白盒测试

白盒测试

- 白盒测试概念
- 测试覆盖标准
- 逻辑驱动测试
- 基本路径测试
- 控制结构测试的变种
- 面向对象的白盒测试

白盒测试

- 白盒测试概念

  白盒测试也称结构测试或逻辑驱动测试，是一种测试用例设计方法，它从程序内部逻辑结构及有关信息来设计和选择测试用例，对程序的逻辑路径进行测试。

  应用白盒法时，手头必须有程序的规格说明以及程序清单。

- 白盒测试的主要目的

  保证一个模块中的所有独立路径至少被执行一次；

  对所有的逻辑值均需要测试真、假两个分支；

  在上下边界及可操作范围内运行所有循环；

  检查内部数据结构以确保其有效性。

白盒测试

- 白盒测试的优缺点
  1. 优点

     迫使测试人员去仔细思考软件的实现

     可以检测代码中的每条分支和路径

     揭示隐藏在代码中的错误

     对代码的测试比较彻底
  2. 缺点

     昂贵

     无法检测代码中遗漏的路径和数据敏感性错误

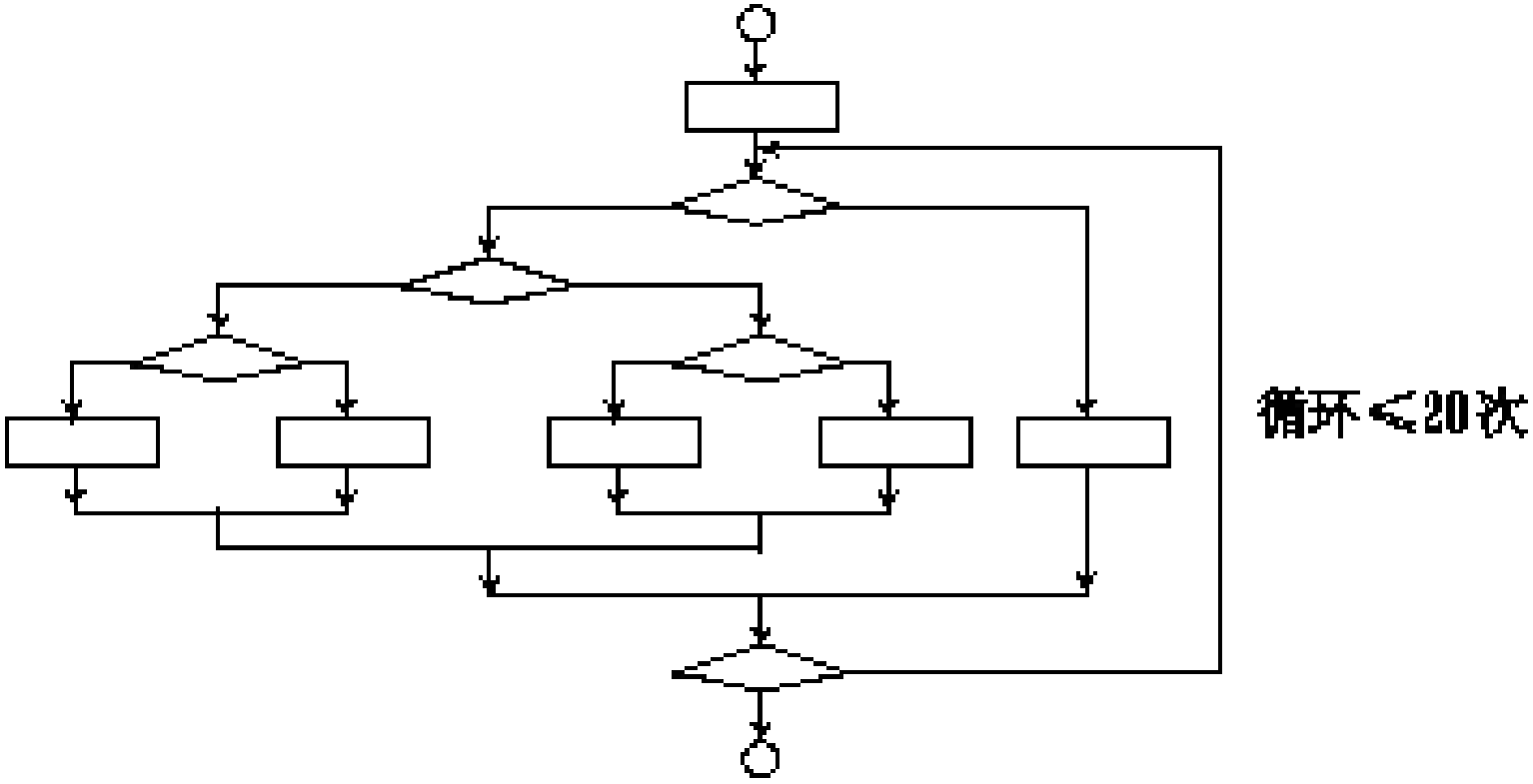
     不验证规格的正确性

测试覆盖标准

- 穷举测试不可行

  白盒法考虑的是测试用例对程序内部逻辑的覆盖程度。最彻底的白盒法是覆盖程序中的每一条路径，但是由于程序中一般含有循环，所以路径的数目极大，要执行每一条路径是不可能的，只能希望覆盖的程度尽可能高些。

测试覆盖标准



测试覆盖标准

- 上页小程序的流程图

  其中包括了一个20次的循环。那么它所包含的不同执行路径数为5^{20}（$\approx 10^{13}$）条，若要对它进行穷举测试，覆盖所有路径。假使测试程序对每一条路径进行测试需1毫秒，假定一天工作24小时，一年工作365 天， 那么要想所有路径测试完，需3170年。

- 上页小程序即使每条路径都测过，仍可能存在错误。因为：
  - 穷举路径测试无法检查出程序本身是否违反了设计规范，即程序是否是一个错误的程序。
  - 穷举路径测试不可能查出程序因为遗漏路径而出错。
  - 穷举路径测试发现不了一些与数据相关的错误。

测试覆盖标准

- 为了衡量测试的覆盖程度，需建立一些标准。

测试覆盖率可以表示出测试的充分性，在测试分析报告中可以作为量化指标的依据，测试覆盖率越高效果越好。

- 测试覆盖率

用于确定测试所执行到的覆盖项的百分比。测试覆盖率包括功能点覆盖率和逻辑覆盖率：

功能点覆盖率大致用于表示软件已经实现的功能与软件需要实现的功能之间的比例关系。

逻辑覆盖率指程序逻辑的覆盖率，可分为语句覆盖、判定覆盖、条件覆盖、判定/条件覆盖、组合覆盖和路径覆盖。

测试覆盖标准

- 覆盖标准从低到高分别是：
  - 语句覆盖 SC（Statement Coverage）

    是一个较弱的测试标准，它的含义是：选择足够的测试用例，使得程序中每个语句至少都能被执行一次。

    它是最弱的逻辑覆盖，效果有限，必须与其它方法交互使用。
  - 判定覆盖（也称为分支覆盖） DC（Decision coverage）

    执行足够的测试用例，使得程序中的每一个分支至少都通过一次。

    判定覆盖只比语句覆盖稍强，但实际效果表明，只是判定覆盖，还不能保证一定能查出在判断条件中存在的错误。因此，还需要更强的逻辑覆盖准则去检验判断内部条件。

测试覆盖标准

- 条件覆盖 CC（Condition Coverage）

执行足够的测试用例，使程序中每个判断的每个条件的每个可能取值至少执行一次；

条件覆盖深入到判定中的每个条件，但可能不满足判定覆盖的要求。

- 判定/条件覆盖 CDC（Condition/ Decision Coverage）

执行足够的测试用例，使得判定中每个条件取到各种可能的值，并使每个分支取到各种可能的结果。

判定/条件覆盖有缺陷：从表面看，它测试了所有条件的取值，但其实往往是某些条件掩盖了另一些条件，会遗漏某些条件取值错误的情况。

测试覆盖标准

- 条件组合覆盖 MCC （Multiple Condition Coverage）

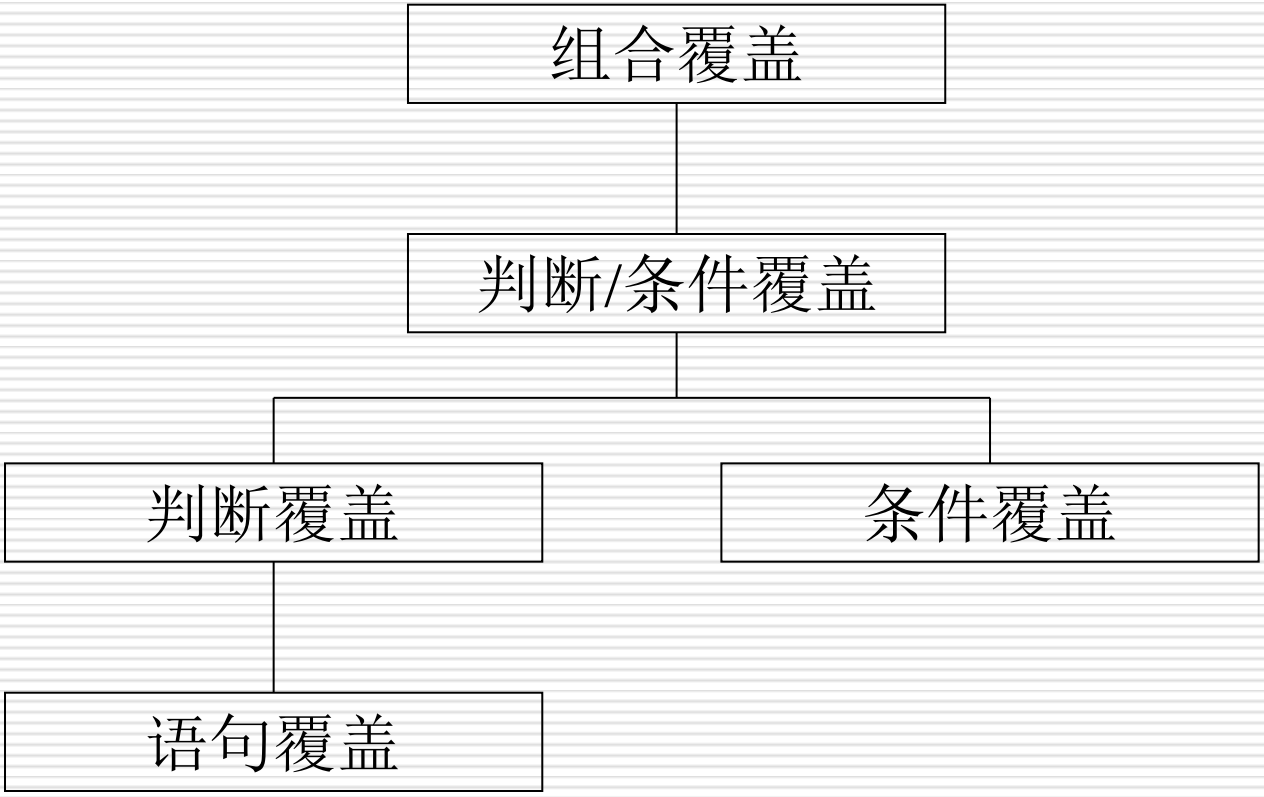
  执行足够的例子，使得每个判定中条件的各种可能组合都至少出现一次。

  这是一种相当强的覆盖准则，可以有效地检查各种可能的条件取值的组合是否正确。它不但可覆盖所有条件的可能取值的组合，还可覆盖所有判断的可取分支，但可能有的路径会遗漏掉。测试还不完全。

- 路径覆盖

  设计足够多的测试用例，要求覆盖程序中所有可能的路径。

测试覆盖标准



逻辑驱动测试

- 概述
- 语句覆盖
- 判定覆盖（分支覆盖）
- 条件覆盖
- 判定/条件覆盖
- 修正条件判定覆盖
- 条件组合覆盖
- 综合举例

白盒测试的主要方法：

- 逻辑驱动测试

  设计足够多的测试用例，运行所测程序，满足某种测试覆盖率要求。基本的有：

  - 语句覆盖
  - 判定覆盖（也称为分支覆盖）
  - 条件覆盖
  - 判定/条件覆盖
  - 条件组合覆盖

- 基本路径测试

  设计足够多的测试用例，运行所测程序，要覆盖程序中所有可能的路径。这是最强的覆盖准则。但在路径数目很大时，真正做到完全覆盖是很困难的，必须把覆盖路径数目压缩到一定限度。

语句覆盖

- 语句覆盖

  一个较弱的测试标准，它的含义是：选择足够的测试用例，使得程序中每个语句至少都能被执行一次。

  - 如，例1：

```
PROCEDURE  M(VAR A，B，X：REAL);
BEGIN
    IF (A>1) AND (B=0)  THEN X:=X/A;
    IF (A=2) OR (X>1)   THEN X:=X+1;
END.
```

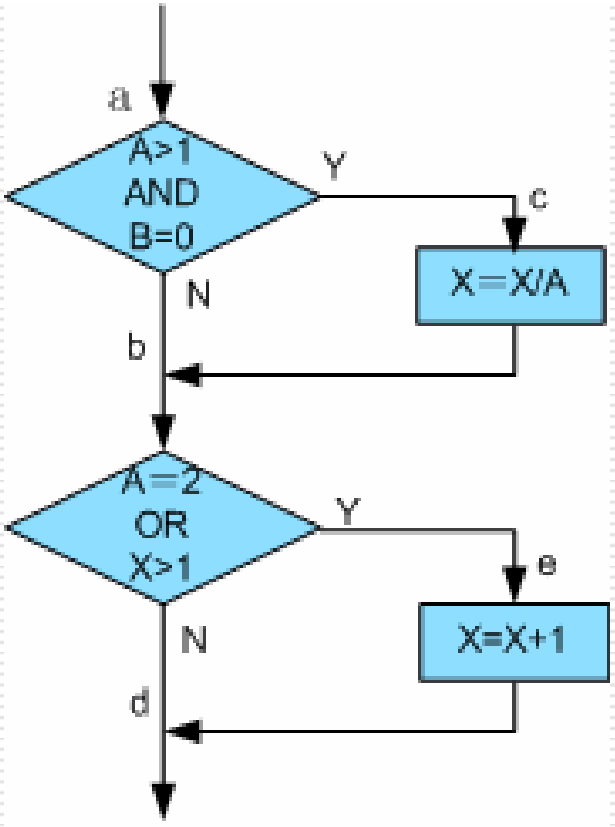
语句覆盖

为使程序中每个语句至少执行一次，只需设计一个能通过路径ace的例子就可以了，如选择输入数据为：

A=2，B=0，X=3

（期望输出：A,B不变，X=2.5）

就可达到“语句覆盖”标准。



语句覆盖

```
PROCEDURE M(VAR A , B , X : REAL) ;
BEGIN
    IF (A>1) AND (B=0) THEN X:=X/A ;
    IF (A=2) OR (X>1)  THEN X:=X+1 ;
END.
```

语句覆盖

优点：可很直观地从源代码得到测试用例，无须细分每条判定表达式。

缺点：从上例可看出，语句覆盖是很弱的，发现不了判定中逻辑运算的错误，即它并不是一种充分的检验方法。如：

1）如果第一个条件语句中的AND错误地写成OR，这个测试用例不能发现这个错误；

2）如第三个条件语句中X＞1误写成X＞0，这个测试用例也不能暴露它；

3）沿着路径abd执行时，X的值应该保持不变，如果这一方面有错误，测试数据也不能发现。

```
PROCEDURE M(VAR A, B, X: REAL);
BEGIN
    IF (A>1) AND (B=0)  THEN X:=X/A;
    IF (A=2) OR (X>1)   THEN X:=X+1;
END.
```

A=2，B=0，X=3

语句覆盖

- 例2：

```
void  DoWork(int x,int y,int z)
{  int  k=0,j=0;
   if((x>3)&&(z<10))
   {       k=x*y-1;      //语句块1
           j=sqrt(k);
   }
   if((x= =4)||(y>5))
           j=x*y+10;    //语句块2
   j=j%3;               //语句块3
}
```

语句覆盖

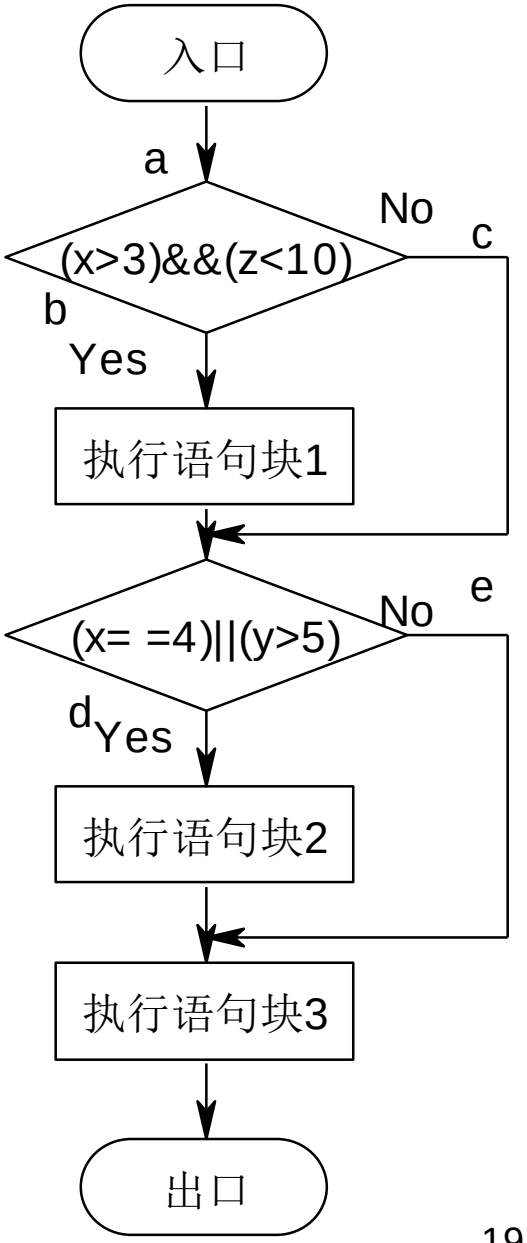
为了满足语句覆盖率，只要设计一个测试用例就可把三个执行语句块中的语句覆盖。

测试用例输入为：

x=4、y=5、z=5

程序执行的路径是：abd 。

```
void  DoWork(int x,int y,int z)
{  int  k=0,j=0;
   if((x>3)&&(z<10))
   {     k=x*y-1;              //语 句 块1
         j=sqrt(k);}
   if((x= =4)||(y>5))
      j=x*y+10;              //语 句 块2
    j=j%3;                   //语 句 块3
}
```



