白盒测试方法

清华大学软件学院 刘强

测试覆盖标准

- 测试需求：测试需求是软件制品的一个特定元素，测试用例必须满足或覆盖这个特定元素。
- 覆盖标准：一个覆盖标准是一条规则，或者是将测试需求施加在一个测试集上的一组规则。
- 测试覆盖：给定一个覆盖标准 *C* 和相关的测试需求集合 *TR*，欲使一个测试集合 *T* 满足*C*，当且仅当对于测试需求集合 *TR* 中的每一条测试需求*tr*，在 *T* 中至少存在一个测试 *t* 可以满足 *tr*。
- 覆盖程度：给定一个测试需求集合 *TR* 和一个测试集合 *T*，覆盖程度就是 *T* 满足的测试需求数占 *TR* 总数的比例。

测试覆盖标准

软心糖豆：6 种口味和 4 种颜色

柠檬味（黄色）、开心果味（绿色）、梨子味（白色）
哈密瓜味（橙色）、橘子味（橙色）、杏味（黄色）

- 可以用什么覆盖标准来选择糖豆进行测试?
- 哪一种覆盖标准更好? 为什么?
- 应该考虑哪些因素来选择覆盖标准?

白盒测试技术

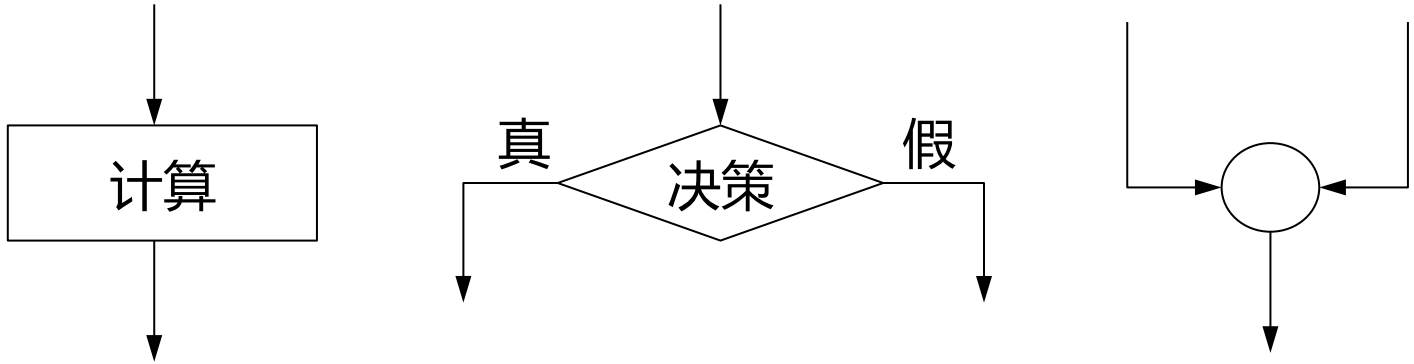
白盒测试是将测试对象看做一个透明的盒子，允许测试人员利用程序内部的逻辑结构及有关信息，设计或选择测试用例，对程序所有逻辑路径进行测试。

控制流图

控制流图（CFG， Control Flow Graph）是一个过程或程序的抽象表示。

控制流图的基本符号：

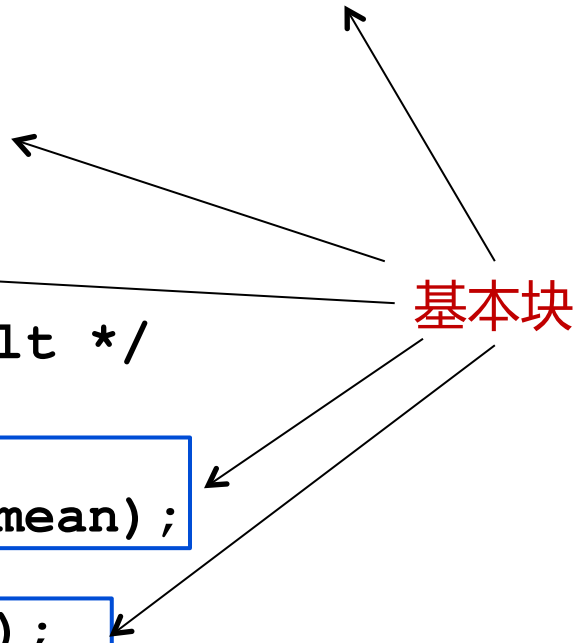
- 矩形代表了连续的顺序计算，也称基本块
- 节点是语句或语句的一部分，边表示语句的控制流

