第十四章 代码优化

- 概述
- 基本块和流图
- 基本块内优化
- 全局优化
- 循环优化

14.1 概述

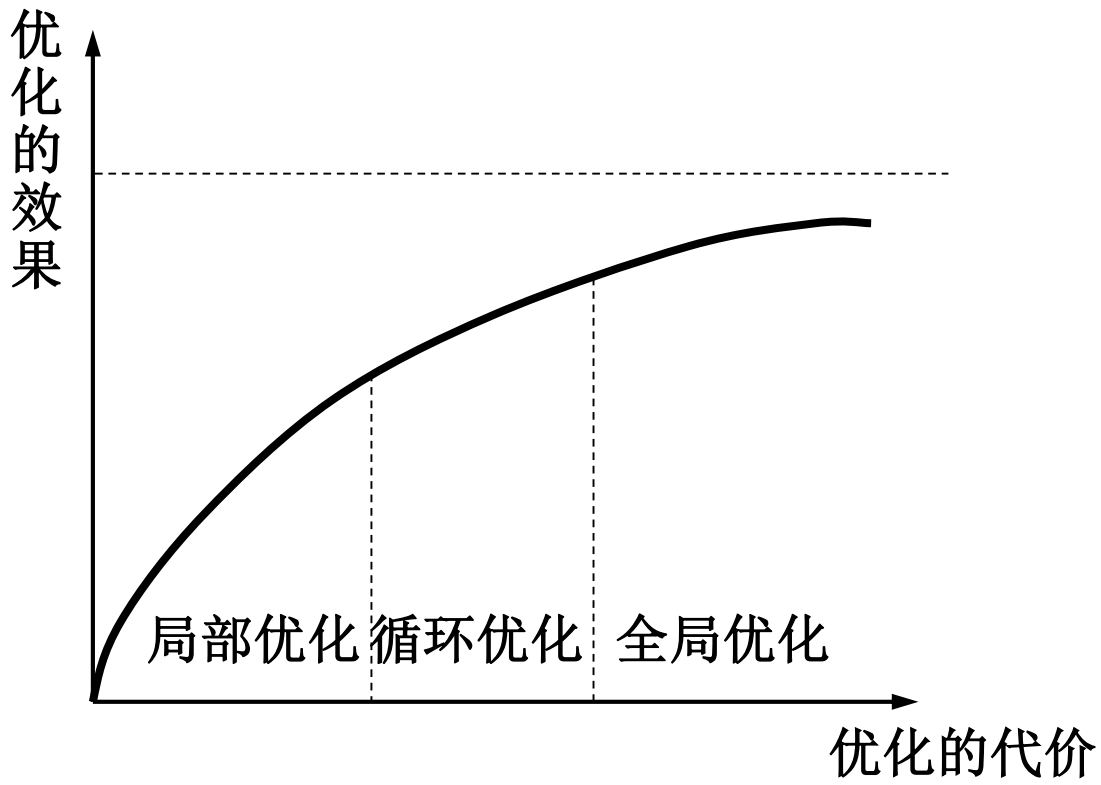
代码优化（code optimization)

指编译程序为了生成高质量的目标程序而做的各种加工和处理。

目的：提高目标代码运行效率
- 时间效率（减少运行时间）
- 空间效率（减少内存容量）

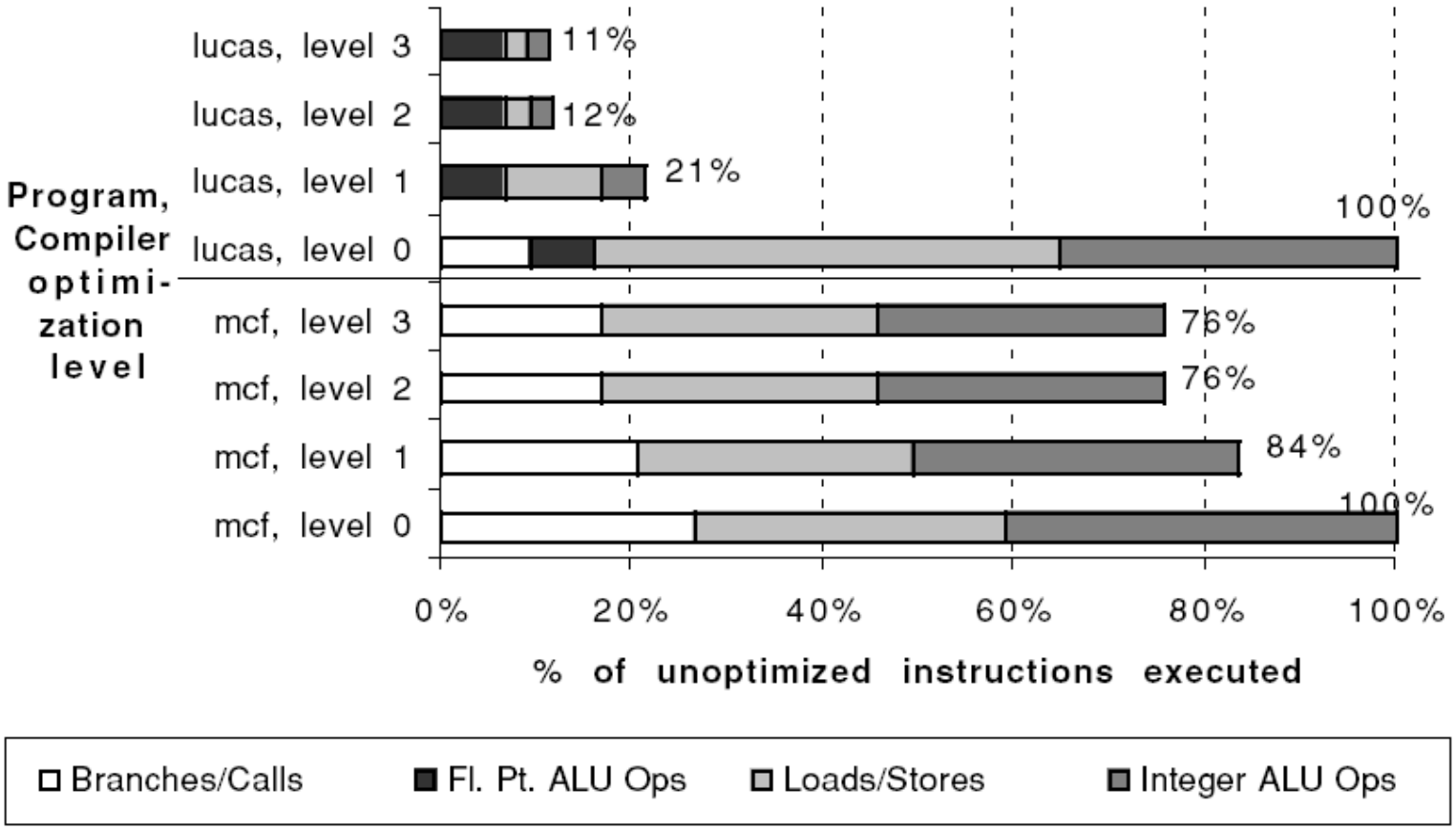
原则：进行优化必须严格遵循“不能改变原有程序语义”原则。

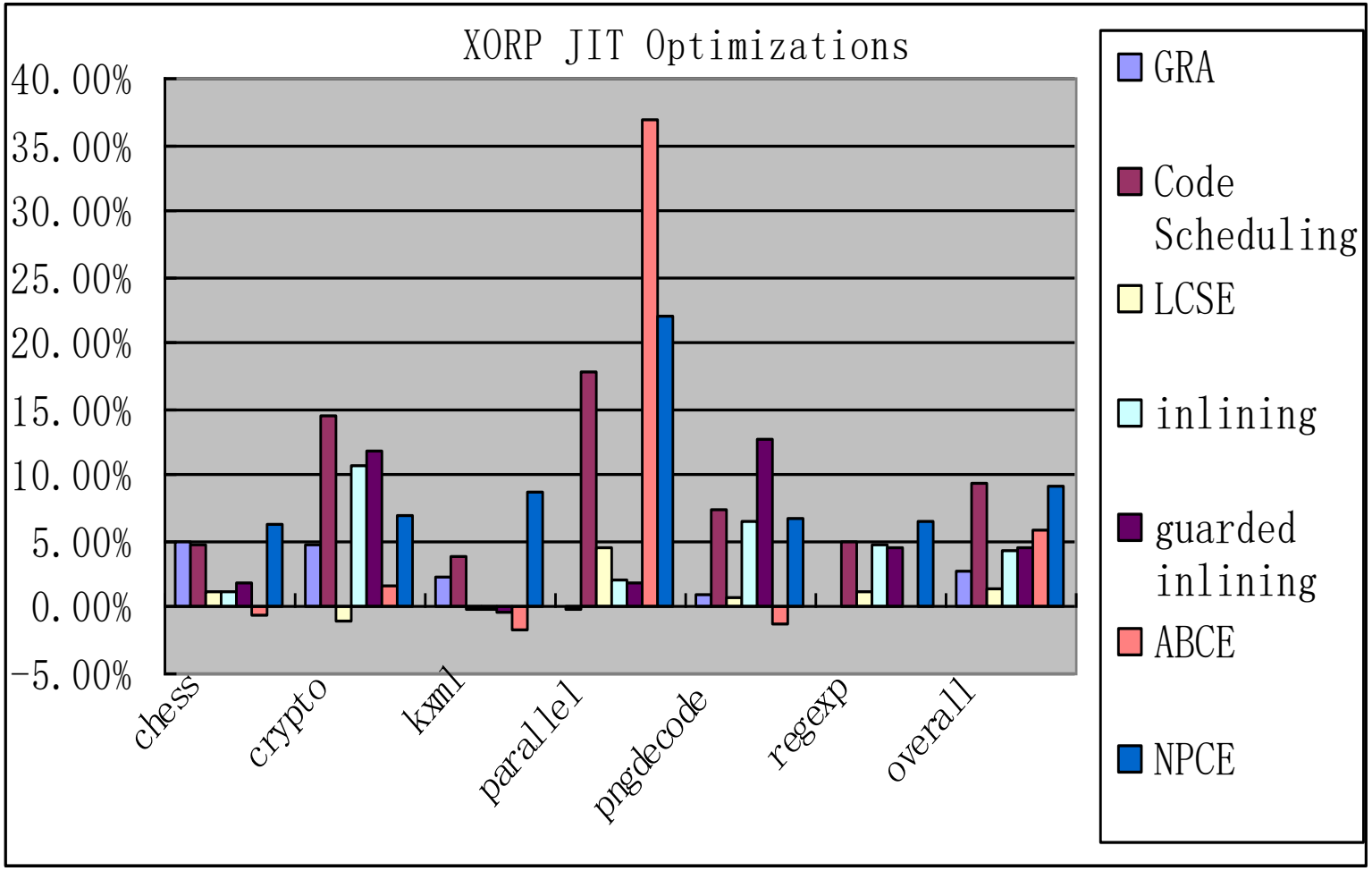
优化所花费的代价和优化产生的效果可用下图表示：



示意图：表示只要做些简单的处理，便能得到明显的优化效果。若要进一步提高优化效果，就要逐步付出更大的代价。

针对 SPEC2000中*lucas*和*mcf* 施加不同级别的编译优化后的运行结果（编译器Alpha Compiler）





全局寄存器分配

指令调度

局部子表达式删除

代码内联

有保护的代码内联

消除数组边界检查

消除空指针检查

为什么要优化?

- 有的大型计算程序一运行就要花上几十分钟，甚至几小时、几天、几十天，这时为优化即使付出些代价是值得的。
- 程序中的循环往往要占用大量的计算时间。所以为减少循环执行时间所进行的优化对减少整个程序的运行时间有很大的意义。 ——尤其有实时要求的程序。
- 对于像学生作业之类的简单小程序（占机器内存，运行速度均可接受），或在程序的调试阶段，花费许多代价去进行一遍又一遍的优化就毫无必要了（尤其机器速度、存储的快速发展）。

优化不能改变源代码的语义!

多线程编程中，多个线程同时存取共享变量x（初始值0）时，需要使用“忙等待”控制多个线程顺序存取之。这里的空语句就不可被优化掉!

```
y = Compute(my_rank);
while (flag!= my_rank);
x = x + y;
flag++;
```

优化方法的分类（按层次分）：

- 与机器无关的优化技术：即与目标机无关的优化，通常是在中间代码上进行的优化。
  - 如：数据流分析，常量传播，公共子表达式删除，死代码删除，循环交换，代码内联等等。
- 与机器相关的优化技术：充分利用系统资源（指令系统，寄存器资源）。
  - 面向超标量超流水线架构、VLIW或者EPIC架构的指令调度方法；面向SMP架构的同步负载优化方法；面向SIMD、MIMD或者SPMD架构的数据级并行优化方法等 。
  - 特点：仅在特定体系结构下有效。

优化方法的分类（按范围分）：

- **局部优化**

  指在基本块内进行的优化。例如，局部公共子表达式删除

- **循环优化**

  对循环语句所生成的中间代码序列上所进行的优化。

- **全局优化**

  跨越基本块，在函数/过程内进行的优化，需要进行全局控制流和数据流分析。例如全局数据流分析。

- **跨函数优化**

  整个程序。例如跨函数别名分析，逃逸分析等

基本块定义

- 基本块中的代码是连续的语句序列。
- 程序的执行（控制流）只能从基本块的第一条语句进入。
- 程序的执行只能从基本块的最后一条语句离开。

14.1 基本块和流图

```
void foo(int* a, int* b)
{
    int prod = 0 ;
    int i ;

    for(i = 1 ; i<=20; i++){
        prod = prod + a[i] * b[i] ;
    }
    …
}
```

```
(1)   prod := 0
(2)   i := 1
(3)   t1 := 4 * i
(4)   t2 := a [ t1 ]
(5)   t3 := 4 * i
(6)   t4 := b [ t3 ]
(7)   t5 := t2 * t4
(8)   t6 := prod + t5
(9)   prod := t6
(10)  t7 := i + 1
(11)  i := t7
(12)  if  i <= 20 goto (3)
(13)  …
```

函数foo的C语言源码和三地址中间代码

分析

```
(1)   prod := 0
(2)   i := 1
(3)   t1 := 4 * i
(4)   t2 := a [ t1 ]
(5)   t3 := 4 * i
(6)   t4 := b [ t3 ]
(7)   t5 := t2 * t4
(8)   t6 := prod + t5
(9)   prod := t6
(10)  t7 := i + 1
(11)  i := t7
(12)  if  i <= 20 goto (3)
(13)  …
```

不分块，不利于优化，因为编译器得不到分支、路径等控制信息，得不到数据流、控制流的信息。

算法14.1 划分基本块

输入：四元式序列

输出：基本块列表。每个四元式仅出现在一个基本块中

方法：

1、首先确定入口语句（每个基本块的第一条语句）的集合。

- 规则1：整个语句序列的第一条语句属于入口语句；
- 规则2：任何能由条件/无条件跳转语句转移到的第一条语句属于入口语句；
- 规则3：紧跟在跳转语句之后的第一条语句属于入口语句。

2、每个入口语句直到下一个入口语句，或者程序结束，它们之间的所有语句都属于同一个基本块。

```
(1)   prod := 0
(2)   i := 1
(3)   t1 := 4 * i
(4)   t2 := a [ t1 ]
(5)   t3 := 4 * i
(6)   t4 := b [ t3 ]
(7)   t5 := t2 * t4
(8)   t6 := prod + t5
(9)   prod := t6
(10)  t7 := i + 1
(11)  i := t7
(12)  if  i <= 20 goto (3)
(13)  …
```

1、首先确定入口语句

（每个基本块的第一条语句）的集合

1.1 整个语句序列的第一条语句属于入口语句

1.2 任何能由条件/无条件跳转语句转移到的第一条语句属于入口语句

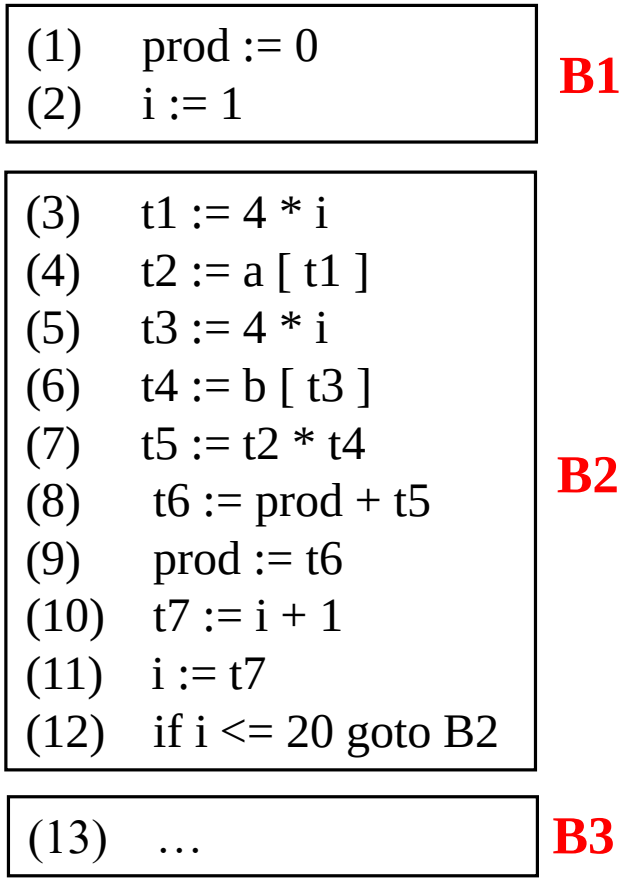
1.3 紧跟在跳转语句之后的第一条语句属于入口语句

2、每个入口语句直到下一个入口语句，或者程序结束，之间的所有语句都属于同一个基本块

基本块

- 基本块中的代码是连续的语句序列
- 程序的执行（控制流）只能从基本块的第一条语句进入
- 程序的执行只能从基本块的最后一条语句离开

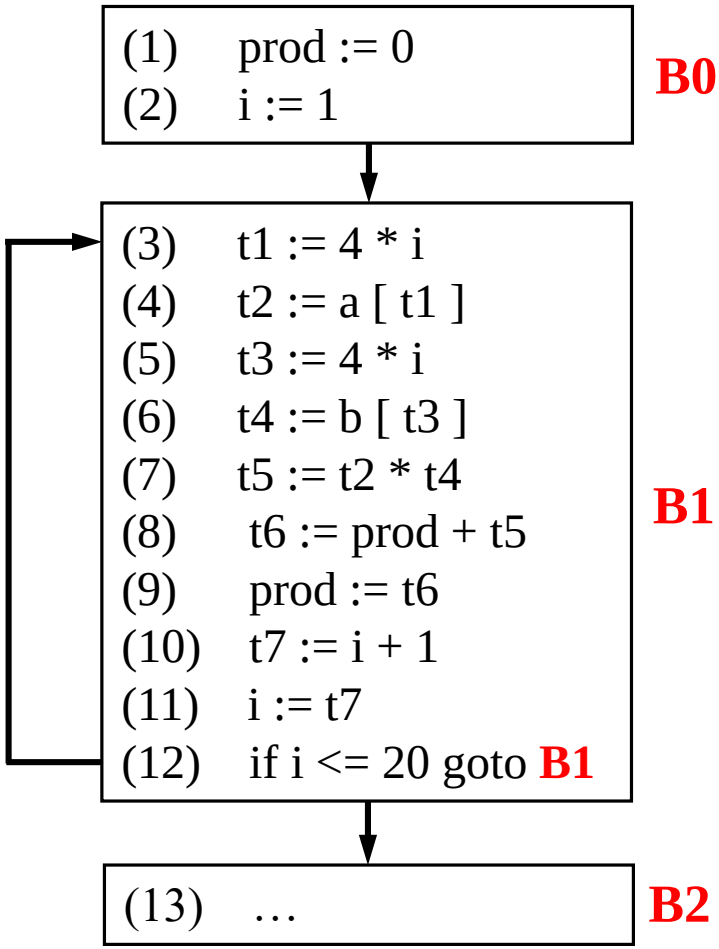
```
(1)   prod := 0
(2)   i := 1
(3)   t1 := 4 * i
(4)   t2 := a [ t1 ]
(5)   t3 := 4 * i
(6)   t4 := b [ t3 ]
(7)   t5 := t2 * t4
(8)   t6 := prod + t5
(9)   prod := t6
(10)  t7 := i + 1
(11)  i := t7
(12)  if  i <= 20 goto (3)
(13)  …
```



流图

- 流图是一种有向图
- 流图的结点是基本块
- 如果在某个执行序列中，B2的执行紧跟在B1之后，则从B1到B2有一条有向边
- 我们称B1为B2的前驱，B2为B1的后继
  - 从B1的最后一条语句有条件或者无条件转移到B2的第一条语句；或者:
  - 按照程序的执行次序，B2紧跟在B1之后，并且B1没有无条件转移到其他基本块。