语句覆盖

该测试用例虽然覆盖了可执行语句，但并不能检查判断逻辑是否有问题，例如在第一个判断中把&&错误的写成了||，则上面的测试用例仍可以覆盖所有的执行语句。

- 一般认为“语句覆盖”是很不充分的一种标准，是最弱的逻辑覆盖准则。

```
void  DoWork(int x,int y,int z)
{  int  k=0,j=0;
   if((x>3)&&(z<10))
   {     k=x*y-1;              //语 句 块1
         j=sqrt(k); }
   if((x= =4)||(y>5))
       j=x*y+10;              //语 句 块2
   j=j%3;                     //语 句 块3
}
```

x=4、y=5、z=5

分支覆盖

- 分支覆盖

  比“语句覆盖”稍强的覆盖标准是“分支覆盖”(或称判定覆盖)标准。

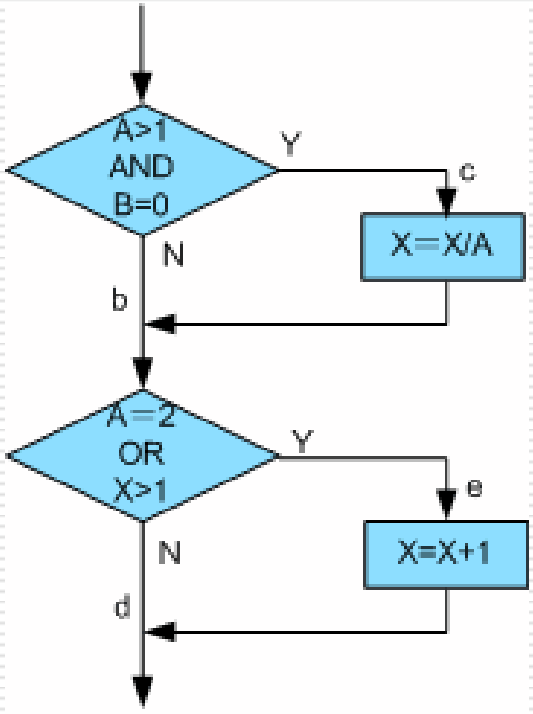
  含义是：执行足够的测试用例，使得程序中的每一个分支至少都通过一次。

分支覆盖

对例1，如设计两个测试用例，使它们能通过路径ace和abd，或者通过路径acd和abe，就可达到“判定覆盖”标准，为此，可以选择输入数据为：

① A=3，B=0，X=1 (沿路径acd执行)；

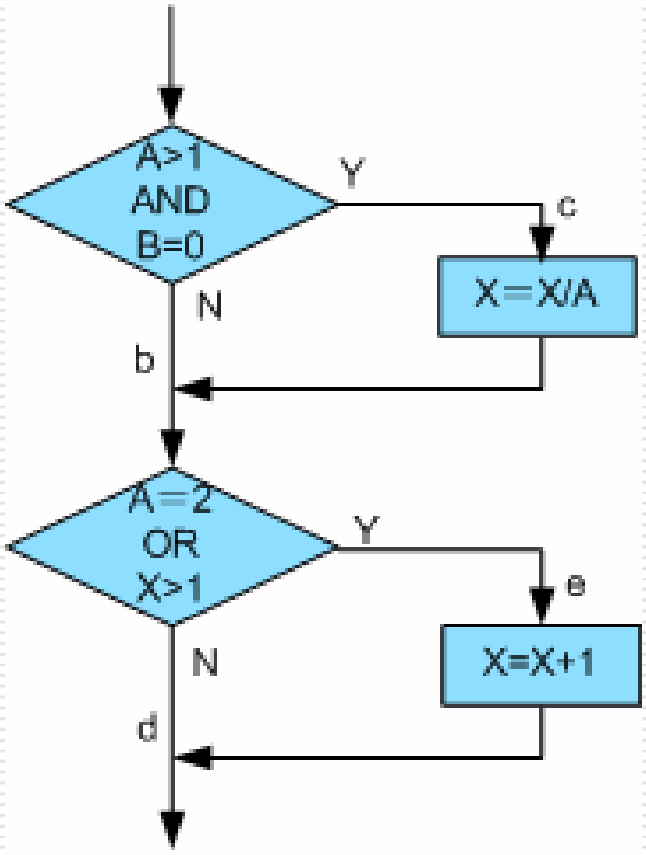
② A=2，B=1，X=3(沿路径abe执行)



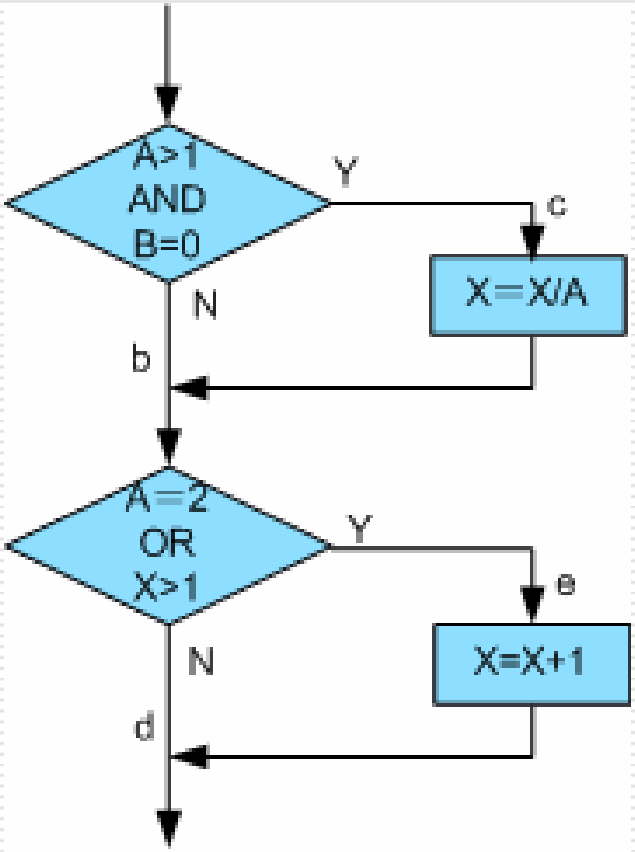

```
PROCEDURE M(VAR A , B , X : REAL) ;
BEGIN
    IF (A>1) AND (B=0) THEN X:=X/A ;
    IF (A=2) OR (X>1)  THEN X:=X+1 ;
END.
```

分支覆盖

A=3，B=0，X=1 (沿路径acd执行)
A=2，B=1，X=3 (沿路径abe执行)



判定覆盖



判定覆盖

分支覆盖

- “分支覆盖”比“语句覆盖”严格

如果每个分支都执行过了，则每个语句也就执行过了。

针对语句覆盖中不能检测的错，在分支覆盖中：

① A=3，B=0，X=1 (沿路径acd执行)；

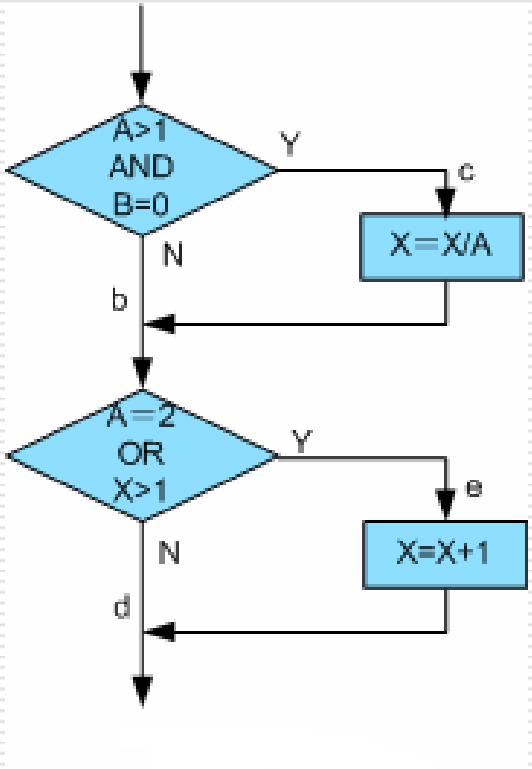
② A=2，B=1，X=3 (沿路径abe执行)

1）如第一个条件语句中的AND错写成OR，则第二个测试用例能发现；

2）如第三个条件语句中X＞1误写成X＞0，第一个测试用例能暴露它；

3）沿着路径abd执行时，X的值应保持不变，如这方面有错，不能发现。

所以，“分支覆盖”还很不够。



分支覆盖

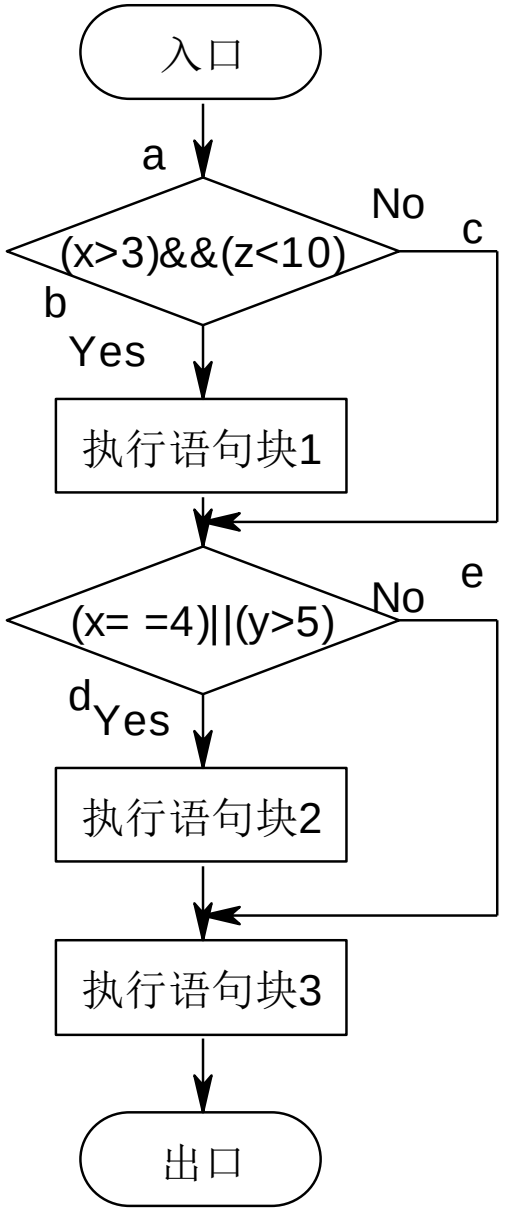
对例2，如设计两个测试用例则可以满足分支覆盖的要求。

测试用例的输入为：

x=4、y=5、z=5 (沿路径abd执行)

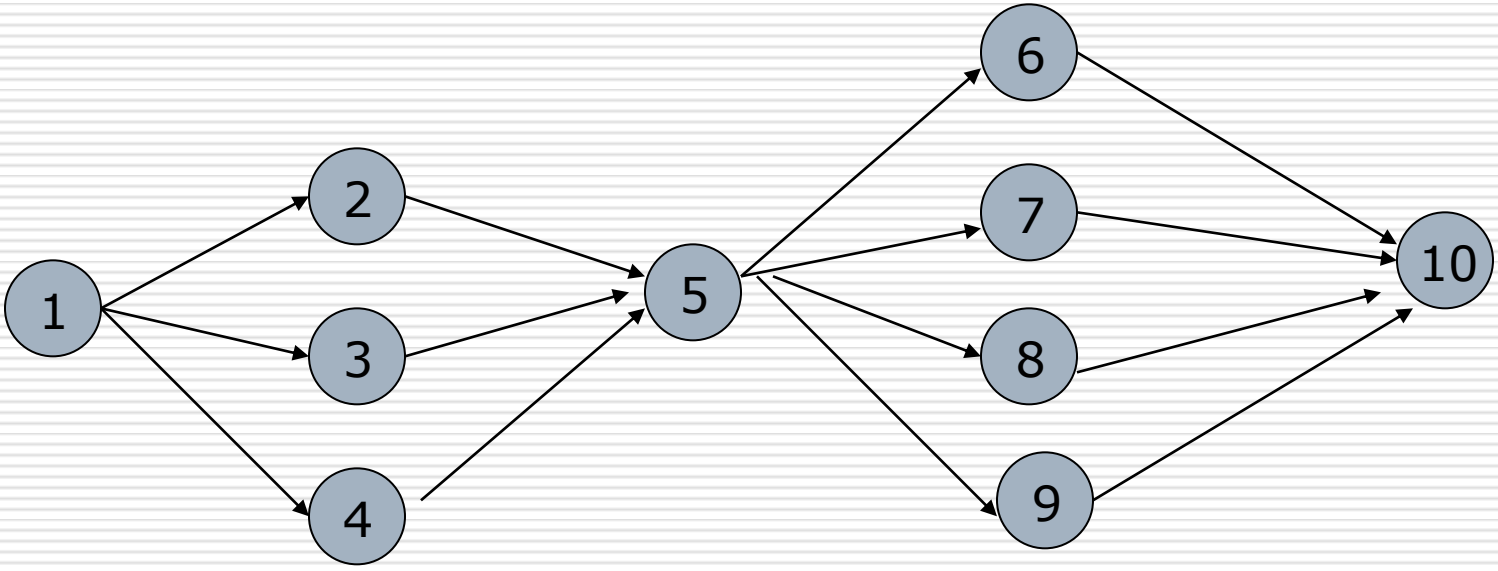
x=2、y=5、z=5 (沿路径ace执行)

上面的两个测试用例虽能满足分支覆盖要求，但也不能对判断条件进行检查，如把第二个条件y>5错写成y<5,、上面的测试用例同样满足了分支覆盖。



分支覆盖

- 说明：以上仅考虑了两出口的判断，我们还应把判定覆盖准则扩充到多出口判断（如Case语句）的情况。因此，判定覆盖更为广泛的含义应该是使得每一个判定获得每一种可能的结果至少一次。



条件覆盖

- 条件覆盖

  一个判定中往往包含若干个条件，如例1，判定 (A＞1) AND (B=0)包含了两个条件： A＞1以及 B=0，所以可引进一个更强的覆盖标准——“条件覆盖”。

  “条件覆盖”的含义是：执行足够的测试用例，使得判定中的每个条件获得各种可能的结果。

条件覆盖

例1的程序有四个条件：

A＞1、 B=0、A=2、X＞1

为达到“条件覆盖”标准，需执行足够的测试用例使得在a点有：

A＞1、A≤1、B=0、B≠0

等各种结果出现，以及在b点有：

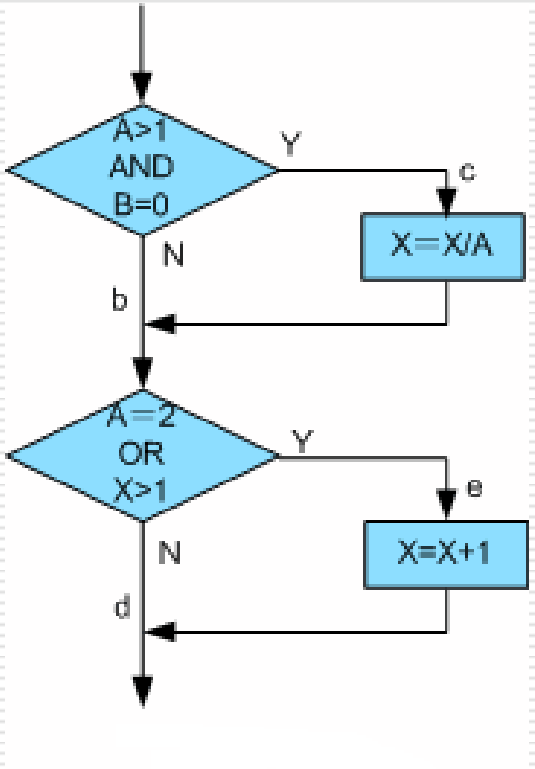
A=2、A≠2、X＞1、X≤1

等各种结果出现。

现在只需设计以下两个测试用例就可满足这一标准：

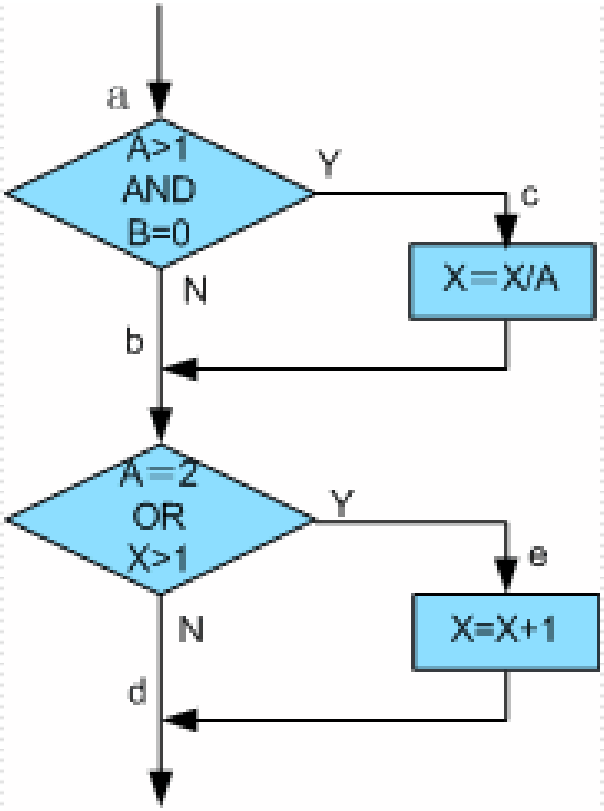
① A=2，B=0，X=4 (沿路径ace执行)；

② A=1，B=1，X=1 (沿路径abd执行)。

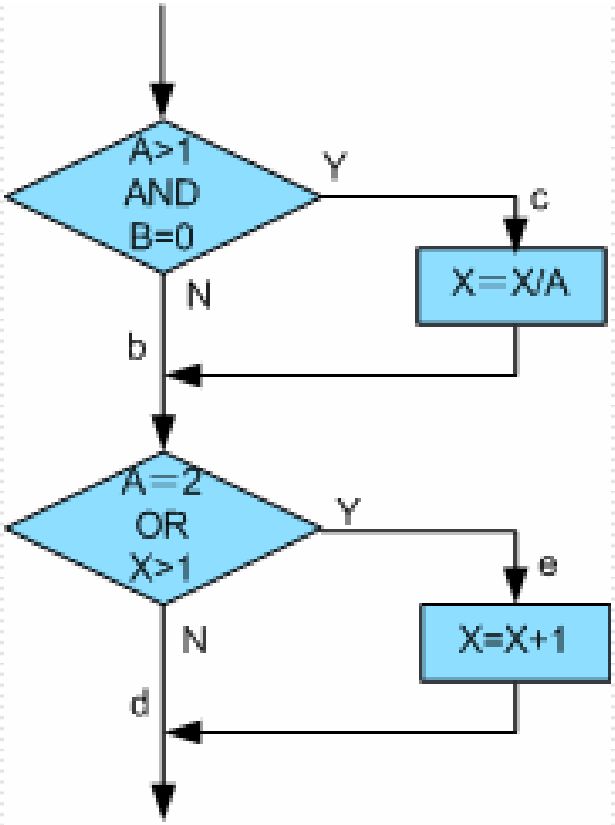


条件覆盖

A=2，B=0，X=4（沿路径ace执行）
A=1，B=1，X=1（沿路径abd执行）



条件覆盖



条件覆盖

条件覆盖

对例2中的所有条件取值加以标记。

对于第一个判断：

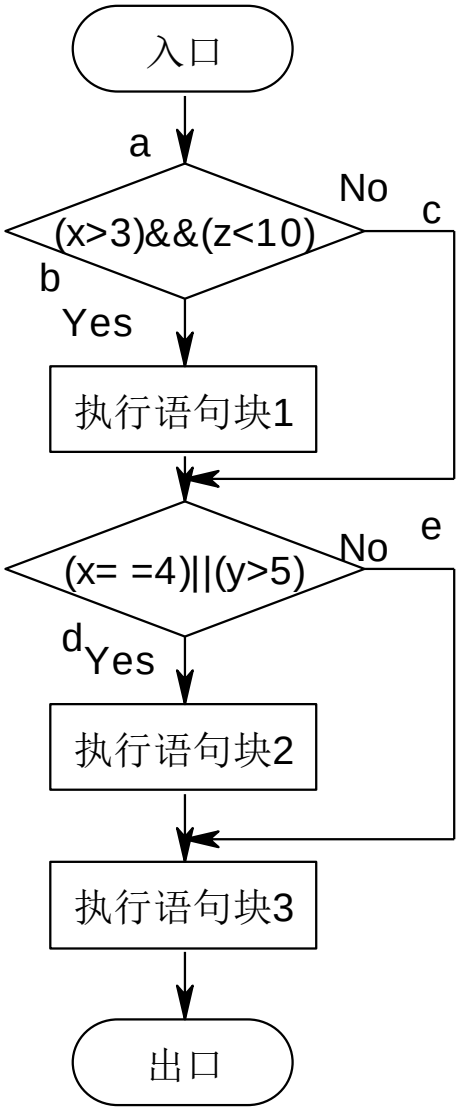
条件x>3 取真值为T1，取假值为-T1

条件z<10 取真值为T2，取假值为-T2

对于第二个判断：

条件x=4 取真值为T3，取假值为-T3

条件y>5 取真值为T4，取假值为-T4

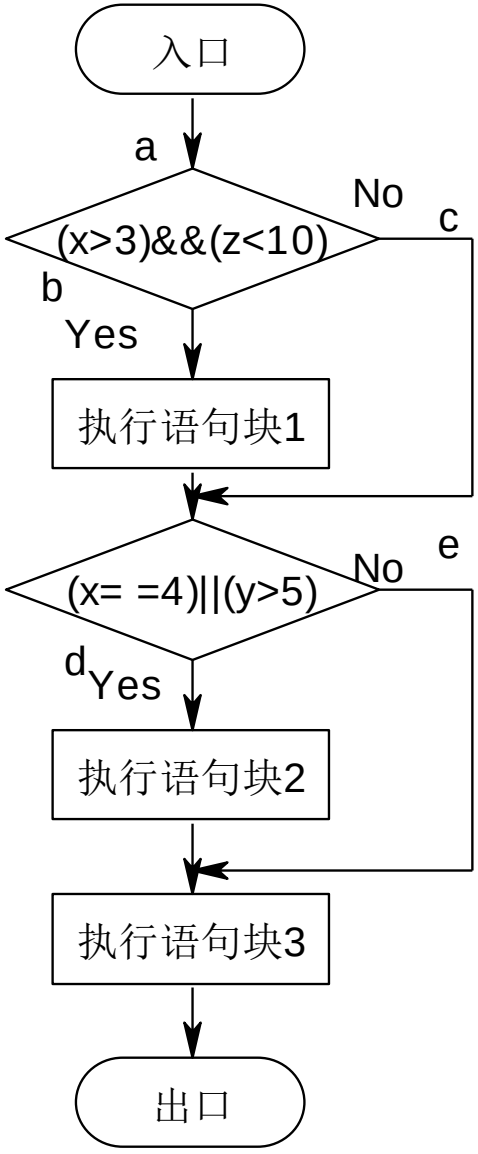


条件覆盖

则可以设计测试用例如下

测试用例	通过路径	条件取值	覆盖分支
x=4、y=6、z=5	abd	T1、T2、T3、T4	bd
x=2、y=5、z=5	ace	-T1、T2、-T3、-T4	ce
x=4、y=5、z=15	acd	T1、-T2、T3、-T4	cd

