

第十四章 代码优化

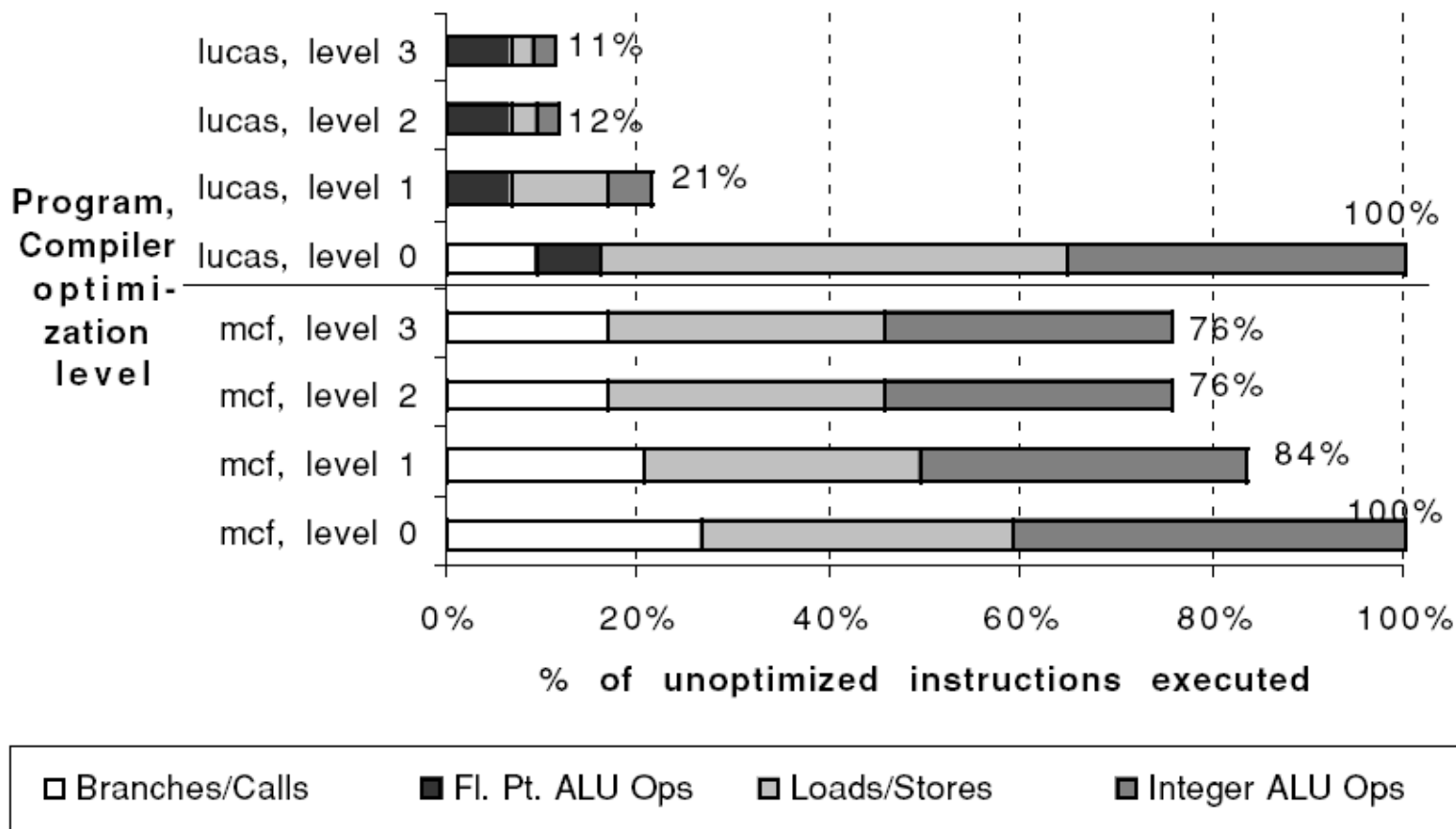
史晓华

北京航空航天大学

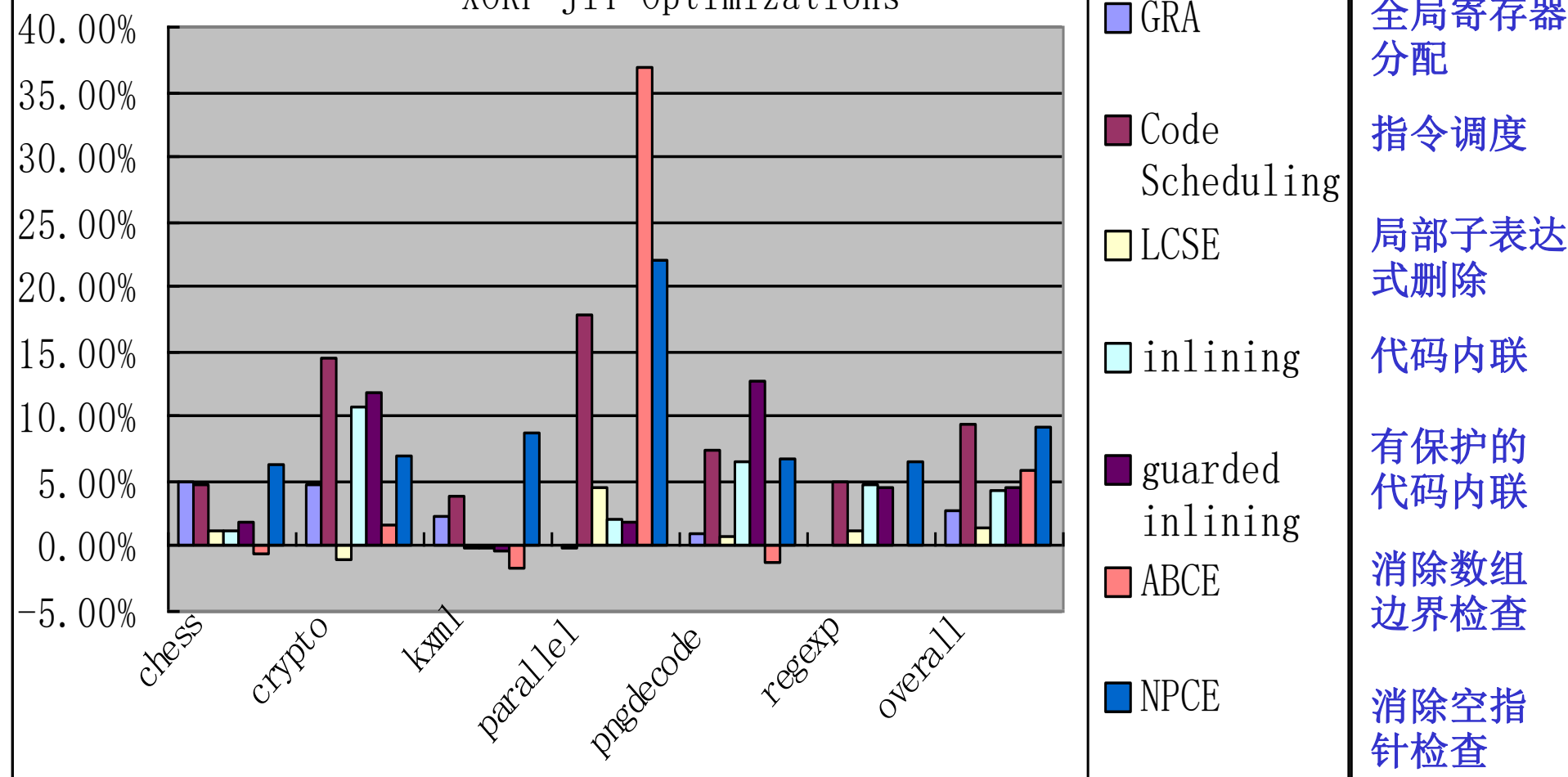
2023.10.16

- 参考书：
 - Computer Architecture: A Quantitative Approach, 3rd version, By John L. Hennessy and David A. Patterson
 - 中文版：计算机体系结构量化研究方法，清华郑维民等译
 - Compilers: Principles, Techniques, and Tools. By Alfred V. AHO, Monica S. Lam, Ravi Sethi and Jeffrey D. ULLMAN
 - 中文版：（龙3）编译原理，赵建华等译，机械工业出版社
 - Advanced Compiler Design and Implementation. By Steven S. Muchnick.
 - 中文版：高级编译器设计与实现，赵克佳，沈志宇译，机械工业出版社

