

## 第十四章 代码优化

- 概述
- 优化的基本方法和例子
- 基本块和流图
- 基本块内的优化
- 全局优化

## 概述

### 代码优化 (code optimization)

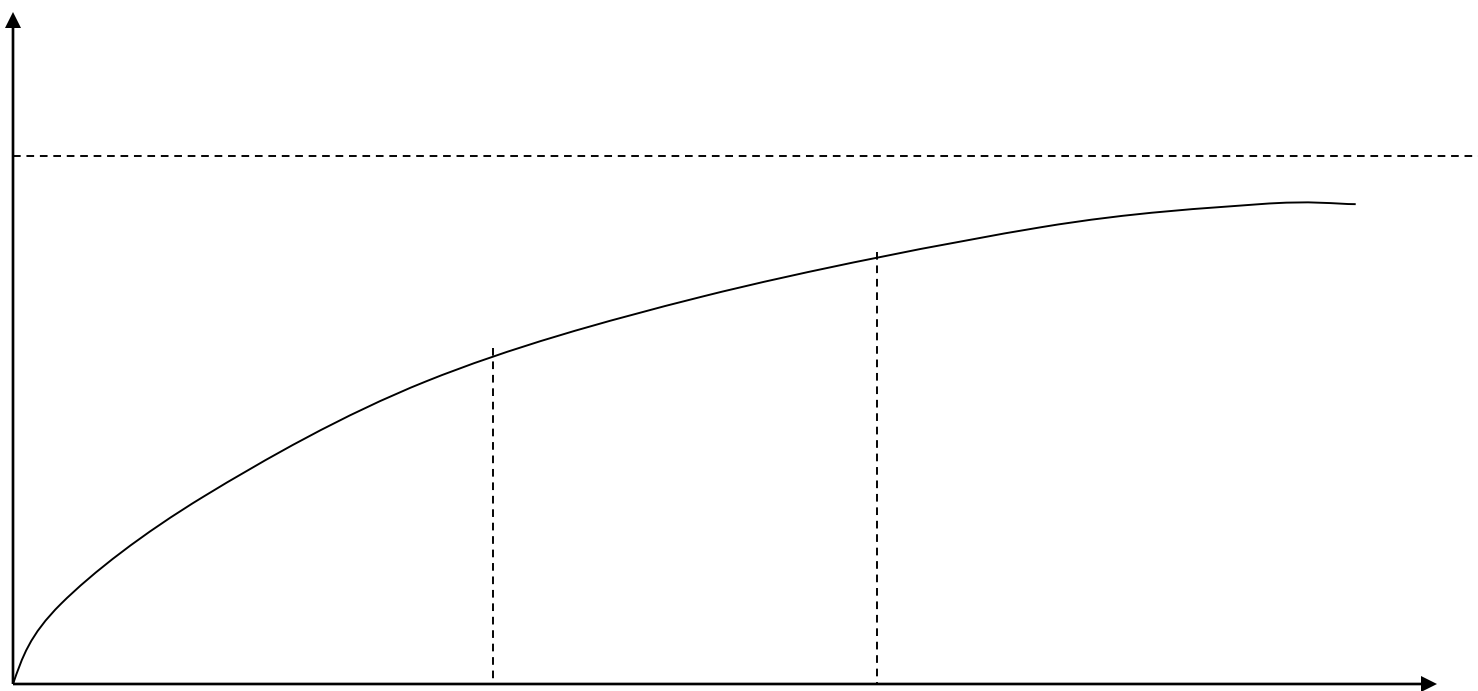
指编译程序为了生成高质量的目标程序而做的各种加工和处理。

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：

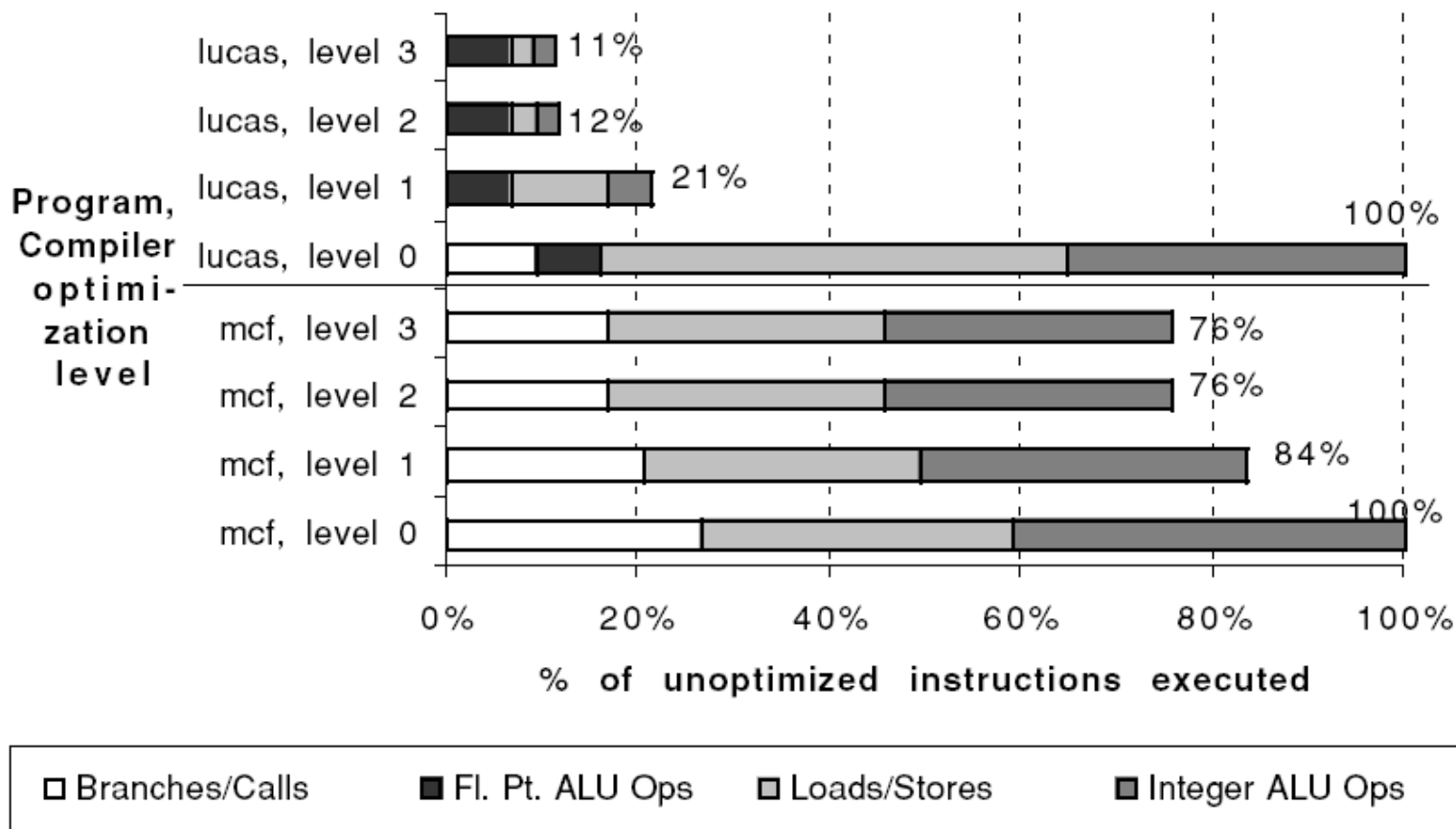
优化的  
效果



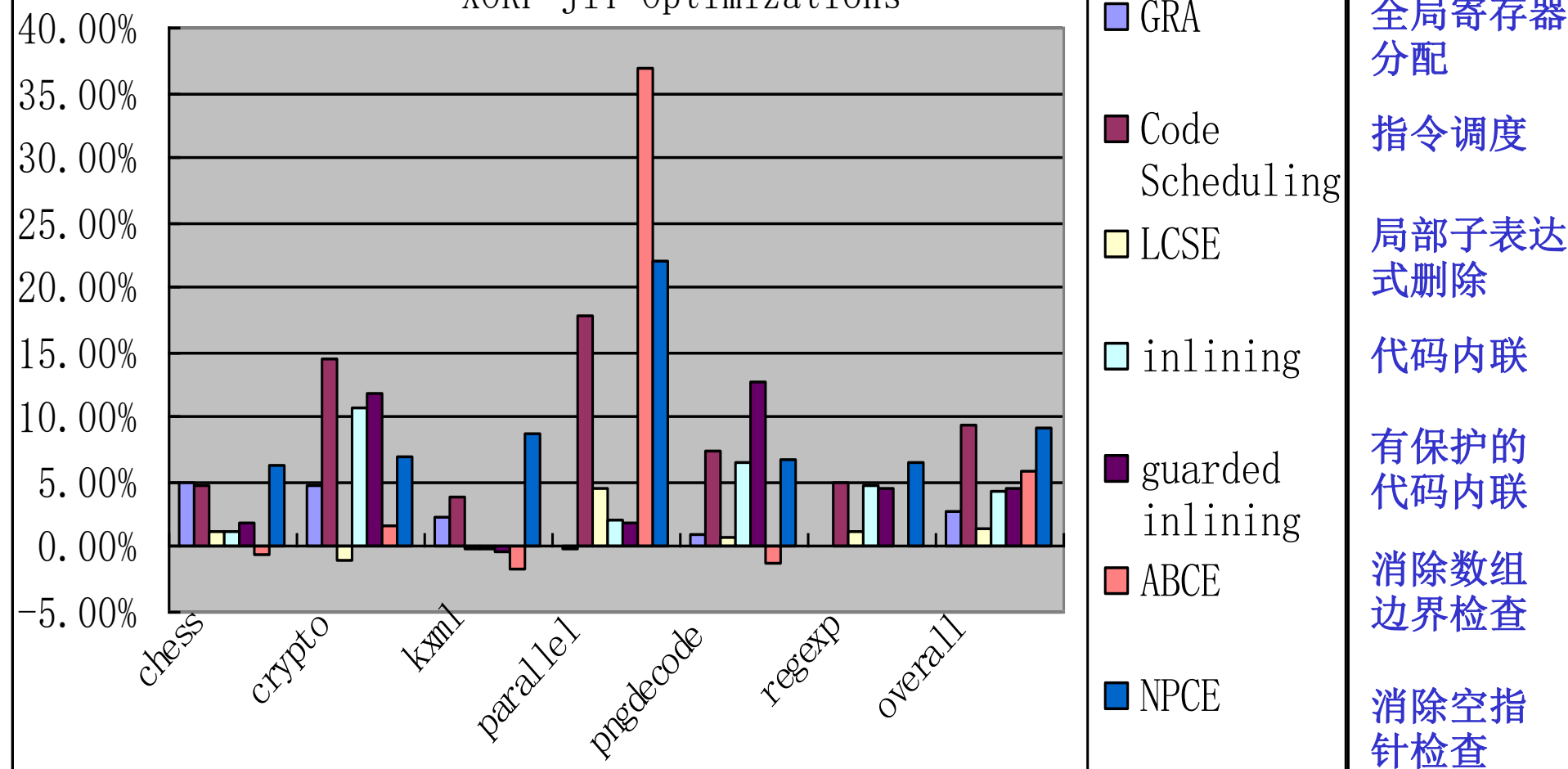
优化的  
代价

•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。 3

## 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）



XORP JIT Optimizations



GRA

全局寄存器分配

Code Scheduling

指令调度

LCSE

局部子表达式删除

inlining

代码内联

guarded inlining

有保护的代码内联

ABCE

消除数组边界检查

NPCE

消除空指针检查

## 为什么要优化？

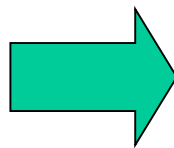
- 有的大型计算程序一运行就要花上几十分钟，甚至几小时、几天、几十天，这时为优化即使付出些代价是值得的。
- 程序中的循环往往要占用大量的计算时间。所以为减少循环执行时间所进行的优化对减少整个程序的运行时间有很大的意义。——尤其有实时要求的程序。
- 至于（像学生作业之类的）简单小程序（占机器内存，运行速度均可接受），或在程序的调试阶段，花费许多代价去进行一遍又一遍的优化就毫无必要了。（尤其机器速度、存储的快速发展。）

## 下面的优化合理吗？

```
int foo(int a)
{
    int count = 0 ;

    for(int i = 1 ; i<=100 ; i++){
        count += i ;
    }

    return a + count ;
}
```



```
int foo(int a)
{
    return a + 5050 ;
}
```

## 优化方法的分类1:

- **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2:

- **局部优化技术**

- 指在**基本块内**进行的优化
