测试充分性评价

副标题

主要内容

- 控制流图、路径
- 测试充分性，测试增强
- 基于控制流的测试充分性准则
- 数据流
- 基于数据流的测试充分性准则

控制流图、路径

控制流图（Control Flow Graph）

- 基本块：一个顺序的语句序列，只有一个入口点和一个出口点
- 控制流图：描述程序中的控制流。也称流图。$G = (N, E)$，N表示图中节点的集合，每个结点对应一个基本块。E表示边的集合，每条边表示基本块之间的控制流。边(i, j)表示从基本块N_i到N_j的控制流
- 流图中没有入边的结点称为起始结点，没有出边的结点称为终止结点

路径（path）

- 考虑流图$G=(N,E)$。假设n_p,n_q,n_r,n_s是N中的结点，对任意$0<i<k$，如果有$e_i=(n_p,n_q)$且$e_{i+1}=(n_r,n_s)$，则$n_q=n_r$，则此k条边序列$(e_1,e_2,\cdots,e_k)$是一条长度为k的路径
- 对于任何$n,m\in N$，如果存在一条从n到m的路径，则称m是n的后继，或称n是m前驱。若$m\neq n$，则m是n的真后继，n是m的真前驱。如果存在$(n,m)\in E$，则称m是n的直接后继，n是m的直接前驱。结点n的直接前驱和直接后继分别记为$prev(n)$和$succ(n)$。起始结点没有前驱，终止结点没有后继。如果存在循环，结点可为自身的前驱和后继
- 流图中的一条路径，如果其首结点为起始结点，末结点为终止结点，则称该路径是完整的。

路径（二）

- 流图中的一条路径p，如果至少存在一条测试用例，其在执行程序时能够遍历p，则称p是可达的，否则称其是不可达的
- 判断指定路径p是否可达是一个不可解问题
- 条件语句会引起路径数量的指数级增长，为什么？
- 循环的存在将极大地增加路径的数量，每循环一次，路径数量至少增加1

控制流图（二）

```
 1 begin
 2    int x, y, power;
 3    float z;
 4    input (x, y);
 5    if (y<0)
 6      power=-y;
 7    else
 8      power=y;
 9    z=1;
10    while (power!=0)
11      z=z*x;
12    power=power-1;
13    }
14    if (y<0)
15      z=1/z;
16    output(z);
17 end
```

基本块	行号	入口点	出口点
1	2, 3, 4, 5	1	5
2	6	6	6
3	8	8	8
4	9	9	9
5	10	10	10
6	11, 12	11	12
7	14	14	14
8	15	15	15
9	16	16	16

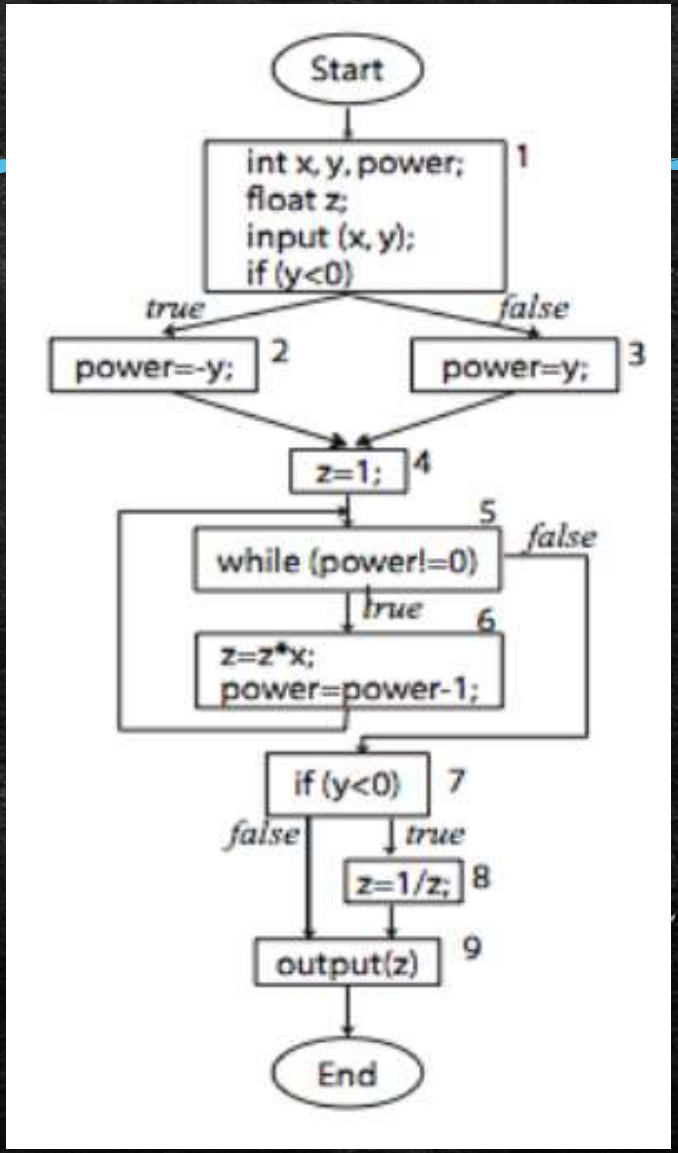
- 为什么1号基本块到第5行结束?
- 为什么第10行while语句构成一个基本块?

- 第1，17，7，13行被忽略
- 为什么第16行单独构成基本块?

控制流图（三）

N={Start, 1, 2, 3, 4, 5, 6, 7, 8, 9, End}

E={(Start,1), (1, 2), (1, 3), (2,4), (3, 4), (4, 5), (5, 6), (6, 5), (5, 7), (7, 8), (7, 9), (9, End)}

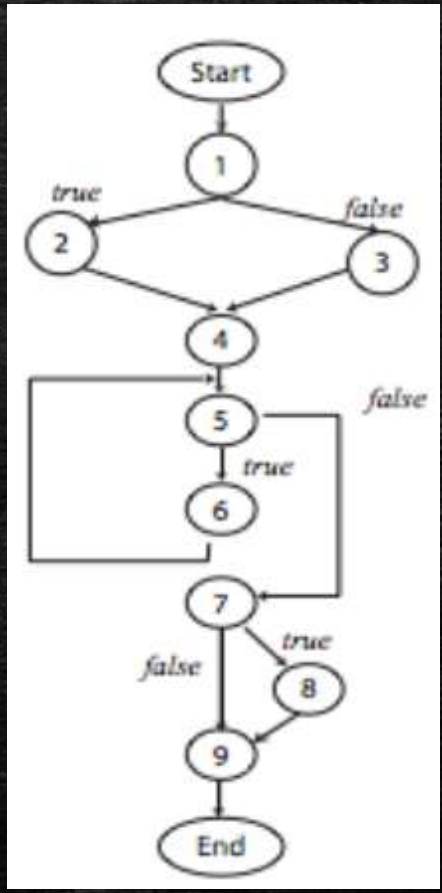


控制流图（四）

N={Start, 1, 2, 3, 4, 5, 6, 7, 8, 9, End}

E={(Start,1), (1, 2), (1, 3), (2,4), (3, 4), (4, 5), (5, 6), (6, 5), (5, 7), (7, 8), (7, 9), (9, End)}

- ((1,3),(3,4),(4,5))是一条长度为4的路径
- 但（(1,3),(3,5),(6,8)）不构成路径
- 两条完整可达路径
  - (Start,1,2,4,5,6,5,7,9,End)
  - (Start,1,3,4,5,6,5,7,8,9,End)



测试充分性、测试增强

什么是测试充分性

- 假设软件P要满足功能需求集合R，记为(P,R)。R中包含n个需求，记为$R_1,R_2,\cdots,R_n$。再假设测试集T包含了k个测试用例用于确认P是否满足R中的所有需求，并假设T中的每个用例都执行成功
- "充分性"用来度量一个给定的测试集是否能验证软件P满足其需求。当测试集满足准则C时，即认为其相对于C是充分的
- 例1：考虑编写程序P，其需求R如下：
  - R_1: 从键盘输入两个整数x和y
  - $R_{2.1}$: 当$x<y$时，求x和y之和，并在屏幕上输出
  - $R_{2.2}$: 当$x\geq y$时，求x和y之积，并在屏幕上输出

$T=\{\langle x=2,y=3\rangle\}$
相对于C充分吗?