上面的测试用例不但覆盖了所有分支的真假两个分支，而且覆盖了判断中的所有条件的可能值。



条件覆盖

- “条件覆盖”并不包含“分支覆盖”

  如对语句

  IF(A AND B) THEN S

  设计2个测试用例，使其满足“条件覆盖”：

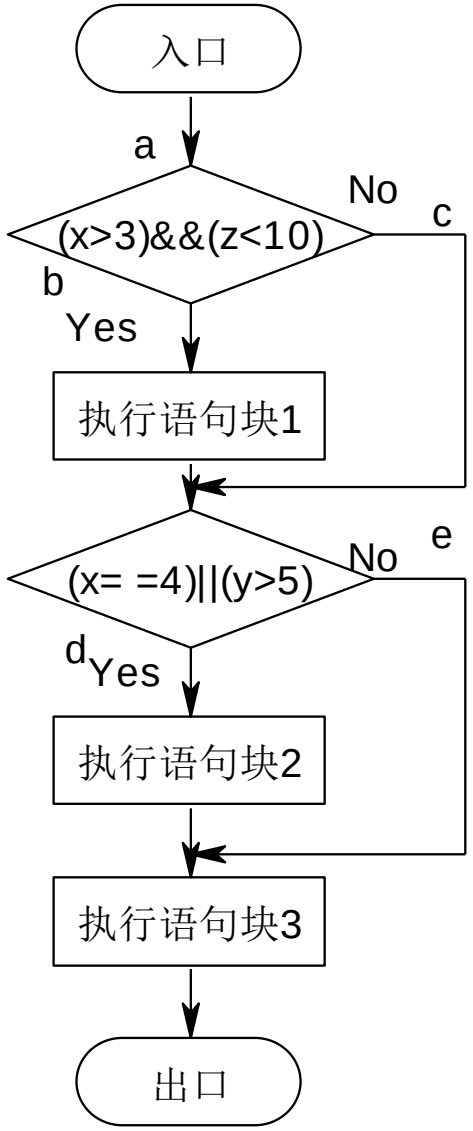
  1）使A为真并使B为假

  2）使A为假而且B为真

  但是它们都未能使语句S得以执行。

条件覆盖

如对例2设计了下面的测试用例，则虽然满足了条件覆盖，但只覆盖了第一个条件的取假分支和第二个条件的取真分支，不满足分支覆盖的要求。



测试用例	通过路径	条件取值	覆盖分支
x=2、y=6、z=5	acd	-T1、T2、-T3、T4	cd
x=4、y=5、z=15	acd	T1、-T2、T3、-T4	cd

分支/条件覆盖

- 分支/条件覆盖

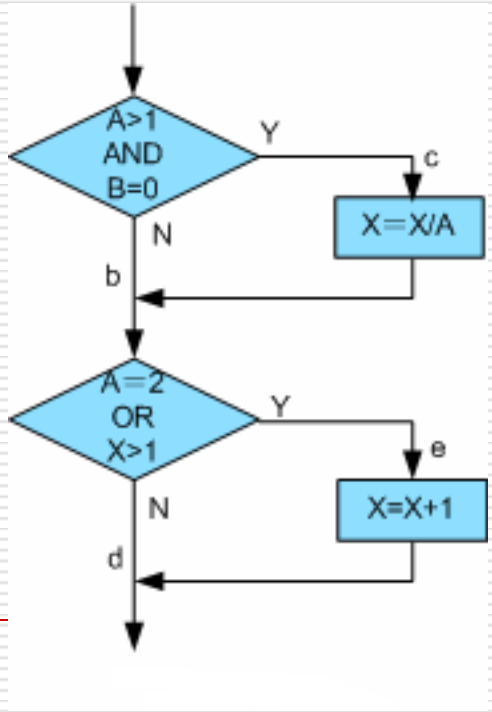
针对上面问题引出了另一种覆盖标准——“分支 / 条件覆盖”，它的含义是：执行足够的测试用例，使得分支中每个条件取到各种可能的值，并使每个分支取到各种可能的结果。

- 对例1的程序，前面的两个例子

  ① A=2，B=0，X=4 (沿ace路)

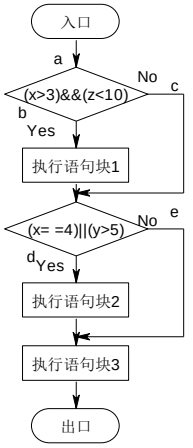
  ② A=1，B=1，X=1 (沿abd路径)

是满足这一标准的。



分支/条件覆盖

对例2，根据定义只需设计以下两个测试用例便可以覆盖8个条件值以及4个判断分支。



测试用例	通过路径	条件取值	覆盖分支
x=4、y=6、z=5	abd	T1、T2、T3、T4	bd
x=2、y=5、z=11	ace	-T1、-T2、-T3、-T4	ce

分支/条件覆盖

- 分支/条件覆盖从表面来看，它测试了所有条件的取值，但是实际上某些条件掩盖了另一些条件。

  如对于条件表达式 (x>3) && (z<10) 来说，必须两个条件都满足才能确定表达式为真。如果（x>3）为假，则一些编译器不再判断是否z<10了。

  对于第二个表达式（x==4）||（y>5）来说，若x==4测试结果为真，就认为表达式的结果为真，这时不再检查（y>5）条件了。

  因此，采用分支/条件覆盖，逻辑表达式中的错误不一定能够查出来了。

条件组合覆盖

- 条件组合覆盖

  针对上述问题又提出了另一种标准——“条件组合覆盖”。

  它的含义是：执行足够的例子，使得每个判定中条件的各种可能组合都至少出现一次。

  显然，满足“条件组合覆盖”的测试用例是一定满足“分支覆盖”、“条件覆盖”和“分支/条件覆盖”的。

条件组合覆盖

再看例1，需选择适当的例子，使得下面8种条件组合都能够出现：

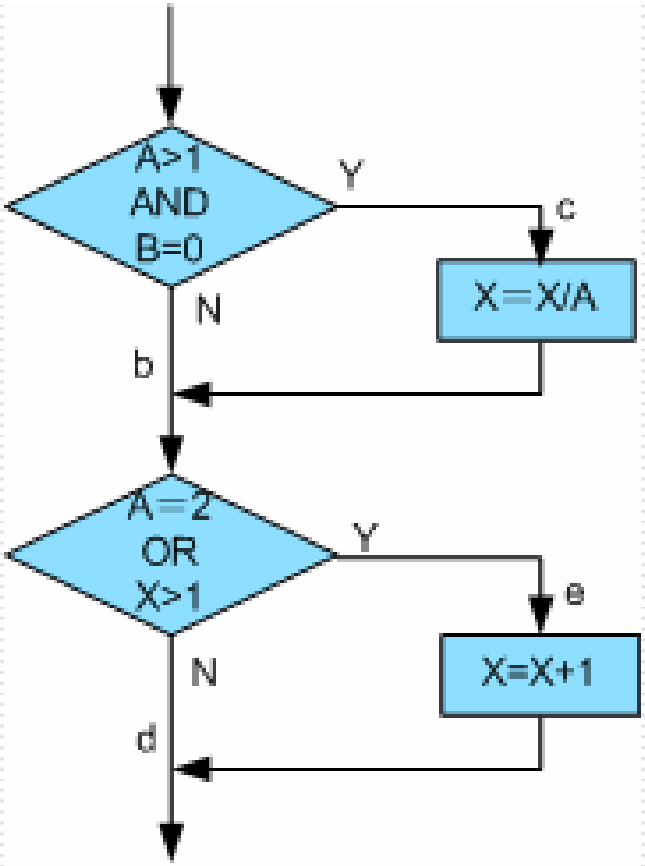
1）A>1，B=0　　2）A>1，B≠0

3）A≤1，B=0　　4）A≤1，B≠0

5）A=2，X>1　　6）A=2，X≤1

7）A≠2，X>1　　8）A≠2，X≤1

5）、 6）、 7）、8）四种情况是第二个IF语句的条件组合，而X的值在该语句之前是要经过计算的，所以还必须根据程序的逻辑推算出在程序的入口点X的输入值应是什么。



条件组合覆盖

下面设计的四个例子可以使上述8种条件组合至少出现一次：

① A=2，B=0，X=4

使 1)、5)两种情况出现；

② A=2，B=1，X=1

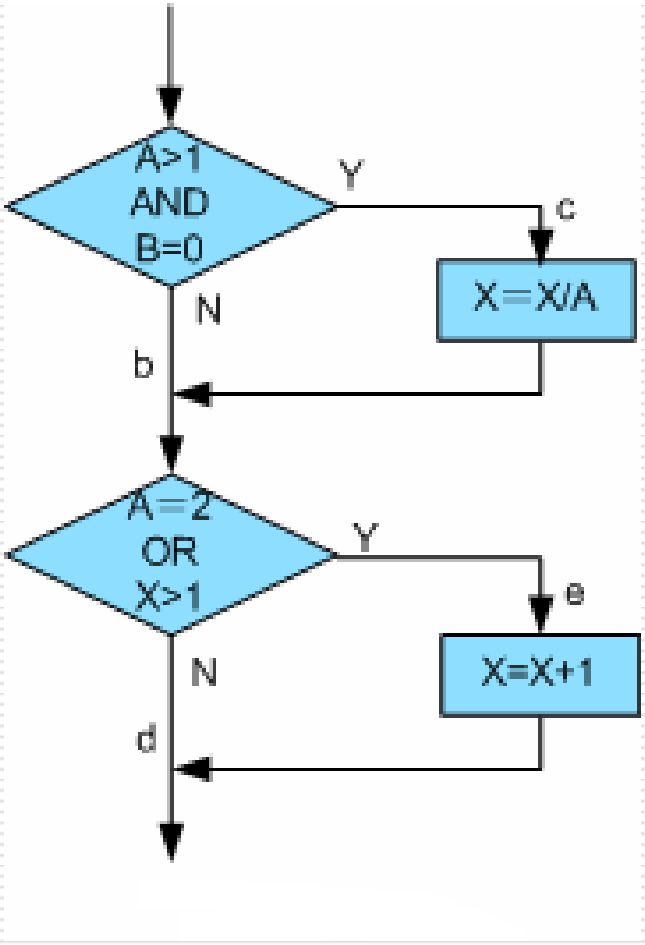
使 2)、6)两种情况出现；

③ A=1，B=0，X=2

使 3)、7)两种情况出现；

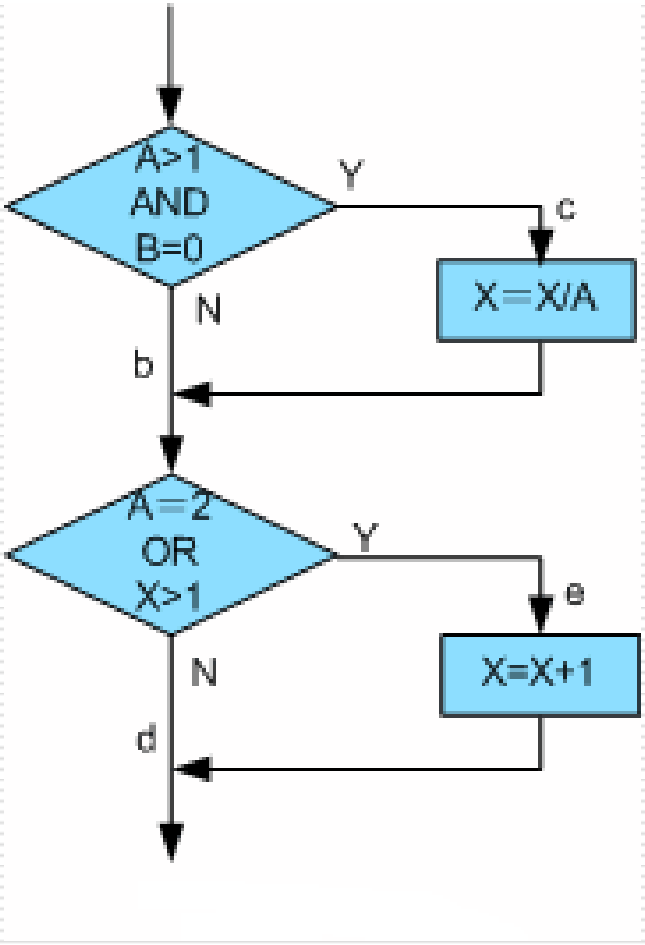
④ A=1，B=1，X=1

使 4)、8)两种情况出现。



条件组合覆盖

上面四个例子虽然满足条件组合覆盖，但并不能覆盖程序中的每一条路径，例如路径acd就没有执行，因此，条件组合覆盖标准仍然是不彻底。



条件组合覆盖

对例2中的各个判断的条件取值组合加以标记如下：

1、x>3,z<10　记做T1 T2，第一个判断的取真分支

2、x>3,z>=10　记做T1 -T2，第一个判断的取假分支

3、x<=3,z<10　记做-T1 T2，第一个判断的取假分支

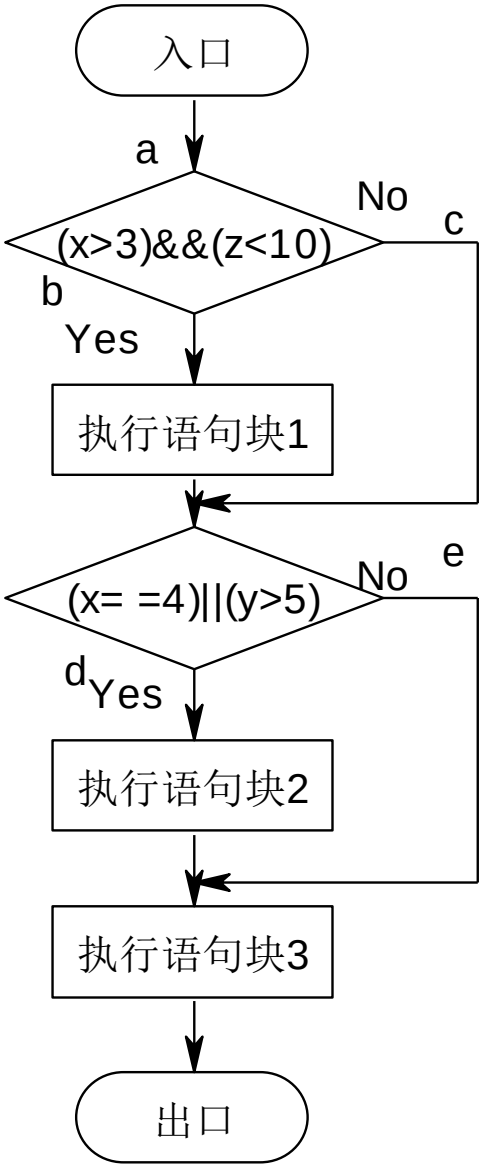
4、x<=3,z>=10 记做-T1 -T2，第一个判断的取假分支

5、x=4,y>5　记做T3 T4，第二个判断的取真分支

6、x=4,y<=5　记做T3 -T4，第二个判断的取真分支

7、x!=4,y>5　记做-T3 T4，第二个判断的取真分支

8、x!=4,y<=5　记做-T3 -T4，第二个判断的取假分支



条件组合覆盖

根据定义取4个测试用例，就可以覆盖上面8种条件取值的组合。
测试用例如下表：

测试用例	通过路径	条件取值	覆盖组合号
x=4、y=6、z=5	abd	T1、T2、T3、T4	1和5
x=4、y=5、z=15	acd	T1、-T2、T3、-T4	2和6
x=2、y=6、z=5	acd	-T1、T2、-T3、T4	3和7
x=2、y=5、z=15	ace	-T1、-T2、-T3、-T4	4和8

上面的测试用例覆盖了所有条件的可能取值的组合，覆盖了所有判断的可取分支，但是却丢失了一条路径abe。

修正条件判定覆盖

- 修正条件判定覆盖

  它是由欧美的航空/航天制造厂商和使用单位联合制定的“航空运输和装备系统软件认证标准”，目前在国外的国防、航空航天领域应用广泛。

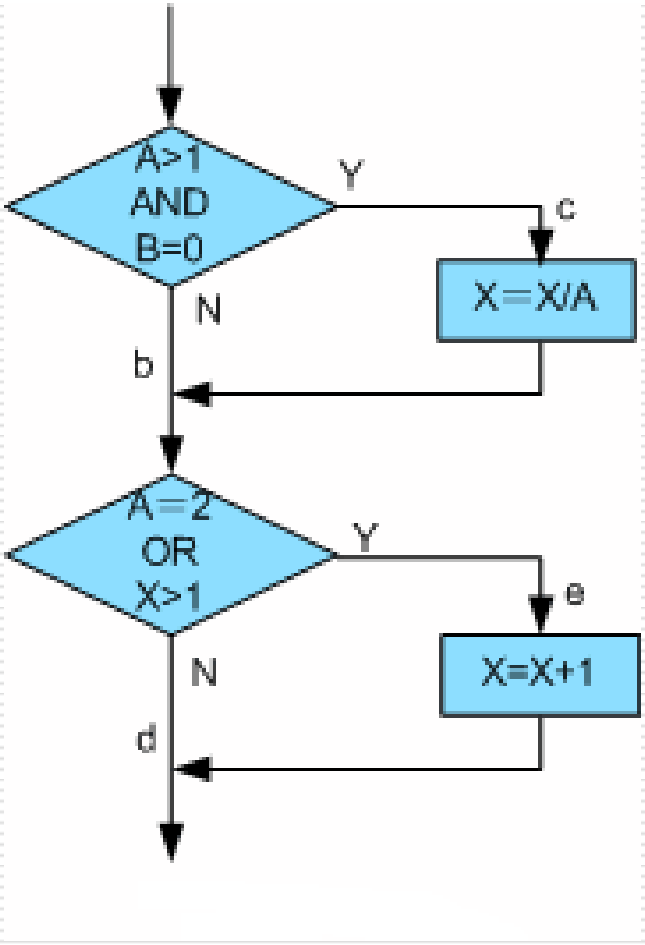
  修正条件判定覆盖方法是在分支/条件覆盖的基础上，加上判定中的每一个条件必须能够独立影响一个判定的输出，即在其他条件不变的前提下仅改变这个条件的值，而使判定结果改变。

