

编译原理

--代码优化

刘爽

中国人民大学信息学院

■ 优化

- 优化是指编译器通过分析编译过程中的**中间代码**，对其进行变换，以生成更好的目标代码

```
for(i = 1; i <= N; i++)  
    sum = sum + a[i] * (x*x);
```



```
t = x*x; //不变量外提  
for(i = 1; i <= N; i++)  
    sum = sum + a[i] * t;
```

循环每次迭代 $x*x$ 是不会发生变化的，可以外提
(**循环不变量外提**)

```
a = 1023*2  
b = a - y
```



```
a = 2046  
b = 2046 - y
```

a 的值可在编译时计算，并传播给 a 的使用
(**常数折叠**和**常数传播**)

■ 优化

- 优化的原则
 - 安全原则，代码变换必须是等价的
 - 代码变换应该为大多数程序生成“更好的”代码，是有利可图的
 - 代码变换是代价适宜的
 - 优化时间
 - 所需要的（内存）资源
 - 实现优化的复杂度

■ 优化

- 根据优化的范围，优化级别
 - 局部优化：基本块内
 - 全局优化：过程内
 - 全程序优化：跨过程/跨文件
- 没有“最优”的编译器
 - 优化遍互相影响
 - 公共子表达式删除vs寄存器分配
 - 指令调度vs寄存器分配
 - 一些优化本身是NP完全问题
 - 一些信息静态编译无法获取……

■ 内容

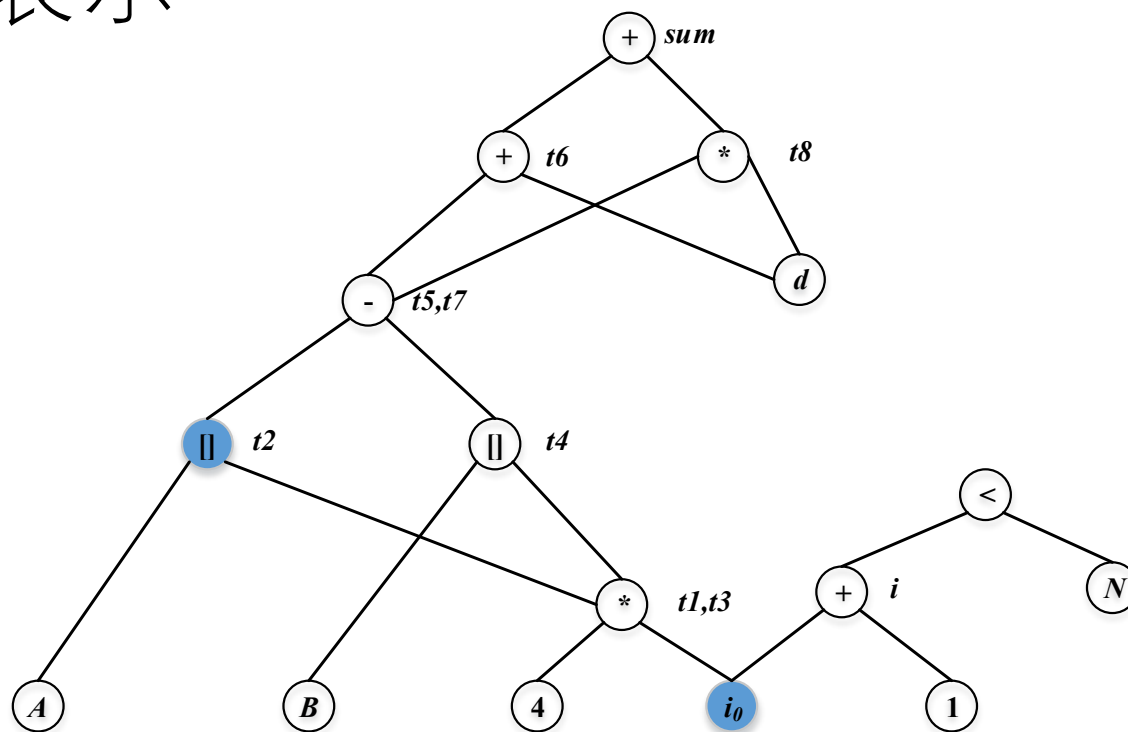
- 基本块内优化（局部优化）
 - DAG表示
 - 基于DAG的优化
- 全局优化
 - 公共子表达式删除
 - 常数折叠和传播
 - 死代码删除
 - 复制传播
- 循环优化
 - 不变量外提
 - 强度削弱
 - 循环展开

■ 一、基本块优化

- 利用有向无环图（DAG）实现基本块内的优化
 - 基本块的DAG是节点带有如下标记的有向无环图，能够清晰地展示基本块内每条语句计算的值如何在基本块内被使用
 1. **叶节点**(没有后继的节点)：以**唯一标识符(变量名)或常数**作为标记，表示该节点代表该变量或常数的值。如果叶节点用来代表某变量A的地址，则用addr(A)作为该节点的标记。有时候，我们还会把叶节点上作为标记的标识符加上下标0，以表示它是该变量的**初值**。
 2. **内部节点**(有后继的节点)：以**操作符**作为标记，表示该节点代表应用该操作符对其后继节点所代表的值进行运算的结果。
 - 节点上可能附加一个或多个**标识符**，表示这些变量具有**该节点所代表的值**

1、基本块的DAG表示

1	$t1 = 4*i$
2	$t2 = A[t1]$
3	$t3 = 4*i$
4	$t4 = B[t3]$
5	$t5 = t2 - t4$
6	$t6 = t5 + d$
7	$t7 = t2 - t4$
8	$t8 = t7 * d$
9	$sum = t6 + t8$
7	$i = i + 1$
8	if ($i < N$) goto L2



DAG

□ 叶节点 i_0 表示在该基本块入口 i 的值

□ $t2$ 的内部节点代表 $A[4*i]$

■ 1、基本块的DAG表示

- 构建DAG

基本块中间代码可分为下面三种：

① $x = y$

② $x = \text{op } y$

③ $x = y \text{ op } z$ 或 $x = y[z]$

$$\text{NODE}(\text{id}) = \begin{cases} n & \text{如果DAG中存在一个节点} n, \text{id 是该节点标记或者是附加的标识符} \\ \text{未定义} & \end{cases}$$

■ 1、基本块的DAG表示

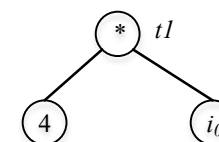
• 构造DAG算法

- 1、如果 $NODE(y)$ 未定义，则创建一个标记为 y 的叶节点，并定义 $NODE(y)$ 为这个节点；如果当前代码是③型，如果 $NODE(z)$ 未定义，则创建一个标记为 z 的叶节点并定义 $NODE(z)$ 为这个节点；
- 2、
 - (1) 如果当前代码是③型，查找是否有标记为 op ，且左儿子是 $node(y)$ 、右儿子为 $node(z)$ 的节点。如果没有，则创建一个符合条件的新节点；否则记录节点号 n
 - (2) 如果当前代码是②型，查找是否有标记为 op ，且左儿子是 $node(y)$ ，如果没有，则创建一个符合条件的新节点；否则记录节点号 n
 - (3) 如果当前代码是①型，令 $n = node(y)$
- 3、从 $node(x)$ 的附加标志符表中删除 x ，在第二步找到（或者创建的）节点 n 的附加标识符表上增加标识符 x ，并且 $node(x)=n$

1、基本块的DAG表示

第一条语句 $t1 = 4*i$ 属于③型

- 首先创建两个新节点，标记分别为4和 i_0
- 创建标记为*的新节点，将标识符 $t1$ 加入到*节点的附加标识符表中

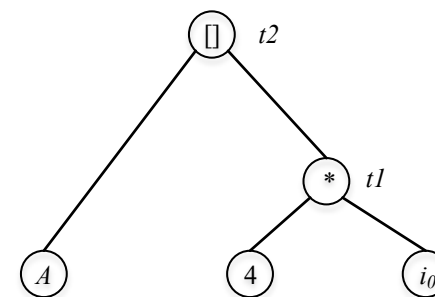


```

1  t1 = 4*i
2  t2 = A[t1]
3  t3 = 4*i
4  t4 = B[t3]
5  t5 = t2-t4
6  t6 = t5+d
7  t7 = t2-t4
8  t8 = t7*d
9  sum = t6+t8
7  i=i+1
8  if (i < N) goto L2
    
```

第二条语句 $t2 = A[t1]$ 属于③型

- 首先创建一个新节点，标记为A;
- 创建标记为[]的新节点，将标识符 $t2$ 加入到[]节点的附加标识符表中



1、基本块的DAG表示

第三条语句 $t3 = 4*i$ 属于③型

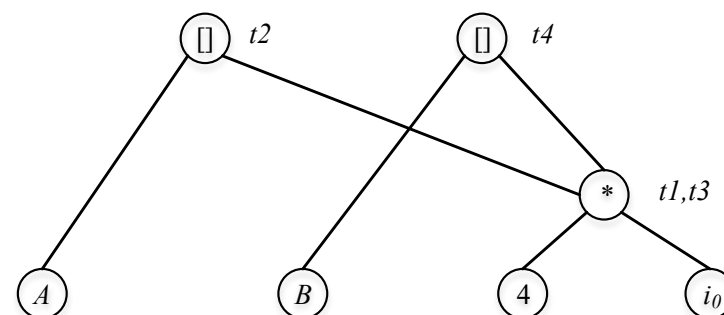
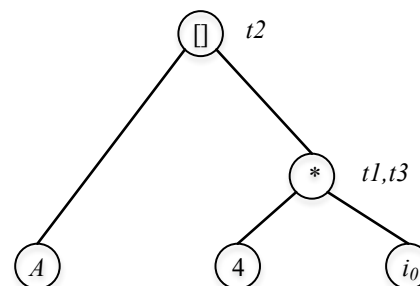
- 已有符合条件的节点，仅仅将标识符 $t3$ 加入到 $*$ 节点的附加标识符表中

```

1  t1 = 4*i
2  t2 = A[t1]
3  t3 = 4*i
4  t4 = B[t3]
5  t5 = t2-t4
6  t6 = t5+d
7  t7 = t2-t4
8  t8 = t7*d
9  sum = t6+t8
7  i=i+1
8  if (i < N) goto L2
    
```

