第八届

GUO JI JI SUAN JI HUI YI LUN WEN XUAN

国际计算机会议论文选

上海市电子学会计算技术专业委员会

上海科学技术文献出版社

第八届国际计算机会议论文选

上海市电子学会计算技术专业委员会编

上海科学技术文献出版社

第八届国际计算机会议论文选
上海市电子学会计算技术专业委员会编
*
上海科学技术文献出版社出版
(上海武康路2号)
新华书店上海发行所发行
江苏宜兴南漕印刷厂印刷
*
开本 787×1092 1/16 印张 7.5 字数 182,000
1983年2月第1版 1983年2月第1次印刷
印数: 1—3,650

书号: 15192·233 定价: 0.95元

《科技新书目》 40-269

前　言

本书选自第八届国际计算机会议的论文集。国际计算机会议是由国际信息处理联合会(IFIP)组织的，每三年召开一次，第八届会议于1980年10月先后在日本东京及澳大利亚墨尔本举行，共有三十余个国家，700多名代表参加，我国现已加入国际信息处理联合会，这是第一次派出代表团参加会议。

会议分十个组宣读论文163篇，这些组是理论、硬件及计算机结构、软件、数据库及信息系统，网络及通信、科学及工业应用，政府及商业应用，社会及经济应用，教育、日常生活应用。在计算机理论方面，对程序设计语言的语义化问题作为重点内容，计算机体系结构的重点是就地分布式结构，软件方面强调设计方法学的研究，计算机教育及应用在这次大会上受到特别重视，由于到会的都是各国计算机科技界的知名人士，因此，该论文选确能反映当前国际上计算机技术的发展动向。

我们根据国内目前情况选译了十二篇文章，其中计算机教育一篇，系统结构三篇，软件三篇，数据库一篇，计算机网络两篇，计算机应用两篇，以供国内从事计算机工作的同志们参考。

上海市电子学会计算技术专业委员会

目　　录

有关程序员的教育：程序的表示法、证明和开发

David Gries

本文讨论程序设计的当前状况。表达的见解是，为了从事有效的程序设计，现在大多数程序员所具有的数学修养是不够的。此外，形式逻辑（它常常被非正式地用来论证各种程序和规格说明）方面和程序设计理论方面的教育，对于增长程序员的能力或许会起很大的作用。这种教育可能使程序设计由一种技巧走向科学。列举的例子都是为此见解提供证据。

1. 引论：七十年代的观点

七十年代我们对程序设计的了解取得了显著的进步。在1968年举行了有关"软件工程"的NATO会议，会上学术界和工业界的人士讨论了软件危机，并承认怎么编程序——怎么讲授程序——实在是一件神秘的事情。由于认识到这种危机，从而鼓励了大量的研究工作。于是在七十年代出现了"结构程序设计"，很多人自称已把它付诸实践出现了程序正确性与如何证明程序正确性的思想，出现了象"主程序员组"这一类管理技术，出现了试图强制程序员更好地编写资料的编写资料工具和设备（如结构流程图、HIPO图），以及出现了新的程序设计表示法（如PASCAL和最近美国国防部的Ada）。

工业界的人士告诉我，今天的年青程序员所受的训练要比十年前同类人员所受的训练好得多，他们编写的程序一般也更加容易读（主要由于这些程序不再是一堆乱头发）。这使人们对现状比十年前的一般要感到满意。

但是，这个领域也还有令人悲观的一面。一般地说，该领域仍缺乏"职业化"（professinalism）；程序设计仍是一种无系统的技巧，而不是一种科学。并且似乎没有几个程序员对深入理解——程序设计以对之进行真正的研究有兴趣。这使我感到有点沮丧，因为我认为，在过去十年中打下的理论基础已经足以使程序设计变成更为有条不紊的职业化过程。

让我对为什么有这种想法稍加说明。多年前，T·霍尔（Tony Hoare）把用一种方法去组织人的思维，以便在合理的时间内，将计算任务用容易理解的方式表示出来这样一个任务称为"结构程序设计"。虽然这个定义没有使人们深入理解怎么样实施结构程序设计，但它确实简明地阐明了我们应该期望从结构程序设计得到什么，从而使我们有可能确定，程序员们是否在实践结构程序设计。答案是否定的，有三点理由。首先，尽管我们取得了许多进展，但是任务仍然不是在合理的时间内完成的，经常的费用超支和脱期就是证据。其次，最后的程序很少象它应当的那样可读和易懂（虽然有些进步）。第三，程序很少满足对它们的规格说明（specifications）；程序充满错误，其中有些错误即使存在多年也发现不了。