- 例如，局部公共子表达式删除

- **全局优化技术**

- **函数/过程内**进行的优化
- 跨越基本块
- 例如，全局数据流分析

- **跨函数优化技术**

- 整个程序
- 例如，跨函数别名分析，逃逸分析 等

# 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)   ...
```

# 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

## 基本块定义

- 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

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

## 基本块的例子：划分基本块

- (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) ...



下列语句序列，哪些属于同一个基本块，哪些不属于？

(1) ~ (6)

(3) ~ (8)

(7) ~ (13)

## 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

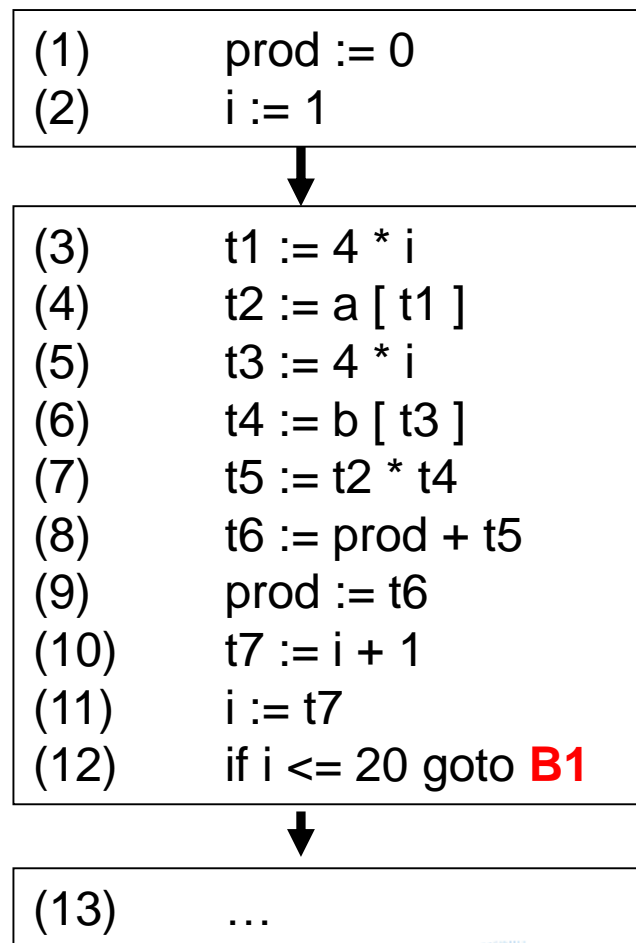
```
(1)    prod := 0
(2)    i := 1
(3)    t1 := 4 * i
(4)    t2 := a [ t1 ]
(5)    t3 := 4 * i
(6)    t4 := b [ t3 ]
(7)    t5 := t2 * t4
(8)    t6 := prod + t5
(9)    prod := t6
(10)   t7 := i + 1
(11)   i := t7
(12)   if i <= 20 goto (3)
(13)   ...
```

- 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)
  - 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  
(3)
  - 1.3 紧跟在跳转语句之后的第一条语句属于入口语句  
(13)
- 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：
  - (1) ~ (2)
  - (3) ~ (12)
  - (13) ~...

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)   ...
```