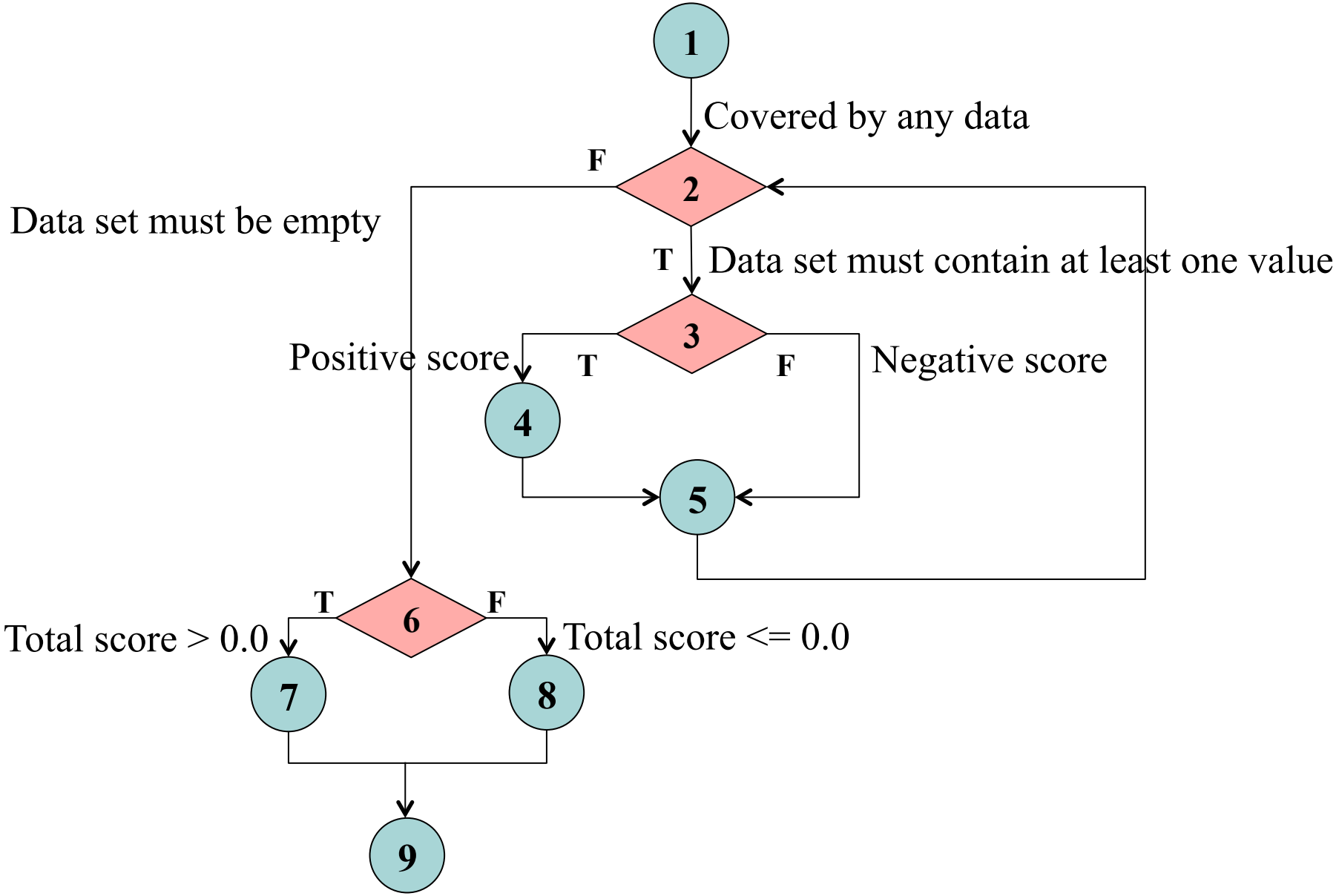


顺序计算　　判断节点　　合并节点

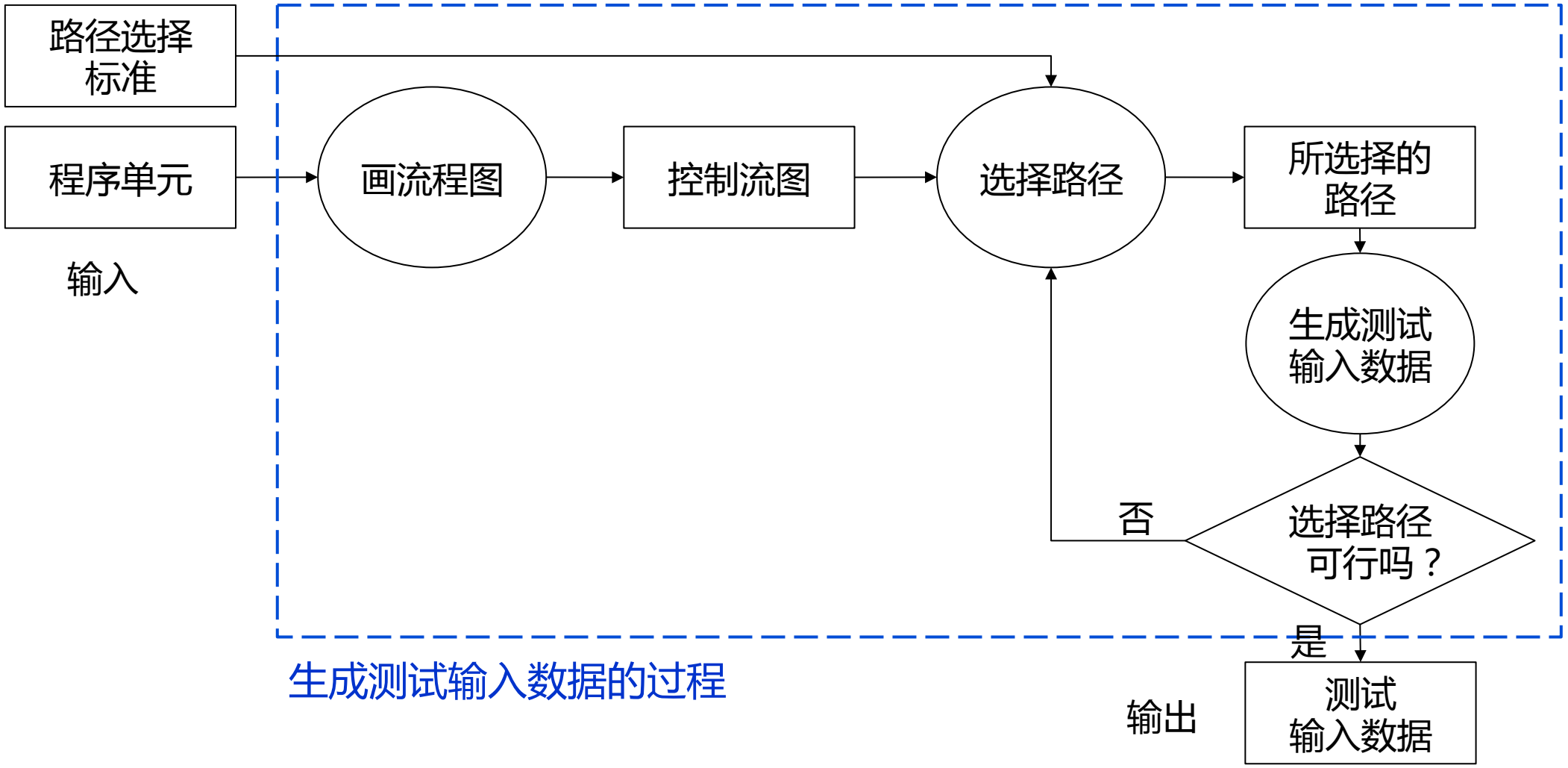
```
FindMean(float *mean, FILE *fp)
{
    float sum = 0.0, score = 0.0;
    int num = 0;

    fscanf(fp, "%f", &score); /* Read and parse into score */
    while (!EOF(fp)) {
        if (score > 0.0) {
            sum += score;
            num++;
        }
        fscanf(fp, "%f", &score);
    }
    /* Compute the mean and print the result */
    if (num > 0) {
        *mean = sum/num;
        printf("The mean score is %f \n", mean);
    } else
        printf("No scores found in file\n");
}
```

```
FindMean(float *mean, FILE *fp)
{
    float sum = 0.0, score = 0.0;
①   int num = 0;

    fscanf(fp, "%f", &score); /* Read and parse into score */
②   while (!EOF(fp)) {
③     if (score > 0.0) {
④       sum += score;
        num++;
      }
⑤     fscanf(fp, "%f", &score);
    }
    /* Compute the mean and print the result */
⑥   if (num > 0) {
⑦     *mean = sum/num;
      printf("The mean score is %f \n", mean);
    } else
⑧     printf("No scores found in file\n");
}
```