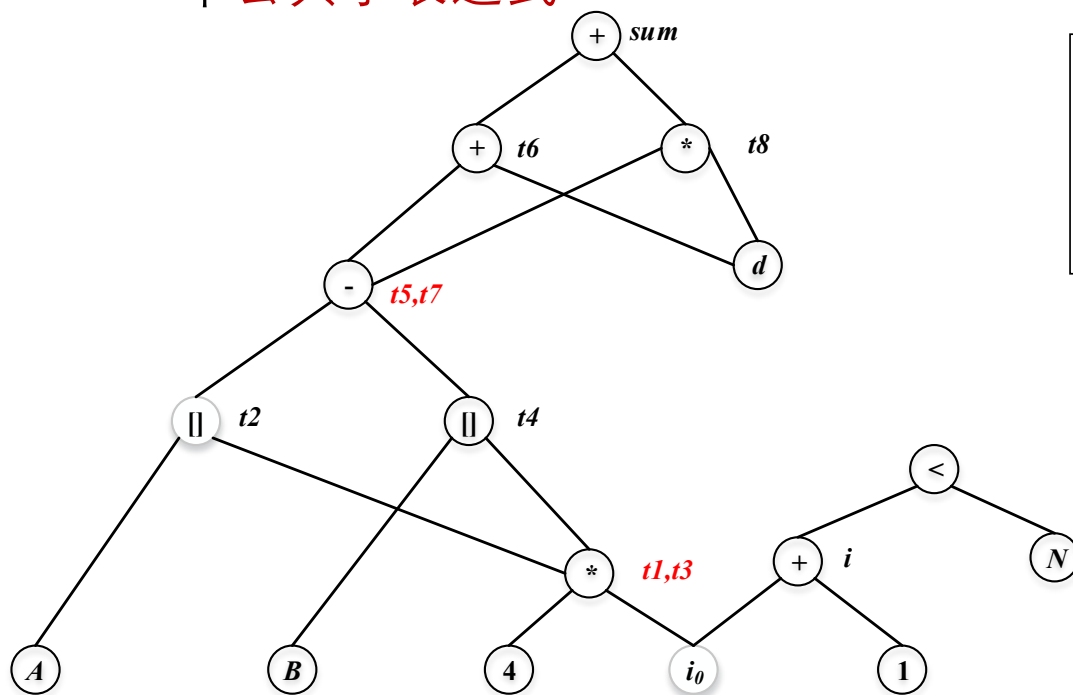
第四条语句 $t4 = B[t3]$ 属于③型

- 首先一个新节点，标记为 B ;
- 创建标记为 $[]$ 的新节点，将标识符 $t4$ 加入到 $[]$ 节点的附加标识符表中



■ 2、基于DAG优化

- 公用子表达式删除 (Common Subexpression Elimination, CSE)
 - 如果DAG某内部节点上附有多个标识符, 则表示计算该节点值的表达式是一个公共子表达式



2 t2 = A[t1]
3 t3 = 4*i
4 t4 = B[t3]

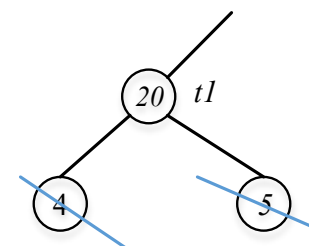
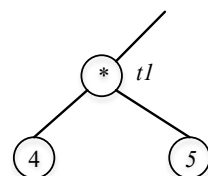


2 t2 = A[t1]
3 t3 = t1
4 t4 = B[t3]

■ 2、基于DAG优化

- 公共子表达式删除
- 常数折叠和无用代码删除

```
t1 = 4*5  
... = t1 ...
```

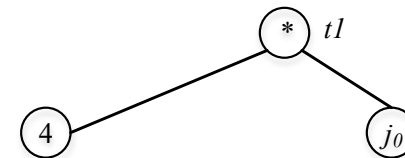
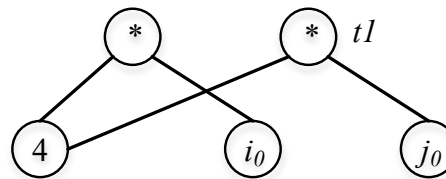
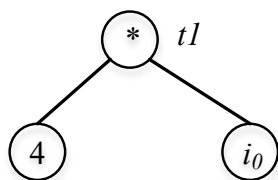


- 构造节点*, 子节点都是**常数节点**, 直接计算 $4*5$, 构造新的常数节点20
- 如果常数节点4和5是该条语句新构建的节点, 也可以删除

■ 2、基于DAG优化

- 公共子表达式删除
- 常数折叠和无用代码删除

```
t1 = 4*i
t1 = 4*j
t2 = t1+t3
```

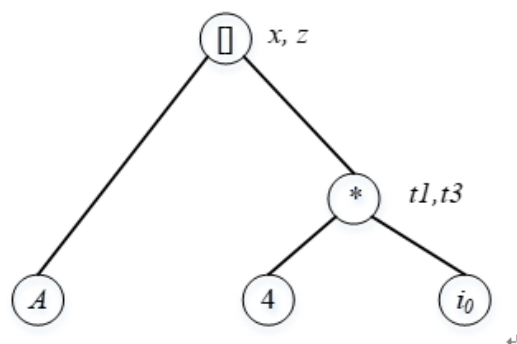


- 如果某变量被赋值后，在它被引用前又重新赋值，根据DAG构造算法的步骤3会把该变量从具有前一个值的节点上删除
- $t1 = 4*i$ 成为无用赋值，可以删除

3、数组、指针及函数调用

- 谨慎处理数组元素引用、指针和函数调用

$x = a[i]$
 $a[j] = y$
 $z = a[i]$



$a[i]$ 成为公用子表达式

$x := a[i]$
 $a[j] := y$
 $z := x$

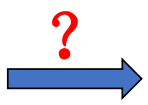
CSE优化后

$a[j]$ 和 $a[i]$ 可能指向同一存储空间，导致优化后代码错误

■ 3、数组、指针及函数调用

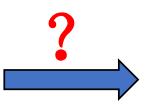
- 谨慎处理数组元素引用、指针和过程调用

- 指针赋值

t1=4*i		t1=4*i
*p = w		*p = w
t2=4*i		t2=t1
	CSE	

p可能指向t1，导致CSE优化后代码错误

- 函数调用

t1=x*i		t1=x*i
f();		f();
t2=x*i		t2=t1
	CSE	

如果缺乏函数f的信息，不得不假设f可能会修改x

■ 4、从DAG重新导出中间代码

- 当把DAG重写成中间代码时，可以进行CSE、无用代码删除等优化，并不是按照原来构造DAG节点的顺序把各节点重写为中间代码
- 为了保证程序正确性，遵循下面规定：
 - 对数组a任何元素的引用或赋值，都必须跟在原来位于其前面的对数组a任何元素的赋值之后
 - 对数组a任何元素的赋值，都必须跟在原来位于其前面的对数组a任何元素的引用之后
 - 对任何标识符的引用或赋值，都必须跟在原来位于其前面的任何过程调用或通过指针的间接赋值之后
 - 任何过程调用或通过指针的间接赋值，都必须跟在原来位于其前面的任何标识符的引用或赋值之后

■ 二、全局优化

- 全局优化作用于整个过程（函数），进行跨基本块的优化
- 为了保证优化的正确性，全局优化通常先进行数据流分析，利用数据流分析的结果进行代码变换

■ 1、公用子表达式删除

- 对程序中表达式的一次出现，如果存在该表达式的另一次出现，并且执行顺序上它总是在该表达式之前计算，并且这两个计算之间表达式的操作数没有发生变换，我们就说该表达式的这次出现是公用子表达式
- 公用子表达式删除就是用保留的结果替换重复计算。

■ 1.1 可用表达式

- 如果一个表达式 $x \otimes y$ 在点 p 满足下面的条件，我们就说该表达式在点 p 可用：
 - 从entry到 p 的每一条路径都计算 $x \otimes y$
 - 在到达 p 之前最后一次计算 $x \otimes y$ 后，没有对 x 或者 y 的赋值
- 可用表达式分析找出每个基本块 B 的边界处所有可用的表达式
- 可用表达式分析的结果用于公用子表达式的删除

■ 1.2 公用子表达式删除

首先进行可用表达式分析

对每个基本块 $B \in N$ ，执行 {

对每条 $instr \in B$ ，如果 $instr$ 形如 $z = x \otimes y$ ，且 $x \otimes y$ 在 B 的入口点可用 {

- ① 判断 $x \otimes y$ 是否在这条指令处可用
- ② 获得到达 z 的 $x \otimes y$ 计算 ($instr$ 除外)
- ③ 创建新的临时变量 t
- ④ 步骤②获得的定值指令 $w = x \otimes y$ ，替换为两条连续指令