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

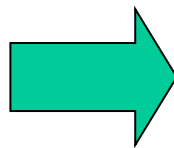


下面的优化合理吗？

```
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
 - 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
 - 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

优化方法的分类2:

- 局部优化技术
 - 基本块内
 - 例如，局部公共子表达式删除
- 全局优化技术
 - 函数/过程内
 - 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

11.1 基本块和流图

- 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

基本块的例子：程序片断

```
(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) ...
```

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

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

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11.1 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
 - 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，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)

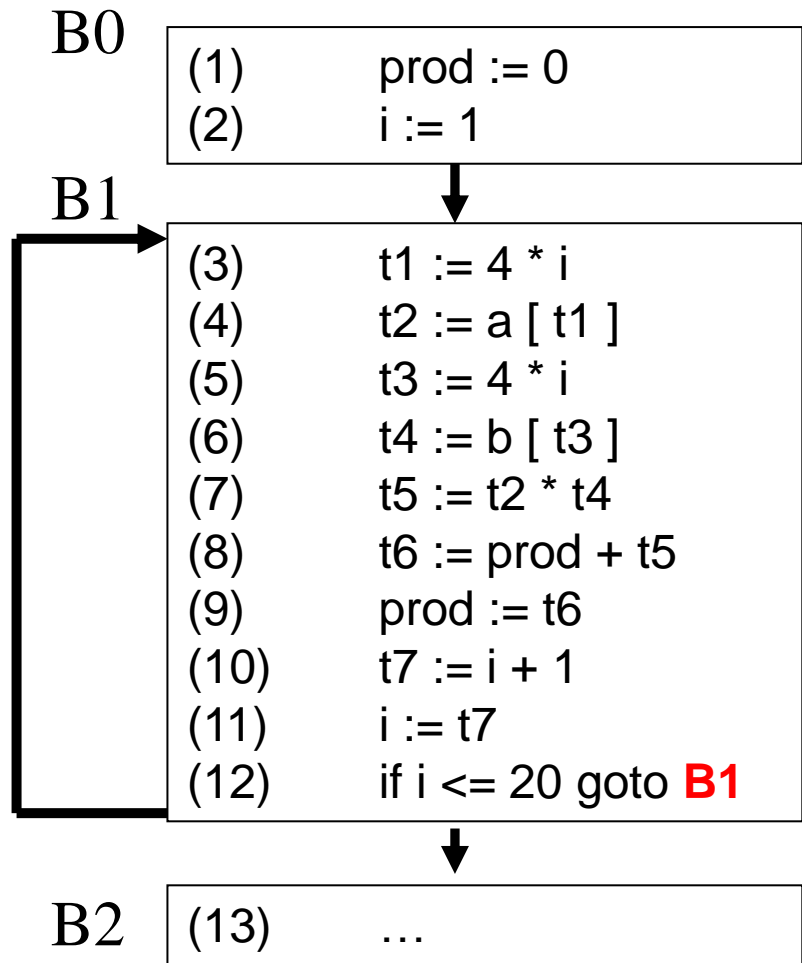
– (3) ~ (12)

– (13) ~...

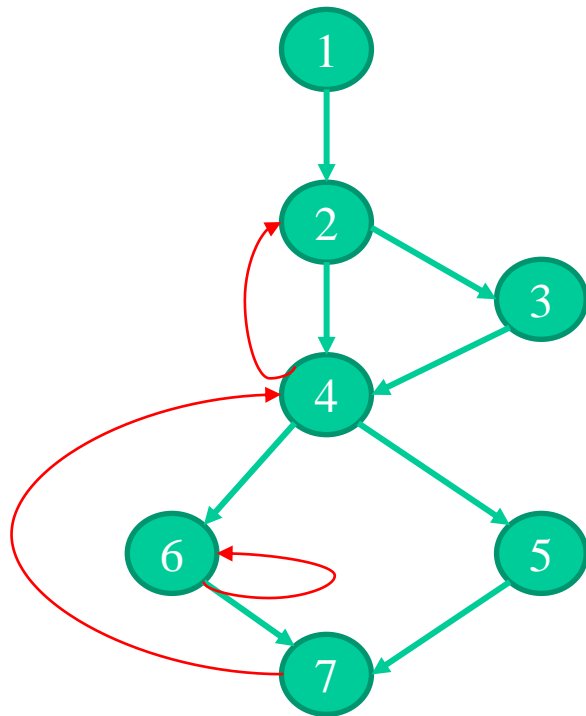
流图

- 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) ...
```



- 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 $A \text{ dom } B$
 - $\text{Dom}(1) = \{1\}$
 - $\text{Dom}(2) = \{1, 2\}$
 - $\text{Dom}(3) = \{1, 2, 3\}$
 - $\text{Dom}(4) = \{1, 2, 4\}$
 - $\text{Dom}(5) = \{1, 2, 4, 5\}$
 - $\text{Dom}(6) = \{1, 2, 4, 6\}$
 - $\text{Dom}(7) = \{1, 2, 4, 7\}$
- 如果有边 $B \rightarrow A$ 且 $A \text{ dom } B$ ，该边即为循环的**回边(Back edge)**，循环体包括能到达B且不经过A的所有节点
 - 循环 $6 \rightarrow 6$ 包括: $\{6\}$
 - 循环 $7 \rightarrow 4$ 包括: $\{4, 5, 6, 7\}$
 - 循环 $4 \rightarrow 2$ 包括: $\{2, 3, 4, 5, 6, 7\}$



优化的基本方法和例子

注：实际的优化应在中间代码或目标代码上进行。但为了便于说明，这里用源程序形式举例。

(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： $a := 5+6+x \rightarrow a := 11+x$

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) 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：

$x:=y ;$		$x:=y ;$
$u:=2*x ;$	\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 为临时变量，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)

(4) 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:

$if \quad debug \quad then \quad ...$ 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)

(5) 循环优化

经验规则告诉我们：“程序运行时间的**80~90%**是由仅占源程序**10~20%**的部分执行的”。这**10~20%**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 := E₁ STEP E₂ TO E₃ 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：

t₁ := b*b - 4.0*a*c

while ... do

x:= ...(t₁) ...

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

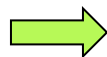
- 循环一次执行5条语句才给一个变量赋初值。展开后，一条语句就能赋一个值，运行效率高。
- 必须知道循环的终值，初值及步长。
- 但并不是所有展开都是合适的。如上例中循环展开后节省执行了转移和测试语句： **$2*30=60$ 语句 (其实，还不止节省60条)**。

∴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，就未必合算了。

∴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
(高性能计算)

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。某FORTRAN 77 编译程序，在进行不同级别的优化后所得的目标代码指令数为：

优化级别	循环内的指令数（包括循环条件判断）
0（不优化）	21
1	16
2	6
3	5

(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。

(7) 其他，如控制流方法

如 **BR** **L** 无条件转移
 ... ———— 为不可达代码

L:

又如：转移到转移指令的指令

BR	L1		BR	L2
...				
L1: BR	L2		L1: BR	L2

还有:

BR_{CC} L1

当条件CC成立，转到L1

BR L2

L1:

可改进为:

BR' _{CC} L2

当条件不能成立时，转到L2

(L1:) ...

