第二讲：白盒测试

LIAN YU
THE SCHOOL OF SOFTWARE AND MICROELECTRONICS
PEKING UNIVERSITY
NO.24 JINYUAN RD, BEIJING 102600

提　纲

白盒测试 (White-box Testing)，有时称为玻璃盒测试(Glass-box Testing)，是一种基于源程序或代码的测试方法，分为静态和动态两种类型。

白盒测试?

- 静态方法是指按一定步骤直接检查源代码来发现错误，而不用生成测试用例并驱动被测程序运行来发现错误，也称为代码检查法；
- 动态方法是指按一定步骤生成测试用例并驱动被测程序运行来发现错误。

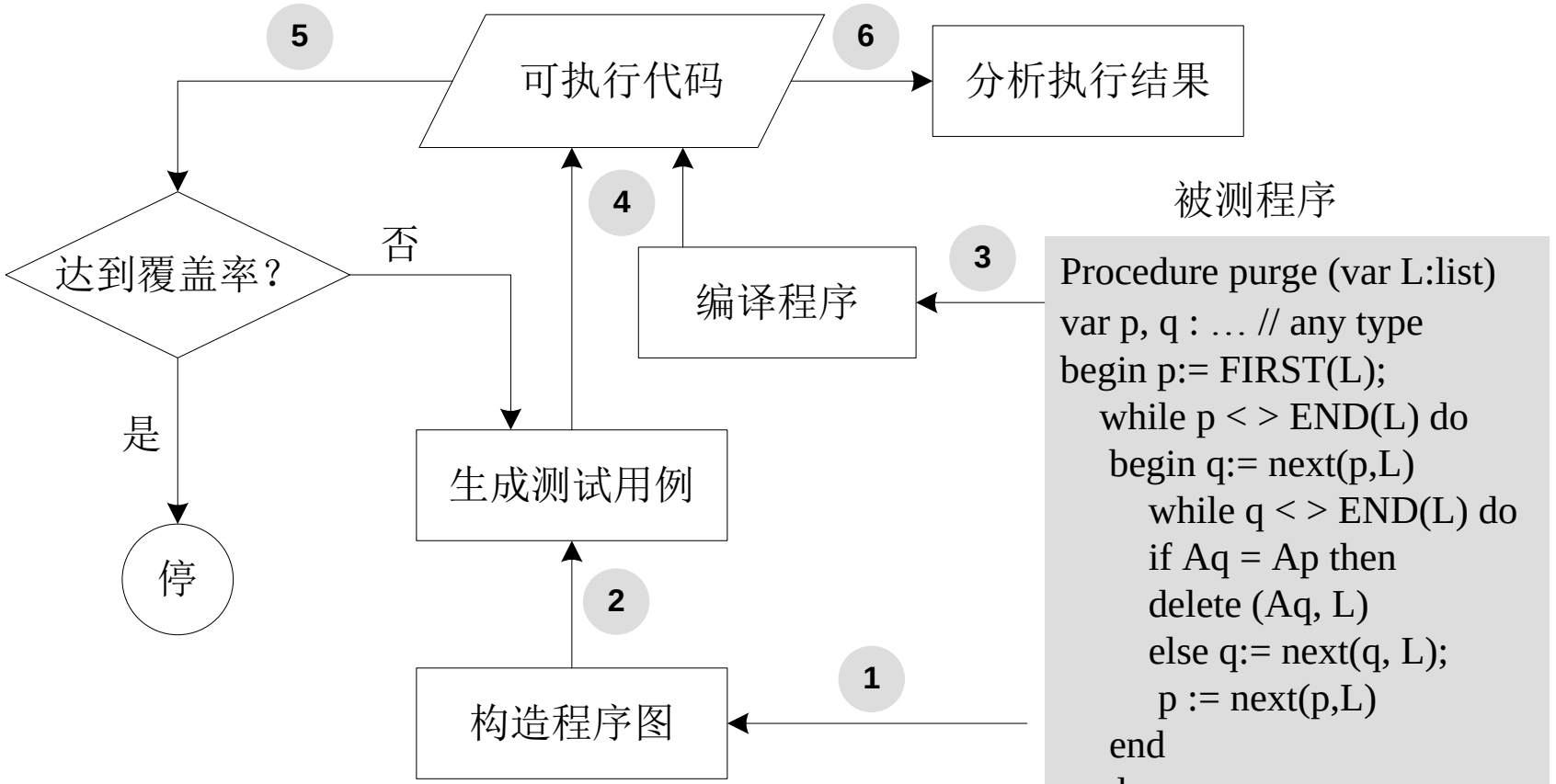
白盒测试（续）

静态方法有:	动态方法有:
• 桌面检查,代码审查,走查 • 静态分析	• 基本路径测试 • 条件测试 • 数据流测试 • 循环测试

在白盒测试中，程序结构通常用程序图来表达，并从程序图来产生测试用例。

白盒测试努力达到一个既定的覆盖标准（比如说，基本路径覆盖，语句覆盖，分支覆盖，等等）。

基于覆盖标准的白盒测试流程



基本路径测试

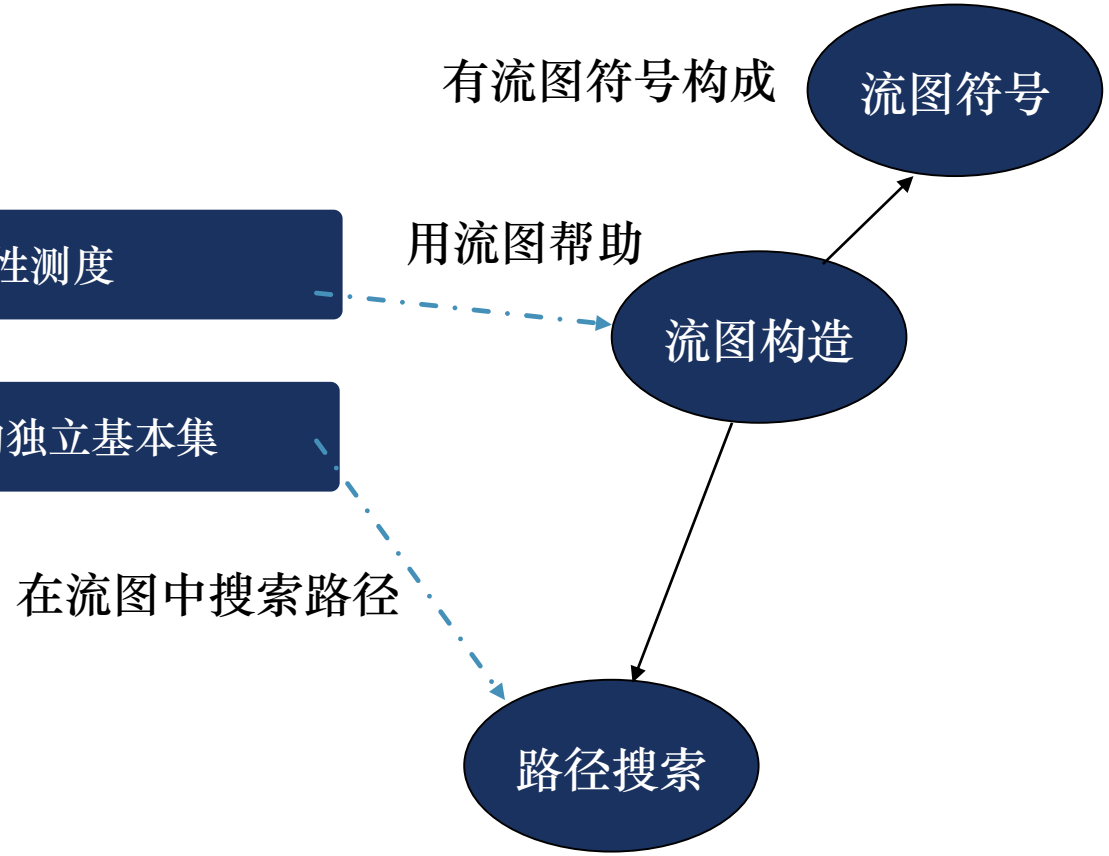
- 基本路径测试是Tom McCabe于1976首先提出的一种白盒测试技术，基本路径测试给测试用例设计者提供方法来
  - 求出程序或过程设计的逻辑复杂性测度，并
  - 使用该测度作为指南来定义执行路径的独立基本集。