也许这是为什么现在整个程序设计工作的百分之七十以上是维护老程序（这意味着

只有百分之三十用于研制新程序），以及为什么这个百分比每年还在增加；我们并不懂得我们正在做的事情。

这里举一个具体例子说明普遍缺乏熟练程度的情形。1976 年，我不得不写一个正文行的右对齐算法——在字间按给定的方式插入空白，以使最后一个字能在某一列结束。我应用了我认为有效的技术（使用了正确性证明概念等）。由于结果相当好，我就考虑把它写出来让别人去读，而审稿人认为很有价值将它发表[2]。

作为一次试验，我曾请了四五十个大学生、教师和来自工业界的程序员去解决这个问题。明确地告诉他们，这是一次试验，首先关心的是正确性和清晰，其次是效率。尽管如此，竟有半数以上的人还是编写出不正确的程序！而我编的程序还比其他所有程序都稍有效些；由于我的这种程序设计风格，我作出了一个不同的解法。虽然这些程序中有许多程序错误较少，但是这些错误具有这样一个特性，即调试可能发现不了它们。尽管在一篇技术报告中描述该设计倒确实花了我不少时间，但产生我的正确程序所花费的时间并不比他们产生不正确程序所花的时间长。

正是这种甚至在编写小程序时也反映出来的，普遍地缺乏熟练程度使我感到忧虑。

在我们美国，现在有十万零五千多年龄在十八岁到二十一岁之间的大学生，正在把计算机科学作为他们的主修课。这是一个巨大的数字——是一个施加持久的好影响以便造就一批将进入工业界并立志变革的年青人的极好机会！可惜，大多数人在获得使用简单控制结构的初步知识时，相当多的人其关于算法的观念受到了 BASIC 和 FORTRAN 语言的不良影响，而受我认为非常重要的坚实的理论基础知识影响的人则太少了。

甚至在研究生这一级，也常常教授错误的观点。我最近跟一个年青的哲学博士交谈，他刚完成程序验证（verification）方面的毕业论文。我问他在他的系有多少学生专修那个领域。他答，选修的人很少，因为它太困难了。人们必须将流程图配上断言，而没有怎样做或为什么要这样做的深入了解，人们必须学习各种深奥的重点在模型和一致性的逻辑学等等，实在太困难了。我问，是否有一门课程讲授关于产生一个程序的同时作出正确性证明（后者实际上起主导作用）的概念。回答是没有，而且这个概念压根儿他就没有领教过！我讲给他听的一些程序编制方法引起了他的兴趣。

这就是一个已经花费三四年时间学习程序验证的计算机科学工作者，甚至还没有听到过有关他工作的常识和原则。我把程序正确性的证明看作是在编制一个程序时的职责，并且把它看作是对我编制程序起指导作用的东西，当然不是把它当作独立的和为它本身而需要进行研究的某种东西。如果德米洛（DeMillo）、利普顿（Lipton）和珀利斯（Perlis）谈及的是无意识的验证，那么我同意他们在《社会进程与定理和程序的证明》[3]一文中的观点。

够了！为了向大家说明整个事情还不是它本来应当成为的样子，我一直在谈此领域的不足之处。现在停止埋怨，开始谈谈它怎样才能成为应当的那样。

2. 程序设计能成为一种职业吗？

对我来说，程序设计是一项困难的智力挑战，它需要很好的数学修养。如同处理数学定理一样，证明程序的正确性是程序员的责任。一个定理，或一个程序是供别人使用的一种工具，但是为了使用它，必须使人们确信它的正确性。

也许我应该解释用“证明”这个字蕴含什么，因为有些人可能被这个字吓住了。根据《第三版韦氏新国际字典》，“证明”（a proof）

是"提供一种有力的证据使人们在思想上不得不接受一条真理或一件事实"。因此"证明"就是提供证据使读者相信陈述的真实性。这一定义并没有谈到用来写该证明的语言，也没有谈到证明必须是怎样的形式。

在这种意义上说，每个程序员都试图"证明"他程序的正确性，因为他总要提供一些论据使他自己相信程序的正确性。如果一个程序员编制的程序甚至经过大量调试之后仍充满错误，则说明他不能胜任这种工作。

