6.4 软件测试技术

1.白盒测试
- 逻辑覆盖
- 基本路经测试
- 控制结构测试
- 数据流测试[*]
- 变异测试[*]

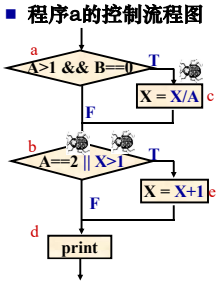
2. 黑盒测试
- 等价类划分
- 边界值分析
- 错误推测法[*]
- 因果图[*]
- 功能图[*]
- 接口测试[*]

请思考:如何检测下述程序中的Bug?

- 程序a

```
foo (int A, int B, int X)
{
 if (A>1 && B==0)
    X=X/A;
   /*bug1: "X/A"→ "A/X"*/
 if (A==2 || X>1)
   /*bug2: "||" → "&&" */
   /*bug3: "X>1"→ "X>2" */
    X++;
 printf("%d,%d,%d", A, B, X);
}
```

- 程序a的控制流程图

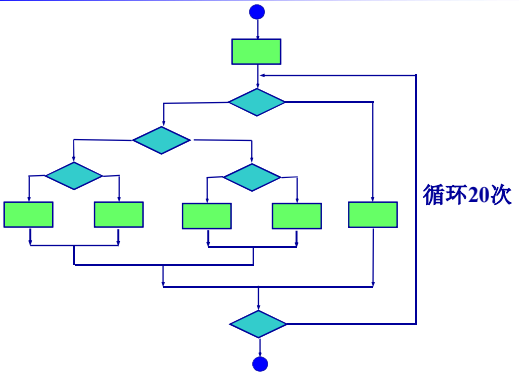


白盒测试

- 把测试对象看做一个玻璃盒子，它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。
- 通过在不同点检查程序的状态，确定实际的状态是否与预期的状态一致。因此白盒测试又称为结构测试或称为基于程序的测试。

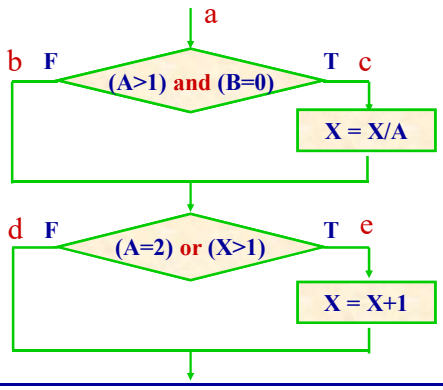
- 软件人员使用白盒测试方法，主要想对程序模块进行如下的检查：
  1) 对所有的逻辑判定，取“真”与取“假”的两种情况都至少测试一次 — 逻辑覆盖测试；
  2) 对程序模块的所有独立的执行路径至少测试一次 — 路径覆盖测试；
  3) 在循环的边界和运行界限内执行循环体 — 控制流测试；
  4) 测试内部数据结构的有效性 — 数据流测试。

- 对一个具有多重选择和循环嵌套的程序，不同的路径数目可能是天文数字。给出一个小程序的流程图，它包括了一个执行20次的循环。
- 包含的不同执行路径数达 5^{20} 条，对每一条路径进行测试需要1毫秒，假定一年工作365×24小时，要想把所有路径测试完，需3170年。



6.4.1 逻辑覆盖

- 逻辑覆盖是以程序内部的逻辑结构为基础的设计测试用例的技术。不同的覆盖标准又可分为：
  - 语句覆盖
  - 判定覆盖
  - 条件覆盖
  - 判定—条件覆盖
  - 条件组合覆盖
  - 路径覆盖



L1 (a → c → e)
= {(A > 1) and (B = 0)} and {(A = 2) or (X/A > 1)}
= (A > 1) and (B = 0) and (A = 2) or
(A > 1) and (B = 0) and (X/A > 1)
= (A = 2) and (B = 0) or
(A > 1) and (B = 0) and (X/A > 1)

L2 (a→ b → d)
= not{(A > 1) and (B = 0)} and not{(A = 2) or (X > 1)}
= { not (A > 1) or not (B = 0) } and
{ not (A = 2) and not (X>1) }
= not (A > 1) and not (A = 2) and not (X > 1) or
not (B = 0) and not (A = 2) and not (X > 1)