- 路径的独立基本集的作用?
  - 从该基本集导出的测试用例保证对程序中的每一条语句至少执行一次。

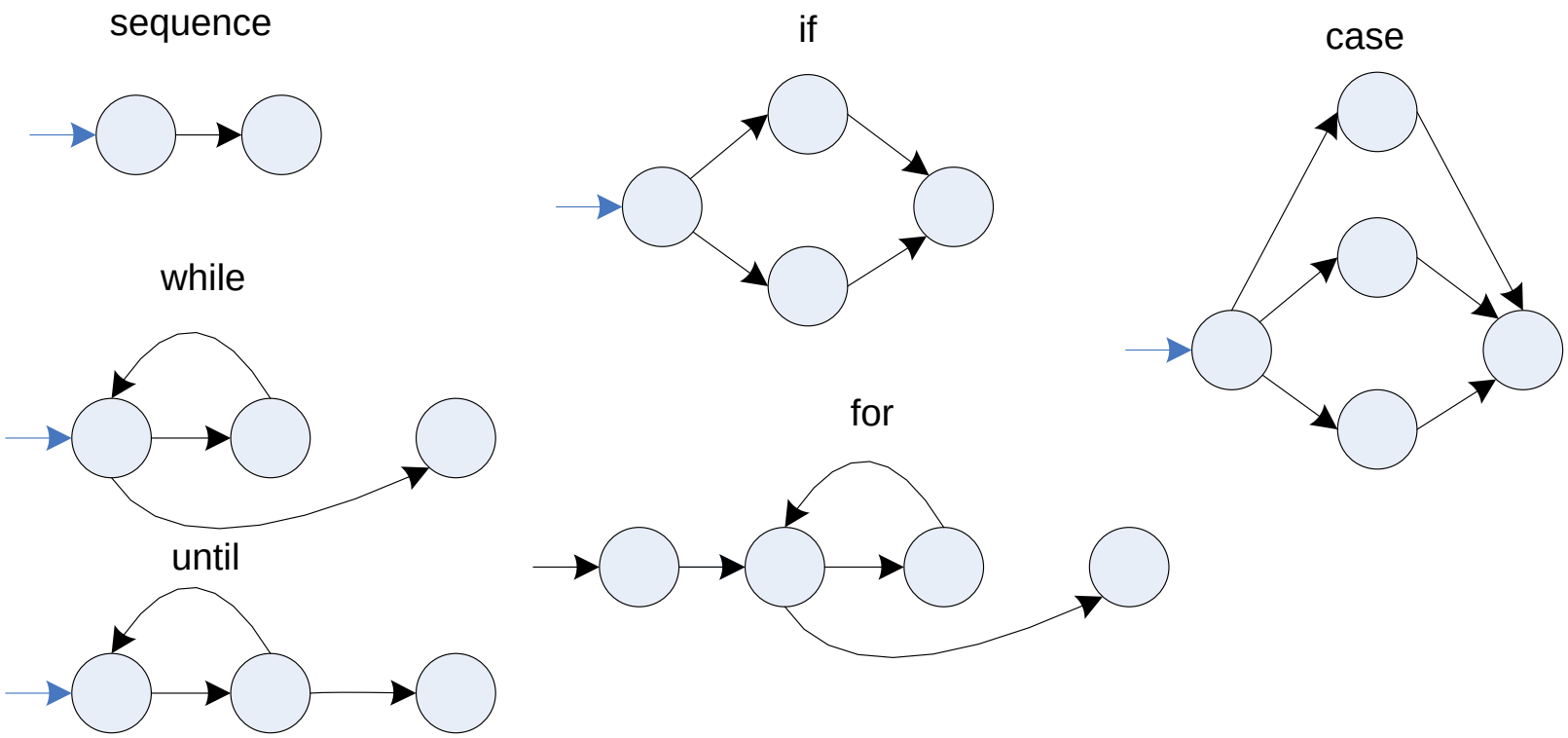
学习思路

求出程序或过程设计的逻辑复杂性测度

使用该测度作为指南来定义执行路径的独立基本集

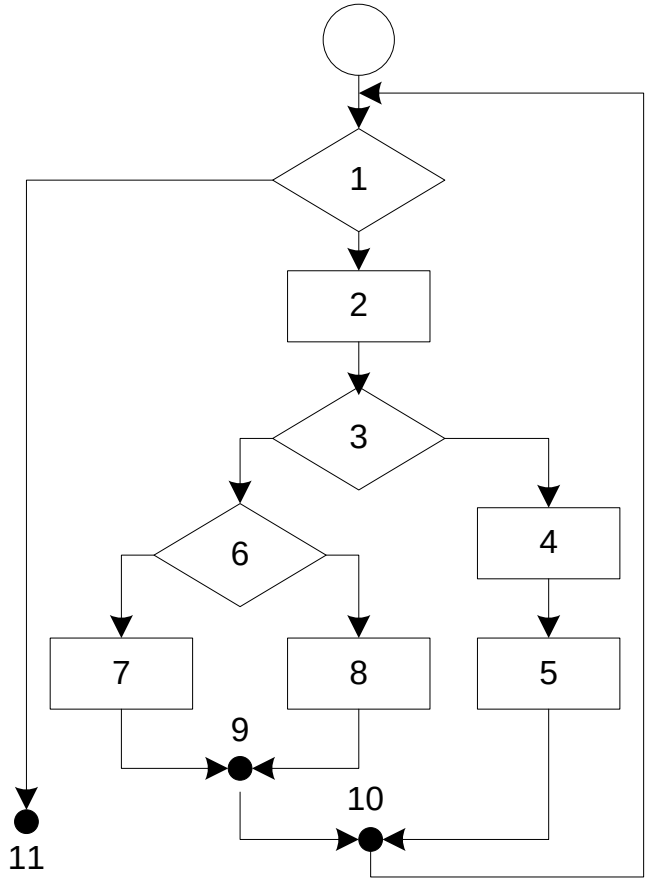


流图（FLOW GRAPH）符号

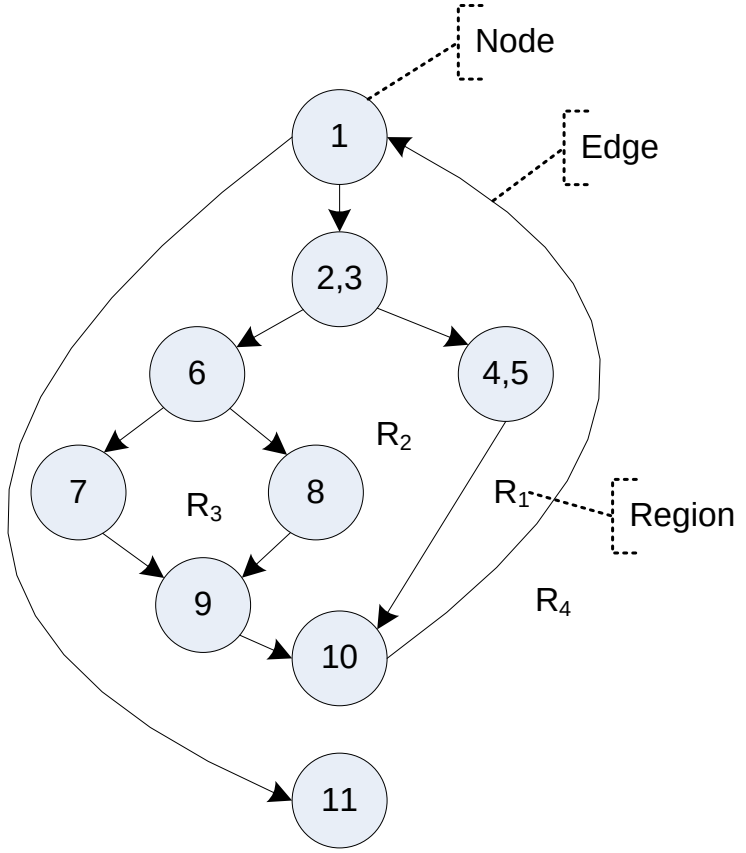


- 每个圆表示一个或多个非分支PDL（Program Design Language）或源代码语句

流程图与流图



(a)

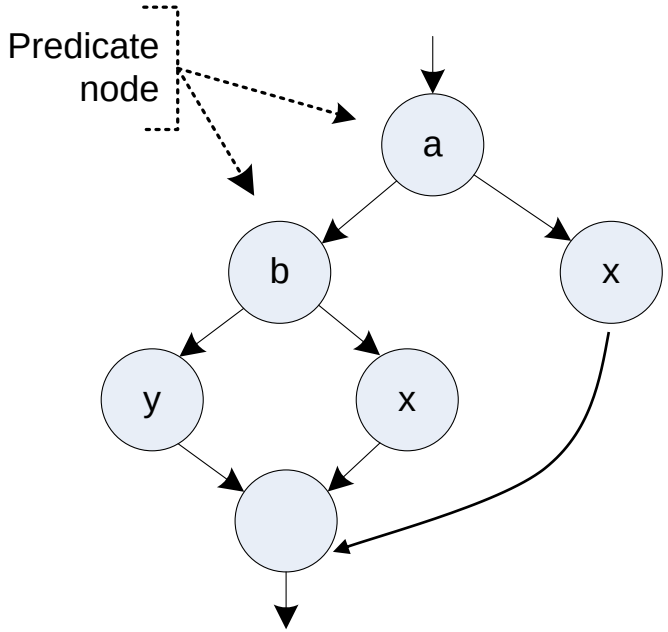


(b)