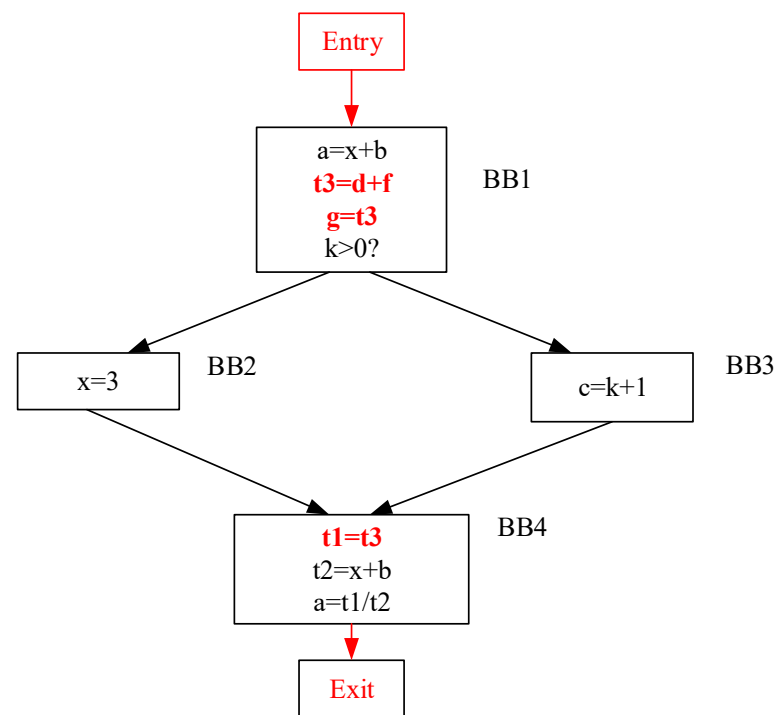
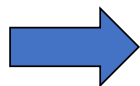
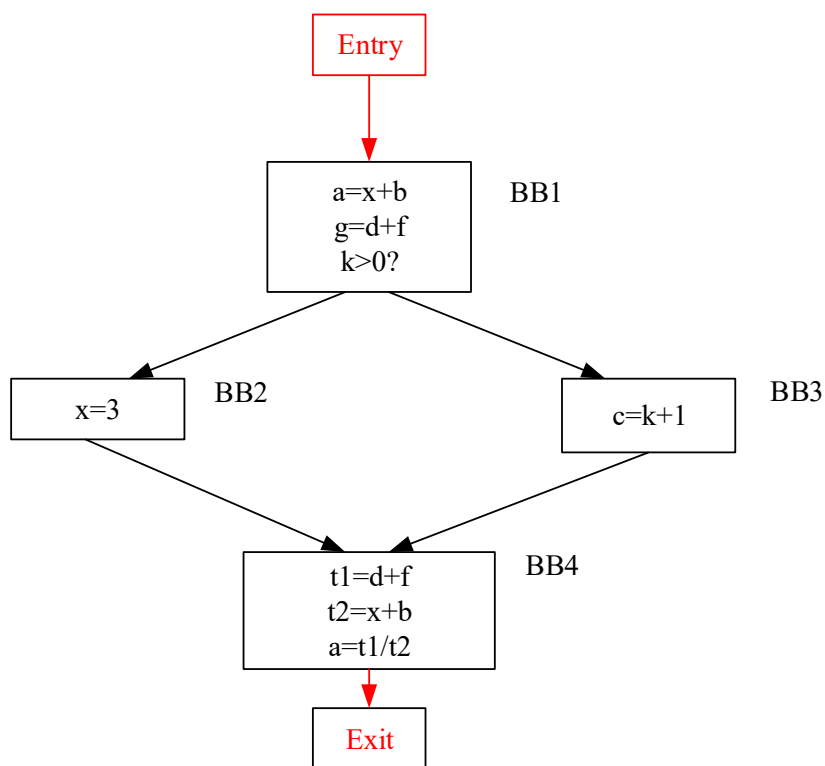
$t = x \otimes y ; w = t$

5. 将 $z = x \otimes y$ 替换为 $z = t$

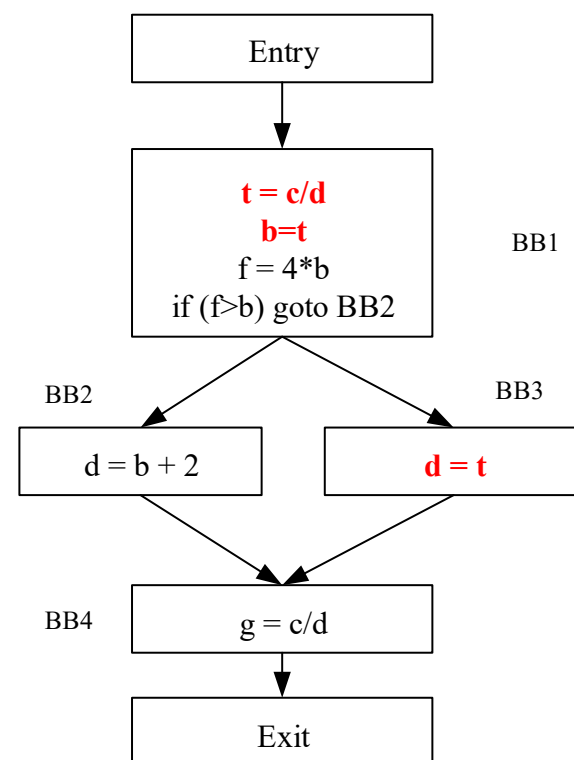
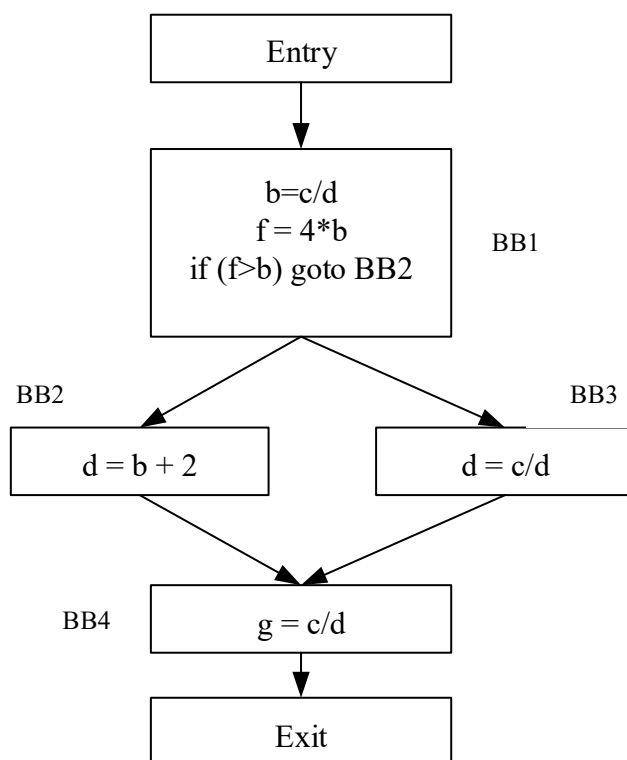
}

1.2 公用子表达式删除

- $d+f$ 在 BB4 入口点是可用表达式
- $x+b$ 在 BB4 入口点不可用



1.2 公用子表达式删除

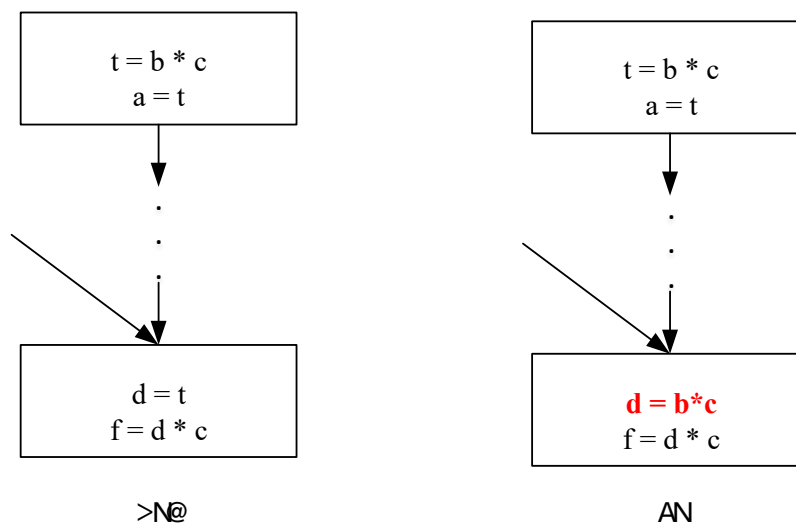


□ c/d 在BB3入口点是可用表达式

□ c/d 在BB4入口点不可用

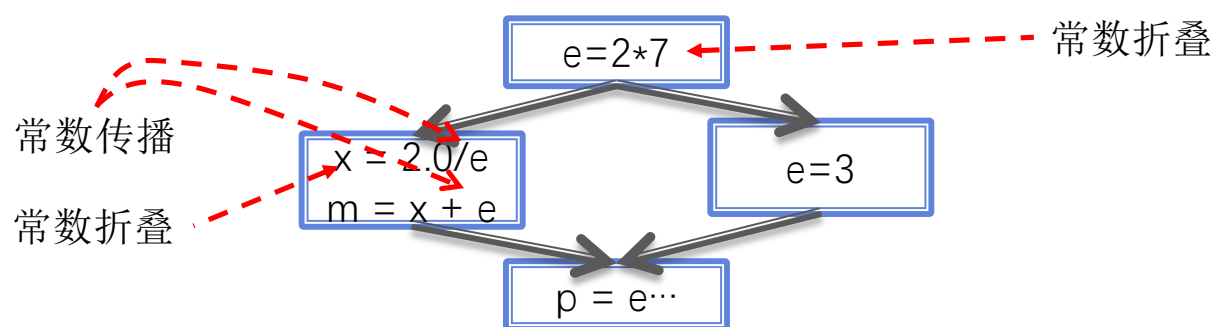
1.3 向前替代

- 向前替代 (Forward Substitution) 是CSE的逆变换。
 - 用重新计算替换复制语句



- 避免t活跃区间过长，以减少寄存器压力

2.1 常数折叠

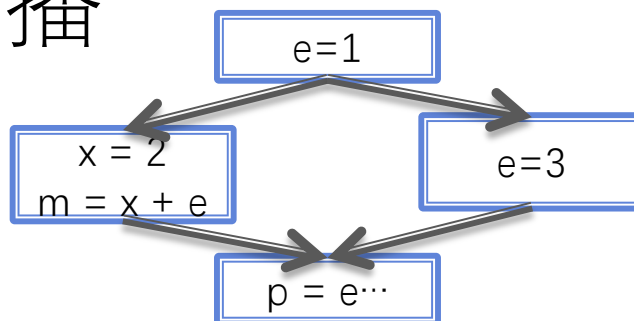


- **常数折叠**也称为常数表达式计算，指的是编译时计算**操作数是常量**的表达式
- **常数传播**是指如果变量 x 定值为常数 C ，在 x 值没有发生改变前，用 C 来替换对 x 的使用
- 在编译实现中，常数折叠往往作为单独的编译遍，和常数传播结合在一起使用
 - 常数折叠后进行常数传播，常数传播提供更多的常数折叠机会

