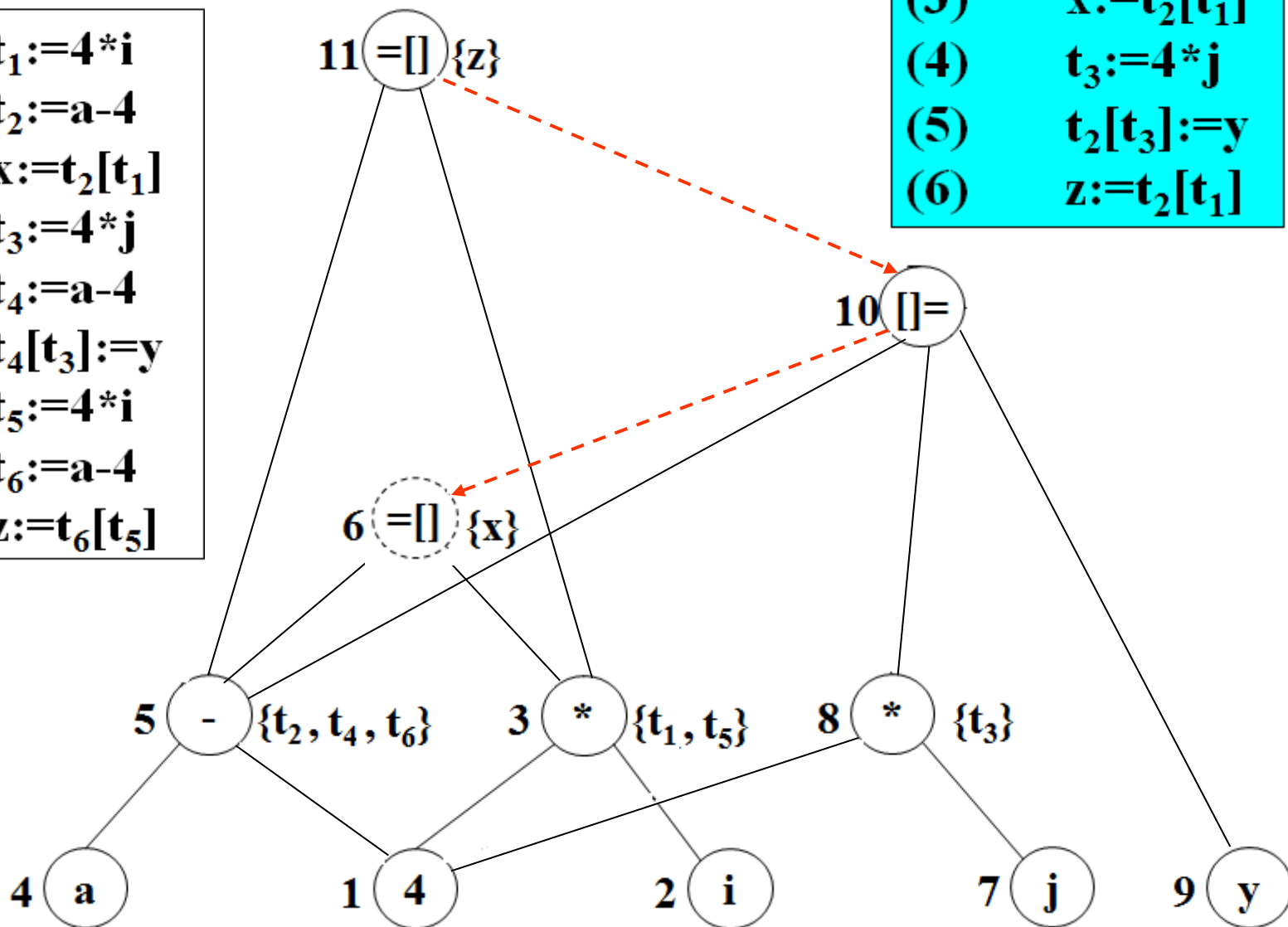
解决方案

- 构造dag过程中，当遇到为数组元素赋值的语句时，先把dag中标记为“=[]”的结点全部注销。
- 一个结点被注销，意味着在此后的dag构造过程中，不可以再选它作为已有结点来代替要构造的新结点。
 - ◆ 不可以再向被注销结点的标识符表中增加新的名字。
 - ◆ 取消了它作为公共子表达式的资格。
 - ◆ 其标识符表中原来的名字仍然存在，仍然取该结点所代表的值作为它们的值，所以，它们仍然可以被引用。

