程序编码

- 结构化程序设计
- 程序设计风格
- 程序效率
- 程序复杂性度量

- 做为软件工程过程的一个阶段，程序编码是设计的继续。
- 程序设计语言的特性和程序设计风格会深刻地影响软件的质量和可维护性。
- 为了保证程序编码的质量，程序员必须深刻地理解、熟练地掌握并正确地运用程序设计语言的特性。此外，还要求源程序具有良好的结构性和良好的程序设计风格。

结构化程序设计

结构化程序设计主要包括两方面：

(1) 在编写程序时，强调使用几种基本控制结构，通过组合嵌套，形成程序的控制结构。尽可能避免使用GOTO语句。

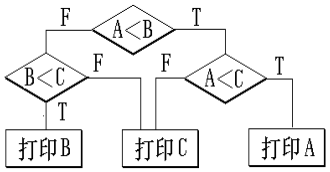
(2) 在程序设计过程中，尽量采用自顶向下和逐步细化的原则，由粗到细，一步步展开。

结构化程序设计的主要原则

- 使用语言中的顺序、选择、重复等有限的基本控制结构表示程序逻辑。
- 选用的控制结构只准许有一个入口和一个出口。
- 程序语句组成容易识别的块，每块只有一个入口和一个出口。
- 复杂结构应该用基本控制结构进行组合嵌套来实现。

- 语言中没有的控制结构，可用一段等价的程序段模拟，但要求该程序段在整个系统中应前后一致。
- 严格控制**GOTO**语句，仅在下列情形才可使用：
  ① 用一个非结构化的程序设计语言去实现一个结构化的构造。
  ② 若不使用**GOTO**语句就会使程序功能模糊。
  ③ 在某种可以改善而不是损害程序可读性的情况下。

例1 打印*A, B, C*三数中最小者的程序



程序1

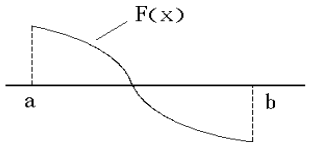
```
      if ( A < B ) goto 120;
      if ( B < C ) goto 110;
100   write ( C );
      goto 140;
110   write ( B );
      goto 140;
120   if ( A < C ) goto 130;
      goto 100;
130   write ( A );
140   end
```


程序2

```
if ( A < B ) and ( A < C ) then
  write ( A )
else
  if ( A ≥ B ) and ( B < C ) then
    write ( B )
  else
    write ( C )
  endif
endif
```


例2 用二分法求方程 $f(x)=0$ 在区间$[a..b]$中的根**的程序**

假设在闭区间$[a..b]$上函数 $f(x)$ 有唯一的一个零点




```
f0 = f(a);  f1 = f(b);        //程序1
if ( f0 * f1 <= 0 ) {
  x0 = a;  x1 = b;
  for ( i = 1;  i <= n;  i++ ){
    xm = ( x0+x1 ) / 2;  fm = f(xm);
    if ( abs (fm) < eps || abs (x1−x0) < eps)
      goto finish;
    if ( f0 * fm > 0 )
      { x0 = xm;  f0 = fm; }
    else x1 = xm;
  }
```



```
finish: printf ("\n The root of this
  equation
        is %d\n", xm);
}
```

- 单入口，两出口
- 正常出口是循环达到 *n* 次，非正常出口是循环中途控制转出到标号 *finish* 所在位置
- 可读性好


```
f0 = f(a);  f1 = f(b);    //程序2
if ( f0 * f1 <= 0 ) {
  x0 = a;  x1 = b;
  for ( i = 1;  i <= n;  i++ ) {  //正常出口
    xm = ( x0+x1 ) / 2;  fm = f(xm);
    if (abs( fm)<eps || abs( x1−x0)<eps)
      break;       //非正常出口
    if ( f0 * fm>0)
      { x0 = xm;  f0 = fm; }
    else
      x1 = xm;
  }
}
```



```
f0 = f(a);  f1 = f(b);    //程序3
if ( f0 * f1 <= 0){
  x0 = a;  x1 = b;  i = 1;  finished = 0;
  while (i <= n && finished == 0) {
    xm = ( x0+x1 ) / 2;  fm = f(xm);
    if (abs( fm)<eps || abs( x1−x0)<eps)
      finished = 1;
    if (finished == 0)
```



```
      if ( f0 * fm>0 )
        { x0 = xm;  f0 = fm; }
      else
        x1 = xm ;
  }
}
```