L3 (a→ b→ e)
= not {(A > 1) and (B = 0)} and {(A = 2) or (X > 1)}
= { not (A > 1) or not (B = 0)} and {(A = 2) or (X > 1)}
= not (A > 1) and (A = 2) or not (A> 1) and (X > 1) or
not (B = 0) and (A = 2)

L4 (a→ c → d)
= {(A > 1) and (B = 0)} and not {(A = 2) or (X/A > 1)}
= (A > 1) and (B = 0) and not (A = 2) and not (X/A > 1)

1. 语句覆盖

- 语句覆盖就是设计若干个测试用例，运行被测程序，使得每一可执行语句至少执行一次。
- 在图例中，正好所有的可执行语句都在路径L1上，所以选择路径 L1设计测试用例，就可以覆盖所有的可执行语句。

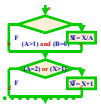
- 测试用例的设计格式如下
【输入的(A, B, X)，输出的(A, B, X)】
- 为图例设计满足语句覆盖的测试用例是:
【(2, 0, 4)，(2, 0, 3)】
- 覆盖 ace【L1】。“语句覆盖是最弱的逻辑覆盖准则” — 如果第一个判 断“and”错为“or”，第二个判断“or”错为“and”，使用以上 测试用例，查不出问题。

(A>1) AND (B=0) → (A>1) OR (B=0)
(A=2) OR (X>1) → (A=2) AND (X>1)

2. 判定覆盖

- 判定覆盖就是设计若干个测试用例，运行被测程序，使得程序中每个判断的取真分支和取假分支至少经历一次。
- 判定覆盖又称为分支覆盖。
- 对于图例，如果选择路径L1和L2，就可得满足要求的测试用例:

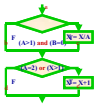
- 【(2, 0, 4)，(2, 0, 3)】覆盖 ace【L1】
  【(1, 1, 1)，(1, 1, 1)】覆盖 abd【L2】



(A = 2) and (B = 0) or
(A > 1) and (B = 0) and (X/A > 1)

not (A > 1) and not (A = 2) and not (X > 1)
or
not (B = 0) and not (A = 2) and not (X > 1)

- 如果选择路径L3和L4，还可得另一组可用的测试用例:
  【(2, 1, 1)，(2, 1, 2)】覆盖 abe【L3】
  【(3, 0, 3)，(3, 0, 1)】覆盖 acd【L4】



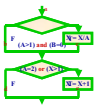
not (A > 1) and (X > 1) or not (B = 0) and
(A = 2) or not (B = 0) and (X > 1)

(A > 1) and (B = 0) and not (A = 2) and
not (X/A > 1)

3. 条件覆盖

- 条件覆盖就是设计若干个测试用例，运行被测程序，使得程序中每个判断的每个条件的可能取值至少执行一次。
- 在图例中，我们事先可对所有条件的取值加以标记。例如，
- 对于第一个判断：
  - ✓ 条件 A>1 取真为T_1，取假为$\overline{T_1}$
    条件 B=0 取真为T_2，取假为$\overline{T_2}$

- 对于第二个判断：
  - ✓ 条件A=2 取真为T_3，取假为 $\overline{T_3}$
    条件X>1 取真为T_4，取假为 $\overline{T_4}$

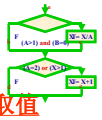


测试用例	覆盖分支	条件取值
【(2, 0, 4)，(2, 0, 3)】	L1(c, e)	$T_1T_2T_3T_4$
【(1, 1, 1)，(1, 1, 1)】	L2(b, d)	$T_1T_2\overline{T_3}T_4$
【(3, 1, 2)，(3, 1, 3)】	L3(b, e)	$T_1\overline{T_2}\overline{T_3}T_4$