■ 2.1 常数折叠

- **常数折叠**首先判断表达式的所有操作数都是常量，然后在编译时计算表达式，并用计算得到的结果替代该表达式
- 计算结果应该与目标机执行时得到的结果完全一致，尤其是对于交叉编译器
 - 布尔类型的表达式总是可以安全地进行常数折叠
 - 整型表达式，注意除零/溢出等异常处理
 - 浮点表达式
 - 精度、舍入模式
 - 异常处理
- 常数折叠也可以编译前端进行
- 基本块内的常数折叠和传播可以基于DAG图实现

■ 2.2 常数传播



- 常数传播在每个基本块的边界，判断每个变量是否是常数？
 - 如果是，值是多少？
- 常数传播是一个前向数据流问题

■ 2.2 常数传播

```
out[Entry] =  $V_{\text{entry}}$ ;  
for (each  $B \in N - \{\text{Entry}\}$ )  
    out[B] = 未定义 //初值  
while(change to any out occur){  
    for(each  $B \in N - \text{Entry}$ ) {  
        in[B] =  $\bigwedge_{p \in \text{pred}[B]} \text{out}[P]$  ;  
        out[B] =  $f_B(\text{in}[B])$  ;  
    }  
    常数传播, 是一个“必然”的问题,  $\bigwedge$ 是“集合交”  
}
```

■ 2.2 常数传播

- $\text{in}[S,x], \text{out}[S,x]$ 分别表示语句 S 前后变量 x 的信息。 $\text{out}[S,x] = f_s(\text{in}[S,x])$
- 边界条件有，对所有变量 x ， $\text{out}[\text{entry},x] = \text{未定义}$ 。

■ 1.1 常数传播

■ 按以下方式定义传递函数

- ⊕ 如果S不是赋值语句，那么 f_s 是单元函数，即 $f_s(x) = x$
- ⊕ 否则，假设 $S: x \leftarrow \text{RHS}$ ，对所有 $v \neq x$ 的变量，有：

$$\text{out}[S,v] = \text{in}[S,v]$$

① 如果RHS是常数 c ，则 $\text{out}[S,x] = c$

② 如果RHS形如 $y \otimes z$ ，则：

如果 $\text{in}[S,y]$ 和 $\text{in}[S,z]$ 都是常数，则 $\text{out}[S,x] = \text{in}[S,y] \otimes \text{in}[S,z]$

如果 $\text{in}[S,y]$ 和 $\text{in}[S,z]$ 中有一个不是常数，那么 $\text{out}[S,x] = \text{NAC}$

否则 $\text{out}[S,x] = \text{undef}$

③ 如果RHS是其他表达式，则 $\text{out}[S,x] = \perp$ //不是常数

■ 2.3 常数传播优化

首先进行常数传播分析

对每个基本块 $B \in N$, 执行 {

对每条 $instr \in B$, $instr$ 使用的 x , 如果在 B 的入口点 x 是常数 c {

① 判断在这条指令处 x 是否为常数 c

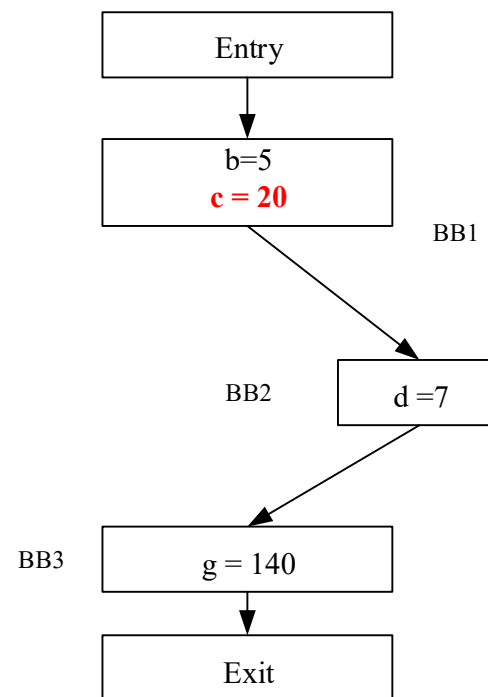
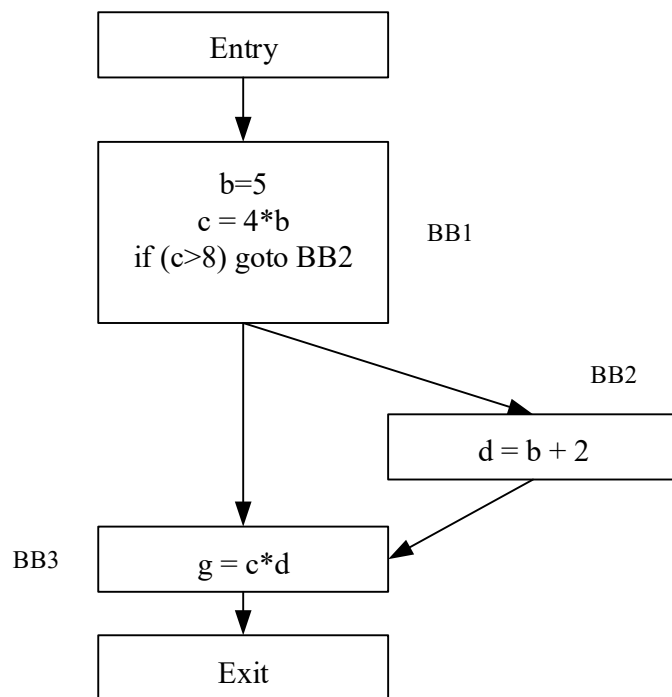
② 替换 $instr$ 中的 x 为 c

}

}

2.3 常数传播优化

b 常数 \rightarrow 常数传播 $\left\{ \begin{array}{l} c = 4 * 5 \\ d = 5 + 2 \end{array} \right. \rightarrow$ 常数折叠 $\begin{array}{l} c = 20 \\ d = 7 \end{array} \rightarrow$ 常数传播 $\text{If } (20 > 8) \dots \rightarrow$ 常数折叠 $\begin{array}{l} g = 20 * 7 \end{array}$



■ 3、死代码删除

- 死代码删除 (Dead Code Elimination, DCE) 优化删除死变量
- 死变量
 - 如果一个变量 v 在点 p 开始的某条路径上使用, 那么变量 v 在点 p 是活跃
 - 否则, 变量 v 在点 p 是死变量
- DCE优化试图删除被定值但是出口不活跃变量

■ 3.1 死代码删除

死代码删除算法

change = false;

循环{

 执行活跃变量分析;

 对每一条语句s定值变量v {

 如果活跃变量out(s)不包含v {

 删除s;

 change = true;

 }

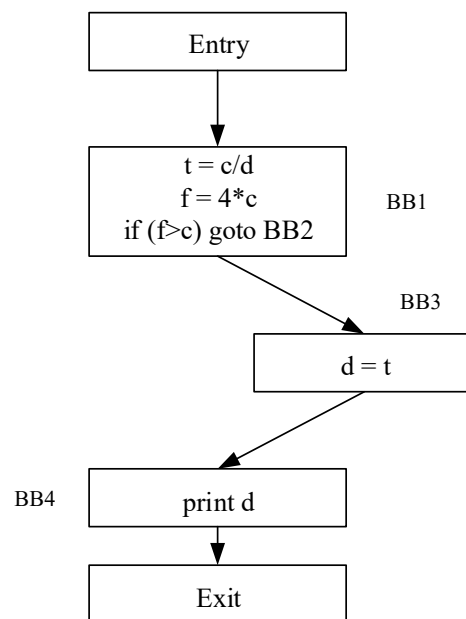
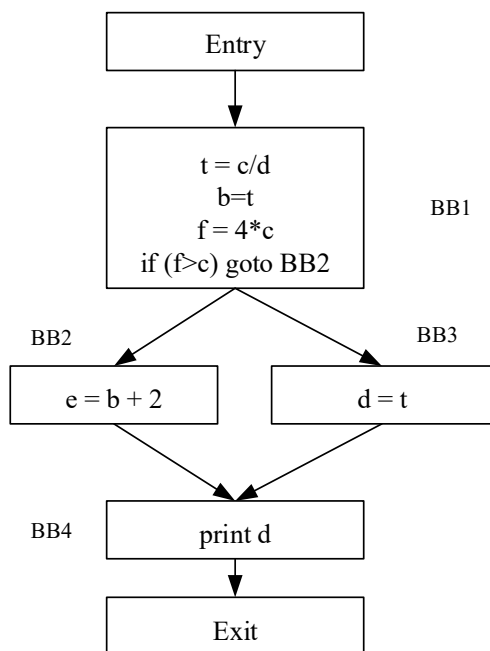
 }

}while(changed)

□如果s为 $v = x \oplus y$ ，当删除定值语句s后，s使用的变量x和y可能变成死变量

□因此删除s后，算法需要更新活跃变量信息。

3.1 死代码删除



□ e在基本块BB2出口处不活跃，可以将其在BB2中的赋值语句 $e = b + 2$ 删除

□ 删除后，对b的使用信息发生改变，b在BB1的出口处不再活跃，基本块BB1中对b的赋值语句 $b = t$ 也可以删除

3.2 死代码删除 (LLVM示例)

- C代码dce.c

```
int foo(int x, int y) {  
    int a = x+y;  
    a = 1;  
    return a;  
}
```

无dce优化 (opt dce.ll -S -mem2reg)

```
define dso_local i32 @foo(i32 %0, i32 %1) #0 {  
    %3 = add nsw i32 %0, %1  
    ret i32 1  
}
```

dce优化 (opt dce.ll -S -mem2reg -dce)

```
define dso_local i32 @foo(i32 %0, i32 %1) #0 {  
    ret i32 1  
}
```

■ 3.2 死代码删除 (LLVM示例)

- C代码dce2.c

```
int b; //global variable
int foo(int x, int y) {
    int a = x+y;
    b = a;
    return x;
}
```

dce优化 (opt dce2.ll -S -mem2reg -dce)

```
define dso_local i32 @foo(i32 %0, i32 %1) #0 {
    %3 = add nsw i32 %0, %1
    store i32 %3, i32* @b, align 4
    ret i32 %0
}
```

DCE优化不能删除对全局变量b的写，因此不能删除对a的赋值

■ 3.2 死代码删除 (LLVM示例)

