编译原理习题课（三）

第八章

图 8-1　代码生成器的位置

- 根据中间表示生成代码
- 代码生成器之前可能有一个优化组件
- 代码生成器的三个主要任务
  - 指令选择：选择适当的指令实现IR语句
  - 寄存器分配和指派：把哪个值放在哪个寄存器中
  - 指令排序：按照什么顺序安排指令执行

主要内容

- 代码生成器设计中的问题
- 目标语言
- 目标代码中的地址
- 基本块和流图
- 基本块优化方法
- 寄存器分配和指派

目标语言

- 加载：LD dst, addr
  - 把地址addr中的内容加载到dst所指寄存器
- 保存：ST x, r
  - 把寄存器r中的内容保存到x中
- 计算：OP dst, src1, src2
  - 把src1和scr2中的值运算后将结果存放到dst中
- 无条件跳转：BR L
  - 控制流转向标号L的指令
- 条件跳转：Bcond r, L
  - 对r中的值进行测试，如果为真则转向L

寻址模式

- 变量x：指向分配x的内存位置
- a(r)：地址是a的左值加上r中的值
- constant(r)：寄存器中内容加上前面的常数即其地址
- *r：寄存器r的内容为其地址
- *constant(r)：r中内容加上常量所指地址中存放的值为其地址
- 常量#constant

1. 假设a和b是元素为4字节值的数组，为下面的三地址语句序列生成代码：

（1）四个语句的序列

```
x = a[i]
y = b[j]
a[i] = y
b[j] = x
```

立即数加#

答案：(不唯一)

```
LD   R1, i
MUL  R1, R1, #4
LD   R2, a(R1)
LD   R3, j
MUL  R3, R3, #4
LD   R4, b(R3)
ST   a(R1), R4
ST   b(R3), R2
```

1. 假设a和b是元素为4字节值的数组，为下面的三地址语句序列生成代码：

(2) 三个语句的序列

```
x = a[i]
y = b[i]
z = x * y
```

数组记得乘偏移量

答案：(不唯一)

```
LD   R1, i
MUL  R1, R1, #4
LD   R2, a(R1)
LD   R3, b(R1)
MUL  R4, R2 * R3
ST   z, R4
```

2. 假设x、y和z存放在内存位置中，为下面的三地址语句序列生成代码：

```
    if x < y goto L1
    z = 0
    goto L2
L1: z = 1
```

store只能从寄存器到内存，不能直接存立即数

答案：(不唯一)

```
    LD   R1, x
    LD   R2, y
    SUB  R1, R1, R2
    BLTZ R1, L1
    LD   R3, #0
    ST   z, R3
    BR   L2
L1: LD R4, #1
    ST z, R4
```

目标代码中的地址

- 静态分配
- 活动记录的大小和布局由符号表决定
- 每个过程静态地分配一个数据区域
  - 开始位置用staticArea表示
- 栈式分配
  - 活动记录的位置存放在寄存器中
  - 偏移地址
- 过程调用指令序列
  - ADD SP, SP, #caller.recordSize //增加栈指针
  - ST 0(SP), #here+16 //保存返回地址
  - BR callee.codeArea //转移到被调用者
- 过程返回指令序列
  - BR *0(SP) //被调用者执行，返回调用者
  - SUP SP, SP, #caller.recordSize //调用者减少栈指针值

3. 假设使用栈式分配而寄存器SP指向栈的顶端，为下列的三地址语句生成代码：

```
call p
call q
return
call r
return
return
```

注意写代码行号

答案：

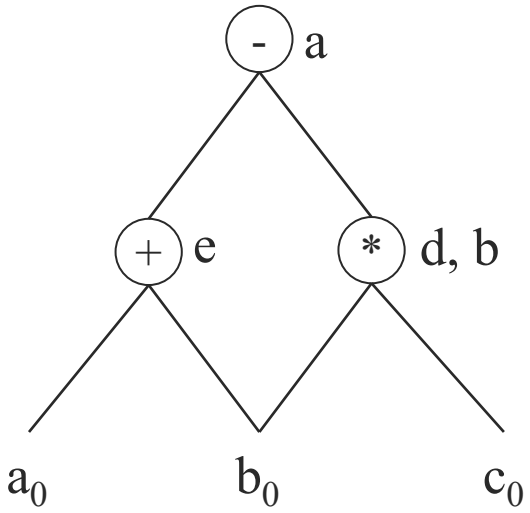
```
// 假设起始位置是100，过程p,q,r的代码分别从地址200，400，600开始
100: LD SP, #stackStart
108: ADD SP, SP, #psize
116: ST 0(SP), #132
124: BR 200
132: SUB SP, SP, #psize
140: ADD SP, SP, #qsize
148: ST 0(SP), #164
156: BR 400
164: SUB SP, SP, #qsize
172: BR *0(SP)
180: ADD SP, SP, #rsize
188: ST 0(SP), #204
196: BR 600
204: SUB SP, SP, #rsize
212: BR *0(SP)
220: BR *0(SP)
```

基本块的DAG表示

- 基本块可以用DAG表示
  - 每个变量有对应的DAG的结点，代表初始值
  - 每个语句s有一个相关的结点N，代表计算得到的值
    - N的子结点对应于（其运算分量当前值的）其它语句
    - 结点N的标号是s的运算符
    - N和一组变量关联，表示s是在此基本块内最晚对它们定值的语句
- 输出结点：结点对应的变量在基本块出口处活跃
- 从DAG，我们可以知道各个变量最后的值和初始值的关系

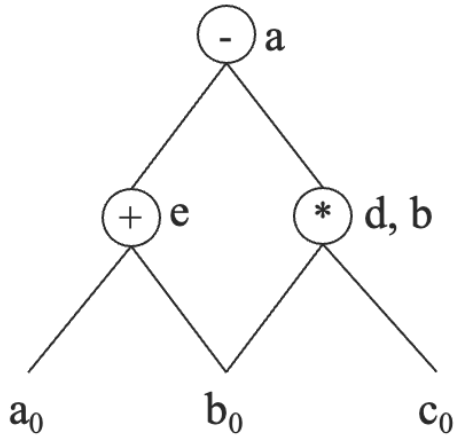
4. 为下面的基本块构造DAG:

```
d = b * c
e = a + b
b = b * c
a = e - d
```



4. 为下面的基本块构造DAG:

```
d = b * c
e = a + b
b = b * c
a = e - d
```



分别按照下列两种假设简化三地址代码:

（1）只有a在基本块的出口处活跃。

```
d = b * c
e = a + b
a = e - d
```

（2）a、b、c在基本块的出口处活跃。

```
e = a + b
b = b * c
a = e - b
```

5. 为下面的每个C语言赋值语句生成三地址代码:

(1) x = a + b*c ;

```
// 先把代码转化为三地址语句
t = b * c
u = a + t
x = u
// 再转化为指令代码
LD R1, b
LD R2, c
MUL R1, R1, R2 //存R1或R2都可以，后面不引用了
LD R3, a
ADD R1, R1, R3
ST x, R1
```

5. 为下面的每个C语言赋值语句生成三地址代码:

(2) x = a[j] + 1;