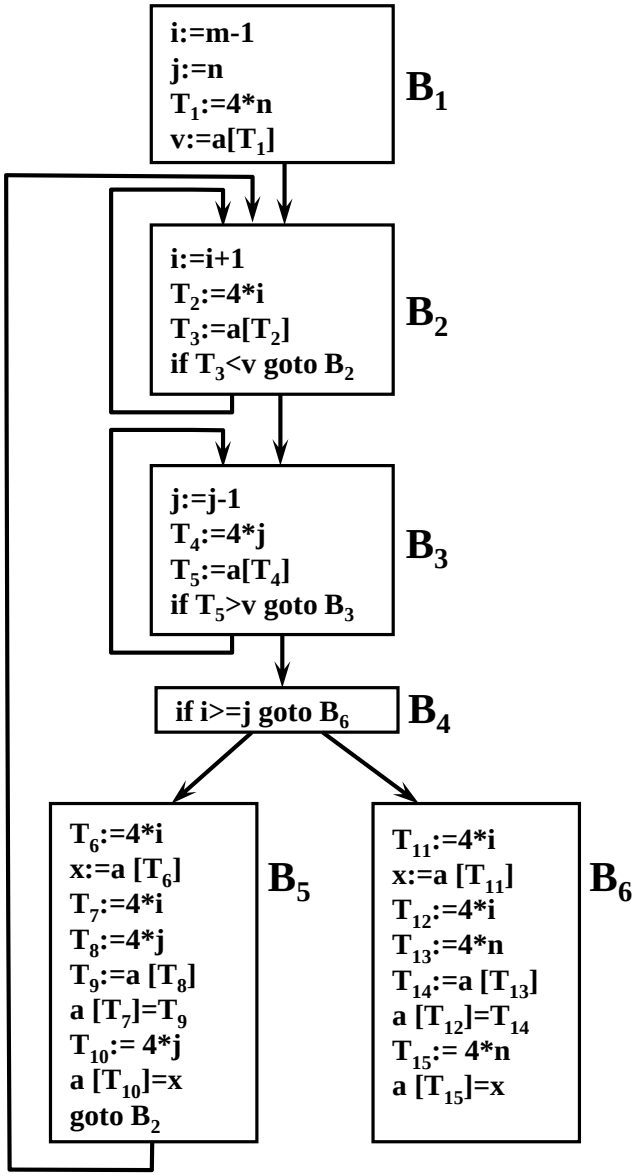
```
(1)   prod := 0
(2)   i := 1
(3)   t1 := 4 * i
(4)   t2 := a [ t1 ]
(5)   t3 := 4 * i
(6)   t4 := b [ t3 ]
(7)   t5 := t2 * t4
(8)   t6 := prod + t5
(9)   prod := t6
(10)  t7 := i + 1
(11)  i := t7
(12)  if  i <= 20 goto (3)
(13)  …
```



例:

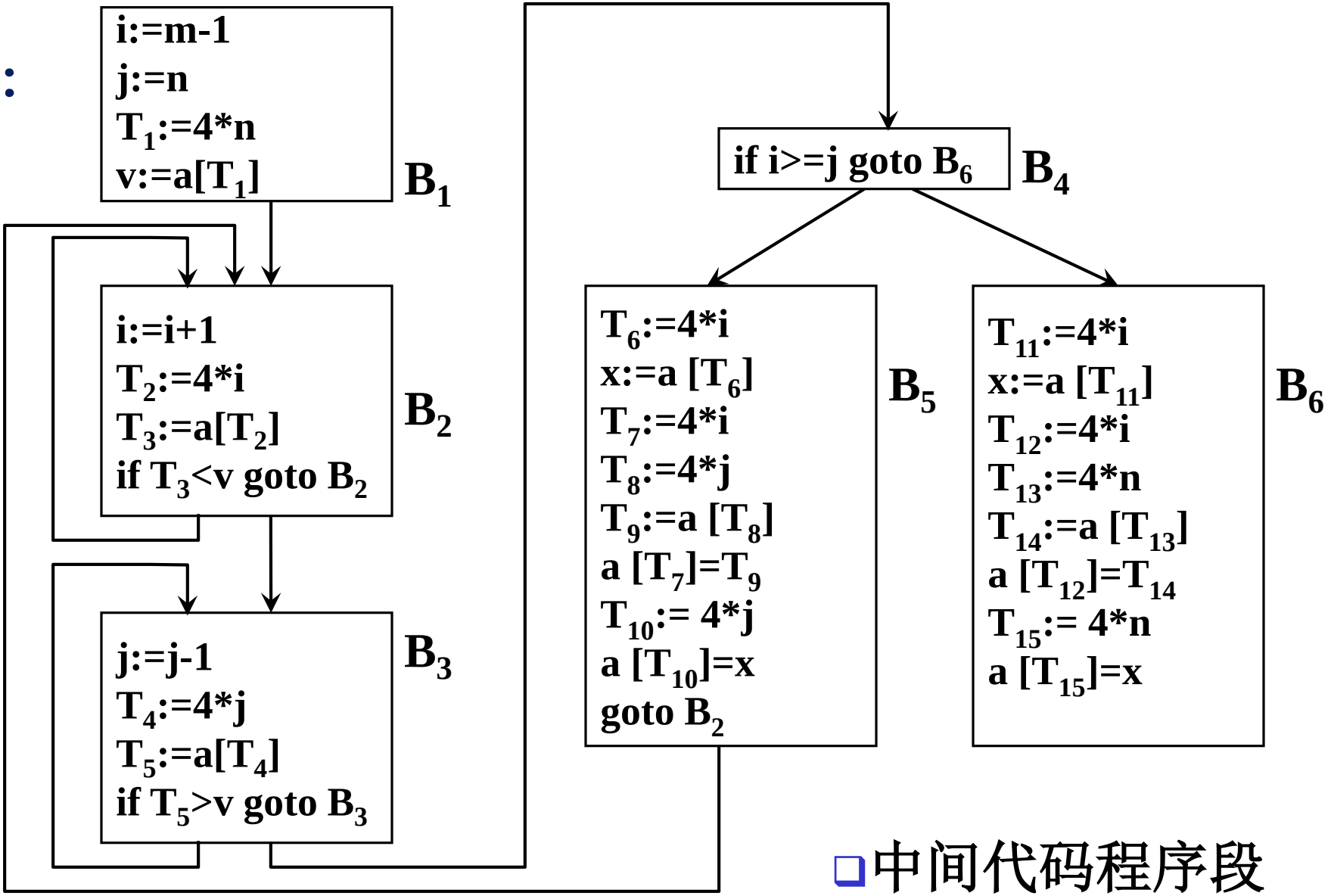
```
void quicksort (m, n);
int m, n;
    {
        int i,  j;
        int v,  x;
        if (n <= m) return;
        /* fragment begins here*/
        i = m - 1;  j = n;  v = a[n];
        while (1) {
            do  i = i + 1;  while (a[i] < v);
            do  j = j - 1;  while (a[j] > v);
            if ( i >= j )  break;
            x = a[i];  a[i] = a[j];  a[j] = x;
        }
        x = a[i];  a[i] = a[n];  a[n] = x;
        /*fragment ends here*/
        quicksort (m,  j); quicksort ( i + 1,  n);
}
```

例:


i:=m-1
j:=n
T1:=4*n
v:=a[T1]
B1
i:=i+1
T2:=4*i
T3:=a[T2]
if T3<v goto B2
B2
j:=j-1
T4:=4*j
T5:=a[T4]
if T5>v goto B3
B3
if i>=j goto B6
B4
T6:=4*i
x:=a [T6]
T7:=4*i
T8:=4*j
T9:=a [T8]
a [T7]=T9
T10:= 4*j
a [T10]=x
goto B2
B5
T11:=4*i
x:=a [T11]
T12:=4*i
T13:=4*n
T14:=a [T13]
a [T12]=T14
T15:= 4*n
a [T15]=x
B6

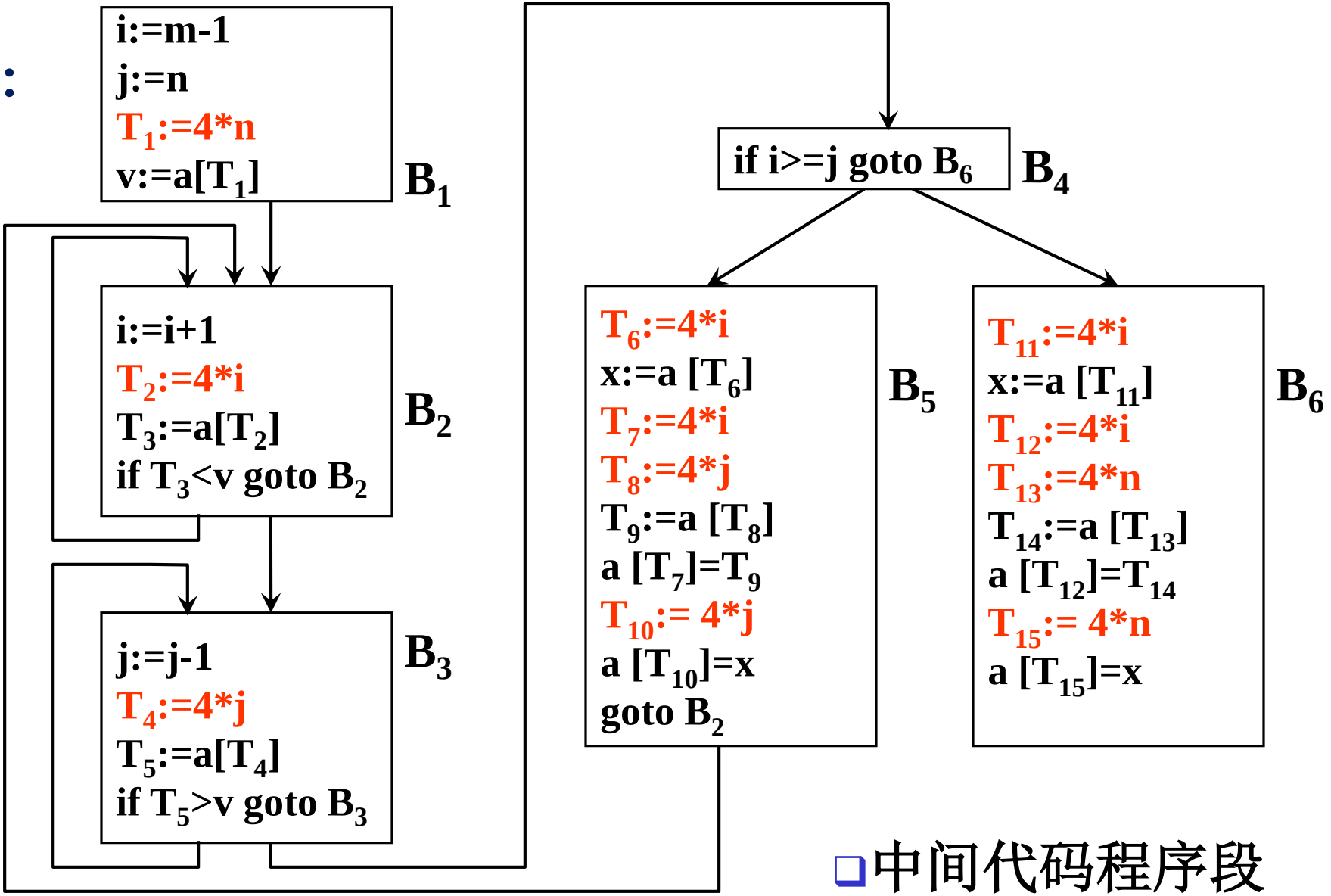
- 基本块及流图

例:


i:=m-1
j:=n
T1:=4*n
v:=a[T1]
B1
i:=i+1
T2:=4*i
T3:=a[T2]
if T3<v goto B2
B2
j:=j-1
T4:=4*j
T5:=a[T4]
if T5>v goto B3
B3
if i>=j goto B6
B4
T6:=4*i
x:=a [T6]
T7:=4*i
T8:=4*j
T9:=a [T8]
a [T7]=T9
T10:= 4*j
a [T10]=x
goto B2
B5
T11:=4*i
x:=a [T11]
T12:=4*i
T13:=4*n
T14:=a [T13]
a [T12]=T14
T15:= 4*n
a [T15]=x
B6

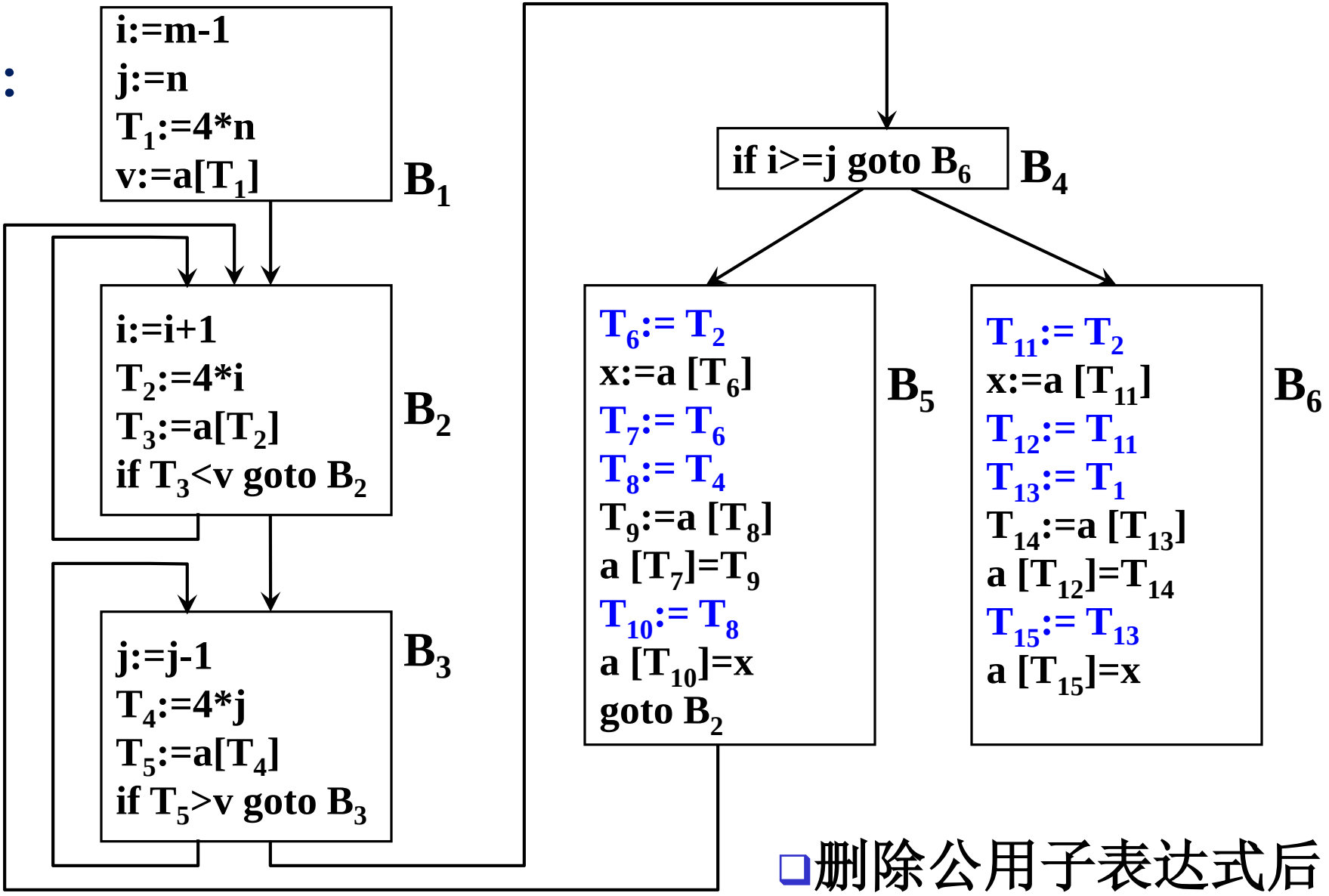
- 中间代码程序段

例:


i:=m-1
j:=n
T1:=4*n
v:=a[T1]
B1
i:=i+1
T2:=4*i
T3:=a[T2]
if T3<v goto B2
B2
j:=j-1
T4:=4*j
T5:=a[T4]
if T5>v goto B3
B3
if i>=j goto B6
B4
T6:=4*i
x:=a [T6]
T7:=4*i
T8:=4*j
T9:=a [T8]
a [T7]=T9
T10:= 4*j
a [T10]=x
goto B2
B5
T11:=4*i
x:=a [T11]
T12:=4*i
T13:=4*n
T14:=a [T13]
a [T12]=T14
T15:= 4*n
a [T15]=x
B6

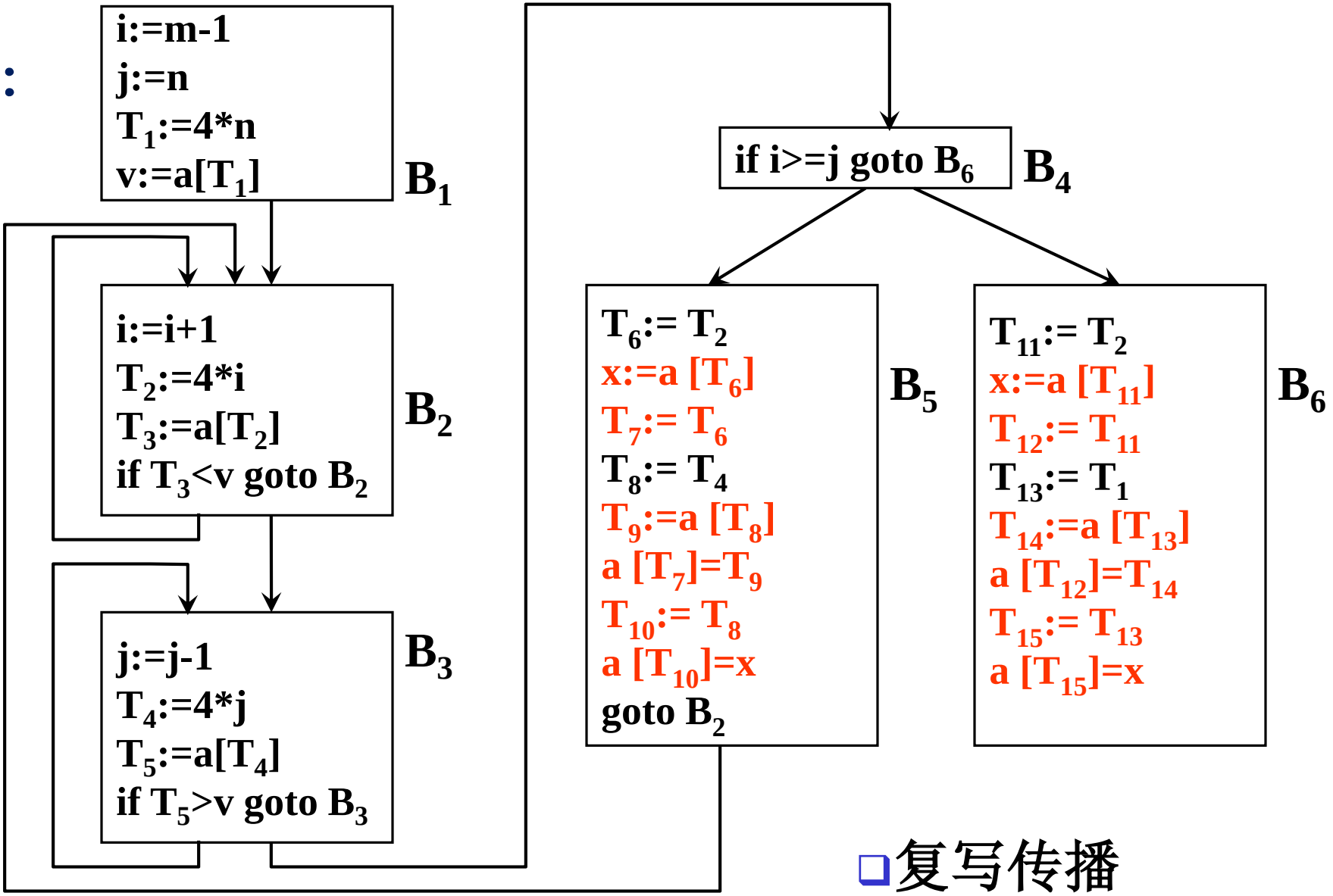
- 中间代码程序段

例:


i:=m-1
j:=n
T1:=4*n
v:=a[T1]
B1
i:=i+1
T2:=4*i
T3:=a[T2]
if T3<v goto B2
B2
j:=j-1
T4:=4*j
T5:=a[T4]
if T5>v goto B3
B3
if i>=j goto B6
B4
T6:= T2
x:=a [T6]
T7:= T6
T8:= T4
T9:=a [T8]
a [T7]=T9
T10:= T8
a [T10]=x
goto B2
B5
T11:= T2
x:=a [T11]
T12:= T11
T13:= T1
T14:=a [T13]
a [T12]=T14
T15:= T13
a [T15]=x
B6

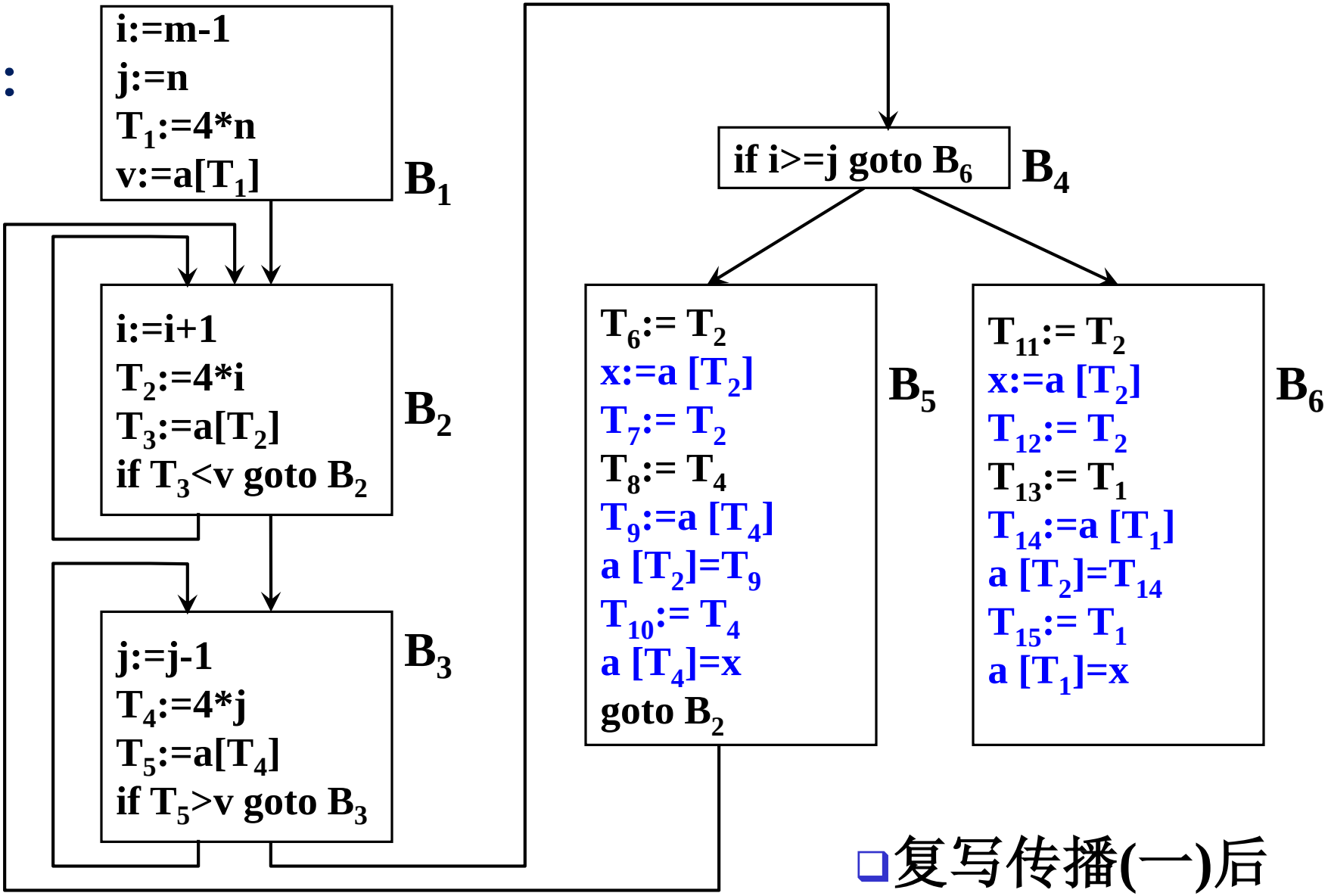
- 删除公用子表达式后

例:


i:=m-1
j:=n
T1:=4*n
v:=a[T1]
B1
i:=i+1
T2:=4*i
T3:=a[T2]
if T3<v goto B2
B2
j:=j-1
T4:=4*j
T5:=a[T4]
if T5>v goto B3
B3
if i>=j goto B6
B4
T6:= T2
x:=a [T6]
T7:= T6
T8:= T4
T9:=a [T8]
a [T7]=T9
T10:= T8
a [T10]=x
goto B2
B5
T11:= T2
x:=a [T11]
T12:= T11
T13:= T1
T14:=a [T13]
a [T12]=T14
T15:= T13
a [T15]=x
B6