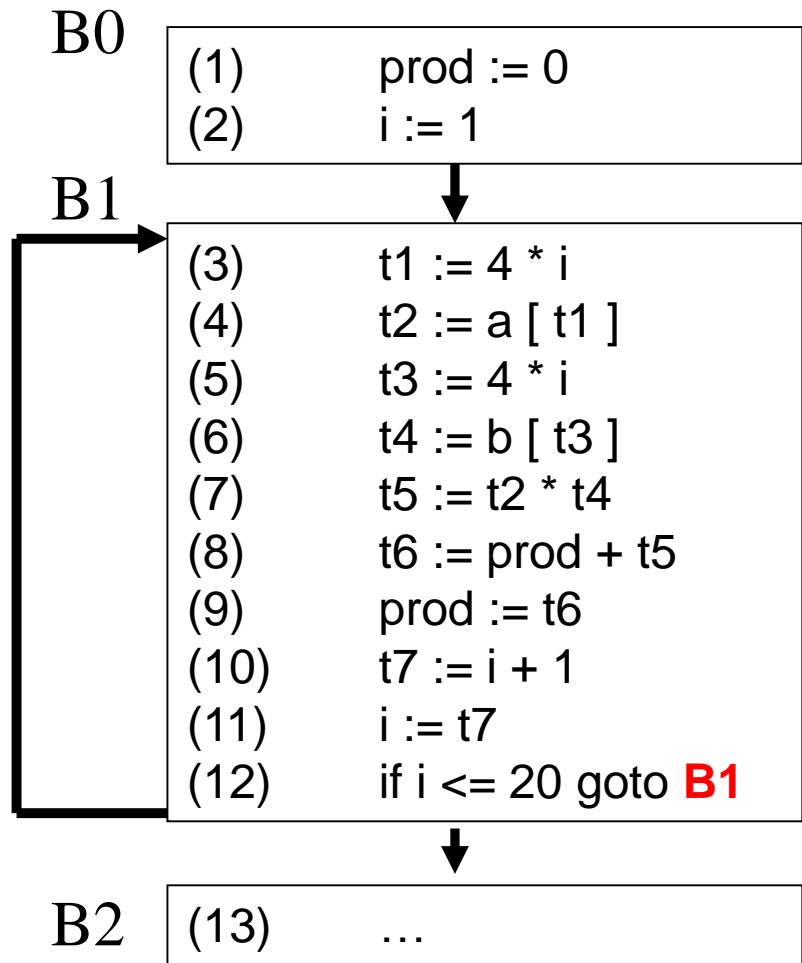
# 流图

- 流图是一种有向图
- 流图的节点是基本块
- 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) ...

```



- 编译器按照“程序—流图----基本块—中间代码”，选择合理的数据结构组织和管理中间代码。

## 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： $a := 5+6+x \rightarrow a := 11+x$  （常数合并）

$PI=3.141592$

$TO-R=PI/180.0 \rightarrow TO-R = 0.0174644$

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

$$x^{**}2 \rightarrow x*x$$

$$3*x \rightarrow x+x+x$$

$8*x$  ,  $4*x$  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

$$x/5 \rightarrow x*0.2 \quad \text{等}$$

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

## (2) 常数合并和传播

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：

$x:=y ;$		$x:=y ;$
$u:=2*x ;$	$\rightarrow$	$u:=2*y ;$
$v:=x+1 ;$		$v:=y+1 ;$

这就是所谓的复写传播。(copy propagation)

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：  $pi = 3.14159$

$r = 0.0174644$

（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:  $FLAG := TRUE$

$IF \quad FLAG \quad THEN...$

$...$

$ELSE...$

}  $FLAG$ 永真

另外在程序中为了调试常有如下:

$if \quad debug \quad then \quad ... \text{ 的语句。}$

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  $\#if \quad DEBUG$  编写程序, 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：  $a = b * (-c) + b * (-c)$

$t1 := -c$

$t2 := b * t1$

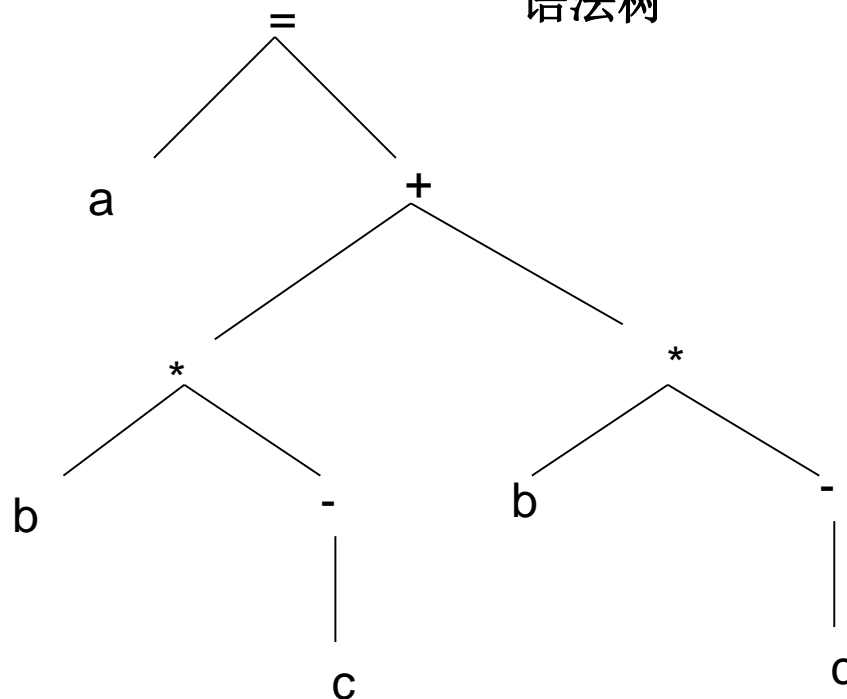
$t3 := -c$

$t4 := b * t3$

$t5 := t2 + t4$

$a := t5$

语法树



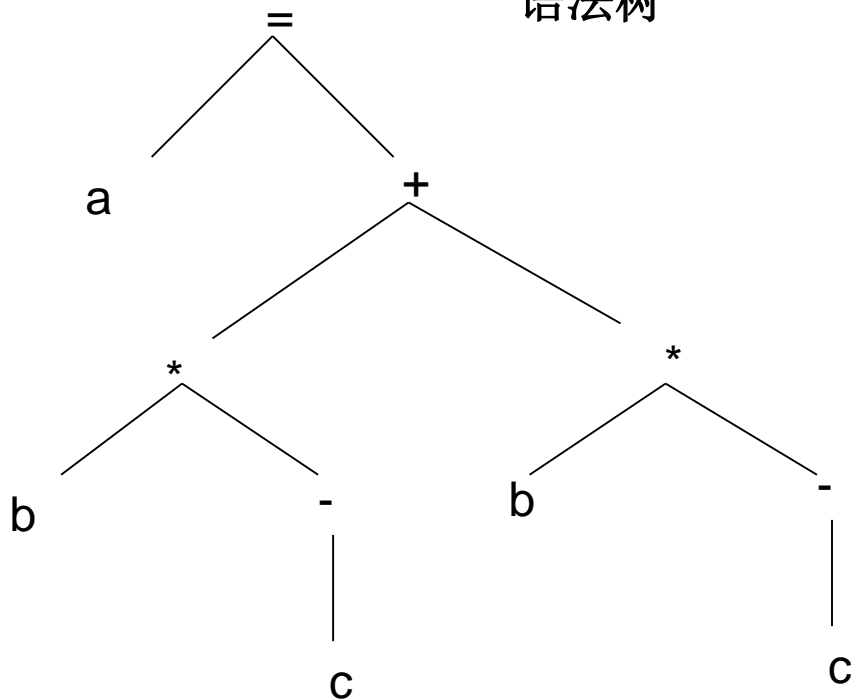
# 基本块内优化: 消除公共子表达式

- 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

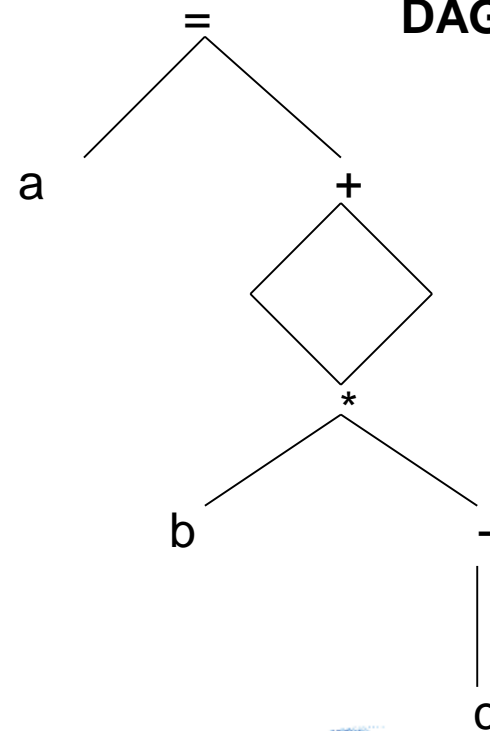
## 14.2.1 基本块的DAG图表示

- 赋值语句:  $a = b * (-c) + b * (-c)$

语法树

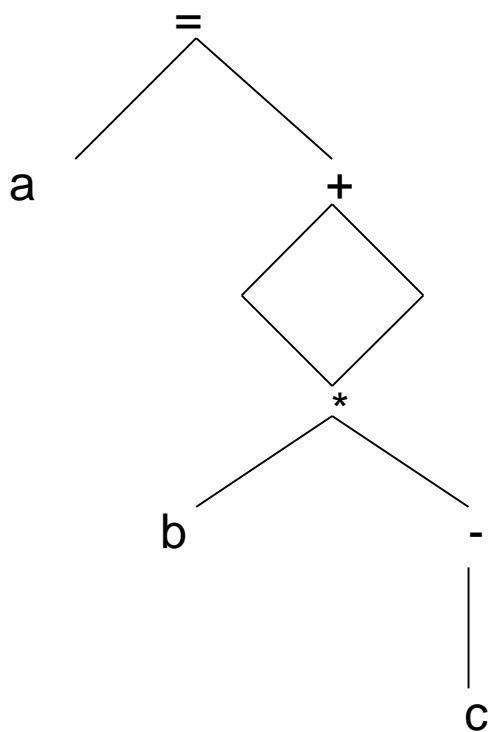


DAG图



# 基本块DAG图的定义

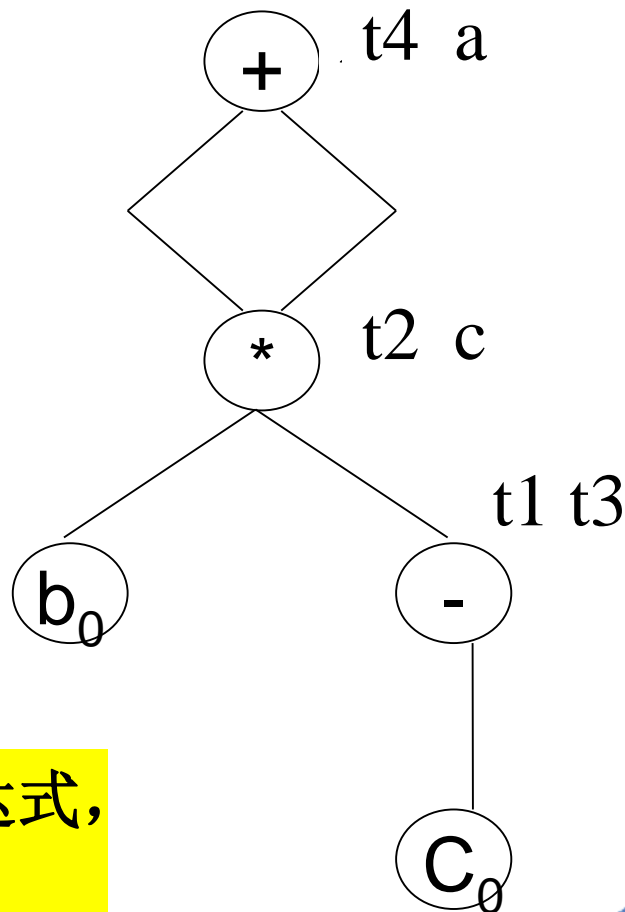
DAG图



- 图的叶节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。
- 图的中间节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节点；具有初值的变量，其初值和最终值可以分别对应不同的节点。

# DAG表示

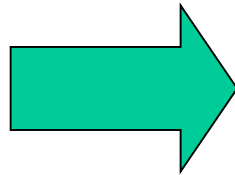
- (1)  $t1 = -c$
- (2)  $t2 = b * t1$
- (3)  $t3 = -c$
- (4)  $c = b * t3$
- (5)  $t4 = t2 + c$
- (6)  $a = t4$



通过DAG图可消除公共子表达式，  
得到更简洁的优化代码

## 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
```