流图（续）

- 程序设计中遇到复合条件时，生成的流图变得稍微为复杂。

```
//PDL procedure
.
.
.
IF a OR b
  then procedure x
else procedure y
ENDIF
.
.
.
```



流图定义

- G=（V，E），其中V是顶点的集合，E是有向边的集合。
  - V = {vb，ve} U D U S，
    - 初始节点vb的入度函数indegree(vb)的值为零，indegree（vb）= 0，结束节点ve的出度函数outdegree（ve）的值为零, outdegree(ve) = 0，
    - D是**原子二元**判定条件的节点集，
    - S是顺序的节点集，每一个节点表示一块连续的语句群；
  - E = D×D U D×S U S×D是有向边的集合。

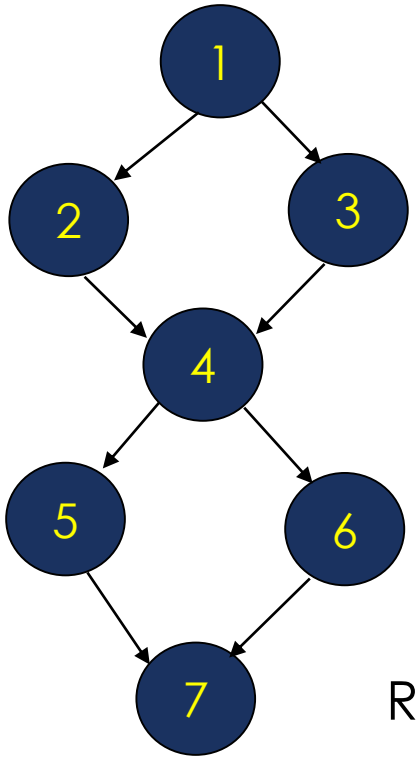
基本路径

- 路径p可以表示为一系列的顶点：
  - p= vb，v1，v2，…，vn，ve，
  - 其中{vb，vi (i=1，…， n)，ve} ⊆ V, {<vb,v1>,<v1,v2>,…,<vn,ve >} ⊆ E。
- 路径p= vb，v1，v2，…，vn，ve是一条基本路径，
  - 如果p不包含两个子序列的s1,s2使得s1 = s2并且length(s1) ﹥1和length(s2) ﹥1。其中：路径p的子序列s1，subseq(s1,p)当且仅当(∃s0)(∃s2)(s0,s1,s2 = p)。
  - 即：¬(∃s1)(∃s2)(subseq(s1,p) ∧ subseq(s2,p) ∧ s1 = s2 ∧ |s1| ﹥1 ∧ |s2| ﹥1)

独立路径集

- An independent basis set is a set of linearly independent test paths.

EXAMPLE



P1: 1-2-4-5-7
P2: 1-3-4-5-7
P3: 1-2-4-6-7

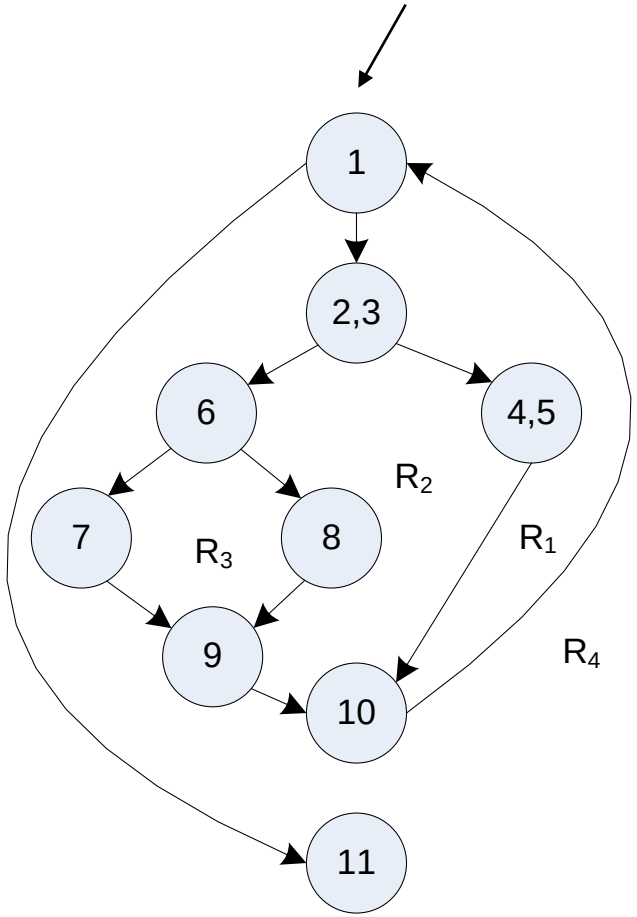
How about P4: 1-3-4-6-7????

X1*P1+X2*P2+X3*P3 =P4

Resolve the equation, and get X1=-1, X2=X3=1

例子:独立路径集

路径1：1-11
路径2：1-2-3-4-5-10-1-11
路径3：1-2-3-6-8-9-10-1-11
路径4：1-2-3-6-7-9-10-1-11



计算复杂性

如何才能知道需要寻找多少条路径呢?

- 对环形复杂性的计算结果为这个问题提供了答案。
- 环形复杂性以图论为基础，为我们提供了非常有用的软件度量。

环形复杂性 (CYCLOMATIC COMPLEXITY)

- 环形复杂性是一种为程序逻辑复杂性提供定量测度的软件度量。
- 当该度量用于基本路径测试方法，计算所得的值给出了程序基本集的独立路径数量，这是为确保所有语句至少执行一次而必须进行测试数量的上界。
- 可用如下三种方法之一来计算复杂性。

三种方法计算复杂性CC(G)