示例

- (1) $t_1 := 4 * i$
- (2) $t_2 := a - 4$
- (3) $x := t_2[t_1]$
- (4) $t_3 := 4 * j$
- (5) $t_4 := a - 4$
- (6) $t_4[t_3] := y$
- (7) $t_5 := 4 * i$
- (8) $t_6 := a - 4$
- (9) $z := t_6[t_5]$

- (1) $t_1 := 4 * i$
- (2) $t_2 := a - 4$
- (3) $x := t_2[t_1]$
- (4) $t_3 := 4 * j$
- (5) $t_2[t_3] := y$
- (6) $z := t_2[t_1]$



dag构造算法的进一步讨论（续）

- 对于指针赋值语句 $*p:=w$ 也有同样的问题，因为编译时不知道指针 p 指向哪里。
- 对于过程调用语句，由于被调用过程可能会对变量进行修改，所以，在不知道被调用过程的情况下，必须假设任何变量都可能被修改。
- 根据dag重新组织基本块代码时，必须遵守以下的限制：
 - ◆ 基本块中涉及数组元素赋值或引用的语句的相对顺序不能改变。
 - ◆ 所有其他语句相对于过程调用语句或指针赋值语句的顺序不能改变。

10.4 循环优化

- 为循环语句生成的中间代码包括如下4部分：
 - ◆ **初始化部分**：对循环控制变量及其他变量赋初值。此部分组成的基本块位于循环体语句之前，可视为构成循环的第一个基本块。
 - ◆ **测试部分**：测试循环控制变量是否满足循环终止条件。这部分的位置依赖于循环语句的性质，若循环语句允许循环体执行0次，则在执行循环体之前进行测试；若循环语句要求循环体至少执行1次，则在执行循环体之后进行测试。
 - ◆ **循环体**：由需要重复执行的语句构成的一个或多个基本块组成。
 - ◆ **调节部分**：根据步长对循环控制变量进行调节，使其增加或减少一个特定的量。可把这部分视为构成该循环的最后一个基本块。
- 循环结构中的**调节部分**和**测试部分**也可以与**循环体**中的其他语句一起出现在基本块中。

循环优化的主要技术

- 一、循环展开
- 二、代码外提/频度削弱
- 三、削弱计算强度
- 四、删除归纳变量

10.4.1 循环展开

- 以空间换时间的优化过程。
 - ◆ 循环次数在编译时可以确定
 - ◆ 针对每次循环生成循环体（不包括调节部分和测试部分）的一个副本。
- 进行循环展开的条件：
 - ◆ 识别出循环结构，而且编译时可以确定循环控制变量的初值、终值、以及变化步长。
 - ◆ 用空间换时间的权衡结果是可以接受的。
- 在重复产生代码时，必须确保每次重复产生时，都对循环控制变量进行了正确的合并。

示例： 考虑C语言的循环语句：

```
for (i=0; i<10; i++)  
    x[i]=0;
```

假定：
int x[10];
其存储空间基址： x

■ 生成三地址代码：

```
100: i:=0  
101: if i<10 goto 103  
102: goto 108  
103: t1:=4*i  
104: x[t1]:=0  
105: t2:=i+1  
106: i:=t2  
107: goto 101  
108: ...
```