C: 如果针对R中的每个需求r，测试集T中至少有一个用例能够证明P满足r，则认为T对于(P,R)是充分的

测试充分性的度量

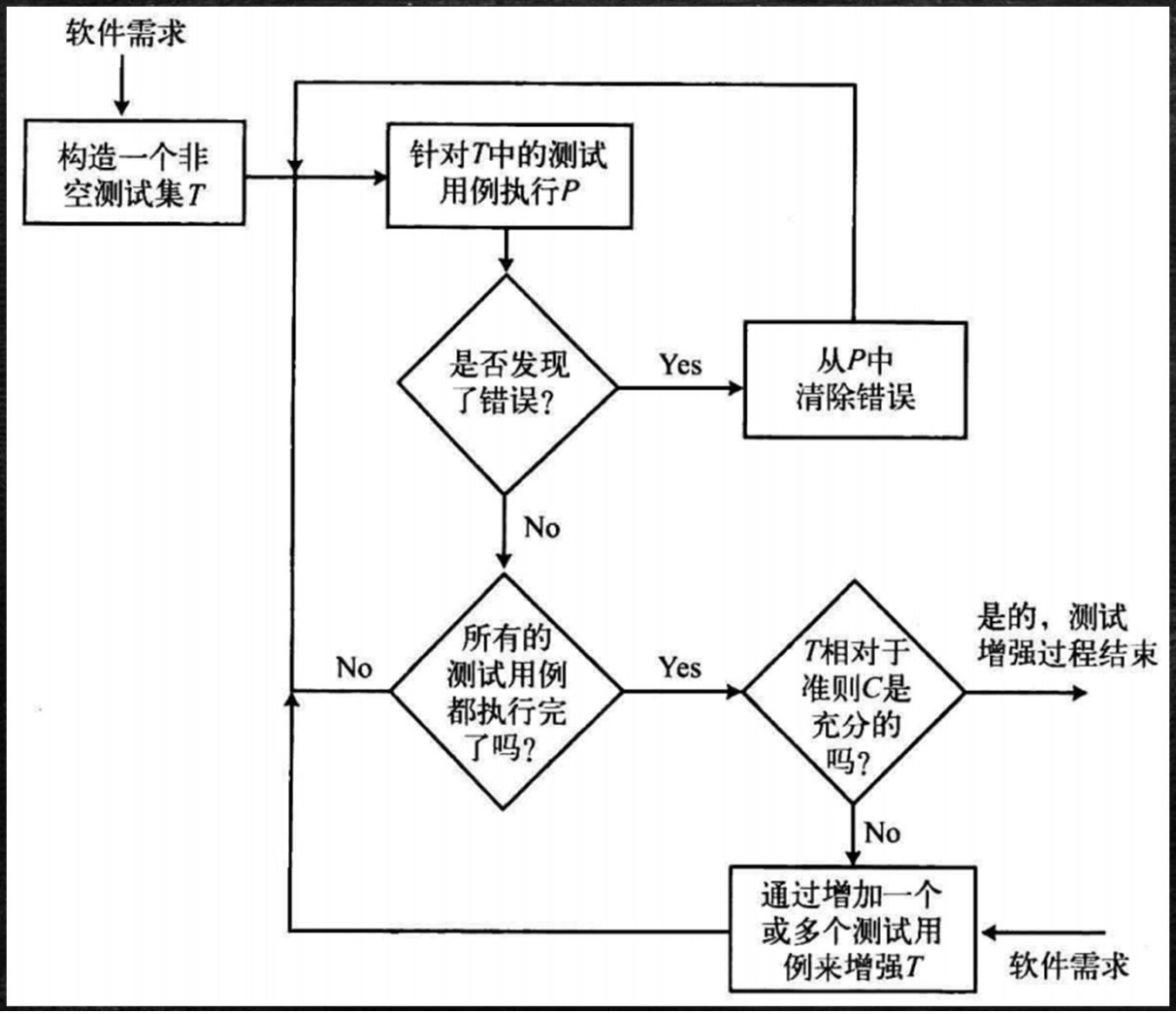
- 覆盖域：用于度量测试集充分性的有限集。每个测试准则都可导出一个覆盖域
- 假设要度量测试集T的充分性。给定覆盖域C_e，它有$n \geq 0$个元素。如果C_e中的每个元素e，在T中都至少有一个测试用例能够测试它，则称T覆盖e。如果T能覆盖C_e中的所有元素，称T对于C_e是充分的。如果T只能覆盖C_e中的$k \leq n$个元素，称T对于C_e是不充分的。比率k/n代表T对于C的充分度，也称T对于C, R, P的覆盖率
- 例2：考虑例1中的P, R, C, T. 令$C_e = \{R_1, R_{2.1}, R_{2.2}\}$. 此时$T$覆盖$R_1$和$R_{2.1}$，但没覆盖$R_{2.2}$。因此$T$对于$C_e$是不充分的，覆盖率为$\frac{2}{3}$
- 例3：例1中的P有两条路径p_1和p_2，分别对应条件$x < y$和$x \geq y$. T对于路径覆盖准则不充分，其只能覆盖p_1，因此覆盖率为0.5

路径覆盖准则：如果程序P中的每条路径都被遍历至少一次，则认为测试集T针对(P, R)是充分的

测试增强

- 虽然测试集对于某些测试准则是充分的，但不能保证程序没有错误。不充分的测试集意味着测试不足。此时，如果能提高测试集的覆盖率，就能提高发现错误的几率，即测试增强
- 例4：例3中的T对于C的覆盖率只有0.5，如果往T中添加测试用例$\langle x = 3, y = 1\rangle$，则扩充后的测试集$T'$能够覆盖$\{p_1, p_2\}$，覆盖率为1
- 通过度量充分性来增强测试
  - 如果测试集T对于准则C是充分的，则无需增强，退出。否则进入下一步
  - 对每个未被覆盖的$e \in C_e$，执行以下步骤
    - 构造测试用例t，它能够覆盖e，把t加入到T中
    - 针对t执行P，如果执行不正确，则修改P，重复此步骤直到P执行正确

测试增强


软件需求
构造一个非空测试集T
针对T中的测试用例执行P
是否发现了错误?
Yes
从P中清除错误
No
所有的测试用例都执行完了吗?
No
Yes
T相对于准则C是充分的吗?
是的，测试增强过程结束
No
通过增加一个或多个测试用例来增强T
软件需求

测试增强（二）

- 例5： *P*为计算x^y的程序。x和y为整数。若$y < 0$，输出错误信息
- 充分性准则C：如果测试集T对x和y中的每一个，至少经过一次等于0和不等于0，则T是充分的$C_e = \{x = 0, y = 0, x \neq 0, y \neq 0\}$
- $T = \{\langle x = 0, y = 1\rangle, \langle x = 1, y = 0\rangle\}$相对于$C$是充分的

```
1    begin
2      int x, y;
3      int product, count;
4      input (x, y);
5      if(y≥0) {
6        product=1; count=y;
7        while(count>0) {
8          product=product*x;
9          count=count-1;
10       }
11     output(product);
12     }
13     else
14       output ( "Input does not match
15   end
```