11.2.5 窥孔优化

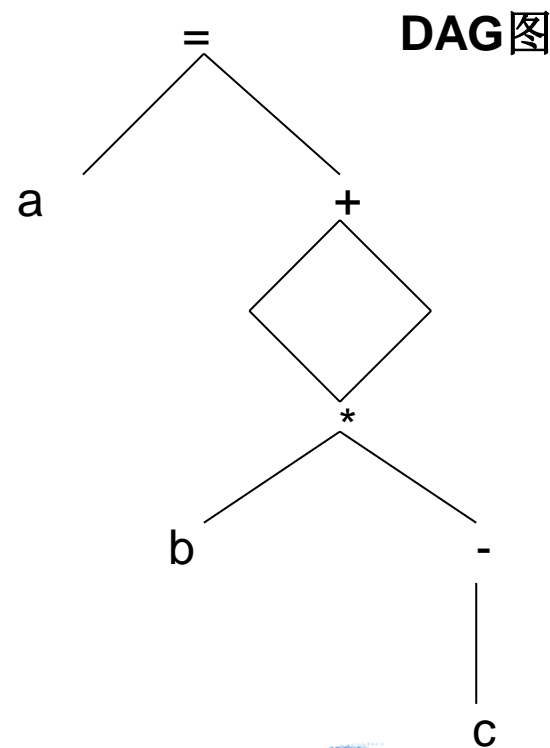
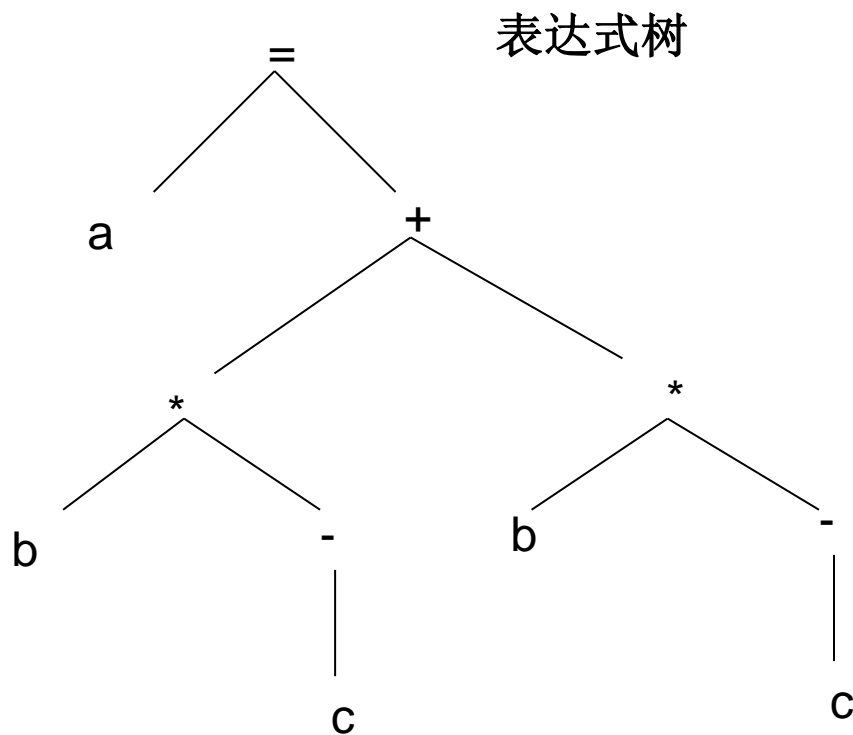
- 窥孔优化关注在目标指令的一个较短的序列上，通常称其为“窥孔”
- 通过删除其中的冗余代码，或者用更高效简洁的新代码来替代其中的部分代码，达到提升目标代码质量的目的

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

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

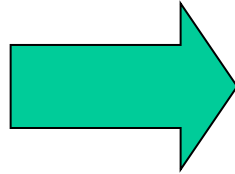
11.2.1 基本块的DAG图表示

- 赋值语句: $a = b * (-c) + b * (-c)$



11.2.2 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
```



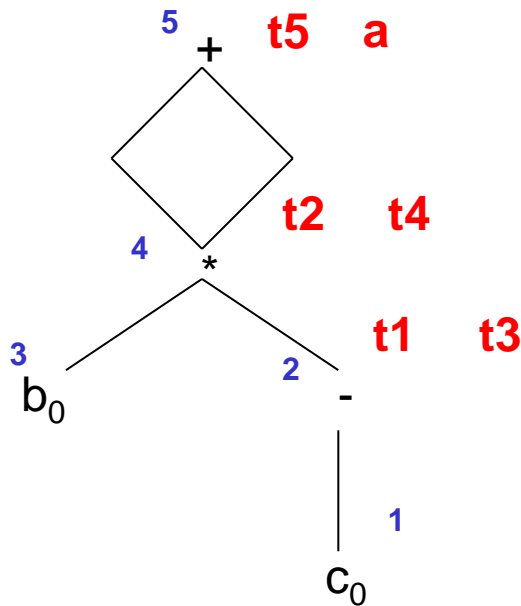
```
t1 := - c
t2 := b * t1
t3 := - c
t4 := b * t3
t5 := t2 + t2 (t4)
a := t5
```

c := c + 1 ?

建立DAG图，例1

```

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



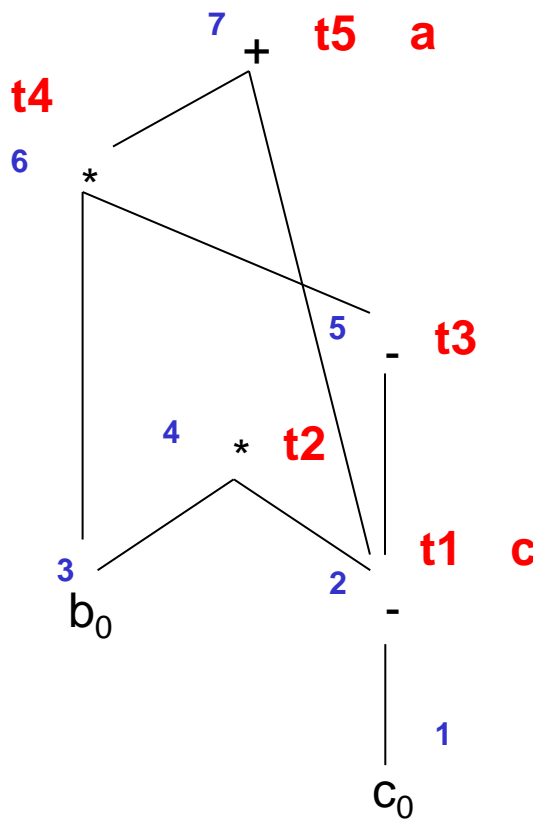
node(x)