十年前，我们对于程序正确性证明应当需要些什么，还没有从技术上来认识；那时我们正在探索达到那种认识的途径。因此，那时要求任何种类的证明都将是愚蠢的。

然而，从那时以来，我们已经走过了一段很长的路程；我们已经打下许多必要的理论基础。因此对于未来，我们是乐观的。我们正接近一个可以开始谈论程序设计科学，而不是程序设计技巧的阶段，此处科学和技巧这两个术语是按福勒(Fowler)和高尔(Gower)的《现代英语用法字典》给定的意思使用的：

科学这个术语已被引伸为表示一个实际工作部门，它依赖于各种原理的知识和这些原理的自觉应用；而另一方面，技巧被理解为只需要传统规则的知识和由习惯获得的技能。

这并不意味着程序设计将成为一种枯燥无味的简单工作。程序设计中将永远会有创造性和激励，它将永远需要技能和智力上的努力。但是，基于理论基础的原理的出现意味着，我们能把程序设计工作做得更好，更为重要的是，我们可以教授别人怎样把它做得更好。

在本报告中，我赞成对称为"小程序"(in-the-small)的程序设计[与此相反的是"大程序"(in-the-large)的程序设计]有个较好的认识和教育，该术语是由F·德里梅尔(Frank DeRemer)和H·克朗(Hans-Kron)在1975年杜撰的。赞成这点的一条理由是，我确实认为，这些方法可以"放大"到大型的程序系统上去，而且还有另外的理由。简单说来，我相信一个人在他能够有效地编制小程序前，是不可能(甚至不能期望)去有效地设计大型程序或程序系统的。

论据是由E·W·戴克斯特拉(Edsger W. Dijkstra)用极有说服力的方式提出的：假设一个程序有n个组成部分，每个组成部分其正确的概率为p。那么，整个程序正确的总概率P必定满足$P<p^n$。既然在任意一个相当规模的程序中n很大，那么要想程序正确，就要求p非常非常接近于1。

请停一会儿，再读读上面的论据。在过去十年里花大量时间研究了小程序的程序设计，已经使我信服这个论据。别人似乎能无视这个论据的含意，但是我不能无视它的含意。

当然，其他种类的教育和研究也是同样必要的。可以举出一些领域，在这些领域中的研究和教育工作对我们将是有帮助的，例如：

(1) 程序设计的表示法，即语言(如Ada)；

(2) 机器的辅助，如调试工具，验证程序；

(3) 管理技术。

但是普遍缺乏程序设计熟练程度(包括我自己和其它人)使我得出结论，要取得有意义进展的最好方法是提高技术能力，并且首先是在"小程序"方面的能力。要注意的是，(1)至(3)条只产生辅助工具，它们对程序员使用其智力工具是有帮助的，但是能完全弥补智力技能不足的辅助工具是没有的。

对这种情况的另一种说法是，提高技术能力是减少不健康程序数量的预防药。在某种程度上，程序中产生错误的原因是缺乏能力，因此提高能力会有助于防止错误。另一方面，象调试辅助工具这样的一些工具，目的在于能对已经染上的疾病进行治疗。尽管预

防和治疗都是必要的，但是从长远观点看，预防则更好。

至此，我已经笼统地讲了需要些什么：更多更好的教育以提高技术能力。现在让我更具体些。在后面几节，我要依次讨论下述内容：

(1) 谓词演算；
(2) 程序设计的理论；
(3) 寻求简明性；
(4) 使用合适的表示法。

3. 讲授谓词演算

程序员们常常处理关于程序变量的命题陈述。他们必须留意逻辑关系，必须从前提推出各种结论，等等。可是，很少向他们教授这种推理的基本原理；又没有给予他们必要的智力工具。他们通常学习布尔表达式，但是并未学会怎么推导它们。

每个程序员都需要上一门形式逻辑（命题演算和谓词演算）方面的基础课程，因为这些课程构成我们推理的基础，不管它是形式的或非形式的。

可惜，对于逻辑的讲授并不是从程序员的观点出发，而是从逻辑学家的观点出发。逻辑学家对逻辑的兴趣较多的是在于逻辑本身，而程序员的兴趣则在于把逻辑当作一种实际的日常工具使用。例如，他需要懂得规格说明如何能描述一组状态；他必须善长于把一个程序要做些什么（通常是模棱两可）的描述，翻译成用谓词演算写的形式规格说明；他应当是一个能熟练使用逻辑公式证明从前提得出结论的专家。一个程序员不需要太深的逻辑，只需要相当好地学习一些基础的逻辑教材。