修正条件判定覆盖

对例1的第一个判定，全部组合条件如下：

测试用例	A>1	B=0	结果
1	T	T	T
2	T	F	F
3	F	T	F
4	F	F	F

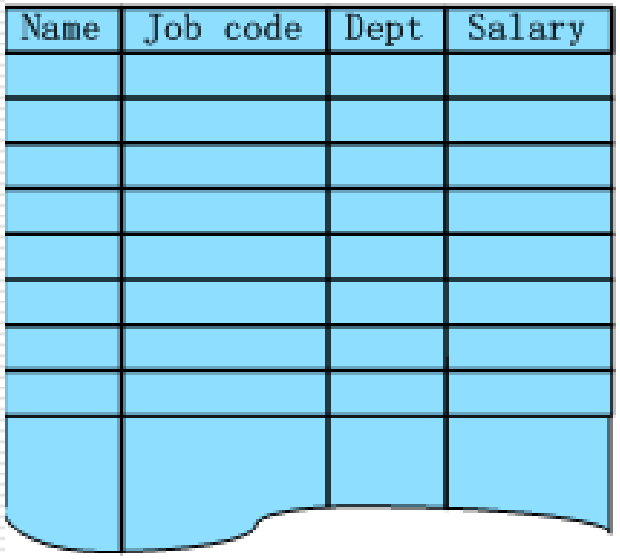
1和3说明条件“A>1”独立影响判定的输出；
1和2说明条件“B=0”独立影响判定的输出；
所以，测试用例1，2，3是必须的



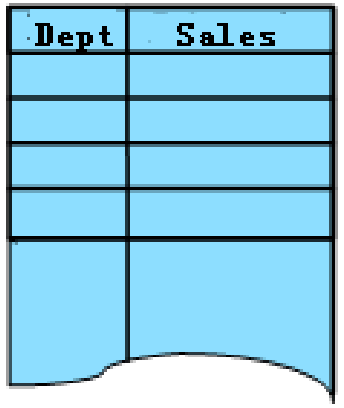
白盒法测试举例 - 工资管理程序测试

例3：工资管理程序BONUS的输入数据是职员表(Employee Table)和部门表(Department Table)(如图)。职员表由姓名(Name)、职务(Job Code)、部门(Dept.)和工资(Salary)四项组成，部门表由部门(Dept)和销售量(Sales)两项组成。

程序的功能是；“为销售量最大的部门中每一个职员增加200元工资，但是，如果某个职员的原有工资已达15000元，或者他的职务是经理，则只给他增加100元，如果程序能正常地完成，则输出出错码0；如果输入表格中没有任何条目，则输出出错码1；如果没有职员在部门表中销售量最大的部门中工作，则输出出错码2”。



Employee table



Department table

白盒法测试举例 - 工资管理程序测试

下面是BONUS的源程序，参数表中EMPTAB、DEPTTAB分别是职员表和部门表，ESIZE、DSIZE分别是两个表的长度，ERRCODE是出错码。

```
PROCEDURE BONUS(EMPTAB，DEPTTAB：TABLE；
    ESIZE，DSIZE，ERRCODE：
                INTEGER)；
……
```

```
1  BEGIN  MAXSALERS:=0；  ERRCODE:=0；
2        IF （ESIZE≤0）OR (DSIZE≤0)
3        THEN   ERRCODE:=1
4        ELSE
5           BEGIN FOR I:=1 TO DSIZE DO
6                       IF SALES(I)>MAXSALES
7                       THEN MAXSALES:=SALES(I)；
8                   FOR J:=1 TO DSIZE DO
9                       IF SALES(J):=MAXSALES
10                      THEN
11                      BEGIN FOUND:=FALSE;
12                              FOR K:=1 TO ESIZE DO
13                                 IF (EMPTAB.DEPT(K)=DEPTTAB.DEFT(J)）
14                                 THEN
15                                 BEGIN FOUND:=TRUE；
16                                       IF (SALARY(K)≥15000.00)
17                                                  OR (JOB(K)=“M”)
18                                       THEN SALARY(K):=SALARY(K)+100.00
19                                       ELSE SALARY(K):=SALARY(K)+200.00
20                                 END；
21                              IF（NOT FOUND）THEN ERRCODE:=2
22                      END
23           END
24  END.
```

白盒法测试举例 - 工资管理程序测试

现用白盒法设计测试用例。 首先列出程序中的判定，考虑所有的条件句和循环句。本例只要输入表格不空，循环句总会经历进入循环体和跳过循环体这两种情况(因为循环终值都大于等于循环初值)，所以就不必专门考虑了，需要分析的只是六个条件语句中的判定。

2	IF (ESIZE≤0) OR (DSIZE≤0)
6	IF (SALES(I)>MAXSALES)
9	IF (SALES(J)=MAXSALES)
13	IF (EMPTAB.DEPT(K)=DEPTTAB.DEFT(J))
16	IF (SALARY(K)≥15000.00) OR (JOB(K)=“M”)
21	IF (NOT FOUND)

白盒法测试举例 - 工资管理程序测试

1.采用“判定覆盖”标准设计测试用例

使得上述 6个判定都取到两种结果，这就需要以下12种情况出现。

白盒法测试举例 - 工资管理程序测试

判定	条件	结果为“真”	结果为“假”
2	(ESIZE≤0) OR (DSIZE≤0)	ESIZE或DSIZE为0	ESIZE和DSIZE都大于0
6	SALES(I)>MAXSALES	总会出现	部门表中，将销售量较小的某个部门放在后面
9	SALES(J)=MAXSALES	总会出现	部门表中，各部门的销售量不全相等
13	EMPTAB.DEPT(K) = DEPTTAB.DEFT(J)	职员表中，有职员在销售量最大的部门工作	职员表中，有个职员不在销售量最大的部门工作
16	(SALARY(K)≥15000.00) OR (JOB(K)=“M”)	销售量最大的部门中，有个职员的工资大于15000，或者是经理	销售量最大的部门中，有个职员不是经理，而且工资小于15000
21	NOT FOUND	销售量最大的部门中没有职员	销售量最大的部门中，有职员

白盒法测试举例 - 工资管理程序测试

设计下面的两个测试用例可以满足“判定覆盖”（图中“职务”一栏，“E”表示是一般职员，“M”表示是经理）。

例	输　　入	输　　出
1	ESIZE＝0	ERRCODE=1 ESIZE,DSIZE,EMPTAB, DEPTTAB　不变
2	ESIZE=DSIZE=3 EMPTAB　　DEPTTAB JONES E D42 21000.00　　D42 10000.00 SMITH E D32 14000.00　　D32 8000.00 LORIN E D42 10000.00　　D95 10000.00	ERRCODE=2 ESIZE,DSIZE,DEPTTAB 不变 EMPTAB JONES E D42 21100.00 SMITH E D32 14000.00 LORIN E D42 10200.00

白盒法测试举例 - 工资管理程序测试

虽然这两个例子满足“判定覆盖”标准，但是它们不能发现程序中许多其他可能的错误，例如没有检查ERRCODE为0、职员是经理、部门表为“空”等情况。

2.采用“条件覆盖”标准设计测试用例

必须使判定中的每一个条件取到两种可能的值，这就需要以下16种情况出现。

白盒法测试举例 - 工资管理程序测试

判定	条　　件	真	假
2	ESIZE≤0	ESIZE≤0	ESIZE>0
2	DSIZE≤0	DSIZE≤0	DSIZE>0
6	SALES(I) ≥ MAXSALES	总会出现	部门表中，各部门的销售量不相等
9	SALES(J) = MAXSALES	总会出现	部门表中，各部门的销售量不全相等
113	EMPTAB.DEPT(K)= DEPTTAB.DEPT(J)	职员表中，有职员在销售量最大的部门工作	有个职员不在销售量最大的部门工作
16	SALARY(K) ≥ 15000.00	销售量最大的部门中有个职员工资多于15000	销售量最大的部门中，有个职员工资小于 15000
16	JOB(K)=“M”	销售量最大的部门中，有个职员是经理	销售量最大的部门中，有个职员不是经理
21	NOT FOUND	某个销售量最大的部门中，没有职员	某个销售量最大的部门中，有职员

白盒法测试举例 - 工资管理程序测试

设计下面的两个测试用例可以满足“条件覆盖”。

例	输　　入	输　　出
1	ESIZE＝DSIZE=0	ERRCODE=1 ESIZE,DSIZE,EMPTAB, DEPTTAB　不变
2	ESIZE=DSIZE=3 EMPTAB　　DEPTTAB JONES E D42 21000.00　　D42 10000.00 SMITH E D32 14000.00　　D32 8000.00 LORIN M D42 10000.00　　D95 10000.00	ERRCODE=2 ESIZE,DSIZE,DEPTTAB 不变 EMPTAB JONES E D42 21100.00 SMITH E D32 14000.00 LORIN E D42 10100.00

白盒法测试举例 - 工资管理程序测试

尽管上面的测试用例满足“条件覆盖”标准，但它们不比满足“判定覆盖”标准的测试用例好，因为它们不能执行每一个语句(如语句19)，而且它们起的作用也不比满足“判定覆盖”的测试用例多许多，如未能使ERRCODE=0，如果语句2误写成(ESIZE=O) AND (DSIZE=0)，这个错误也不能被发现。

3.采用“判定 / 条件覆盖”标准设计测试用例

可克服“条件覆盖”中测试用例的弱点，需要提供足够的测试用例使得所有判定和条件都取到两个不同的值。

这里只需使“条件覆盖”测试用例中的职员JONES为经理，而使LORIN不是经理，则判定 16就可取到两种结果，语句19因而得以执行。

白盒法测试举例 - 工资管理程序测试

例	输　　入	输　　出
1	ESIZE＝DSIZE=0	ERRCODE=1 ESIZE,DSIZE,EMPTAB, DEPTTAB　不变
2	ESIZE=DSIZE=3 EMPTAB　　DEPTTAB JONES M D42 21000.00　　D42 10000.00 SMITH E D32 14000.00　　D32 8000.00 LORIN E D42 10000.00　　D95 10000.00	ERRCODE=2 ESIZE,DSIZE,DEPTTAB 不变 EMPTAB JONES E D42 21100.00 SMITH E D32 14000.00 LORIN E D42 10200.00

白盒法测试举例 - 工资管理程序测试

问题:如果所用的编译系统将含有“OR”的表达式处理成：遇到第一项为“真”就不再检查后面的项，则这样的两个测试用例并不能检查到 JOB(K)=“M”这一部分。

4.采用“条件组合覆盖”标准设计测试用例

它需要足够的例子，使得每个判定中条件的各种组合情况都出现一次。

本例中判定6、9、13和 21各有两种组合，判定2和16各有 4种组合。可以先选一个测试用例使其包含尽可能多的组合情况。再选另一测试用例使其包含尽可能多的余下的组合情况．．．，直至得到一组测试用例能包含所有的组合情况。

下面是满足“条件组合覆盖”标准的一组测试用例，它比前面几组测试用例都全面。

白盒法测试举例 - 工资管理程序测试

例	输　　入	输　　出
1	ESIZE=DSIZE=0	ERRCODE=1 ESIZE,DSIZE,EMPTAB, DEPTTAB　不变
2	ESIZE=0, DSIZE>0	同上
3	ESIZE>0, DSIZE=0	同上
4	ESIZE=5, DSIZE=4 EMPTAB　　DEPTTAB JONES M D42 21000.00　　D42 10000.00 WARNS M D95 12000.00　　D32 8000.00 LORIN E D42 10000.00　　D95 10000.00 TOY　E D95 16000.00　　D44 10000.00 SMITH E D32 14000.00	ERRCODE=2 ESIZE,DSIZE,DEPTTAB 不变 EMPTAB JONES M D42 21100.00 WARNS M D95 12100.00 LORIN E D42 10200.00 TOY　E D95 16100.00 SMITH E D32 14000.00

白盒法测试举例 - 工资管理程序测试

可以看出：即使是满足“条件组合覆盖”标准的例子仍不能发现BONUS中许多其他的错误。例如：

- 没有检查 ERRCODE＝0的情况，所以如果语句1中的 ERRORCODE:=0; 被遗漏了就查不出；
- 如语句16中 15000.00误写成 15000.01也是发现不了的，
- 如SALARY(K)＞=15000误写成SALARY(K)＞15000也是发现不了的；
- 如果 BONUS程序没有对部门表或职员表的最后一行进行处理，这个错误也不一定能发现。

白盒法测试举例 - 工资管理程序测试

- 通过前面例子的讨论，可以得到两点结论：
  - “条件组合覆盖”标准比其他标准优越。
  - 即使达到任何一种覆盖标准，其测试效果仍然是不彻底的，我们还需要用其他的测试方法作补充。

- 下面来讨论一下用黑盒法补充测试用例：

综合策略 - 黑盒法补充测试用例

- 白盒法和黑盒法各有长短，每种方法都可提供一组有用的测试用例，这组测试用例容易发现某种类型的错误，但不易发现其他类型的错误，然而没有一种方法能提供一组“完整的”测试用例。因此，实际软件测试方案设计是不同方法的综合应用。
- 一个参考的黑盒法补充策略是：

1) 在任何情况下都需使用边界值分析(这个方法应包括对输入和输出的边界值进行分析)。

2) 必要的话，再用等价分类法补充一些测试用例。

3) 再用错误推测法附加测试用例。

4) 检查上述例子的逻辑覆盖程度，如果未能满足某些覆盖标准，则再增加足够的测试用例。

5) 如果功能说明中含有输入条件的组合情况，则一开始就可先用因果图(判定表)法。

综合策略 - 黑盒法补充测试用例

以工资管理为例: (用黑盒法补充测试用例)

该例不必检查输入条件的组合情况，所以不需要用因果图(判定表)法，这里我们先用边界值分析法，该程序中输入的边界情况有：

1) EMPTAB具有 1个记录。

2*) EMPTAB具有最大个数的记录(如 65535个记录)。

3) EMPTAB具有零个记录。

4) DEPTAB具有 1个记录。

5*) DEPTTAB具有 65535个记录。

6) DEPTTAB具有零个记录，

7) 销售量最大的部门有1个职员。

黑盒法补充测试用例 - 工资管理程序测试

8*) 销售量最大的部门有 65535个职员。

9) 销售量最大的部门没有职员。

10) 所有部门的销售量相等。

11) DEPTTAB中，第一个部门的销售量最大。

12) DEPTTAB中，最后一个部门的销售量最大。

13) EMPTAB中，第一个职员在销售量最大的部门工作。

14) EMPTAB中，最后一个职员在销售量最大的部门工作。

15) 销售量最大的部门中有一个职员是经理。

16) 销售量最大的部门中有一个职员不是经理。

17) 销售量最大的部门中有一个职员(不是经理)的工资是 14 999.99。

黑盒法补充测试用例 - 工资管理程序测试

18) 销售量最大的部门中有一个职员(不是经理)的工资是 15 000.00。

19) 销售量最大的部门中有一个职员(不是经理)的工资是 15 000.01。

输出的边缘情况是：

20) ERRCODE=0。

21) ERRCODE=1。

22) ERRCODE=2。

23) 增加后的工资为 99999.99(即数据项SALARY的最大允许值)。

黑盒法补充测试用例 - 工资管理程序测试

再用错误推测法还可增加一个测试用例。

24) DEPTTAB中，在销售量最大但没有职员的部门之后，有一个销售量最大但有职员的部门(检查程序在产生ERRCODE＝2后是否会错误地终止对输入文件的处理)。

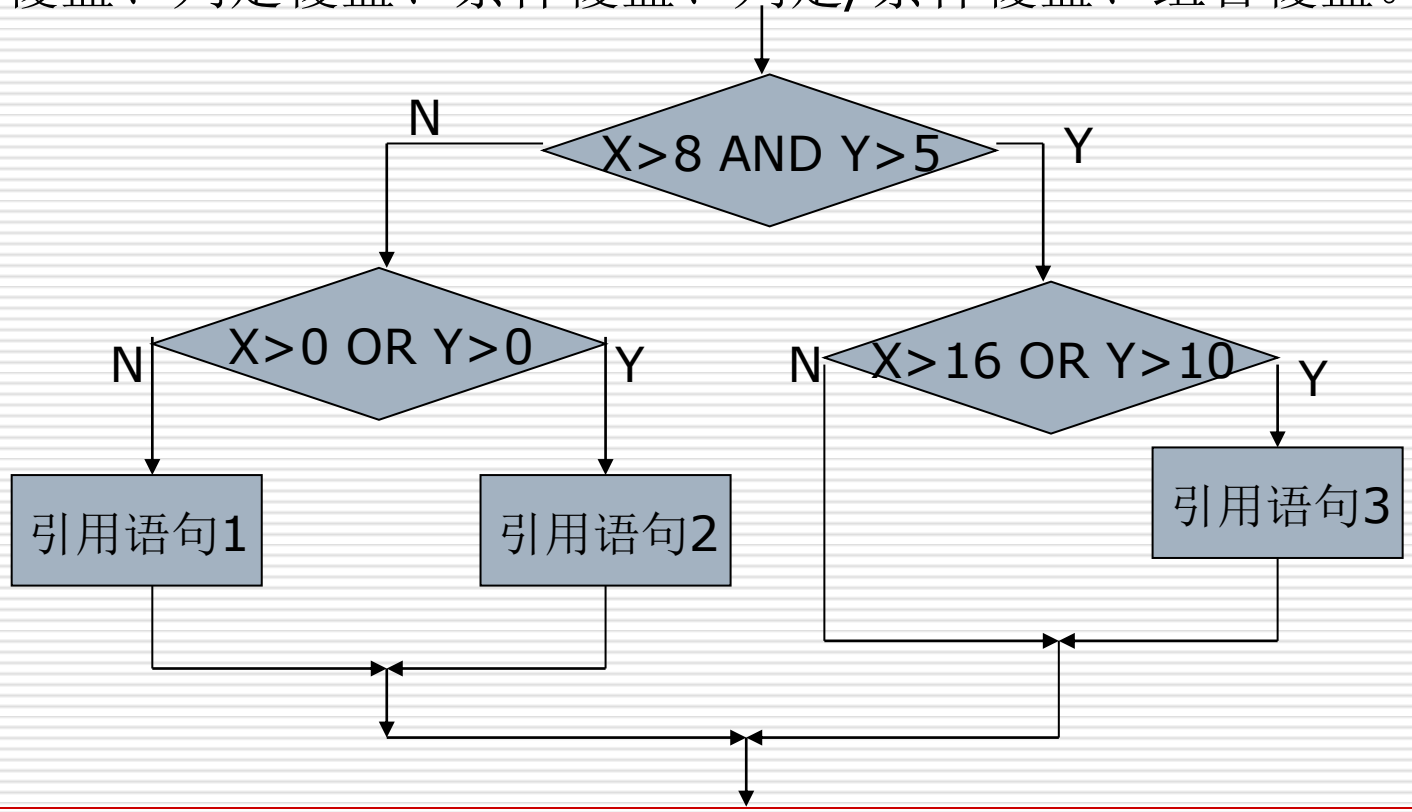
上述24种情况中，第 2)、5)、8)种情况不会发生，所以可以不考虑，再将用白盒法设计的测试用例与余下的21种情况作比较，可以看出许多情况已包括在这4个测试用例中了，尚未包括的情况是 1)、4)、7)、10)、14)、17)、18)、19)、20)、23)和24)等11种。