1
Covered by any data
F
2
Data set must be empty
T
Data set must contain at least one value
3
Positive score
T
F
Negative score
4
5
T
6
F
Total score > 0.0
Total score <= 0.0
7
8
9

基于控制流的测试

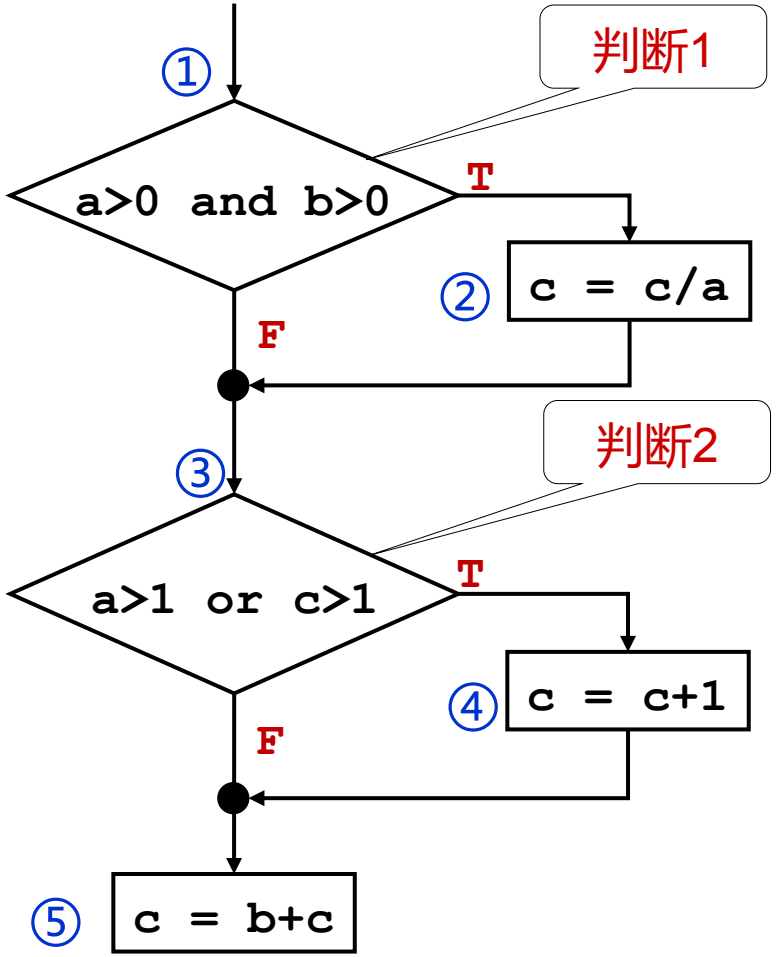

路径选择
标准
程序单元
输入
画流程图
控制流图
选择路径
所选择的
路径
生成测试
输入数据
选择路径
可行吗？
否
是
测试
输入数据
输出
生成测试输入数据的过程