- 需要注意的是，在测试用例中可能有些条件取值在执行时覆盖不到，需要增加测试用例。

4. 判定－条件覆盖

- 判定－条件覆盖就是设计足够的测试用例，使得判断中每个条件的可能取值至少执行一次，而且每个判断中的每个分支至少执行一次。

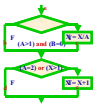


测 试 用 例	覆盖分支	条件取值
【(2, 0, 4)，(2, 0, 3)】	L1(c, e)	$T_1T_2T_3T_4$
【(1, 1, 1)，(1, 1, 1)】	L2(b, d)	$T_1T_2\overline{T_3}T_4$
【(3, 1, 2)，(3, 1, 3)】	L3(b, e)	$T_1\overline{T_2}\overline{T_3}T_4$

5. 条件组合覆盖

- 条件组合覆盖就是设计足够的测试用例，运行被测程序，使得每个判断的所有可能的条件取值组合至少执行一次。

记 ①A>1, B=0 作 T_1T_2
②A>1, B≠0 作 $T_1\overline{T_2}$
③A≯1, B=0 作 $\overline{T_1}T_2$
④A≯1, B≠0 作 $\overline{T_1}\overline{T_2}$

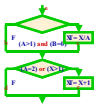


⑤A=2, X>1 作 T_3T_4
⑥A=2, X≯1 作 $T_3\overline{T_4}$
⑦A≠2, X>1 作 $\overline{T_3}T_4$
⑧A≠2, X≯1 作 $\overline{T_3}\overline{T_4}$

测试用例	覆盖条件	覆盖组合
【(2, 0, 4), (2, 0, 3)】(L1)	$T_1T_2T_3T_4$	①,⑤
【(2, 1, 1), (2, 1, 2)】(L3)	$T_1\overline{T_2}T_3\overline{T_4}$	②,⑥
【(1, 0, 3), (1, 0, 4)】(L3)	$\overline{T_1}T_2\overline{T_3}T_4$	③,⑦
【(1, 1, 1), (1, 1, 1)】(L2)	$\overline{T_1}\overline{T_2}\overline{T_3}\overline{T_4}$	④,⑧

6. 路径测试

- 路径测试就是设计足够的测试用例，覆盖程序中所有可能的路径。



测试用例	通过路径	覆盖条件
【(2, 0, 4), (2, 0, 3)】	ace (L1)	$T_1T_2T_3T_4$
【(1, 1, 1), (1, 1, 1)】	abd (L2)	$\overline{T_1}\overline{T_2}\overline{T_3}\overline{T_4}$
【(1, 1, 2), (1, 1, 3)】	abe (L3)	$\overline{T_1}\overline{T_2}\overline{T_3}T_4$
【(3, 0, 3), (3, 0, 1)】	acd (L4)	$T_1T_2\overline{T_3}\overline{T_4}$

逻辑覆盖小结

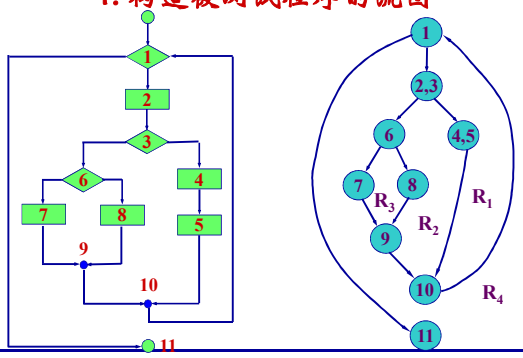
- 不同逻辑覆盖标准对源程序检测的详尽程度不一样：
  - 语句覆盖是最弱的逻辑覆盖标准（对程序的逻辑覆盖很少）
  - 判定覆盖比语句覆盖强（关心判定表达式的值）
  - 条件覆盖通常比判定覆盖强(关心判定表达式中的每个条件)

- 判定-条件覆盖通常比判定覆盖和条件覆盖强,有时判定-条件覆盖也并不比条件覆盖更强（关心判定表达式及其条件的取值）
- 条件组合覆盖是最强的逻辑覆盖标准，比语句覆盖，判定覆盖，条件覆盖，判定-条件覆盖都要强。

6.4.2 基本路径测试

- 基本路径测试是McCabe提出的一种白盒测试技术。基本路径测试方法把覆盖的路径数压缩到一定限度内，程序中的循环体最多只执行一次。
- 在程序控制流图的基础上，分析控制构造的环路复杂性，导出基本可执行路径集合，设计测试用例的方法。设计出的测试用例要保证在测试中，程序的每一个可执行语句至少要执行一次，而且每个条件在执行时都将分别取真、假两种值。
- 采用基本路径测试技术设计测试用例的步骤。