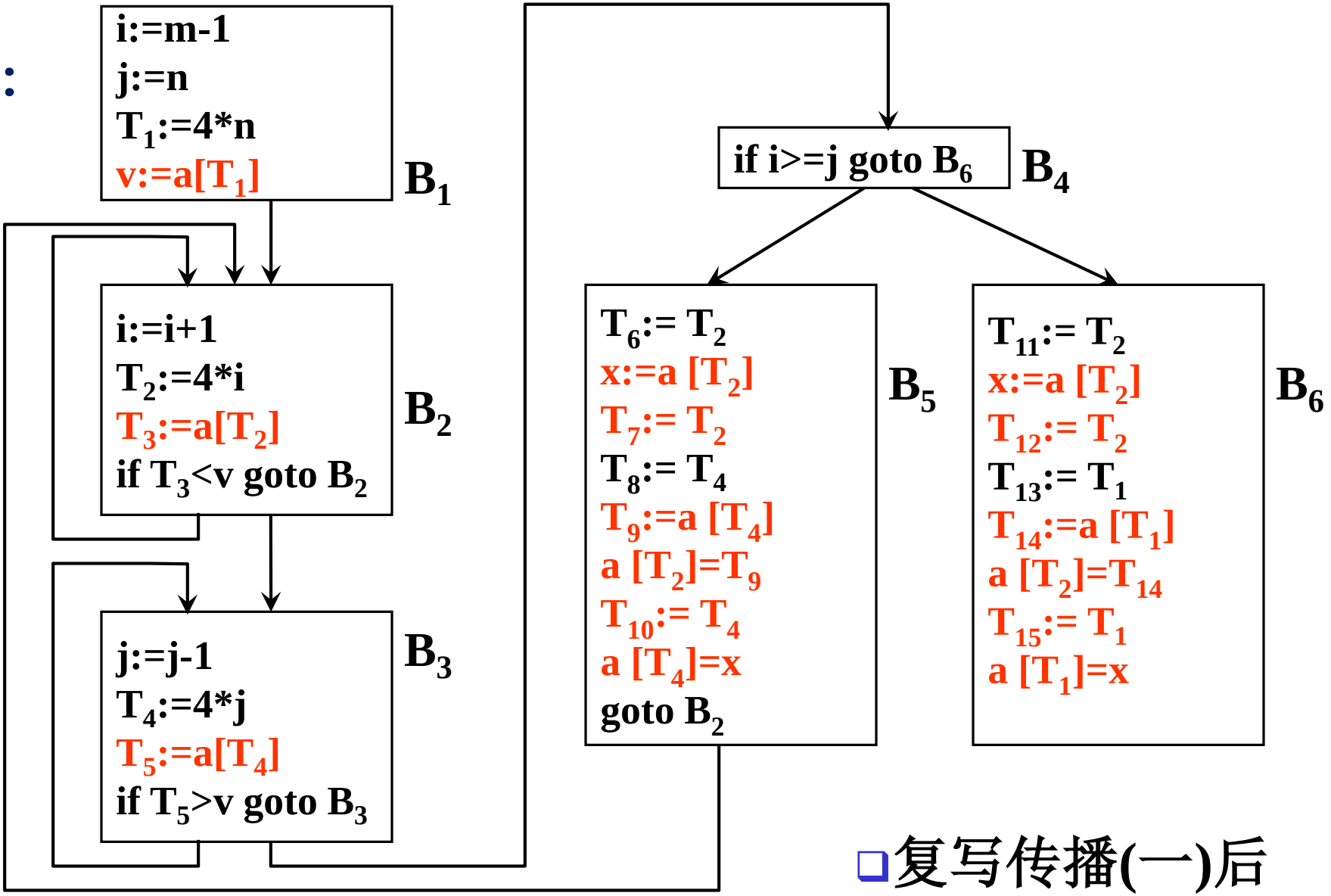
- 复写传播

例:


i:=m-1
j:=n
T1:=4*n
v:=a[T1]
B1
i:=i+1
T2:=4*i
T3:=a[T2]
if T3<v goto B2
B2
j:=j-1
T4:=4*j
T5:=a[T4]
if T5>v goto B3
B3
if i>=j goto B6
B4
T6:= T2
x:=a [T2]
T7:= T2
T8:= T4
T9:=a [T4]
a [T2]=T9
T10:= T4
a [T4]=x
goto B2
B5
T11:= T2
x:=a [T2]
T12:= T2
T13:= T1
T14:=a [T1]
a [T2]=T14
T15:= T1
a [T1]=x
B6

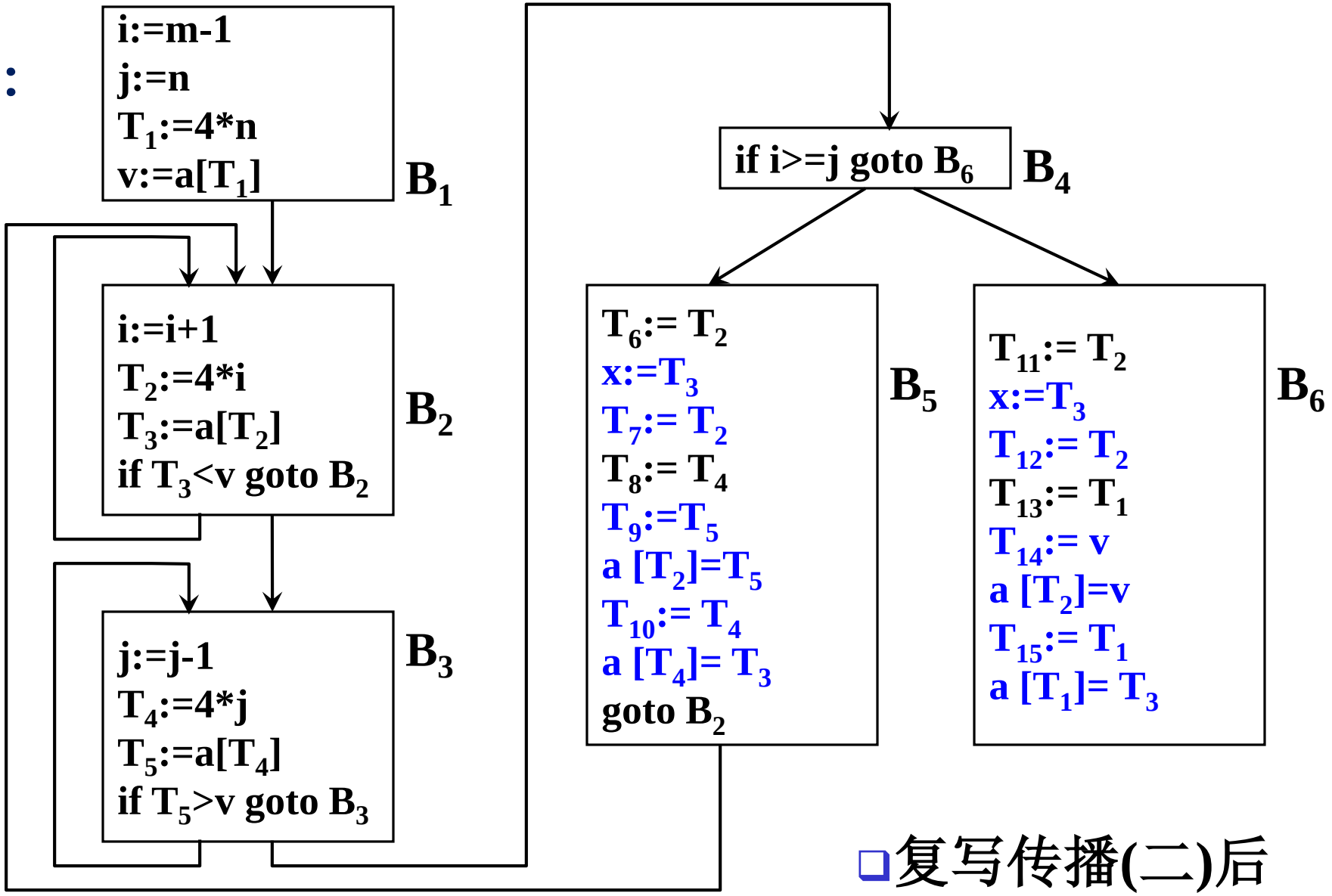
- 复写传播(一)后

例:


i:=m-1
j:=n
T1:=4*n
v:=a[T1]
B1
i:=i+1
T2:=4*i
T3:=a[T2]
if T3<v goto B2
B2
j:=j-1
T4:=4*j
T5:=a[T4]
if T5>v goto B3
B3
if i>=j goto B6
B4
T6:= T2
x:=a [T2]
T7:= T2
T8:= T4
T9:=a [T4]
a [T2]=T9
T10:= T4
a [T4]=x
goto B2
B5
T11:= T2
x:=a [T2]
T12:= T2
T13:= T1
T14:=a [T1]
a [T2]=T14
T15:= T1
a [T1]=x
B6

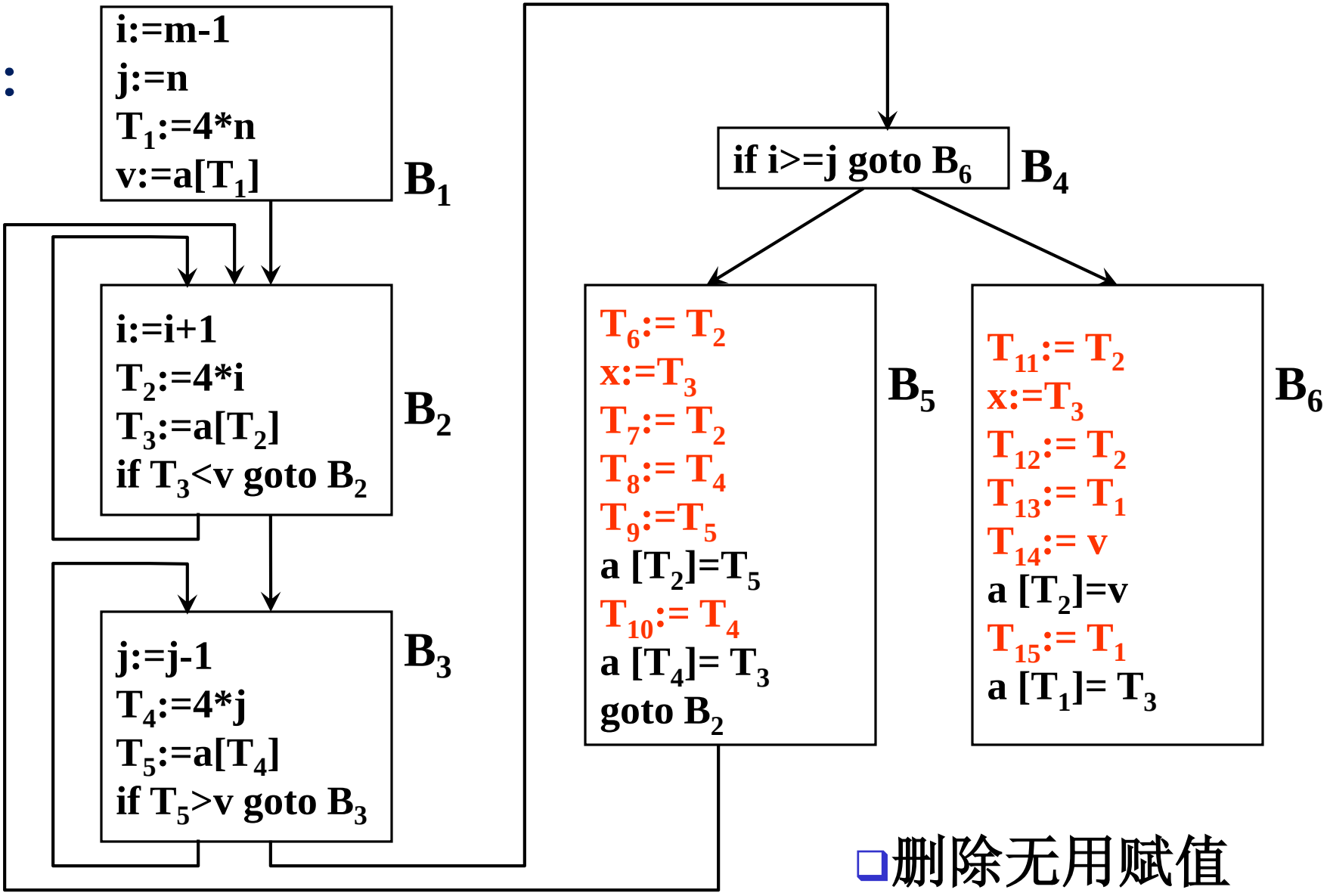
- 复写传播(一)后

例:


i:=m-1
j:=n
T1:=4*n
v:=a[T1]
B1
i:=i+1
T2:=4*i
T3:=a[T2]
if T3<v goto B2
B2
j:=j-1
T4:=4*j
T5:=a[T4]
if T5>v goto B3
B3
if i>=j goto B6
B4
T6:= T2
x:=T3
T7:= T2
T8:= T4
T9:=T5
a [T2]=T5
T10:= T4
a [T4]= T3
goto B2
B5
T11:= T2
x:=T3
T12:= T2
T13:= T1
T14:= v
a [T2]=v
T15:= T1
a [T1]= T3
B6

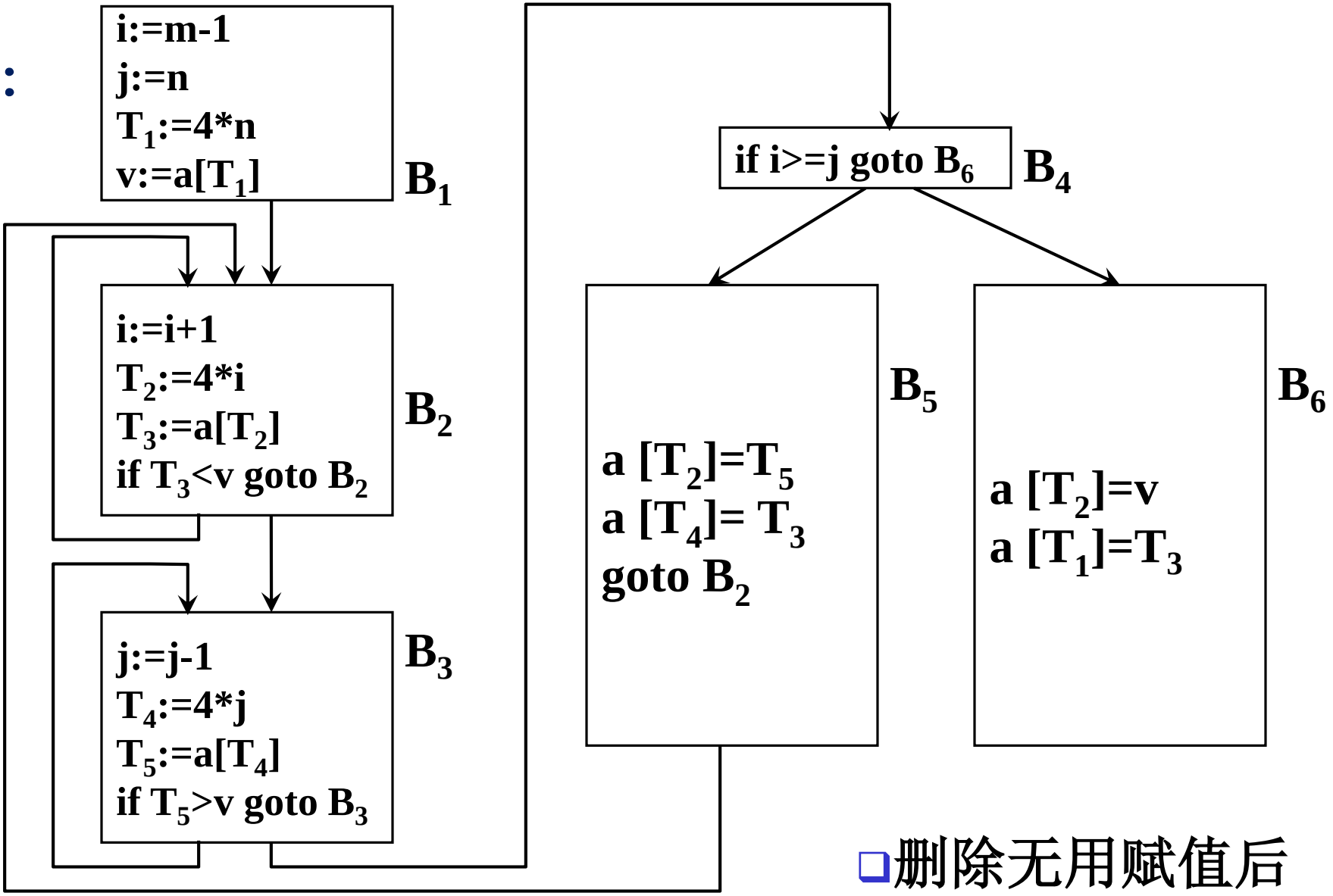
- 复写传播(二)后

例:


i:=m-1
j:=n
T1:=4*n
v:=a[T1]
B1
i:=i+1
T2:=4*i
T3:=a[T2]
if T3<v goto B2
B2
j:=j-1
T4:=4*j
T5:=a[T4]
if T5>v goto B3
B3
if i>=j goto B6
B4
T6:= T2
x:=T3
T7:= T2
T8:= T4
T9:=T5
a [T2]=T5
T10:= T4
a [T4]= T3
goto B2
B5
T11:= T2
x:=T3
T12:= T2
T13:= T1
T14:= v
a [T2]=v
T15:= T1
a [T1]= T3
B6

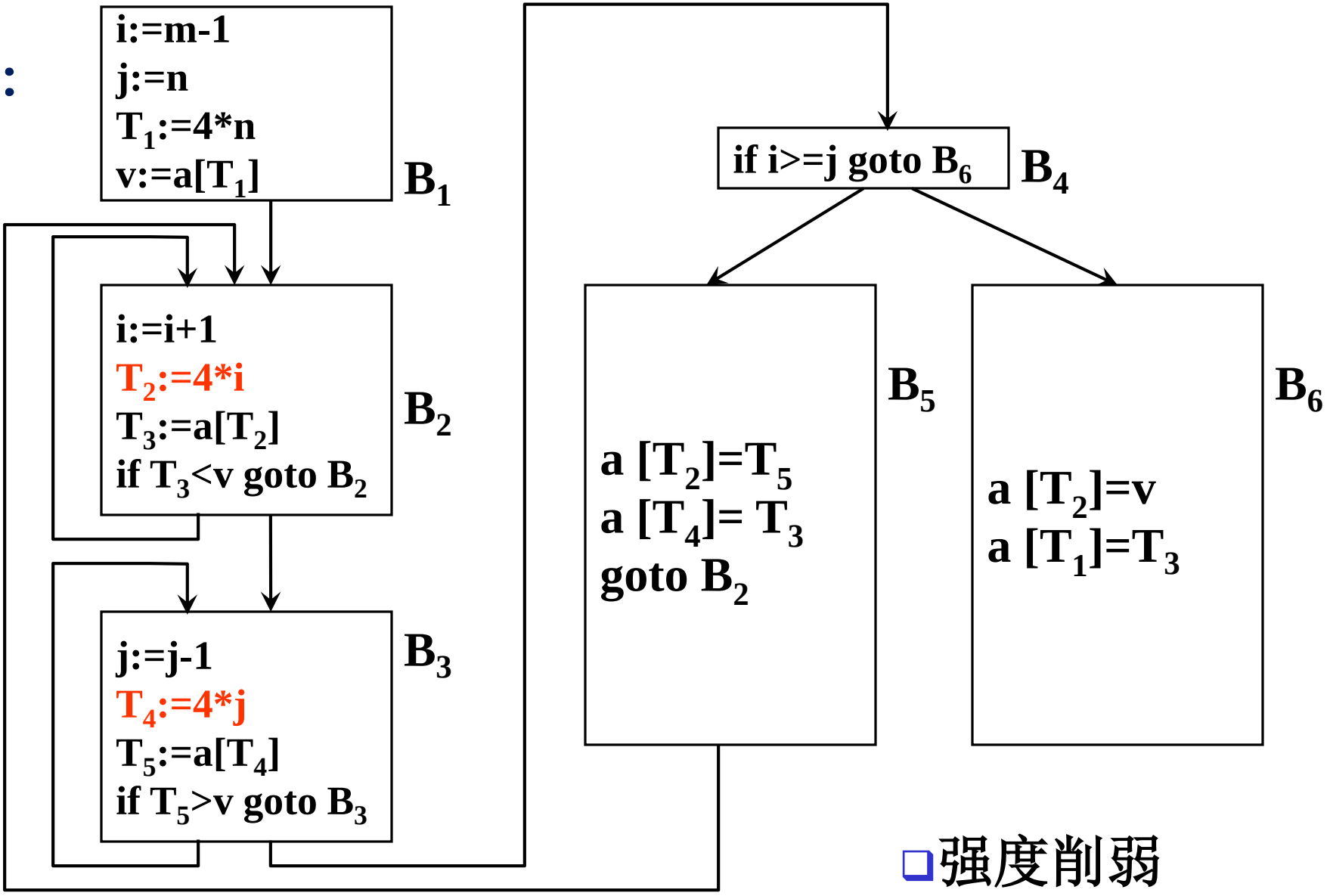
- 删除无用赋值

例:


i:=m-1
j:=n
T1:=4*n
v:=a[T1]
B1
i:=i+1
T2:=4*i
T3:=a[T2]
if T3<v goto B2
B2
j:=j-1
T4:=4*j
T5:=a[T4]
if T5>v goto B3
B3
if i>=j goto B6
B4
a [T2]=T5
a [T4]= T3
goto B2
B5
a [T2]=v
a [T1]=T3
B6