代码覆盖标准

代码覆盖率描述的是代码被测试的比例和程度，通过代码覆盖率可以得知哪些代码没有被覆盖，从而进一步补足测试用例。

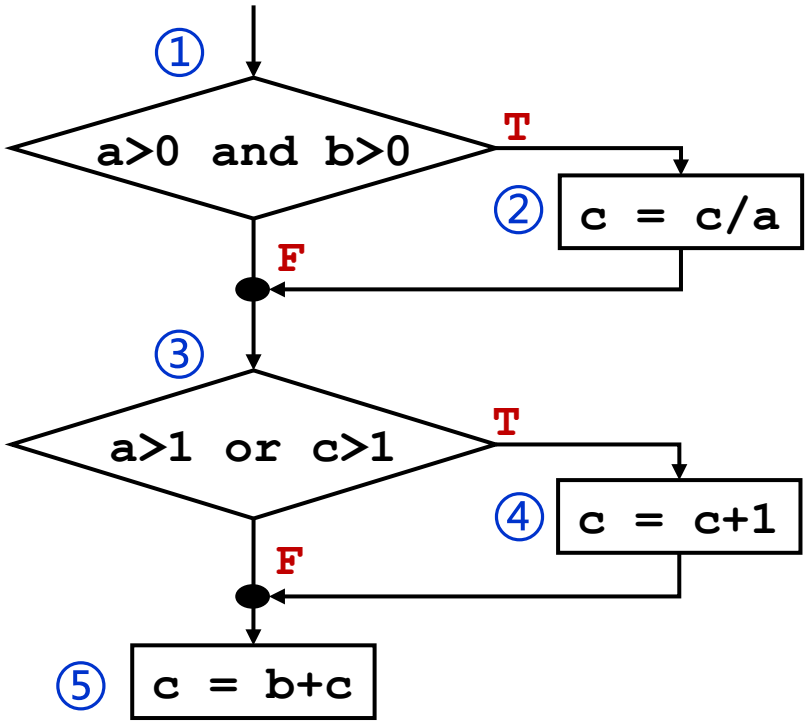
示例描述

```
double func1(int a, int b, double c)
{
    if (a>0 && b>0) {
        c = c/a;
    }

    if (a>1 || c>1) {
        c = c+1;
    }

    c = b+c;

    return c;
}
```