1. 构造被测试程序的流图



2. 计算程序环路复杂性

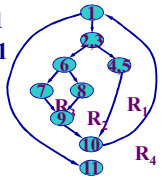
- 环形复杂度定量度量程序的逻辑复杂性。有了描绘程序控制流的流图之后，可以用先前讲述的3种方法之一计算环形复杂度。
- 图示的流图复杂性为4。

3.确定线性独立路径的基本集合

- 所谓独立路径是指至少引入程序的一个新处理语句集合或一个新条件的路径。从流图来看，一条独立路径是至少包含有一条在其他独立路径中从未有过的边的路径。
- 程序环路复杂性给出了程序基本路径集中的独立路径条数，这是确保程序中每个可执行语句至少执行一次所必需的测试用例数目的上界。

- 例如，在图示的流图中，一组独立的路径是

path1：1 - 11
path2：1 - 2 - 3 - 4 - 5 - 10 - 1 - 11
path3：1 - 2 - 3 - 6 - 8 - 9 - 10 - 1 - 11
path4：1 - 2 - 3 - 6 - 7 - 9 - 10 - 1 – 11



- 路径 path1~path4组成了流图的基本路径集。

4. 导出测试用例

- 导出测试用例，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执行 — 用逻辑覆盖的方法。
- 每个测试用例执行之后，与预期结果进行比较。如果所有测试用例都执行完毕，则可以确信程序中所有的可执行语句至少被执行了一次。
- 注意：一些独立的路径(如path1)，往往不是完全孤立的。有时它是程序正常的控制流的一部分，这时，将这些路径作为另一条路径的一部分来测试。

6.4.3 控制结构测试

- 尽管基本路径测试技术简单而且高效，但是仅有这种技术还不够，还需要使用其他控制结构测试技术，才能进一步提高白盒测试的质量。
  - 条件测试：检查程序中包含逻辑条件的测试用例设计方法。
  - 循环测试：侧重于循环构成元素的有效性。
  - 数据流测试：按照程序中变量定义和使用的位置来选择程序的测试路经。

1. 条件测试

- 用条件测试技术设计出的测试用例，能够检查程序模块中包含的逻辑条件。其目的是检测程序条件的错误和程序的其他错误。
- 条件分为简单条件和复合条件：
  - ✓简单条件：一个布尔变量或一个关系表达式，在它们之前还可能有一个NOT算符；
  - ✓复合条件：两个或多个简单条件、布尔算符和括弧组成。
  - ✓不包含关系表达式的条件称为布尔表达式。

- 如果条件不正确，则至少条件的一个成分不正确。常见的条件错误的类型如下：
  - ✓布尔算符错（布尔算符不正确，遗漏布尔算符或有多余的布尔算符）
  - ✓布尔变量错
  - ✓布尔括弧错
  - ✓关系算符错
  - ✓算术表达式错

- 几种条件测试策略：
  - ✓分支测试：对于复合条件C来说，C的真分支和假分支以及C中的每个简单条件，都应该至少执行一次。最简单的条件测试策略。
  - ✓域测试：一个关系表达式执行3个或4个测试。对于E1<关系算符>E2来说，需要3个测试（分别使E1的值大于、等于或小于E2的值）。为了发现E1和E2中的错误，应该使这两个值之间的差别尽可能小的测试数据（E1值大于或小于E2值）。

- ✓分支与关系运算符测试：K.C.Tai提出的被称为BRO (Branch and Relational Operator)测试的条件测试策略。如果在条件中所有布尔变量和关系算符都只出现一次而且没有公共变量，则BRO测试保证能发现该条件中的分支错和关系算符错。