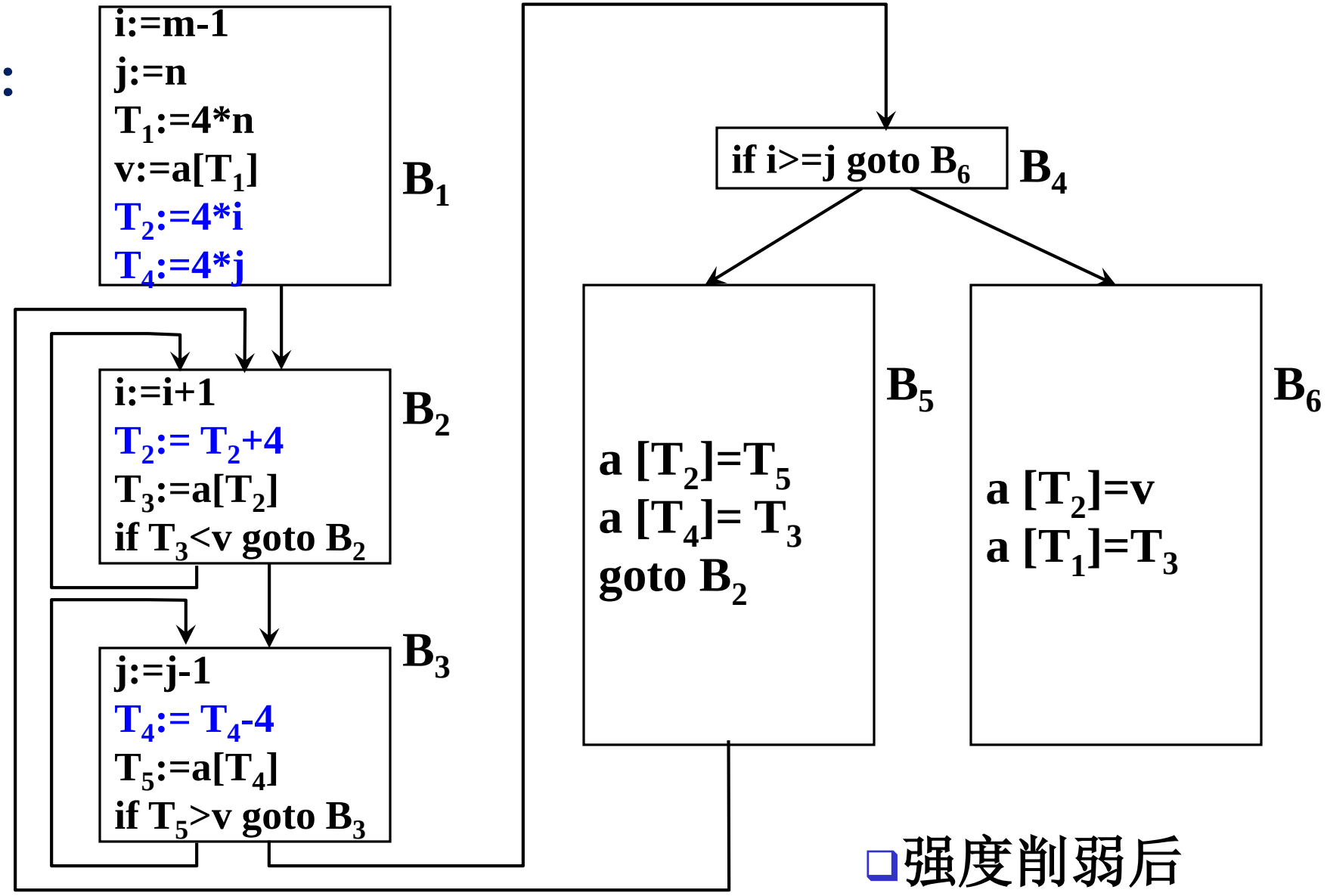
- 删除无用赋值后

例:


i:=m-1
j:=n
T1:=4*n
v:=a[T1]
B1
i:=i+1
T2:=4*i
T3:=a[T2]
if T3<v goto B2
B2
j:=j-1
T4:=4*j
T5:=a[T4]
if T5>v goto B3
B3
if i>=j goto B6
B4
a [T2]=T5
a [T4]= T3
goto B2
B5
a [T2]=v
a [T1]=T3
B6

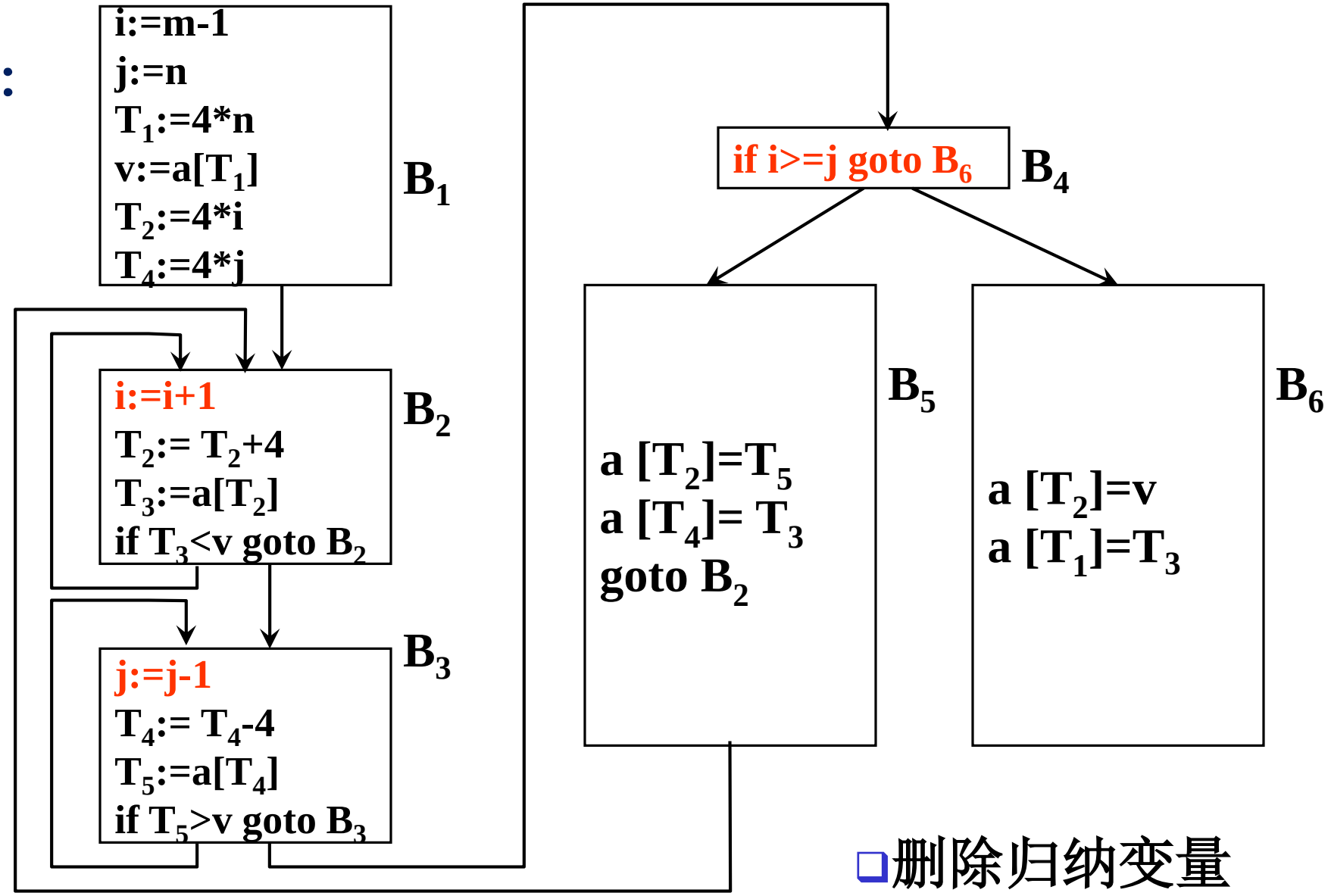
- 强度削弱

例:


i:=m-1
j:=n
T1:=4*n
v:=a[T1]
T2:=4*i
T4:=4*j
B1
i:=i+1
T2:= T2+4
T3:=a[T2]
if T3<v goto B2
B2
j:=j-1
T4:= T4-4
T5:=a[T4]
if T5>v goto B3
B3
if i>=j goto B6
B4
a [T2]=T5
a [T4]= T3
goto B2
B5
a [T2]=v
a [T1]=T3
B6

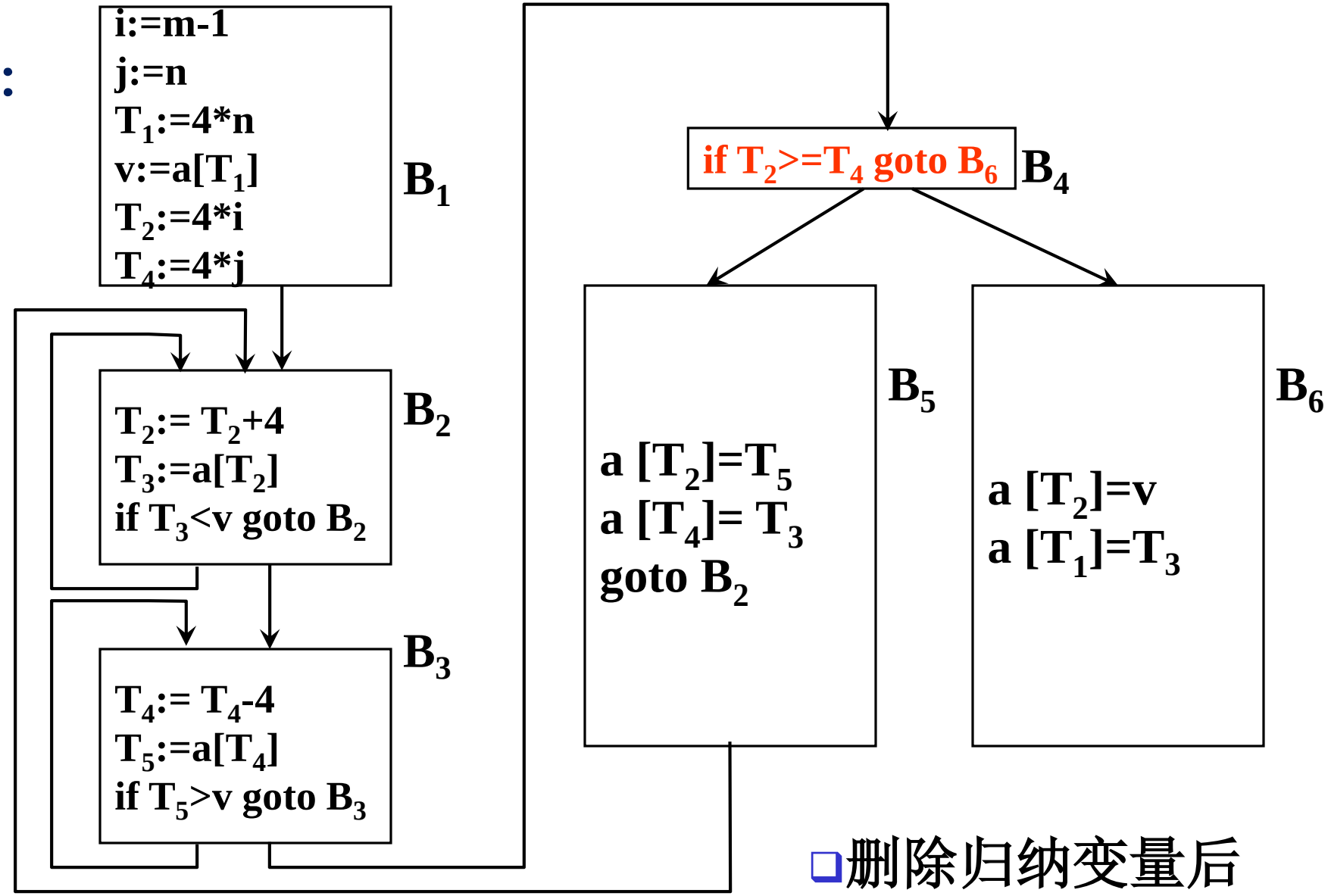
- 强度削弱后

例:


i:=m-1
j:=n
T1:=4*n
v:=a[T1]
T2:=4*i
T4:=4*j
B1
i:=i+1
T2:= T2+4
T3:=a[T2]
if T3<v goto B2
B2
j:=j-1
T4:= T4-4
T5:=a[T4]
if T5>v goto B3
B3
if i>=j goto B6
B4
a [T2]=T5
a [T4]= T3
goto B2
B5
a [T2]=v
a [T1]=T3
B6

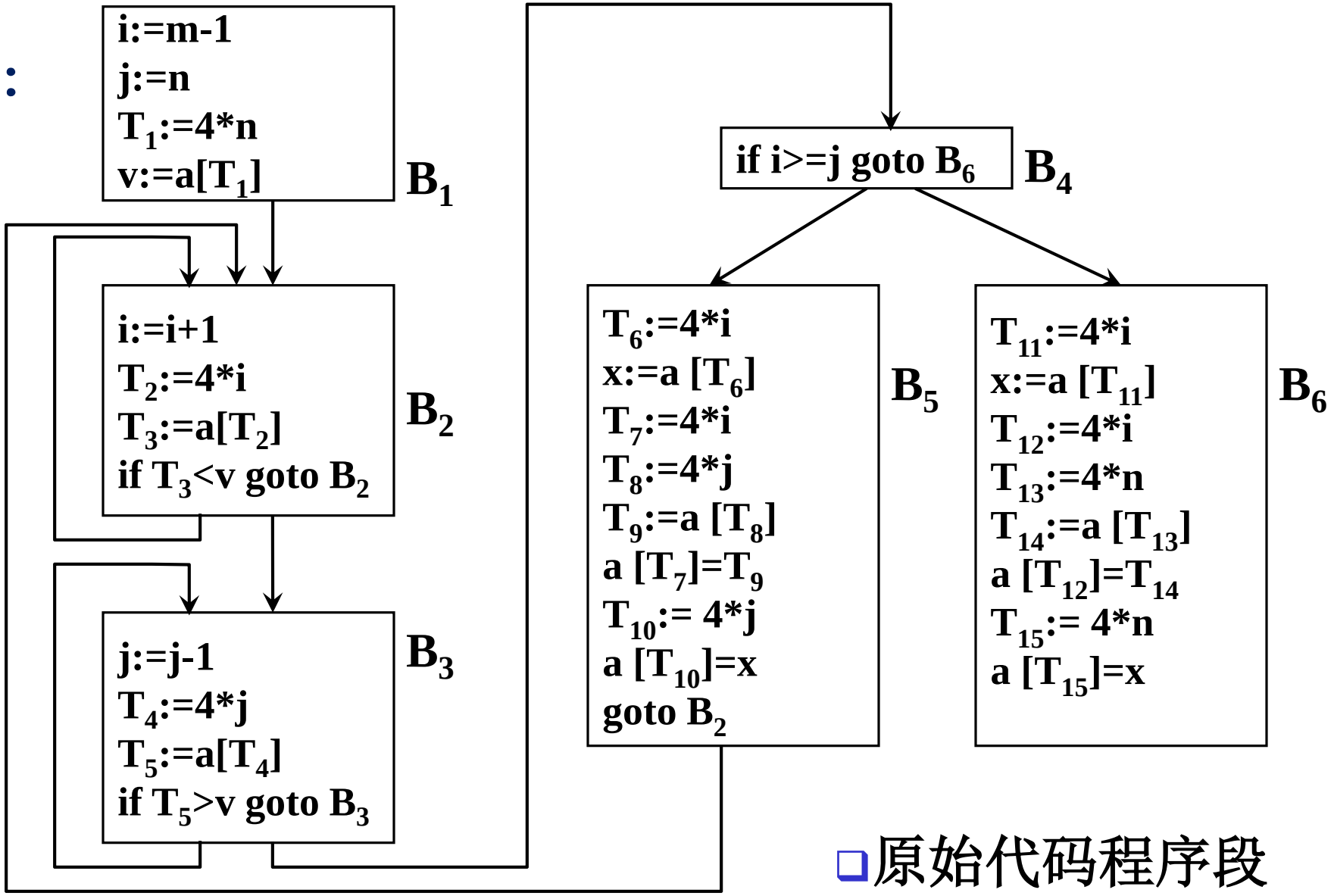
- 删除归纳变量

例:


i:=m-1
j:=n
T1:=4*n
v:=a[T1]
T2:=4*i
T4:=4*j
B1
T2:= T2+4
T3:=a[T2]
if T3<v goto B2
B2
T4:= T4-4
T5:=a[T4]
if T5>v goto B3
B3
if T2>=T4 goto B6
B4
a [T2]=T5
a [T4]= T3
goto B2
B5
a [T2]=v
a [T1]=T3
B6

- 删除归纳变量后

例:


i:=m-1
j:=n
T1:=4*n
v:=a[T1]
B1
i:=i+1
T2:=4*i
T3:=a[T2]
if T3<v goto B2
B2
j:=j-1
T4:=4*j
T5:=a[T4]
if T5>v goto B3
B3
if i>=j goto B6
B4
T6:=4*i
x:=a [T6]
T7:=4*i
T8:=4*j
T9:=a [T8]
a [T7]=T9
T10:= 4*j
a [T10]=x
goto B2
B5
T11:=4*i
x:=a [T11]
T12:=4*i
T13:=4*n
T14:=a [T13]
a [T12]=T14
T15:= 4*n
a [T15]=x
B6

- 原始代码程序段

14.2 基本块内优化

（1）利用代数性质（代数变换）

- 编译时完成常量表达式的计算，整数类型与实型的转换。

  例： a := 5 + 6 + x → a := 11 + x （常数合并）

  PI = 3.141592

  TO_R = PI / 180.0 → TO_R = 0.0174644

  又如： 设x为实型，x := 3 + 1 可变换成x := 4.0

- 下标变量引用时，其地址计算的一部分工作可在编译时预先做好（运行时只需计算“可变部分”即可）。

● 运算强度削弱：用一种需要较少执行时间的运算代替另一种运算，以减少运行时的运算强度(时、空开销)。

如　　x ** 2 → x * x

3 * x → x + x + x

8 * x ，　4 * x　等换成左移运算

x / 2 ，　x / 16　等换成右移运算

x := x + 1　　变为INC x指令

x / 5 → x * 0.2 等

利用机器硬件所提供的一些功能，如左移、右移操作做乘法或除法，具有更高的代码效率。

（2）复写(copy)传播

如 x := y 这样的赋值语句称为复写语句。由于 x 和 y 值相同，所以当满足一定条件时，在该赋值语句下面出现的 x 可用 y 来代替。

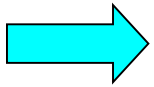
例如:

```
x := y ;
u := 2 * x ;
v := x + 1 ;
```

若以后的语句中不再用到 x 时，则上面的 x := y 可删去。

若上例中不是x := y而是 x := 3。则复写传播变成了常量传播。即

```
x := y;
u := 2 * x;
v := x + 1;
```



```
x := 3;
u := 2 * x;
v := x + 1;
```

u := 6; v := 4;

又如 t_1 := y / z; x := t_1;

若这里 t_1为暂时(中间)变量，以后不再使用，则可变换为

x := y / z;