语句覆盖

程序中的每个可执行语句至少被执行一次。



输入：a=2，b=1，c=6

语句覆盖

程序中的每个可执行语句至少被执行一次。

测试用例：

输入：**a=2，b=1，c=6**

程序中三个可执行语句均被执行一次，满足语句覆盖标准。

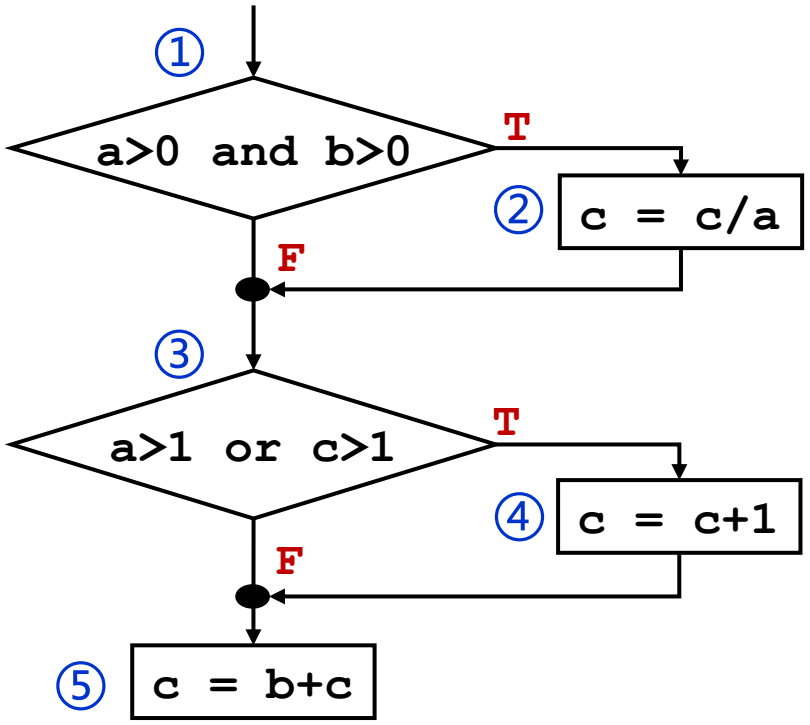
问题分析：

测试用例虽然覆盖可执行语句，但无法检查判断逻辑是否存在问题，例如第一个条件判断中“**&&**”被错误地写成“**||**”。

语句覆盖是最弱的逻辑覆盖准则。

判定覆盖（分支覆盖）

程序中每个判断的取真和取假分支至少经历一次，即判断真假值均被满足。



输入：a=2,b=1,c=6

输入：a=-2,b=-1,c=-3

判定覆盖（分支覆盖）

程序中每个判断的取真和取假分支至少经历一次，即判断真假值均被满足。

测试用例：

输入：`a=2,b=1,c=6`，覆盖判断1的 T分支和判断2的 T分支

输入：`a=-2,b=-1,c=-3`，覆盖判断1的 F分支和判断2的 F分支

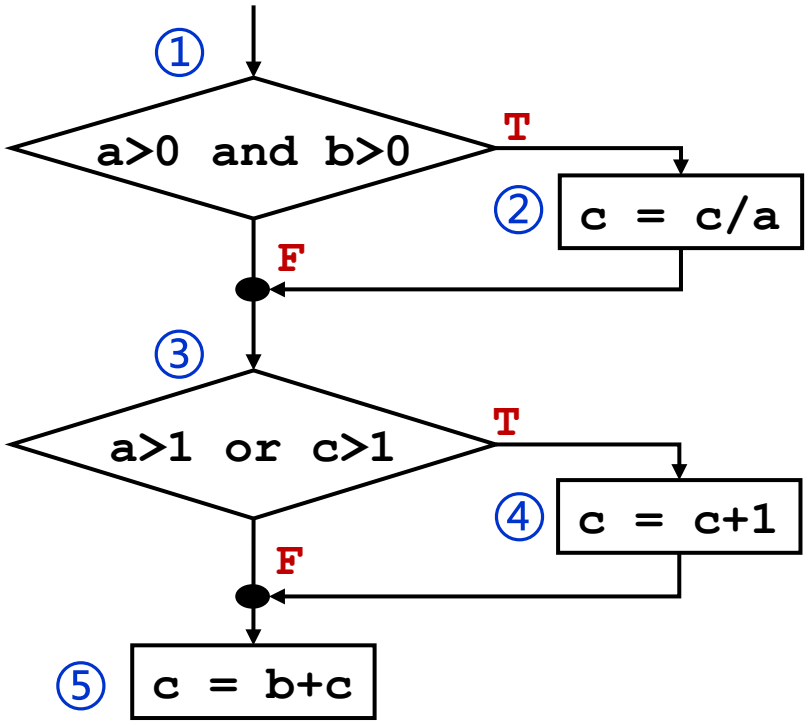
问题分析：

由于大部分判定语句是由多个逻辑条件组合而成，若仅判断其整个最终结果，而忽略每个条件的取值情况，必然会遗漏部分测试路径。

判定覆盖具有比语句覆盖更强的测试能力，但仍是弱的逻辑覆盖。

条件覆盖

每个判断中每个条件的可能取值至少满足一次。



输入： a=2,b=-1,c=-2

覆盖 a>0,b<=0,a>1,c<=1 条件

输入：a=-1,b=2,c=3

覆盖 a<=0,b>0,a<=1,c>1 条件

条件覆盖

每个判断中每个条件的可能取值至少满足一次。

测试用例：

输入： `a=2,b=-1,c=-2`，覆盖 `a>0,b<=0,a>1,c<=1` 条件

输入：`a=-1,b=2,c=3`，覆盖 `a<=0,b>0,a<=1,c>1` 条件

问题分析：

条件覆盖不一定包含判定覆盖，例如上面测试用例就没有覆盖判断1的T分支和判断2的 F分支。

条件覆盖只能保证每个条件至少有一次为真，而没有考虑整个判定结果。

判定条件覆盖

判断中所有条件的可能取值至少执行一次，且所有判断的可能结果至少执行一次。

测试用例：

输入：a=2,b=1,c=6，覆盖 a>0,b>0,a>1,c>1 且判断均为T

输入：a=-1,b=-2,c=-3，覆盖 a<=0,b<=0,a<=1,c<=1且判断均为F

问题分析：

判定条件覆盖能够同时满足判定、条件两种覆盖标准。

没有考虑条件的各种组合情况。

条件组合覆盖

判断中每个条件的所有可能取值组合至少执行一次，并且每个判断本身的结果也至少执行一次。

组合编号	覆盖条件取值	判定条件取值	判定-条件组合
1	a>0，b>0	判断1取T	a>0，b>0，判断1取T
2	a>0，b<=0	判断1取F	a>0，b<=0，判断1取F
3	a<=0，b>0	判断1取F	a<=0，b>0，判断1取F
4	a<=0，b<=0	判断1取F	a<=0，b<=0，判断1取F
5	a>1，c>1	判断2取T	a>1，c>1，判断2取T
6	a>1，c<=1	判断2取T	a>1，c<=1，判断2取T
7	a<=1，c>1	判断2取T	a<=1，c>1，判断2取T
8	a<=1，c<=1	判断2取F	a<=1，c<=1，判断2取F

条件组合覆盖

测试用例	覆盖条件	覆盖路径	覆盖组合
输入：a=2，b=1，c=6	a>0,b>0 a>1,c>1	1-2-4	1，5
输入：a=2，b=-1，c=-2	a>0,b<=0 a>1,c<=1	1-3-4	2，6
输入：a=-1，b=2，c=3	a<=0,b>0 a<=1,c>1	1-3-4	3，7
输入：a=-1，b=-2，c=-3	a<=0,b<=0 a<=1,c<=1	1-3-5	4，8

问题分析：

条件组合覆盖准则满足判定覆盖、条件覆盖和判定条件覆盖准则。

覆盖了所有组合，但覆盖路径有限，上面示例中1-2-5没覆盖。

路径覆盖

覆盖程序中的所有可能的执行路径。

测试用例	覆盖条件	覆盖路径	覆盖组合
输入：a=2，b=1，c=6	a>0,b>0 a>1,c>1	1-2-4	1，5
输入：a=1，b=1，c=-3	a>0,b>0 a<=1,c<=1	1-2-5	1，8
输入：a=-1，b=2，c=3	a<=0,b>0 a<=1,c>1	1-3-4	3，7
输入：a=-1，b=-2，c=-3	a<=0,b<=0 a<=1,c<=1	1-3-5	4，8