- Count：CC (G)=流图中区域的数量。

区域的数量

- 计算：CC (G)=E-N+2
  - E是流图中边的数量，N是流图节点数量。

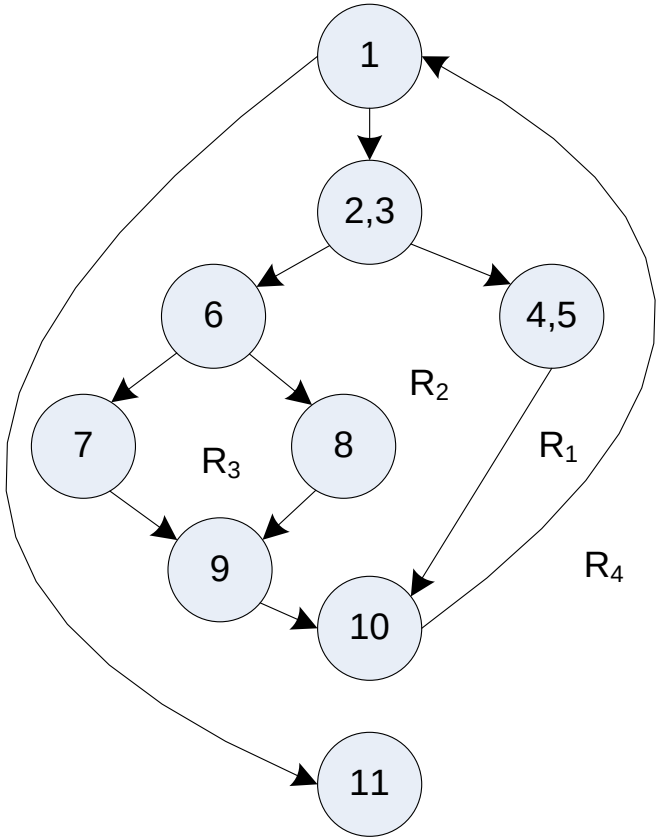
依据边和点

- 计算：CC (G)=P+1
  - P是流图G中判定节点的数量。

判定节点的数量

例 子

1. CC (G) =流图有4个区域
2. CC (G) = 11条边 - 9个节点 + 2 = 4
3. CC (G)= 3个判定节点 + 1 = 4



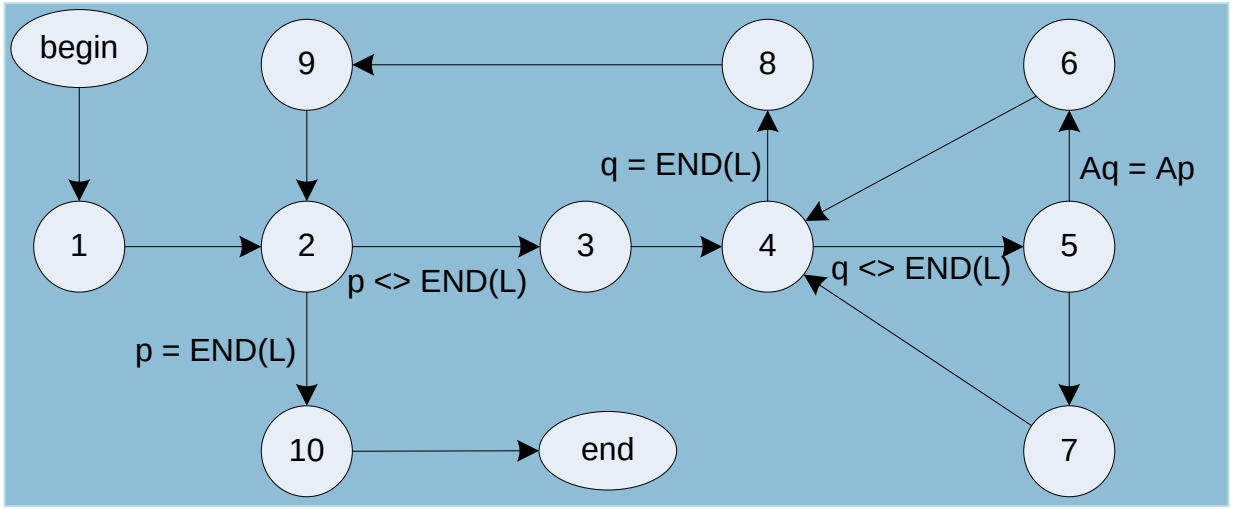
导出测试用例

利用基本路径测试产生测试用例的一系列步骤

- 以设计或代码为基础，画出相应的流图
- 确定结果流图的环形复杂性
- 确定线性独立的路径的一个基本集
- 准备测试用例，强制执行基本集中每条路径

生成测试用例 示例

- 1.以设计或代码为基础，画出相应的流图。
  - 使用符号和构造规则创建一个流图。
  - 创建流图时，要对将被映射为流图节点的语句进行标号（1）－（10）




```
Procedure purge (var L:list)
      var p, q: …//define p,q
      begin
(1)   p:= FIRST(L);
(2)       begin while p < > END(L) do
(3)           q:= next(p,L);
(4)           begin while q < > END(L) do
(5)               if Aq = Ap then
(6)                   delete (Aq, L)
(7)               else q:= next(q,L);
(8)           end
(9)           p := next(p,L)
(10)      end;
      end;
```

生成测试用例 示例

CC(G) = 4个区域
CC(G) = 14条边 - 12个节点 + 2 = 4
CC(G) = 3个判定节点 + 1 = 4

- 2.确定结果流图的环形复杂性。
  - 可采用上一节中的任意一种算法来计算环形复杂性——CC (G)。

- 补充：
  - 计算CC (G)并不一定要画出流图，计算PDL中的所有原子条件语句数量(while及if条件语句)，然后加1即可得到环形复杂性。
  - 应该注意到，如果是复合条件语句，需要求出原子条件语句数量。

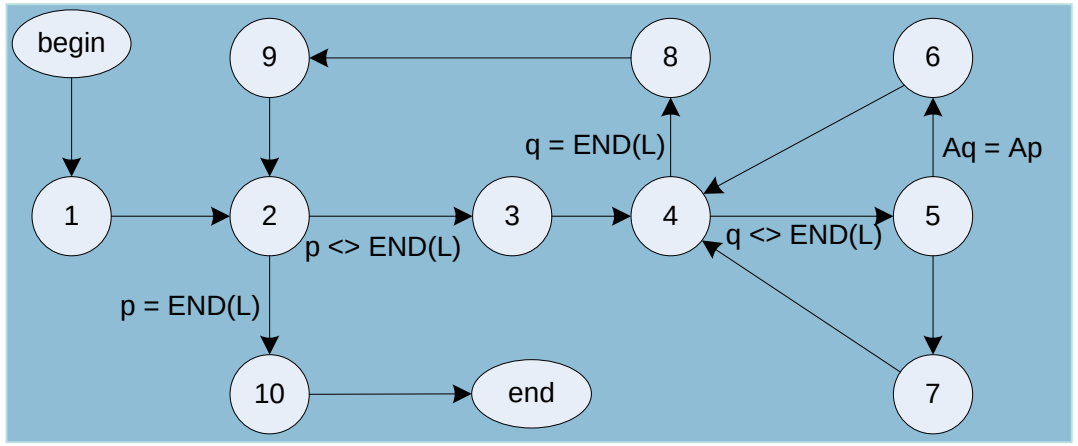
生成测试用例 示例

- 3. 确定线性独立的路径的一个基本集。
  - CC (G)的值提供了程序控制结构中线性独立的路径的数量，通常在导出测试用例时，识别判定节点是很有必要的。本例中，节点2、4和5是判定节点。
  - **启发式算法**：“每一条基本路径覆盖至少一条未被其他路径覆盖到的边。”

路径1：1-2-10
路径2：1-2-3-4-8-9-2-10
路径3：1-2-3-4-5-6-4-8-9-2-10
路径4：1-2-3-4-5-7-4-8-9-2-10

生成测试用例 示例

- 4. 准备测试用例，强制执行基本集中每条路径。
  - 测试人员可选择数据以便在测试每条路径时适当设置判定节点的条件。



路径1测试用例：

输入条件：L=( ),即列表为空

期望结果：L=( )，无列表清理处理

路径2测试用例：

输入条件：L=(Ap)

期望结果：L=(Ap)

路径3测试用例：

输入条件：L=(Ap Aq)且 Ap = Aq

期望结果：L=(Ap)，从列表中清理Aq

路径4测试用例：

输入条件：L=(Ap Aq)且 Ap <> Aq

期望结果：L=(Ap Aq)

```java
public static Object[] typeCompare(Object[] arr){
    int t = 0;
    Object[] tempArr = new Object[arr.length];
    for(int i = 0; i < arr.length; i++){
        boolean isRepeat = false;
        for(int j=i+1;j<arr.length;j++){
            if(arr[i]==arr[j]){
                isRepeat = true;
                break;
            }
        }
        if(!isRepeat){
            tempArr[t] = arr[i];
            t++;
        }
    }
    Object[]  newArr = new Object[t];
    System.arraycopy(tempArr,0,newArr,0,t);
    return newArr;
}
```

此处采用==进行判断，只能判断基础数据类型。比如（123==123将会返回true）

若是入参为引用类型则不能进行正确判断(此处不考虑java中字符串的存储策略，只是通过引用类型来说明问题)。比如（"123"=="123"将会返回false，需要采用equals进行比较）

```
----------------------不同数据类型测试开始----------------------
字符串类型的输入参数为：["123","123"]
字符串类型的期望结果为：["123"]
字符串类型的输出结果为：["123","123"]
与期望结果不一致，字符串类型未能去重，去重失败。应采用equals方法进行比较
----------------------
整数类型的输入参数为：[123, 123]
整数类型的期望结果为：[123]
整数类型的输出结果为：[123]
与期望结果一致，整数类型去重成功
----------------------不同数据类型测试结束----------------------
```

与期望结果不一致

与期望结果一致

```java
public static Object[] logicError(Object[] arr){
    int t = 0;
    Object[] tempArr = new Object[arr.length];
    for(int i = 0; i < arr.length; i++){
        boolean isRepeat = false;
        for(int j=i+1;j<arr.length;j++){
            if(arr[i]!=arr[j]){
                isRepeat = true;
                break;
            }
        }
        if(!isRepeat){
            tempArr[t] = arr[i];
            t++;
        }
    }
    Object[]  newArr = new Object[t];
    System.arraycopy(tempArr,0,newArr,0,t);
    return newArr;
}
```

此处判断逻辑发生错误，导致结果不正确。将arr[i]==arr[j];条件写成了不等于（!=）

```
----------------------逻辑错误测试开始----------------------
输入参数为：[123, 123, 456, 456]
期望结果为：[123,456]
输出结果为：[456]
去重失败
----------------------逻辑错误测试结束----------------------
```

与期望结果不一致

```java
public static Object[] missBreak(Object[] arr){
    int t = 0;
    Object[] tempArr = new Object[arr.length];
    for(int i = 0; i < arr.length; i++){
        boolean isRepeat = false;
        for(int j=i+1;j<arr.length;j++){
            if(arr[i]==arr[j]){
                isRepeat = true;

            }
        }
        if(!isRepeat){
            tempArr[t] = arr[i];
            t++;
        }
    }
    Object[]  newArr = new Object[t];
    System.arraycopy(tempArr,0,newArr,0,t);
    return newArr;
}
```

此处缺少退出语句;应添加该行代码：break; 否则导致程序不能输出正确结果。

```
----------------------缺失退出语句测试开始----------------------
输入参数为：[123, 123, 456, 456]
期望结果为：[123]
输出结果为：[123, 456, 456]
去重失败
----------------------缺失退出语句测试结束----------------------
```