- 下面的C代码dce3.c?

```
int b; //global variable
int foo(int x, int y) {
    volatile int a = x+y;
    b = a;
    return x;
}
```

dce优化 (opt dce3.ll -S -mem2reg -dce)

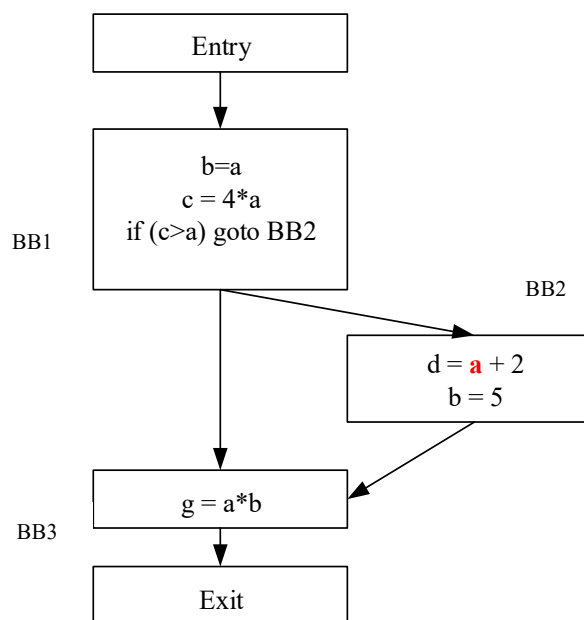
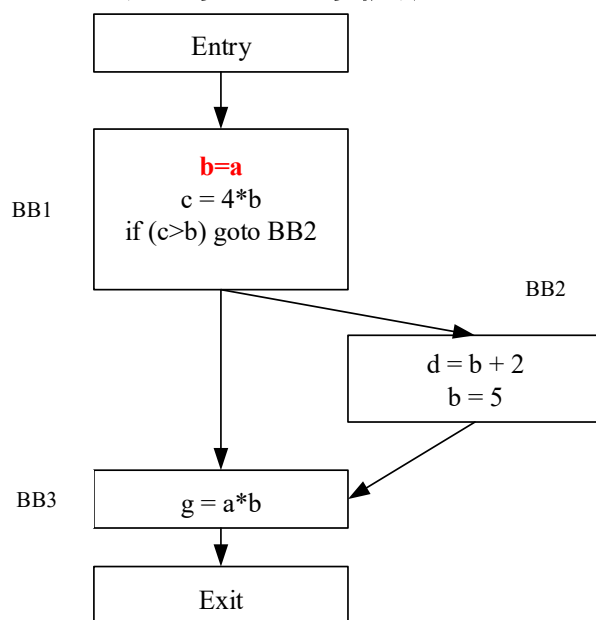
```
define dso_local i32 @foo(i32 %0, i32 %1) #0 {
    %3 = alloca i32, align 4
    %4 = add nsw i32 %0, %1
    store volatile i32 %4, i32* %3, align 4
    %5 = load volatile i32, i32* %3, align 4
    store i32 %5, i32* @b, align 4
    ret i32 %0
}
```

DCE优化通常不删除对
volatile 变量的读写

4. 复制传播

- 复制传播 (Copy Propagation)

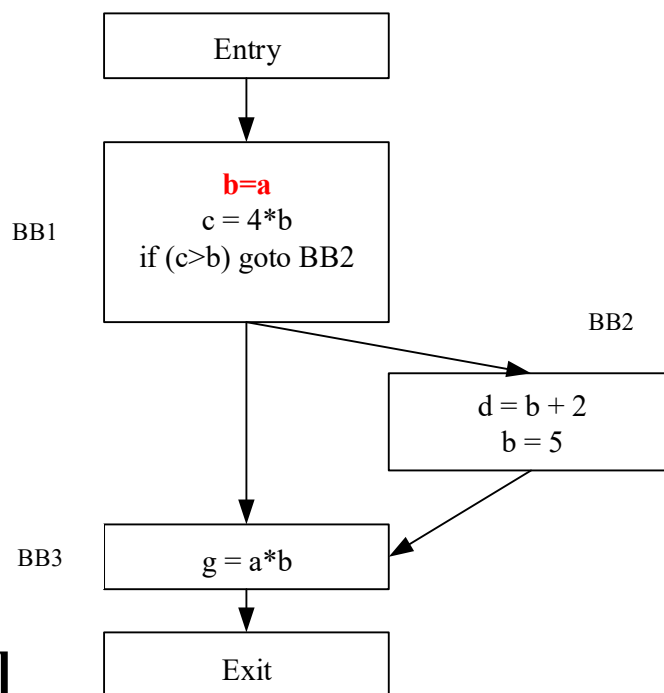
- 复制指令：形如 $x=y$ 的定值语句
- 在 x 和 y 的值都没有发生改变之前（即没有对 x 和 y 定值），用对 y 的使用代替后面出现的对 x 的使用



■ 4. 复制传播

- 传播复制指令S: $x=y$
 - 判断x的定值的所有使用
 - 对每个使用u,
 - 复制指令S必须是唯一的可以到达u的定值, 并且
 - 从S到u的每一条路径上, 都没有对y的复制
- 可用的复制表达式计算与可用表达式计算类似

4.1 可用复制指令



$$in[B] = \bigcap_{P \in pred(B)} out[P]$$

$$out[B] = copy(B) \cup (in[B] - kill[B])$$

- $copy(B)$: 在B中形如 $x=y$ 的复制指令，并且在B中该指令之后，没有对x或y的复制
- $Kill(B)$: 被B杀死的复制指令
- $in[B]$: 在基本块B的入口可用的复制指令集合
- $out[B]$: 在基本块B的出口可用的复制指令集合

■ 4.2 复制传播

- 传播复制指令S: $x=y$
- 全局的复制传播
 - 对每个基本块B, 如果复制指令 $x=y \in \text{in}[B]$, 执行复制传播, 直到复制指令被基本块中的其他指令杀死

■ 四、循环优化

- 循环往往会迭代执行很多次，执行时间占整个程序执行时间大部分
- 冗余删除优化
 - 循环不变量外提
 - 强度削弱
 - 删除归纳变量等
- 简单循环变换
 - 循环展开
- 高级循环变换
 - 改变循环迭代执行顺序
 - 循环置换、循环铺砌 (Tiling) 等

■ 1. 循环不变量外提

- 如果循环中的一个计算在循环的每次迭代都产生相同的值，则该计算被称作循环不变量。
- 当循环的基本块的一个操作的操作数都满足下列条件时，该操作是循环不变的：
 - 操作数是常数，或者
 - 所有到达操作数的定值都在循环之外，或者
 - 只存在一个到达操作数的定值，并且该定值在循环内，并且该定值本身是循环不变的

■ 1.1 循环不变量

- 标记循环中的**不变量**集合

1、首先标记两种**简单的不变量**

- ①标记所有操作数都是**常数**的指令为不变的
- ②标记所有操作数的**到达定值都在循环外**的指令为不变的

2、循环直到找出了所有的**循环不变量**：

标记操作数符合下面条件，并且没有被标记的指令，为不变的

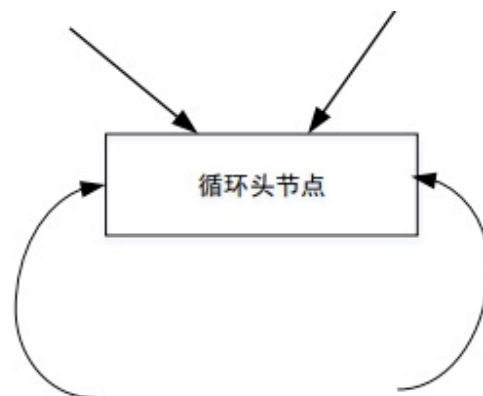
- ①操作数是**循环不变量**，或者
- ②只有**一个定值**到达操作数，并且该定值已经标记为**循环的不变量**

■ 1.2 循环前置节点

■ 思考？

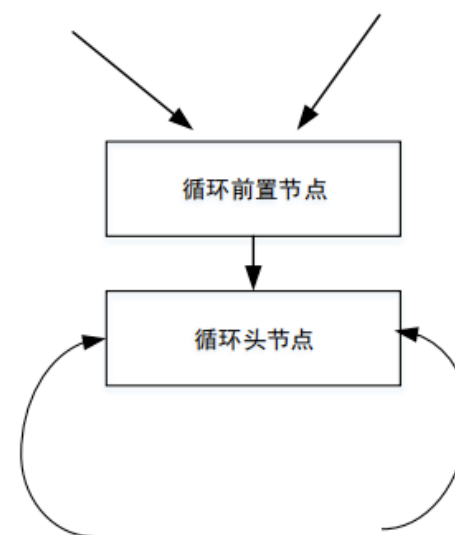
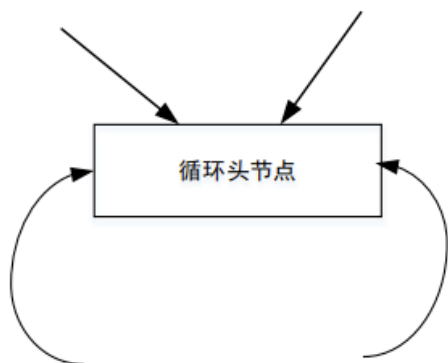
不变量外提后，放在哪里？