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

11.2.3 数组、指针及函数调用

- 当中间代码序列中出现了数组成员、指针或函数调用时，中间代码或者算法13.1需要作出一定的调整，否则将得出不正确的优化结果

(1) $x = a[i]$

$a[j] = y$

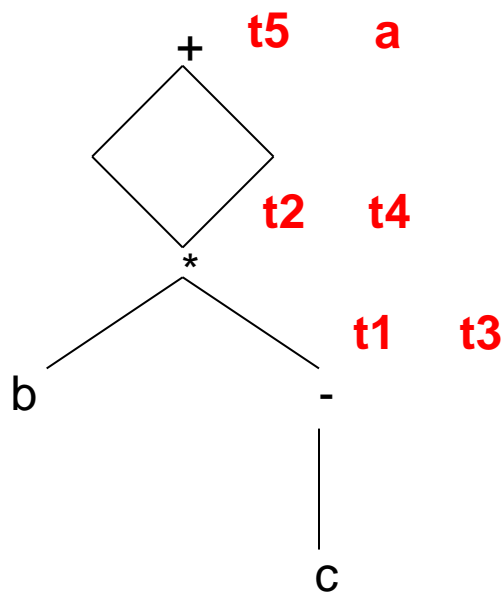
$z = a[i]$

(2) $x = *p$

$*q = y$

$z = *p$

11.2.4从DAG图重新导出中间代码

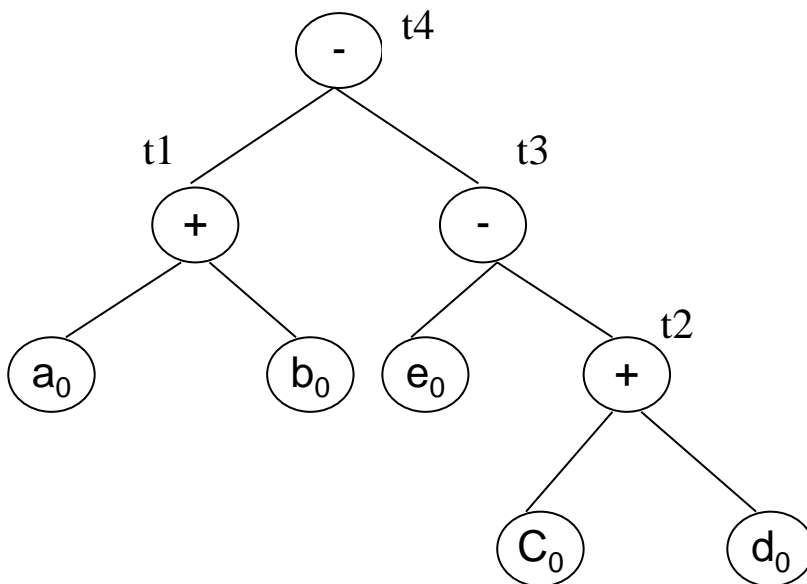


$t1 := -c$

$t2 := b * t1$

$a := t2 + t2$

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

(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

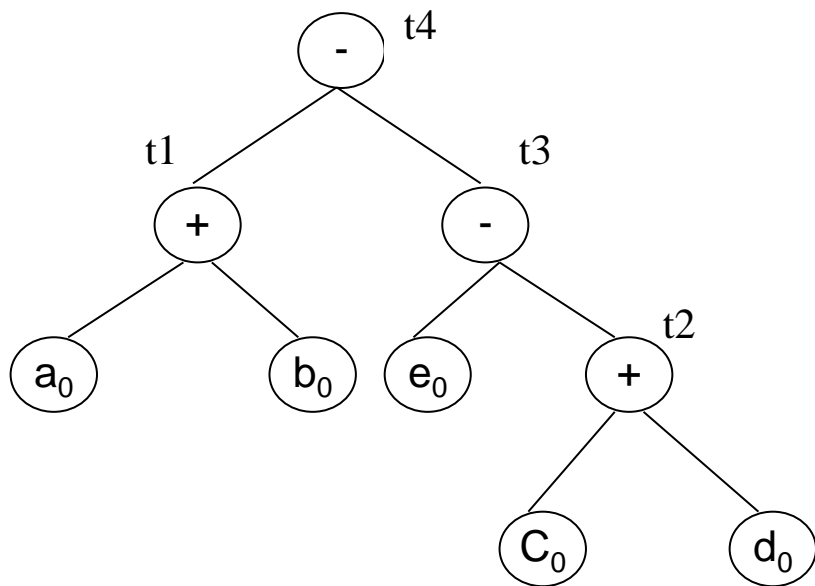
add eax, b

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

sub eax, edx

算法11.3 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



- 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
----	----	----	----

- 作业

P 356 (273) : 1,2,3

11.3 全局优化

11.3.1 数据流分析

- 编译器需要了解一些非常重要的信息，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

数据流分析方程

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

到达定义（reaching definition）分析

- 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
 - 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

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
 - $\text{kill}[B] = \text{kill}[d_1] \cup \text{kill}[d_2] \dots \cup \text{kill}[d_n]$, $d_1 \sim d_n$ 依次为基本块中的语句
 - $\text{gen}[B] = \text{gen}[d_n] \cup (\text{gen}[d_{n-1}] - \text{kill}[d_n]) \cup (\text{gen}[d_{n-2}] - \text{kill}[d_{n-1}] - \text{kill}[d_n]) \dots \cup (\text{gen}[d_1] - \text{kill}[d_2] - \text{kill}[d_3] \dots - \text{kill}[d_n])$

例:

- $\text{out}[B] = \text{gen}[B] \cup (\text{in}[B] - \text{kill}[B])$
 - $\text{in}[B]$ 为进入基本块时的数据流信息
 - $\text{kill}[B] = \text{kill}[d_1] \cup \text{kill}[d_2] \dots \cup \text{kill}[d_n]$, $d_1 \sim d_n$ 依次为基本块中的语句
 - $\text{gen}[B] = \text{gen}[d_n] \cup (\text{gen}[d_{n-1}] - \text{kill}[d_n]) \cup (\text{gen}[d_{n-2}] - \text{kill}[d_{n-1}] - \text{kill}[d_n]) \dots \cup (\text{gen}[d_1] - \text{kill}[d_2] - \text{kill}[d_3] \dots - \text{kill}[d_n])$

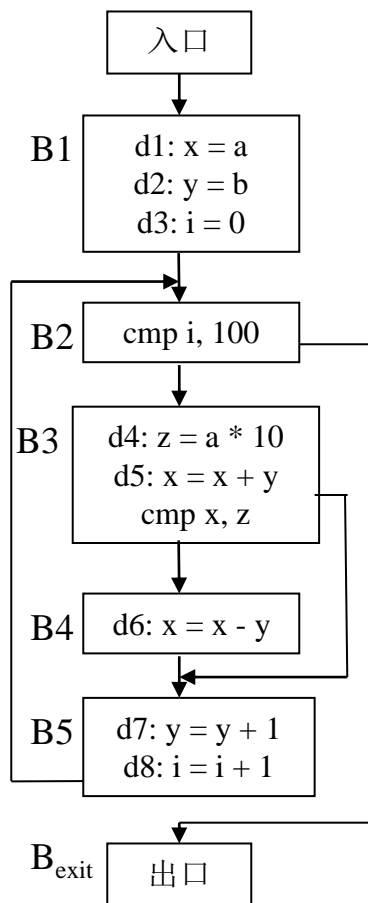
d1: $a = b + 1$

d2: $a = b + 2$

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

$$\begin{aligned} \text{gen}[B] &= \text{gen}[d2] \cup (\text{gen}[d1] - \text{kill}[d2]) = \\ &\quad \{d2\} \cup (\{d1\} - \{d1\}) = \{d2\} \end{aligned}$$

$$\begin{aligned} \text{out}[B] &= \text{gen}[B] \cup (\text{in}[B] - \text{kill}[B]) = \\ &\quad \{d2\} \cup (\text{in}[B] - \{d1, d2\}) \end{aligned}$$