与期望结果不一致

```java
public static Object[] indexOutOf(Object[] arr){
    int t = 0;
    Object[] tempArr = new Object[arr.length];
        for(int i = 0; i <= arr.length; i++){
            boolean isRepeat = false;
            for(int j=i+1;j<arr.length;j++){
                if(arr[i]==arr[j]){
                    isRepeat = true;
                    break;
                }
            }
            if(!isRepeat){
                tempArr[t] = arr[i];
                t++;
            }
        }
    Object[]  newArr = new Object[t];
    System.arraycopy(tempArr,0,newArr,0,t);
    return newArr;
}
```

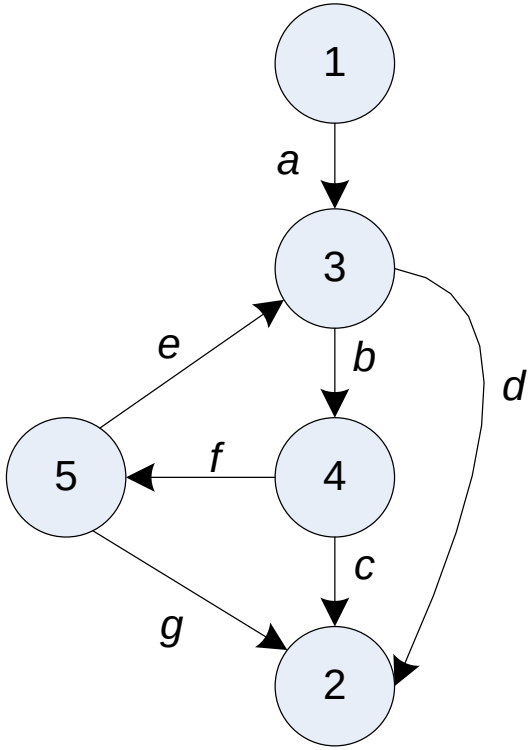
此处数组长度的取值会导致数组越界异常。应修改为i<arr.length.
错误原因：数组是从索引0开始取值的，最大索引为arr.length -1,不是arr.length

```
----------------------数组越界测试开始----------------------
输入参数为：[123, 123]
整数类型的期望结果为：[123]
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 2
        at cn.edu.pku.demo.BasicPathFailTest.indexOutOf(BasicPathFailTest.java:139)
        at cn.edu.pku.demo.BasicPathFailTest.testIndexOutOf(BasicPathFailTest.java:66)
        at cn.edu.pku.demo.BasicPathFailTest.main(BasicPathFailTest.java:20)
```

抛出数组越界异常，与期望结果不一致

图矩阵法

- 导出流图和决定基本测试路径的过程都需要机械化或自动化手段来支持。
- 为了开发辅助基本路径测试的软件工具，一种称为图矩阵(graph matrix)的数据结构很有用。
- 图矩阵是一个正方形矩阵，其大小(即列数和行数)等于流图的节点数。
- 每列和每行都对应于标识的节点，矩阵项对应于节点间的连接(边)

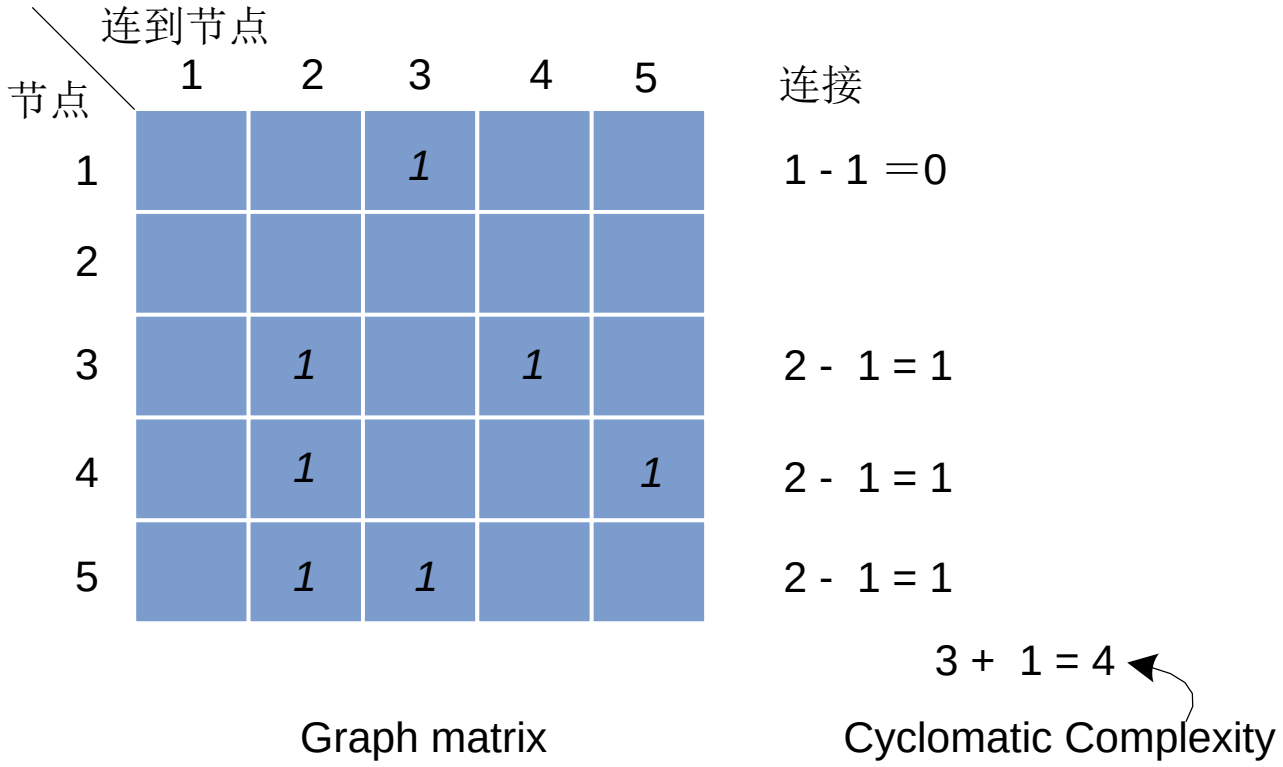


Flow graph

节点 \ 连到节点	1	2	3	4	5
1			*a*		
2					
3		*d*		*b*	
4		*c*			*f*
5		*g*	*e*		

Graph matrix

连接矩阵(CONNECTION MATRIX)



条件测试

- 条件测试是检查程序模块中所包含逻辑条件的测试用例设计方法。
- 一个简单条件是指一个布尔变量或一个关系表达式,两者前面可能带有NOT(“¬”)操作符。
- 关系表达式的形式如下:

  E1 ＜关系操作符＞ *E2*

  - 其中E1和E2是算术表达式，而＜关系操作符＞是下列之一：“＜”，“≤”，“=”，“≠”(“¬=”)，“＞”，或“≥”。

条件测试（续）

- 复杂条件由两个或多个简单条件、布尔操作符和括弧组成。
  - 可用于复杂条件的布尔操作符包括OR “|”，AND “&”和NOT “¬”。
- 不含关系表达式的条件称为布尔表达式。

条件测试（续）

- 布尔操作符错误
  - 遗漏布尔操作符，布尔操作符多余或布尔操作符不正确
- 布尔变量错误
- 布尔括弧错误
- 关系操作符错误
- 算术表达式错误

条件测试（续）

- 条件测试方法注重于测试程序中的条件。
- 条件测试策略主要有两个优点：
  - 首先，一个条件测试的覆盖率度量是简单的，
  - 其次，程序的各个条件测试覆盖率为产生另外的程序测试提供了指导。

条件测试（续）

- 分支测试可能是最简单的条件测试策略：
  - 对于复合条件C，C的真分支和假分支以及C中的每个简单条件都需要至少执行一次［MYE79］。

A	B	A && B
T	T	T
F	F	F

条件测试（续）

- 域测试(Domain testing)［WHI80］要求从关系表达式中导出三个或四个测试用例，关系表达式的形式如：

  E1 ＜关系操作符＞ E2

  - 需要三个测试用例分别用于计算E1的值是大于（＞）、等于（＝）或小于（＜）E2的值［HOW82］。
  - 如果＜关系操作符＞错误，而E1和E2正确，则这三个测试用例能够发现关系操作符的错误。
  - 为了发现E1和E2的错误，计算E1小于或大于E2的测试用例应使两个值间的差别尽可能小。

BRO策略BRANCH AND RELATIONAL OPERATOR

- 假设在一个条件中所有的布尔变量和关系表达式只出现一次而且没有公共变量，BRO保证能发现该条件中的分支（布尔）操作符和关系操作符错误。

```
(x>3) && (y<=5) || (a && (z>6))
(x>3) && (y<=5) || (a && (x>6)) || a
(x>3) && ( x+y<=5) || (a && (z>6))
```