因此再增加的测试用例如下：

例	输　　入	输　　出
5	ESIZE=3 DSIZE=2 EMPTAB　　DEPTTAB ALLY　E D36 14999.99　　D33　55400.01 BEST　E D33 15000.00　　D36　55400.01 CELTO E D33 15000.01	ERRCODE=0 ESIZE,DSIZE,DEPTTAB不变 EMPTAB ALLY　E D36 15199.99 BEST　E D33 15100.00 ECLTO E D33 15100.01
6	ESIZE=1 DSIZE=1 EMPTAB　　DEPTTAB CHIEF M D99 99899.99　　D99 99000.00 雇员表中只有单个记录　　部门表中只有单个记录	ERRCODE=0 ESIZE,DSIZE,DEPTAB不变 EMPTAB CHIEF M D99 99999.99
7	ESIZE=2 DSIZE=2 EMPTAB　　DEPTTAB DOLE　E D67 10000.00　　D66 20000.00 FORD　E D22 33333.33　　D67 20000.00	ERRCODE=2 ESIZE,DSIZE,DEPTAB不变 EMPTAB DOLE　E D67 10200.00 FORD　E D22 33333.33

习题

- 为以下流程图所示的程序段设计一组测试用例，要求分别满足语句覆盖、判定覆盖、条件覆盖、判定/条件覆盖、组合覆盖。



路径覆盖测试

- 路径覆盖测试

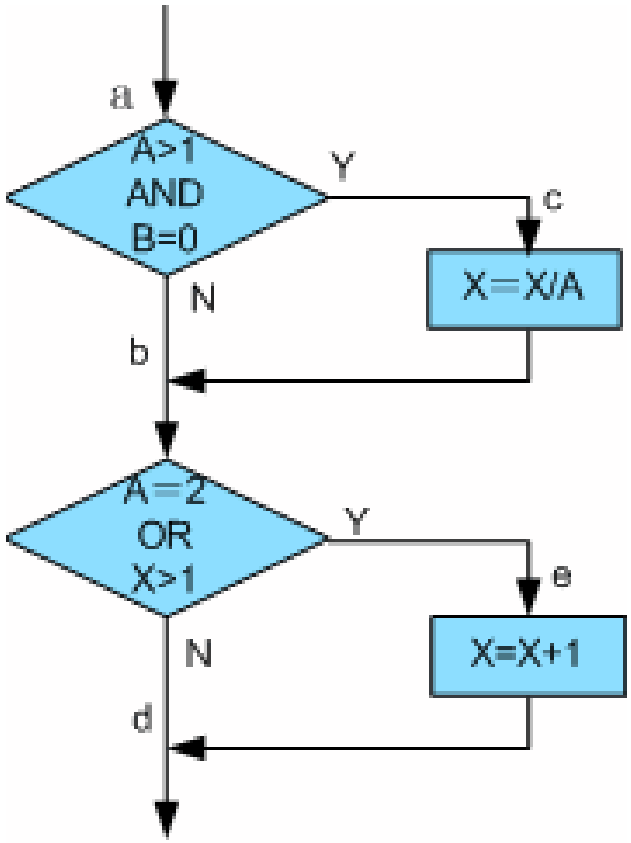
  前面提到的5种逻辑覆盖都未涉及到路径的覆盖。然而，只有当程序中的每一条路径都受到了检验，才能使程序受到全面检验。

  路径覆盖的目的就是要使设计的测试用例能覆盖被测程序中所有可能的路径。

路径测试

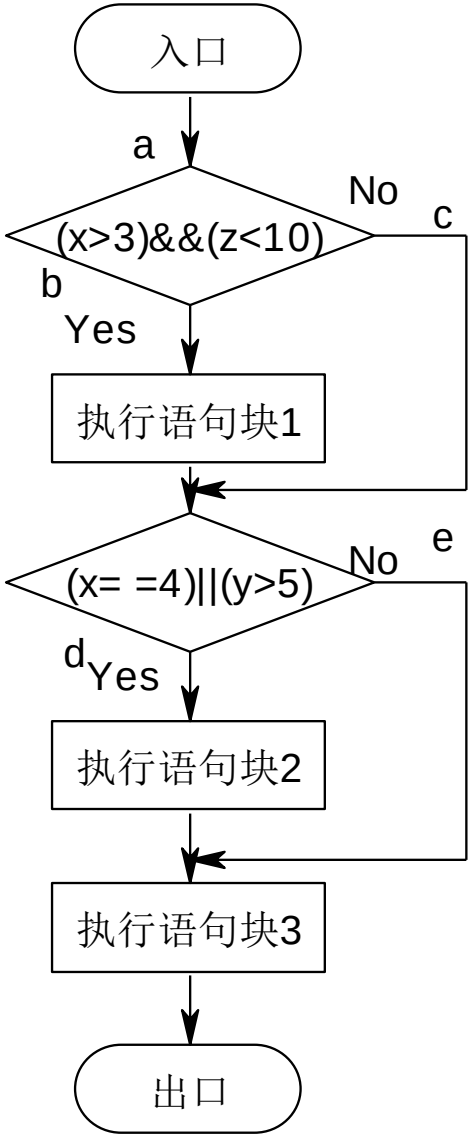
- 路径测试就是设计足够多的测试用例，覆盖被测试对象中的所有可能路径。

对于例1，下面的测试用例则可对程序进行全部的路径覆盖。



测试用例	通过路径
A=2、B=0、X=3	ace
A=1、B=0、X=1	abd
A=2、B=1、X=1	abe
A=3、B=0、X=1	acd

- 对于例2，下面的测试用例则可对程序进行全部的路径覆盖。



测试用例	通过路径	覆盖条件
x=4、y=6、z=5	abd	T1、T2、T3、T4
x=4、y=5、z=15	acd	T1、-T2、T3、-T4
x=2、y=5、z=15	ace	-T1、-T2、-T3、T4
x=5、y=5、z=5	abe	T1、T2、-T3、-T4

路径覆盖

- 满足路径覆盖不一定满足组合条件覆盖

  虽然前面一组测试用例满足了路径覆盖，但并没有覆盖程序中所有的条件组合（丢失了组合3和7：-T1、T2 和 -T3、T4 ），即满足路径覆盖的测试用例并不一定满足组合覆盖。

基本路径测试

- 基本路径测试

例1、例2都是很简单的程序函数，只有四条路径。但在实践中，一个不太复杂的程序，其路径都是一个庞大的数字，要在测试中覆盖所有的路径是不现实的。

为了解决这一难题，只得把覆盖的路径数压缩到一定限度内，例如，程序中的循环体只执行一次。

基本路径测试就是这样一种测试方法，它在程序控制图的基础上，通过分析控制构造的环行复杂性，导出基本可执行路径集合，从而设计测试用例的方法。设计出的测试用例要保证语句覆盖。

基本路径测试

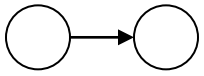
- 基本路径测试方法包括4个步骤：
  1. 画出程序的控制流图：描述程序控制流的一种图示方法。
  2. 计算程序圈复杂度：McCabe复杂性度量。从程序的环路复杂性可导出程序基本路径集合中的独立路径条数，这是确定程序中每个可执行语句至少执行一次所必须的测试用例数目的上界。
  3. 导出测试用例：根据圈复杂度和程序结构设计测试用例。
  4. 准备测试用例：确保基本路径集中的每一条路径的执行。

控制流图的符号

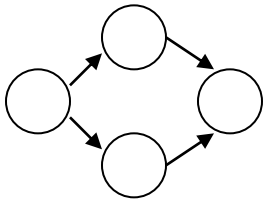
- 流图

  流图，一种简单的控制流表示方法，是对待测试程序过程处理的一种表示，是对程序流程图简化后得到的，它可更加突出表示程序控制流的结构。

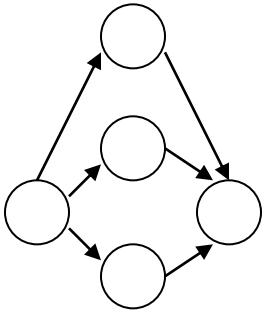
  流图使用下面的符号描述逻辑控制流，每一种结构化构成元素有一个相应的流图符号。



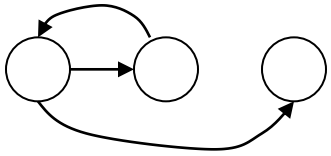
顺序结构



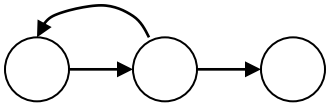
if 结构



Case 结构



while 结构



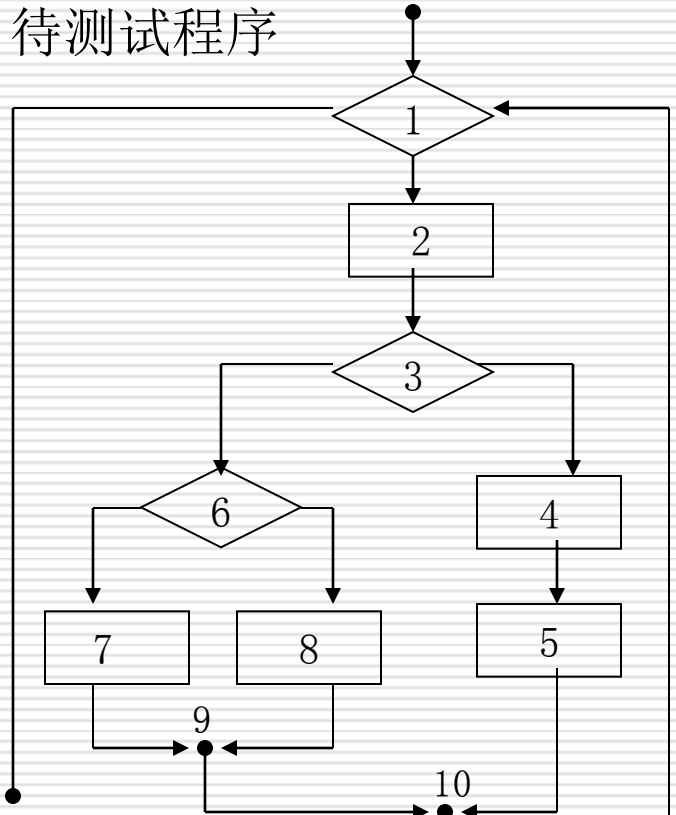
until 结构

控制流图

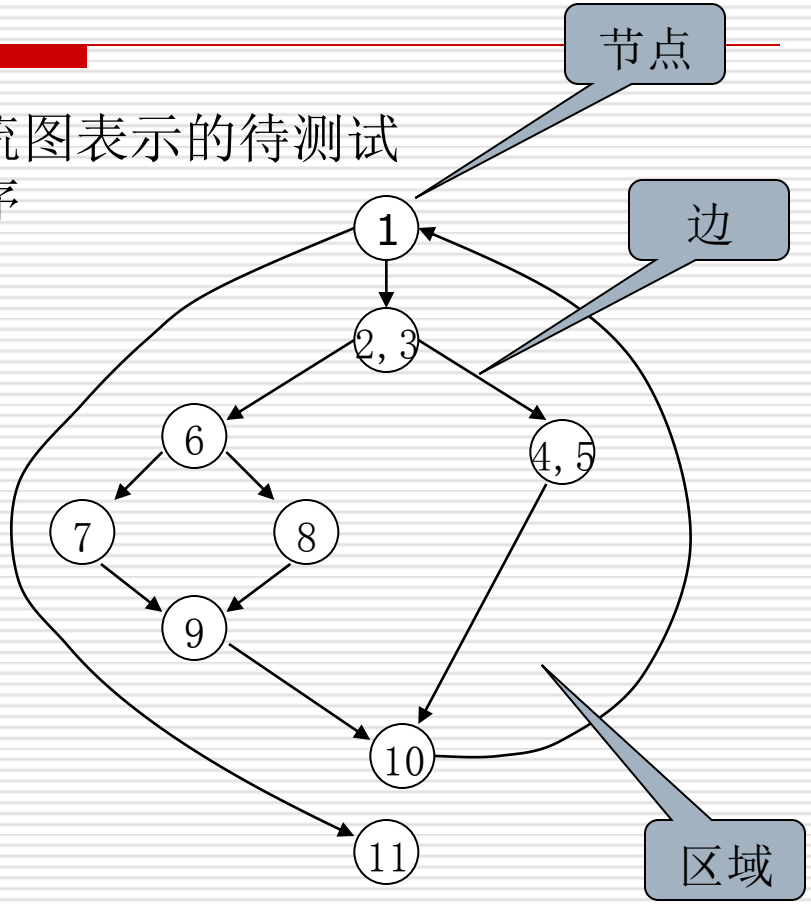
- 流图只有二种图形符号
  - 图中的每一个圆称为流图的结点，代表一条或多条语句，一个处理框序列和一个条件判定框（假设不包含复合条件）。
  - 流图中的箭头，称为边或连接，代表控制流。
- 任何过程设计都要被翻译成控制流图。
- 在将程序流程图简化成控制流图时，应注意：
  - 包含条件或多分支的节点被称为判定节点（也叫谓词节点），在选择或多分支结构中，分支的汇聚处应有一个汇聚结点。
  - 边和结点圈定的区域叫做区域，当对区域计数时，图形外的区域也应记为一个区域。

控制流图

待测试程序



用流图表示的待测试程序



区域：由边和解点封闭起来的区域

计算区域：不要忘记区域外的部分

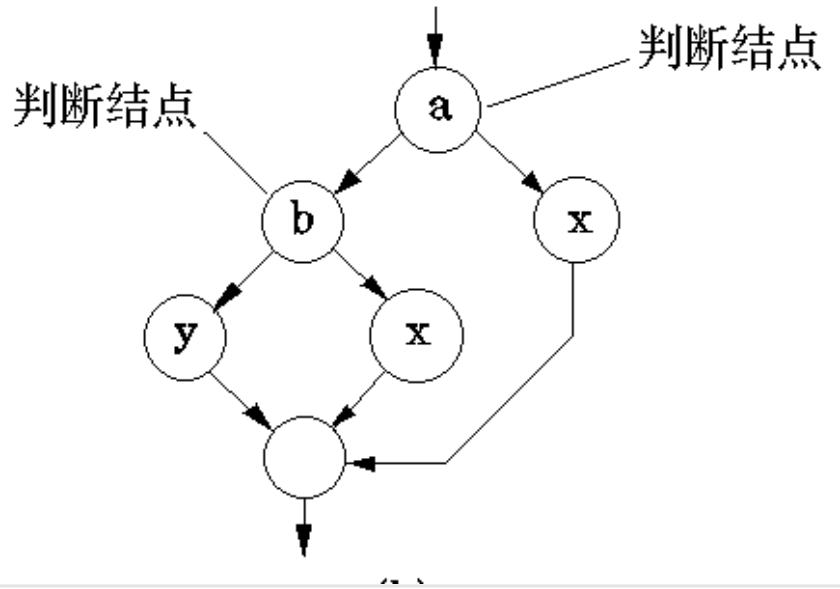
控制流图

- 如果判断中的条件表达式是由一个或多个逻辑运算符 (OR，AND, NAND，NOR) 连接的复合条件表达式，则需要改为一系列只有单条件的嵌套的判断。

  例如：

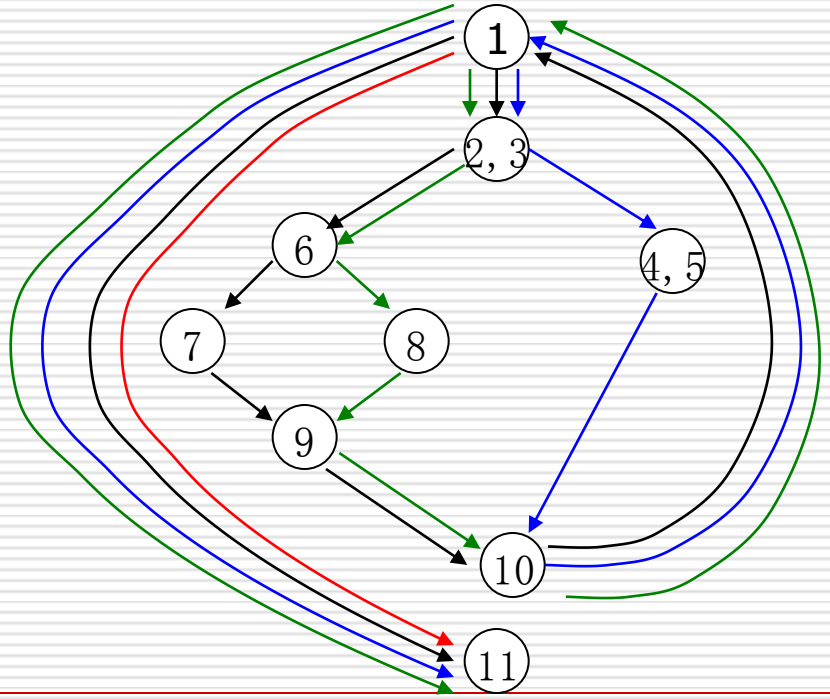
  ```
  1 if a or b
  2    x
  3 else
  4    y
  ```

  对应的逻辑为：



独立路径

独立路径：至少沿一条新的边移动的路径



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

对以上路径的遍历，就是至少一次地执行了程序中的所有语句。

基本路径测试

- 第一步：画出控制流图

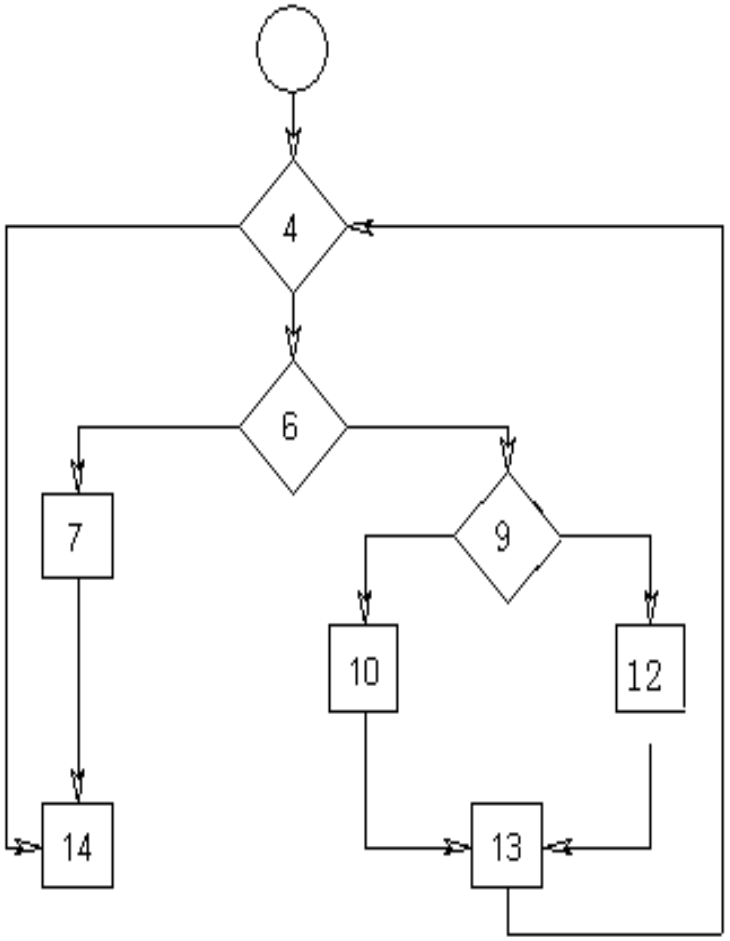
  将流程图映射到一个相应的流图(假设流程图的菱形决定框中不包含复合条件)：

  - 一个流程图的处理方框序列和一个菱形决测框可被映射为一个流图结点；
  - 流图中的箭头（边）类似于流程图中的箭头。一条边必须终止于一个结点，即使该结点并不代表任何语句(例如：if-else-then结构)。
  - 由边和结点限定的范围称为区域。计算区域时应包括图外部的范围。