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

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

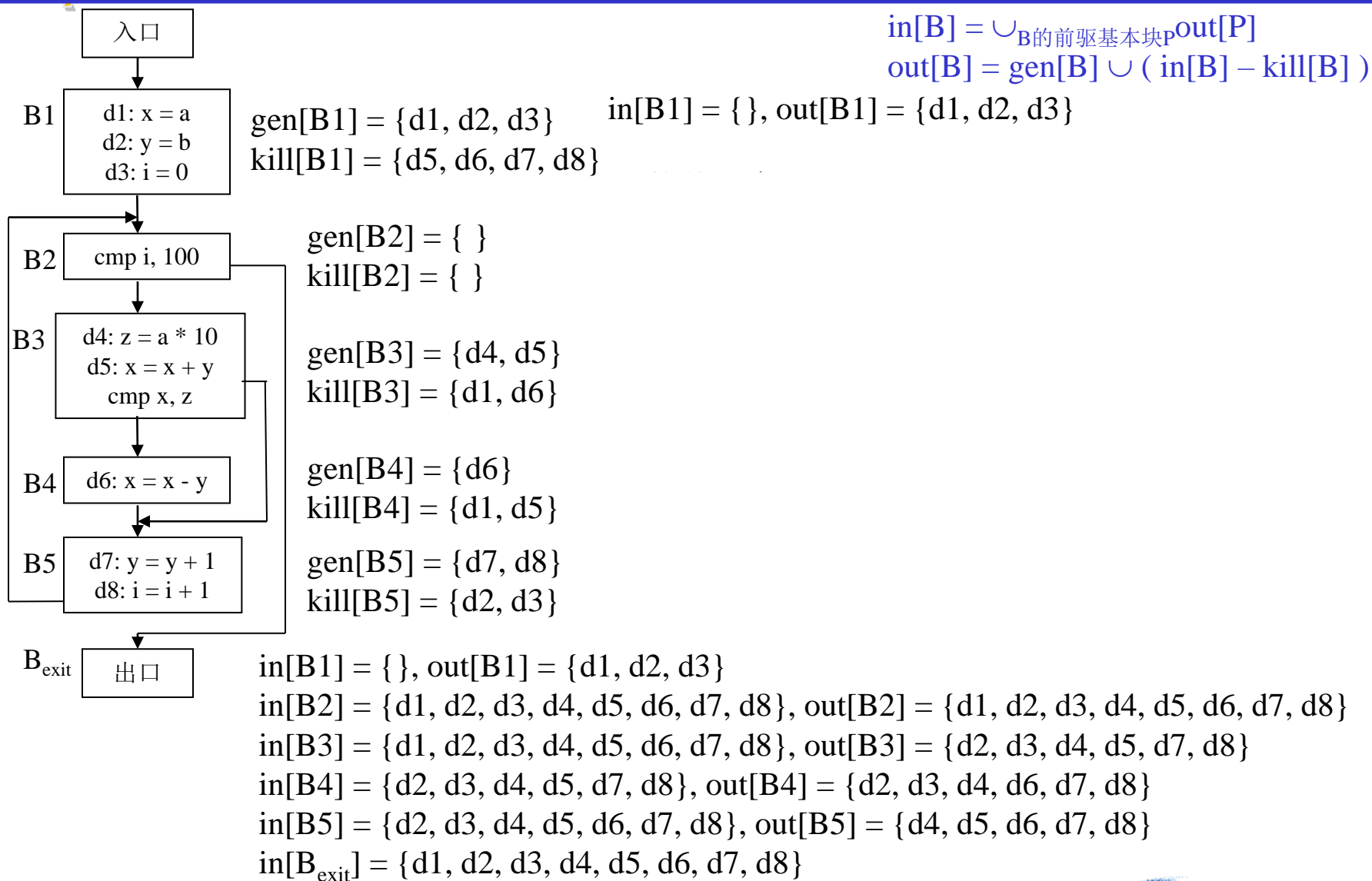
$\text{gen}[B3] = \{d4, d5\}$
 $\text{kill}[B3] = \{d1, d6\}$

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

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

算法11.4 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 $in[B] = \bigcup_{B \text{ 的前驱基本块 } P} out[P]$, $out[B] = gen[B] \cup (in[B] - kill[B])$ ，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。



11.3.2 活跃变量分析

- 变量 x 在某个执行点 p 是活跃的
 - 在流图中沿着从 p 开始的某条路经中可能引用变量 x 在 p 点的值
- 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

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:

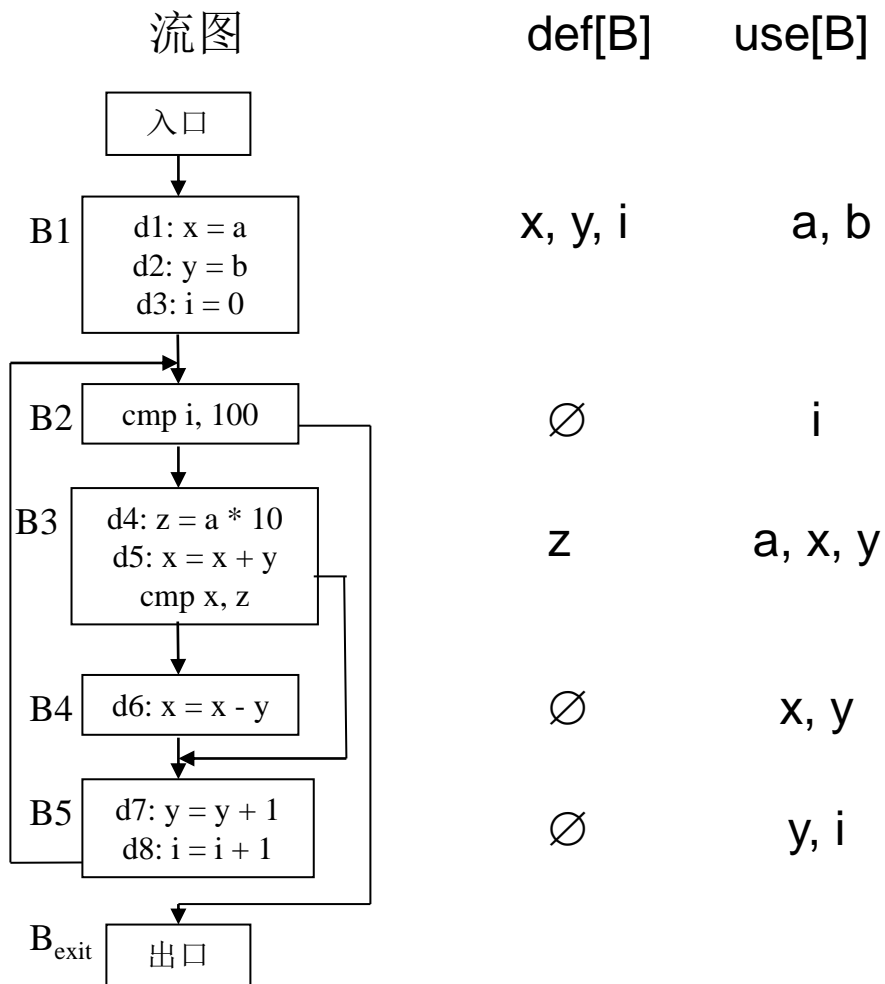
$\text{out}[B] = \text{gen}[B] \cup (\text{in}[B] - \text{kill}[B])$

- 采用 $\text{use}[B]$ 代表当前基本块新生成的数据流信息
- 采用 $\text{def}[B]$ 代表当前基本块消除的数据流信息
- 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
 - 在引用该变量前已经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