- **引入布尔变量 *finished*，改 for 型循环为 while 型，将单入口多出口结构改为单入口单出口结构。**

自顶向下，逐步求精

- 在详细设计和编码阶段，应当采取自顶向下，逐步求精的方法。
- 把一个模块的功能逐步分解，细化为一系列具体的步骤，进而翻译成一系列用某种程序设计语言写成的程序。

例，用筛选法求100以内的素数

- 筛选法就是从2到100中去掉2, 3, …, 9, 10的倍数，剩下的就是100以内的素数。
- 为了解决这个问题，可先按程序功能写出一个框架。

```
main ( ) {          //程序框架
   建立2到100的数组A[ ]，其中A[i]＝i；
     ----------------------------------1
   建立2到10的素数表 B[ ]，其中存放2
     到10以内的素数；----------2
   若A[i]＝i是B[ ]中任一数的倍数，则
     剔除A[i]；----------------------3
   输出A[ ]中所有没有被剔除的数；
     ----------------------------------4
}
```

17

```
main ( ) {
   /*建立2到100的数组A[ ]，其中A[i]＝i*/
   for ( i = 2；i <= 100；i++ ) A[i] = i；
   /* 建立2到10的素数表B[ ]，其中存放2
   到10以内的素数*/
   B[1] =2；B[2] = 3；B[3] = 5；B[4] = 7；
   /*若A[i]＝i是B[ ]中任一数的倍数，则
   剔除A[i]*/
   for ( j = 1；j <= 4；j++ )
     检查A[ ]所有的数能否被B[j]整除并将
     能被整除的数从A[ ]中剔除； -----3.1
```

18

```
   /*输出A[ ]中所有没有被剔除的数*/
   for ( i = 2；i <= 100；i++ )
     若A[i]没有被剔除，则输出之 ---4.1
}
```

- 对框架中的局部再做细化，得到整个程序。

19

```
main ( ) {
   /*建立2到100的数组A[ ]，其中A[i]＝i*/
   for ( i = 2；i <= 100；i++ ) A[i] = i；
   /* 建立2到10的素数表B[ ]，其中存放2
   到10以内的素数*/
   B[1] =2；B[2] = 3；B[3] = 5；B[4] = 7；
   /*若A[i]＝i是B[ ]中任一数的倍数，则剔
   除A[i]*/
   for ( j = 1；j <= 4；j++ )
   /*检查A[ ]所有的数能否被B[j]整除并将
   能被整除的数从A[ ]中剔除*/
```

20

```
   for ( i = 2；i <= 100；i++）
     if ( A[i] / B[j] * B[j] == A[i] )
       A[i] = 0；
       /*输出A[ ]中所有没有被剔除的
       数*/
   for ( i = 2； i <= 100；i++）
     /*若A[i]没有被剔除，则输出之*/
   if ( A[i] != 0 ）
     printf ( “A[%d]＝%d\n”，i，A[i] )；
}
```

21

自顶向下，逐步求精方法的优点

- 符合人们解决复杂问题的普遍规律。可提高软件开发的成功率和生产率
- 用先全局后局部，先整体后细节，先抽象后具体的逐步求精的过程开发出来的程序具有清晰的层次结构，程序容易阅读和理解

22

- 程序自顶向下，逐步细化，分解成一个树形结构。在同一层的节点上的细化工作相互独立。有利于编码、测试和集成
- 程序清晰和模块化，使得在修改和重新设计一个软件时，可复用的代码量最大
- 每一步工作仅在上层节点的基础上做不多的设计扩展，便于检查
- 有利于设计的分工和组织工作。

程序设计风格

- 程序实际上也是一种供人阅读的文章，有一个文章的风格问题。应该使程序具有良好的风格。
  - 源程序文档化
  - 数据说明
  - 语句结构
  - 输入／输出方法

24

源 程 序 文 档 化

- 标识符的命名
- 安排注释
- 程序的视觉组织

25

符号名的命名

- 符号名即标识符，包括模块名、变量名、常量名、标号名、子程序名、 、数据区名以及缓冲区名等。
- 这些名字应能反映它所代表的实际东西，应有一定实际意义。
- 例如，表示次数的量用*Times*，表示总量的用*Total*，表示平均值的用*Average*，表示和的量用*Sum*等。