空间？
执行时间？

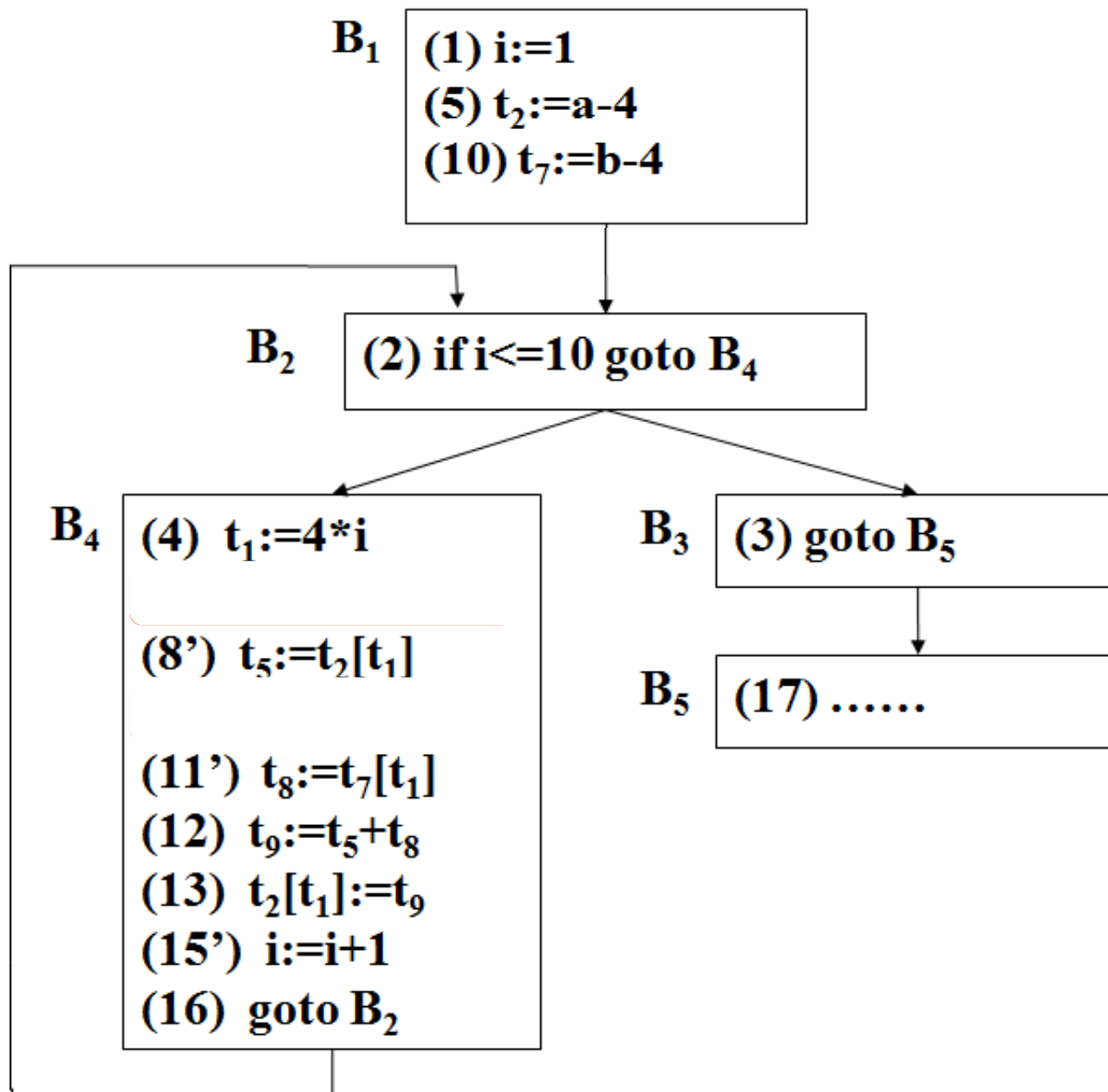
■ 循环展开：

```
100: x[0]:=0  
101: x[4]:=0  
102: x[8]:=0  
103: x[12]:=0  
104: x[16]:=0  
105: x[20]:=0  
106: x[24]:=0  
107: x[28]:=0  
108: x[32]:=0  
109: x[36]:=0
```