- 循环头节点？循环头节点的前驱？



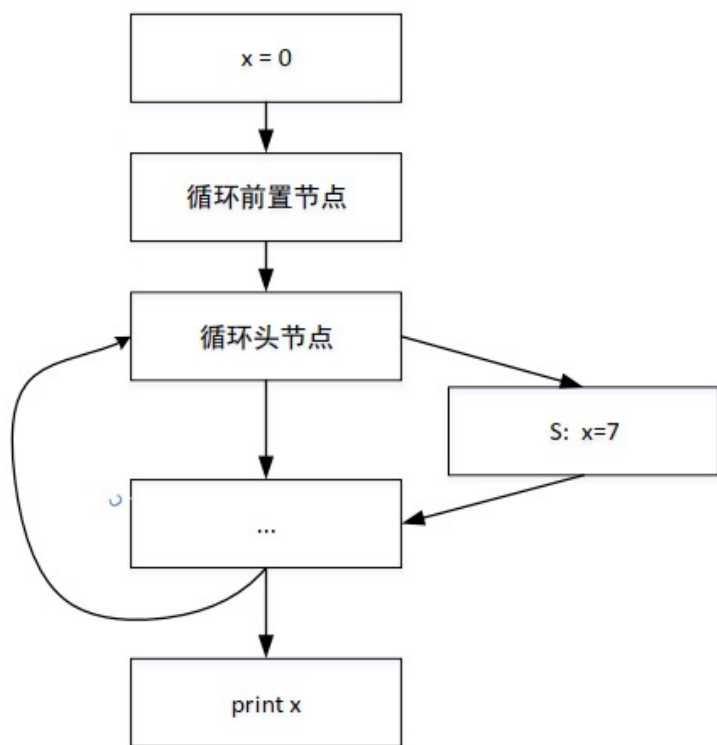
1.2 循环前置节点

- 循环前置节点
 - 只有一个后继：循环头节点
 - 来自于循环之外、到循环头节点的边，改为到前置节点
 - 来自于循环内部，到循环头节点的边，保持不变



1.3 循环不变量外提

- 循环不变量移动是将发现的不变量外提到循环前置节点。
- 所有不变量都可以外提么？



`x=7` 可以外提吗？

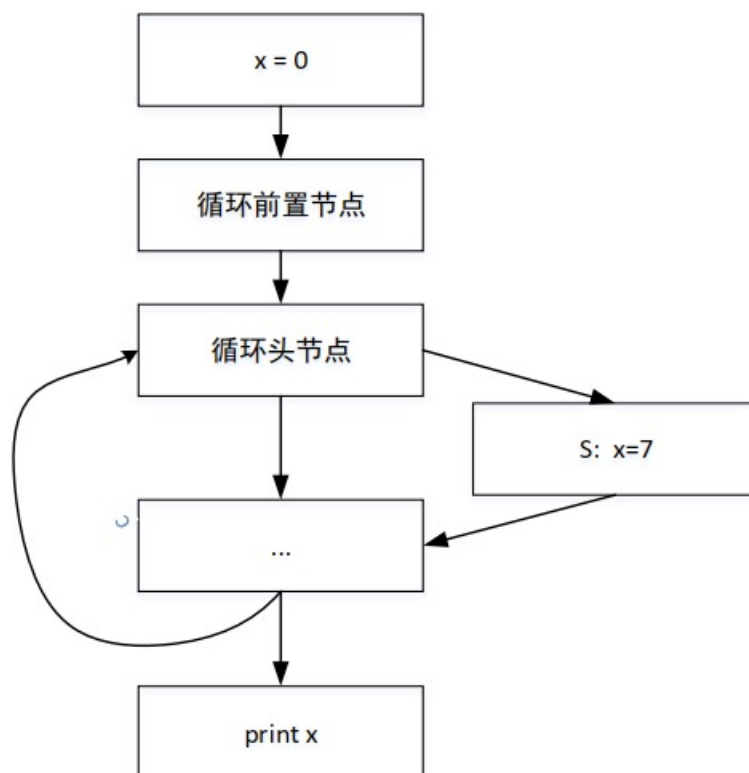
■ 1.3 循环不变量外提

- 循环不变量移动是将发现的**不变量外提到循环前置节点**。
- 循环不变量外提条件

假设不变量s: $x = y \otimes z$, 对**x**进行定值, 使用**y**、**z**

■ 1.3 循环不变量外提

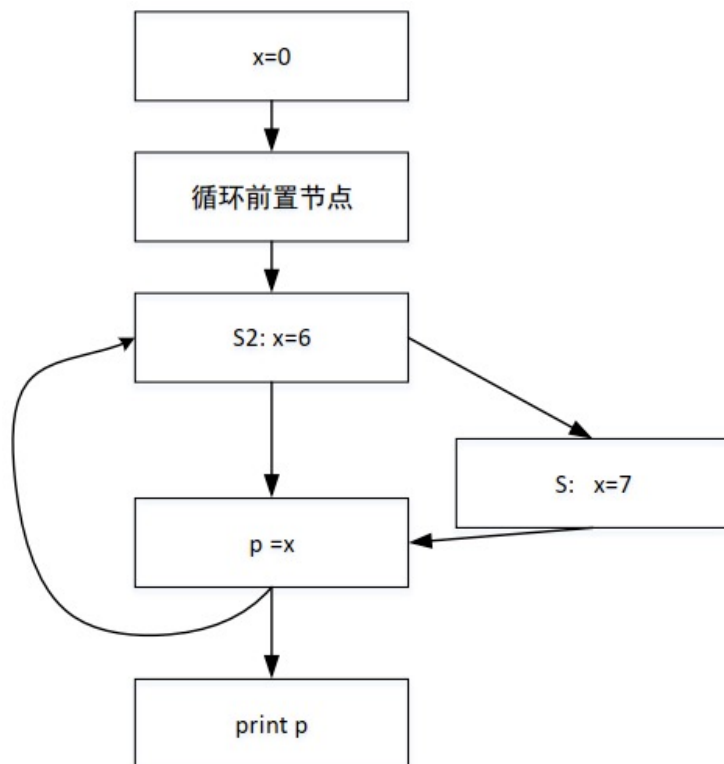
- 不变量外提条件一



条件一：语句**s**是循环**L****所**
有出口的必经节点

1.3 循环不变量外提

- 不变量外提条件二

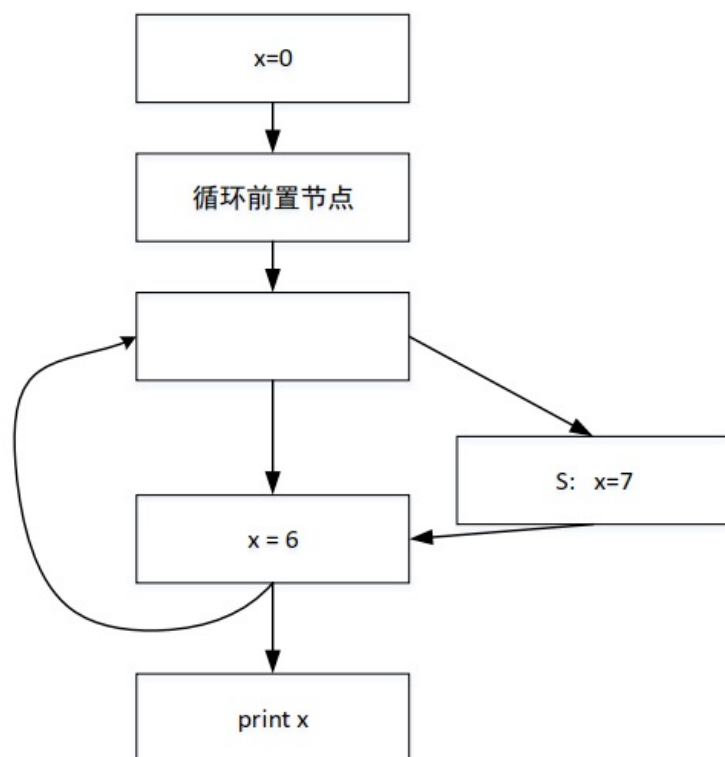


S2: $x=6$ 满足条件一，但不能外提

条件二：循环中对 x 的所有使用
只能被 s 中的定值到达

1.3 循环不变量外提

- 不变量外提条件三

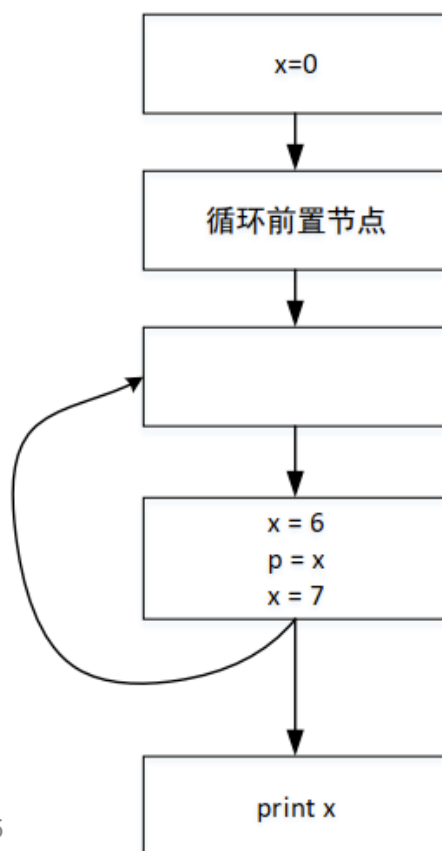


S: $x=7$ 不满足条件一, 不能外提
 $x=6$ 满足条件一和二, 但外提可能导致结果错误

条件三: **v**不能在循环L内有
其他定值

1.3 循环不变量外提

- 不变量外提条件四



根据前三个条件, $p=x$ 被外提, 但是 $x=6$ 和 $x=7$ 不能外提, 显然发生了错误, 因此限制条件四: 考虑不变量使用的操作数

条件四: 不变量的使用的 x 、 y 的定值都来自于循环之外

(可以是一开始就在循环外, 或者外提后在循环外)

■ 1.3 循环不变量外提

执行到达定值分析，并构建UD链

找出循环L中的循环不变量

对每个循环不变量指令s，假设s定值v，使用x和y，执行{

如果 (1. s所在基本块是L的所有出口的必经节点 &&

2. 在循环L中v没有其他定值 &&

3. 并且L中v的所有使用只被s中的定值到达 &&

4. x和y的定值在循环外 (可以是一开始就在循环外，或者外提后在循环外)