例



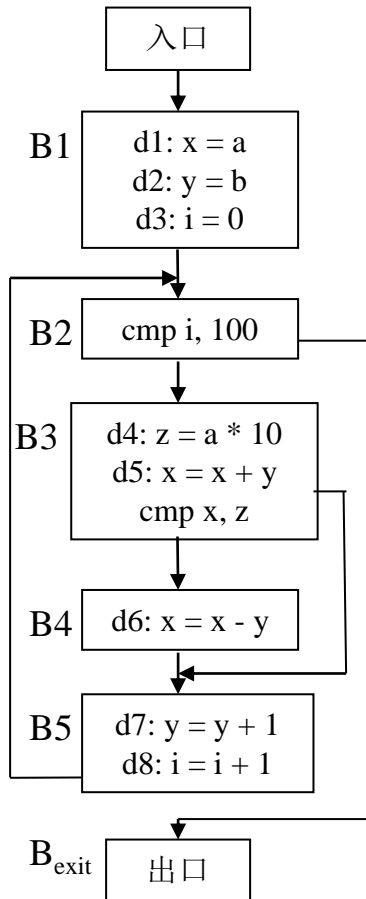
算法11.5 基本块的活跃变量数据流分析

- 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. 将流图内所有基本块的in集合，初始化为空集。
 2. 根据方程 $out[B] = \bigcup_{B \text{ 的后继基本块 } P} in[P]$ ， $in[B] = use[B] \cup (out[B] - def[B])$ ，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] =  $\emptyset$  ;  
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
```

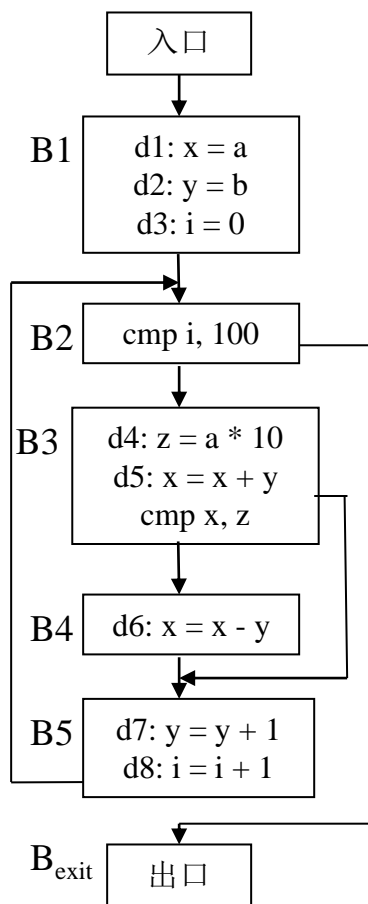

例

流图



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

流图

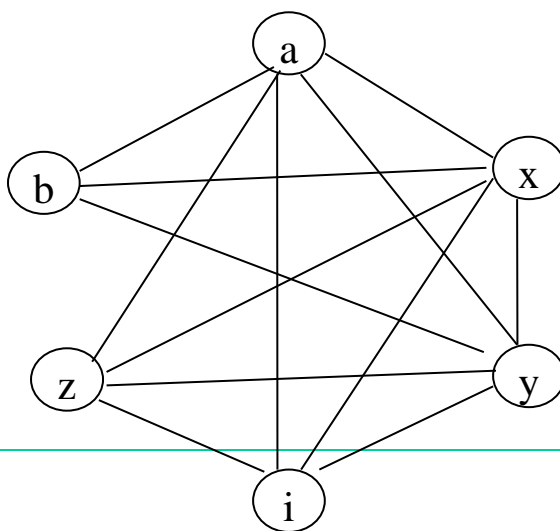


- 变量x, y, i: 均定义于B1, 在B2~B5入口处均活跃。注意, x在B3、B4中都被重新定义过, 但x被定义前均被使用过, 因此其在同一基本块中发生在使用之前的定义仅余B1。变量y和i的情况类似。

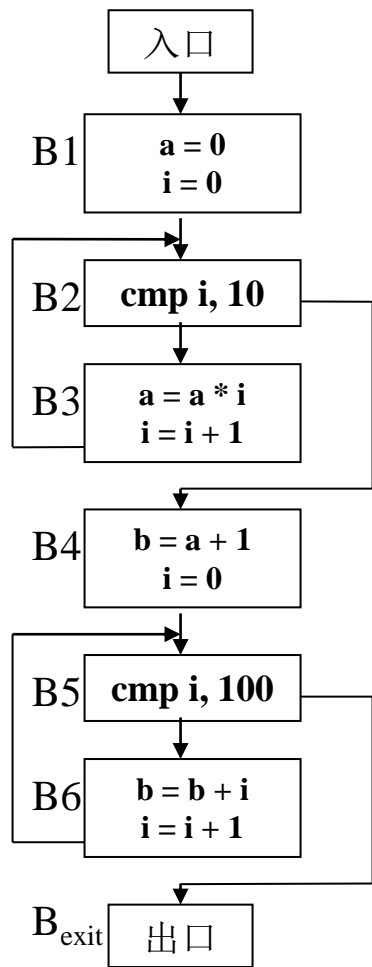
- 变量a (形参), : 在流图中无定义点, 在x,y,i,z的定义点处均活跃。

- 变量b (形参), : 在流图中无定义点, 在x,y的定义点处活跃。a,b同为形参, 互相冲突。

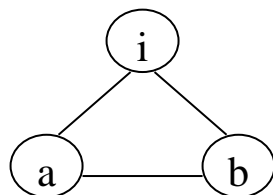
- 变量z: 在z的定义点, a,x,y,i均活跃。



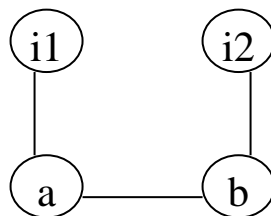
11.3.3 定义-使用链和网



(a)



(b)



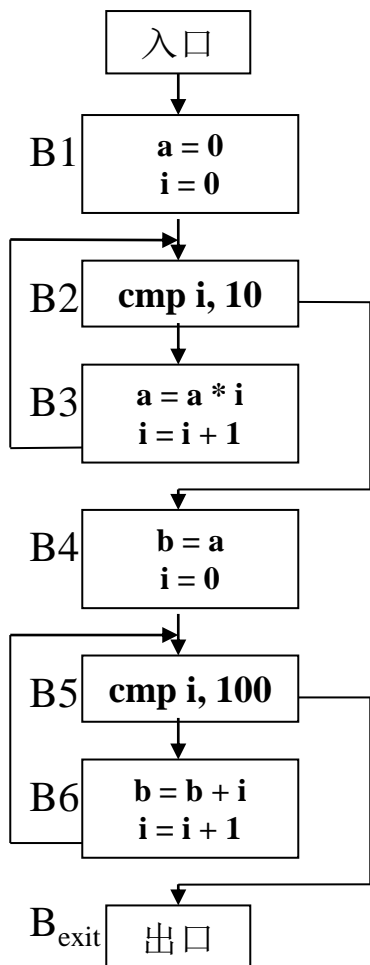
(c)

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

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，是否可以重命名为*i1*,*i2*，从而提高全局寄存器的使用效率？