注意三地址码需要地址偏移量的计算

```
// 假设数组长度为4
t = a[j]
u = t + 1
x = u
//
LD R1, j
MUL R1, R1, 4
LD R2, a(R1)
ADD R2, R2, 1
ST x, R2
```

补充题

为基本块

b = a [i]

c = b + d

a [j] = c

构造 DAG 图

补充题

为基本块

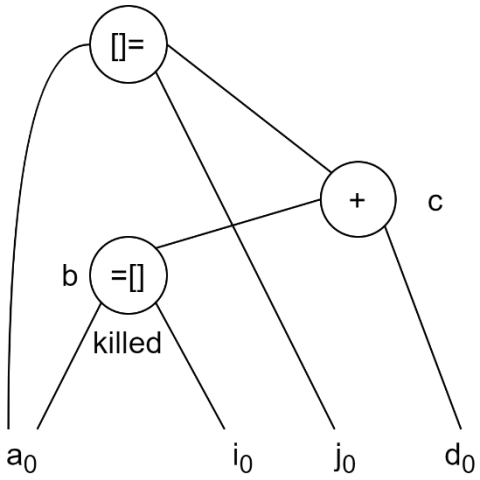
b = a [i]

c = b + d

a [j] = c

构造 DAG 图

只要有对数组的赋值，所有有关该数组的运算节点都应kill



练习 8.6.4：假设有三个可用的寄存器，使用本节中的简单代码生成算法，把在练习 8.6.1 中得到的三地址代码转换为机器代码。请给出每一个步骤之后的寄存器和地址描述符。

（1）x = a + b*c ;

基本思路

- 依次考虑各三地址指令，尽可能把值保留在寄存器中，以减少寄存器/内存之间的数据交换
- 为一个三地址指令生成机器指令时
  - 只有当运算分量不在寄存器中时，才从内存载入
  - 尽量保证只有当寄存器中的值不被使用时，才把它覆盖掉

基本数据结构

- 记录各个值对应的位置
  - 寄存器描述符：跟踪各个寄存器都存放了哪些变量的当前值
  - 地址描述符：某个变量的当前值存放在哪个或哪些位置 包括内存位置和寄存器）上

代码生成算法

- 运算语句：x = y + z
  - 调用getReg(x=y+z)，为x,y,z选择寄存器Rx, Ry, Rz
  - 查Ry的寄存器描述符，如果y不在Ry中则生成指令
    - LD Ry y’　　(y’表示存放y值的当前位置)
  - 类似地，确定是否生成LD Rz, z’
  - 生成指令ADD Rx, Ry, Rz
- 赋值语句：x=y
  - getReg(x=y)总是为x和y选择相同的寄存器
  - 如果y不在Ry中，生成机器指令LD Ry, y
- 基本块的收尾
  - 如果变量x在出口处活跃，且x现在不在内存，那么生成指令ST x, Rx

代码生成算法

- 代码生成同时更新寄存器和地址描述符
- 处理普通指令时生成LD R x
  - R的寄存器描述符：只包含x
  - x的地址描述符：R作为新位置加入到x的位置集合中
  - 从任何不同于x的变量的地址描述符中删除R
- ST x，R
  - 修改x的地址描述符，包含自己的内存位置

代码生成算法

- ADD Rx, Ry, Rz
  - Rx的寄存器描述符只包含x
  - x的地址描述符只包含Rx（不包含x的内存位置！）
  - 从任何不同于x的变量的地址描述符中删除Rx。
- 处理x=y时，
  - 如果生成LD Ry y，按照第一个规则处理；
  - 把x加入到Ry的寄存器描述符中（Ry存放了x和y的当前值）；
  - 修改x的地址描述符，使它只包含Ry

练习 8.6.4：假设有三个可用的寄存器，使用本节中的简单代码生成算法，把在练习 8.6.1 中得到的三地址代码转换为机器代码。请给出每一个步骤之后的寄存器和地址描述符。

(1) x = a + b*c ;

	R1	R2	R3	x	a	b	c	t	u
t = b * c				x	a	b	c		
LD R1, b	b					b, R1			
LD R2, c		c					c, R2		
MUL R1, R1, R2	t					b		R1	
	t	c		x	a	b	c, R2	R1	

练习 8.6.4：假设有三个可用的寄存器，使用本节中的简单代码生成算法，把在练习 8.6.1 中得到的三地址代码转换为机器代码。请给出每一个步骤之后的寄存器和地址描述符。

(1) x = a + b*c ;

注意应当考虑先将空闲寄存器占满再覆盖

	R1	R2	R3	x	a	b	c	t	u
	t	c		x	a	b	c, R2	R1	
u = a + t									
LD R3, a			a		a, R3				
ADD R1, R1, R3	u							/	R1
	u	c	a	x	a, R3	b	c, R2		R1

练习 8.6.4：假设有三个可用的寄存器，使用本节中的简单代码生成算法，把在练习 8.6.1 中得到的三地址代码转换为机器代码。请给出每一个步骤之后的寄存器和地址描述符。

(1) x = a + b*c ;

	R1	R2	R3	x	a	b	c	t	u
	u	c	a	x	a, R3	b	c, R2		R1
x = u									
ST x, R1	u, x			x, R1					
	u, x	c	a	x, R1	a, R3	b	c, R2		R1

练习 8.6.5：重复练习 8.6.4，但是假设只有两个可用的寄存器。

练习 8.6.4：假设有三个可用的寄存器，使用本节中的简单代码生成算法，把在练习 8.6.1 中得到的三地址代码转换为机器代码。请给出每一个步骤之后的寄存器和地址描述符。

(1) x = a + b*c ;

练习 8.6.5：重复练习 8.6.4，但是假设只有两个可用的寄存器。

练习 8.6.4：假设有三个可用的寄存器，使用本节中的简单代码生成算法，把在练习 8.6.1 中得到的三地址代码转换为机器代码。请给出每一个步骤之后的寄存器和地址描述符。

(1) x = a + b*c ;

	R1	R2	x	a	b	c	t	u
t = b * c			x	a	b	c		
LD R1, b	b				b, R1			
LD R2, c		c				c, R2		
MUL R2, R1, R2		t				c	R2	
	b	t	x	a	b, R1	c	R2	

练习 8.6.5：重复练习 8.6.4，但是假设只有两个可用的寄存器。

练习 8.6.4：假设有三个可用的寄存器，使用本节中的简单代码生成算法，把在练习 8.6.1 中得到的三地址代码转换为机器代码。请给出每一个步骤之后的寄存器和地址描述符。

(1) x = a + b*c;

	R1	R2	x	a	b	c	t	u
	b	t	x	a	b, R1	c	R2	
u = a + t								
LD R1, a	a			a, R1	b			
ADD R1, R1, R2	u			a				R1
	u	t	x	a	b	c	R2	R1

练习 8.6.5：重复练习 8.6.4，但是假设只有两个可用的寄存器。

练习 8.6.4：假设有三个可用的寄存器，使用本节中的简单代码生成算法，把在练习 8.6.1 中得到的三地址代码转换为机器代码。请给出每一个步骤之后的寄存器和地址描述符。

(1) x = a + b*c ;