26

- 名字不是越长越好，应当选择精炼的意义明确的名字。必要时可使用缩写名字，但这时要注意缩写规则要一致，并且要给每一个名字加注释。同时，在一个程序中，一个变量只应用于一种用途。
- *NEW.BALANCE.ACCOUNTS.PAYABLE* (PASCAL)
- *NBALAP* (FORTRAN)
- *N* (BASIC)

27

程序的注释

- 夹在程序中的注释是程序员与日后的程序读者之间通信的重要手段。
- 注释决不是可有可无的。
- 一些正规的程序文本中，注释行的数量占到整个源程序的1／3到1／2，甚至更多。
- 注释分为序言性注释和功能性注释。

28

序言性注释

- 通常置于每个程序模块的开头部分，它应当给出程序的整体说明，对于理解程序本身具有引导作用。有些软件开发部门对序言性注释做了明确而严格的规定，要求程序编制者逐项列出。
- 有关项目包括：
  - 程序标题；

29

- 有关本模块功能和目的的说明；
- 主要算法；
- 接口说明：包括调用形式，参数描述，子程序清单；
- 有关数据描述：重要的变量及其用途，约束或限制条件，以及其它有关信息；
- 模块位置：在哪一个源文件中，或隶属于哪一个软件包；
- 开发简历：模块设计者，复审者，复审日期，修改日期及有关说明等。

30

功能性注释

- 功能性注释嵌在源程序体中，用以描述其后的语句或程序段是在做什么工作，或是执行了下面的语句会怎么样。而不要解释下面怎么做。
- 例如，

```
/* ADD AMOUNT TO TOTAL */
TOTAL = AMOUNT +TOTAL
```

不好。

31

- 如果注明把月销售额计入年度总额，便使读者理解了下面语句的意图：

```
/* ADD MONTHLY-SALES TO ANNUAL-
TOTAL */
TOTAL = AMOUNT +TOTAL
```

- 要点
  - 描述一段程序，而不是每一个语句；
  - 用缩进和空行，使程序与注释容易区别；
  - 注释要正确。

32

视觉组织　空格、空行和移行

- 恰当地利用空格，可以突出运算的优先性，避免发生运算的错误。
- 例如，将表达式
  (*A*＜－17)*ANDNOT*(*B*＜＝49)*ORC*
  写成
  (*A*＜－17) *AND NOT* (*B*＜＝49) *OR C*
- 自然的程序段之间可用空行隔开；

- 移行也叫做向右缩格。它是指程序中的各行不必都在左端对齐，都从第一格起排列。这样做使程序完全分不清层次关系。
- 对于选择语句和循环语句，把其中的程序段语句向右做阶梯式移行。使程序的逻辑结构更加清晰。
- 例如，两重选择结构嵌套，写成下面的移行形式，层次就清楚得多。


```
IF（…） THEN
    IF（…） THEN
        ……
    ELSE
        ……
    ENDIF
    ……
ELSE
 ……
ENDIF
```


数据说明

- 在设计阶段已经确定了数据结构的组织及其复杂性。在编写程序时，则需要注意数据说明的风格。
- 为了使程序中数据说明更易于理解和维护，必须注意以下几点。
  1. 数据说明的次序应当规范化
  2. 说明语句中变量安排有序化
  3. 使用注释说明复杂数据结构

数据说明的次序应当规范化

- 数据说明次序规范化，使数据属性容易查找，也有利于测试，排错和维护。
- 原则上，数据说明的次序与语法无关，其次序是任意的。但出于阅读、理解和维护的需要，最好使其规范化，使说明的先后次序固定。

- 例如，在FORTRAN程序中数据说明次序
  ① 常量说明
  ② 简单变量类型说明
  ③ 数组说明
  ④ 公用数据块说明
  ⑤ 所有的文件说明
- 在类型说明中还可进一步要求。例如，可按如下顺序排列：
  ① 整型量说明
  ② 实型量说明
  ③ 字符量说明
  ④ 逻辑量说明

说明语句中变量安排有序化

- 当多个变量名在一个说明语句中说明时，应当对这些变量按字母的顺序排列。带标号的全程数据(如FORTRAN的公用块)也应当按字母的顺序排列。
- 例如，把
  integer *size, length, width, cost, price*
  写成
  integer *cost, length, price , size, width*