- 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

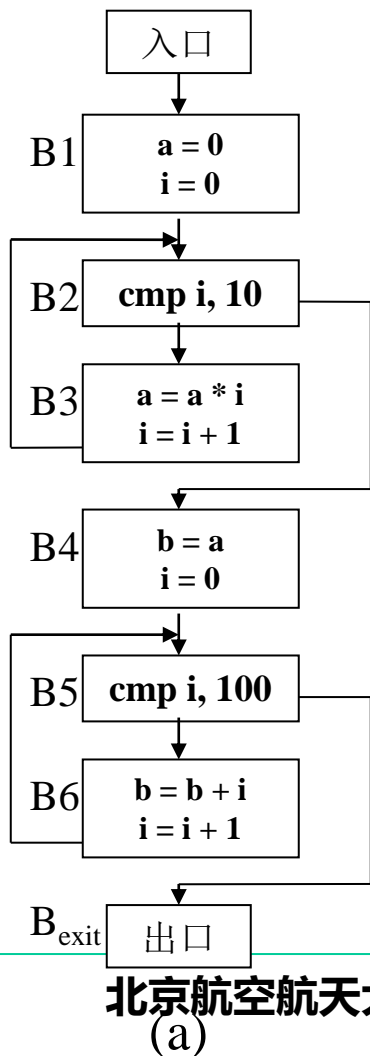


变量a: L1 {<B1, 1>, <B3, 1>, <B4, 1>}
L2 {<B3, 1>, <B3, 1>, <B4, 1>}

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>}

- 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>}

变量b:

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

L3 {<B4, 1>, <B6, 1>}

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

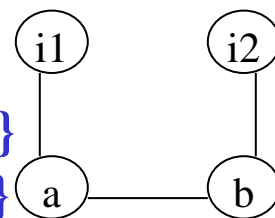
变量i:

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, 2>}}

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>}}

作业

P 356 (274) : 4,5,6

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。

$y := 1$

...

$y := 2$

$x := y + z$

$y1 := 1$

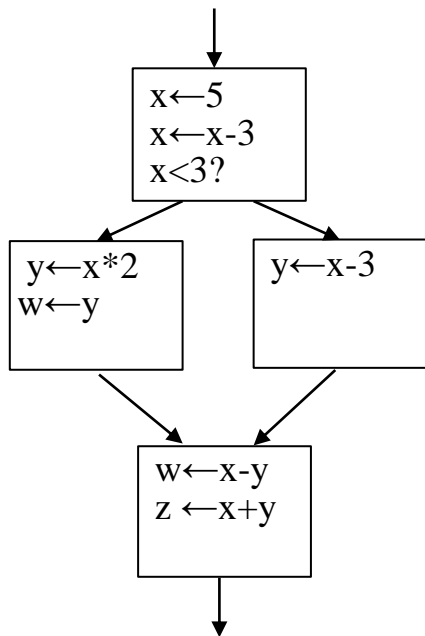
...

$y2 := 2$

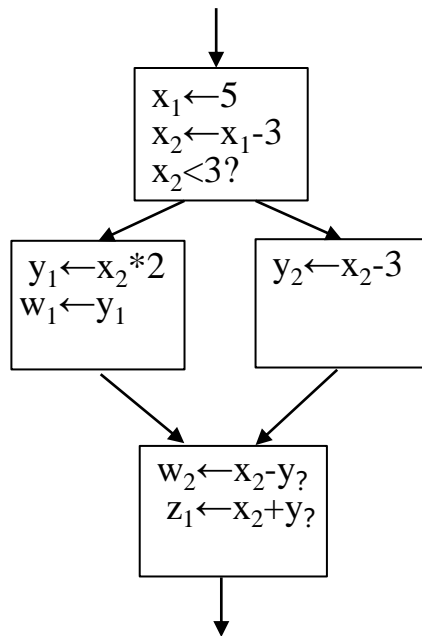
$x := y2 + z$

例, 很容易分析出y1是可以优化掉的变量

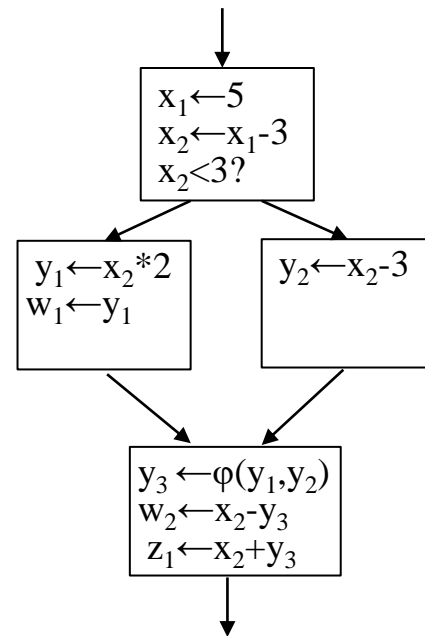
SSA可以从普通的四元式转化而来。如何转化？



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： $x_{n+1} = \Phi(x_1, x_2, x_3, \dots, x_n)$ 。

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 $DF(x) =$