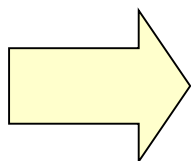


```
t1 := - c
t2 := b * t1
t3 := - c
t4 := b * t3
t5 := t2 + t2 (t4)
a := t5
```

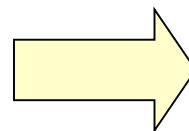
*c := c + 1 ?*

## 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. 首先建立节点表，该表记录了变量名和常量值，以及它们当前所对应的DAG图中节点的序号。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 $z = x \text{ op } y$ 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， $\text{op}$ 为操作符：首先在节点表中寻找 $x$ ，如果找到，记录下 $x$ 当前所对应的节点号 $i$ ；如果未找到，在DAG图中新建一个叶节点，假设其节点号仍为 $i$ ，标记为 $x$ （如 $x$ 为变量名，该标记更改为 $x_0$ ）；在节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，假设其对应节点号为 $j$ 。



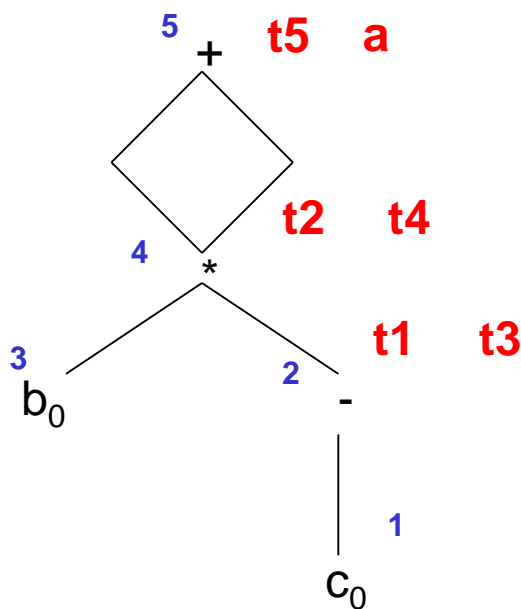
## 算法14.2 通过构建DAG图消除局部公共子表达式（续）

4. 在DAG图中寻找中间节点，其标记为 $op$ ，且其左操作数节点号为 $i$ ，右操作数节点号为 $j$ 。如果找到，记录下其节点号 $k$ ；如果未找到，在DAG图中新建一个中间节点，假设其节点号仍为 $k$ ，并将节点 $i$ 和 $j$ 分别与 $k$ 相连，作为其左子节点和右子节点；
5. 在节点表中寻找 $z$ ，如果找到，将 $z$ 所对应的节点号更改为 $k$ ；如果未找到，在节点表中新建一项 $(z, k)$ ，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。

## 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
    
```



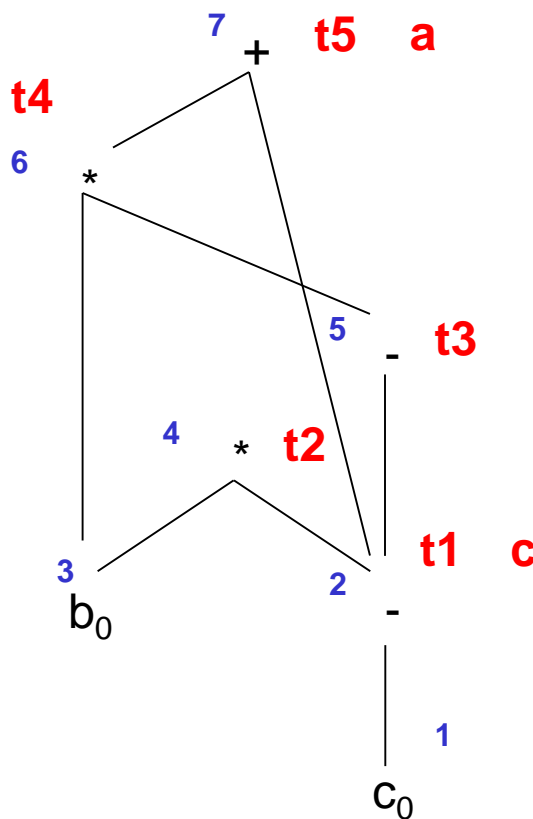
*node(x)*

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

## 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： $x = a[i]$

$a[j] = y$

$z = a[i]$

$x = z ?$

不一定。  
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 $x = a \text{ [] } i$ ，其中 $[]$ 为数组取值操作符；形如 $a[j] = y$ 的中间代码都表示为 $a = j \text{ []=} y$ ，其中 $[]=$ 为数组成员赋值操作符。

## 指针：保守处理

$x = *p$

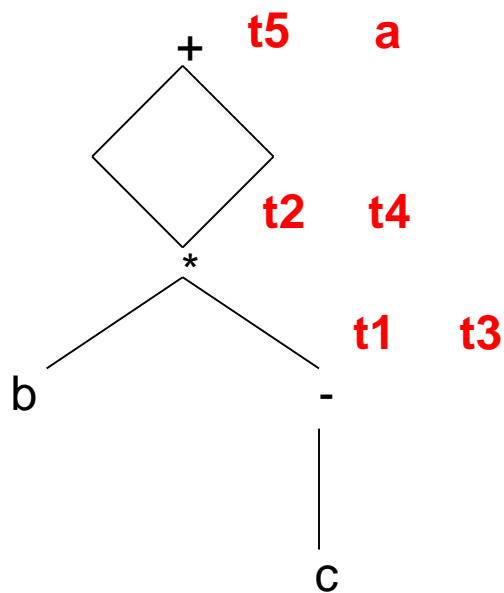
$*q = y$

$z = *p$

- 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

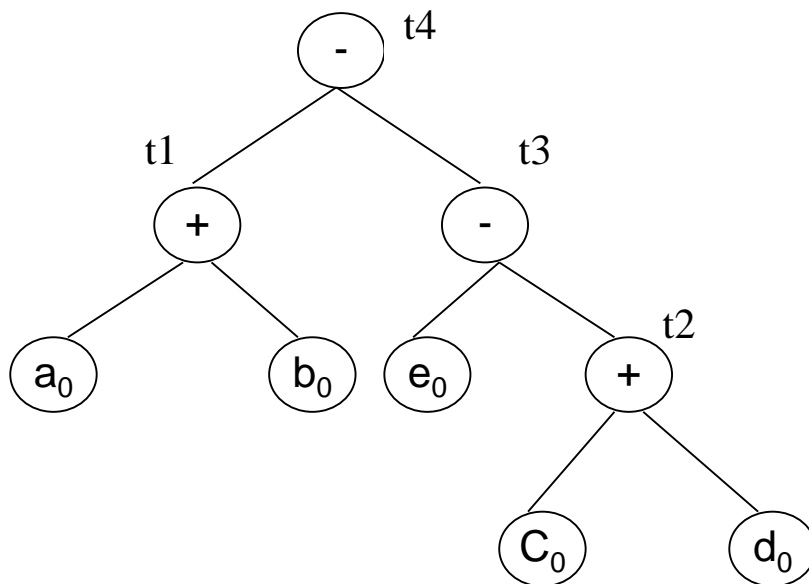


$t1 := -c$

$t2 := b * t1$

$a := t2 + t2$

## 从DAG图重新导出中间代码



$$\begin{aligned}
 (1) \quad & t1 = a + b \\
 & t2 = c + d \\
 & t3 = e - t2 \\
 & t4 = t1 - t3
 \end{aligned}$$

$$\begin{aligned}
 (2) \quad & t2 = c + d \\
 & t3 = e - t2 \\
 & t1 = a + b \\
 & t4 = t1 - t3
 \end{aligned}$$

(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 ; t3 = e - t2`

`mov eax, e`

`sub eax, edx`

`mov [ESP+0CH], edx ; t4 = t1 - t3`

`mov edx, [ESP+08H]`

`sub edx, eax`

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**



(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** ;  $t3 = e - t2$

mov eax, e

sub eax, edx

~~**mov [ESP+0CH], edx**~~ ;  $t4 = t1 - t3$

**mov edx, [ESP+08H]**

sub edx, eax

(2)  $t2 = c + d$

$t3 = e - t2$

$t1 = a + b$

$t4 = t1 - t3$

mov eax, c ;  $t2 = c + d$

add eax, d

mov edx, e ;  $t3 = e - t2$

sub edx, eax

~~**mov [ESP+0CH], eax**~~ ;  $t1 = a + b$

mov eax, a

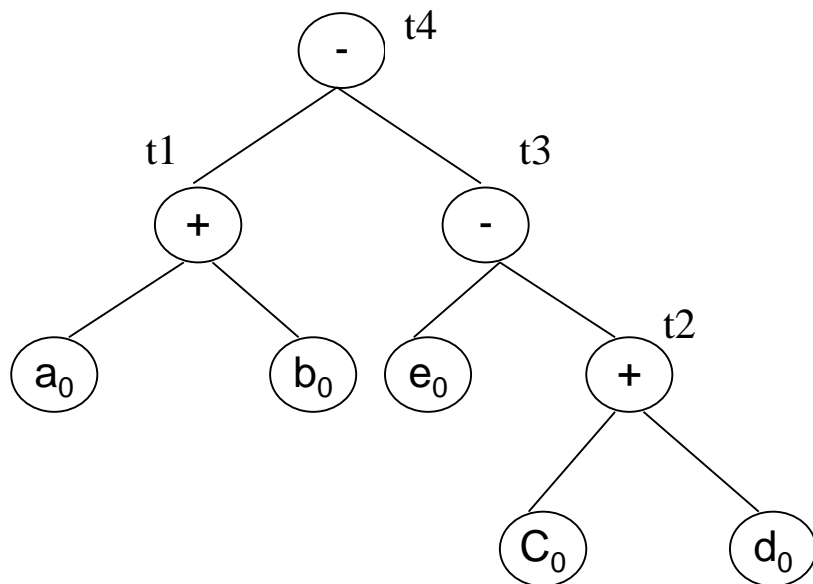
add eax, b

~~**mov [ESP+08H], eax**~~ ;  $t4 = t1 - t3$

sub eax, edx

## 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图中还有中间节点未进入队列，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**所有父节点均已进入队列**的中间节点n，将其加入队列；或选取**没有父节点**的中间节点，将其加入队列
  4. 如果n的**最左子节点**符合步骤3的条件，将其加入队列；并沿着当前节点的最左边，**循环访问其最左子节点**，最左子节点的最左子节点等，将符合步骤3条件的中间节点依次加入队列；如果出现不符合步骤3条件的最左子节点，执行步骤2
  5. 将中间节点队列**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



- 1、初始化一个放置DAG图中间节点的队列
- 2、如果DAG图中还有中间节点未进入队列，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**所有父节点均已进入队列**的中间节点n，将其加入队列；或选取**没有父节点的中间节点**，将其加入队列
- 4、如果n的最左子节点符合步骤3的条件，将其加入队列；并沿着当前节点的最左边，循环访问其**最左子节点**，最左子节点的最左子节点等，将符合步骤3条件的中间节点依次加入队列；如果出现不符合步骤3条件的最左子节点，执行步骤2。
- 5、将中间节点队列逆序输出，便得到中间节点的计算顺序，将其整理成中间代码序列

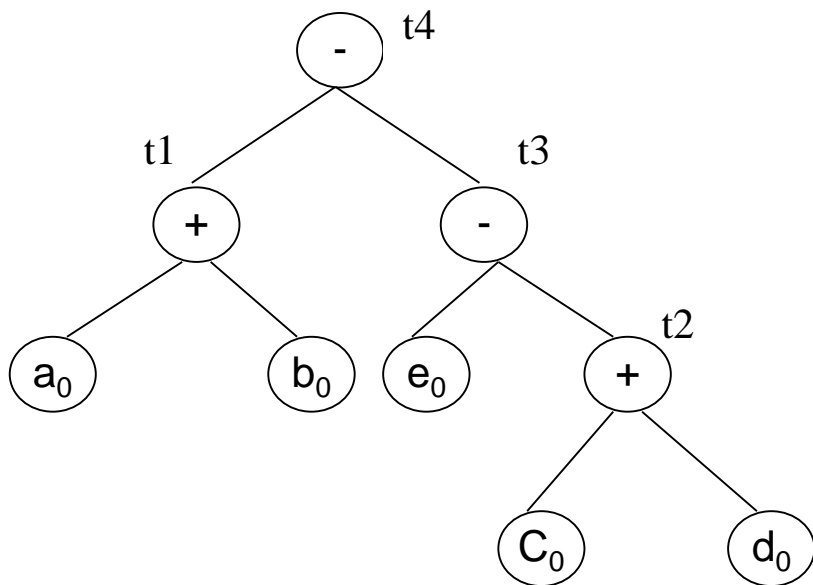
中间节点队列：

t4	t1	t3	t2
----	----	----	----

中间节点队列:

t4	t1	t3	t2
----	----	----	----

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



$$t2 = c + d$$

$$t3 = e - t2$$

$$t1 = a + b$$

$$t4 = t1 - t3$$

## 14.2.5 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”
- 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

但是窥孔优化并不局限在同一个基本块中。以下方法，如利用代数性质等都可以用。