10.4.2 代码外提/频度削弱

- 降低计算频度的优化方法。
 - 将循环结构中的循环无关代码提到循环结构的外面（通常提到循环结构的前面），从而减少循环中的代码总数。
 - 如C语言程序中的语句：
while (i<=limit-2) {
 ...
}
- ```
t:=limit-2;
while (i<=t) {
 ...
}
```
- 如果limit的值在循环过程中保持不变，则limit-2的计算与循环无关。

例如：

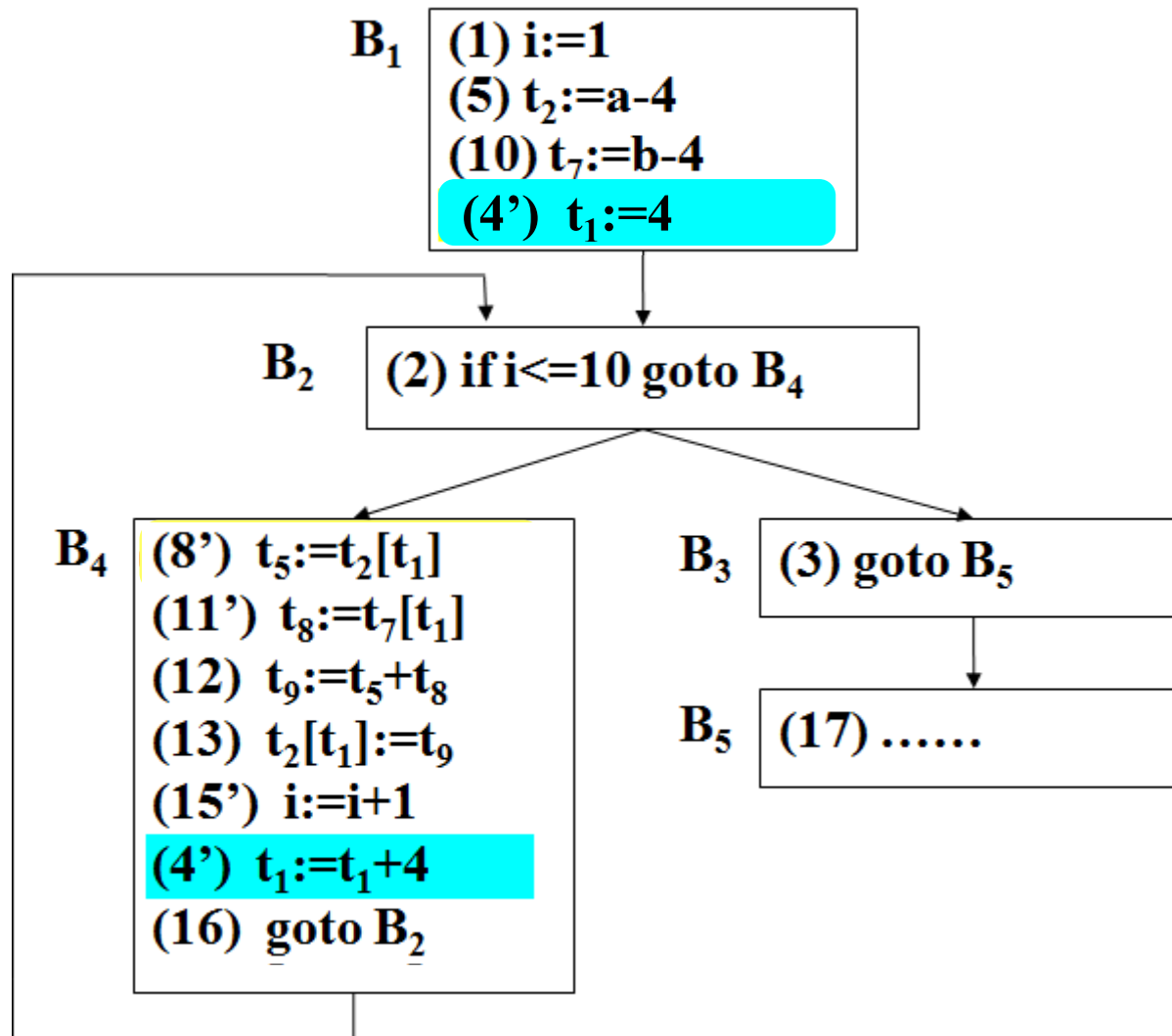




## 10.4.3 削弱计算强度

- 将当前运算类型代之以需要较少执行时间的运算类型的优化方法。
- 大多数计算机上乘法运算比加法运算需要更多的执行时间。
- 如可用 '+' 代替 '\*', 则可节省许多时间, 特别是当这种替代发生在循环中时更是如此。

例如：