	R1	R2	x	a	b	c	t	u
	u	t	x	a	b	c	R2	R1
x = u								
ST x, R1	u, x		x, R1					
	u, x	t	x, R1	a	b	c	R2	R1

第九章——主要内容

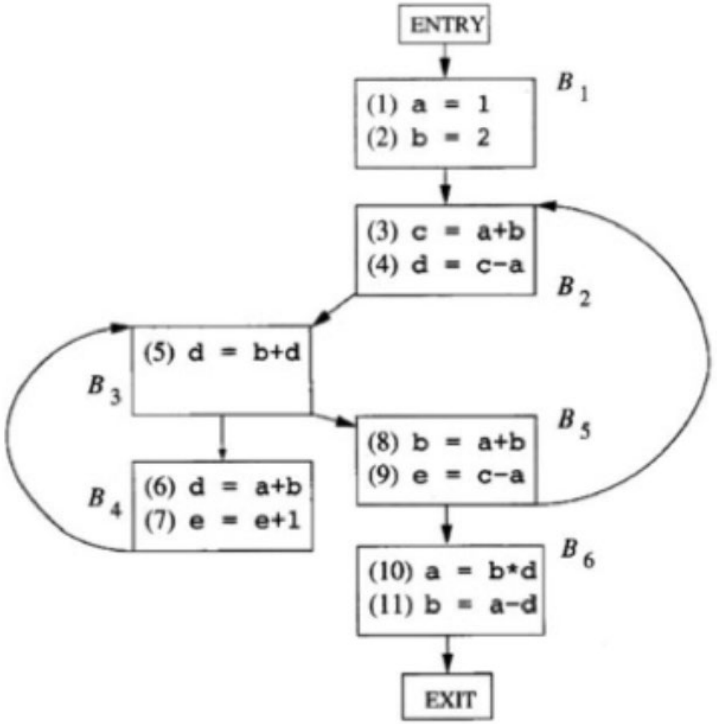
- 优化的来源
  - 全局公共子表达式
  - 复制传播
  - 死代码消除
  - 代码移动
  - 归纳变量和强度消减
- 数据流分析
  - 到达定值分析
  - 活跃变量分析
  - 可用表达式分析
- 循环的优化

第九章

- 代码优化或者代码改进
  - 在目标代码中消除不必要的指令
  - 把一个指令序列替换为一个完成相同功能的更快的指令序列
- 全局优化
  - 具体的优化实现基于数据流分析技术
  - 用以收集程序相关信息的算法

第九章

1. 对于图9-10中的流图:
（1）找出流图中的循环。
（2）B1中的语句(1)和(2)都是复制语句。其中a和b都被赋予了常量值。我们可以对a和b的哪些使用进行复制传播，并把对它们的使用替换为对一个常量的使用?
（3）对每个循环，找出所有的全局公共子表达式。

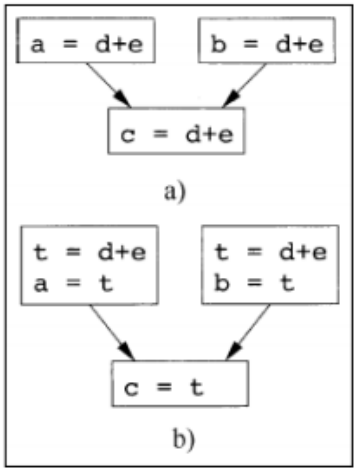


语义不变的优化方法

- 公共子表达式消除（题1）
- 复制传播 （题1）
- 死代码消除（通常在复制传播优化后）
- 常量折叠
- 代码移动
- 归纳变量和强度消减 （题2）

语义不变的优化方法

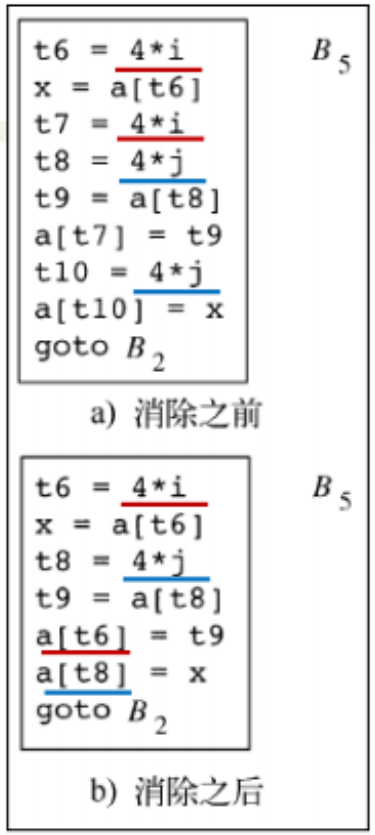
- 复制传播
- 死代码消除
  - 形如$u = v$的复制语句使得语句后面的程序点上，u的值等于v的值
    - 如果在某个位置上u一定等于v，那么可以把u替换为v
    - 有时可以彻底消除对u的使用，从而消除对u的赋值语句
  - 右图所示，消除公共子表达式时引入了复制语句
  - 如果尽可能用t来替换c，可能就不需要c = t这个语句了



语义不变的优化方法

➢ 公共子表达式消除

- 如果E
  - 在某次出现之前必然已被计算过，且
  - E的运算分量在该次计算之后没有被改变
  - 那么，E的本次出现就是一个**公共子表达式**
- 如果上一次E的值赋给了x，且x的值没有被修改过，那么我们可以使用x，而不需要计算E



由于4*i和4*j的值没变无需重复计算

语义不变的优化方法

➢ 归纳变量和强度消减

- 归纳变量
  - 每次对x赋值都使x增加c
  - 把赋值改为增量操作，可消减计算强度
  - 两个归纳变量步调一致，可删除一个
  - 例子
    - 循环开始保持t4 = 4 * j
    - j = j − 1后面的t4 = 4 * j每次赋值使t4减4
    - 可替换为t4 = t4 − 4
    - t2也可同样处理

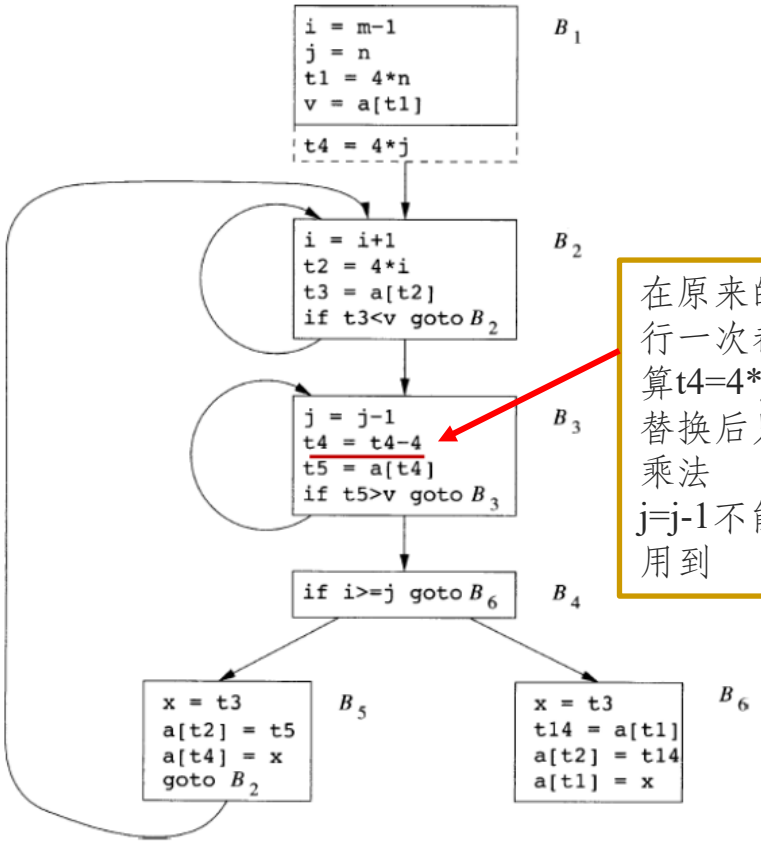


图 9-8　对基本块 B_3 中的 $4 * j$ 应用强度消减优化

第九章

1. 对于图9-10中的流图:

（1）找出流图中的循环。

B3-B4

B2-B3-B4-B5

（2）B1中的语句(1)和(2)都是复制语句。其中a和b都被赋予了常量值。我们可以对a和b的哪些使用进行复制传播，并把对它们的使用替换为对一个常量的使用?

B2: 将语句(3)替换为c=1+b

将语句(4)替换为d=c-1

B4: 将语句(6)替换为d=1+b

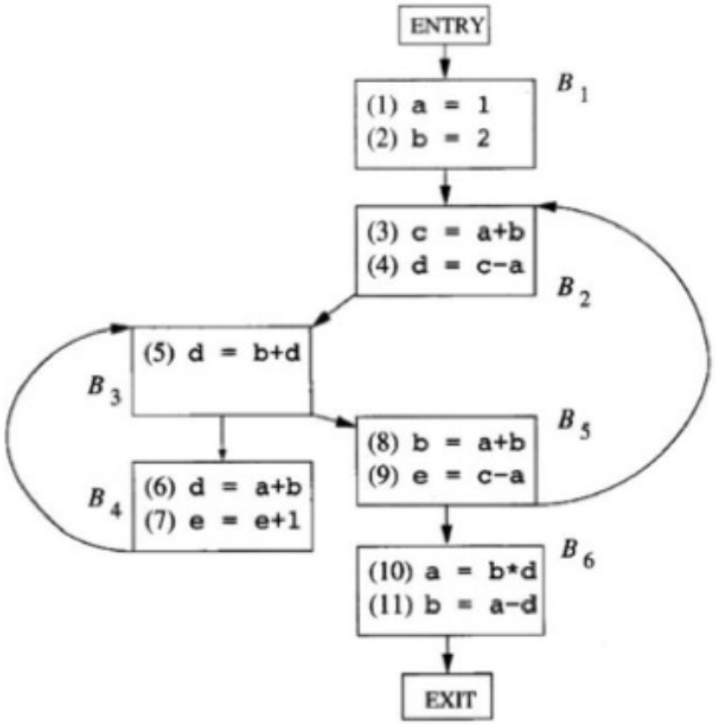
B5: 将语句(8)替换为b=1+b

将语句(9)替换为e=c-1

（3）对每个循环，找出所有的全局公共子表达式。

B3-B4 : 无

B2-B3-B4-B5： a+b c-a



2. 下图是用来计算两个向量A和B的点积的中间代码。尽你所能，通过下列方式优化这个代码：消除公共子表达式，对归纳变量进行强度消减，消除归纳变量。

```
    dp = 0.
    i  = 0
L:  t1 = i*8
    t2 = A[t1]
    t3 = i*8
    t4 = B[t3]
    t5 = t2*t4
    dp = dp+t5
    i  = i+1
    if i<n goto L
```

消除公共子表达式：
删除t3=i*8
t4=B[t3]改为t4=B[t1]

强度消减，消除归纳变量（乘法改为加法）：
把t1=i*8插入到循环之前
把t1=t1+8插入到if语句之前

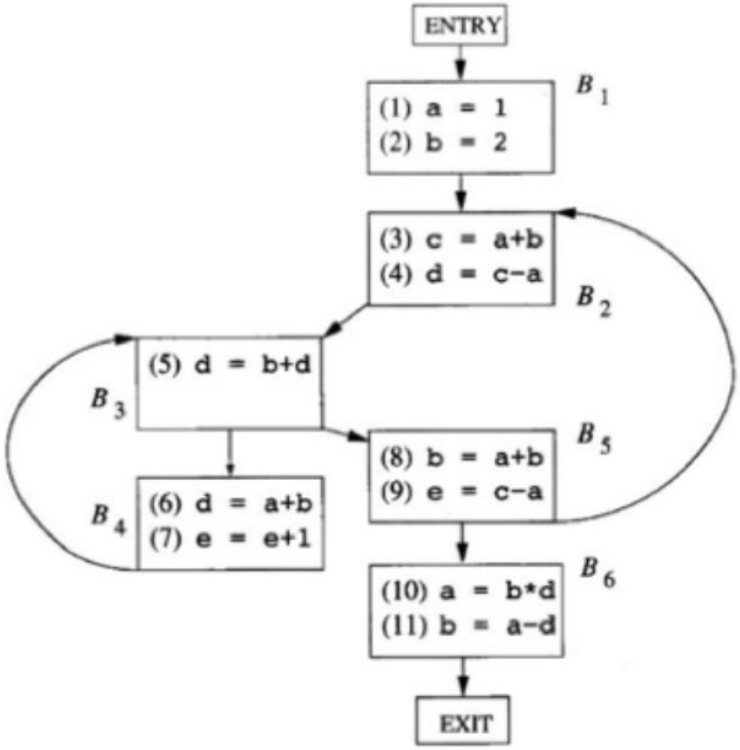
第九章

- 数据流分析
  - 用于获取数据沿着程序执行路径流动信息的相关技术
  - 是优化的基础
- 例如
  - 两个表达式是否一定计算得到相同的值？(公共子表达式) 可用表达式分析
  - 一个语句的计算结果是否可能被后续语句使用？(死代码消除) 活跃变量分析

	到达定值	活跃变量	可用表达式
域	Sets of definitions	Sets of variables	Sets of expressions
方向	Forwards	Backwards	Forwards
传递函数	$gen_B \cup (x - kill_B)$	$use_B \cup (x - def_B)$	$e\_gen_B \cup (x - e\_kill_B)$
边界条件	$\text{OUT}[\text{ENTRY}] = \emptyset$	$\text{IN}[\text{EXIT}] = \emptyset$	$\text{OUT}[\text{ENTRY}] = \emptyset$
交汇运算($\wedge$)	$\cup$	$\cup$	$\cap$
方程组	$\text{OUT}[B] = f_B(\text{IN}[B])$ $\text{IN}[B] = \bigwedge_{P,pred(B)} \text{OUT}[P]$	$\text{IN}[B] = f_B(\text{OUT}[B])$ $\text{OUT}[B] = \bigwedge_{S,succ(B)} \text{IN}[S]$	$\text{OUT}[B] = f_B(\text{IN}[B])$ $\text{IN}[B] = \bigwedge_{P,pred(B)} \text{OUT}[P]$
初始值	$\text{OUT}[B] = \emptyset$	$\text{IN}[B] = \emptyset$	$\text{OUT}[B] = U$