```
mov EAX, [ESP+08H]  
mov [ESP+08H], EAX
```

```
jmp B2  
B2: ...
```

条件是：必须出现在同一个基本块中。

## 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

- 考察在程序的某个执行点的数据流信息。
- $\text{out}[S] = \text{gen}[S] \cup (\text{in}[S] - \text{kill}[S])$ 
  - 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）
  - $\text{out}[S]$ 代表在该语句**末尾得到**的数据流信息
  - $\text{gen}[S]$ 代表该语句**本身产生**的数据流信息
  - $\text{in}[S]$ 代表**进入**该语句时的数据流信息
  - $\text{kill}[S]$ 代表该语句**注销**的数据流信息

# 数据流分析方程

- 考察在程序的某个执行点的数据流信息。
- $out[S] = gen[S] \cup (in[S] - kill[S])$ 
  - S代表某条语句（可以是基本块，或者语句集合，或者基本块等）
  - out[S]代表在这点得到的数据流信息
  - gen[S]代表该语句本身产生的数据流信息
  - in[S]代表进入该语句时的数据流信息
  - kill[S]代表该语句注销的数据流信息

含义：当执行控制流通过某一条语句时，在该语句**末尾得到的数据流信息**等于该语句**本身产生的数据流信息**，**合并进入**该语句时的数据流信息**减去**该语句**注销的数据流信息**后的数据流信息。



# 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 $\text{in}[S]$ 定义 $\text{out}[S]$ ，也可以反向定义，由 $\text{out}[S]$ 定义 $\text{in}[S]$
- 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。
- 一种常用的数据流分析方法：达到定义。

# 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

## 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

- 对于基本块中的某一条中间代码：

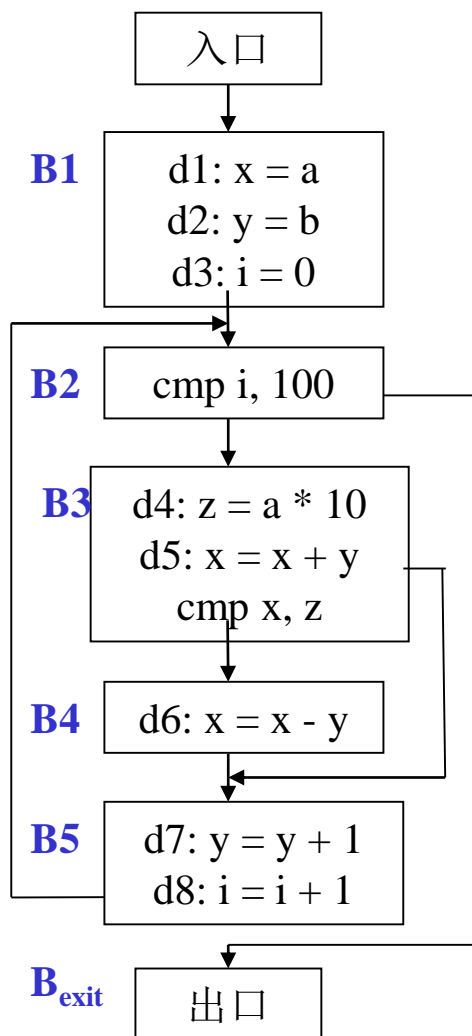
d1:  $u = v \text{ op } w$ ,  $v$ 和 $w$ 为变量,  $\text{op}$ 为操作符

- 代码对应的到达定义数据流方程是：

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

- 其中

- $\text{gen}[d1] = \{d1\}$ , 表明该语句产生了一个定义点（定义了变量 $u$ ）
- $\text{kill}[d1]$  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 $d2$ ,  $\text{in}[d2]$ 等价于 $\text{out}[d1]$



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

$\text{gen}[B2] = \{ \}$   
 $\text{kill}[B2] = \{ \}$

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

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

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

**d1~d8 八个定义点**

## 基本块B的到达定义数据流方程

- $\text{out}[B] = \text{gen}[B] \cup (\text{in}[B] - \text{kill}[B])$ 
  - $\text{in}[B]$ 为进入基本块时的数据流信息
  - $\text{kill}[B] = \text{kill}[d1] \cup \text{kill}[d2] \dots \cup \text{kill}[dn]$ ,  $d1 \sim dn$ 依次为基本块中的语句
  - $\text{gen}[B] = \text{gen}[dn] \cup (\text{gen}[d(n-1)] - \text{kill}[dn]) \cup (\text{gen}[d(n-2)] - \text{kill}[d(n-1)] - \text{kill}[dn]) \dots \cup (\text{gen}[d1] - \text{kill}[d2] - \text{kill}[d3] \dots - \text{kill}[dn])$

## 例:

- $out[B] = gen[B] \cup (in[B] - kill[B])$ 
  - $in[B]$ 为进入基本块时的数据流信息
  - $kill[B] = kill[d1] \cup kill[d2] \dots \cup kill[dn]$ ,  $d1 \sim dn$ 依次为基本块中的语句
  - $gen[B] = gen[dn] \cup (gen[d(n-1)] - kill[dn]) \cup (gen[d(n-2)] - kill[d(n-1)] - kill[dn]) \dots \cup (gen[d1] - kill[d2] - kill[d3] \dots - kill[dn])$

d1:  $a = b + 1$

d2:  $a = b + 2$

$$kill[B] = kill[d1] \cup kill[d2] = \{d2\} \cup \{d1\} = \{d1, d2\}$$

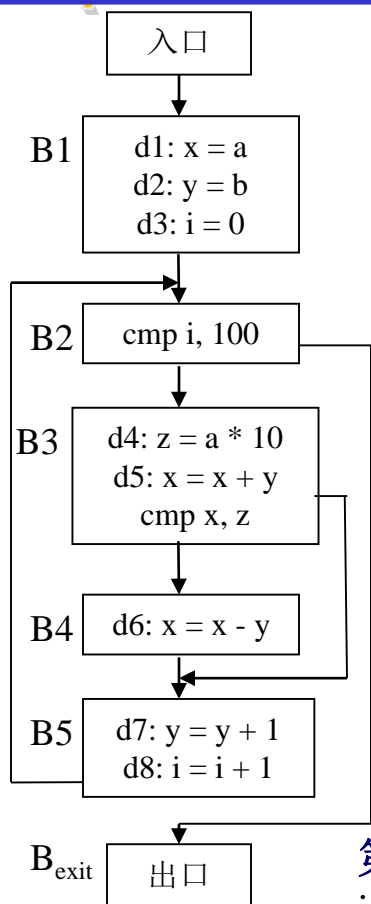
$$gen[B] = gen[d2] \cup (gen[d1] - kill[d2]) = \{d2\} \cup (\{d1\} - \{d1\}) = \{d2\}$$

$$out[B] = gen[B] \cup (in[B] - kill[B]) = \{d2\} \cup (in[B] - \{d1, d2\})$$