此外常量传播，引起常量计算，如:

pi = 3.14159 r = pi / 180.0

此时: pi = 3.14159 r = 0.0174532 （常量计算）

（3）删除冗余代码

冗余代码就是毫无实际意义的代码，又称死代码 (dead code)或无用代码(useless code)。

例如: **x := x + 0;** **x := x * 1;** 等

又例:

```
FLAG := TRUE
IF  FLAG  THEN…
   …
ELSE…
```

FLAG永真

另外在程序中为了调试常有如下:

if debug then … 的语句。

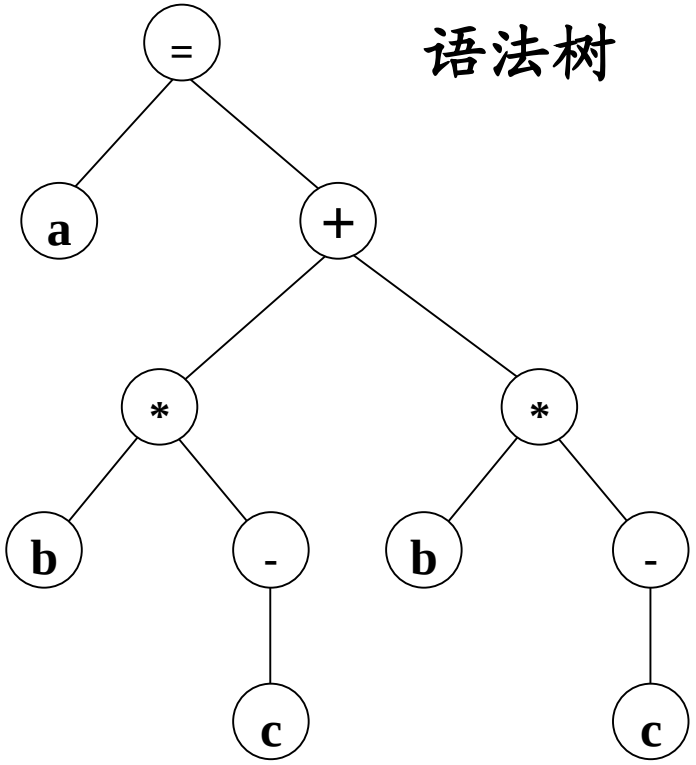
但当debug为false时，then后面的语句便永远不会执行。这就是可删去的冗余代码。

(可用条件编译 #if DEBUG 程序编写，而源代码中还应留着)

基本块内优化: 消除公共子表达式

- 赋值语句: a=b*(-c) + b*(-c)

```
t1 := - c
t2 := b * t1
t3 := - c
t4 := b * t3
t5 := t2 + t4
a := t5
```



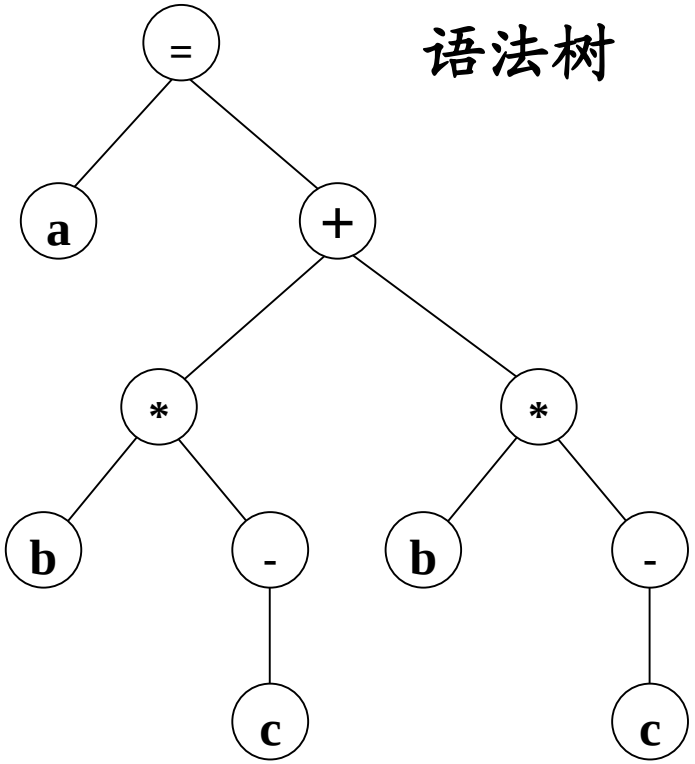
语法树

基本块内优化: 消除公共子表达式

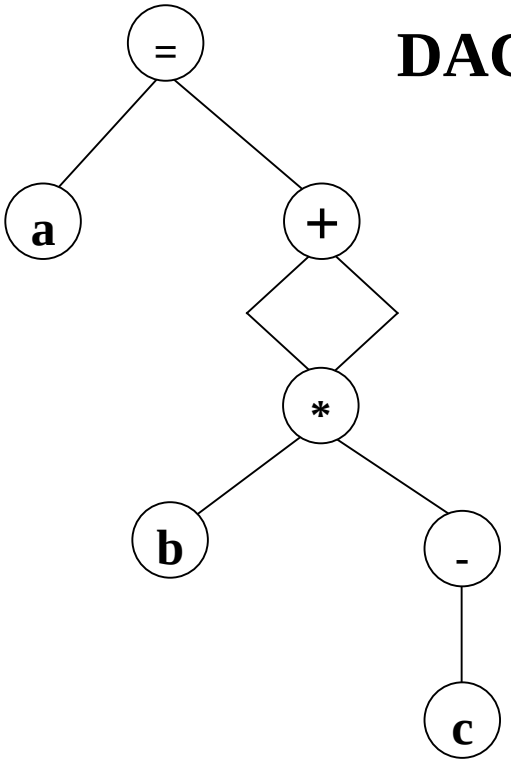
- **DAG图:**
  - **Directed Acyclic Graph 有向无环图**
  - **用来表示基本块内各中间代码之间的关系**
- **可通过DAG图消除公共子表达式**

14.2.1 基本块的DAG图表示

- 赋值语句：a=b*(-c) + b*(-c)



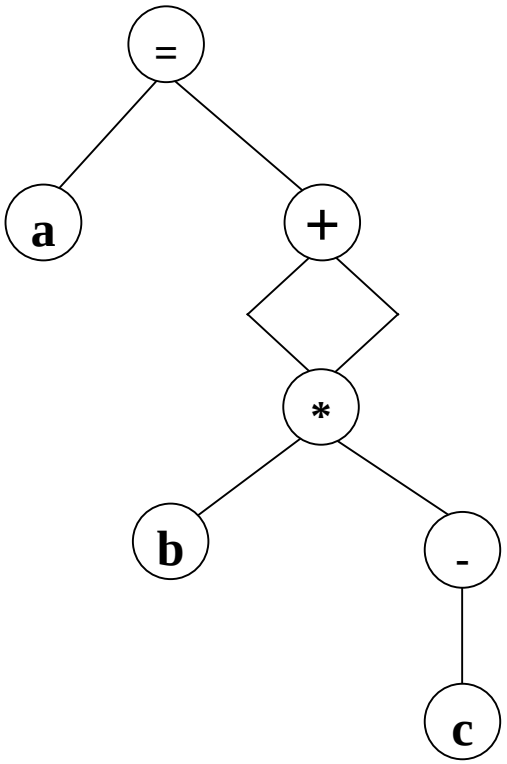
语法树



DAG图

14.2.1 基本块的DAG图表示

DAG图



- 图的叶结点由变量名或常量所标记。（对于那些在基本块内先引用再赋值的变量，可以采用变量名加下标0的方式命名其初值）。
- 图的中间结点由中间代码的操作符所标记，代表着基本块中一条或多条中间代码。
- 基本块中变量的最终计算结果都对应着图中的一个结点；具有初值的变量，其初值和最终值可以分别对应不同的结点。

DAG表示　　a=b*(-c) + b*(-c)

(1)　t1 = - c

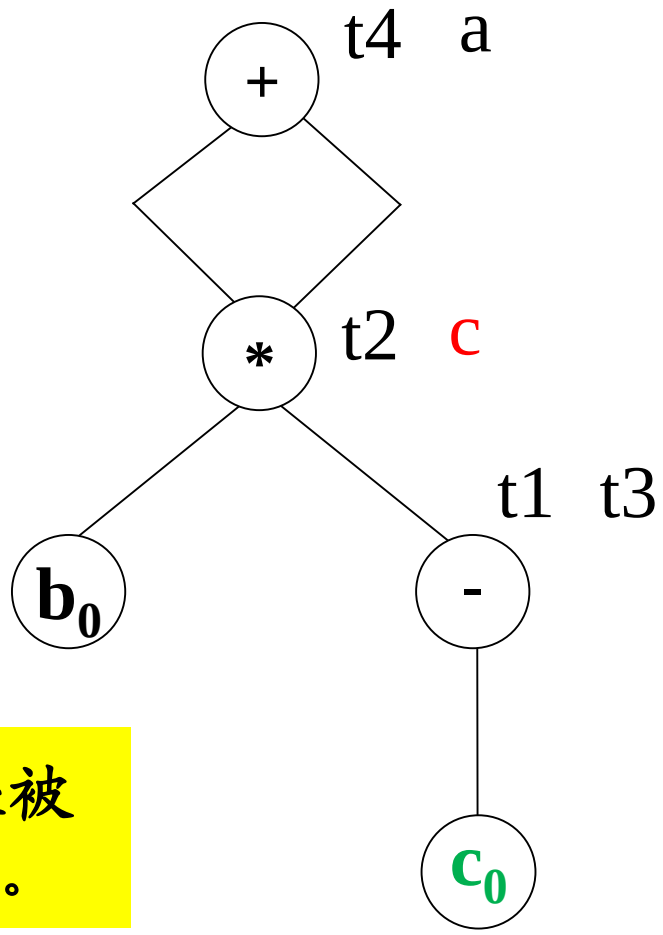
(2)　t2 = b * t1

(3)　t3 = - c

(4)　c = b * t3

(5)　t4 = t2 + c

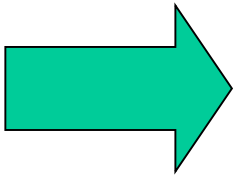
(6)　a = t4



这里，由于变量c被赋值前曾经被引用，因此有c和c_0两种不同的表示。

14.2.2 消除局部公共子表达式

```
t1 := - c
t2 := b * t1
t3 := - c
t4 := b * t3
t5 := t2 + t4
a := t5
```

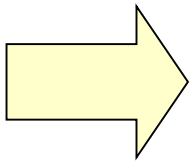


```
t1 := - c
t2 := b * t1
t3 := - c
t4 := b * t3
t5 := t2 + t2 (t4)
a := t5
```

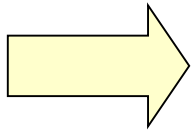
c := c + 1 ?

消除局部公共子表达式

```
t1 := - c
t2 := b * t1
t3 := - c
t4 := b * t3
t5 := t2 + t4
a := t5
```



DAG图



```
t1 := - c
t2 := b * t1
t5 := t2 + t2
a := t5
```

需要两个算法:

1、DAG图的生成算法

2、从DAG图导出代码的算法

算法14.2 构建DAG图的算法—消除公共子表达式

- 输入：基本块内的中间代码序列
- 输出：完成局部公共子表达式删除后的DAG图
- 方法：

1. 首先建立结点表，该表记录了变量名和常量值，以及它们当前所对应的DAG图中结点的序号。该表初始状态为空。
2. 从第一条中间代码开始，按照以下规则建立DAG图。
3. 对于形如z = x op y的中间代码，其中z为记录计算结果的变量名，x为左操作数，y为右操作数，op为操作符：首先在结点表中寻找x，如果找到，记录下x当前所对应的结点号i；如果未找到，在DAG图中新建一个叶结点，假设其结点号仍为i，标记为x（如x为变量名，该标记更改为x_0）；在结点表中增加新的一项(x, i)，表明二者之间的对应关系。右操作数y与x同理，假设其对应结点号为j。

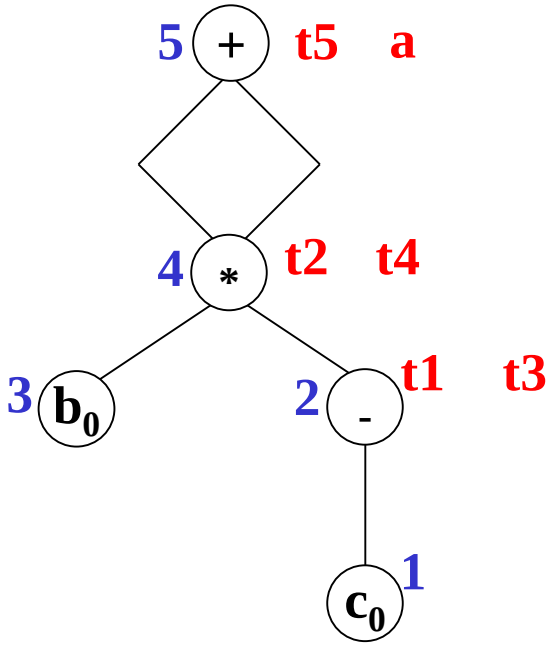
算法14.2 构建DAG图的算法—消除公共子表达式（续）

4. 在DAG图中寻找中间结点，其标记为op，且其左操作数结点号为i，右操作数结点号为j。如果找到，记录下其结点号k；如果未找到，在DAG图中新建一个中间结点，假设其结点号仍为k，并将结点i和j分别与k相连，作为其左子结点和右子结点。
5. 在结点表中寻找z，如果找到，将z所对应的结点号更改为k；如果未找到，在结点表中新建一项(z, k)，表明二者之间的对应关系。
6. 对输入的中间代码序列依次重复上述步骤3～5。

3. 对于形如z = x op y的中间代码，其中z为记录计算结果的变量名，x为左操作数，y为右操作数，op为操作符：首先在结点表中寻找x，如果找到，记录下x当前所对应的结点号i；如果未找到，在DAG图中新建一个叶结点，假设其结点号仍为i，标记为x（如x为变量名，该标记更改为x_0）；在结点表中增加新的一项(x, i)，表明二者之间的对应关系。右操作数y与x同理，假设其对应结点号为j。

建立DAG图，例1 a=b*(-c) + b*(-c)

```
t1 := - c
t2 := b * t1
t3 := - c
t4 := b * t3
t5 := t2 + t4
a := t5
```



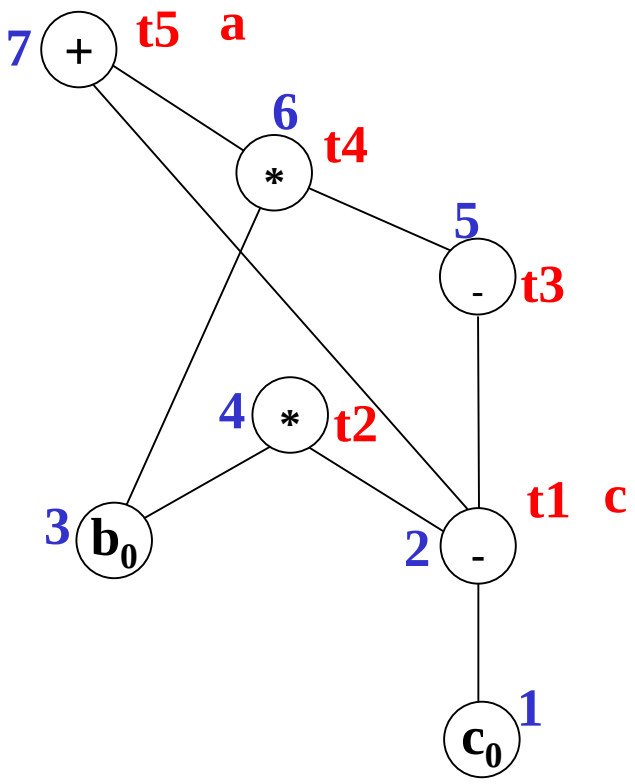
node(x)

c	1
t1	2
b	3
t2	4
t3	2
t4	4
t5	5
a	5

如中间代码形为z=x，只需按照规则3找到x所对应的结点i，再按照规则5，将z所对应的结点号更改为i，或在结点表中新建一项(z, i)即可。

建立DAG图，例2

```
t1 := - c
t2 := b * t1
c := t1
t3 := - c
t4 := b * t3
t5 := c + t4
a := t5
```



node(x)

c	2
t1	2
b	3
t2	4
t3	5
t4	6
t5	7
a	7

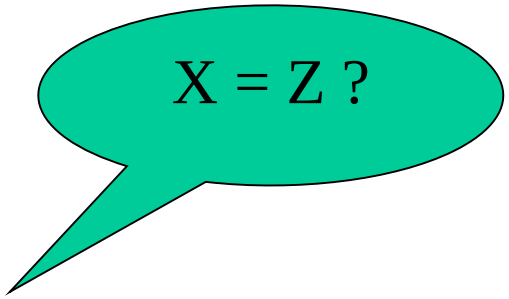
数组、指针及函数调用的DAG图

当中间代码序列中出现了数组成员、指针或函数调用时，算法11.2需要作出一定的调整，否则将得出不正确的优化结果。

例：x = a[i]

a[j] = y

z = a[i]



不一定！
因为如果 j = i

将数组变量a作为一个单独的变量进行考虑，将形如x = a[i]的中间代码都表示为x = a []i，其中[]为数组取值操作符；形如a[j] = y的中间代码都表示为a = j “[]=” y，其中“[]=”为数组成员赋值操作符。

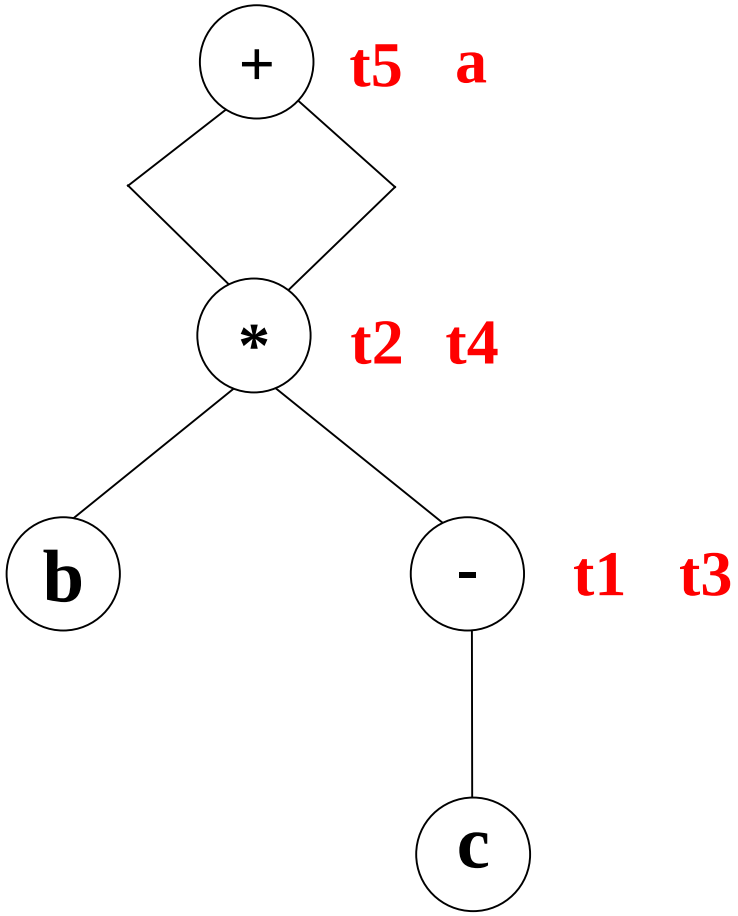
指针：保守处理

```
x = *p

*q = y

z = *p
```

函数调用

在缺乏跨函数数据流分析的支持下，需要保守地假设函数调用改变了所有它可能改变的数据。

从DAG图重新导出中间代码

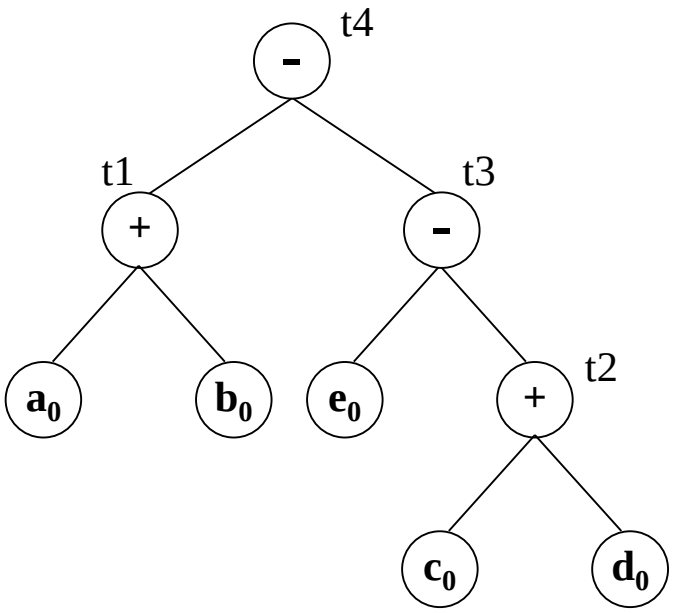


t1 := -c

t2 := b * t1

a := t2 + t2

从DAG图重新导出中间代码



(1)
t1 = a + b
t2 = c + d
t3 = e – t2
t4 = t1 – t3

(2)
t2 = c + d
t3 = e – t2
t1 = a + b
t4 = t1 – t3

从DAG图重新导出中间代码

(1) t1 = a + b
t2 = c + d
t3 = e – t2
t4 = t1 – t3

```
mov  eax,a              ;  t1=a+b
add  eax,b
mov  edx,c              ;  t2=c+d
add  edx,d
mov  [ESP+08H],eax      ;  暂存t1(a+b)
mov  eax,e
sub  eax,edx            ;  t3=e-t2
mov  [ESP+0CH],edx      ;  暂存t2(c+d)
mov  edx,[ESP+08H]      ;  取回t1
sub  edx,eax            ;  t4=t1-t3
```

假设:

- **局部变量a, b, c, d, e 均不占用寄存器**
- **仅有两个寄存器eax, edx 可供使用**

[ESP+08H],[ESP+0CH]

均为临时变量在运行栈上的临时保存单元地址

补偿代码

(1) t1 = a + b
t2 = c + d
t3 = e – t2
t4 = t1 – t3

```
mov eax, a
add eax, b          ; t1 = a+b
mov edx, c
add edx, d          ; t2 = c+d
mov [ESP+08H], eax
mov eax, e
sub eax, edx        ; t3 = e-t2
mov [ESP+0CH], edx
mov edx, [ESP+08H]
sub edx, eax        ; t4 = t1-t3
```