三种数据流方程的总结

3. 对图9-10中的流图，计算下列值：
（1）每个基本块的gen和kill集合。
（2）每个基本块的IN和OUT集合。

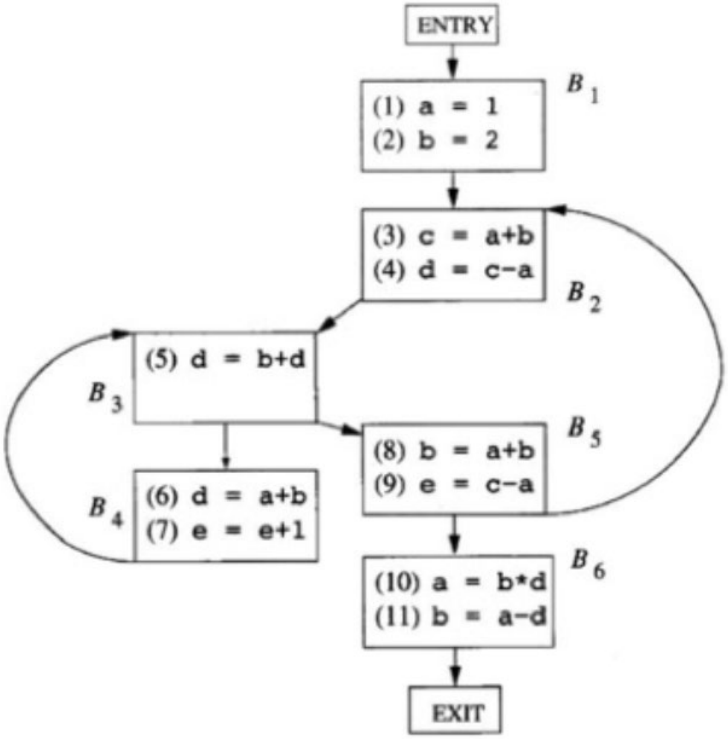


- **到达定值**
  - 假定x有定值d，如果存在一个路径，从紧随d的点到达某点p，并且此路径上面没有x的其他定值点，则称x的定值d到达p
  - 如果在这条路径上有对x的其它定值，我们说变量x的这个定值d被**杀死**了

Block	*gen*	*use*
B1	*d1, d2*	*d8, d10, d11*
B2	*d3, d4*	*d5, d6*
B3	*d5*	*d4, d6*
B4	*d6, d7*	*d4, d5, d9*
B5	*d8, d9*	*d2, d7, d11*
B6	*d10, d11*	*d1, d2, d8*

第九章

3. 对图9-10中的流图，计算下列值：
（1）每个基本块的gen和kill集合。
（2）每个基本块的IN和OUT集合。

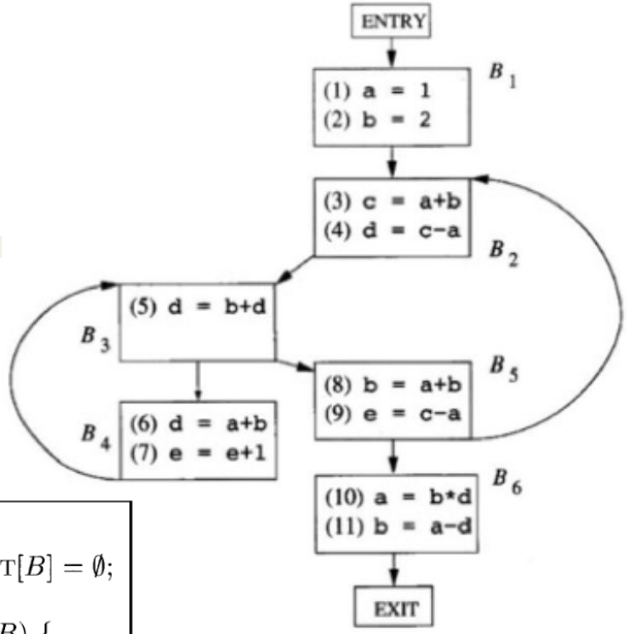


- 输入：流图、各基本块的*kill*和*gen*集合
- 输出：IN[*B*]和OUT[*B*]
- 方法

```
1)  OUT[ENTRY] = ∅;
2)  for (除ENTRY之外的每个基本块B) OUT[B] = ∅;
3)  while (某个OUT值发生了改变)
4)      for (除ENTRY之外的每个基本块B) {
5)          IN[B] = ⋃_{P是B的一个前驱} OUT[P];
6)          OUT[B] = gen_B ∪ (IN[B] − kill_B);
        }
```

Block	*gen*	*kill*
B1	*d1, d2*	*d8, d10, d11*
B2	*d3, d4*	*d5, d6*
B3	*d5*	*d4, d6*
B4	*d6, d7*	*d4, d5, d9*
B5	*d8, d9*	*d2, d7, d11*
B6	*d10, d11*	*d1, d2, d8*

第九章



3. 对图9-10中的流图，计算下列值：

（1）每个基本块的gen和kill集合。

（2）每个基本块的IN和OUT集合。

Block	*gen*	*use*
B1	*d1, d2*	*d8, d10, d11*
B2	*d3, d4*	*d5, d6*
B3	*d5*	*d4, d6*
B4	*d6, d7*	*d4, d5, d9*
B5	*d8, d9*	*d2, d7, d11*
B6	*d10, d11*	*d1, d2, d8*

```
1)  OUT[ENTRY] = ∅;
2)  for (除ENTRY之外的每个基本块B) OUT[B] = ∅;
3)  while (某个OUT值发生了改变)
4)      for (除ENTRY之外的每个基本块B) {
5)          IN[B] = ⋃_{P是B的一个前驱} OUT[P];
6)          OUT[B] = gen_B ∪ (IN[B] − kill_B);
        }
```

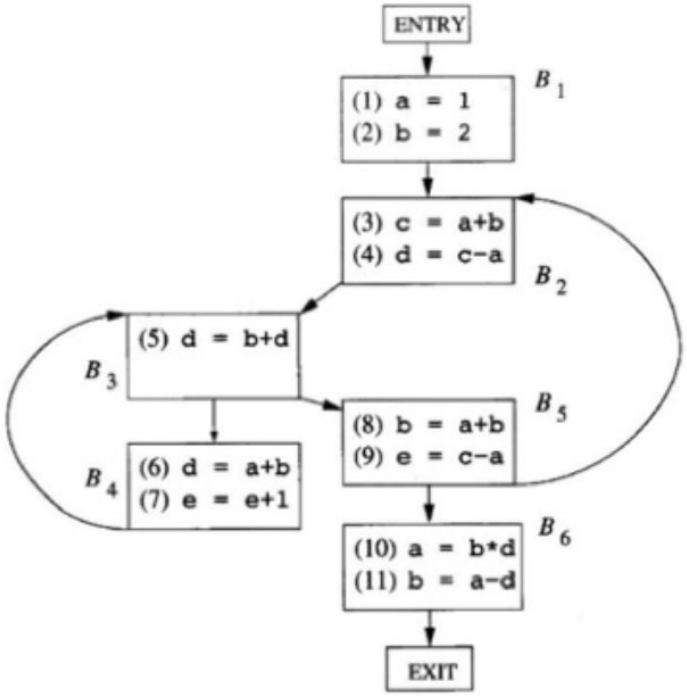
$$IN[B_2]^1 = OUT[B_1]^1 \bigcup OUT[B_5]^0 = 110\ 0000\ 0000 + 000\ 0000\ 0000 = 110\ 0000\ 0000$$

$$OUT[B_2]^1 = gen_{B2} \bigcup (IN[B_2]^1 - kill_{B2}) = 001\ 1000\ 0000 + (110\ 0000\ 0000 - 000\ 0110\ 0000) = 111\ 1000\ 0000$$