路径覆盖

问题分析：前面的测试用例完全覆盖所有路径，但没有覆盖所有条件组合。

下面结合条件组合和路径覆盖两种方法重新设计测试用例：

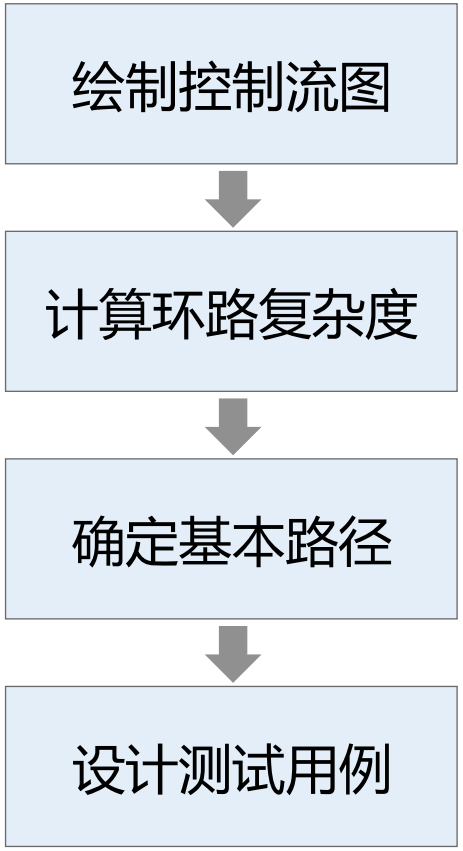
测试用例	覆盖条件	覆盖路径	覆盖组合
输入：a=2，b=1，c=6	a>0,b>0 a>1,c>1	1-2-4	1，5
输入：a=1，b=1，c=-3	a>0,b>0 a<=1,c<=1	1-2-5	1，8
输入：a=2，b=-1，c=-2	a>0,b<=0 a>1,c<=1	1-3-4	2，6
输入：a=-1，b=2，c=3	a<=0,b>0 a<=1,c>1	1-3-4	3，7
输入：a=-1，b=-2，c=-3	a<=0,b<=0 a<=1,c<=1	1-3-5	4，8

如何看待测试覆盖率

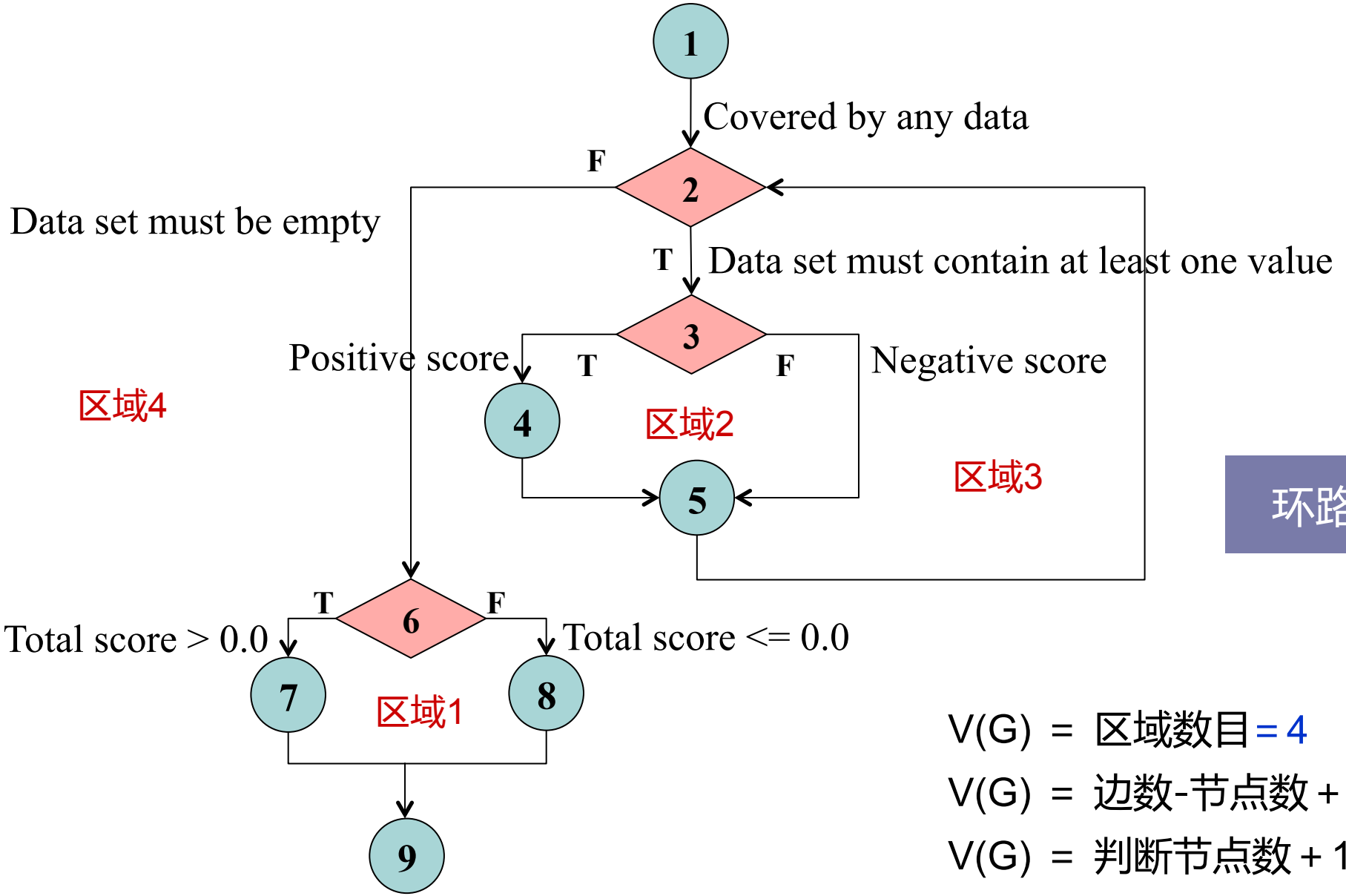
- 覆盖率数据只能代表测试过哪些代码，不能代表是否测试好这些代码。
- 较低的测试覆盖率能说明所做的测试还不够，但反之不成立。
- 路径覆盖 > 判定覆盖 > 语句覆盖
- 测试人员不能盲目追求代码覆盖率，而应该想办法设计更好的测试用例。
- 测试覆盖率应达到多少需要考虑软件整体的覆盖率情况以及测试成本。