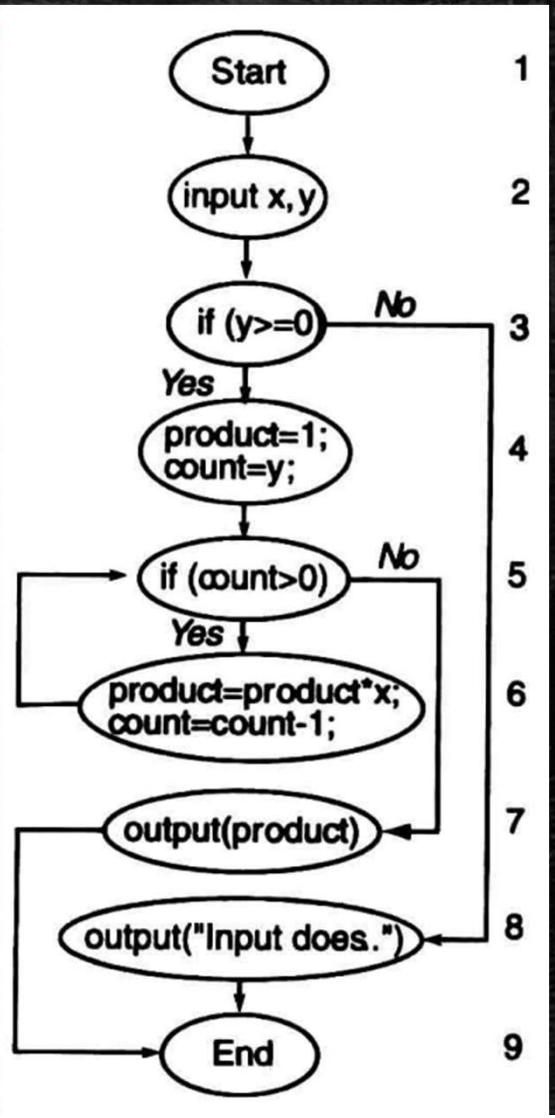
测试增强（三）

- 例6：将例5中的测试准则替换为路径覆盖准则。由于while语句的存在，流图中路径的个数具有不确定性，其依赖y的值。因此其路径个数可能会非常大，此时，无法确定路径覆盖准则的覆盖域。此时，可将路径覆盖准则简化为C'：

C': 如果测试集T测试了所有路径，则它是充分的。若程序包含循环，则只要T遍历过循环体0次和1次即可

- 根据C'，可导出$C'_e = \{p_1, p_2, p_3\}$。其中，
  - $p_1 = (1,2,3,4,5,7,9)$
  - $p_2 = (1,2,3,4,5,6,5,7,9)$
  - $p_3 = (1,2,3,8,9)$

- 测试增强：$T' = T \cup \{\langle x = 5, y = -1\rangle\}$。 ?



基于控制流的测试充分性准则

语句覆盖和块覆盖

- 语句覆盖：测试集T针对(P,R)的语句覆盖率计算为$\frac{|S_c|}{|S_e|-|S_i|}$。其中$S_c$是所有被覆盖语句的集合，$S_i$是所有不可达语句的集合，$S_e$是软件中所有语句的集合，即语句覆盖域。如果$T$针对 (P,R)的语句覆盖率为1，则称T相对于语句覆盖准则是充分的

- 块覆盖：测试集T针对(P,R)的语句覆盖率计算为$\frac{|B_c|}{|B_e|-|B_i|}$。其中$B_c$是所有被覆盖基本块的集合，$B_i$是所有不可达基本块的集合，$B_e$是软件中所有基本块的集合，即块覆盖域。如果$T$针对 (P,R)的块覆盖率为1，则称T相对于块覆盖准则是充分的

- 上述定义中的不可达是指无效路径中的语句和基本块

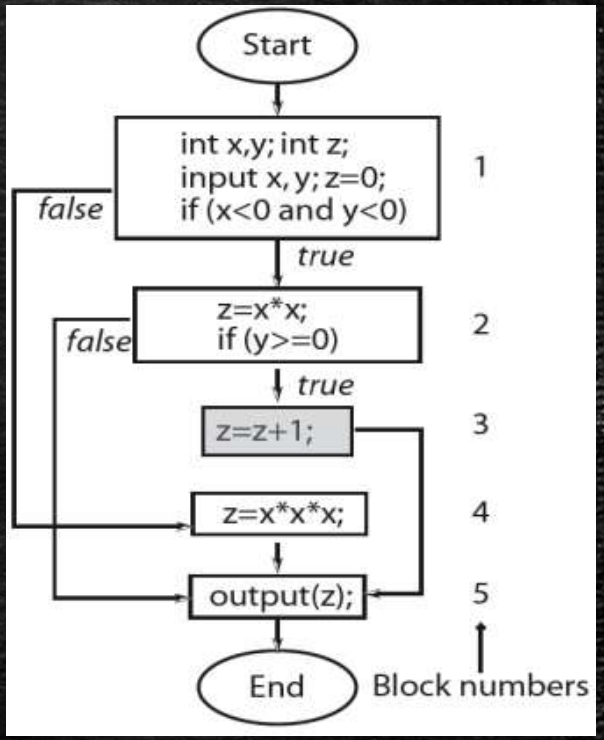
语句覆盖和块覆盖（二）

- $S_e = \{2,3,4,5,6,7,7b,9,10\}$
- $T_1 = \{t_1, t_2\}$，$t_1: \langle x = -1, y = -1 \rangle$，$t_2: \langle x = 1, y = 1 \rangle$
- t_1覆盖2,3,4,5,6,7,10，t_2覆盖2,3,4,5,9,10
- 语句$7b$失效的元素
- $|S_e| = 9, |S_i| = 1, |S_c| = 8$
- T的语句覆盖率为$\frac{8}{9-1} = 1$，其在语句覆盖准则下是充分的

```
1     begin
2       int x, y;
3       int z;
4       input (x, y); z=0;
5       if(x<0 and y<0){
6         z=x*x;
7         if(y≥ 0) z=z+1;
8         }
9       else z=x*x*x;
10      output(z);
11      }
12    end
```

语句覆盖和块覆盖（三）

- $B_e = \{1,2,3,4,5\}$
- $T = \{t_1, t_2, t_3\}$
  - t_1: $\langle x = -1, y = -1 \rangle$，覆盖块1,2,5
  - t_2: $\langle x = -3, y = -1 \rangle$，覆盖块1,2,5
  - t_3: $\langle x = -1, y = -1 \rangle$，覆盖块1,2,5
- $|B_e| = 5, |B_c| = 3, |B_i| = 1$
- 块覆盖率为$\frac{3}{5-1} = 0.75$，因此T对于块覆盖率是不充分的
- 测试增强：令$T' = T \cup \{\langle x = 1, y = 1 \rangle\}$，$T'$对于块覆盖率是充分的



条件与判定

- 任何计算结果为true或false的表达式都是一个条件
- 简单条件：由变量和至多一个关系运算符构成，如布尔变量A，$x > y$
- 复合条件：两个或多个简单条件经一个或多个布尔运算符连接而成，如$A||x > y$
- 作为判定的条件：包含在if，while中的条件称为判定
- 一个判定有三种输出：真、假和未定义。每个真或假的输出都对应一个路径
  - bool foo(){ while(true) {} }
    if(foo()) {...} //因为foo()函数永远不会返回，此处的条件是未定义的
- 后面不考虑未定义情况