- BRO测试策略及举例：

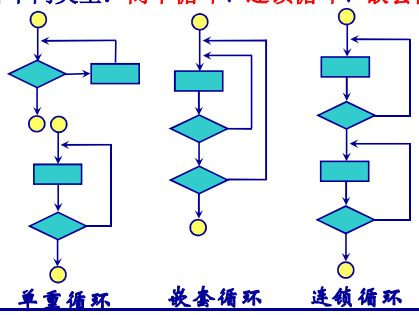
包含n个简单条件的条件C的条件约束定义为（D_1，D_2，…，D_n），其中D_i(0<i≤n)表示条件C中第i个简单条件的输出约束。如果在条件C的一次执行过程中，C中每个简单条件的输出都满足D中对应的约束，则称C的这次执行覆盖了C的条件约束D。对于布尔变量B来说，B的输出约束指出，B必须是真（t）或假（f）。类似地，对于关系表达式来说，用符号>，=和<指定表达式的输出约束。

例子：条件C为B_1 and B_2

BRO策略要求约束集{(t,f),(t,t),(f,t)}为C的执行所覆盖。（n个变量需要2^n个测试）。

2. 循环测试

- 3种不同类型：简单循环、连锁循环、嵌套循环。

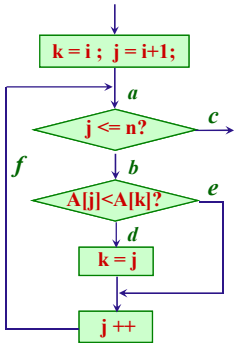


(1) 简单循环

① 零次循环：从循环入口到出口
② 一次循环：检查循环初始值
③ 二次循环：检查多次循环
④ m次循环： 检查在多次循环
⑤ 最大次数循环、比最大次数多一次、少一次的循环。

简单循环设计测试用例的例子：求最小值

```
k = i;
for ( j = i+1; j <= n; j++ )
  if ( A[j] < A[k] )
    k = j;
```



测试用例选择

循环	*i*	*n*	A[*i*]	A[*i*+1]	A[*i*+2]	*k*	路 径
0	1	1				*i*	*a c*
1	1	2	1	2		*i*	*a b e f c*
			2	1		*i*+1	*a b d f c*
2	1	3	1	2	3	*i*	*a b e f e f c*
			2	3	1	*i*+2	*a b e f d f c*
			3	2	1	*i*+2	*a b d f d f c*
			3	1	2	*i*+1	*a b d f e f c*

d 改 *k* 的值，*e* 不改 *k* 的值

(2) 嵌套循环 (Beizer提出)

① 对最内层循环做简单循环的全部测试。所有其他层的循环变量置为最小值；

② 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，所有其他嵌套内层循环的循环变量取“典型”值。

③ 反复进行，直到所有各层循环测试完毕。

④ 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。

(3) 连锁循环

如果各个循环互相独立，则可以用与简单循环相同的方法进行测试。但如果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。

3. 数据流覆盖[*]

- 控制流测试面向程序的结构，不关心程序中每个语句是如何实现的。与控制流测试思想不同，**数据流测试**是面向程序中的变量。
- **数据流测试**着重测试程序中的数据的定义和使用是否正确。按照程序中变量**定义**(Definition)和**使用**(Use)的位置来选择程序的**测试路径**。

黑盒测试

- 把测试对象看做一个黑盒，测试人员完全不考虑程序内部的逻辑结构和内部特性，只依据程序的需求和功能规格说明，检查程序的功能是否符合它的功能说明。
- 黑盒测试叫做功能测试或基于规格说明的测试。

- 黑盒测试方法主要是为了发现以下错误:
  1) 是否有不正确或遗漏了的功能?
  2) 在接口上，输入能否正确地接受? 能否输出正确的结果?
  3) 是否有数据结构错误或外部信息 (例如数据文件) 访问错误?
  4) 性能上是否能够满足要求?
  5) 是否有初始化或终止性错误?

- 用黑盒测试发现程序错误，必须在所有可能的输入条件和输出条件中确定测试数据，检查程序能否产生正确的输出。
- 但这是不可能的。例如，设一个程序P有输入量X和Y及输出量Z。在字长为32位的计算机上运行。若X、Y取整数，按黑盒方法进行穷举测试：可能采用的测试数据组个数：

  $2^{32} \times 2^{32} = 2^{64}$

- 如果测试一组数据需要1毫秒, 一年工作365×24小时，完成所有测试需5亿年。

6.4.4等价类划分

- 等价类划分是一种典型的黑盒测试方法，使用这一方法时，完全不考虑程序的内部结构，只依据程序的规格说明来设计测试用例。
- 等价类划分方法把所有可能的输入数据，即程序的输入域划分成若干部分，然后从每一部分中选取少数代表性的数据做为测试用例。
- 使用这一方法设计测试用例要经历划分等价类（列出等价类表）和选取测试用例两步。