基本路径测试

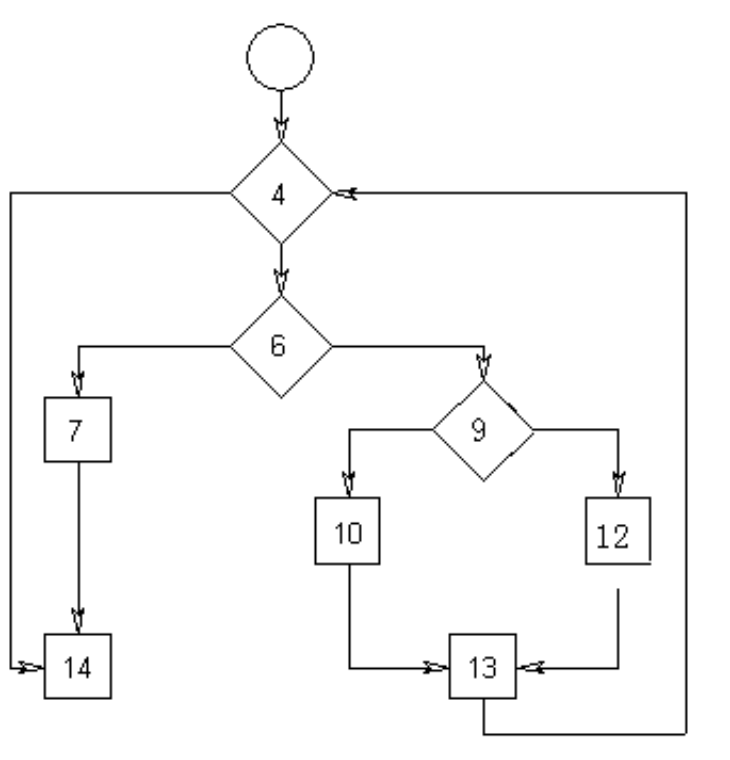
例4：有下面的C函数，用基本路径测试法进行测试

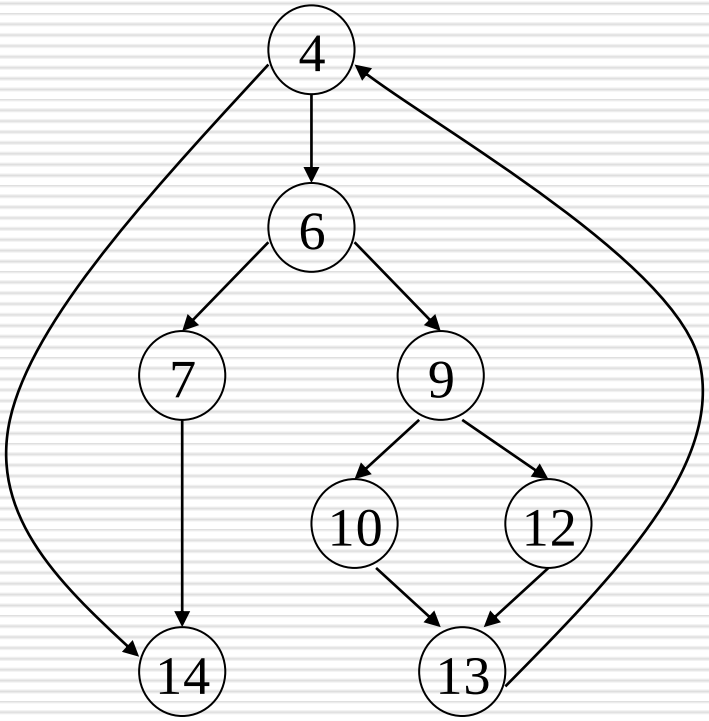
```
    void  Sort(int iRecordNum,int iType)
1.  {
2.     int x=0;
3.     int y=0;
4.     while (iRecordNum-- > 0)
5.     {
6.          if(0= =iType)
7.              { x=y+2; break;}
8.          else
9.              if (1= =iType)
10.                     x=y+10;
11.             else
12.                     x=y+20;
13.    }
14. }
```



基本路径测试

- 画出对应的控制流图





基本路径测试 - 计算圈复杂度

- 第二步：计算圈复杂度

  圈复杂度是一种为程序逻辑复杂性提供定量测度的软件度量，将该度量用于计算程序的基本的独立路径数目，为确保所有语句至少执行一次的测试数量的上界。

  有以下三种方法计算圈复杂度：

  1)流图中区域的数量对应于环型的复杂性；

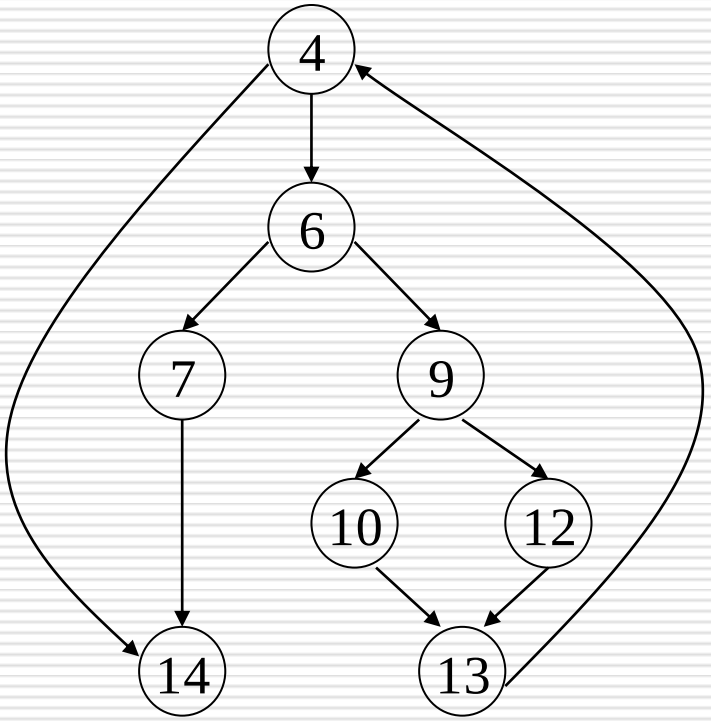
  2)给定流图G的圈复杂度V(G)，定义为V(G)=E-N+2，E是流图中边的数量，N是流图中结点的数量；

  3)给定流图G的圈复杂度V(G)，定义为V(G)=P+1，P是流图G中判定结点的数量。（包含条件的节点被称为判定节点，也叫谓词节点）

基本路径测试 - 计算圈复杂度

对应图中的圈复杂度，计算如下：

- ✓ 流图中有四个区域；
- ✓ V(G)=10条边-8结点+2=4;
- ✓ V(G)=3个判定结点+1=4。



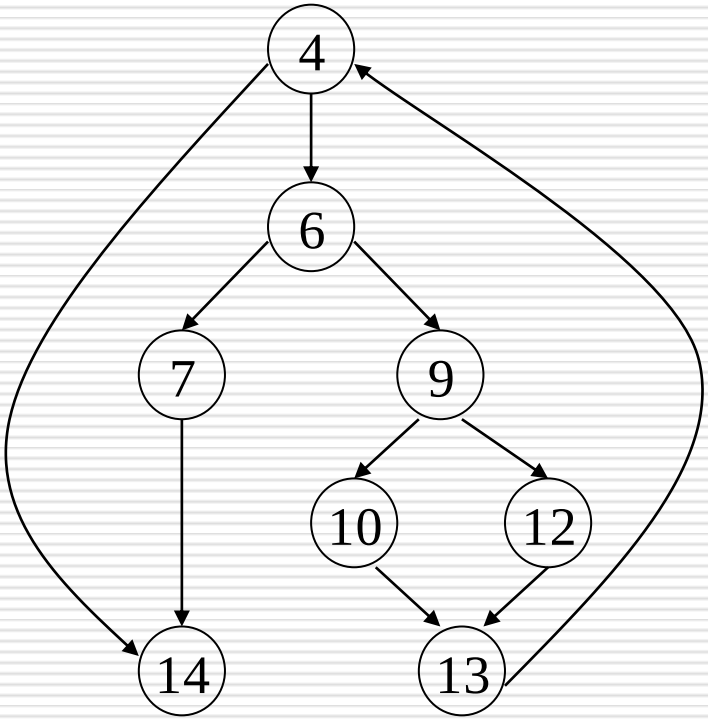
基本路径测试 - 导出测试用例

- 第三步：导出测试用例

根据上面的计算方法，可得出四个独立的路径。

- ✓ 路径1：4-14
- ✓ 路径2：4-6-7-14
- ✓ 路径3：4-6-9-10-13-4-14
- ✓ 路径4：4-6-9-12-13-4-14

根据上面的独立路径，去设计输入数据，使程序分别执行到上面四条路径。



基本路径测试 - 准备测试用例

- 第四步：准备测试用例

  为确保基本路径集中的每一条路径的执行，根据判断结点给出的条件，选择适当的数据以保证某一条路径可以被测试到。

  满足上面例子基本路径集的测试用例是：

基本路径测试 - 准备测试用例

路径1：4-14
输入数据：
　　iRecordNum=0，或
　　取iRecordNum<0的某一个值
预期结果：x=0

路径2：4-6-7-14
输入数据：
　　iRecordNum=1,iType=0
预期结果：x＝2

```
void  Sort(int iRecordNum,int iType)
1.   {
2.     int x=0;
3.     int y=0;
4.     while (iRecordNum-- > 0)
5.     {
6.         if(0= =iType)
7.             {x=y+2; break;}
8.         else
9.           if(1= =iType)
10.                x=y+10;
11.          else
12.                x=y+20;
13.    }
14.  }
```

基本路径测试 - 准备测试用例

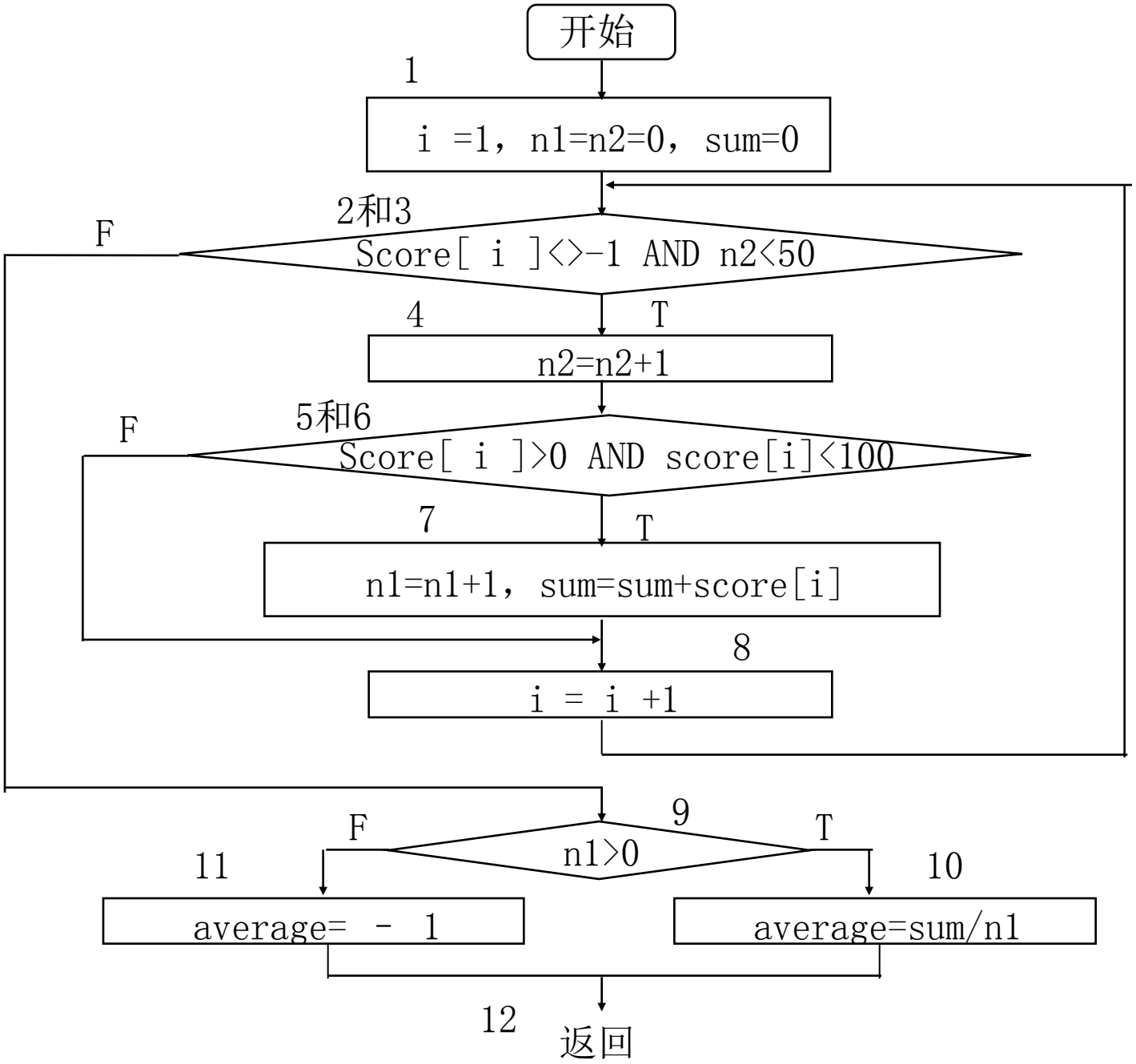
路径3：4-6-9-10-13-4-14
输入数据：
iRecordNum=1,iType=1
预期结果：x＝10

路径4：4-6-9-12-13-4-14
输入数据：
iRecordNum=1,iType=2
预期结果：x＝20

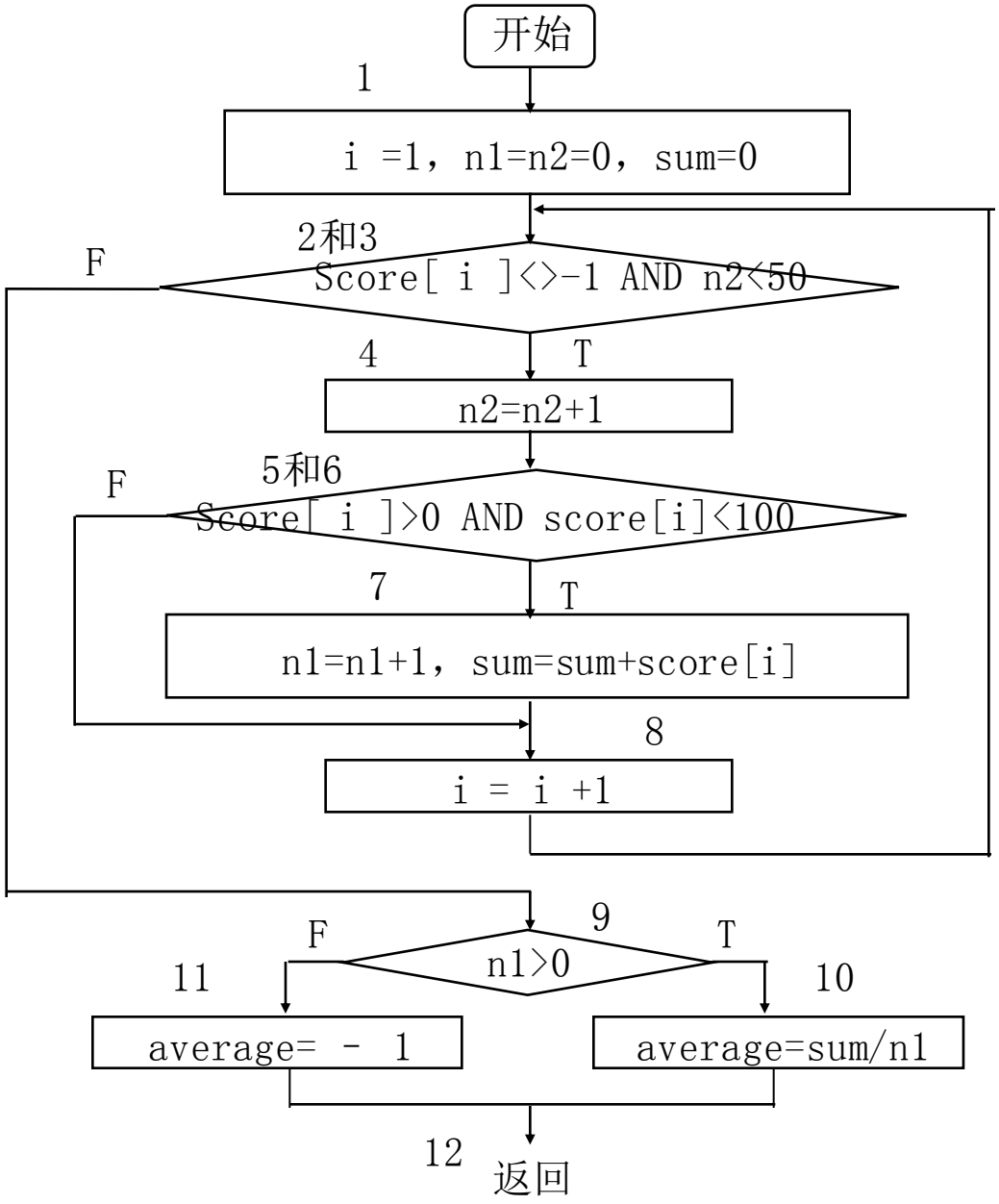
```
void  Sort(int iRecordNum,int iType)
1.    {
2.      int x=0;
3.      int y=0;
4.      while (iRecordNum-- > 0)
5.      {
6.          if(0= =iType)
7.              {x=y+2; break;}
8.          else
9.            if(1= =iType)
10.                 x=y+10;
11.           else
12.                 x=y+20;
13.     }
14.   }
```

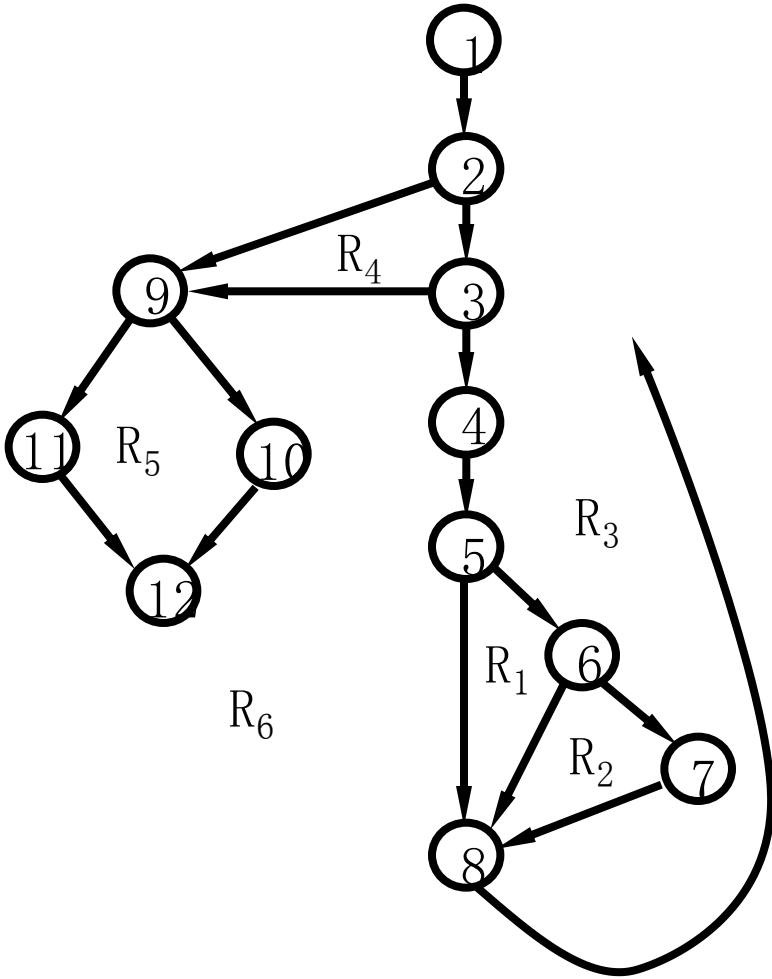
基本路径测试再举例

例6：下例程序流程图描述了最多输入50个值（以－1作为输入结束标志），计算其中有效的学生分数（0～100）的个数、总分数和平均值。


开始
1
i =1，n1=n2=0，sum=0
2和3
F
Score[i]<>-1 AND n2<50
4
T
n2=n2+1
5和6
F
Score[i]>0 AND score[i]<100
7
T
n1=n1+1，sum=sum+score[i]
8
i = i +1
9
F
n1>0
T
11
average= - 1
10
average=sum/n1
12
返回