$\{y | \text{如果 } z \text{ 是 } y \text{ 的前驱且 } x \text{ 支配 } z, \text{ 且 } x \text{ 不严格支配 } y\},$

严格支配即 $x \text{ dom } y \ \& \ x \neq y,$

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

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

例， $DF(B3) = \{B2\}$, 提示： $B_i \text{ dom } B_j$

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

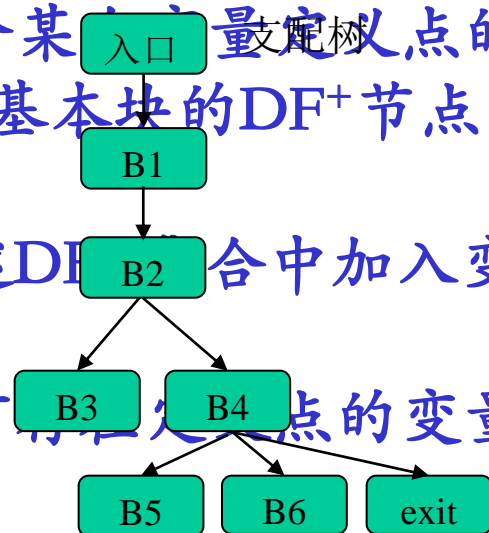
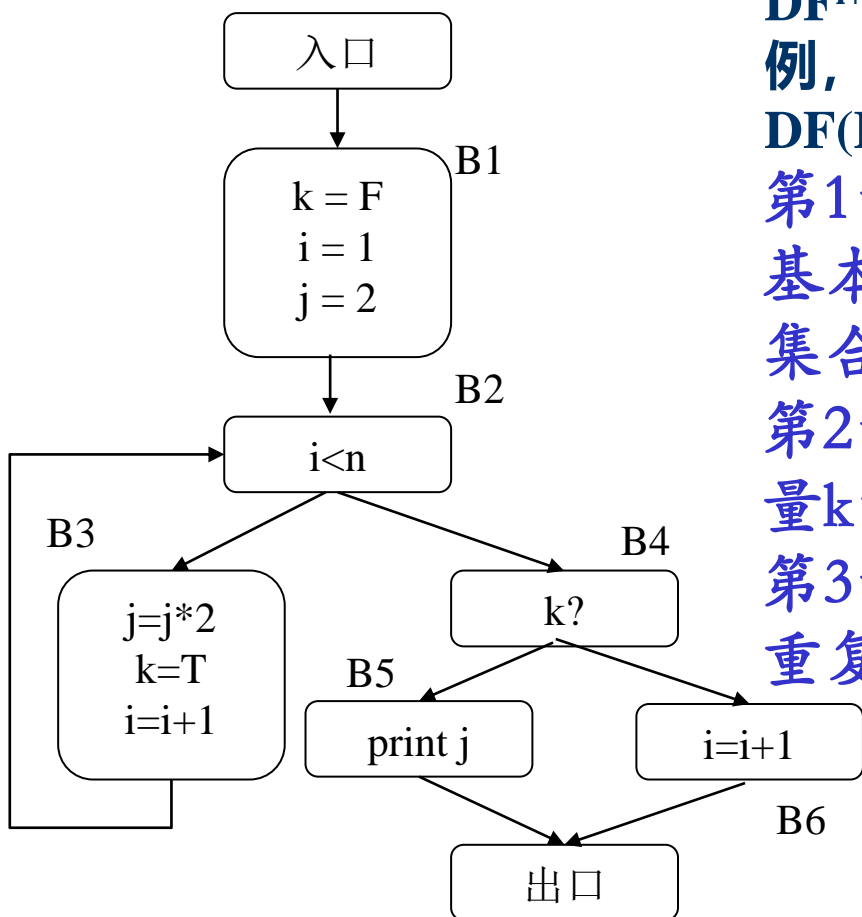
第1步：将包含某变量定义点的基本块和入口基本块的 DF^+ 节点集合计算出来

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 DF^+ 集合中的变量重复步骤1和2

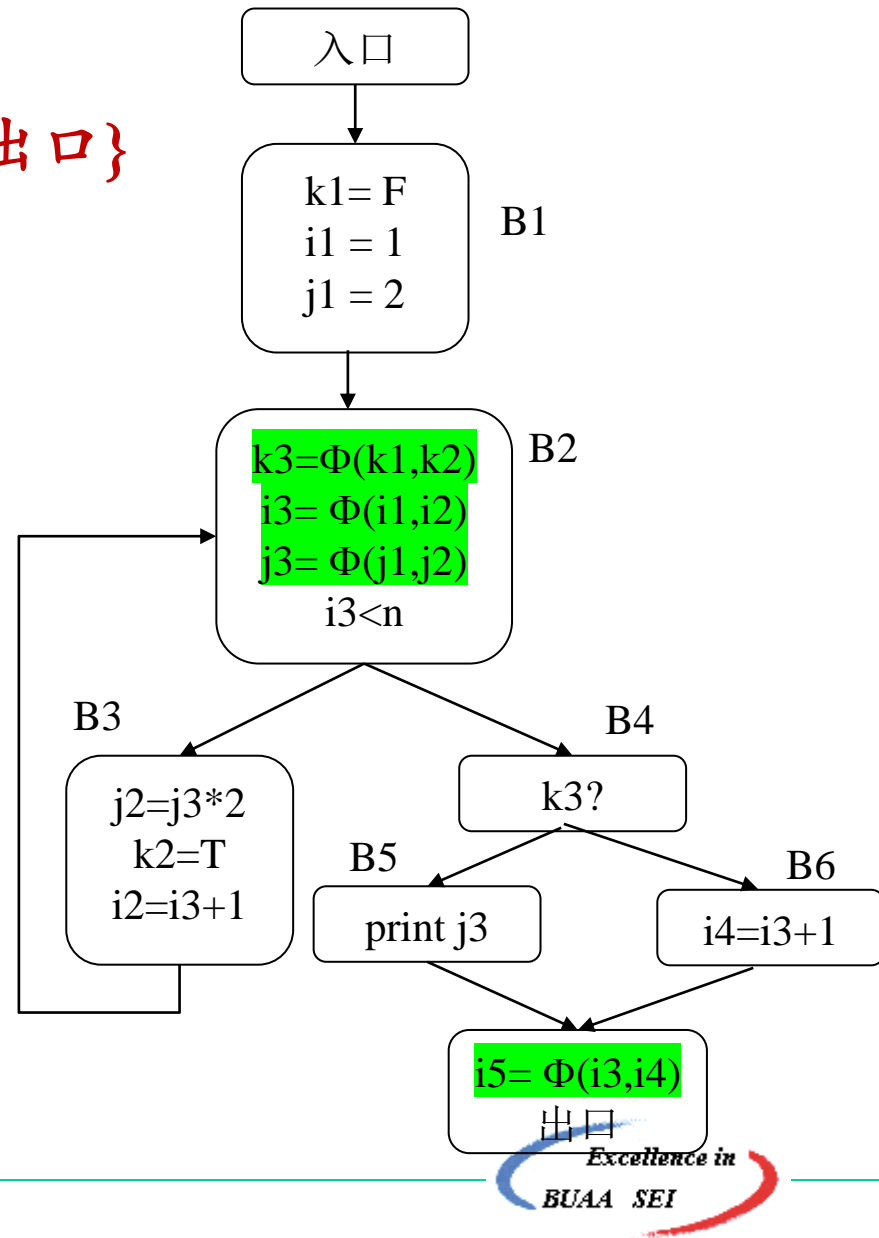
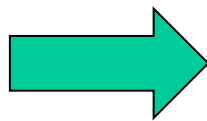
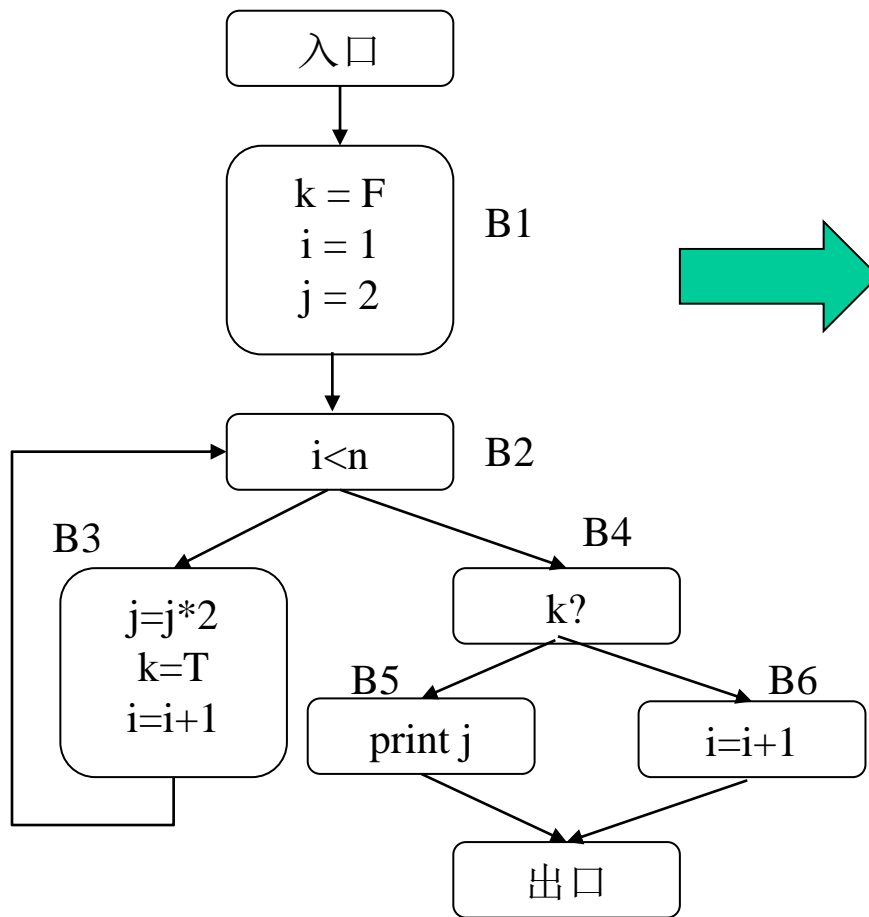
变量 k, j : $DF^+(\text{入口}, B1, B3) = \{B2\}$

变量 i : $DF^+(\text{入口}, B1, B3, B6) = \{B2, \text{出口}\}$

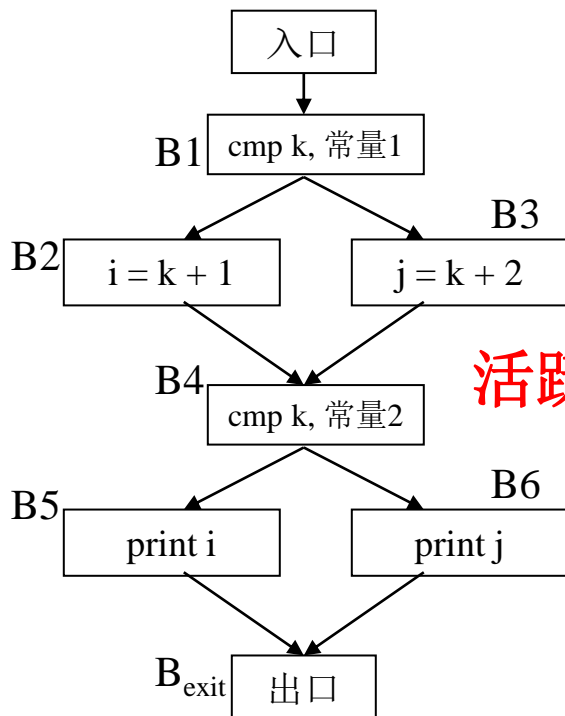


变量k, j: $DF^+(\text{入口}, B1, B3) = \{B2\}$

变量i: $DF^+(\text{入口}, B1, B3, B6) = \{B2, \text{出口}\}$



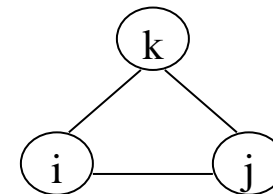
BACKUP



活跃变量: **i, j, k**

(a)

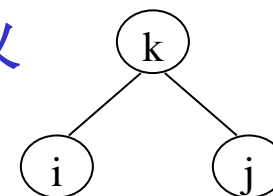
- 变量*i*和变量*j*在B2和B3中被分别定义，并在B5和B6中被分别使用。根据活跃变量分析结果，*i*和*j*一定同时在B4的入口处活跃



(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)