(2) t2 = c + d
t3 = e – t2
t1 = a + b
t4 = t1 – t3

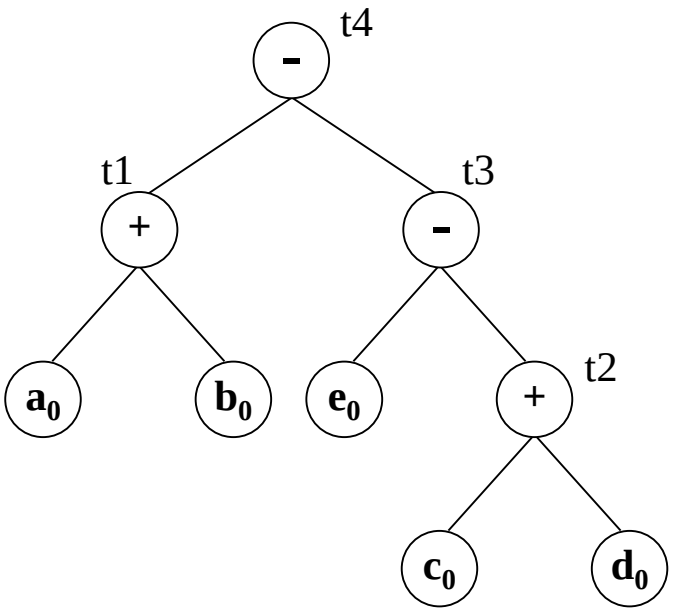
```
mov eax, c
add eax, d          ; t2 = c+d
mov edx, e
sub edx, eax        ; t3 = e-t2
mov [ESP+0CH], eax
mov eax, a
add eax, b          ; t1 = a+b
mov [ESP+08H], eax
sub eax, edx        ; t4 = t1-t3
```

算法14.3 从DAG导出中间代码的启发式算法

- 输入：DAG图
- 输出：中间代码序列
- 方法：

1. 初始化一个放置DAG图中间结点的队列。
2. 如果DAG图中还有中间结点未进入队列，则执行步骤3，否则执行步骤5
3. 选取一个尚未进入队列，但其所有父结点均已进入队列的中间结点n，将其加入队列；或选取没有父结点的中间结点，将其加入队列
4. 如果n的最左子结点符合步骤3的条件，将其加入队列；并沿着当前结点的最左边，循环访问其最左子结点，最左子结点的最左子结点等，将符合步骤3条件的中间结点依次加入队列；如果出现不符合步骤3条件的最左子结点，执行步骤2
5. 将中间结点队列逆序输出，便得到中间结点的计算顺序，将其整理成中间代码序列

算法14.3 从DAG导出中间代码的启发式算法



1、初始化一个放置DAG图中间结点的队列。

2、如果DAG图中还有中间结点未进入队列，则执行步骤3，否则执行步骤5。

3、选取一个尚未进入队列，但其**所有父结点均已进入队列**的中间结点n，将其加入队列；或选取**没有父结点的**中间结点，将其加入队列。

4、如果n的最左子结点符合步骤3的条件，将其加入队列；并沿着当前结点的最左边，循环访问其**最左子结点**，最左子结点的最左子结点等，将符合步骤3条件的中间结点依次加入队列；如果出现不符合步骤3条件的最左子结点，执行步骤2。

5、将中间结点队列逆序输出，便得到中间节点的计算顺序，将其整理成中间代码序列。

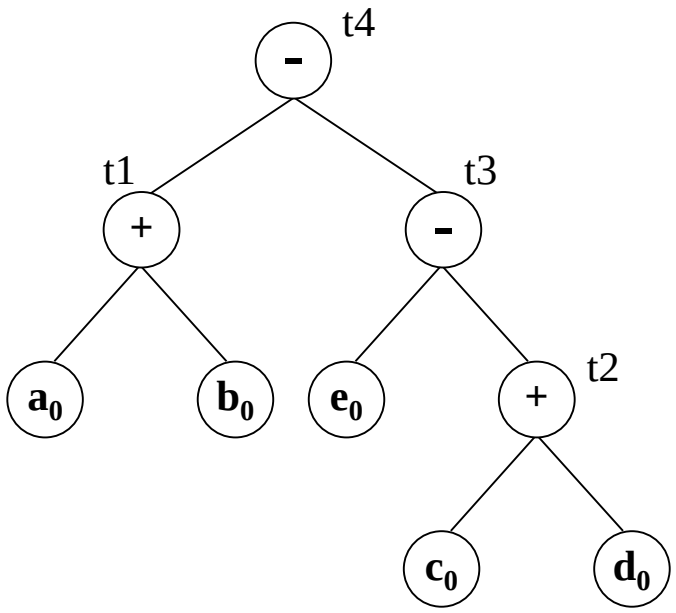
中间结点队列：

t4	t1	t3	t2

中间结点队列：

t4	t1	t3	t2

5、将中间结点队列逆序输出，便得到中间节点的计算顺序，将其整理成中间代码序列。



$t2 = c + d$

$t3 = e - t2$

$t1 = a + b$

$t4 = t1 - t3$

14.2.3 窥孔优化

- 窥孔优化关注在目标指令的一个较短的序列上，通常称其为“窥孔”（peephole）。
- 通过删除其中的冗余代码，或者用更高效简洁的新代码来替代其中的部分代码，达到提升目标代码质量的目的 。

窥孔优化并不局限在同一个基本块中，可跨越基本块，甚至包含多个基本块。

```
mov  EAX, [ESP+08H]          jmp B2
mov  [ESP+08H], EAX          B2: …
```

必须出现在同一个基本块中，程序运行不可能从其他指令转移到它们中间。

14.3 全局优化

14.3.1 数据流分析

- 用于获取数据在程序执行路径中如何流动的有关信息。例如:
  - 某个变量在某个特定的执行点（语句前后）是否还“存活”？
  - 某个变量的值，是在什么地方定义的？
  - 某个变量在某一执行点上被定义的值，可能在哪些其他执行点被使用？
- 是全局优化的基础

数据流分析方程

含义：当执行控制流通过S时，在S末尾得到的数据流信息等于S本身产生的数据流信息，合并进入S时的数据流信息减去S注销的数据流信息后的数据流信息。

- 考察在程序的某个执行点的[illegible]流信息。
- **out[S] = gen[S] ∪ (in[S] – kill[S])**
  - S代表某条语句（也可以是基本块，或者语句集合，或者基本块集合等）
  - out[S]代表在S末尾得到的数据流信息
  - gen[S]代表S本身产生的数据流信息
  - in[S]代表进入S时的数据流信息
  - kill[S]代表S注销的数据流信息
  - “∪”和“–”均为集合运算

数据流方程求解过程中的3个关键因素

- 当前语句产生和注销的信息取决于需要解决的具体问题：可以由in[S]定义out[S]，也可以反向定义，由out[S]定义in[S]。
- 由于数据是沿着程序的执行路径，也就是控制流路径流动，因此数据流分析的结果受到程序控制结构的影响 。
- 代码中出现的诸如过程调用、指针访问以及数组成员访问等操作，对定义和求解一个数据流方程都会带来不同程度的困难 。

程序的状态

- 程序的执行过程 = 程序状态的变换过程
  - 程序状态由程序中的变量和其它数据结构组成。
  - 每一条执行指令都可能改变程序的状态。
- 通过数据流分析，可以了解程序的状态。
  - 例如，如果得知在某条中间代码之后，无论程序在实际执行时通过哪条路径，某个变量都不会再被访问，那么该变量此前所占有的全局寄存器或临时寄存器就可以安全地被其他变量重新使用。
  - 例如，如果得知在程序的某个点上，对某个变量进行引用时，无论程序如何运行，该变量都仅具有某个唯一的常量值，那么就可以将该常量引入中间代码，在代码生成时生成更高效的指令。
- 一种常用的数据流分析方法：到达定义。

到达定义（reaching definition）分析

通过到达定义分析，希望知道:

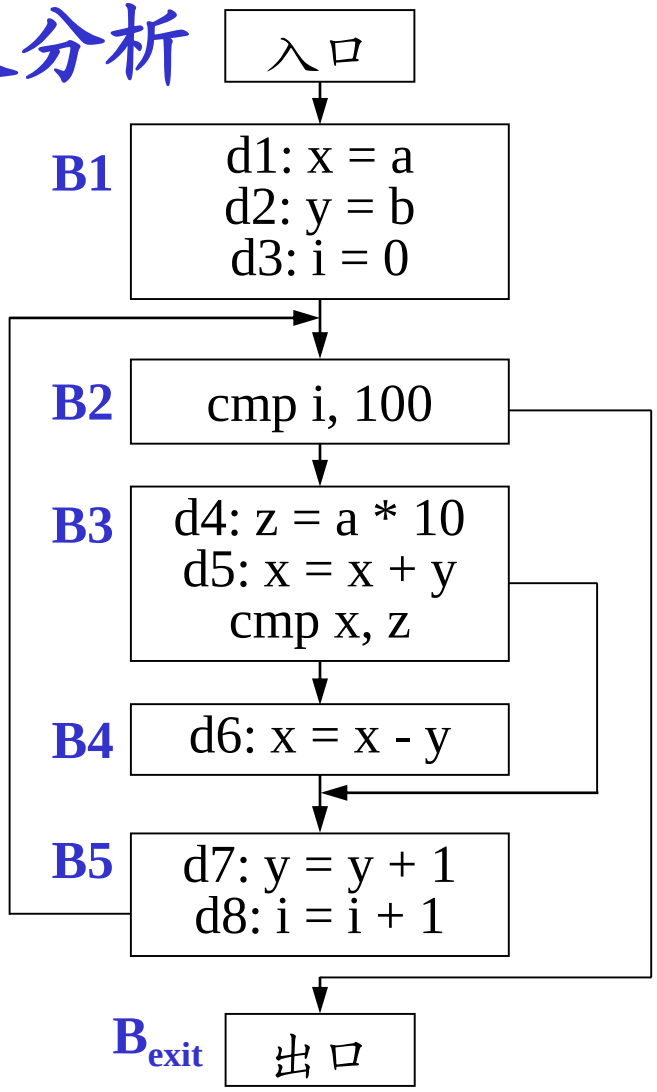
- 在程序的某个静态点p，例如某条中间代码之前或者之后，某个变量可能出现的值都是在哪里被定义的?
- 在p处对该变量的引用，取得的值是否在d处定义?
  - 如果从定义点d出发，存在一条路径达到p，并且在该路径上，不存在对该变量的其他定义语句，则认为“变量的定义点d到达静态点p”。
  - 如果路径上存在对该变量的其他赋值语句，那么路径上的前一个定义点就被路径上的后一个定义点“杀死”，或者消除了。

到达定义分析

说明：

- 变量的定义：赋值语句、过程参数、指针引用等多种形式。
- 不能判断时：保守处理

到达定义分析



gen[B1] = {d1, d2, d3}
kill[B1] = {d5, d6, d7, d8}

gen[B2] = { }
kill[B2] = { }

gen[B3] = {d4, d5}
kill[B3] = {d1, d6}

gen[B4] = {d6}
kill[B4] = {d1, d5}

gen[B5] = {d7, d8}
kill[B5] = {d2, d3}

d1~d8 八个定义点

单条语句的到达定义数据流方程

- 对于基本块中的某一条中间代码:

  d1:　u = v op w，v和w为变量，op为操作符

- 代码对应的到达定义数据流方程是:

  $$out[d1] = gen[d1] \cup (in[d1] - kill[d1])$$

- 其中
  - **gen[d1] = {d1}**，表明该语句产生了一个定义点（定义了变量u）
  - **kill[d1]** 是程序中所有对变量u定义的其他定义点的集合（包括d1之前或之后的定义点）
  - 对于该代码在同一基本块中紧邻的后继代码，假设其为d2，in[d2]等价于out[d1]

多条语句的到达定义数据流方程

$$out[d1] = gen[d1] \cup (in[d1] - kill[d1])$$

$in[d2] = out[d1]$

$$out[d2] = gen[d2] \cup (in[d2] - kill[d2])$$

$in[d3] = out[d2]$

$$out[d3] = gen[d3] \cup (in[d3] - kill[d3])$$

……

$in[dn] = out[d(n-1)]$

$$out[dn] = gen[dn] \cup (in[dn] - kill[dn])$$

基本块B的到达定义数据流方程

- out[B] = gen[B]∪(in[B] – kill[B])
  - in[B]为进入基本块B时的数据流信息
  - kill[B] = kill[d1]∪kill[d2]…∪kill[dn]， d1~dn依次为基本块中的语句
  - gen[B] = gen[dn]∪

    (gen[d(n-1)] – kill[dn])∪

    (gen[d(n-2)] – kill[d(n-1)] – kill[dn])…∪

    (gen[d1] – kill[d2] – kill[d3]… – kill[dn])

- out[B] = gen[B]∪(in[B] – kill[B])
  - in[B]为进入基本块B时的数据流信息
  - kill[B] = kill[d1]∪kill[d2]…∪kill[dn]， d1~dn依次为基本块中的语句
  - gen[B] = gen[dn]∪
    (gen[d(n-1)] – kill[dn])∪
    (gen[d(n-2)] – kill[d(n-1)] – kill[dn])…∪
    (gen[d1] – kill[d2] – kill[d3]… – kill[dn])

例:

d1: a = b + 1
d2: a = b + 2

kill[B] = kill[d1]∪kill[d2] = {d2}∪{d1} = {d1, **d2**}

同时存在！

gen[B] = gen[d2]∪(gen[d1]–kill[d2]) = {d2}∪({d1}–{d1}) = {**d2**}

out[B] = gen[B]∪(in[B]–kill[B]) = {d2}∪(in[B]–{d1, d2})

可以看出，不论in[B]中包含哪些定义点，B出口处的到达定义数据流信息out[B]中肯定包含定义点d2，且肯定不包含d1。

算法14.5 基本块的到达定义数据流分析

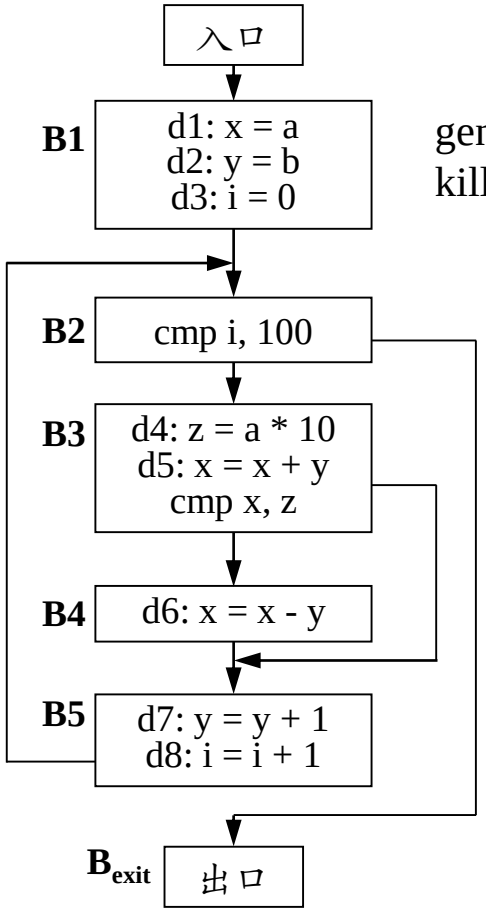
- 输入：程序流图，且基本块的kill集合和gen集合已经计算完毕
- 输出：每个基本块入口和出口处的in和out集合，即in[B]和out[B]
- 方法：

1. 将包括代表流图出口基本块B_{exit}的所有基本块的out集合，初始化为空集。
2. 根据方程**$in[B] = \cup_{B的前驱基本块P} out[P]$**，$out[B] = gen[B] \cup (in[B] – kill[B])$，为每个基本块B依次计算集合in[B]和out[B]。如果某个基本块计算得到的out[B]与该基本块此前计算得出的out[B]不同，则循环执行步骤2，直到所有基本块的out[B]集合不再产生变化为止。

例：到达定义数据流分析

$in[B] = \cup_{B的前驱基本块P} out[P]$

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



gen[B1] = {d1, d2, d3}
kill[B1] = {d5, d6, d7, d8}

gen[B2] = { }
kill[B2] = { }

gen[B3] = {d4, d5}
kill[B3] = {d1, d6}

gen[B4] = {d6}
kill[B4] = {d1, d5}

gen[B5] = {d7, d8}
kill[B5] = {d2, d3}

第一次步骤2:

in[B1] = {}, out[B1] = {d1, d2, d3}

B2的前驱为B1和B5

in[B2] = {d1, d2, d3}, out[B2] = {d1, d2, d3}

B3的前驱为B2

in[B3] = {d1, d2, d3}, out[B3] = { d2, d3, d4, d5}

B4的前驱为B3

in[B4] = {d2, d3, d4, d5}, out[B4] = {d2, d3, d4, d6}

B5的前驱为B3和B4

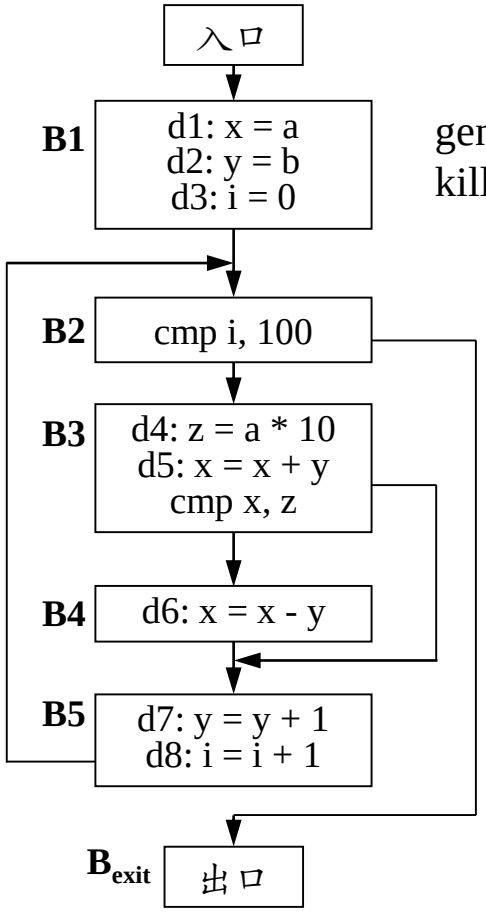
in[B5] = {d2, d3, d4, d5, d6}, out[B5]
= {d4, d5, d6, d7, d8}

Bexit的前驱为B2

in[Bexit] = {d1, d2, d3}

例：到达定义数据流分析

$in[B] = \cup_{B的前驱基本块P} out[P]$
$out[B] = gen[B] \cup (in[B] - kill[B])$



gen[B1] = {d1, d2, d3}
kill[B1] = {d5, d6, d7, d8}

gen[B2] = { }
kill[B2] = { }

gen[B3] = {d4, d5}
kill[B3] = {d1, d6}

gen[B4] = {d6}
kill[B4] = {d1, d5}

gen[B5] = {d7, d8}
kill[B5] = {d2, d3}

第二次步骤2:

in[B1] = {}, out[B1] = {d1, d2, d3}

in[B2] = {d1, d2, d3, d4, d5, d6, d7, d8}
out[B2] = {d1, d2, d3, d4, d5, d6, d7, d8}

in[B3] = {d1, d2, d3, d4, d5, d6, d7, d8}
out[B3] = {d2, d3, d4, d5, d7, d8}

in[B4] = {d2, d3, d4, d5, d7, d8}
out[B4] = {d2, d3, d4, d6, d7, d8}

in[B5] = {d2, d3, d4, d5, d6, d7, d8}
out[B5] = {d4, d5, d6, d7, d8}

$in[B_{exit}]$ = {d1, d2, d3, d4, d5, d6, d7, d8}

实现

- 集合“∪”和“–”运算：可以采用位向量（Bit Vector）的方式完成。
- 将集合中的每个定义点，根据其下标映射为一个无限位二进制数的某一位，例如，可以将d1映射为第1位，d3映射为第3位，以此类推。
  - 例如，out[B3] = { d2, d3, d4, d5, d7, d8}，其对应的二进制位向量为11011110，该位向量从低位到高位依次对应d1~d8。
  - 基于这样的设定，集合之间的“∪”运算等价于位向量之间的或运算，集合之间的“–”运算等价于将后者取补（取反）后，和前者进行按位与运算。
- 在数据流分析方法的实现中，位向量是常用的手段之一。

14.3.2 活跃变量分析（Live-variable Analysis）

- 到达定义分析是沿着流图路径的，有的数据流分析是反方向计算的。
- 活跃变量分析:
  - 了解变量x在某个执行点p是活跃的
    - 变量x的值在p点或沿着从p出发的某条路经中会被使用，则称x在p点是活跃的。
  - 通过活跃变量分析，可以了解到某个变量x在程序的某个点上是否活跃，或者从该点出发的某条路径上是否会被使用。如果存在被使用的可能，x在该程序点上便是活跃的；否则就是非活跃，或者死的。

14.3.2 活跃变量分析（Live-variable Analysis）

- 活跃变量信息对于寄存器分配，不论是全局寄存器分配还是临时寄存器分配都有重要意义。
  - 如果拥有寄存器的**变量x**在p点开始的任何路径上**不再活跃**，可以释放寄存器。
  - 如果两个变量的**活跃范围不重合**，则可以共享同一个寄存器。

活跃变量分析

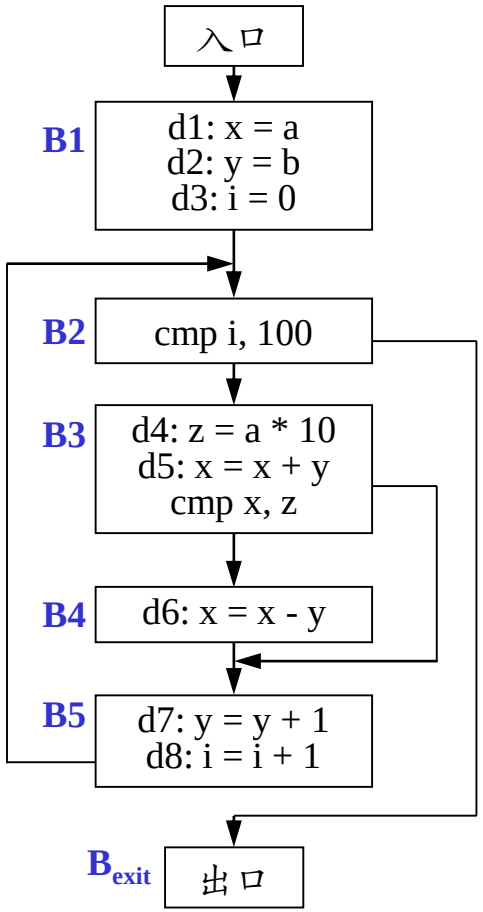
- 数据流方程如下：

  $in[B] = use[B] \cup (out[B] - def[B])$

  - $out[B] = \cup_{B的后继基本块P} in[P]$
  - def[B]：变量在B中被定义（赋值）先于任何对它们的使用【先定义后使用】。
  - use[B]：变量在B中被使用先于任何对它们的定义【先使用后定义】。