步骤1：导出过程的流图。


开始
1
i =1，n1=n2=0，sum=0
2和3
F
Score[i]<>-1 AND n2<50
4
T
n2=n2+1
5和6
F
Score[i]>0 AND score[i]<100
7
T
n1=n1+1，sum=sum+score[i]
8
i = i +1
9
F
n1>0
T
11
10
average= - 1
average=sum/n1
12
返回


1
2
3
4
5
6
7
8
9
10
11
12
R1
R2
R3
R4
R5
R6

基本路径测试再举例

步骤2:确定环形复杂性度量V(G)：

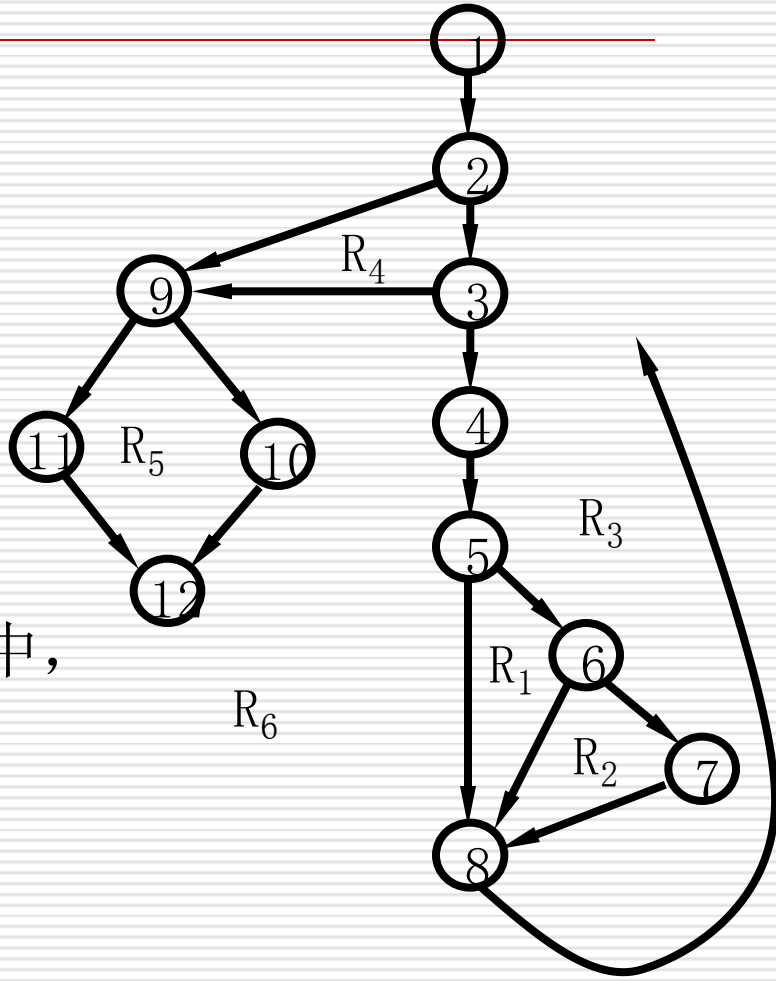
1）V(G)= 6 （个区域）

2）V(G)=E－N+2=16－12+2=6

其中E为流图中的边数，N为结点数；

3）V(G)=P+1=5+1=6

其中P为谓词结点的个数。在流图中，结点2、3、5、6、9是谓词结点。



基本路径测试再举例

步骤3：确定基本路径集合（即独立路径集合）。于是可确定6条独立的路径：

路径1：1-2-9-10-12

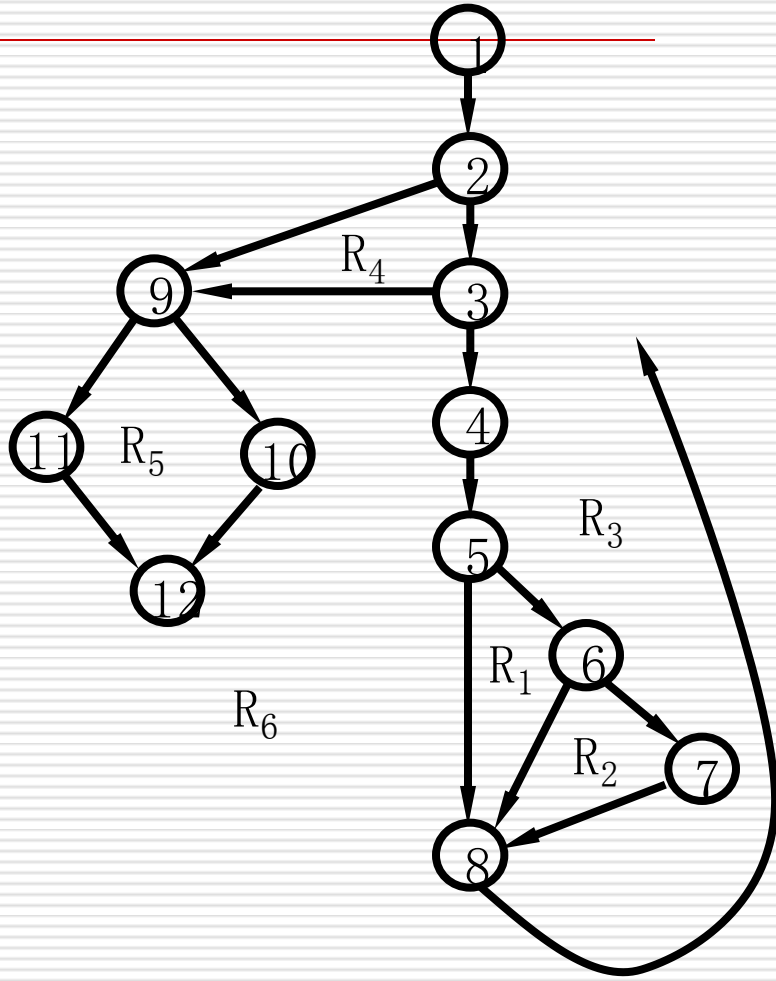
路径2：1-2-9-11-12

路径3：1-2-3-9-10-12

路径4：1-2-3-4-5-8-2…

路径5：1-2-3-4-5-6-8-2…

路径6：1-2-3-4-5-6-7-8-2…



步骤4：为每一条独立路径各设计一组测试用例，以便强迫程序沿着该路径至少执行一次。

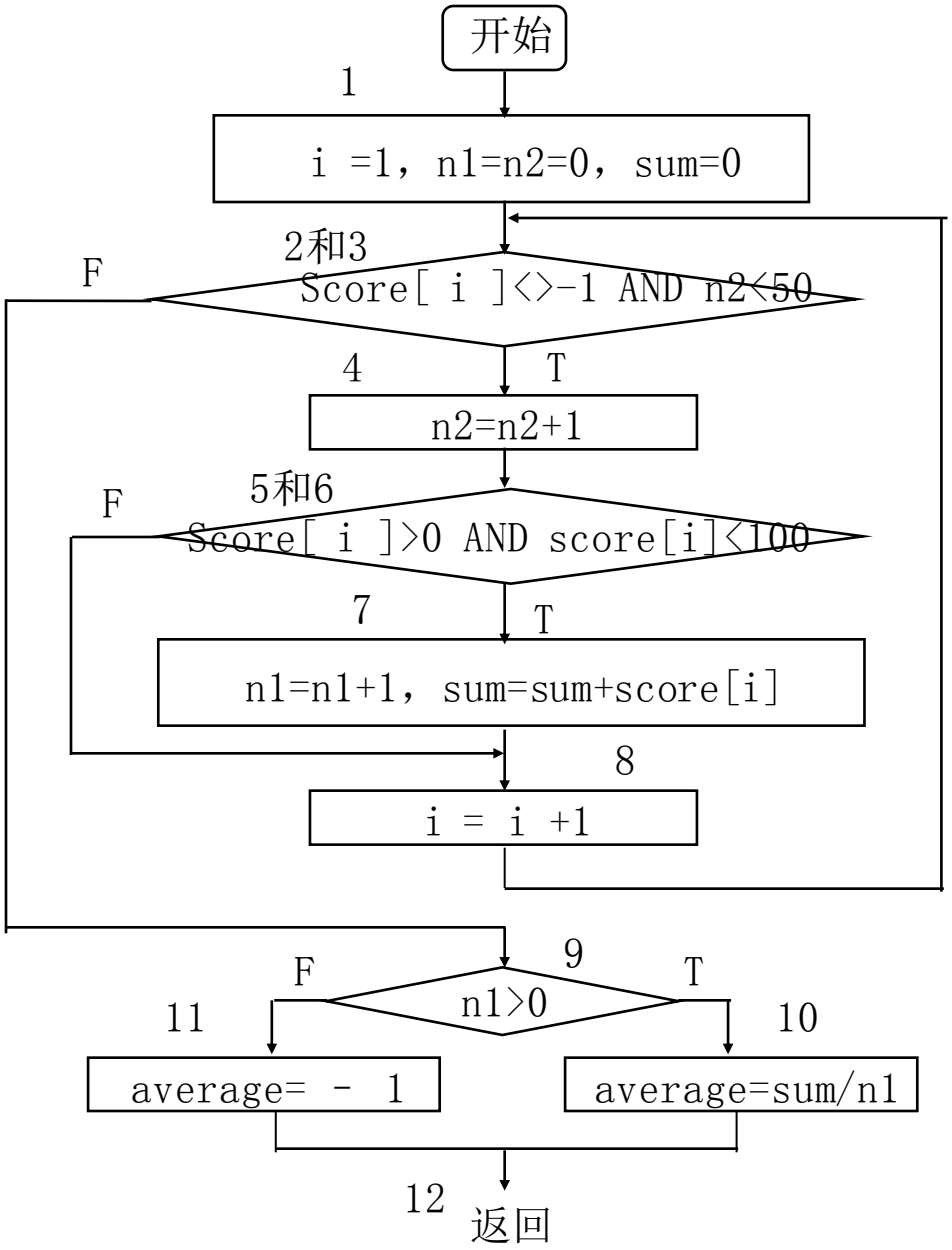
1）路径1(1-2-9-10-12)的测试用例：

score[k]=有效分数值，当k ＜ i ；

score[i]=－1, 2≤i≤50;

期望结果：

根据输入的有效分数算出正确的分数个数n1、总分sum和平均分average。



基本路径测试再举例

2）路径2(1-2-9-11-12)的测试用例：

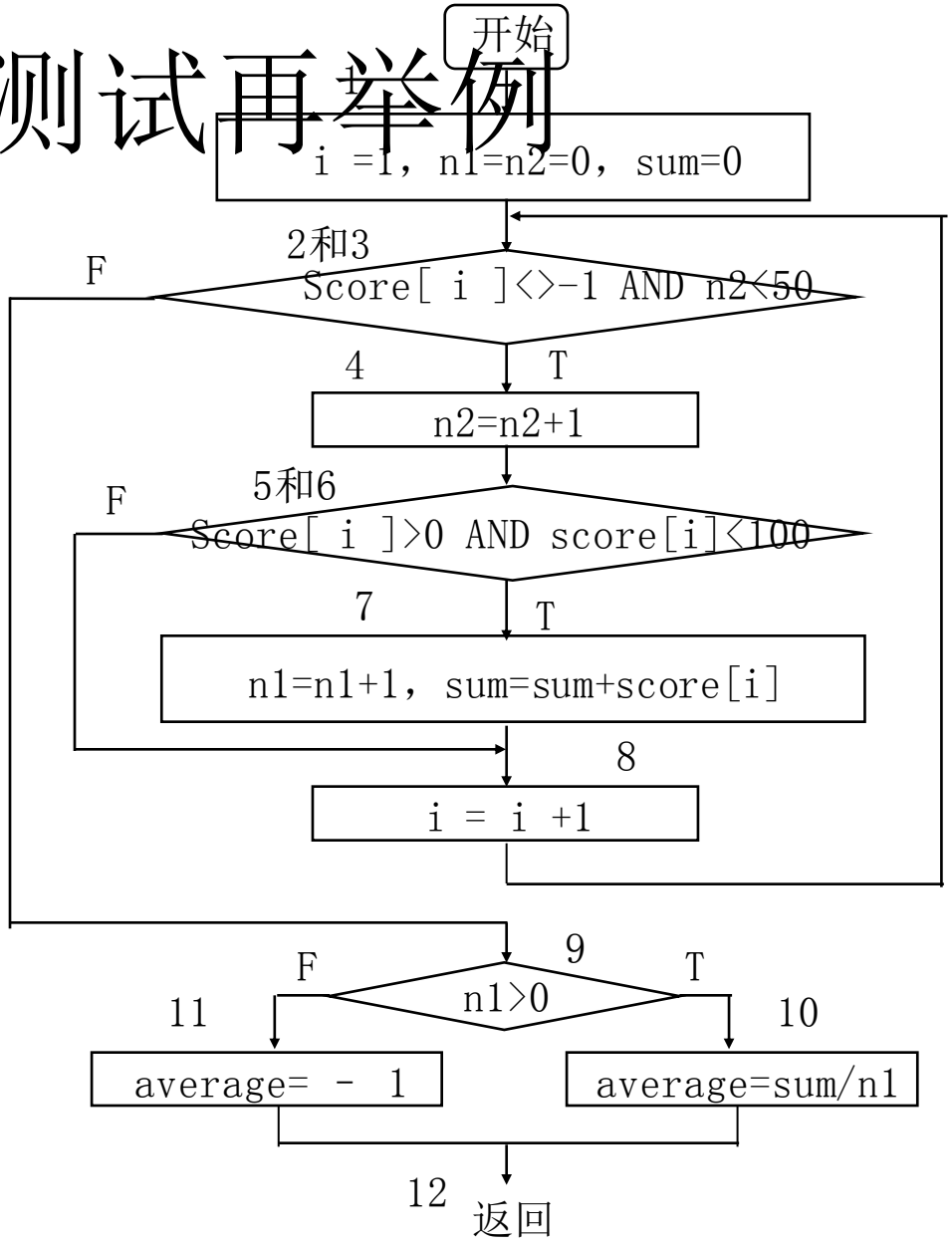
score[1]= - 1 ;

期望的结果：average = - 1 ，其他量保持初值。

3）路径3(1-2-3-9-10-12)的测试用例：

输入多于50个有效分数，即试图处理51个分数，要求前51个为有效分数；

期望结果：n1=50、且算出正确的总分和平均分。



基本路径测试再举例

4）路径4(1-2-3-4-5-8-2…)的测试用例：

score[i]=有效分数，当i<50;

score[k]<0，　　k< i　;

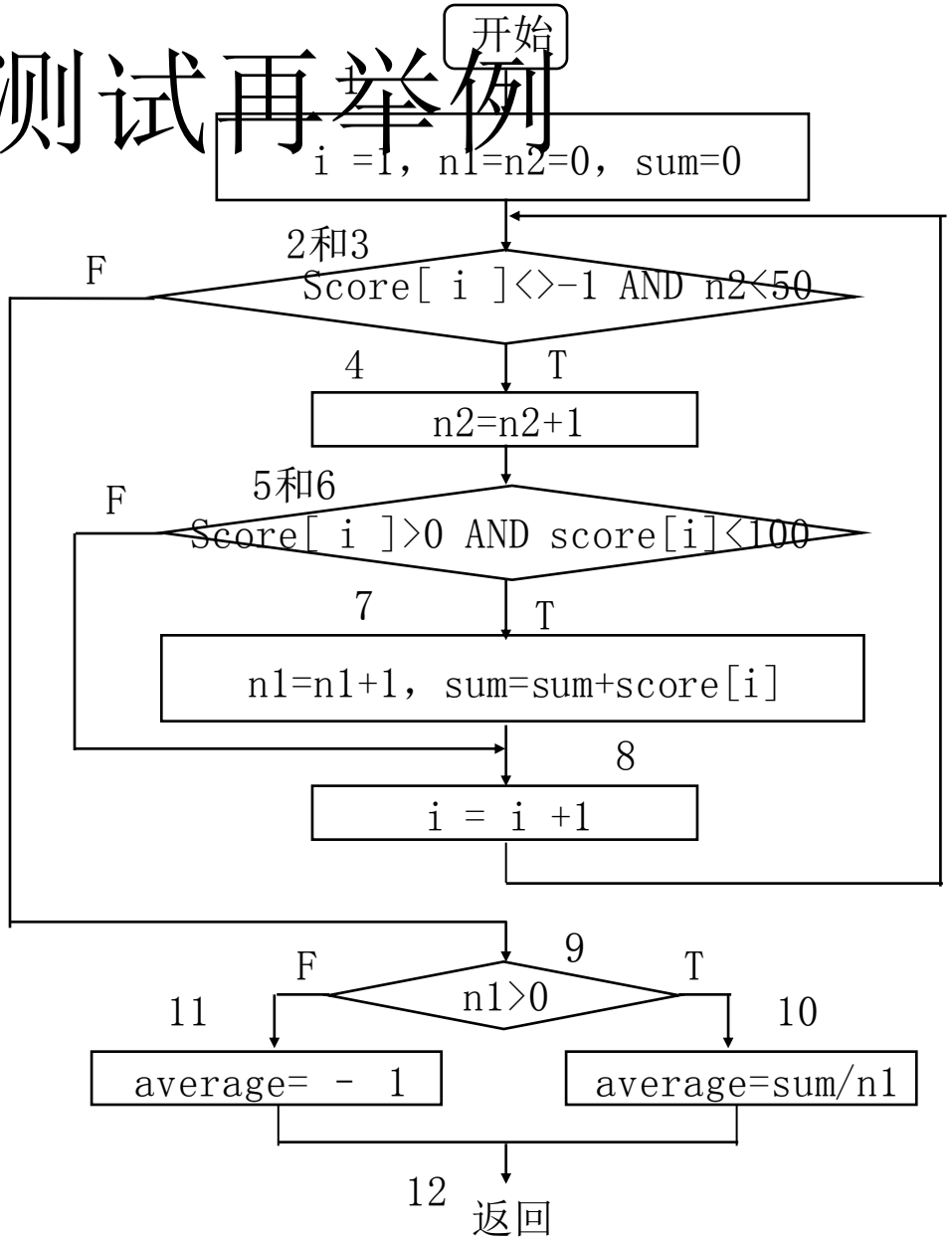
期望结果：根据输入的有效分数算出正确的分数个数n1、总分sum和平均分average。

5）路径5（1-2-3-4-5-6-8-2…）的测试用例：

score[i]=有效分数，　当i<50;

score[k]>100，　　k< i　;