判定覆盖

- 判定覆盖：测试集T针对(P,R)的判定覆盖率计算为$\frac{|D_c|}{|D_e|-|D_i|}$。其中$D_c$是所有被覆盖判定的集合，$D_i$是所有不可达判定的集合，$D_e$是软件中所有判定的集合，即判定覆盖域。如果$T$针对$(P,R)$的判定覆盖率为1，则称$T$相对于判定覆盖准则是充分的
- 需求：当x<0时，先转换为正数，然后调用foo-1生成z；如果x>=0，调用foo-2生成z
- 程序如右图所示。有错误。当x>=0时没调用foo-2
- $T=\{t_1:\langle x=-5\rangle\}$. 容易验证$T$对于语句和块覆盖都是充分的
- T对于判定覆盖是不充分的，它不能检测出程序中的错误
- 测试增强：令$T'=T\cup\{\langle x=3\rangle\}$. T'对于判定覆盖是充分的，它能够发现程序中的错误

```
1    begin
2      int x, z;
3      input (x);
4      if(x<0)
5        z=-x;
6      z=foo-1(x);
7      output(z);
8    end
```

条件覆盖

- 条件覆盖：测试集T针对(P,R)的条件覆盖率计算为$\frac{|C_c|}{|C_e|-|C_i|}$。其中$C_c$是所有被覆盖简单条件的集合，$C_i$是所有不可达简单条件的集合，$C_e$是软件中所有简单条件的集合，即条件覆盖域。如果$T$针对$(P,R)$的条件覆盖率为1，则称$T$相对于条件覆盖准则是充分的

- 条件既可以是简单条件，也可以是复合条件。如果简单条件的真和假都被取到，称简单条件被覆盖。如果复合条件中的每个简单条件都被覆盖，则称复合条件被覆盖
- 判定覆盖不考虑复合条件

条件覆盖（二）

- $T = \{t_1, t_2\}. C_e = \{x < 0, y < 0\}. C_i = \emptyset$
  - $t_1 = \langle x = -3, y = -2 \rangle$
  - $t_2 = \langle x = -4, y = 2 \rangle$
- T对于语句、块、判定覆盖是充分的，但 $C_c = \{y < 0\}$. 条件覆盖率=1/(2-0)=0.5
- 测试增强：$T' = T \cup \{\langle x = 3, y = 4 \rangle\}$

```
1     begin
2       int x, y, z;
3       input (x, y);
4       if(x<0 and y<0)
5         z=foo1(x,y);
6       else
7         z=foo2(x,y);
8       output(z);
9     end
```

x< 0	y< 0	Output (z)
true	true	foo1(x,y)
true	false	foo2(x,y)
false	true	foo2(x,y)
false	false	foo1(x,y)

条件/判定覆盖

- 条件/判定覆盖：测试集T针对(P,R)的条件覆盖率计算为$\frac{|C_c|+|D_c|}{(|C_e|-|C_i|)+(|D_e|-|D_i|)}$。其中$C_c, D_c$是所有被覆盖简单条件和判定的集合，$C_i, D_i$是所有不可达简单条件和判定的集合，$C_e, D_e$是软件中所有简单条件和判定的集合。如果$T$针对$(P,R)$的条件覆盖率为1，则称$T$相对于条件/判定覆盖准则是充分的

- 必要性：判定覆盖不需要考虑复合条件，而条件覆盖不需要考虑判定判定的真/假路径
- $T_1=\{t_1,t_2\}, T_2=\{t_1,t_3\}, T=\{t_1,t_4\}$
  - $t_1=\langle x=-3, y=2\rangle$，$t_2=\langle x=4, y=2\rangle$
  - $t_3=\langle x=4, y=-2\rangle$，$t_4=\langle x=4, y=2\rangle$
- T_1对于条件覆盖是充分的，但对于判定覆盖不充分
- T_2对于判定覆盖是充分的，但对于条件覆盖是不充分的
- T对于条件/判定覆盖充分的

```
1    begin
2      int x, y, z;
3      input (x, y);
4      if(x<0 and y<0)
5        z=foo1(x,y);
6      else
7        z=foo2(x,y);
8      output(z);
9    end
```

多重条件覆盖

- 多重条件覆盖：测试集T针对(P,R)的多重条件覆盖率计算为$\frac{|C_c|}{|C_e|-|C_i|}$。其中$C_c$是所有被覆盖简单条件组合的集合，$C_i$是所有不可达简单条件组合的集合，$C_e$是软件中所有简单条件组合的集合。如果$T$针对$(P,R)$的条件覆盖率为1，则称$T$相对于多重条件覆盖准则是充分的

MC/DC覆盖准则

- 当满足以下条件时，称测试集T对于程序P针对MC/DC覆盖是充分的
  - P的每一个基本块都被覆盖了（基本块覆盖）
  - P的每一个简单条件的true和false都被覆盖（条件覆盖）
  - P的每个判定的分支都被覆盖（判定覆盖）
  - P中复合条件C中的每个简单条件对C的输出结果的影响是独立的（MC覆盖）

数据流概念

数据流概念

- 确保顺序/分支/循环三种结构被完全测试是基于控制流测试充分性评价的最高目标
- 即使完全测试了这三种结构，也不能检测出程序中的所有错误
- 基于数据流的测试充分性评价主要关注软件中的数据定义和使用，可用来改进针对控制流的测试充分性评价的测试集
- 例：$T = \{t_1, t_2\}, t_1 = \langle x = 0, y = 0, z = 0.0\rangle, t_2 = \langle x = 1, y = 1, z = 1.0\rangle$对于程序是条件/判定覆盖是充分的。但是它无法发现第8行的错误（被零除），其正确条件应为$y \neq 0\ and\ x \neq 0$

```
1    begin
2      int x, y; float z;
3      input (x, y);
4      z=0;
5      if (x! =0)
6        z=z+y;
7      else z=z-y;
8      if (y! =0) ← This
9        z=z/x;
10     else z=z*x;
11     output(z);
12   end
```

如下表所示，结合数据流（此处为z的每个定义-使用对都被执行）可发现错误

Test	x	y	z	*def-use pairs covered
t_1	0	0	0.0	(4, 7), (7,10)
t_2	1	1	1.0	(4, 6), (6,9)
t_3	0	1	0.0	(4, 7), (7,9)
t_4	1	0	1.0	(4, 6), (6,10)

*In the pair (l_1, l_2), z is defined in l_1 and used in line l_2.

定义（def）与使用（use）

- 变量通过赋值语句进行定义，并在表达式中进行使用
- 变量的声明也被视作变量的定义
- 输入函数也被用来视作对变量的定义
- 参数x的传值调用被认为是对x的一次使用，而其传址调用被认为是对x的一次定义和使用

```
x = y + z; //定义x，使用y和z
x = x + y; //定义x，使用x和y
int x, y, A[10]; //定义了x, y, A
printf("%d", x); //使用x

scanf(“%d %d", &x, &y); //定义x和y

z = &x; //定义z
y = z + 1; //定义y，使用z
*z = 25; //通过z定义了x
y = *z + 1; //定义了y，通过z使用了x

int A[10]; //定义了变量A
A[i] = x + y; //定义了A中的元素还是整个A?
```

c-use和p-use

- 如果变量被用在表达式，输出语句中，或被当作参数传递给函数，或者被用在下标表达式中，都称为此变量的c-use，其中c表示计算
- 如果变量被用在分支语句的条件表达式中（如if和while语句），则称为变量的p-use，其中p表示谓词
- 有的情况下，不容易区分p-use和c-use

- c-use
  - z = x + 1; //表达式
  - A[x – 1] = B[2]; //下标
  - foo(x * x); //参数传递
  - output(x); //参数传递
- p-use
  - if (z > 0) { output(x); }
  - while (z > x) { … }
- c-use or p-use?
  - if (A[x + 1] > 0) { output(x); }

全局和局部的定义和使用

- 变量可能在同一个基本块中被定义、使用和重定义。考虑以下基本块

```
- p = y + z;   // p的局部定义，因为此定义被第二个定义覆盖，没有超出此基本块
               // y和z的全局使用，因为其定义没有出现在此基本块中
  x = p + 1;   // x的全局定义。它的值可被后续的基本块使用
               // p的局部使用，因为其定义出现在基本块中
  p = z * z;   // p的全局定义，其值可被后续的基本块使用
               // z的全局使用，因为其定义没有出现在此基本块中
```