使用注释说明复杂数据结构

- 如果设计了一个复杂的数据结构，应当使用注释来说明在程序实现时这个数据结构的固有特点。
- 例如, 对*PL/1*的链表结构和*Pascal*中用户自定义的数据类型，都应当在注释中做必要的补充说明。

语句结构

- 在设计阶段确定了软件的逻辑流结构，但构造单个语句则是编码阶段的任务。语句构造力求简单，直接，不能为了片面追求效率而使语句复杂化。

1. 在一行内只写一条语句

- 在一行内只写一条语句，并且采取适当的移行格式，使程序的逻辑和功能变得更加明确。
- 许多程序设计语言允许在一行内写多个语句。但这种方式会使程序可读性变差。因而不可取。

- 例如，有一段排序程序

```
FOR I:=1 TO N－1 DO BEGIN T:=I;
FOR J:=I＋1 TO N DO IF A[J]＜A[T]
THEN T:=J；IF T≠I THEN BEGIN
WORK:=A[T]；A[T]:=A[I];
A[I]:=WORK；END END;
```

- 由于一行中包括了多个语句，掩盖了程序的循环结构和条件结构，使其可读性变得很差。


```
FOR I:=1 TO N-1 DO      //改进布局
  BEGIN
    T:=I;
    FOR J:=I＋1 TO N DO
      IF A[J]＜A[T] THEN T:=J;
    IF T≠I THEN
      BEGIN
        WORK:=A[T];
        A[T]:=A[I];
        A[I]:=WORK;
      END
  END;
```


2. 程序编写首先应当考虑清晰性

- 程序编写首先应当考虑清晰性，不要刻意追求技巧性，使程序编写得过于紧凑。
- 例如，有一个用Pascal语句写出的程序段：

```
A[I]:=A[I]＋A[T];
A[T]:=A[I]－A[T];
A[I]:=A[I]－A[T];
```


- 此段程序可能不易看懂，有时还需用实际数据试验一下。
- 实际上，这段程序的功能就是交换**A[I]**和**A[T]**中的内容。目的是为了节省一个工作单元。如果改一下：

```
WORK:=A[T];
A[T]:=A[I];
A[I]:=WORK;
```

就能让读者一目了然了。

3. 程序要能直截了当地说明程序员的用意。

- 程序编写得要简单，写清楚，直截了当地说明程序员的用意。例如，

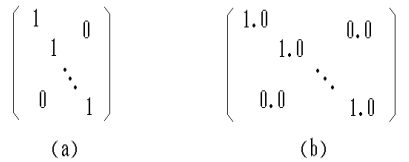
```
DO 5 I＝1, N
DO 5 J＝1, N
5   V(I, J) ＝ (I / J) * (J / I)
```

除法运算（/）在除数和被除数都是整型量时，其结果只取整数部分，而得到整型量。

当**I＜J**时, **I / J = 0**
当**J＜I**时, **J / I = 0**
得到的数组
当**I≠J**时
V(I, J) = (I / J) * (J / I) = 0
当**I＝J**时
V(I, J) = (I / J) * (J / I) = 1
这样得到的结果V是一个单位矩阵。

- 写成以下的形式，就能让读者直接了解程序编写者的意图。

```
DO 5 I=1，N
  DO 5 J=1，N
    IF ( I .EQ. J ) THEN
      V( I, J ) = 1.0
    ELSE
      V( I, J ) = 0.0
    ENDIF
5   CONTINUE
```

4. 除非对效率有特殊的要求，程序编写要做到清晰第一，效率第二。不要为了追求效率而丧失了清晰性。事实上，程序效率的提高主要应通过选择高效的算法来实现。
5. 首先要保证程序正确，然后才要求提高速度。反过来说，在使程序高速运行时，首先要保证它是正确的。

6. 避免使用临时变量而使可读性下降。例如，有的程序员为了追求效率，往往喜欢把表达式

```
A[I]＋1 / A[I];
```

写成

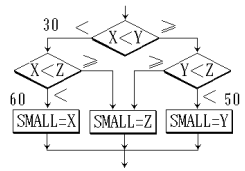
```
AI＝A[I];
X＝AI＋1 / AI;
```

这样将一句分成两句写，会产生意想不到的问题。

7. 让编译程序做简单的优化。
8. 尽可能使用库函数
9. 避免不必要的转移。同时如果能保持程序可读性，则不必用 **GO TO**语句。

例如，有一个求三个数中最小值的程序：


```
    IF ( X＜ Y ) GOTO 30
    IF (Y＜ Z) GOTO 50
    SMALL＝Z
    GOTO 70
30  IF ( X ＜ Z）GOTO 60
    SMALL＝Z
    GOTO 70
50  SMALL＝Y
    GOTO 70
60  SMALL＝X
70  CONTINUE
```