基本路径测试

基本路径测试是在程序控制流图基础上，通过分析控制构造的环路复杂性，导出基本可执行路径集合，从而设计测试用例的方法。



```
FindMean(float *mean, FILE *fp)
{
    float sum = 0.0, score = 0.0;
    int num = 0;

    fscanf(fp, "%f", &score); /* Read and parse into score */
    while (!EOF(fp)) {
        if (score > 0.0) {
            sum += score;
            num++;
        }
        fscanf(fp, "%f", &score);
    }
    /* Compute the mean and print the result */
    if (num > 0) {
        *mean = sum/num;
        printf("The mean score is %f \n", mean);
    } else
        printf("No scores found in file\n");
}
```

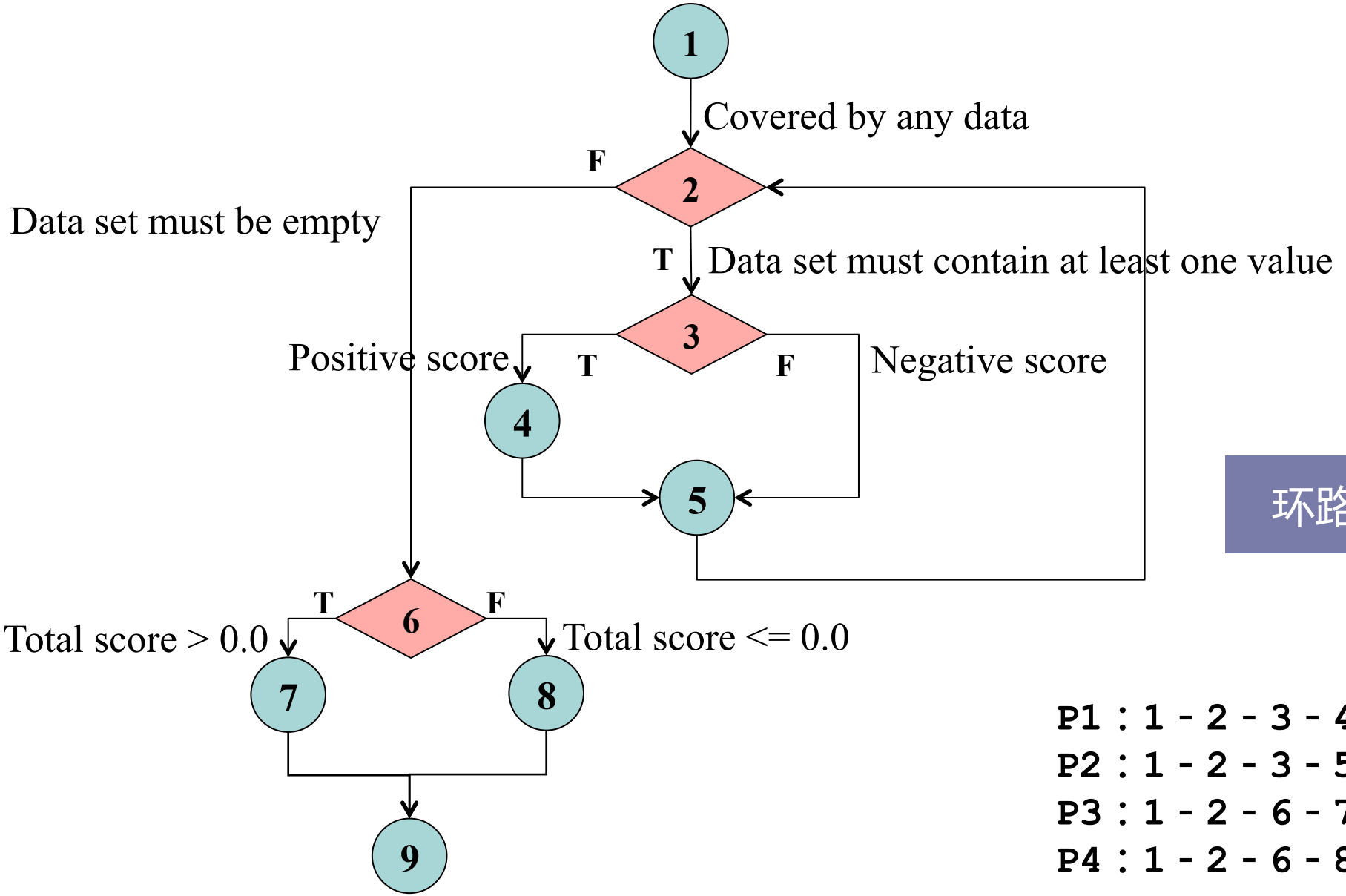


环路复杂性：4

V(G) = 区域数目 = 4

V(G) = 边数-节点数 + 2 = 11-9+2 = 4

V(G) = 判断节点数 + 1 = 3+1 = 4



环路复杂性：4

P1：1 - 2 - 3 - 4 - 5 - 2 - •••
P2：1 - 2 - 3 - 5 - 2 - •••
P3：1 - 2 - 6 - 7 - 9
P4：1 - 2 - 6 - 8 - 9

示例：基本路径测试

① 输入：**fp≠NULL**，文件有数据**{60.0,100.0,-1.0,110.0}**，覆盖路径**P1**

输出：打印信息“**The mean score is 90.0**”

② 输入：**fp≠NULL**，文件有数据**{0.0,-1.0,75.0}**，覆盖路径**P2**

输出：打印信息“**The mean score is 75.0**”