- 全局和局部是相对基本块而言，与传统的全局变量和局部变量有所区别

数据流图（Data Flow Graph）

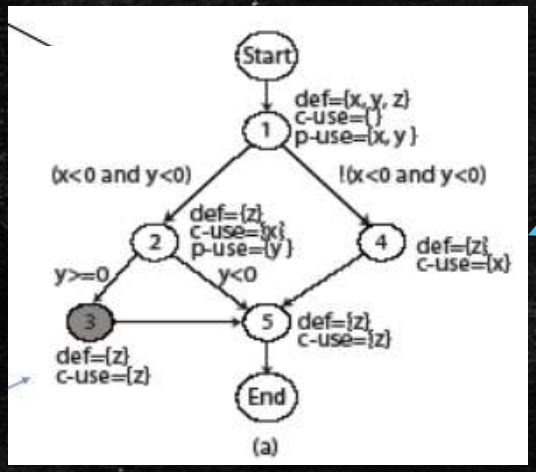
- 也称def-use图。勾画了程序中变量在不同基本块间的定义流。其中的结点、边、路径的概念与控制流图保持一致
- 数据流图可由控制流图导出。
- 假设$G = (N, G)$表示程序P的CFG，其包含k个基本块，即$b_1, b_2, \cdots, b_k$。用def_i表示定义在b_i中的变量的集合，用$c - use_i$表示在b_i中c-use变量的集合，$p - use_i$表示 b_i中p-use变量的集合。变量x在b_i中的定义记为$d_i(x)$，其在b_i中的使用记为$u_i(x)$. 此处只关心全局的定义和使用

- 对于右侧基本块b_i而言，有
  - $def_i = \{p, A\}$
  - $c - use_i = \{y, z, p, q, number, x, i\}$
  - $p - \mathrm{use_i} = \{\mathrm{x}, \mathrm{y}\}$

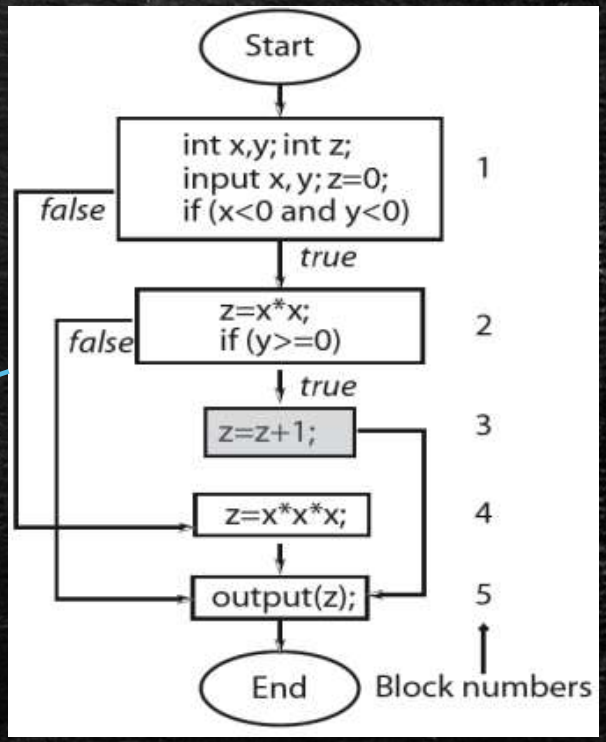
```
p = y + z;
foo(p + q, number);
A[i] = x + 1;
if(x > y) {...}
```

数据流图

- 根据程序P及其CFG构造DFG的过程
  - 计算P中每个基本块b_i的$def_i, c-use_i, p-use_i$
  - 将结点集N中的每个结点i与$def_i, c-use_i, p-use_i$关联起来
  - 针对每个具有非空p-use集并且在条件C处结束的结点i，如果条件C为真时执行的是边(i,j)，C为假时执行的边为(i,k)，分别将边$(i,j),(i,k)$与C和!C关联起来

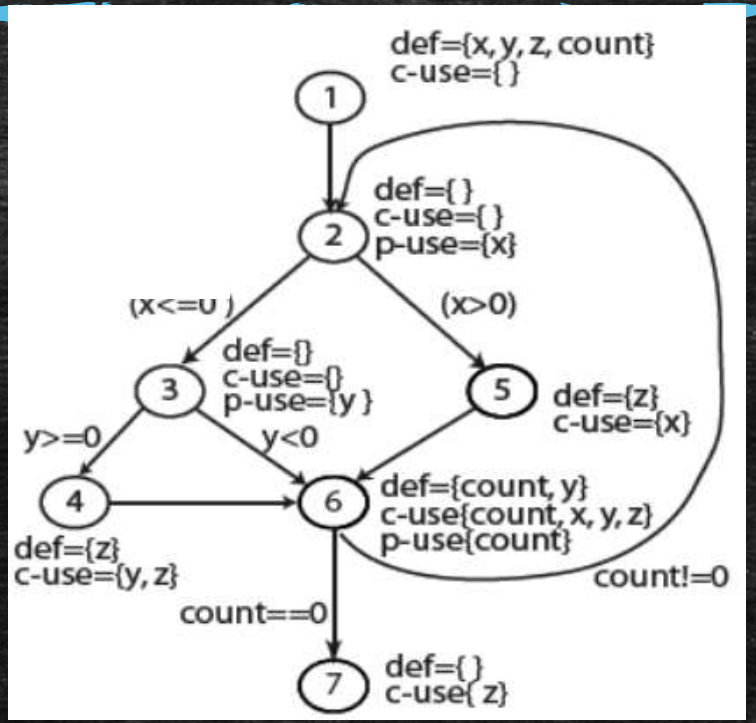


Node (or Block)	def	c-use	p-use
1	{x, y, z}	{ }	{x, y}
2	{z}	{x}	{y}
3	{z}	{z}	{ }
4	{z}	{x}	{ }
5	{ }	{z}	{ }



def-clear路径

假设变量x在结点i中定义，在结点j中使用。考虑路径$p = (i, n_1, n_2, \cdots, n_k, j), k \geq 0$. 该路径开始于结点$i$，结束于结点$j$，并且结点$i, j$在子路径$n_1, n_2, \cdots, n_k$中没有出现，并且如果变量$x$在子路径$n_1, n_2, \cdots, n_k$中没有被重定义，称$p$是$x$的def-clear路径。也称$x$在结点$i$处的定义$d_i(x)$在结点$j$处是活跃的




```
1    begin
2      float x, y, z=0.0;
3      int count;
4      input (x, y, count);
5      do {
6        if (x≤0) {
7          if (y≥0) {
8            z=y*z+1;
9          }
10       }
11       else{
12         z=1/x;
13       }
14       y=x*y+z
15       count=count-1
16     while (count>0)
17     output (z);
18   end
```

路径$p = \{1,2,5,6\}$对于$d_1(x), u_6(x), d_1(count), u_6(count)$是def-clear路径。$p$对于$d_1(z), u_6(z)$不是def-clear路径，$d_1(x), d_1(count)$在结点6处活跃。路径$q = \{6,2,5,6\}$如何？其他路径呢？