{

将s外提，移到L的前置节点}

}

}

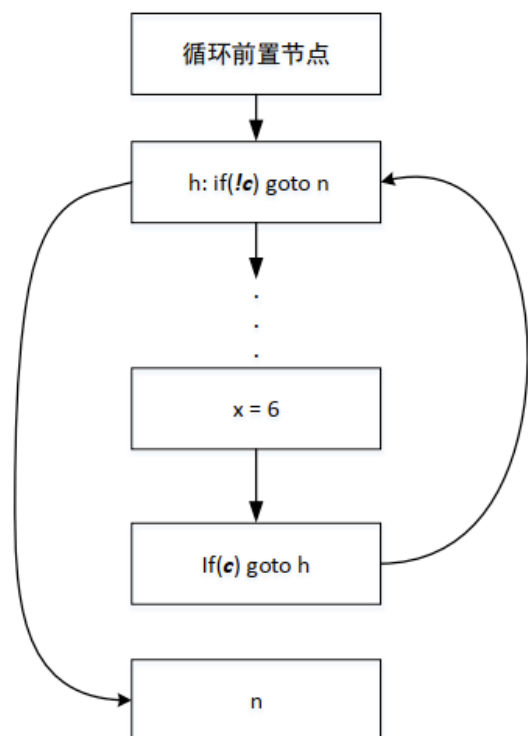
2025/5/25

刘爽，中国人民大学信息学院

54

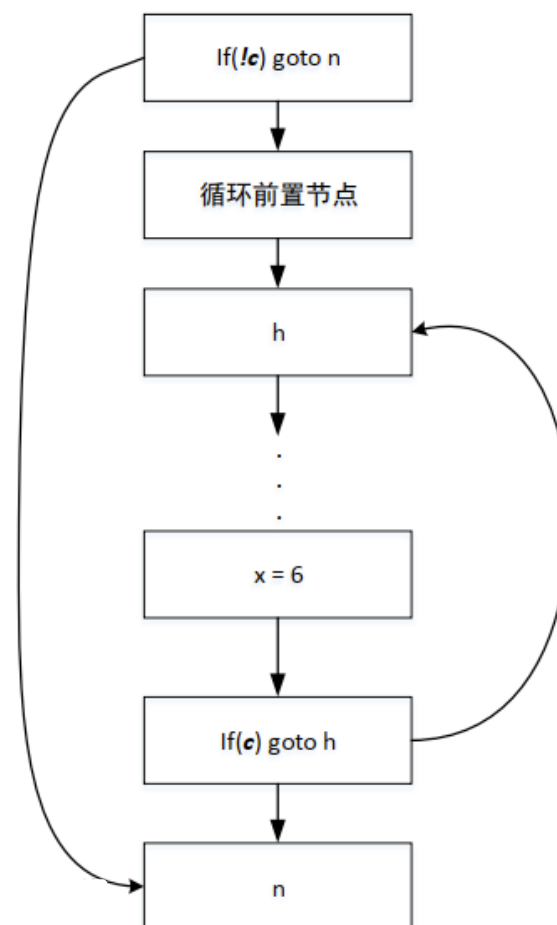
1.3 迭代次数为0?

- 如果循环迭代次数为0, 怎么办?
- 由先测试循环是否执行来避免



2025/5/25

刘爽, 中国人民大学信息学院



55

■ 2. 归纳变量与强度削弱

- 归纳变量 (Induction Variable)

- 在循环中，如果变量x的值每次改变都是增加或者减少某个固定的常数，那么x就是循环的归纳变量

```
for (i=0; i<N; i++){  
    j = 2*i + 1;  
    a[i]=j  
}
```



```
for (i=0; i<N; i++){  
    j += 2;  
    a[i]=j  
}
```

- 循环控制变量i每次递增1
- $j=2*i+1$ ，每次递增2

强度削弱

■ 2.1 识别归纳变量

- 基本归纳变量
 - 只有唯一的形如 $i := i \pm C$ 的定值， C 是常数
- 导出归纳变量
 - 变量 j 在循环中的定值总是可化归为归纳变量 i 的同一线性函数，即 $j = C1 \oplus \pm C2$ ，其中 $C1$ 和 $C2$ 都是循环不变量
- j 是与 i 同族的导出归纳变量

2.1 识别归纳变量

1 找出循环L中的循环不变量

$IV = \Phi$

对循环L中每一条语句S，循环

如果（S形如 $i \leftarrow i \pm C$ 并且C是循环不变量）{

5 $IV = IV \cup \{i\};$ // i是基本归纳变量

与i关联的三元组为 $(i, 1, \pm C)$;

}

如果（变量j在循环L中只有一次定值，并且定值语句S具有以下形式之一：

$j = C*k, j = k*C, j = k/C, j = k \pm C, j = C \pm k$ ，其中k是归纳变量，C是循环不变量）{

如果（k是基本归纳变量）{

10 $IV = IV \cup \{j\};$

根据S，计算与j关联的三元组：

case $j = C*k$ 或者 $j = k*C$: 三元组为 $(k, C, 0)$

case $j = k \pm C$: 三元组为 $(k, 1, \pm C)$

case $j = k/C$: 三元组为 $(k, 1/C, 0)$

15 case $j = C+k$: 三元组为 $(k, 1, C)$

case $j = C-k$: 三元组为 $(k, -1, C)$

} 否则{

假设k属于i族（i是基本归纳变量）

如果（循环L中对j的定值和对k的定值之间没有修改i，并且循环L外没有k的定值到达S）{

20 $IV = IV \cup \{j\};$

根据S和与k关联的三元式 (i, C_1, C_2) 计算与j关联的三元式

}

}}

//(1) 识别基础归纳变量

//(2) 识别导出归纳变量

//三元式模板为 (k, C_1, C_2)

//表示计算 $C_1*k + C_2$

■ 2.2 强度削弱

- 用代价较低的指令替换代价较高的指令
- 强度削弱算法

依次处理每一个基本归纳变量 i ，对每一个与 i 同族的归纳变量 j ，其三元式为 $(i, C1, C2)$ ，执行以下步骤：

1. 建立新变量 s ，如果归纳变量 j_1 和 j_2 三元式相同，则只创建一个新变量
2. 用 $j=s$ 代替原来对 j 的赋值
3. 在 L 中，紧跟在 $i=i+C$ （ C 是循环不变量）语句之后，插入语句