Block	$OUT[B]^0$	$IN[B]^1$	$OUT[B]^1$	$IN[B]^2$	$OUT[B]^2$
B1	0000000000	00000000000	11000000000	00000000000	11000000000
B2	0000000000	11000000000	11110000000	11101001100	11110001100
B3	0000000000	11110000000	11101000000	11110111100	11101011100
B4	0000000000	11101000000	11100110000	11101011100	11100111000
B5	0000000000	11101000000	10101001100	11101011100	10101001100
B6	0000000000	10101001100	00101000111	10101001100	00101000111

第三轮迭代的结果与第二轮相同，因此第二轮迭代后算法终止

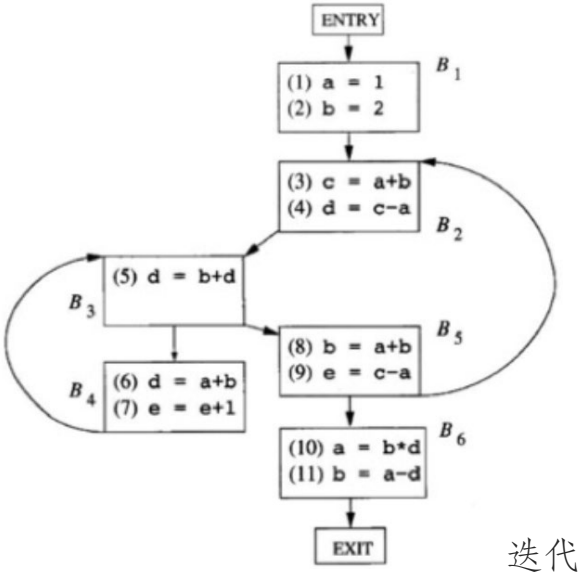
4. 对图9-10的流图，计算活跃变量分析中的def、use、IN和OUT集合。



- 活跃变量分析
  - x在p上的值是否会在某条从p出发的路径中使用
  - 变量x在p上活跃，当且仅当存在一条从p开始的路径，该路径的末端使用了x，且路径上没有对x进行覆盖
- 基本块的传递函数仍然是生成-杀死形式，但是从OUT值计算出IN值（逆向）
  - use_B：在B中先于定值被使用
  - def_B：在B中先于使用被定值

IN[EXIT] = ∅;
for (除EXIT之外的每个基本块B) IN[B] = ∅;
while (某个IN值发生了改变)
    for (除EXIT之外的每个基本块B) {
        OUT[B] = $\bigcup_{S\text{是}B\text{的一个后继}}$ IN[S];
        IN[B] = $use_B \cup$ (OUT[B] − def_B);
    }

4. 对图9-10的流图，计算活跃变量分析中的def、use、IN和OUT集合。



- 基本块的传递函数仍然是生成-杀死形式，但是从OUT值计算出IN值（逆向）
  - use_B：在B中先于定值被使用
  - def_B：在B中先于使用被定值

如果已经use了，就不生成def

```
IN[EXIT] = ∅;
for (除EXIT之外的每个基本块B) IN[B] = ∅;
while (某个IN值发生了改变)
        for (除EXIT之外的每个基本块B) {
                OUT[B] = ⋃_{S是B的一个后继} IN[S];
                IN[B] = use_B ∪ (OUT[B] − def_B);
        }
```

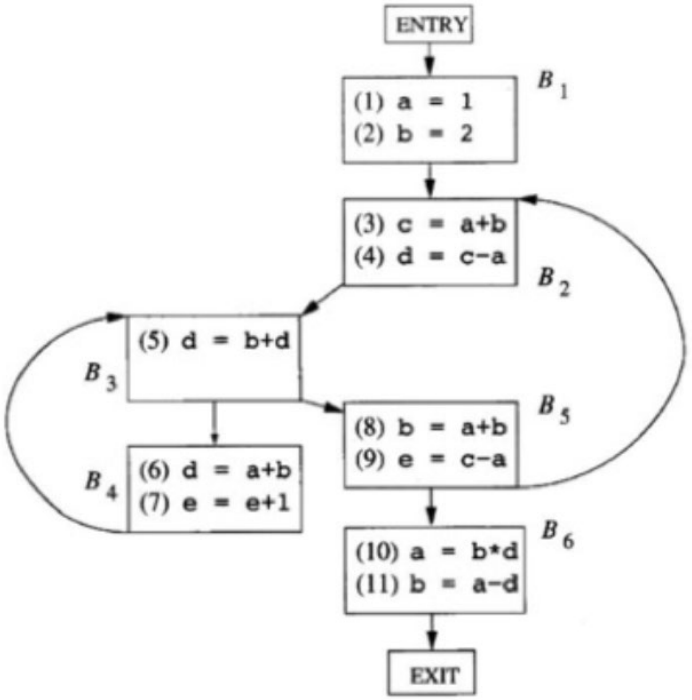
迭代开始：$IN[B6]^1=\{b,d\}\cup(\emptyset-\{a\})$

Block	*def*	*use*	$IN[B]^0$	$IN[B]^1$	$OUT[B]^1$	$IN[B]^2$	$OUT[B]^2$
B1	a,b	Ø	Ø	e	a,b,e	e	a,b,e
B2	c,d	a,b	Ø	a,b,e	a,b,c,d,e	a,b,e	a,b,c,d,e
B3	Ø	b,d	Ø	a,b,c,d,e	a,b,c,d,e	a,b,c,d,e	a,b,c,d,e
B4	d	a,b,e	Ø	a,b,e	Ø	a,b,c,e	a,b,c,d,e
B5	e	a,b,c	Ø	a,b,c,d	b,d	a,b,c,d	a,b,d,e
B6	a	b,d	Ø	b,d	Ø	b,d	Ø

第九章

5. 对图9-10的流图：

（1）计算支配关系。
（2）寻找每个结点的直接支配结点。
（3）构造支配结点树。
（4）找出该流图的一个深度优先排序。
（5）指明(4)中的前进、后退和交叉边以及树的边。
（6）这个流图是可规约的吗?
（7）计算这个流图的深度。
（8）找出这个流图的自然循环。



5. 对图9-10的流图：

（1）计算支配关系。

支配 (Dominate)