Node	Lines
1	1, 2, 3, 4
2	5, 6
3	7
4	8, 9, 10
5	11, 12, 13
6	14, 15, 16
7	17, 18

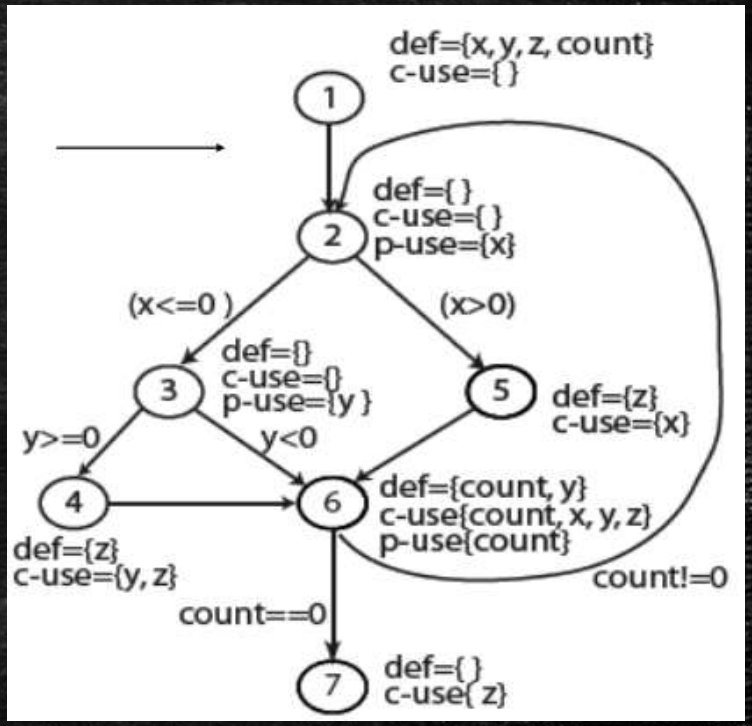
def-use对

- 勾画了变量的一次特定的定义和使用。
  - 变量x在第4行的定义和第9行的使用构成了一个def-use对
- 两种类型的def-use对
  - dcu：定义及其c-use构成的def-use对
  - dpu：定义及其p-use构成的def-use对
- $dcu(d_i(x))$表示所有结点j的集合。结点j满足：存在$u_j(x)$，存在一条从结点i到结点j的def-clear路径。$dcu(d_i(x))$也记为$dcu(x,i)$
- 当$u_k(x)$出现在判定条件中时，$dpu(d_i(x))$表示边(k,l)的集合。边(k,l)满足：存在一条从结点i到边(k,l)的针对x的def-clear路径。也可记为$dpu(x,i)$

```
1    begin
2      int x, y;
3      int z;
4      input (x, y); z=0;
5      if(x<0 and y<0){
6        z=x*x;
7        if(y≥ 0) z=z+1;
8        }
9      else z=x*x*x;
10     output(z);
11     }
12   end
```

def-use对：例子

Variable (v)	Defined in node (n)	dcu (v, n)	dpu (v, n)
x	1	{5, 6}	{(2, 3), (2, 5)}
y	1	{4, 6}	{(3, 4), (3, 6)}
y	6	{4, 6}	{(3, 4), (3, 6)}
z	1	{4, 6, 7}	{ }
z	4	{4, 6, 7 }	{ }
z	5	{4, 6, 7}	{ }
count	1	{6}	{(6, 2), (6, 7) }
count	6	{6}	{(6, 2), (6, 7) }



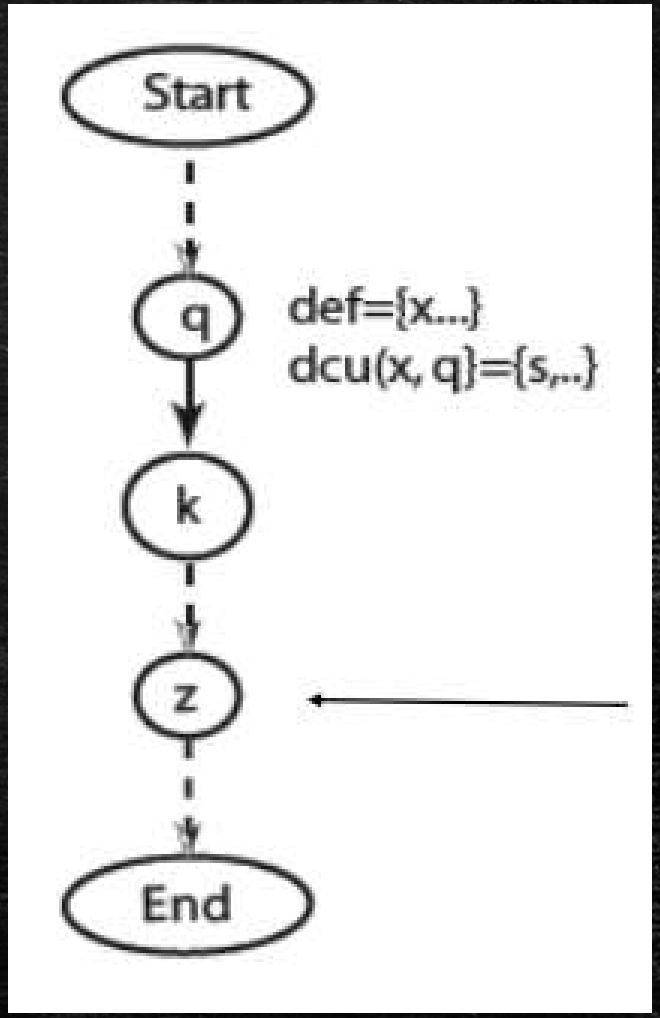
$$n = 4, d_1 = 1, d_2 = 2, d_3 = 3, d_4 = 2, CU = 17, PU = 10$$

基于数据流的测试充分性准则

- 假设程序P的数据流图包含k个结点，n_1, n_k分别表示开始和结束结点，当测试用例t执行程序P时，如果遍历了完整路径$(n_{i_1}, n_{i_2}, \cdots, n_{i_{m-1}}, n_{i_m})$，则称程序$P$数据流图的结点$s = n_{i_j}, 1 \leq j \leq m, m \leq k$被$t$覆盖了。同理，此完整路径中包含的边$(r, s), r = n_{i_j}, n_{i_{j+1}}, 1 \leq j \leq m-1$也被$t$覆盖了。
- 设CU, PU分别表示程序P中定义的所有变量的$c-use$和$p-use$总数目。设$v = \{v_1, \cdots, v_n\}$表示程序P中定义所有变量的集合。$d_i, 1 \leq i \leq n$表示变量v_i的定义次数，显然有$0 \leq d_i \leq |N|$，N为数据流图结点集合。则有$CU = \sum_{i=1}^{n} \sum_{j=1}^{d_i} |dcu(v_i, n_j)|$，$PU = \sum_{i=1}^{n} \sum_{j=1}^{d_i} |dpu(v_i, n_j)|$，其中$n_j$是变量$v_i$第$j$次被定义时所在的结点
- 例子

c-use覆盖

- 设z是$dcu(x,q)$中的一个结点，即结点z包含在结点q处定义的变量x的一个$c-use$
- 假设当测试用例t_c执行程序P，遍历了完整路径$p=(n_1,n_{i_1},\cdots,n_{i_l},n_{i_{l+1}},\cdots,n_{i_m},n_{i_{m+1}},\cdots,n_k)$，其中$2\le i_j<k,1\le j\le k$. 如果$q=n_{i_l},s=n_{i_m}$，$(n_{i_l},n_{i_{l+1}},\cdots,n_{i_m})$是一个从$q$到$z$的def-use路径，则称变量$x$的该c-use被覆盖。如果$dcu(x,q)$中的每个结点在程序$P$的一次或多次执行中都被 覆盖了，则称变量$x$的所有$c-use$被覆盖。如果程序$P$中所有变量的所有c-use都被覆盖，则称程序中所有$c-use$被覆盖。