- 到达定义数据流分析，其数据流信息是沿着流图中路径的方向进行计算的。
- 活跃变量分析的数据流信息，需要沿着流图路径的反方向计算得出。

活跃变量分析



d1~d8 八个定义点

- def[B]：变量在B中**被定义（赋值）先于**任何对它们的使用。【先定义后使用】。
- use[B]：变量在B中**被使用先于**任何对它们的定义。【先使用后定义】

use[B3] = {a, x, y}

def[B3] = {z}

与到达定义分析的区别

活跃变量分析:
$in[B] = use[B] \cup (out[B] - def[B])$

到达定义分析:
$out[B] = gen[B] \cup (in[B] - kill[B])$

- 采用use[B]代表当前基本块新生成的数据流信息（用了）。
- 采用def[B]代表当前基本块消除的数据流信息（定义的）。
- 采用in[B]而不是out[B]来计算当前基本块中的数据流信息。
- 采用out[B]而不是in[B]来计算其它基本块汇集到当前基本块的数据流信息。
- 在汇集数据流信息时，考虑的是后继基本块而不是前驱基本块。

活跃变量分析的直观理解：如果在路径后方的某个基本块中，变量x被使用，则沿着执行路径的逆向直到x被定义的基本块，x都是活跃的。

算法14.5 基本块的活跃变量数据流分析

- 输入：程序流图，且基本块的use集合和def集合已经计算完毕
- 输出：每个基本块入口和出口处的in和out集合，即in[B]和out[B]
- 方法：

1. 将包括代表流图出口基本块B_{exit}在内的所有基本块的in集合，初始化为空集。
2. 根据方程$out[B] = \cup_{B的后继基本块P} in[P]$，$in[B] = use[B] \cup (out[B] - def[B])$，为每个基本块B依次计算集合out[B]和in[B]。如果计算得到某个基本块的in[B]与此前计算得出的该基本块in[B]不同，则循环执行步骤2，直到所有基本块的in[B]集合不再产生变化为止。

for 每个基本块B do in[B] = Φ ;

while 集合in发生变化 do

for 每个基本块B do begin

$out[B] = \cup_{B的所有后继S} in[S]$

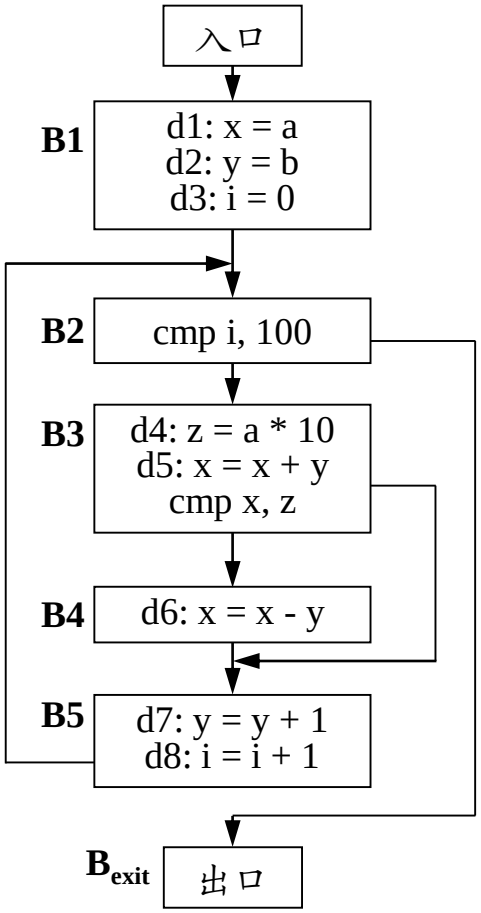
$in[B] = use[B] \cup (out[B] - def[B])$

end

for 每个基本块B do in[B] = Φ ;
while 集合in发生变化 do
    for 每个基本块B do begin
        out[B] = ∪ $_{B的所有后继S}$ in[S]
        in[B] = use[B] ∪ (out[B] – def[B])
    end

例

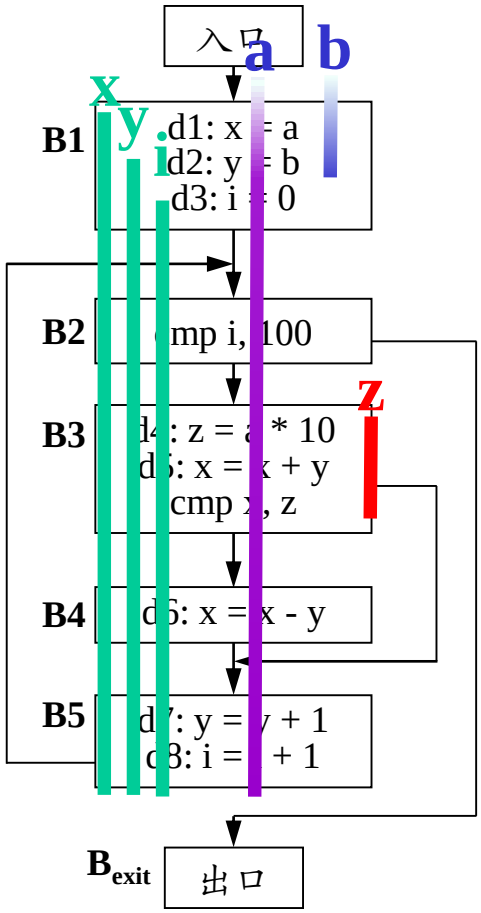
流图



	def[B]	use[B]	in[B]	out[B]	in[B]	out[B]	in[B]	out[B]
B1	x, y, i	a, b	a, b	a,x,y,i	a, b	a,x,y,i	a, b	a,x,y,i
B2	Φ	i	a,x,y,i	a,x,y,i	a,x,y,i	a,x,y,i	a,x,y,i	a,x,y,i
B3	z	a, x, y	a,x,y,i	x, y, i	a,x,y,i	a,x,y,i	a,x,y,i	a,x,y,i
B4	Φ	x, y	x, y, i	y, i	a,x,y,i	a,x,y,i	a,x,y,i	a,x,y,i
B5	Φ	y, i	y, i	Φ	a,x,y,i	a,x,y,i	a,x,y,i	a,x,y,i
B exit			Φ	Φ	Φ	Φ	Φ	Φ

从活跃变量到冲突图

流图



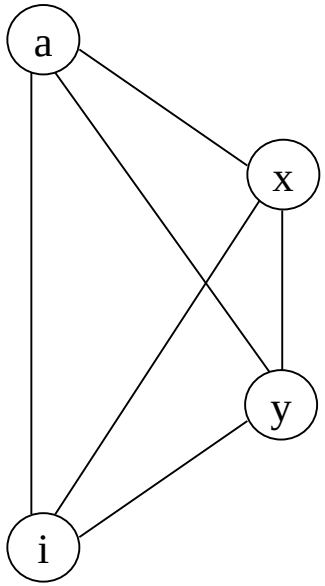
	in[B]	out[B]
B1	a, b	a,x,y,i
B2	a,x,y,i	a,x,y,i
B3	a,x,y,i	a,x,y,i
B4	a,x,y,i	a,x,y,i
B5	a,x,y,i	a,x,y,i
B_{exit}	Φ	Φ

- 变量x, y, i: 均定义于B1，在B2~B5入口处均活跃。
  注意，x在B3、B4中都被重新定义过，但x被定义前均被使用过，因此其在同一基本块中发生在使用之前的定义仅余B1。变量y和i的情况类似。
- 变量a: 在流图中无定义点，在B1~B5入口处均活跃。
- 变量b: 在流图中无定义点，在B1入口处活跃。
- 变量z: 定义于B3，且仅在B3中被使用。

冲突图

	in[B]	out[B]
B1	a, b	a,x,y,i
B2	a,x,y,i	a,x,y,i
B3	a,x,y,i	a,x,y,i
B4	a,x,y,i	a,x,y,i
B5	a,x,y,i	a,x,y,i
B_{exit}	Φ	Φ

假设只有跨越基本块活跃的变量才能分配到全局寄存器，并且**活跃范围重合**的变量之间无法共享全局寄存器。

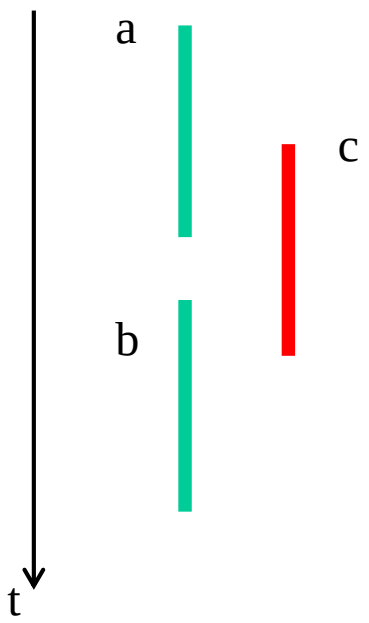


根据活跃变量分析计算得出的变量冲突图

14.3.3 定义-使用链、网和冲突图

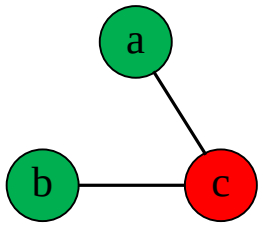
- **冲突图**：其结点是待分配全局寄存器的变量，当**两个变量中的一个变量在另一个变量定义（赋值）处是活跃的**，它们之间便有一条边连接。

活跃变量冲突的定义



- 变量a和变量b不冲突
- 变量a和变量c冲突
- 变量b和变量c冲突

不冲突可以着同种色



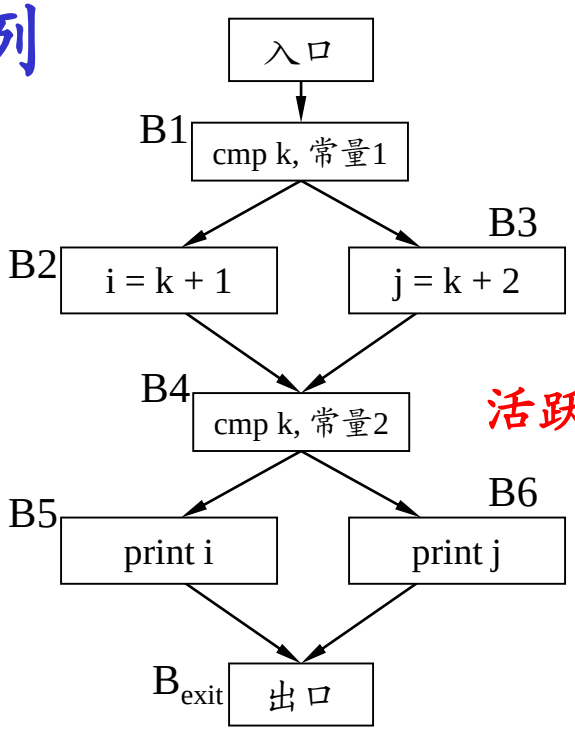
(b)

a和c使用不同的寄存器
b和c使用不同的寄存器

冲突图：连线多画了，不影响程序的正确性；
少连线了，会影响程序的正确性。

活跃变量冲突的不同定义

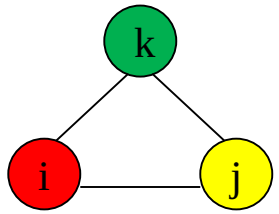
又例



(a)

活跃变量: **i, j, k**

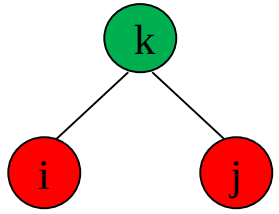
- 变量 i 和变量 j 在B2和B3中被分别定义，并在B5和B6中被分别使用。根据活跃变量分析结果，i和j一定同时在B4的入口处活跃



(b)

但即使i和j使用同一寄存器，程序运行结果仍符合语义

冲突图中两个结点（变量）间存在边的**条件约束**为:
其中一个变量在另一个变量的定义点处活跃!

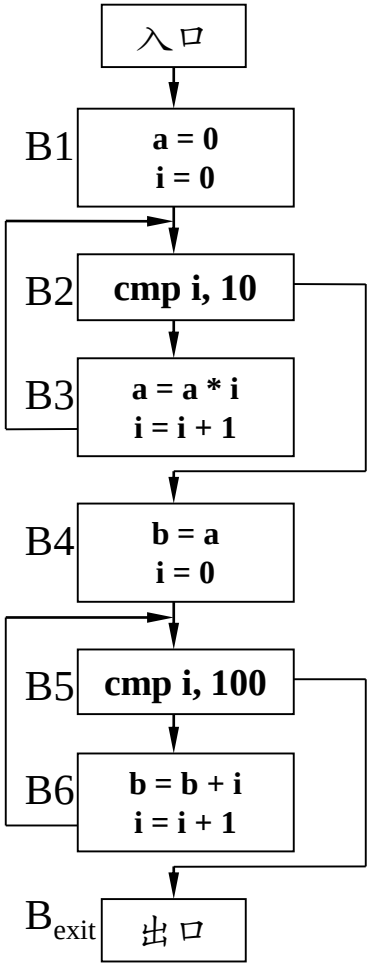


(c)

关于变量冲突的判断

- 两个变量中的一个变量在另一个变量定义（赋值）处是活跃的，它们就是冲突的。
  - 算法一：在每一个变量的定义点计算活跃变量。
  - 算法二：计算基本块入口处的活跃变量（in的集合），这些变量在该基本块中的定义点活跃，因而冲突。之后，在基本块内部，进一步计算每个定义点的活跃变量（基本块范围内计算），降低了计算的复杂度，因为基本块内部是线性的。

被多次定义的变量和冲突图

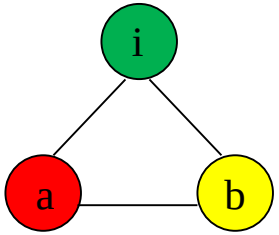


(a)

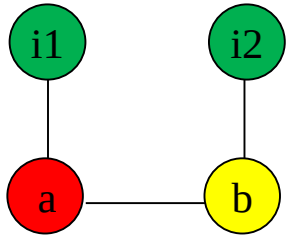
变量i在第一个循环中被定义和使用，执行第二个循环前，i被重新定义和使用。

变量a或变量b伴随着变量i一同使用。

变量i在第一个循环和第二个循环中，是否可以使用不同的全局寄存器？



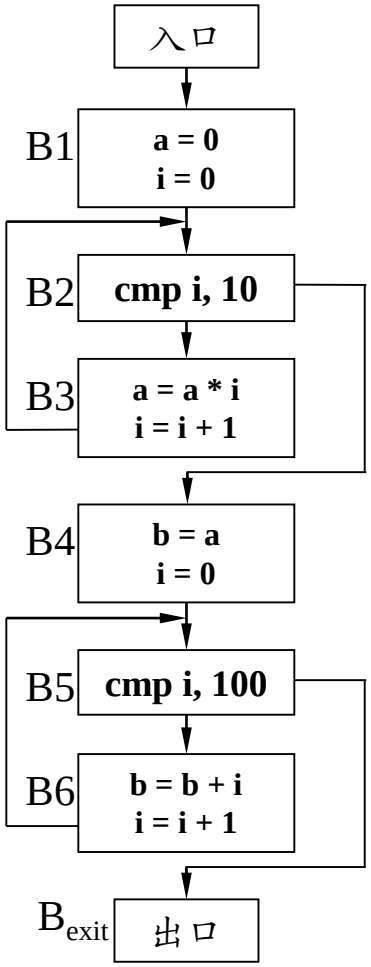
(b)



(c)

Define-use链

所谓**变量的定义-使用链**，是指变量的某一定义点，以及所有可能使用该定义点所定义变量值的使用点所组成的一个链



变量a: **L1 {<B1, 1>, <B3, 1>, <B4, 1>}**

L2 {<B3, 1>, <B3, 1>, <B4, 1>}

变量b: **L3 {<B4, 1>, <B6, 1>}**

L4 {<B6, 1>, <B6, 1>}

变量i: **L5 {<B1, 2>, <B2, 1>, <B3, 1>, <B3, 2>}**

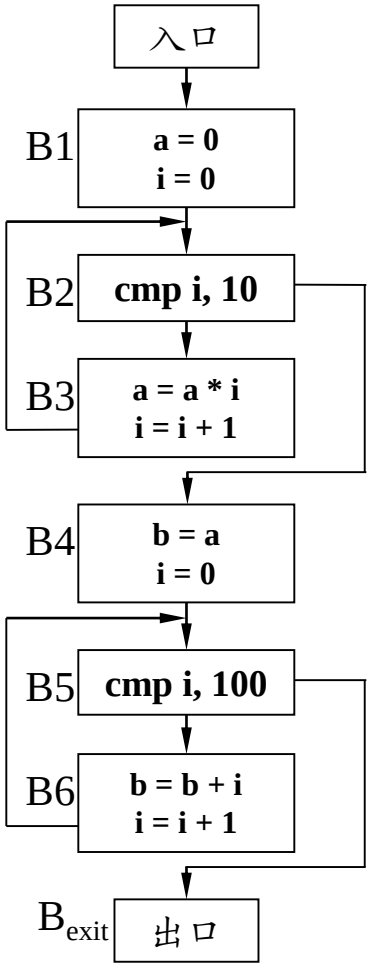
L6 {<B3, 2>, <B2, 1>, <B3, 1>, <B3, 2>}

L7 {<B4, 2>, <B5, 1>, <B6, 1>, <B6, 2>}

L8 {<B6, 2>, <B5, 1>, <B6, 1>, <B6, 2>}

Define-use链

所谓**变量的定义-使用链**，是指变量的某一定义点，以及所有可能使用该定义点所定义变量值的使用点所组成的一个链



变量a: **L1 {<B1, 1>, <B3, 1>, <B4, 1>}**

L2 {<B3, 1>, <B3, 1>, <B4, 1>}

变量b: **L3 {<B4, 1>, <B6, 1>}**

L4 {<B6, 1>, <B6, 1>}

变量i: **L5 {<B1, 2>, <B2, 1>, <B3, 1>, <B3, 2>}**

L6 {<B3, 2>, <B2, 1>, <B3, 1>, <B3, 2>}

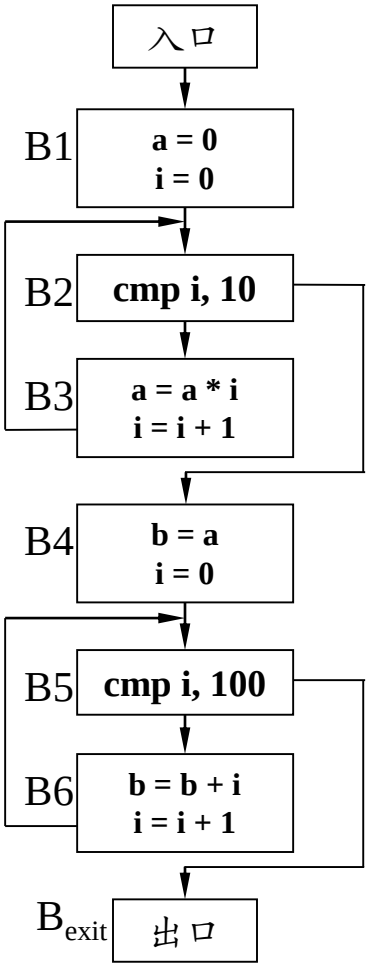
L7 {<B4, 2>, <B5, 1>, <B6, 1>, <B6, 2>}

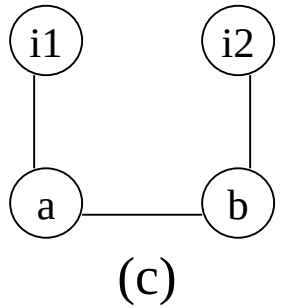
L8 {<B6, 2>, <B5, 1>, <B6, 1>, <B6, 2>}

可以发现：L5、L6和L7、L8是没有关系的。

从后面的网可以发现，同一个变量的定义-使用链分裂了，是两个网！

同一变量的多个定义-使用链，如果它们拥有某个同样的使用点，则合并为同一个网





(c)

变量a： L1 {<B1, 1>, <B3, 1>, <B4, 1>}
L2 {<B3, 1>, <B3, 1>, <B4, 1>}

变量b： L3 {<B4, 1>, <B6, 1>}
L4 {<B6, 1>, <B6, 1>}

变量i： L5 {<B1, 2>, <B2, 1>, <B3, 1>, <B3, 2>}
L6 {<B3, 2>, <B2, 1>, <B3, 1>, <B3, 2>}
L7 {<B4, 2>, <B5, 1>, <B6, 1>, <B6, 2>}
L8 {<B6, 2>, <B5, 1>, <B6, 1>, <B6, 2>}