1. 等价类划分

- 等价类是指某个输入域的子集合，在该子集合中，各个输入数据对于揭露程序中的错误都是等效的，测试某等价类的代表值就等价于对这一类其他值的测试。
- 等价类的划分有两种不同的情况:
  ① 有效等价类：合理的、有意义的输入数据构成的集合。
  ② 无效等价类：不合理的、无意义的输入数据构成的集合。
- 在设计测试用例时，要同时考虑有效等价类和无效等价类的设计。

划分等价类的启发式规则

(1) 如果输入条件规定了取值范围或值的个数，则可确立一个有效等价类和两个无效等价类。

例如，在程序规格说明中对输入条件有一句话：

“…… 项数可以从1到999 ……”

则有效等价类是“1≤项数≤999”

两个无效等价类是“项数＜1”或“项数＞999”。

(2) 如果输入条件规定了输入值的集合，或者是规定了“必须如何”的条件，这时可确立一个有效等价类和一个无效等价类。

例如，在Pascal语言中对变量标识符规定为“以字母打头的……串”。那么所有以字母打头的构成有效等价类，而不在此集合内（不以字母打头）的归于无效等价类。

(3) 如果输入条件是一个布尔量，则可以确定一个有效等价类和一个无效等价类。

(4) 如果规定了输入数据的一组值，而且程序要对每个输入值分别进行处理。这时可为每一个输入值确立一个有效等价类，此外针对这组值确立一个无效等价类，它是所有不允许的输入值的集合。

例如，在教师上岗方案中规定对教授、副教授、讲师和助教分别计算分数，做相应的处理。因此可以确定4个有效等价类为教授、副教 授、讲师和助教，一个无效等价类，它是所有不符合以上身分的人员的输入值的集合。

(5) 如果规定了输入数据必须遵守的规则，则可以确立一个有效等价类（符合规则）和若干个无效等价类（从不同角度违反规则）。

例如，Pascal语言规定 “一个语句必须以分号‘;’结束”。这时可以确定一个有效等价类“以‘;’结束”，若干个无效等价类“以‘:’结束”、“以‘,’结束”、“以‘ ’结束”、“以LF结束”。

2.确立测试用例

a) 在确立了等价类之后，建立等价类表，列出所有划分出的等价类。

输入条件	有效等价类	无效等价类
……	……	……
……	……	……
……	……	……

b) 从划分出的等价类中按以下原则选择测试用例：

1) 为每一个等价类规定一个唯一编号；
2) 设计一个新的测试用例，使其尽可能多地覆盖尚未被覆盖的有效等价类，重复这一步，直到所有的有效等价类都被覆盖为止；
3) 设计一个新的测试用例，使其仅覆盖一个尚未被覆盖的无效等价类，重复这一步，直到所有的无效等价类都被覆盖为止。

- 用等价类划分法设计测试用例的实例

在某一PASCAL语言版本中规定：

“标识符是由字母开头，后跟字母或数字的任意组合构成。有效字符数为8个，最大字符数为80个。”

并且规定：“标识符必须先说明，再使用。”“在同一说明语句中，标识符至少必须有一个。”

用等价类划分方法，建立输入等价类表：

输入条件	有效等价类	无效等价类
说明语句中标识符个数	1个(1)，多个(2)	0个(3)
标识符中字符数	1~8个(4)	0个(5)，>8个(6)，>80个(7)
标识符组成	字母(8)，数字(9)	非字母数字字符(10)，保留字(11)
标识符第一个字符	字母(12)	非字母(13)
标识符使用	先说明后使用(14)	未说明就使用(15)

- 下面选取了 9 个测试用例，它们覆盖了所有的等价类。

① VAR *x*，T1234567：REAL；
BEGIN *x* := 3.414；
T1234567 := 2.732；
……
(1), (2), (4), (8), (9), (12), (14)

② VAR ： REAL； (3)

③ VAR *x*，： REAL； (5)