## 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 $B_{\text{exit}}$ 的所有基本块的out集合，**初始化**为空集。
  2. 根据方程 $\mathbf{in[B] = \bigcup_{B \text{的前驱基本块} P} out[P]}$ ， $out[B] = gen[B] \cup (in[B] - kill[B])$ ，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。



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}

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

### 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}

**B\_exit的前驱为B2**, in[B\_exit] = {d1, 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 “ $\cup$ ” 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， $\text{out}[B3] = \{d2, d3, d4, d5, d7, d8\}$ ，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 “ $\cup$ ” 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（取反加一）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)

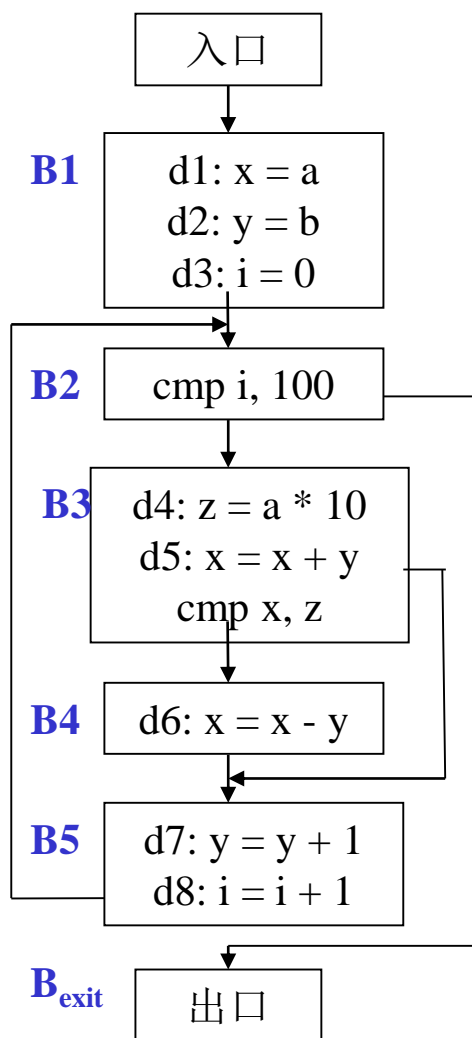
- 达到定义分析是沿着流图路径的，有的数据流分析是方向计算的
- 活跃变量分析：
  - 了解变量 $x$ 在某个执行点 $p$ 是活跃的
    - 变量 $x$ 的值在 $p$ 点或沿着从 $p$ 出发的某条路径中会被使用，则称 $x$ 在 $p$ 点是活跃的。
  - 通过活跃变量分析，可以了解到某个变量 $x$ 在程序的某个点上是否活跃，或者从该点出发的某条路径上是否会被使用。如果存在被使用的可能， $x$ 在该程序点上便是活跃的，否则就是非活跃，或者死的。

# 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：
  - $\text{in}[B] = \text{use}[B] \cup (\text{out}[B] - \text{def}[B])$
  - $\text{out}[B] = \bigcup_{B \text{ 的后继基本块 } P} \text{in}[P]$
  - $\text{def}[B]$  : 变量在B中被定义（赋值）先于任何对它们的使用
  - $\text{use}[B]$  : 变量在B中被使用先于任何对它们的定义
- 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



- $\text{def}[B]$  : 变量在B中被定义（赋值）**先于**任何对它们的使用
- $\text{use}[B]$  : 变量在B中被使用**先于**任何对它们的定义

$\text{use}[B3] = \{a, x, y\}$   
 $\text{def}[B3] = \{z\}$

**d1~d8 八个定义点**

活跃变量分析:

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

到达定义分析:

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

- 采用 $\text{use}[B]$ 代表当前基本块新生成的数据流信息（用了）
- 采用 $\text{def}[B]$ 代表当前基本块消除的数据流信息（定义的）
- 采用 $\text{in}[B]$ 而不是 $\text{out}[B]$ 来计算当前基本块中的数据流信息
- 采用 $\text{out}[B]$ 而不是 $\text{in}[B]$ 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



## def和use

- $\text{def}[B]$ 指的是在基本块 $B$ 中，在**使用前**被定义的变量集合
  - 在引用改变量前已经明确地对该变量进行了赋值
- $\text{use}[B]$ 指的是在基本块 $B$ 中，在**定义前**被使用的变量集合
  - 在该变量的任何定义之前对其引用

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

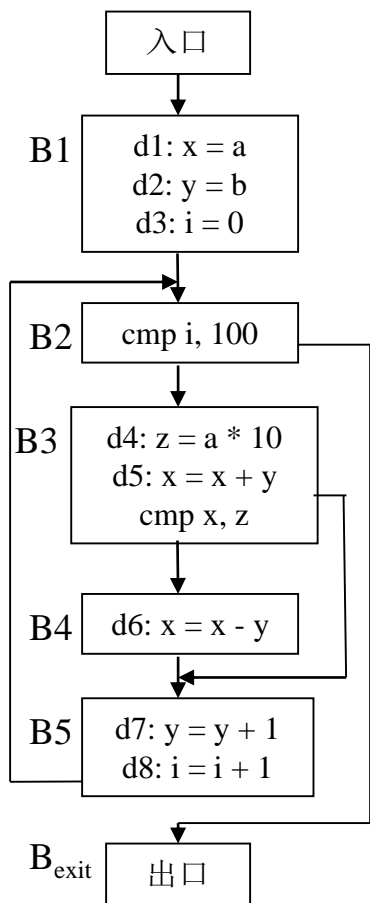
- 输入：程序流图，且基本块的use集合和def集合已经计算完毕
- 输出：每个基本块入口和出口处的in和out集合，即in[B]和out[B]
- 方法：
  1. 将包括代表流图出口基本块 $B_{\text{exit}}$ 在内的所有基本块的in集合，初始化为空集。
  2. 根据方程 $\text{out}[B] = \bigcup_{B \text{ 的后继基本块 } P} \text{in}[P]$ ， $\text{in}[B] = \text{use}[B] \cup (\text{out}[B] - \text{def}[B])$ ，为每个基本块B依次计算集合out[B]和in[B]。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] =  $\emptyset$  ;
while 集合in发生变化 do
    for 每个基本块B do begin
        out[B] =  $\cup_{B \text{ 的所有后继 } S} \text{in}[S]$ 
        in[B] = use[B]  $\cup$  (out[B] – def[B])
    end
end
    
```

## 例

流图



for 每个基本块B do in[B] =  $\emptyset$  ;

while 集合in发生变化 do

for 每个基本块B do begin

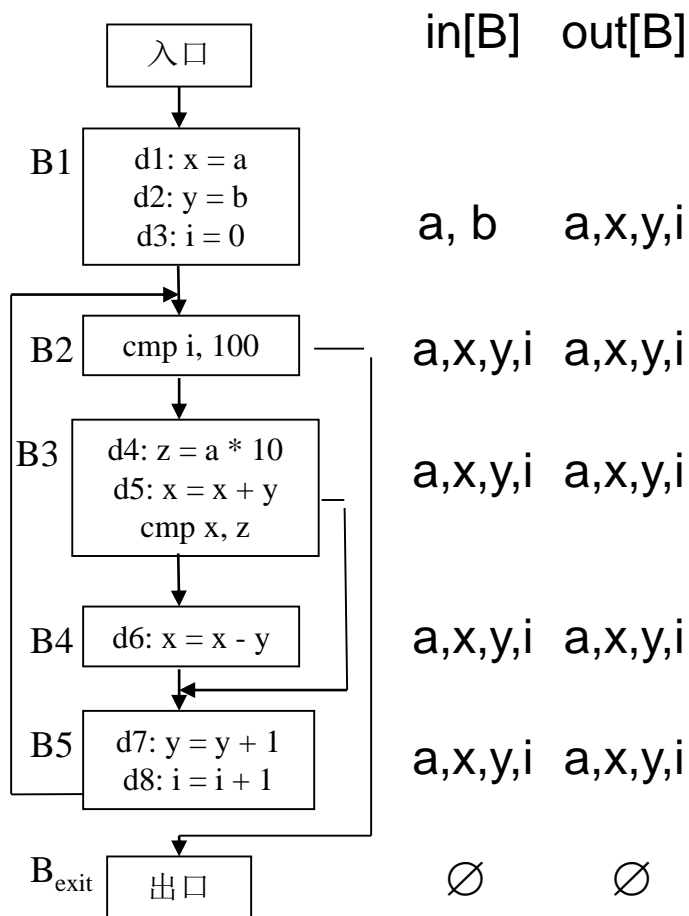
out[B] =  $\cup_{B \text{ 的所有后继 } S} \text{in}[S]$