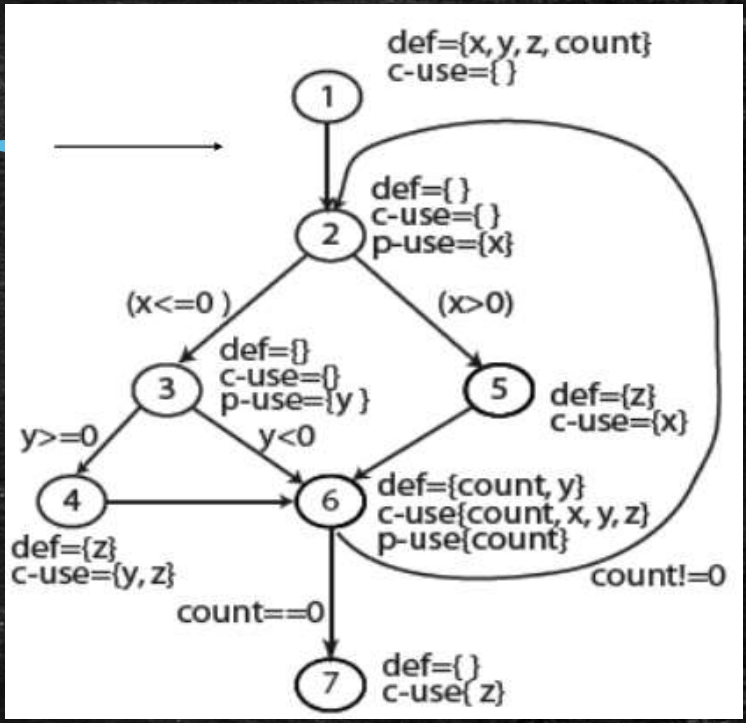


c-use覆盖准则

- 测试集T针对(P,R)的$c-use$覆盖率为：$\frac{CU_c}{CU-CU_i}$，其中CU为程序P中所有变量的$c-use$总数，CU_c为覆盖的$c-use$数，CU_i是无效的$c-use$数。如果T对(P,R)的$c-use$覆盖率为1，则称T相对于$c-use$覆盖准则是充分的。

c-use覆盖准则

- 设计一个测试用例t，使其能覆盖$dcu(z,5)$中的结点6，即覆盖在结点5所定义的变量z在结点6的c-use。
- $t=\langle x=5, y=-1, count=1\rangle$
- t同时覆盖结点7，但不覆盖4
- 右图中共有12个c-use（只考虑最小c-use集），t只覆盖了2个。覆盖率为1/6=0.167



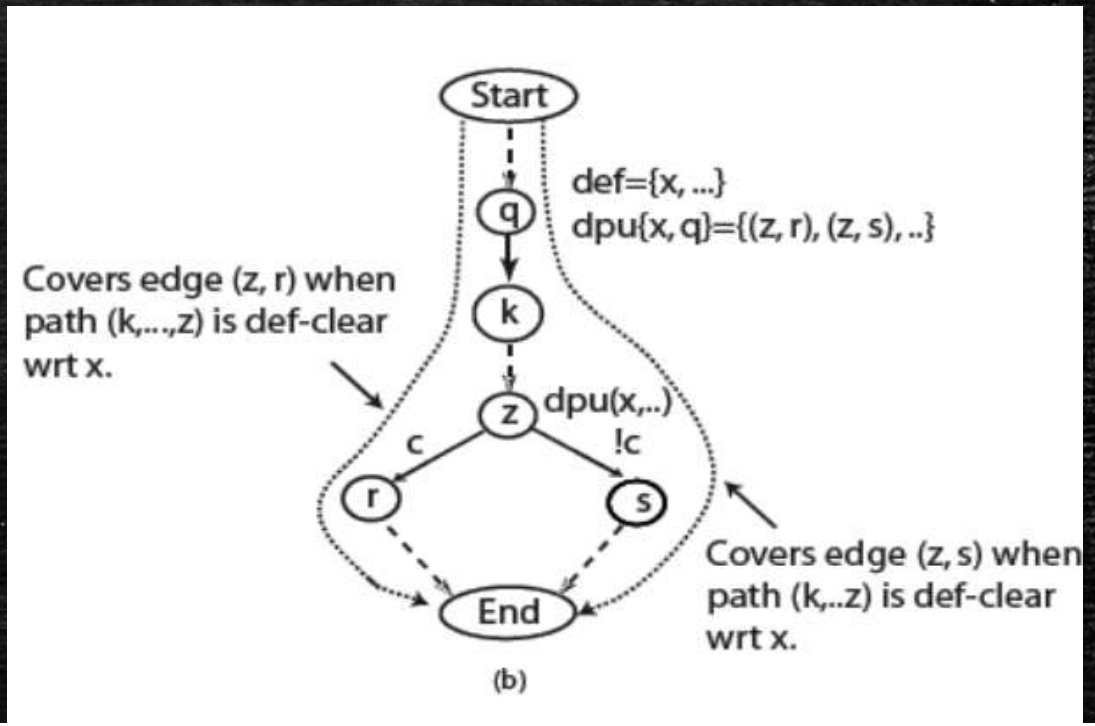
变量（v）	定义所在的结点（n）	dcu(v,n)	dpu(v,n)
y	6	{4, 6}	{(3, 4), (3, 6)}
z	1	{4, 6, 7}	{ }
z	4	{4, 6, 7}	{ }
z	5	{4, 6, 7}	{ }
count	6	{6}	{(6, 2), (6, 7)}

p-use覆盖

- 设$(z,r),(z,s)$是$dpu(x,q)$中的两条边，即结点z包含变量x的一个$p-use$，x是在结点q中定义的。针对测试用例t_p执行程序P，遍历了完整路径$p=(n_1,n_{i_1},\cdots,n_{i_l},n_{i_{l+1}},\cdots,n_{i_m},n_{i_{m+1}},\cdots,n_k)$，其中$2\le i_j<k,1\le j\le k$．如果$q=n_{i_l},z=n_{i_m},r=n_{i_{m+1}}$并且$(n_{i_l},n_{i_{l+1}},\cdots,n_{i_m},n_{i_{m+1}})$对$x$而言是一个$def-clear$路径，则称结点$q$所定义变量$x$在结点$z$的$p-use$的边$(z,r)$被覆盖。如果$dcu(x,q)$中的每条边在程序$P$的一次或多次执行中都被覆盖了，则称变量$x$的所有$p-use$被覆盖。如果程序$P$中所有变量的所有p-use都被覆盖，则称程序中所有$c-use$被覆盖。

p-use覆盖准则

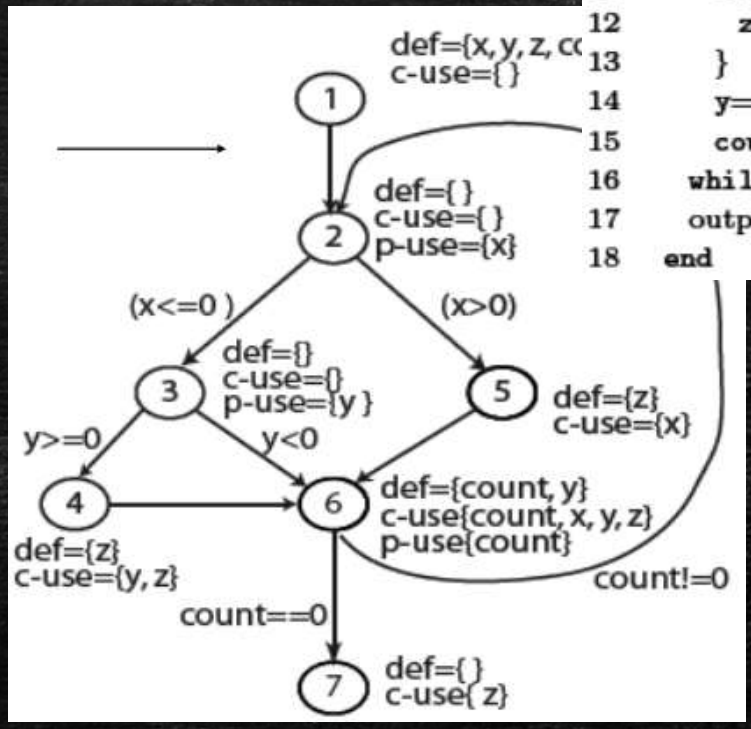
- 测试集T针对(P, R)的$\mathrm{p}-use$覆盖率为：$\frac{PU_c}{PU-PU_i}$，其中PU为程序P中所有变量的$p-use$总数，PU_c为覆盖的$p-use$数，PU_i是无效的$p-use$数。如果T对(P, R)的$p-use$覆盖率为1，则称T相对于$p-use$覆盖准则 是充分的。