④ VAR T12345678：REAL； (6)

⑤ VAR T12345……：REAL； (7)
多于80个字符

⑥ VAR T$：CHAR； (10)

⑦ VAR GOTO：INTEGER； (11)

⑧ VAR 2T：REAL； (13)

⑨ VAR PAR：REAL； (15)
BEGIN ……
PAP := SIN (3.14 * 0.8) / 6；

6.4.5 边界值分析

- 边界值分析是一种黑盒测试方法，对等价类划分方法的补充(常常联合使用等价类划分和边界值分析两种技术)。
- 人们从长期的测试工作经验得知，大量的错误是发生在输入或输出范围的边界上，而不是在输入范围的内部。因此针对各种边界情况设计测试用例，可以查出更多的错误。

- 例如，有一段用C编写的小程序：

```
int A[20]; int i;
for ( i = 1; i <= 10; i++ )
A[i] = −1;
```

- 因为C语言中数组下标从0开始，而本程序中从1开始赋值，如果以后用户不了解，可能从1开始使用，就会出错。所以边界值可能查出更多的问题来。

- 如何确定边界？通常的边界检查原则：
  - a)类型：数字、字符、位置、质量、大小、速度、方位、尺寸、空间等。
  - b)边界值：最大 / 最小、首位 / 末位、上 / 下、最快 / 最慢、最高 / 最低、最短 / 最长、空 / 满等。
- 使用边界值分析，最重要的是确定正确的边界值域。对于输入 / 输出等价类，选取正好等于、刚刚大于和刚刚小于边界值的数据作为测试数据。

- 选取测试用例的原则：

(1) 如果输入条件规定了值的范围，则应取刚刚到达这个范围边界的值，以及刚刚超过这个范围边界的值作为测试输入数据。

例如，某数据的取值范围为-1.0～1.0，测试数据可取-1.0、1.0，以及-1.1、1.1。

(2) 如果输入条件规定了值的个数，则应取最大个数、最小个数、比最大个数多1，比最小个数少1的数作为测试输入数据。

例如，某文件有255个记录，测试数据可取1、255，以及0、256。

(3) 根据规格说明和每个输出条件，使用原则(1)。

例如，研究生录取分数范围84～150，测试数据可取84、150，以及83、151。

(4) 根据规格说明和每个输出条件，使用原则(2)。

例如，研究生录取人数34人，测试数据可取1、34、以及0、35。

(5) 如果程序的规格说明给出的输入域或输出域是有序集合（如有序表），则选取集合的第一个元素和最后一个原则作为测试用例。

例如，学生文件的学生记录按学号存放，班上总共30人，测试数据可取第1、第30个学生。

(6) 如果程序中使用了一个内部数据结构，则应选择此数据结构的边界上的值作为测试用例。

(7) 分析规格说明，找出其他可能的边界条件。

6.4.6 错误推测法[*]

- 人们也可以靠经验和直觉推测程序中可能存在的各种错误，从而有针对性地编写检查这些错误的例子。这就是错误推测法。
- 错误推测法的基本想法是：列举出程序中所有可能有的错误和容易发生错误的特殊情况，根据它们选择测试用例。
- 可以利用不同测试阶段的经验和对软件系统的认识来设计测试用例。
  - ✓例如，在单元测试中某程序模块已经遇到错误，

在系统测试中可以在这些可能出现问题的地方再组织测试用例。

- ✓在前一个版本中发现的常见错误在下一个版本的测试中有针对性地设计测试用例。
- 根据以上想法，测试用例的设计原则：
  - ✓客观因素：产品以前版本已出现的问题。
  - ✓已经因素：语言、操作系统、浏览器的限制可能带来的问题。
  - ✓经验：由模块之间关联所联想到的测试；由修复软件的错误可能会带来的问题。

使用各种测试方法的综合策略

- 在任何情况下都必须使用边界值分析法。用这种方法设计出测试用例发现程序错误的能力最强。
- 必要时用等价类划分法补充一些测试用例。
- 用错误推测法再追加一些测试用例。
- 对照程序逻辑，检查已有测试用例的逻辑覆盖程度。如果未达到要求的覆盖标准，应再补充足够的测试用例。