程序只需编写成：

```
SMALL＝X
IF ( Y .LT. SMALL ) SMALL＝Y
IF ( Z .LT. SMALL ) SMALL＝Z
```

所以程序应当简单，不必过于深奥，避免使用**GOTO**语句绕来绕去。

10. 尽量只采用三种基本的控制结构来编写程序。除顺序结构外，使用IF-THEN-ELSE来实现选择结构；使用DO-UNTIL或DO-WHILE来实现循环结构。

11. 避免使用空的ELSE语句和IF…THEN IF…的语句。这种结构容易使读者产生误解。例如，

```
IF ( CHAR＞＝'A' ) THEN
IF ( CHAR＜＝'Z' ) THEN
PRINT “This is a letter。”
ELSE
PRINT “This is not a letter。”
```

可能产生二义性问题。

12. 避免采用过于复杂的条件测试。
13. 尽量减少使用“否定”条件的条件语句。例如，如果在程序中出现

```
IF NOT ( ( CHAR＜‘0’ ) OR
( CHAR＞‘9’ ) ) THEN ……
```

改成

```
IF ( CHAR＞＝'0’ ) AND
( CHAR＜＝'9’ ) THEN ……
```

不要让读者绕弯子想。

14. 尽可能用通俗易懂的伪码来描述程序的流程，然后再翻译成必须使用的语言。
15. 数据结构要有利于程序的简化。
16. 要模块化，使模块功能尽可能单一化，模块间的耦合能够清晰可见。
17. 利用信息隐蔽，确保每一个模块的独立性。

18. 从数据出发去构造程序。
19. 不要修补不好的程序，要重新编写。也不要一味地追求代码的复用，要重新组织。
20. 对太大的程序，要分块编写、测试，然后再集成。
21. 对递归定义的数据结构尽量使用递归过程。

输入和输出

- 输入和输出信息是与用户的使用直接相关的。输入和输出的方式和格式应当尽可能方便用户的使用。一定要避免因设计不当给用户带来的麻烦。
- 因此，在软件需求分析阶段和设计阶段，就应基本确定输入和输出的风格。系统能否被用户接受，有时就取决于输入和输出的风格。

- 不论是批处理的输入／输出方式，还是交互式的输入／输出方式，在设计和程序编码时都应考虑下列原则：

1. 对所有的输入数据都要进行检验，识别错误的输入，以保证每个数据的有效性；
2. 检查输入项的各种重要组合的合理性，必要时报告输入状态信息；
3. 使得输入的步骤和操作尽可能简单，并保持简单的输入格式；

4. 输入数据时，应允许使用自由格式输入；
5. 应允许缺省值；
6. 输入一批数据时，最好使用输入结束标志，而不要由用户指定输入数据数目；
7. 在交互式输入输入时，要在屏幕上使用提示符明确提示交互输入的请求，指明可使用选择项的种类和取值范围。同时，在数据输入的过程中和输入结束时，也要在屏幕上给出状态信息；

8. 当程序设计语言对输入／输出格式有严格要求时，应保持输入格式与输入语句的要求的一致性；
9. 给所有的输出加注解，并设计输出报表格式。

输入／输出风格还受到许多其它因素的影响。如输入／输出设备（例如终端的类型，图形设备，数字化转换设备等）、用户的熟练程度、以及通信环境等。

程序效率

- 讨论效率的准则

程序的效率是指程序的执行速度及程序所需占用的内存的存储空间。程序编码是最后提高运行速度和节省存储的机会，因此在此阶段不能不考虑程序的效率。让我们首先明确讨论程序效率的几条准则

- ◆ 效率是一个性能要求，应当在需求分析阶段给出。软件效率以需求为准，不应以人力所及为准。
- ◆ 好的设计可以提高效率。
- ◆ 程序的效率与程序的简单性相关。
- ◆ 一般说来，任何对效率无重要改善，且对程序的简单性、可读性和正确性不利的程序设计方法都是不可取的。