## 10.4.4 删除归纳变量

- 如果循环中对变量 $i$ 只有唯一的形如  $i:=i+c$  的赋值，并且 $c$ 为循环不变量，则称 $i$ 为循环中的基本归纳变量。
- 如果 $i$ 是循环中的一个基本归纳变量， $j$ 在循环中的定值总可以化归为 $i$ 的同一线性函数，即 $j:=c_1*i+c_2$ ，这里 $c_1$ 和 $c_2$ 都是循环不变量，则称 $j$ 是归纳变量，并称 $j$ 与 $i$ 同族。
- 如：基本块 $B_4$ 中
  - ◆  $i$ 是基本归纳变量
  - ◆  $t_1:=4*i$
  - ◆  $t_1$ 是与 $i$ 同族的归纳变量

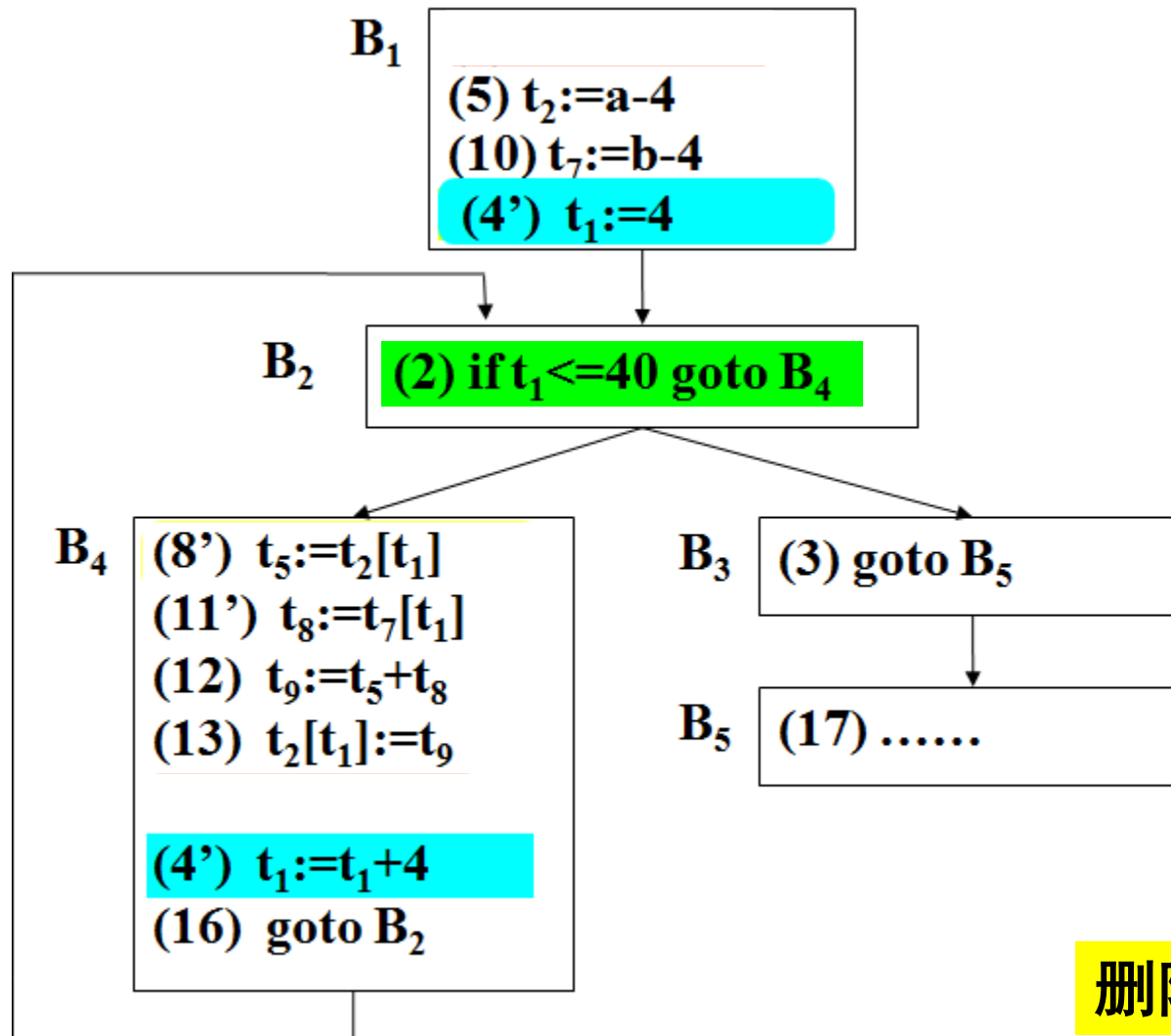
$B_4$

```
(8') $t_5:=t_2[t_1]$
(11') $t_8:=t_7[t_1]$
(12) $t_9:=t_5+t_8$
(13) $t_2[t_1]:=t_9$
(15') $i:=i+1$
(4') $t_1:=t_1+4$
(16) goto B_2
```