期望结果：根据输入的有效分数算出正确的分数个数n1、总分sum和平均分average。

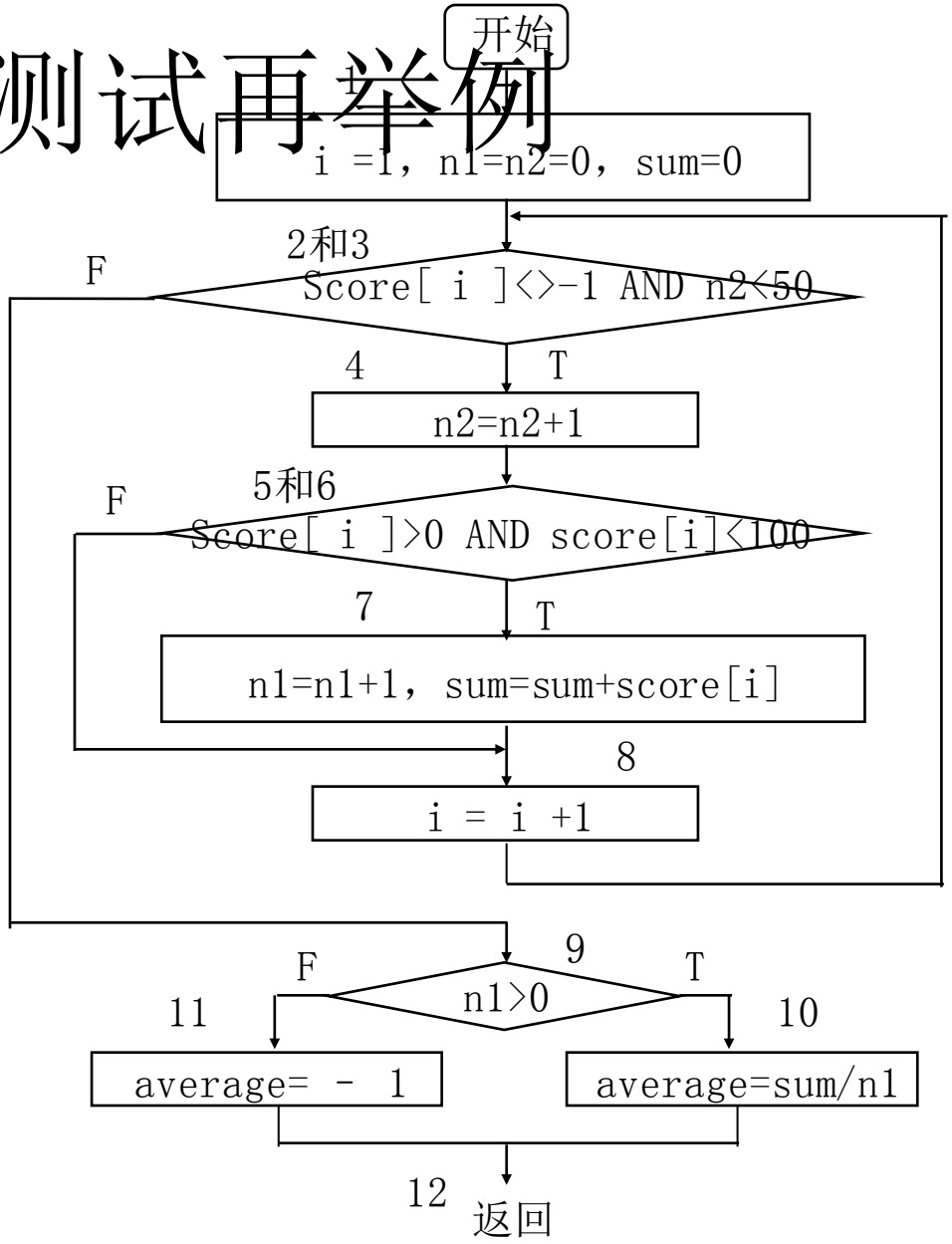


基本路径测试再举例

6） 路径6(1-2-3-4-5-6-7-8-2…)的测试用例：

score[i]=有效分数， 当i<50；

期望结果：根据输入的有效分数算出正确的分数个数n1、总分sum和平均分average。



基本路径测试

- 注意

1）一些独立的路径，往往不是完全孤立的，有时它是程序正常控制流的一部分，这时，这些路径的测试可以是另一条路径测试的一部分。

2）在有些情况下，一些执行路径是不可能被执行的，如：

If （!A）B++;

If （!A）D--;

这两个语句实际只包括了2条执行路径，即A为真或假时候对B和D的处理，真或假不可能都存在，而路径覆盖测试则认为是包含了真与假的4条执行路径。这样不仅降低了测试效率，而且大量的测试结果的累积，也为排错带来麻烦。

习题

1、使用基本路径测试方法，为以下程序段设计测试用例。

```
    void Do (int X,int A,int B)
    {
1       if ( (A>1)&&(B=0) )
2           X = X/A;
3       if ( (A=2)||(X>1) )
4           X = X+1;
5   }
```

2、在三角形问题中，要求输入三个边长：a，b，c。当三边不可能构成三角形时提示错误，可构成三角形时计算三角形的周长。若是等腰三角形打印“等腰三角形”，若是等边三角形，则打印“等边三角形”。画出相应的程序流程图，并采用基本路径测试方法为该程序设计测试用例。

控制结构测试的变种

前面所述的基本路径测试技术是控制结构测试技术之一。尽管基本路径测试简单高效，但是，其本身并不充分。

下面讨论控制结构测试的其他变种，这些测试覆盖并提高了白盒测试的质量。包括：

1. 条件测试
2. 数据流测试
3. 循环测试。

条件测试

条件测试方法注重测试程序中的条件。是检查程序模块中所包含逻辑条件的测试用例设计方法。

- 条件

程序中的条件分为简单条件和复合条件。

- 简单条件：

  是一个布尔变量 或 一个可能带有NOT(“！”)操作符的关系表达式。关系表达式的形式如：

  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

- 条件（续）
  - 复合条件

    由简单条件通过逻辑运算符（AND、OR、NOT）和括号连接而成，不含关系表达式的条件称为布尔表达式。

    所以条件的成分类型包括：布尔变量、关系操作符或算术表达式、逻辑运算符、括弧(括住简单或复杂条件)。

条件测试

- 条件的错误类型

  如条件不正确，则至少有一个条件成分不正确，这样，条件的错误类型如下：

  1. 布尔变量错误
  2. 关系操作符错误；
  3. 算术表达式错误。
  4. 逻辑运算符错误(遗漏，多余或不正确)
  5. 括弧错误；

条件测试

- 条件测试的目的

  条件测试是测试程序条件错误和程序的其他错误。

  如果程序的测试集能够有效地检测程序中的条件错误，则该测试集可能也会有效地检测程序中的其他错误。

  此外，如果测试策略对检测条件错误有效，则它也可能有效地检测程序错误。

条件测试

- 条件测试策略
  - 穷举测试

    （条件组合）

    有n个变量的布尔表达式需要2^n个可能的测试(n＞0)。这种策略可以发现布尔操作符、变量和括弧的错误，但是只有在n很小时实用。
  - 分支测试

    分支测试可能是最简单的条件测试策略，它是真假分支必须至少执行一次的路径策略，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。

条件测试 - 分支测试

- 域测试(Domain testing)

  域测试是对于大于、小于和等于值的测试策略。

  域测试要求从有理表达式中导出三个或四个测试用例，有理表达式的形式如：

  E1 ＜关系操作符＞ E2

  需要三个测试分别用于计算E1的值是大于、等于或小于E2的值。如果 ＜关系操作符＞ 错误，而E1和E2正确，则这三个测试能够发现关系算子的错误。

  为了发现E1和E2的错误，计算E1小于或大于E2的测试应使两个值间的差别尽可能小。

条件测试

- BRO(branch and relational)测试

  如在一个判定的复合条件表达式中每个布尔变量和关系运算符最多只出现一次，且没有公共变量，应用一种称之为BRO（分支与关系运算符）的测试法可以发现多个布尔运算符或关系运算符错，以及其他错误。

  BRO策略引入条件约束的概念：设有n个简单条件的复合条件C，其条件约束为D= (D1，D2，…，Dn)，其中Di(0＜i≤n)是条件C中第i个简单条件的输出约束。如果在C的执行过程中，其每个简单条件的输出都满足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。

  对于布尔变量或布尔表达式B，B的输出约束必须是真（t）或假（f）；

  对于关系表达式，其输出约束为符号>、=、<。

条件测试

例1：一种简单的情况，考虑条件

C1：B1&B2

其中B1和B2是布尔变量。

C1的输出约束为(D1，D2)：其中D1和D2是“T”或“F”，值(T，F)是C1可能的一个约束，覆盖此约束的测试（一次运行）将令B1为t，B2为f。

BRO测试策略要求约束集{(t,t),(f,t),(t,f),(f,f)}由C1的执行所覆盖。

如果布尔运算符有错，这四组测试用例的运行结果必有一组导致C1失败。

条件测试

例2，考虑

C2：B1＆(E3=E4)

其中B1是布尔表达式，而E3和E4是算术表达式。

C2的条件约束形式如(D1，D2)，其中D1是“T”或“F”，D2是＜，=或＞。

除了C2的第二个简单条件是关系表达式以外，C2和C1相同。所以可修改C1的约束集｛(T，T)，(F，T)，(T，F),(F，F)｝，得到C2的约束集：

｛(T，=)，(F，=)，(T，＜)，(T，＞)，(F，＜)，(F，＞)｝。

上述条件约束集的覆盖率将保证检测C2的布尔和关系算子的错误。

条件测试

例3，考虑

C3：(E1＞E2)＆(E3=E4)

其中E1、E2、E3和E4是算术表达式。

C3的条件约束形式如(D1，D2)，其中D1和D2是＜、=或＞。

除了C3的第一个简单条件是关系表达式以外，C3和C2相同，所以可以修改C2的约束集得到C3的约束集，结果为

{(＞，=)，(=，=)，(＜，=)，(＞，＜)，(＞，＞)，(＞，＜)，(=，＜)，(＜，＜)，(＞，＞)，(=，＞)，(＜，＞)}

去掉重复，结果为：

{(＞，=)，(=，=)，(＜，=)，(＞，＜)，(＞，＞)，(=，＜)，(＜，＜)，(=，＞)，(＜，＞)}

上述条件约束集能够保证检测C3的关系操作符的错误。

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来选择程序的测试路径，以发现数据处理异常。如没有初始化、定义变量；变量被定义但没有使用；变量在使用前被定义多次；变量在使用时失效等。

这些应通过测试用例来覆盖。

数据流测试方法不再详细介绍，下面介绍一类似的数据流分析方法。

- 数据流分析

  此方法在程序代码经过的路径上检查数据的用法，以期发现异常（不一定会导致软件失效）。

数据流分析

- 三种变量状态定义
  - 已定义的（d）：变量已赋值；
  - 引用的（r）：访问变量；
  - 没有定义的（u）：变量没有定义具体的值。
- 数据流异常的三种情况
  - ur异常：程序路径（r）上访问了没有定义（u）的变量；
  - du异常：变量已赋值（d），但此变量已无效或未定义（u），同时未被引用；
  - Dd异常：变量接受了第二个值（d），同时第一个值没被使用。

数据流分析

- 例，下面函数的功能是：如x<y,则交换x，y。

```
void exchange(int &x,int &y)
{   int t;
    if (x<y)
    {       y=t;    // t的ur异常
            y=x;    // y的dd异常
            t=x;    // t的du异常
    }
}
```

数据流分析

- 变量t的ur异常

  t未赋初值（初始化）就被使用。

- 变量y的dd异常

  y连续使用了二次。第一次可忽略。

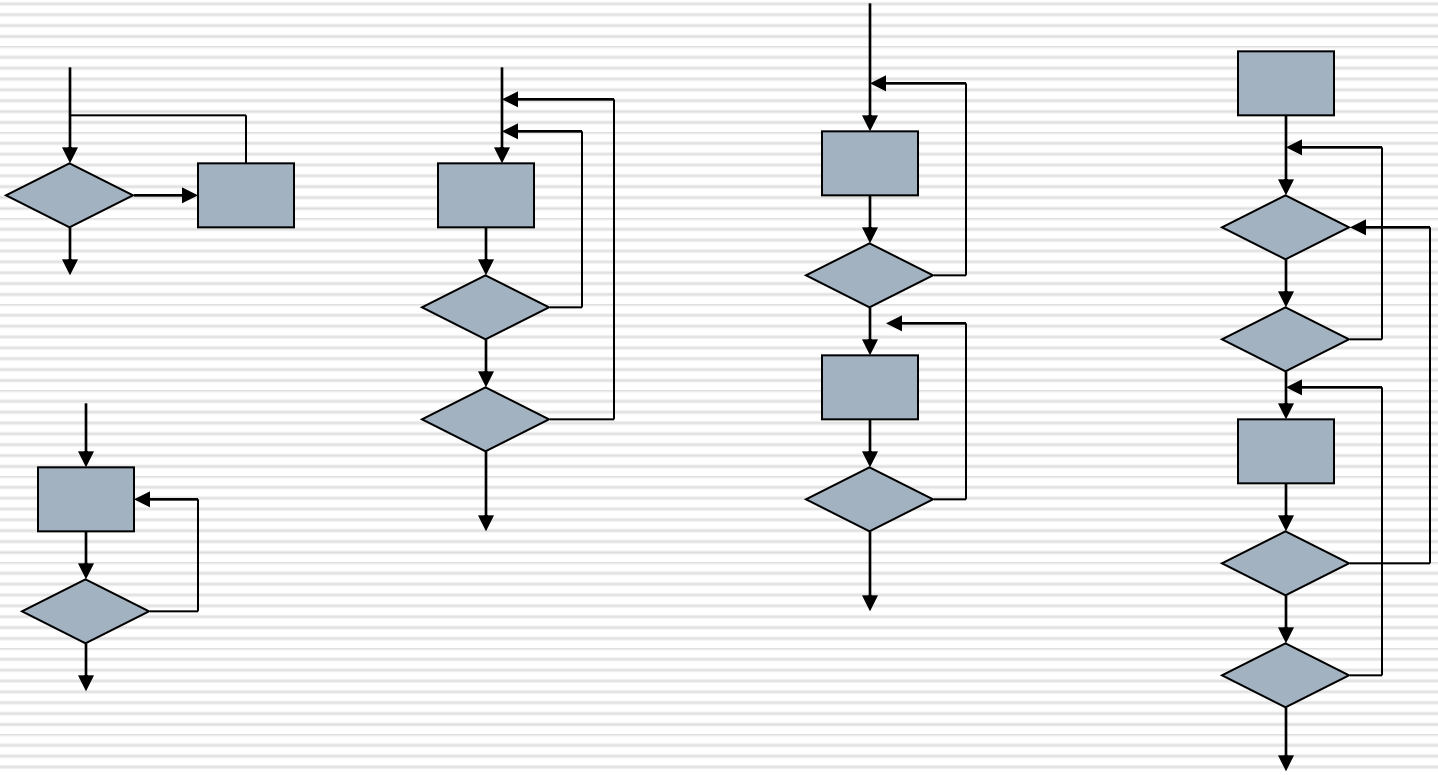
- 变量t的du异常

  最后一句，t被赋了任何地方都不能用的值。

循环测试

- 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：简单循环，串接(连锁)循环，嵌套循环和不规则循环。

循环测试



简单循环　　嵌套循环　　串接循环　　无结构循环

循环测试 - 简单循环

- 简单循环

对于简单循环，测试应包括以下几种，其中的n表示循环允许的最大次数。

1） 零次循环：从循环入口直接跳到循环出口。

2） 一次循环：查找循环初始值方面的错误。

3） 二次循环：检查在多次循环时才能暴露的错误。

4） m次循环：此时的m＜n，也是检查在多次循环时才能暴露的错误。

5） n(最大)次数循环、n+1(比最大次数多一)次的循环、n-1(比最大次数少一)次的循环。

循环测试 - 嵌套循环

- 嵌套循环

对于嵌套循环，不能将简单循环的测试方法扩大到嵌套循环，因为可能的测试数目将随嵌套层次的增加呈几何倍数增长。下面是一种有助于减少测试数目的测试方法。

- 从最内层循环开始，设置所有其他层的循环为最小值；
- 对最内层循环做简单循环的全部测试。

  测试时保持所有外层循环的循环变量为最小值。另外，对越界值和非法值做类似的测试。
- 逐步外推，对其外面一层循环进行测试。测试时保持所有外层循环的循环变量取最小值，所有其它嵌套内层循环的循环变量取“典型”值。

循环测试 - 嵌套循环

- 嵌套循环（续）
  - 反复进行，直到所有各层循环测试完毕。
  - 对全部各层循环同时取最小循环次数，或者同时取最大循环次数。对于后一种测试，由于测试量太大，需人为指定最大循环次数。
- 串接循环

  对于串接循环，要区别两种情况：
  - 如各个循环互相独立，则串接循环可以用与简单循环相同的方法进行测试。
  - 如果有两个循环处于串接状态，而前一个循环的循环变量的值是后一个循环的初值。则这几个循环不是互相独立的，则需要使用测试嵌套循环的办法来处理。

循环测试 - 非结构循环

- 非结构循环

  对于非结构循环，不能测试，应重新设计循环结构，使之成为其它循环方式，然后再进行测试。

其他程序结构的测试方法

- 域测试
  - 针对域错误，对输入空间进行分析，选择适当的测试点，检验输入空间中的每个输入是否产生正确的结果。
  - 假设限制过多，难以运用到实际中。
- 符号测试
  - 另辟途径解决测试用例选择问题。
  - 基于代数运算执行测试，是测试和验证的折衷方法。
- 程序插装

  借助往被测程序中插入操作来实现测试目的的方法。

其他程序结构的测试方法

- 程序变异
  - 是一种错误驱动测试，针对某类特定程序错误实现测试。
  - 程序强变异
  - 程序弱变异

（可参考郑人杰：计算机软件测试技术
清华大学出版社）

面向对象的白盒测试

对OO软件的类测试相当于传统软件的单元测试。和传统软件的单元测试不同，他往往关注模块的算法细节和模块接口间流动的数据，OO软件的类测试是由封装在类中的操作和类的状态行为所驱动的。OO软件测试的特点：

- 因为属性和操作是被封装的，对类之外操作的测试通常是徒劳的。封装使对对象的状态快照难于获得。
- 继承也给测试带来了难度，即使是彻底复用的，对每个新的使用语境也需要重新测试。
- 多重继承更增加了需要测试的语境的数量，使测试进一步复杂化。如果从超类导出的测试用例被用于相同的问题域，有可能对超类导出的测试用例集可以用于子类的测试，然而，如果子类被用于完全不同的语境，则超类的测试用例将没有多大用途，必须设计新的测试用例集。

类测试方式

- 类测试一般有两种主要的方式：

  功能性测试和结构性测试，即对应于传统结构化软件的黑盒测试和白盒测试。

  功能性测试以类的规格说明为基础，它主要检查类是否符合其规格说明的要求。例如，对于Stack类，即检查它的操作是否满足LIFO规则；

  结构性测试则从程序出发，它需要考虑其中的代码是否正确，同样是Stack类，就要检查其中代码是否动作正确且至少执行过一次。

结构性测试方法（白盒测试）

结构性测试对类中的方法进行测试，它把类作为一个单元来进行测试。测试分为两层：

第一层考虑类中各独立方法的代码；

第二层考虑方法之间的相互作用。

- 方法的单独测试

  结构性测试的第一层是考虑各独立的方法，这可与过程的测试采用同样的方法。

  两者之间最大的差别在于：方法改变了它所在实例的状态，这就要取得隐藏的状态信息来估算测试的结果，传给其它对象的消息被忽略，而以桩来代替，并根据所传的消息返回相应的值，测试数据要求能完全覆盖类中代码，可以用传统的测试技术来获取。

结构性测试方法（白盒测试）

- 方法的综合测试

  第二层要考虑一个方法调用本对象类中的其它方法和从一个类向其它类发送信息的情况。

  单独测试一个方法时，只考虑其本身执行的情况。而没有考虑动作的顺序问题，测试用例中加入了激发这些调用的信息，以检查它们是否正确运行了。

  对于同一类中方法之间的调用，一般只需要极少甚至不用附加数据，因为方法都是对类进行存取，故这一类测试的准则是要求遍历类的所有主要状态。

白盒测试工具：

内存资源泄漏检查：Numega中的bouncechecker,Rational的Purify等；

代码覆盖率检查：Numega中的truecoverage,Rational的Purecoverage，Telelogic公司的logiscope,Macabe公司的Macabe等；

开源覆盖率测试软件gCov等。

总结

控制结构测试

- 路径测试
  - •利用流图表示控制逻辑
  - •根据流图标识独立路径
  - •确定覆盖测试路径上界的计算（环复杂度计算）
  - •用基本路径法导出测试案例的步骤
- 条件测试
  - 分支测试：真假分支必须至少执行一次的路径策略
  - 域测试：对于大于、小于和等于值的测试路径策略
- 数据流测试
  - 由变量的定义到变量的使用，构成DU链，覆盖每个DU链至少一次。用此方法为包含循环和嵌套语句的程序选择测试路径的策略
- 循环测试
  - 对于简单循环、嵌套循环、串接循环和无结构循环的路径选择策略