p-use覆盖准则

- 设计一个测试用例t，使其能覆盖在结点6定义的变量y在结点3的$p-use$
- $t=\langle x=-2, y=-1, count=3\rangle$
- 第一次执行只覆盖了$y<0$那条边，因此对其在结点3的p-use并未完全覆盖
- 但在后续执行中，会执行到$y\geq 0$那条边，因此此用例能够覆盖y在结点3的$p-use$
- 请分析：此用例的p-use覆盖率是多少?

```
1   begin
2     float x, y, z=0.0;
3     int count;
4     input (x, y, count);
5     do {
6       if (x≤0) {
7         if (y≥0) {
8           z=y*z+1;
9         }
10      }
11      else{
12        z=1/x;
13      }
14      y=x*y+z
15      count=count-1
16    while (count>0)
17    output (z);
18  end
```

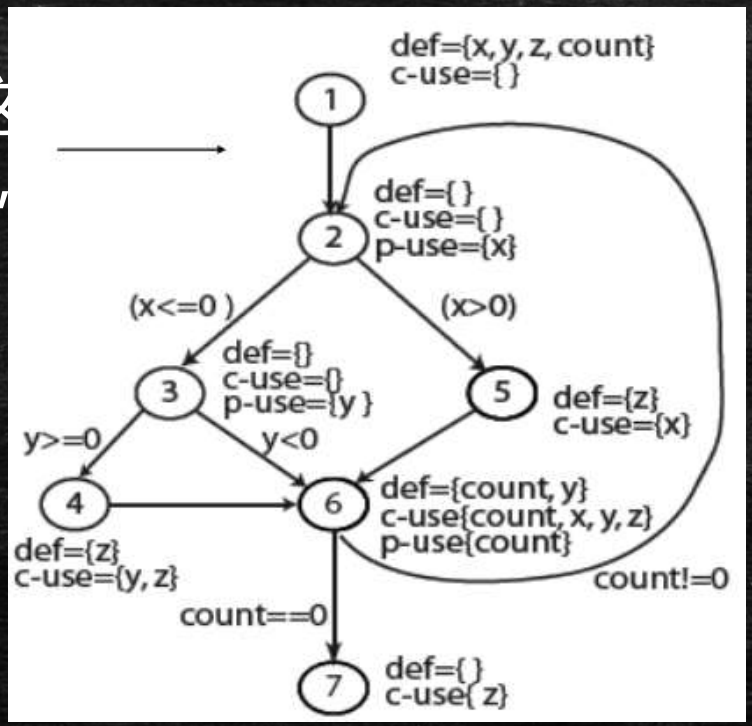


all-use覆盖准则

- 当所有的c-use和p-use都被覆盖时，称为all-use。其定义如下
- 测试集T针对(P,R)的$\mathrm{p}-use$覆盖率为：$\frac{CU_c+PU_c}{(CU-CU_i)+(PU-PU_i)}$，其中$CU,PU$为程序$P$中所有变量的$c-use,p-use$总数，$CU_c,PU_c$为覆盖的$c-use,p-use$数，$CU_i,PU_i$是无效的$c-use,p-use$数。如果$T$对$(P,R)$的$all-use$覆盖率为1，则称$T$相对于$all-use$覆盖准则是充分的。
- $T=\{t_c,t_p\}$

无效的c-use和p-use

- 为了覆盖一个c-use或p-use，要求遍历程序中的某条路径。如果此路径是无效的，则对应的c-use或p-use也可能是无效的。如果没有除非依然测试工具，很难发现无效的c-use和p-use
- 考虑变量在结点4的c-use，它是在结点5被定义的。为遍历这个c-use，必须要先达到结点5，这要求x>0，同时，要使控制流从结点5转到结点4，必须遍历6,2,3，这是不可能的，因为只有在x<=0时才能访问结点2,3



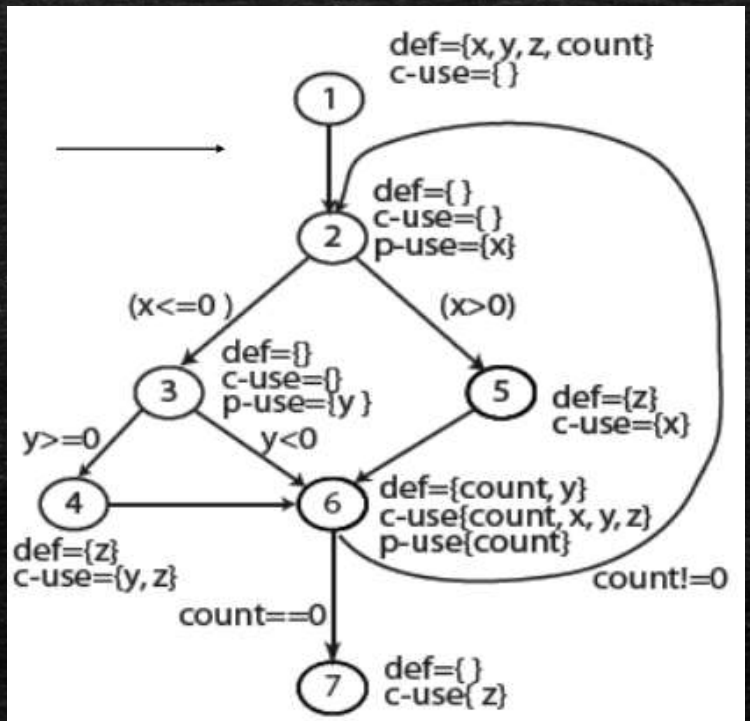

```
1    begin
2      float x, y, z=0.0;
3      int count;
4      input (x, y, count);
5      do {
6        if (x≤0) {
7          if (y≥0) {
8            z=y*z+1;
9          }
10       }
11       else{
12         z=1/x;
13       }
14       y=x*y+z
15       count=count-1
16     while (count>0)
17     output (z);
18   end
```

控制流和数据流

- 基于控制流的测试充分性准则旨在测试程序中大量的甚至是无穷多路径中很少的一部分路径。例如，如果测试时遍历的路径涉及了被测程序中所有的基本块，则认为满足了基本块覆盖准则。
- 基于数据流的测试充分性准则的目的也是从被测程序的众多路径中选择一些来执行。例如，如果测试时遍历的路径覆盖了所有的c-use，则认为满足了c-use覆盖准则。
- 有时，基于数据流的测试覆盖比基于控制流的测试覆盖具有更强的错误检测能力。

控制流与数据流

- $T = \{\langle x = -2, y = 2, count = 2\rangle, \langle x = 2, y = 2, count = 1\rangle\}$能够覆盖基本块、条件、判定，条件-判定等所有基于控制流的覆盖准则。但是其对于c-use和p-use的覆盖率只有58.3%和75%.




```
1    begin
2      float x, y, z=0.0;
3      int count;
4      input (x, y, count);
5      do {
6        if (x≤0) {
7          if (y≥0) {
8            z=y*z+1;
9          }
10       }
11       else{
12         z=1/x;
13       }
14       y=x*y+z
15       count=count-1
16     while (count>0)
17     output (z);
18   end
```