③ 输入：**fp≠NULL**，文件无任何数据，覆盖路径**P3**

输出：该路径不可达

④ 输入：**fp≠NULL**，文件无任何数据，覆盖路径**P4**

输出：打印信息“**No scores found in file**”

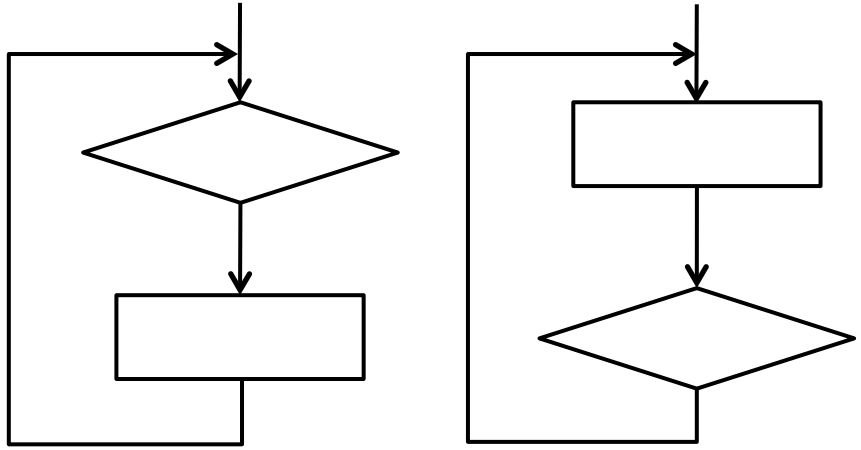
循环测试

循环测试

- 目的：检查循环结构的有效性
- 类型：简单循环、嵌套循环、串接循环和非结构循环

简单循环（次数为 n）

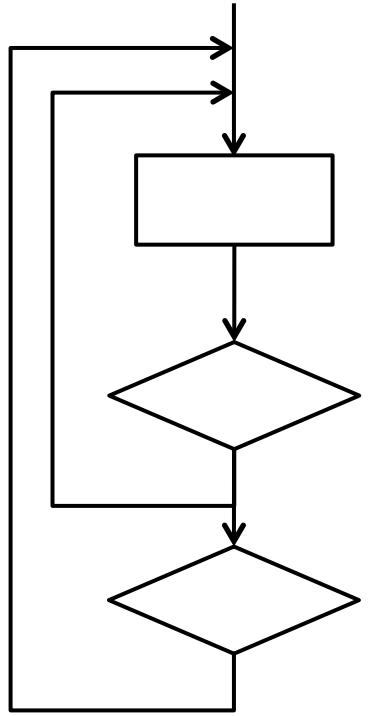
- 完全跳过循环
- 只循环 1 次
- 只循环 2 次
- 循环 m（ m < n ）次
- 分别循环 n-1, n, n+1 次



循环测试

嵌套循环

- 从最内层循环开始，所有外层循环次数设为最小值；
- 对最内层循环按照简单循环方法进行测试；
- 由内向外进行下一个循环的测试，本层循环的所有外层循环仍取最小值，而由本层循环嵌套的循环取某些“典型”值；
- 重复上一步的过程，直到测试完所有循环。



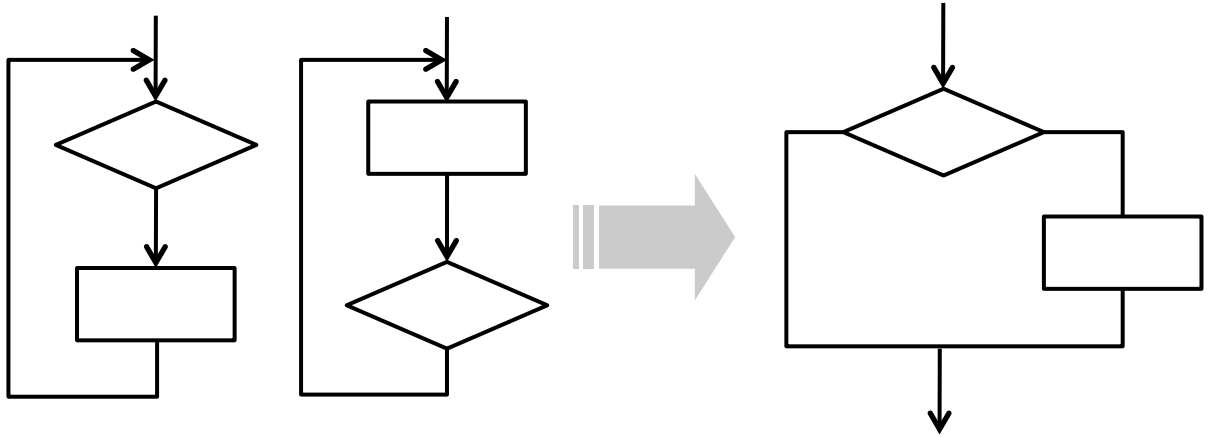
串接循环

- 独立循环：分别采用简单循环的测试方法；
- 依赖性循环：采用嵌套循环的测试方法。

循环测试

Z路径覆盖下的循环测试

- 这是路径覆盖的一种变体，将程序中的循环结构简化为选择结构的一种路径覆盖。
- 循环简化的目的是限制循环的次数，无论循环的形式和循环体实际执行的次数，简化后的循环测试只考虑执行循环体一次和零次（不执行）两种情况。



在循环简化的思路下，循环与判定分支的效果是一样的，即循环要么执行、要么跳过。

谢谢大家！

THANKS