变量a： W1 { L1 {<B1, 1>, <B3, 1>, <B4, 1>}, L2 {<B3, 1>, <B3, 1>, <B4, 1>}}

变量b： W2 { L3 {<B4, 1>, <B6, 1>}, L4 {<B6, 1>, <B6, 1>}}

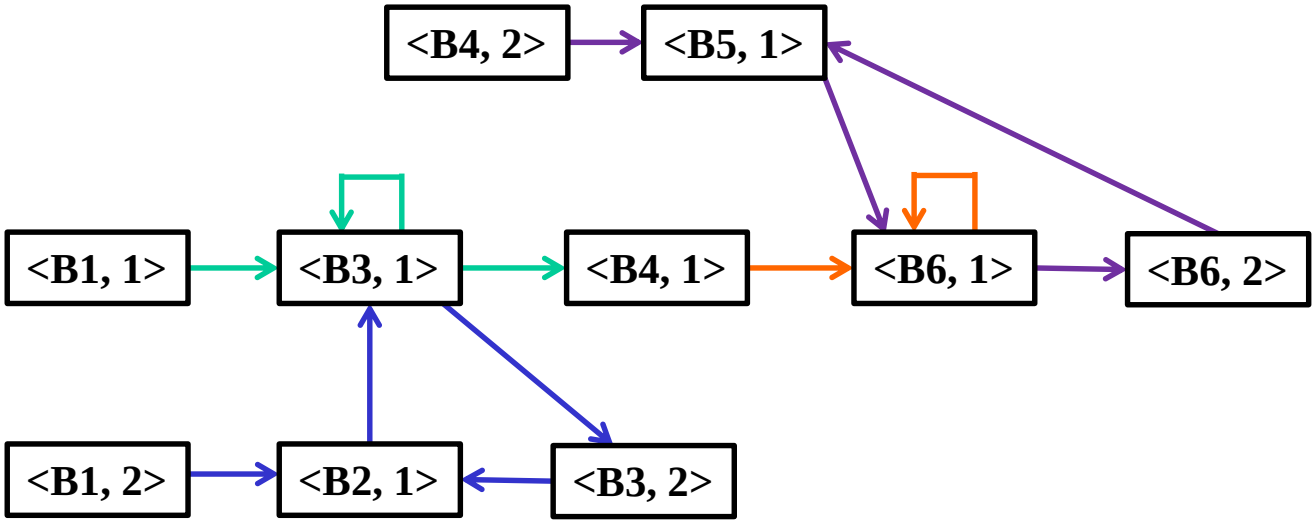
变量i： W3 { L5 {<B1, 2>, <B2, 1>, <B3, 1>, <B3, 2>}, L6 {<B3, 2>, <B2, 1>, <B3, 1>, <B3, 2>}}
W4 { L7 {<B4, 2>, <B5, 1>, <B6, 1>, <B6, 2>}, L8 {<B6, 2>, <B5, 1>, <B6, 1>, <B6, 2>}}

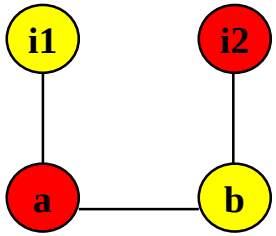
变量a： W1 { L1 {<B1, 1>, <B3, 1>, <B4, 1>}, L2 {<B3, 1>, <B3, 1>, <B4, 1>}}

变量b： W2 { L3 {<B4, 1>, <B6, 1>}, L4 {<B6, 1>, <B6, 1>}}

变量i： W3 { L5 {<B1, 2>, <B2, 1>, <B3, 1>, <B3, 2>}, L6 {<B3, 2>, <B2, 1>, <B3, 1>, <B3, 2>}}

W4 { L7 {<B4, 2>, <B5, 1>, <B6, 1>, <B6, 2>}, L8 {<B6, 2>, <B5, 1>, <B6, 1>, <B6, 2>}}





消除全局公共子表达式

- 自学

14.4 循环优化

经验告诉我们：“程序运行时间的80%是由仅占源程序20%的部分执行的”。——二八定律 (Pareto Principle)

这20%的源程序就是循环部分，特别是多重循环的最内层的循环部分。因此减少循环部分的目标代码对提高整个程序的时间效率有很大作用。

```
for  i = 1  to  10
   for   j = 1  to  100
        x := x + 0 ;
        y := 5 + 7 + x ;
```

优化一条，少10*100次运算

除了对循环体进行优化，还有专用于循环的优化

a) 循环不变式的代码外提

不变表达式：不随循环控制变量改变而改变的表达式或子表达式。

如：FOR I := E_1 STEP E_2 TO E_3 DO

BEGIN

S := 0.2 * 3.1416 * R

P := 0.35 * I

V := S * P

……

不变表达式可外提

不能外提!

如　　while　…　do

　　x := …（b * b - 4.0 * a * c）…

若a, b, c的值在该循环体中不改变时，则可将循环不变式移到循环之外，即变为:

t_1 := b * b - 4.0 * a * c

while　…　do

　　x := …(t_1) …

从而减少计算次数——也称为频度削弱。

b) 循环展开

循环展开是一种优化技术。它将构成循环体的代码(不包括控制循环的测试和转移部分)，重新产生许多次(这可在编译时确定)，而不仅仅是一次。以空间换时间!

例：PL/1中的初始化循环

```
DO  I = 1  TO  30
    A[ I ] = 0.0
END
```

展开

```
      I := 1
L1:   IF  I > 30  THEN
      GOTO  L2
      A[ I ] = 0.0
      I = I + 1
      GOTO  L1
L2:
```

代码5条语句
共执行5*30
条语句

```
A[1] = 0.0
A[2] = 0.0
……
A[30] = 0.0
```

30条语句
（指令）执行
也是30条语句

说明：

- 循环一次执行5条语句才给一个变量赋初值。展开后，一条语句就能赋一个值，运行效率高。
- 优化在生成代码时进行，并不是修改源程序。
- 必须知道循环的初值、终值及步长。
- 但非所有展开都是合适的。如上例中循环展开后节省了测试和转移语句：2*30=60语句。

完成具体业务逻辑的代码 / 辅助性代码

∴增加29条省60条

但若循环体中不是一条而是40条，则展开将有40*30=1200条，但省的仍是60条，就不算优化了。

∴判断准则：1. 主存资源丰富
处理机时间昂贵
2. 循环体语句越少越好

⇒ 循环展开有利（大型机）

```
DO    I = 1   TO   30
      A[ I ] = 0.0
END
```

实现步骤：

1. 识别循环结构，确定循环的初值、终值和步长。
2. 判断。以空间换时间是否合算来决定是否展开。
3. 展开。重复产生循环体所需的代码个数。

比较复杂：

∴在对空间与时间进行权衡时，还可以考虑一种折衷的办法，即部分展开循环。如上例展为：

```
DO    I = 1   TO   30   STEP   3
      A[ I ] = 0.0
      A[ I + 1 ] = 0.0
      A[ I + 2 ] = 0.0
END;
```

空间只多二条，但省了20次测试时间（只循环10次）

c) 归纳变量的优化和条件判断的替换

归纳变量(induction variable): 在每一次执行循环迭代的过程中，若某变量的值固定增加（或减少）一个常量值，则称该变量为归纳变量(induction variable)。即若当前执行循环的第 j 次迭代，归纳变量的值应为c * j + c′, 这里 c 和 c′ 都是循环不变式。

例:
```
for  i := 1  to  10  do
    a[i] := b[i] + c[i]
```

```
for  i:= 1  to  10  do
    a[i] := b[i]  +  c[i]
```

```
1)  i := 1
2)  labb:
3)  if  i > 10  goto  labe
4)  t1 := 4 * i
5)  t2 := b [ t1 ]
6)  t3 := 4 * i
7)  t4 := c [ t3 ]
8)  t5 := t2 + t4
9)  t6 := 4 * i
10) a [ t6 ] := t5
11)  i := i + 1
12) goto  labb
13) labe:
```

优化

```
1)  u := 4
2)  labb:
3)  if  u >40  goto  labe
4)  tb := b [ u ]
5)  tc := c [ u ]
6)  t  :=  tb + tc
7)  a[ u ] := t
8)  u := u + 4
9)  goto  labb
10) labe:
```

中间变量t_1，t_3，t_6都是归纳变量。

$t_1 := 4 * i$，　$t_3 := 4 * i$，　$t_6 := 4 * i$

d) 其它循环优化方法

- 把多重嵌套的循环变成单层循环。
- 把 n 个相同形式的循环合成一个循环等。

对于循环优化的效果是很明显的。

in_line 展开

把过程（或函数）调用改为in_line展开可节省许多处理过程（函数）调用所花费的开销。

如：
```
procedure m( i , j : integer;   max : integer)
   begin   if   i > j   then   max := i   else   max := j   end;
```

若有过程调用 m (k , 0, max);

则内置展开后为:

```
if   k > 0   then   max := k   else   max := 0;
```

省去了函数调用时参数压栈，保存返回地址等指令。这也仅仅限于简单的函数。

一种特殊的四元式表达方式：SSA

Single Static Assignment form(SSA form)静态单一赋值形式的 IR 主要特征是每个变量只赋值一次。

SSA的优点： 1）可以简化很多优化的过程；

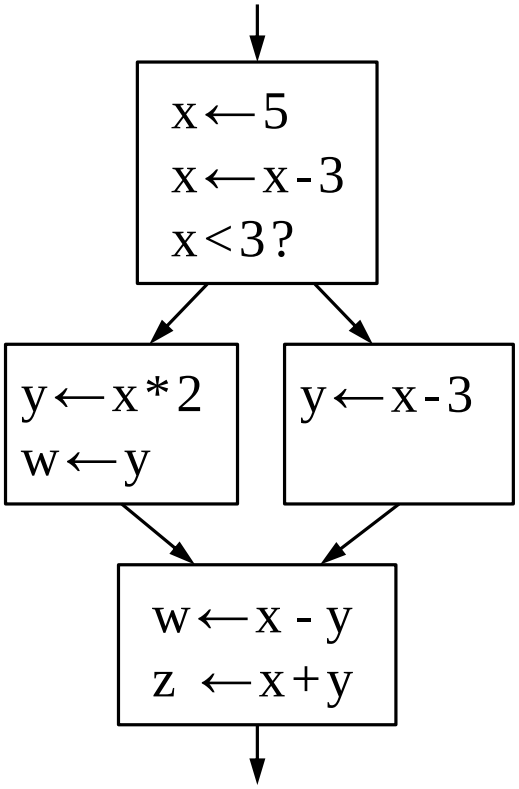
2）可以获得更好的优化结果。

```
y := 1
...
y := 2
x := y + z
```

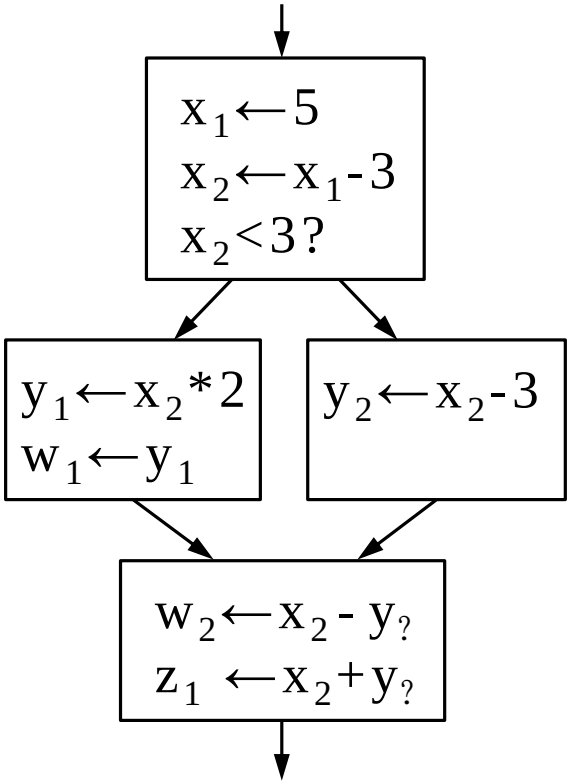
```
y1 := 1
...
y2 := 2
x := y2 + z
```

例，很容易分析出y1是可以优化掉的变量

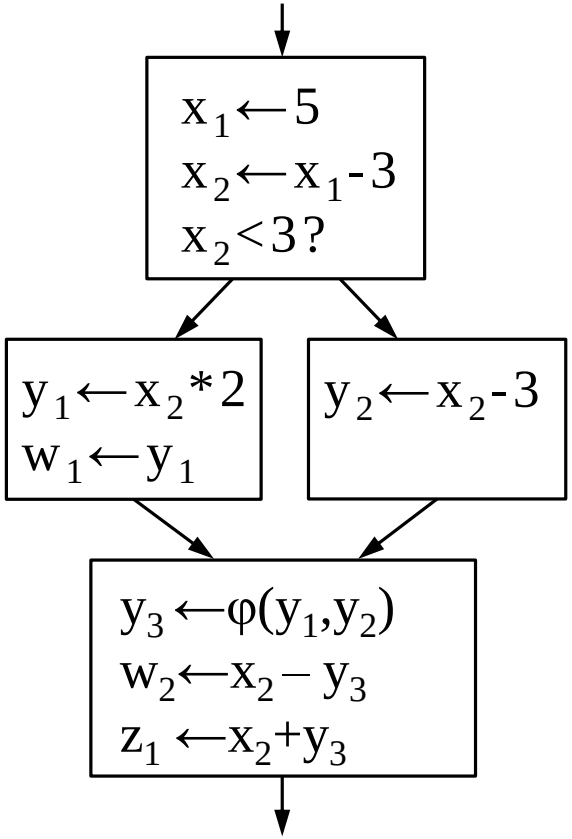
SSA可以从普通的四元式转化而来。如何转化？



原四元式和流图



转换SSA过程中...



加入Φ节点

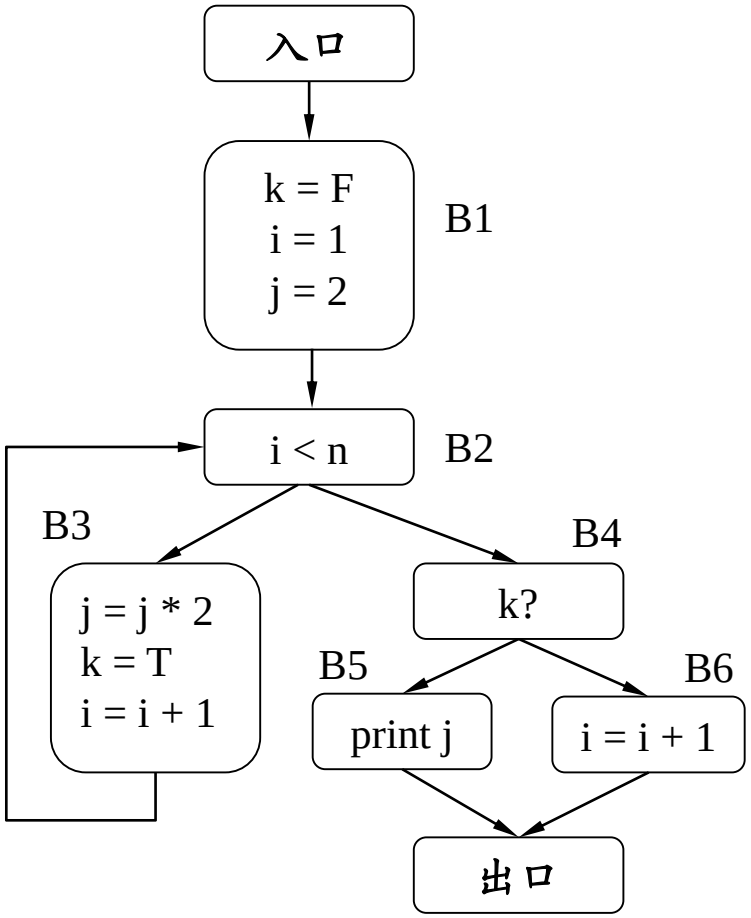
SSA的关键问题——如何加入Φ节点？

Φ节点的参数应包括所有可能到达其位置的同一个变量的所有定义：$x_{n+1}=\Phi(x_1,x_2,x_3,\ldots,x_n)$。

在某个分支汇聚点，如果有同一个变量的多个定义点可能到达，就需要在此处增加相应的Φ节点，汇聚所有可能到达的定义，将其转化为新的定义。

可以采用“最小SSA”的转换方法，生成较少的Φ节点。

“最小SSA”转换方法



定义：基本块节点x的DF(x) =

{ y | 如果z是y的前驱且x支配z，且x不严格支配y }，

严格支配即x dom y & x ≠ y，

$DF^{+}(x) = \lim DF^{i}(x)$, $DF^{1}(x) = DF(x)$,

$DF^{i+1}(x) = DF(S \cup DF^{i}(x))$

例：DF(B3) = {B2}，提示：B_i dom B_i

$DF(B2) = \{B2\}$, $DF^{+}(B2) = \{B2\}$

第1步：将包含某个变量定义点的基本块和入口基本块的DF^{+}节点集合计算出来

第2步：在上述DF^{+}集合中加入变量k的Φ节点

第3步：为所有存在定义点的变量重复步骤1和2

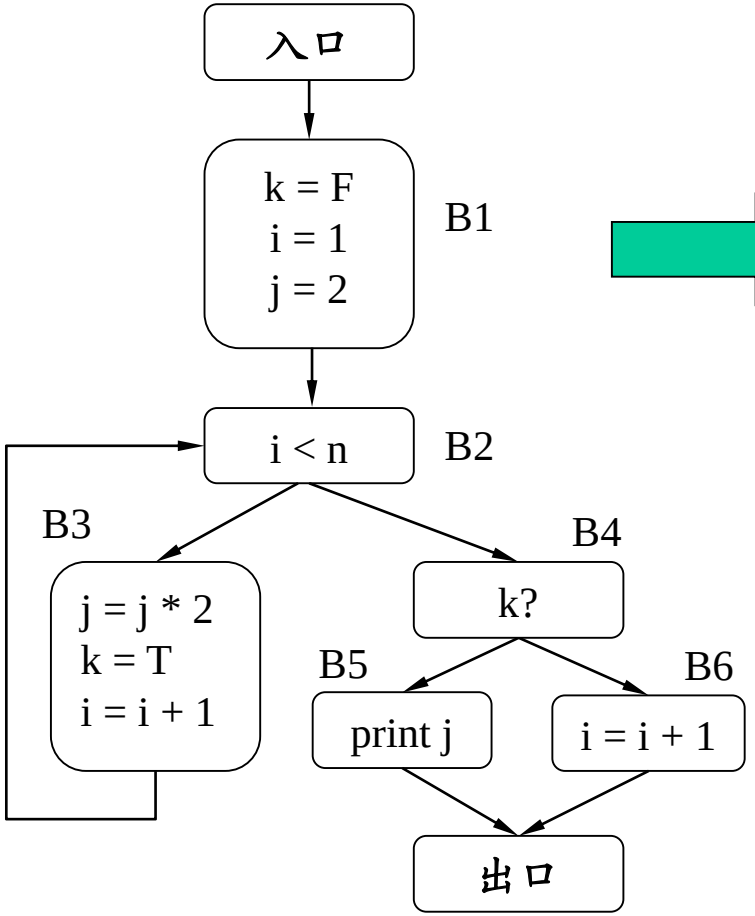
变量k, j：DF^{+}(入口, B1, B3) = {B2}

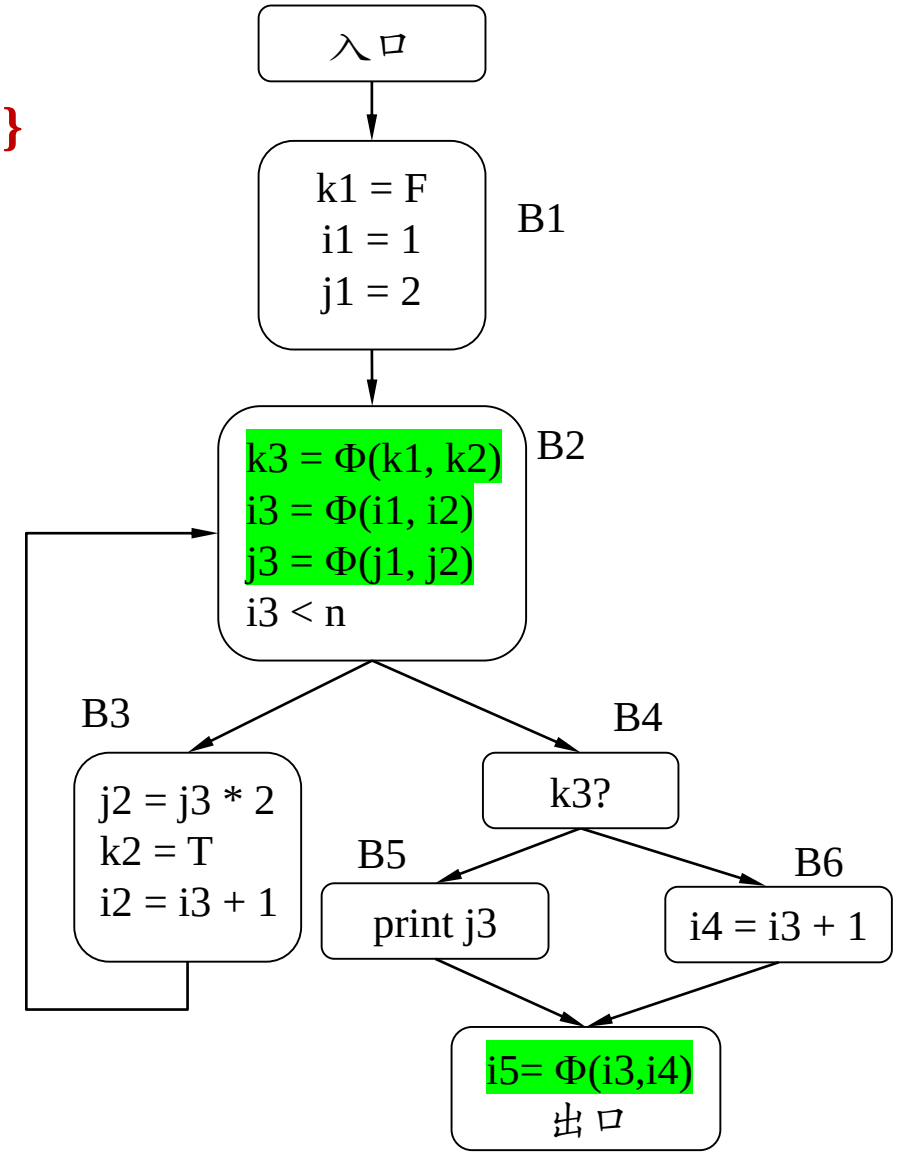
变量i：DF^{+}(入口, B1, B3, B6) = {B2,出口}

“最小SSA”转换方法

变量k, j: DF^+(入口, B1, B3) = {B2}

变量i: DF^+(入口, B1, B3, B6) = {B2, 出口}





第14章作业：1到6题