# 删除归纳变量 (续)

- 通常，一个基本归纳变量除用于其自身的递归定值外，往往只用于计算其他归纳变量的值、以及用来控制循环的进行。
- 由于 $t_1$ 和 $i$ 之间具有线性函数关系： $t_1=4*i$   
所以， $i \leq 10$  与  $t_1 \leq 40$  是等价的。  
因此，可以用  $t_1 \leq 40$  来替换  $i \leq 10$   
语句(2)变换为：if  $t_1 \leq 40$  goto  $B_4$

例如：



删除死代码！

# 10.5 窥孔优化

- 对目标代码进行局部改进的简单有效的技术
- 窥孔：在目标程序上设置的一个可移动的小窗口。
  - ◆ 通过窥孔，能看到目标代码中有限的若干条指令。
  - ◆ 窥孔中的代码可能不连续。
- 窥孔优化：依次考察通过窥孔可以见到的目标代码中很小范围内的指令序列，只要有可能，就代之以较短或较快的等价的指令序列。
  - ◆ 特点：每个改进都可能带来新的改进机会。
  - ◆ 通常需要对目标代码重复扫描。
- 常用技术：删除冗余指令、删除死代码、控制流优化、削弱计算强度及代数化简。
- 常作为改进目标代码质量的技术，也可用于中间代码的优化。

# 10.5.1 删除冗余的传送指令

- 如果窥孔中出现如下指令：
  - (1) `MOV R0, a`
  - (2) `MOV a, R0`
- 若这两条指令在同一基本块中，删除(2)是安全的。
  - ◆ 指令(1)的执行已经保证a的当前值同时存放在其存储单元和寄存器R<sub>0</sub>中。
- 如果指令(2)是一个基本块的入口语句，则不能删除
  - ◆ 不能保证指令(2)紧跟在(1)之后执行。