算法对效率的影响

- 源程序的效率与详细设计阶段确定的算法的效率直接有关。在详细设计翻译转换成源程序代码后，算法效率反映为程序的执行速度和存储容量的要求。
- 设计向程序转换过程中的指导原则：

① 在编程序前，尽可能化简有关的算术表达式和逻辑表达式；

② 仔细检查算法中的嵌套的循环，尽可能将某些语句或表达式移到循环外面；

③ 尽量避免使用多维数组；

④ 尽量避免使用指针和复杂的表；

⑤ 采用“快速”的算术运算；

⑥ 不要混淆数据类型，避免在表达式中出现类型混杂；

⑦ 尽量采用整数算术表达式和布尔表达式；

⑧ 选用等效的高效率算法；

- 许多编译程序具有“优化”功能，可以自动生成高效率的目标代码。

影响存储器效率的因素

- 在大中型计算机系统中，存储限制不再是主要问题。在这种环境下，对内存采取基于操作系统的分页功能的虚拟存储管理。存储效率与操作系统的分页功能直接有关。

- 采用结构化程序设计，将程序功能合理分块，使每个模块或一组密切相关模块的程序体积大小与每页的容量相匹配，可减少页面调度，减少内外存交换，提高存储效率。

- 在微型计算机系统中，存储器的容量对软件设计和编码的制约很大。因此要选择可生成较短目标代码且存储压缩性能优良的编译程序，有时需采用汇编程序。
- 提高存储器效率的关键是程序的简单性。

影响输入／输出的因素

- 输入／输出可分为两种类型：
  - ◆ 面向人(操作员)的输入／输出
  - ◆ 面向设备的输入／输出
- 如果操作员能够十分方便、简单地录入输入数据，或者能够十分直观、一目了然地了解输出信息，则可以说面向人的输入／输出是高效的。

- 关于面向设备的输入/输出，可以提出一些提高输入/输出效率的指导原则：
  - ◆ 输入/输出的请求应当最小化；
  - ◆ 对于所有的输入/输出操作，安排适当的缓冲区，以减少频繁的信息交换。
  - ◆ 对辅助存储(例如磁盘)，选择尽可能简单的，可接受的存取方法；

- ◆ 对辅助存储的输入/输出，应当成块传送；
- ◆ 对终端或打印机的输入/输出，应考虑设备特性，尽可能改善输入/输出的质量和速度；
- ◆ 任何不易理解的，对改善输入/输出效果关系不大的措施都是不可取的；
- ◆ 任何不易理解的所谓“超高效”的输入/输出是毫无价值的；

程序复杂性度量

- 程序复杂性主要指模块内程序的复杂性。它直接关联到软件开发费用的多少，开发周期的长短和软件内部潜伏错误的多少。
- 减少程序复杂性，可提高软件的简单性和可理解性，并使软件开发费用减少，开发周期缩短，软件内部潜藏错误减少。

复杂性度量需要满足的假设

- 为了度量程序复杂性，要求：
  - ◆ 它可以用来计算任何一个程序的复杂性；
  - ◆ 对于不合理的程序，例如对于长度动态增长的程序，或者对于原则上无法排错的程序，不应当使用它进行复杂性计算；
  - ◆ 如果程序中指令条数、附加存储量、计算时间增多，不会减少程序的复杂性。

代码行度量法

- 源代码行数度量法基于两个前提：
  - ◆ 程序复杂性随着程序规模的增加不均衡地增长；
  - ◆ 控制程序规模的方法最好是采用分而治之的办法。将一个大程序分解成若干个简单的可理解的程序段。

- 方法的基本考虑是统计一个程序模块的源代码行数目，并以源代码行数做为程序复杂性的度量。
- 设每行代码的出错率为每**100**行源程序中可能有的错误数目。
- *Thayer*曾指出，程序出错率的估算范围是从0.04%～7%之间，即每100行源程序中可能存在0.04～7个错误。他还指出，每行代码的出错率与源程序行数之间不存在简单的线性关系。

- *Lipow*指出，对于小程序，每行代码出错率为**1.3%**～**1.8%**；对于大程序，每行代码的出错率增加到**2.7%**～**3.2%**之间，这只是考虑了程序的可执行部分，没有包括程序中的说明部分。
- *Lipow*及其他研究者得出一个结论：对于少于**100**个语句的小程序，源代码行数与出错率是线性相关的。随着程序的增大，出错率以非线性方式增长。