BRO策略（续）

- BRO策略利用某条件C的条件约束。
  - 有n个简单条件的条件C的条件约束定义为(D1，D2，…，Dn)，其中Di(0<i≤n)表示条件C中第i个简单条件的结果（outcome）约束。
  - 如果执行条件C过程中，C的每个简单条件的结果都满足D中对应的约束，则称条件C的条件约束D由C的执行所覆盖。

BRO策略（续）

- 对于布尔变量B，B输出的约束指定B必须是真(t)或假(f)。
- 类似地，对于关系表达式符号<、=、>用于指定表达式输出的约束。

X	Y	implementation	specification
		X >= Y	X < Y
3	2	T	F
4	4	T	F
5	6	F	T

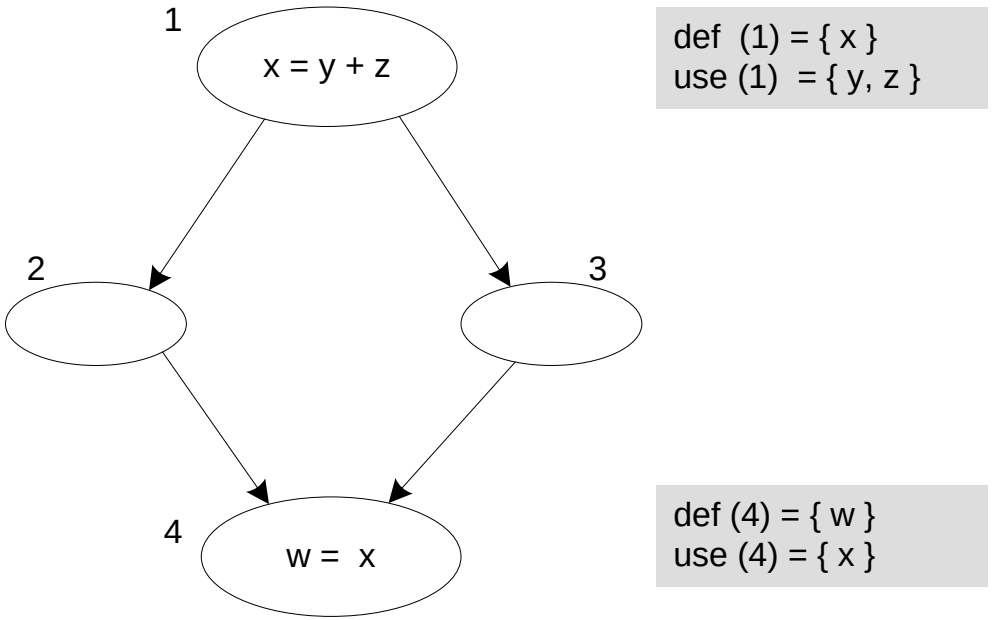
BRO策略（续）

- C1: B1 & B2
  - BRO测试策略要求约束集｛(t，t)，(f，t)，(t，f)｝由C1的执行所覆盖，如果C1由于布尔操作符的错误而不正确，至少有一个约束强制C1失败。
- C2: B1 & (E3 = E4)
  - C2的约束集｛(t，=)，(f，=)，(t，<)，(t，>)｝
- C3: (E1＞E2) & (E3=E4)
  - C3的约束集：｛(＞，=)，(=，=)，(＜，=)，(＞，＜)，(＞，＞)｝

数据流测试

- 数据流测试方法按照程序中的变量定义和使用的位置来选择程序的测试路径。
- 已经有不少关于数据流测试策略的研究
  - 如参考文献［FRA88］、[NTA88]和[FRA93]
- 为了说明数据流测试方法，假设程序的每条语句都赋予了唯一的语句号，而且每个函数都不改变其参数和全局变量。

- def(S)= {x | 语句S包含X的定义}
- use(S)= {x | 语句S包含X的使用}



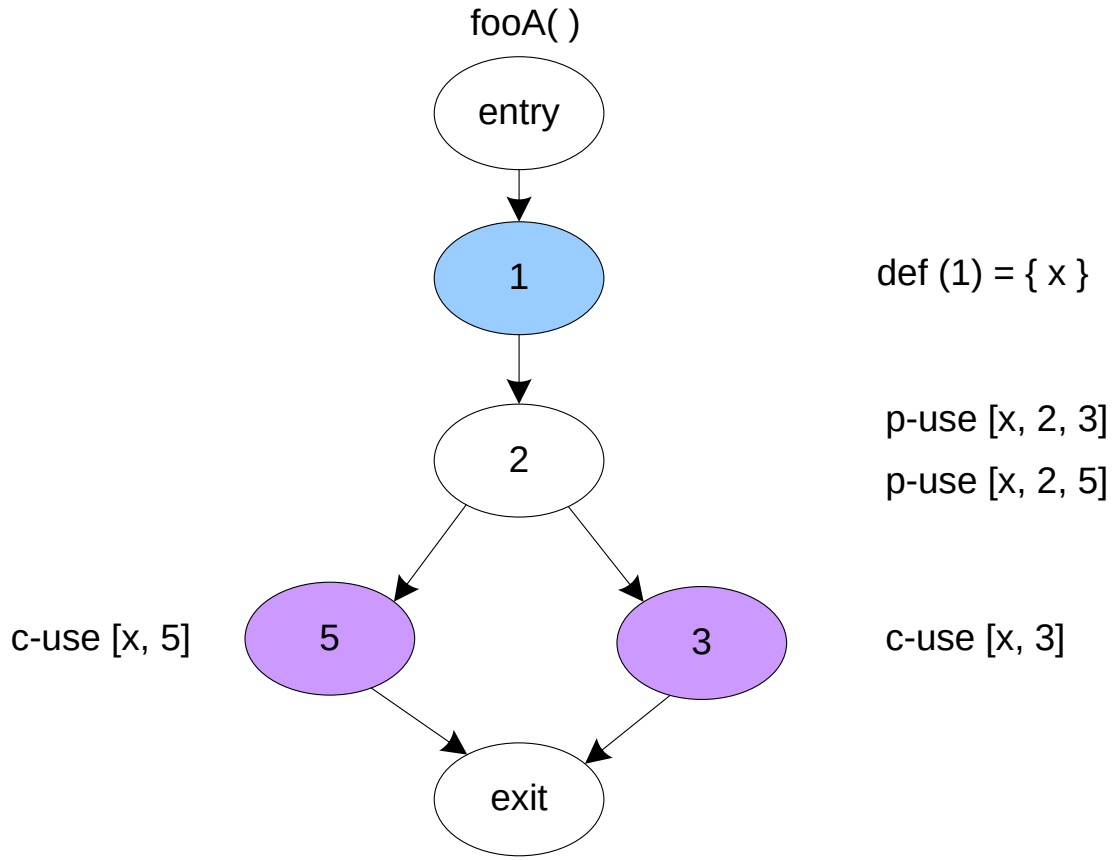
数据流测试

- 如果存在从S到S'的路径，并且该路径不含X的其他定义，则称变量X在语句S处的定义在语句S'仍有效或称为定义清纯（def-clear）。
- 变量x的“定义使用关联”(du-association)形式如［x，S，S'］，其中S和S'是语句号，x在def(S)和use(S')中，而且语句S定义的X在语句S'有效。
- 图中变量x的du-关联为［x，1，4］。