## 10.5.2 删除死代码

- 死代码：程序中控制流不可到达的一段代码。
- 如果无条件转移指令的下一条指令没有标号，即没有控制转移到此语句，则它是死代码，应该删除。
  - ◆ 删除死代码的操作有时会连续进行，从而删除一串指令。
- 如果条件转移语句中的条件表达式的值是个常量，则生成的目标代码势必有一个分支成为死代码。
  - ◆ 为了调试一个较大的C语言程序，通常需要在程序里插入一些用于跟踪调试的语句，当调试完成后，可能不删除这些语句，而只令其成为死代码。



# 示例：程序里插入的跟踪调试语句

```
#define debug 0
```

```
...
```

```
if debug {
```

```
...
```

```
/* 输出调试信息 */
```

```
}
```

- 翻译该 if 语句，得到的中间代码可能是：

```
if debug=1 goto L1
```

```
goto L2
```

```
L1: ... /* 输出调试信息 */
```

```
L2: ...
```

- 需要把从 if 到 L<sub>2</sub>所标识的语句之前的全部语句删除。

## 10.5.3 控制流优化

### ■ 连续跳转的goto语句:

goto L<sub>2</sub>

...

L<sub>1</sub>: goto L<sub>2</sub>

### ■ 条件转移语句:

if a < b goto L<sub>2</sub>

...

L<sub>1</sub>: goto L<sub>2</sub>

### ■ 如果控制结构为:

goto L<sub>1</sub>

...

L<sub>1</sub>: if a < b goto L<sub>2</sub>

L<sub>3</sub>: ...

### ■ 如果只有这一个语句转移到L<sub>1</sub>:

if a < b goto L<sub>2</sub>

goto L<sub>3</sub>

...

L<sub>3</sub>: ...

## 10.5.4 强度削弱及代数化简

- 削弱计算强度：用功能等价的执行速度较快的指令代替执行速度慢的指令。
  - ◆ 特定的目标机器上，某些机器指令比其它一些指令执行要快得多。如：
    - 用  $x*x$  实现  $x^2$  比调用指数函数要快得多。
    - 用移位操作实现定点数乘以2或除以2的幂运算比进行乘/除运算要快。
    - 浮点数除以常数用乘以常数近似实现要快等。
- 窥孔优化时，有许多代数化简可以尝试。  
但经常出现的代数恒等式只有少数几个，如：  
$$x := x + 0 \quad \text{或}$$
$$x := x * 1$$
- 在简单的中间代码生成算法中经常出现这样的语句，它们很容易由窥孔优化删除。

# 充分利用目标机器的特点

- 目标机器可能有高效实现某些专门操作的硬指令，找出允许使用这些指令的情况可明显缩短执行时间。
- 如：  
某些机器有加1或减1的硬件指令（INC/DEC），用这些指令实现语句  $i:=i+1$  或者  $i:=i-1$ ，可大大改进代码质量。

# 小结

- 代码优化程序的功能
  - ◆ 等价变换
  - ◆ 执行时间
  - ◆ 占用空间
- 代码优化程序的组织
  - ◆ 控制流分析
  - ◆ 数据流分析
  - ◆ 代码变换
- 优化种类
  - ◆ 基本块优化
  - ◆ 循环优化
  - ◆ 窥孔优化

# 小结 (续)

- 基本块优化的主要技术
  - ◆ 常数合并与常数传播
  - ◆ 删除冗余的公共表达式
  - ◆ 复制传播
  - ◆ 删除死代码
  - ◆ 削弱计算强度
  - ◆ 改变计算次序
- dag在基本块优化中的应用

# 小结（续）

## ■ 循环优化的主要技术

- ◆ 循环展开
- ◆ 代码外提/频度削弱
- ◆ 削弱计算强度
- ◆ 删除归纳变量

## ■ 窥孔优化的主要技术

- ◆ 删除冗余指令
- ◆ 删除死代码
- ◆ 控制流优化
- ◆ 削弱计算强度及代数化简