McCabe度量法

- **McCabe**度量法，又称环路复杂性度量，是一种基于程序控制流的复杂性度量方法。
- 它基于一个程序模块的程序图中环路的个数，因此计算它先要画出程序图。
- 程序图是退化的程序流程图。流程图中每个处理都退化成一个结点，流线变成连接不同结点的有向弧。

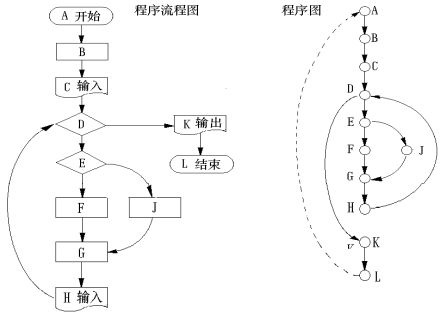
- 程序图仅描述程序内部的控制流程，完全不表现对数据的具体操作，以及分支和循环的具体条件。
- 计算环路复杂性的方法：根据图论，在一个强连通的有向图G中，环的个数由以下公式给出：

  $V(G)=m-n+p$

  其中，$V(G)$是有向图G中环路个数，m是图G中弧数，n是图G中结点数，p是图G中的强连通分量个数。

- *Myers*建议，对于复合判定，例如，(A＝0)∩(C＝D)∪(X＝'A') 算做三个判定。
- 为使图成为强连通图，从图的入口点到出口点加一条用虚线表示的有向边，使图成为强连通图。这样就可以使用上式计算环路复杂性。
- 在例示中，结点数n＝11，弧数m＝13，p＝1，则有

  $V(G)=m-n+p=13-11+1=3$.
- 等于程序图中弧所封闭的区域数。



几点说明

- 环路复杂度取决于程序控制结构的复杂度。当程序的分支数目或循环数目增加时其复杂度也增加。环路复杂度与程序中覆盖的路径条数有关。
- 环路复杂度是可加的。例如，模块A的复杂度为3，模块B的复杂度为4，则模块A与模块B的复杂度是7。

- *McCabe*建议，对于复杂度超过10的程序，应分成几个小程序，以减少程序中的错误。*Walsh*用实例证实了这个建议的正确性。在*McCabe*复杂度为10的附近，存在出错率的间断跃变。
- *McCabe*环路复杂度隐含的前提是：错误与程序的判定加上例行子程序的调用数目成正比。加工复杂性、数据结构、录入与打乱输入卡片的错误可以忽略不计。

- 这种度量的缺点是：
  - ◆ 对于不同种类的控制流的复杂性不能区分
  - ◆ 简单IF语句与循环语句的复杂性同等看待
  - ◆ 嵌套IF语句与简单CASE语句的复杂性是一样的
  - ◆ 模块间接口当成一个简单分支一样处理
  - ◆ 一个具有1000行的顺序程序与一行语句的复杂性相同

*Halstead*的软件科学

- ***Halstead***软件科学研究确定计算机软件开发中的一些定量规律，它采用以下一组基本的度量值。
- 这些度量值通常在程序产生之后得出，或者在设计完成之后估算出。

- 程序长度(预测的***Halstead***长度)

  令$n1$表示程序中不同运算符(包括保留字)的个数，令$n2$表示程序中不同运算对象的个数，令H表示"程序长度"，则有

  $H=n1*\log_2 n1+n2*\log_2 n2$
- 这里，H是程序长度的预测值，它不等于程序中语句个数。

- 在定义中，运算符包括：

  算术运算符　赋值符(=或:=)

  逻辑运算符　分界符(，或；或:)

  关系运算符　括号运算符

  子程序调用符　数组操作符

  循环操作符等。
- 特别地，成对的运算符，例如"**BEGIN…END**"、"**FOR…TO**"、"**REPEAT …UNTIL**"、"**WHILE…DO**"、"**IF…THEN…ELSE**"、"（…）"等都当做单一运算符。