数据流测试

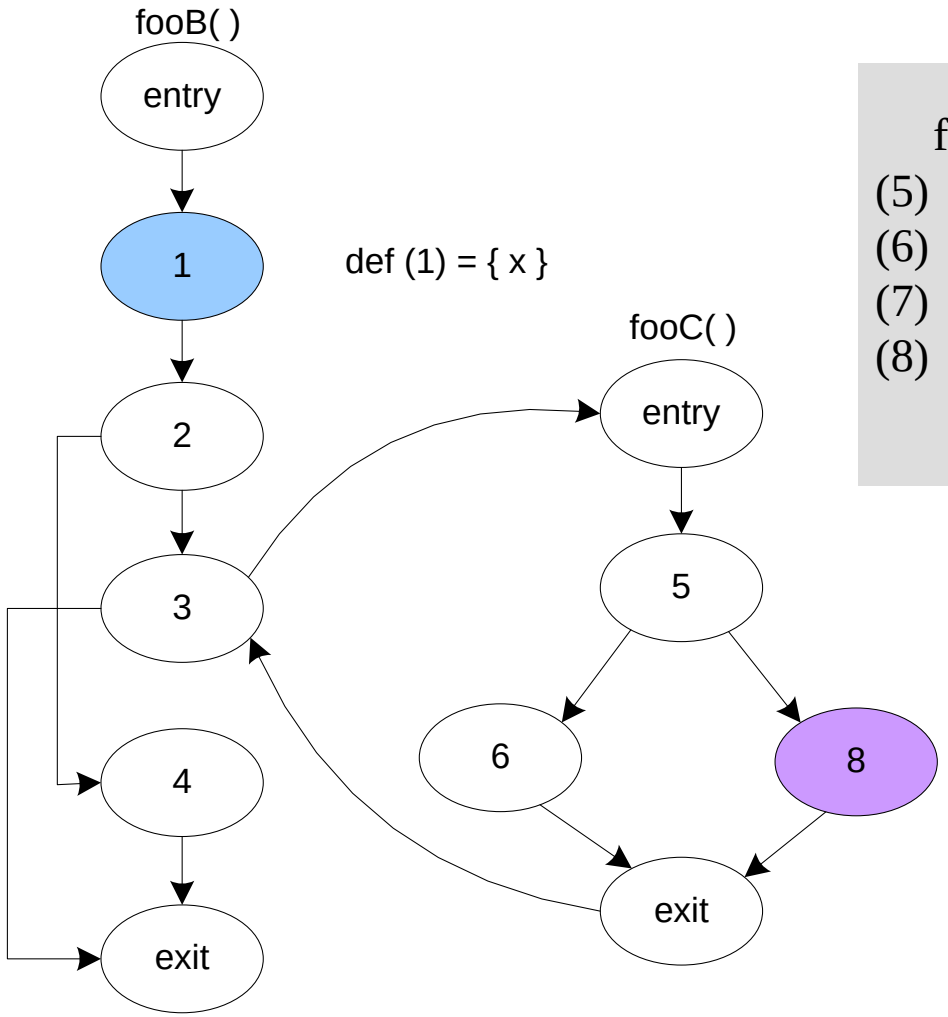
- 一个变量的使用可以是计算使用 (c-use)或断言使用 (p-use)。
- 一个变量的du-链是一条路径，这条路径是从变量的定义到变量的使用之间无任何重定义（也就是说，definition-clear）。
  - 对于c-use来说，du-链是从含有定义语句到含有计算使用语句之间的路径。
  - 对于p-use来说,du-链是从含有定义语句到包含两个执行分支断言使用语句之间的路径。

```
    fooA( ){
(1)     x = read( );
(2)   if (x > 0)
(3)       print x-1;
(4)   else
(5)       print x;
    }
```



过程内部du－链

```
    fooB( ){
(1)      x = read( );
(2)    if (x > 0)
(3)         x = fooC(x);
       else
(4)         print x;
    }
```




```
    fooC( ){
(5)     if (x > 0)
(6)         return 5;
(7)    else
(8)         return x;
    }
```

p-use [x, 5, 6]

p-use [x, 5, 8]

c-use [x, 8]

过程间du－链

生成测试用例的步骤

- 数据流测试方法生成测试用例的步骤：
  - 为每个变量确定“定义－使用链”(du-链)
  - 确定测试标准，比如说，取全部定义，取全部使用或取全部路径等
  - 设计测试用例来符合测试的标准

- 一种简单的数据流测试策略是要求覆盖每个DU链至少一次。我们将这种策略称为DU测试策略。

```
proc x
(1)         B1；//define X at end of B1
(2)         do while C1
(3)               if C2
(4)                     then
(5)                           if C4
(6)                                 then B4；//use X at the beginning of B4, define X at end of B4
(7)                                 else B5；//use X at the beginning of B5, define X at end of B5
(8)                           endif；
(9)                     else
(10)                          if C3
(11)                                then B2；//use X at the beginning of B2, define X at end of B2
(12)                                else B3；//use X at the beginning of B3, define X at end of B3
(13)                          endif；
(14)              endif；
(15)        enddo；
(16)        B6；//use X at the beginning of B6
end proc；
```

DU测试策略

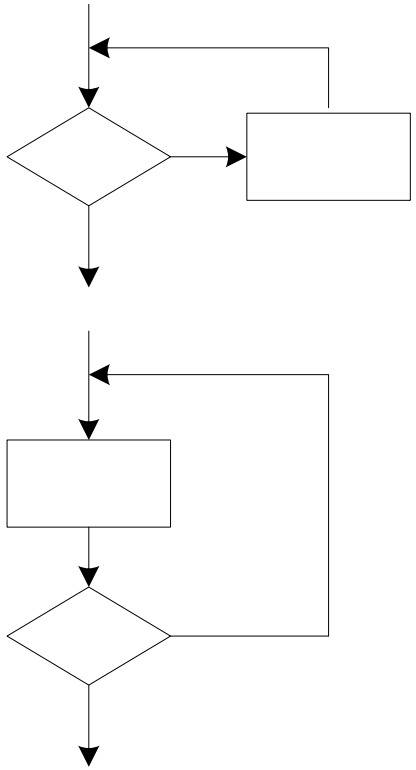
DU-路径1：(1)-(2)-(15)-(16)
DU-路径2：(1)-(2)-(3)-(4)-(5)-(6)-(8)-(14)-(2)-(15)-(16)
DU-路径3：(1)-(2)-(3)-(4)-(5)-(7)-(8)-(14)-(2)-(15)-(16)
DU-路径4：(1)-(2)-(3)-(9)-(10)-(11)-(13)-(14)-(2)-(15)-(16)
DU-路径5：(1)-(2)-(3)-(9)-(10)-(12)-(13)-(14)-(2)-(15)-(16)

数据流测试与条件测试

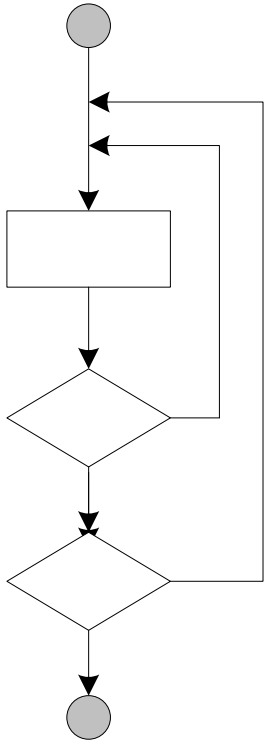
- 由于变量的定义和使用，程序中的语句都彼此相关，所以数据流测试方法能够有效地发现错误，
- 但是，数据流测试的覆盖率度量和路径选择比条件测试更为困难。

循环测试

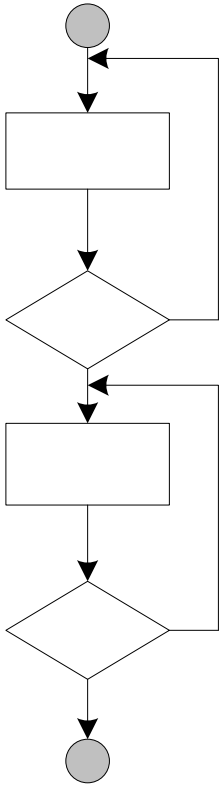
- 循环是大多数软件实现算法的重要部分，但是在软件测试时却往往较少注意它们。
- 循环测试是一种白盒测试技术，注重于循环构造的有效性。
- 有四种循环［BEIZER,1990］：简单循环，串接循环，嵌套循环和不规则循环



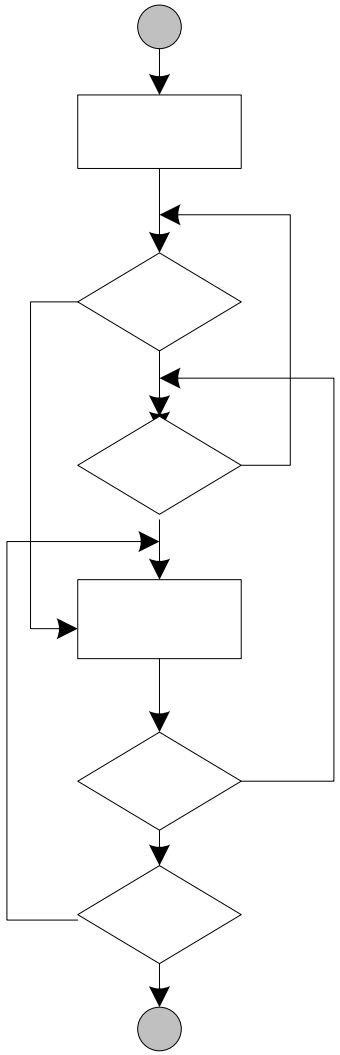
Simple loops



Nested loops



Concatenated loops



Unstructured loops

简单循环

- 下列测试集可以用于简单循环，其中n是允许通过循环的最大次数。
  - 跳过整个循环。
  - 只执行一次循环。
  - 执行两次循环。
  - 执行m次循环，其中m＜n。
  - 执行n-1，n，n+1次循环。