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

end

def[B]	use[B]	in[B]	out[B]	in[B]	out[B]	in[B]	out[B]
x, y, i	a, b	a, b	a,x,y,i	a, b	a,x,y,i	a, b	a,x,y,i
$\emptyset$	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
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
$\emptyset$	x, y	x, y, i	y, i	a,x,y,i	a,x,y,i	a,x,y,i	a,x,y,i
$\emptyset$	y, i	y, i	$\emptyset$	a,x,y,i	a,x,y,i	a,x,y,i	a,x,y,i
$\emptyset$		$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

流图



- 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, 因此其在使用之前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。

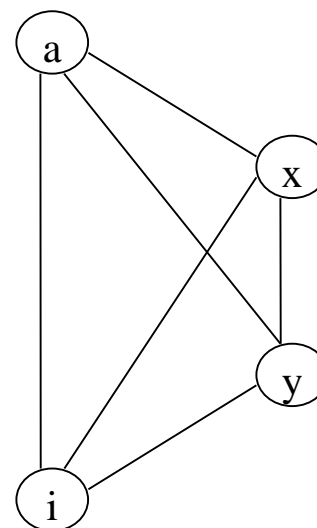
- 变量b: 在流图中无定义点, 在B1入口处活跃。

- 变量z: 定义于B3, 且仅在B3中被使用。

# 冲突图

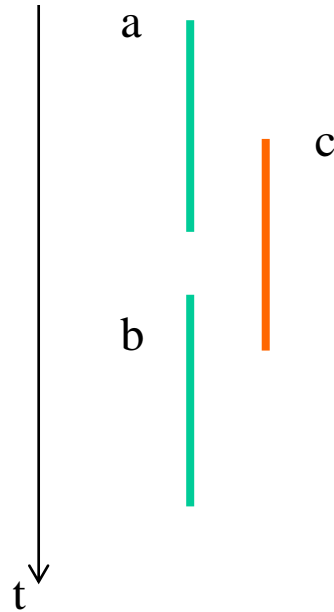
- 假设只有跨越基本块活跃的变量才能分配到全局寄存器，并且**活跃范围重合**的变量之间无法共享全局寄存器

	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 <sub>exit</sub>	∅	∅

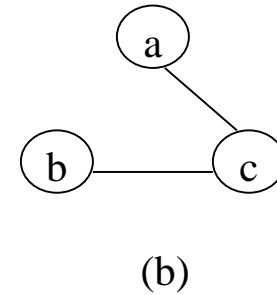


### 14.3.3 定义-使用链、网和冲突图

- **冲突图：**其节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。



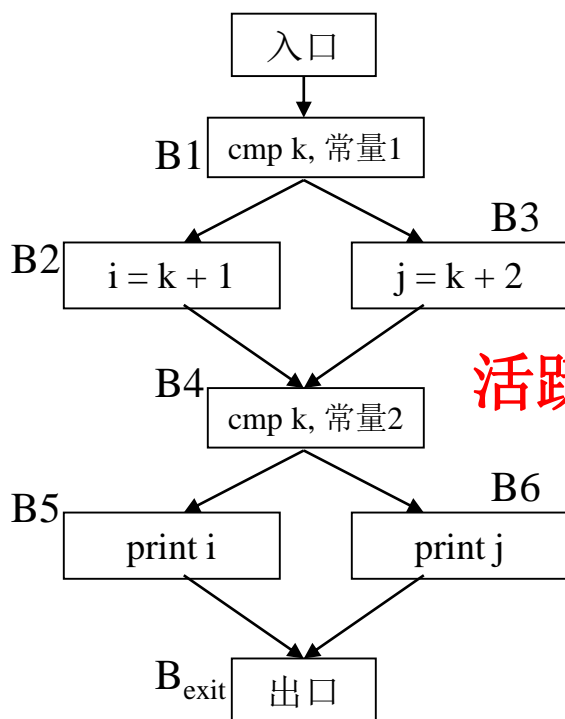
- 变量a和变量b不冲突  
可以着色



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。

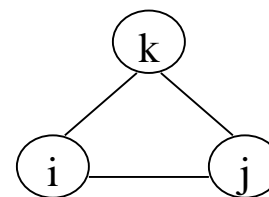




活跃变量: **i, j, k**

(a)

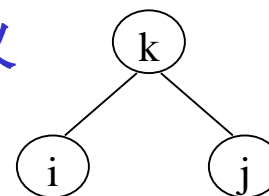
- 变量*i*和变量*j*在B2和B3中被分别定义，并在B5和B6中被分别使用。根据活跃变量分析结果，*i*和*j*一定同时在B4的入口处活跃



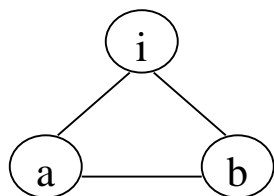
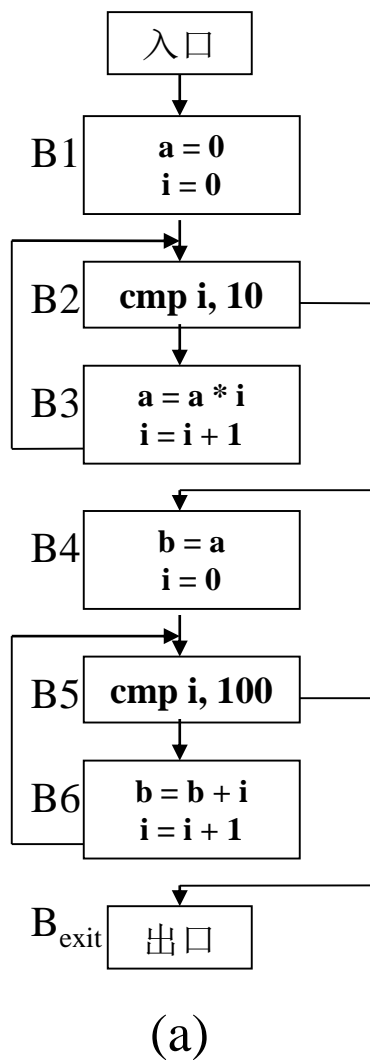
(b)

但即使*i*和*j*使用同一寄存器，程序运行结果仍符合语义

冲突图中两个节点（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

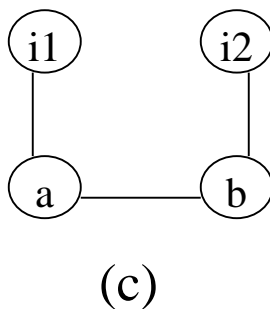


(c)



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

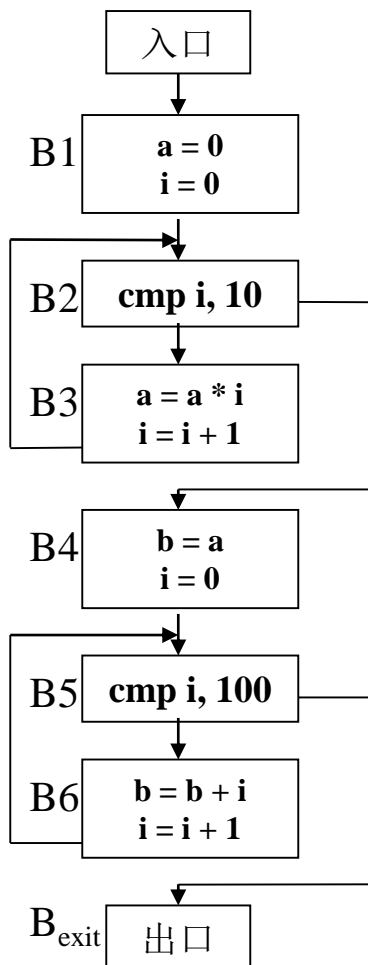
(三色)