- 如果每条从入口结点到达n的路径都经过d，那么d支配n，记为d dom n

支配集D(n)表示支配n的所有节点d的集合

D(B1)={B1}
D(B2)={B1, B2}
D(B3)={B1, B2, B3}
D(B4)={B1, B2, B3, B4}
D(B5)={B1, B2, B3, B5}
D(B6)={B1, B2, B3, B5, B6}

直接支配结点

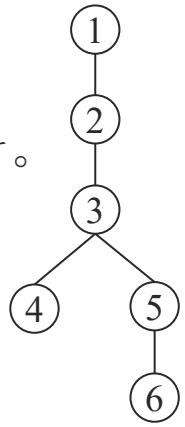
- 从入口结点到达n的任何路径（不含n）中，它是路径中最后一个支配n的结点

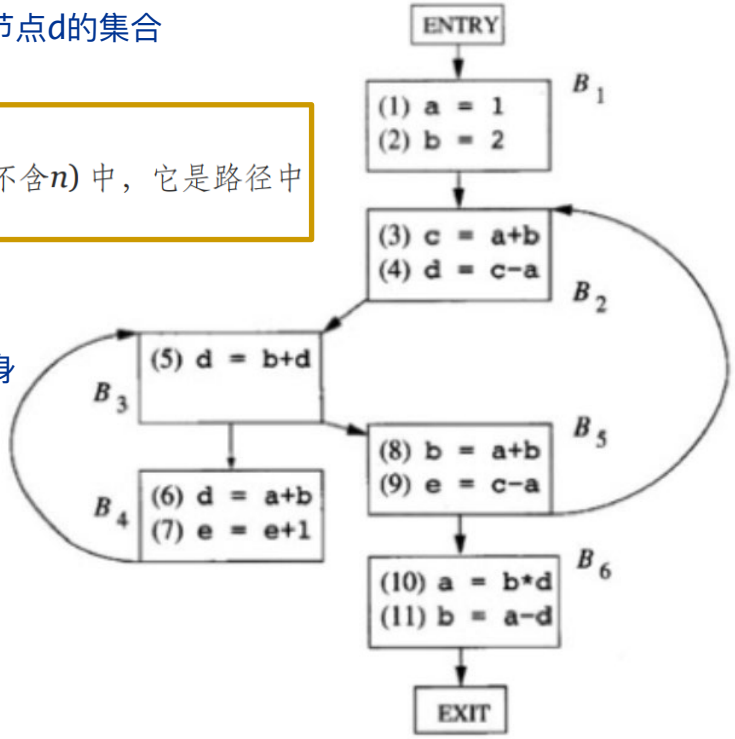
（2）寻找每个结点的直接支配结点。

B1: 无　B2: B1　B3: B2
B4: B3　B5: B3　B6: B5

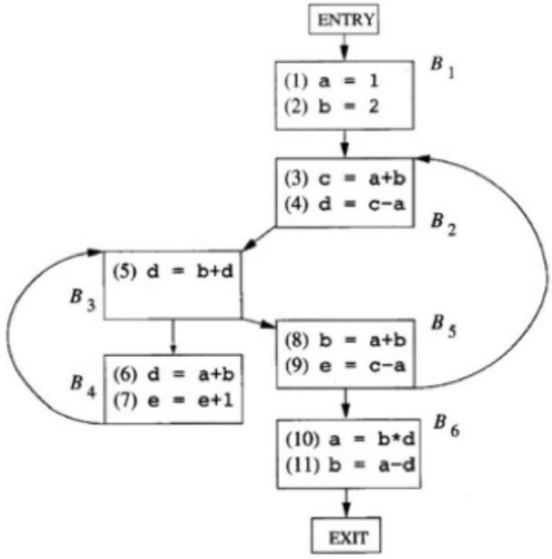
直接支配节点是不包含自身的

（3）构造支配结点树。





第九章



5. 对图9-10的流图：

（4）找出该流图的一个深度优先排序。

（5）指明(4)中的前进、后退和交叉边以及树的边。

（6）这个流图是可规约的吗?

（7）计算这个流图的深度。

（8）找出这个流图的自然循环。

- 为一个流图构造出DFST之后，流图中的边可以分为三类
  - **前进边**：从结点m到达m在DFST树中的一个真后代结点的边 (DFST中的所有边都是前进边)
  - **后退边**：从m到达m在DFST树中的某个祖先 (包括m) 的边
  - **交叉边**：边的*src*和*dest*都不是对方的祖先

- **回边**
  - 边$a \rightarrow b$，头b支配了尾a
  - 每条回边都是后退边，但不是所有后退边都是回边
- 如果一个流图的任何深度优先生成树中的所有后退边都是回边，那么该流图就是**可归约**的
  - 可归约流图的DFST的后退边集合就是回边集合
  - 不可归约流图的DFST中可能有一些后退边不是回边

5. 对图9-10的流图：

（4）找出该流图的一个深度优先排序。

B1-B2-B3-B4-B5-B6

（5）指明(4)中的前进、后退和交叉边以及树的边。

前进边: B1-B2 B2-B3 B3-B4 B3-B5 B5-B6

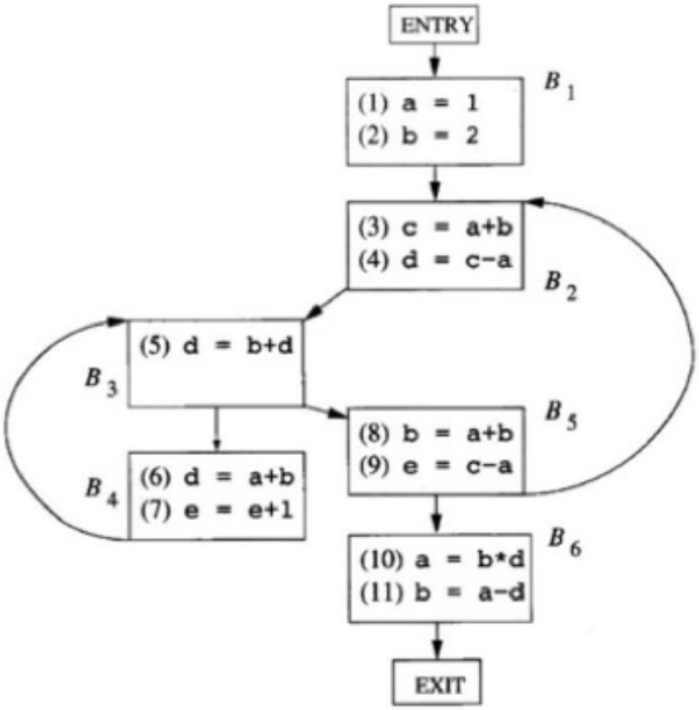
后退边: B4-B3 B5-B2

交叉边:无

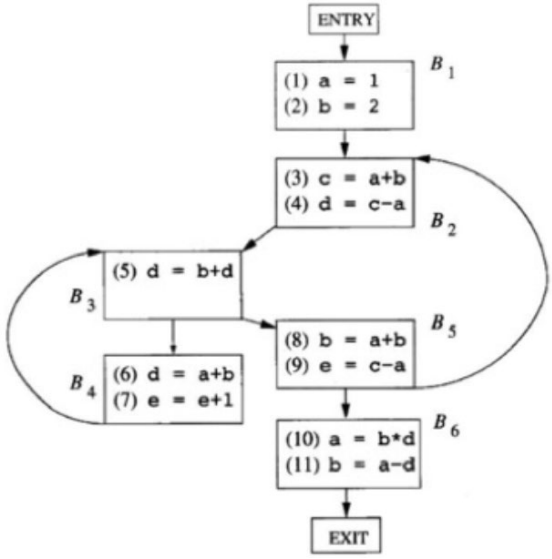
树(DFST)的边:所有前进边

（6）这个流图是可规约的吗?