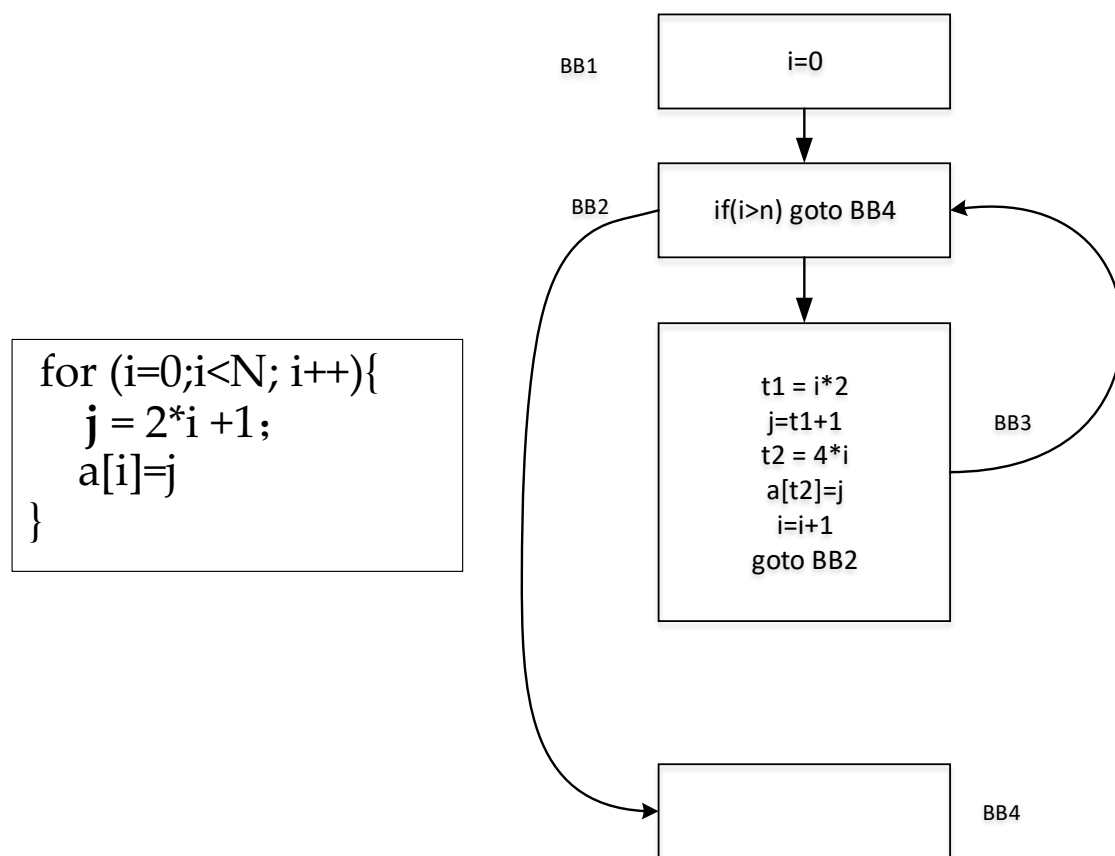
$$s = s + C1 * C$$

如果 $C1 * C$ 是常数，则计算常数值； $C1 * C$ 是循环不变量，则引入临时变量 $t = C1 * C$ ， $s = s + t$ ；

通过不变量外提优化，外提到循环前置节点；

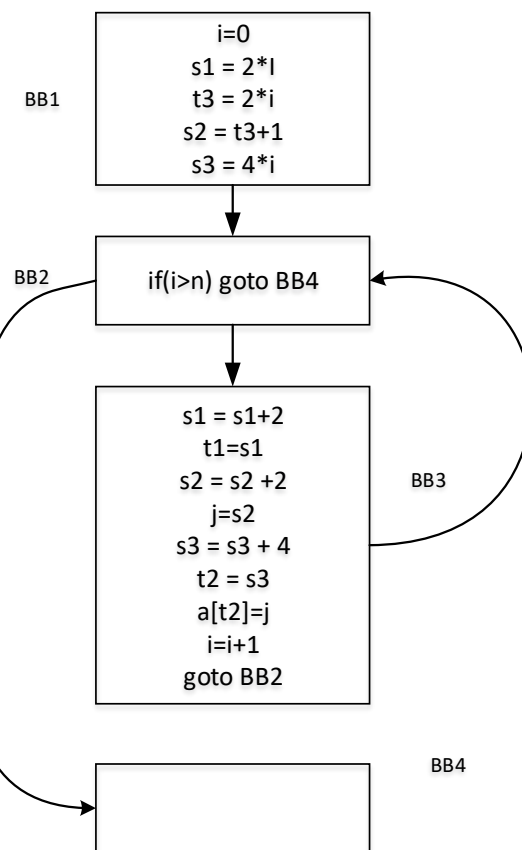
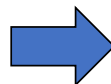
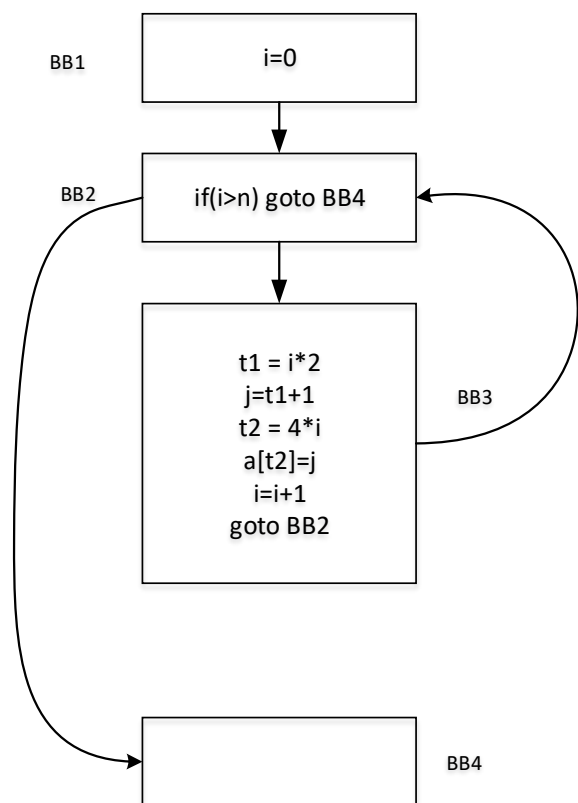
4. 在前置节点末尾为 s 设置初值 $s = C1 * i + C2$
5. 用 s 替换源程序中每一个 j

2.2 强度削弱



- i是基本归纳变量
- t、j、t2都是i的同族归纳变量
 - t1: 三元式 (i,2,0)
 - j: 三元式(i,2,1)
 - t2: 三元式(i,4,0)

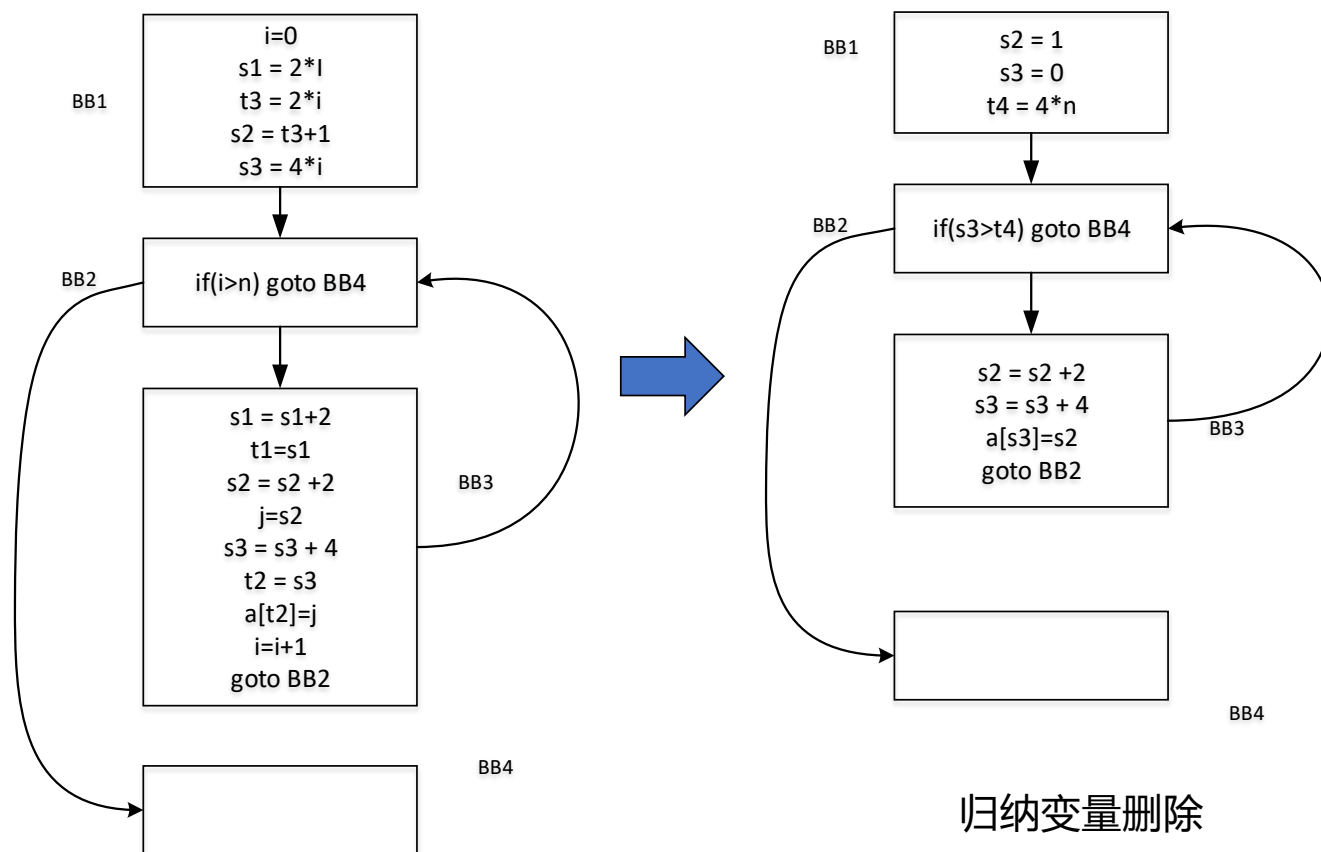
2.2 强度削弱



强度削弱

2.2 归纳变量删除

- 强度削弱后，通过常数传播优化，可以对代码进行进一步优化
- 归纳变量*i*只用于分支测试和自身定值，并没有其他使用，可以利用与*i*同族的其他归纳变量作为测试变量，从而删除归纳变量*i*
- 复制传播进一步删除归纳变量



■ 3. 循环展开

- 简单循环变换，不改变迭代执行顺序
- 分为**头尾展开**和**体展开**
 - 头尾展开指的是将循环的首迭代或者尾迭代删除，形成单独的代码，也被称为**循环剥除** (Loop Peeling)

```
for ( i=1; i < n; i++) {  
    if (i == 1)  
        a(i) = 0;  
    else  
        a(i) = b(i-1);  
}
```



```
a(1) = 0;  
for ( i=2; i < n; i++) {  
    a(i) = b(i-1)  
}
```

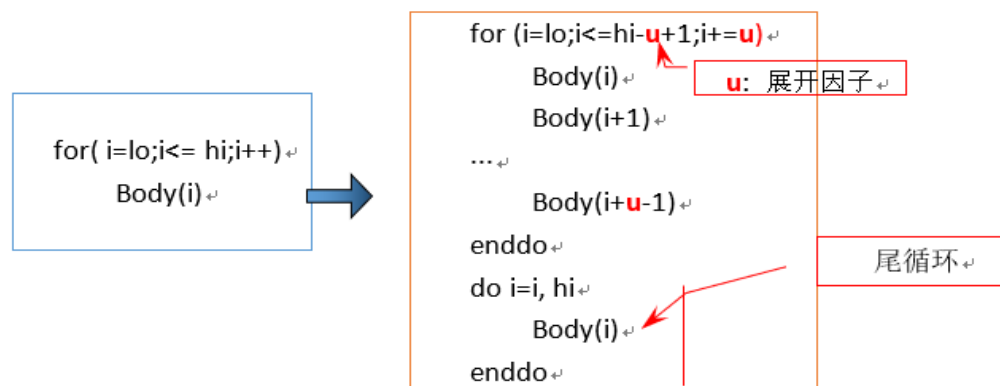
- 删除和循环控制变量相关的条件分支语句，简化循环体

应用场景

- 向量化 (**SIMD**) 前优化：剥离不规则的边界元素。
- 移除条件分支：提高分支预测命中率。
- 循环展开前简化控制结构。

3. 循环展开

• 体展开



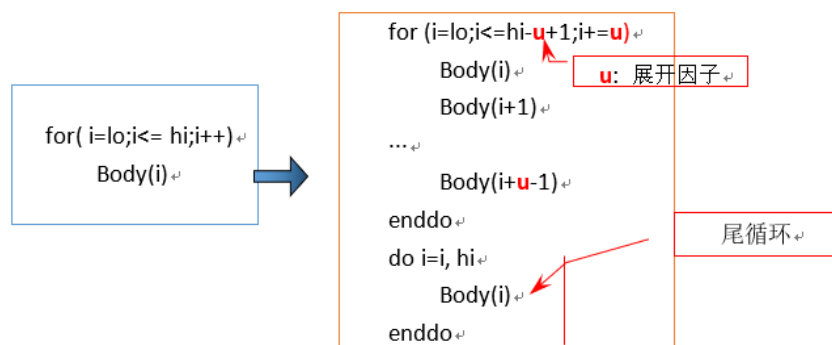
- 扩大循环体，为公用子表达式删除、强度削弱、指令调度等提供更多的机会，提高数据局部性

<https://www2.cs.uh.edu/~jhuang/JCH/JC/loop.pdf>

3. 循环展开

• 循环展开优化框架：

- ① 根据循环展开代价模型，判断循环展开收益，选择需要展开的循环
- ② 选择合适的展开因子 u
- ③ 将循环体复制 u 份，调整相应的循环索引变量
- ④ 形成和处理尾循环



第①和②步是循环展开的关键，考虑：

- ❑ 目标机的体系结构,例如，硬件寄存器的个数，指令Cache的大小和组织结构，以及指令流水线等
- ❑ 与循环本身有关，包括循环体的大小、操作数的类型和个数