- 运算对象包括变量名和常数。
- 实际的*Halstead*长度

  设$N1$为程序中实际出现的运算符总个数，$N2$为程序中实际出现的运算对象总个数，N为实际的*Halstead*长度，则有

  $N = N1 + N2$

- 程序的词汇表

  *Halstead*定义程序的词汇表为不同的运算符种类数$n1$和不同的运算对象种类数$n2$的总和。若令n为程序的词汇表，则有

  $n = n1+n2$

  例如，用***FORTRAN***语言写出的交换排序的例子

```
SUBROUTINE SORT ( X, N )
DIMENSION  X( N )
```



```
      IF ( N .LT. 2 ) RETURN
      DO  20  I=2, N
         DO  10  J=1, I
            IF ( X(I) .GE. X(J) ) GO TO 10
            SAVE = X(I)
            X(I) = X(J)
            X(J) = SAVE
   10       CONTINE
   20    CONTINE
         RETURN
         END
```


运算符	计数	运算对象	计数
可执行语句结束	7	X	6
数组下标	6	I	5
=	5	J	4
IF（ ）	2	N	2
DO	2	2	2
,	2	SAVE	2
程序结束	1	1	1
.LT.	1	$M1$=7	$N2$=22
.GE.	1		
GO TO 10	1		
$n1$=10	$n2$=28		

- 程序量

  程序量 V 可用下式得到

  $V = N * \log_2 n$

  它表明了程序在词汇上的复杂性。其最小值为

  $V^* = (2+n2^*) * \log_2(2+n2^*)V$

  这里，2表明程序中至少有两个运算符：赋值符 = 和函数调用符 f ( ) ，$n2^*$表示输入／输出变量个数。

- 对于上面的例子，利用$n1$，$N1$，$n2$，$N2$，可以计算得

  $H = 10 * \log_2 10+7 * \log_2 7 = 52.87$

  $N = 28+22 = 50$

  $V = (28+22) * \log_2(10+7) = 204$
- 等效的汇编语言程序的V＝328。这说明汇编语言比*FORTRAN*语言需要更多的信息量(以bit表示)。

- 程序量比率 (语言的抽象级别)

  $L = V^* / V$

  或 $L = (2 / n1)*(n2 / N2)$

  它表明了一个程序的最紧凑形式的程序量与实际程序量之比，反映了程序的效率。其倒数

  $D = 1 / L$

  表明了实现算法的困难程度。

- **程序员工作量**
  $E = V / L$
- **程序的潜在错误**
  *Halstead*度量可以用来预测程序中的错误。预测公式为
  $B = (N1+N2)*\log_2(n1+n2) / 3000$
  B为该程序的错误数。它表明程序中可能存在的差错 B 应与程序量V成正比。

- 例如，一个程序对75个数据库项共访问1300次，对150个运算符共使用了1200次，那么预测该程序的错误数：
  $B = (1200+1300)*\log_2(75+150)/3000 \approx 6.5$
  即预测该程序中可能包含6～7个错误

- **Halstead的重要结论**
- 程序的实际***Halstead***长度*N*可以由词汇表*n*算出。即使程序还未编制完成，也能预先算出程序的实际***Halstead***长度*N*， 虽然它没有明确指出程序中到底有多少个语句。这个结论非常有用。经过多次验证，预测的***Halstead***长度与实际的***Halstead***长度是非常接近的。

Halstead度量的缺点

- 没有区别自己编的程序与别人编的程序。这是与实际经验相违背的。这时应将外部调用乘上一个大于1的的常数***Kf*** (应在1～5之间，它与文档资料的清晰度有关)。
- 没有考虑非执行语句。补救办法：在统计*n*1、*n*2、*N*1、*N*2时，可以把非执行语句中出现的运算对象，运算符统计在内。

- 在允许混合运算的语言中，每种运算符与它的运算对象相关。
- 如果一种语言有整型、实型、双精度型三种不同类型的运算对象，则任何一种基本算术运算符(＋、－、×、／)实际上代表了$A_3^2 = 6$ 种运算符。在计算时应考虑这种因数据类型而引起差异的情况。

- 没有注意调用的深度。***Halstead*** 公式应当对调用子程序的不同深度区别对待。在计算嵌套调用的运算符和运算对象时，应乘上一个调用深度因子。这样可以增大嵌套调用时的错误预测率。
- 没有把不同类型的运算对象，运算符与不同的错误发生率联系起来，而是把它们同等看待。例如，对简单IF语句与WHILE语句就没有区别。

- 忽视了嵌套结构(嵌套的循环语句、嵌套IF语句、括号结构等)。一般地，运算符的嵌套序列，总比具有相同数量的运算符和运算对象的非嵌套序列要复杂得多。解决的办法是对嵌套结果乘上一个嵌套因子。