是可归约的，因为后退边B4-B3, B5-B2都是回边



第九章



5. 对图9-10的流图：

（4）找出该流图的一个深度优先排序。

（5）指明(4)中的前进、后退和交叉边以及树的边。

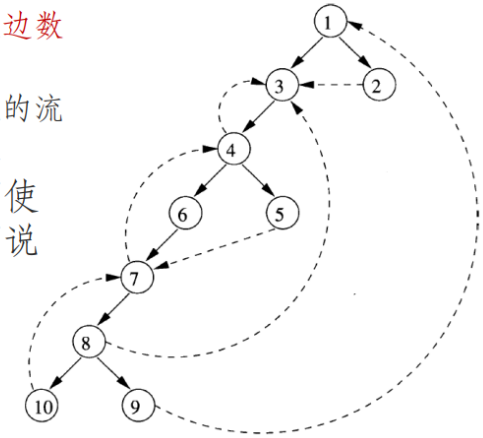
（6）这个流图是可规约的吗?

（7）计算这个流图的深度。

（8）找出这个流图的自然循环。

注意路径可以任意起点

- **流图相对于DFST的深度**
  - 各条无环路径上后退边数中的最大值
  - 不会大于直观上所说的流图中的循环嵌套深度
- 对可归约的流图，可使用回边来定义，且可说是**流图的深度**
- 右边的流图深度为3
  - 10→7→4→3



自然循环构造算法

- 输入：流图G和回边$n \rightarrow d$
- 输出：给定回边$n \rightarrow d$的自然循环中的所有结点的集合$loop$
- 方法
  - $loop = \{n, d\}$，d标记为visited
  - 从n开始，逆向对流图进行深度优先搜索，把所有访问到的结点都加入$loop$，加入$loop$的结点都标记为visited
  - 搜索过程中，不越过标记为visited的结点

5. 对图9-10的流图：

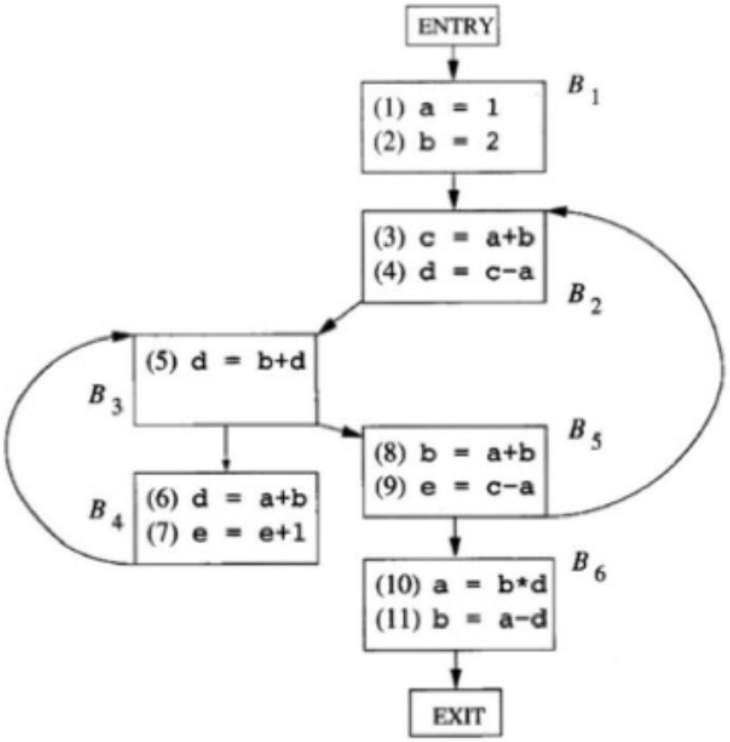
（7）计算这个流图的深度。

深度为2，存在无环路径B4-B3-B5-B2，其中有两条后退边

（8）找出这个流图的自然循环。

对于回边B4-B3，自然循环为{B4, B3}

对于回边B5-B2，自然循环为{B2, B3, B4, B5}



随堂测试

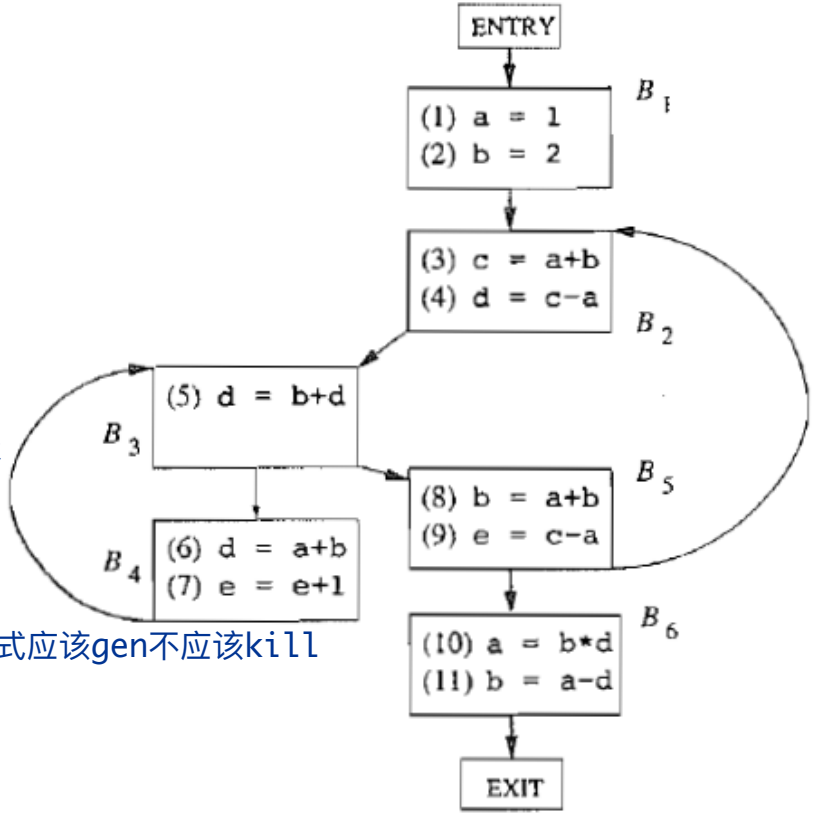
1. 龙书练习8.6.1 (4)

   4) `a[i] = b[c[i]];`

2. 龙书练习9.2.2

   对图 9-10 的流图，计算可用表达式问题中的 *e_gen*、*e_kill*、IN 和 OUT 集合。

gen的求法：基本块内部先gen后kill，最后存活下来的表达式是基本块的gen，
kill的求法：kill掉除了已求得的gen中的表达式



后于定值的表达式应该gen不应该kill