嵌套循环

- 如果要将简单循环的测试方法用于嵌套循环，可能的测试数就会随嵌套层数成几何级增加，这会导致不实际的测试数目，Beizer提出了一种减少测试数的方法：
  1. 从最内层循环开始，将其他循环设置为最小值。
  2. 对最内层循环使用简单循环测试，而使外层循环的迭代参数(即循环计数)最小，并增加其他的测试用例来测试范围外或排除的值进行。
  3. 由内向外构造下一个循环的测试，但其他的外层循环为最小值，并使其他的嵌套循环为“典型”值。
  4. 继续直到测试完所有的循环。

串接循环

- 如果串接循环的循环都彼此独立，可以使用简单循环测试策略来测试串接循环。
- 但是，如果两个循环串接起来，而第一个循环的循环计数是第二个循环的初始值，则这两个循环并不是独立的。
  - 这种情况下，即如果循环不独立，则推荐使用嵌套循环的方法进行测试。

不规则循环

- 在尽可能的情况下，要将这类循环重新设计为结构化的程序结构。

代码审查

- 在十九世纪七十年代中叶， Michael Fagan 在I B M制定出了审查的过程（Fagan 1976），其他人在此基础上又做了扩展和修改（Gilb and Graham 1993）。
- 该过程被认为是软件业最佳的实践（Brown 1996）。人们可以审查任何一种软件工作产品，包括需求和设计文档、源代码、测试文档及项目计划等等。
- 审查定义为多阶段过程，涉及到由受过培训的参与者组成的小组，他们把重点放在查找工作产品缺陷上。
- 审查提供了一个质量关卡，文档在最终确定以前，必须通过该关卡的检查。虽然，对于Fagan的方法是否最有效并且是不是最有效的审查的形式还存在争议(Glass, 1999)，但是审查是强有力的质量技术，这是毫无疑问的。

代码审查 — 角色

主持人

- 主持人负责保证审查以既定的速度进行，使其既能保证效率，又能发现尽可能多的错误。
- 主持人在技术上面必然能够胜任——虽然不一定是被检查的代码方面的专家，但必须能够理解有关的细节。
- 主持人还负责管理审查的其他方面，例如分派审查代码的任务，分发审查所需的核对表，预定会议室，报告审查结果，以及负责跟踪审查会议上指派的任务。

作者

- 直接参与代码设计和编写的人，该角色在审查中扮演相对次要的角色。
- 审查的目标之一就是让代码本身能够表达自己。如果它不够清晰，那么就需要向作者分配任务，使其更加清晰。
- 除此之外，作者的责任就是解释代码中不清晰的部分，偶尔还需要解释那些看起来好像有错的地方为什么实际是可以接受的。
- 如果参与评论的人对项目不熟悉，作者可能还需要陈述项目的概况，为审查会议做准备。

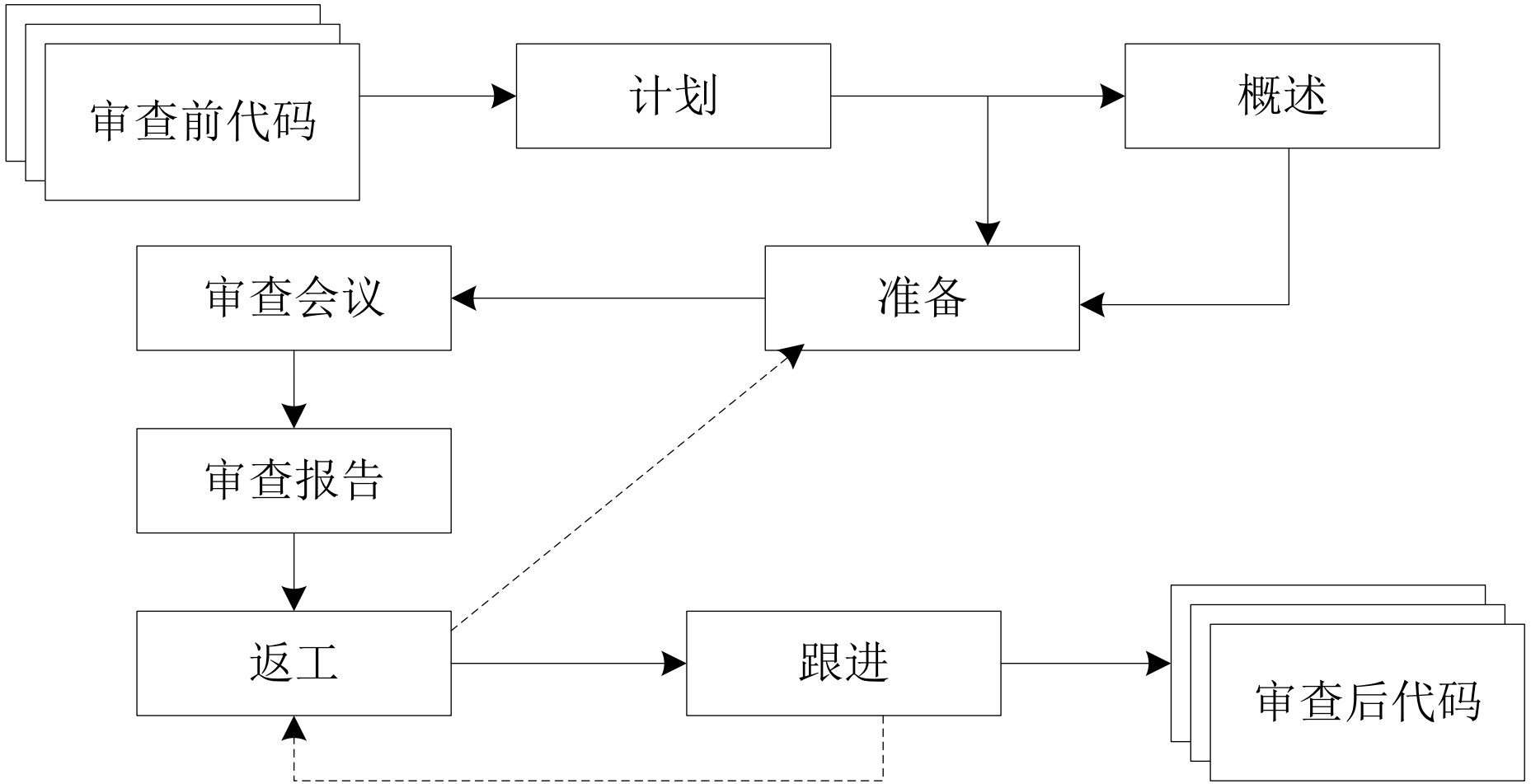
评论员

- 评论员是同代码有直接关系，但又不是作者的人。测试人员或者高层架构师也可以参与。评论员的责任是找出缺陷，他们通常在为审查会议做准备的阶段就已经找出了部分缺陷，然后随着审查会议中对代码的讨论，他们应该能够找出更多的缺陷。

记录员

- 记录员将审查会议期间发现的错误，以及指派的任务记录下来。作者和主持人都不应该担任记录员。

代码审查的一般步骤



桌面检查

- 桌面检查是一种人工检查程序的方法，通过对源程序代码进行分析、检验，来发现程序中的错误。
- 桌面检查关注的是变量的值和程序逻辑，所以执行桌面检查要严格按照程序中的逻辑顺序。
- 检查人员使用笔和纸记录下检查结果。
- 桌面检查可以由程序作者本人来执行，但这对于大多数程序员来说效率并不高。因为这违反了一条测试原则——人们在测试自己的程序时效率通常是很低的。所以，桌面检查最好由另一个人来执行而不是程序作者本人（比如可以两个程序员互相检查对方的程序）。
- 但这种方法不如审查或走查过程有效，因为审查或走查需要一个团队，团队会议营造一种健康的竞争环境，团队成员以找出错误来体现自己的价值。而桌面检查过程只有一个人在阅读代码，没有团队成员之间的协作效应。
- 总之，桌面检查比什么都不做要好，但不如审查或走查有效。

代码走查

- 代码走查和代码审查具有很多相同的步骤，但在查找错误的方法上有些小小的不同。
- 会议的进程与代码审查不同。不是简单地读程序和对照核对表进行检查，而是让与会者“充当”计算机。首先由测试者为所测程序准备一批有代表性的测试用例，提交给走查小组。
- 在会议上，与会者集体扮演计算机的角色，让测试用例沿程序的逻辑运行一遍，随时记录程序的踪迹和状态（也就是各变量的值），供分析和讨论用。
- 当然用例数量不能太多，太复杂，因为人们“执行”程序的速度要比计算机慢很多。用例本身并不起主要作用，它们只是作为媒介来向代码作者提出有关程序设计和逻辑方面的问题。在大多数走查过程中，更多的错误是在提问的过程中被发现，而不是直接运行测试用例的过程中。

总 结

- 白盒测试概念
- 基本路径测试的过程
- 数据流测试方法
- 代码审查的作用