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？

- 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

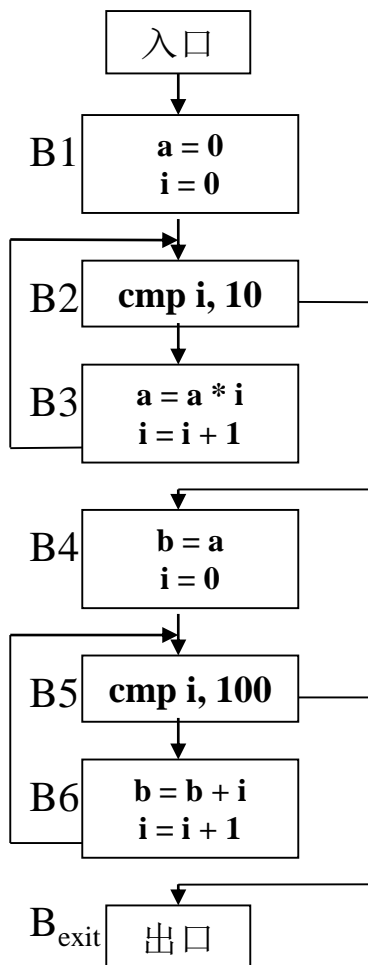


变量a: L1 {<B1, 1>, <B3, 1>, <B4, 1>}  
L2 {<B3, 1>, <B3, 1>, <B4, 1>}

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>}

- 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



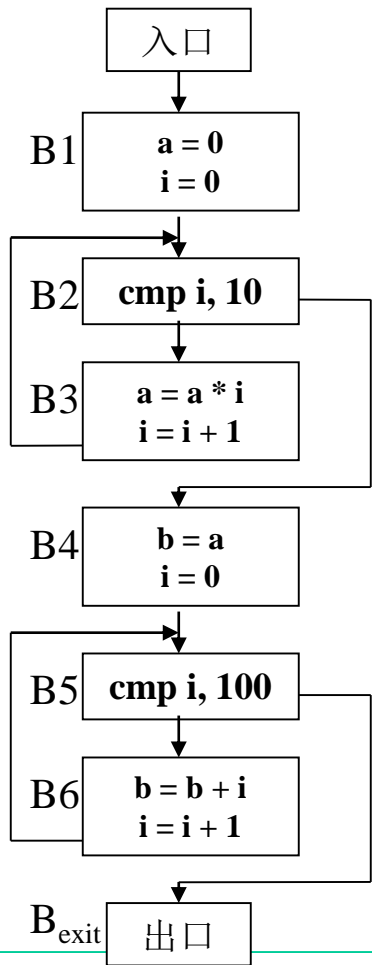
变量a: L1 {<B1, 1>, **<B3, 1>**, <B4, 1>}  
 L2 {<B3, 1>, **<B3, 1>**, <B4, 1>}

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。**  
 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



变量a:

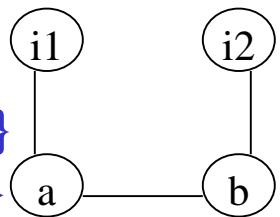
L1 {<B1, 1>, <B3, 1>, <B4, 1>}  
 L2 {<B3, 1>, <B3, 1>, <B4, 1>}

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>}



(c)

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>}}

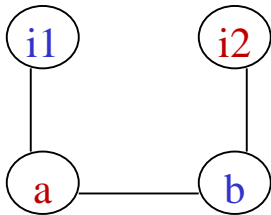
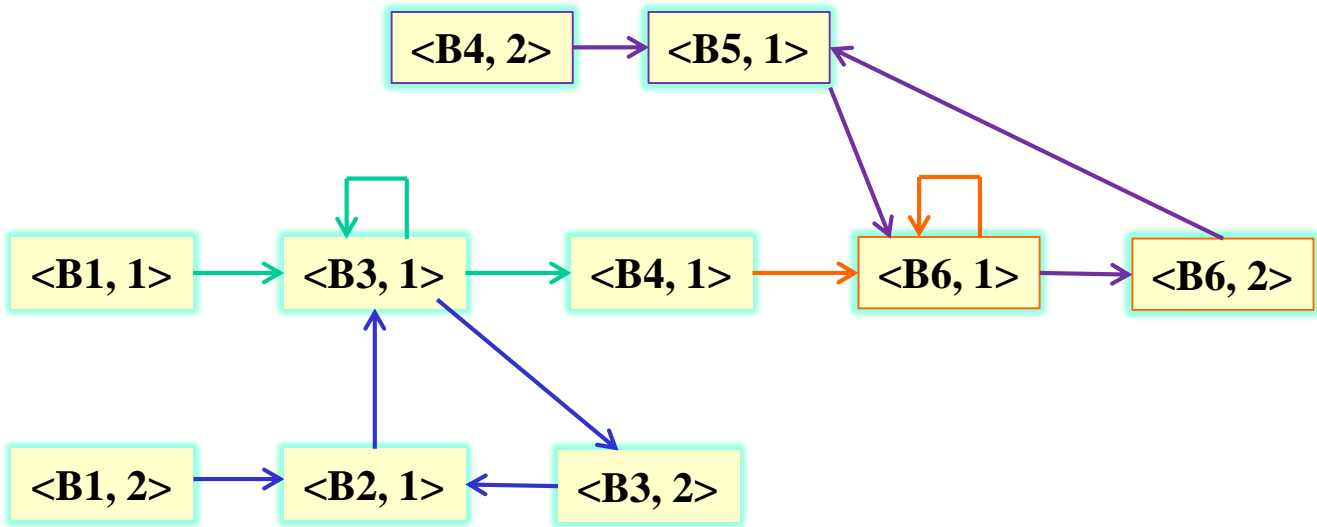
W4 { L7 {<B4, 2>, <B5, 1>, <B6, 1>, <B6, 2>}, L8 {<B6, 2>, <B5, 1>, <B6, 1>, <B6, 2>}}

变量a:  $W1 \{ L1 \{ \langle B1, 1 \rangle, \langle B3, 1 \rangle, \langle B4, 1 \rangle \}, L2 \{ \langle B3, 1 \rangle, \langle B3, 1 \rangle, \langle B4, 1 \rangle \} \}$

变量b:  $W2 \{ L3 \{ \langle B4, 1 \rangle, \langle B6, 1 \rangle \}, L4 \{ \langle B6, 1 \rangle, \langle B6, 1 \rangle \} \}$

变量i:  $W3 \{ L5 \{ \langle B1, 2 \rangle, \langle B2, 1 \rangle, \langle B3, 1 \rangle, \langle B3, 2 \rangle \}, L6 \{ \langle B3, 2 \rangle, \langle B2, 1 \rangle, \langle B3, 1 \rangle, \langle B3, 2 \rangle \} \}$

$W4 \{ L7 \{ \langle B4, 2 \rangle, \langle B5, 1 \rangle, \langle B6, 1 \rangle, \langle B6, 2 \rangle \}, L8 \{ \langle B6, 2 \rangle, \langle B5, 1 \rangle, \langle B6, 1 \rangle, \langle B6, 2 \rangle \} \}$



# 消除全局公共子表达式

- 自学

## 14.4 循环优化

经验规则告诉我们：“程序运行时间的**80%**是由仅占源程序**20%**的部分执行的”。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。因为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<sub>1</sub> STEP E<sub>2</sub> TO E<sub>3</sub> 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的值在该循环中不改变时，则可将循环不变式移到循环之外，即变为：

**t<sub>1</sub> := b\*b - 4.0\*a\*c**

**while    ...    do**

**x:= ...( t<sub>1</sub>) ...**

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），重复产生许多次（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

- 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。
- 必须知道循环的终值，初值及步长。
- 但并不是所有展开都是合适的。如上例中循环展开后节省执行了转移和测试语句： **$2*30=60$ 语句 (其实，还不止节省60条) 。**

∴增加29条省60条

但若循环体中不是一条而是40条语句，则展开后将有 $40*30$ 条=1200，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   BY   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 $c'$ 都循环不变式。

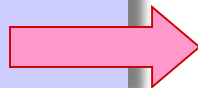
例:        **for      $i := 1$         to     10        do**  
               **$a[i] := b[i] + c[i]$**

```

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:



```

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

```

```

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t1, t3, t6 都是归纳变量

t1 := 4\*i, t3 := 4\*i, t6 := 4\*i

## d) 其它循环优化方法

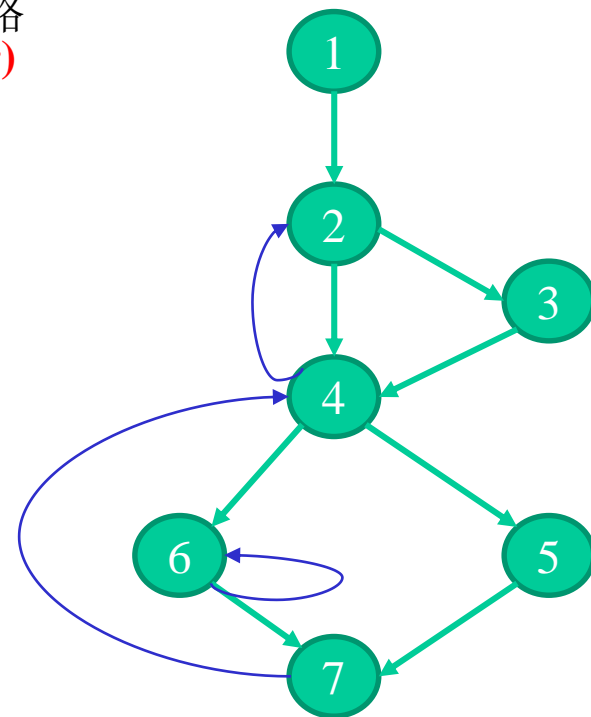
- 把多重嵌套的循环变成单层循环。
- 把 $n$ 个相同形式的循环合成一个循环等。

对于循环优化的效果是很明显的。



# 循环的查找

- 程序流图中，如果从首节点出发，任何到达节点B的路径上都要经过A，那么A就是B的**必经节点(Dominator)**，记为A dom B
  - Dom(1) = {1}
  - Dom(2) = {1,2}
  - Dom(3) = {1,2,3}
  - Dom(4) = {1,2,4}
  - Dom(5) = {1,2,4,5}
  - Dom(6) = {1,2,4,6}
  - Dom(7) = {1,2,4,7}
- 如果有边B→A 且A dom B，该边即为循环的**回边 (Back edge)**，循环体包括能到达B且不经A的所有节点
  - 循环6→6包括: {6}
  - 循环7→4包括: {4,5,6,7}
  - 循环4→2包括: {2,3,4,5,6,7}



## (6) 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。

## 作业： 14章的1到5题