为了领会后面将要加以解释的理论根据，尤其需要很好掌握谓词演算方面的健全知识。例如，程序设计方法中间的一个基本概念是关于语句 S 相对于谓词 R 的"最弱前提"$wp(S, R)$概念：谓词 R 描述这样一个最大的状态集，使得在这些状态中的任何一个状态开始的 S 的执行，都能保证在满足 R 的某个状态结束。要想透彻领会 wp，就需要知道谓词是怎么描述状态集合的知识。例如，使用这个概念，赋值语句被定义为：

$$wp(x:=e,\ R)=R_e^x$$

此处"="号右边的谓词可通过用 e 取代 R 中所有自由出现的 x 而取得的。例如，$wp("x:=x+1",\ x>0)=x+1>0$。这个规则产生出一个数学意义上的赋值，并可作实际应用。但是，除非一个人完全懂得正文置换，以及一个谓词与它所示状态集合之间的关系，否则是不能完全领会或使用这种简单的证明规则的。

这里是程序员应当能容易完成的推理类型的一个例子。这个例子不是取自程序设计领域，而是取自游戏 W E F'N 证明——«现逻辑游戏»。

迟到的公共汽车问题。设已知下列三个陈述：

(1) 如果比尔乘公共汽车，那么若公共汽车迟到，则比尔就要赶不上他的约会。

(2) 若(a)比尔赶不上他的约会，且(b)比尔感到沮丧，则比尔不应回家。

(3) 如果比尔未得到该工作，那么(a)比尔感到沮丧，且(b)比尔应回家。

假设这些陈述是真实的，那么下列哪些陈述是有效的*？给出有效陈述的证明，并给出无效陈述的反例。

(1) 若公共汽车迟到，则(a)比尔未乘公共汽车，或者(b)若比尔未得到工作，则比尔没有赶不上他的约会。

(2) 若比尔得到该工作，则(a)比尔不感到沮丧，或者(b)比尔不应回家。

(3) 若(a)比尔应回家，且比尔乘公共汽车，则若公共汽车迟到则(b)比尔并不感到沮丧。

在解决这样的一些难题中，部分困难在

于英文：其中有些句子的分解是困难的。理解英文写的规格说明，并把这些说明翻译成更正规的形式时，程序员们遇到类似的困难。此外，程序员一旦理解了说明，就需要知道怎么有效地推敲这个说明——如何说明从前提得出的结论。

可惜，1978年制订的计算机科学课程[4]并未包括逻辑课，虽然数学中的七门其他课程因为与计算机科学"有关"而被包括在其中。但我希望下次修改会更好些。

4. 讲授程序设计理论

谈及程序设计活动本身，我相信着重点应该放在构造正确性程度极高的程序方面。我认为这是能做到的，只要我们原谅我们往往容易犯的典型语法错误——因为我们毕竟是人。不过，构造正确性程度极高的程序要求我们对于以编制程序和给出其证明同时进行的编程过程为基础的程序编制原理，有一定的知识和经验。这就要求我们具有比现在许多程序员所具有的程度更高的数学修养。

我非常欣赏E·W·戴克斯特拉关于程序设计的思想方法。我认为他的书[5]反映了我们对程序设计的理解方面的重大进步。这里我想说明他的两个主要观点，目的在于使大家相信，程序设计的这种训练方法是有些道理的。还请记住，我只能使用很小的例子说明几个基本要点。许多人在看了类似的介绍时怀疑说，"在那个例子上我明白了它怎么工作，但是这能推广吗?"或者，他们怀疑，这个材料对"一般水平的程序员"来说学起来是否太困难了。

对于第一个问题，我的回答是肯定的。除此之外，对这个方法的学习和实践，还会产生一个有关程序设计过程的新见解，它将提高你对程序设计的理解和熟练程度，其作用是难以估量的。它影响了我的教学；我感到对基本原理有了较好的掌握，即使我在一门引论性的课程中还没有讲授这种理论的所有形式化的东西，这个理论就为我提供了介绍某些概念和观点的根据。

对于第二个问题，我必须回答我的工作是考察程序设计的过程，如果通过考察表明了这样的事实，即某些技术材料确实是必要的，表明程序设计本质上是困难的，那么我不能不如实地告诉你们。你们可以出于政治或社会原因的考虑而决定拒绝接受我的技术观点，但请你们务必认识到你们这些考虑的性质。

4.1 "最弱前提和反推"

正如前面提到的，在Dijkstra的著作中，一个关键重点是最弱前提 $wp(S, R)$ 概念。以前曾使用 $\{P\}S\{R\}$ 作为程序 S 的规格说明，意思是：如果 S 的执行从满足断言 P 的一个状态下开始，那么它的执行将在满足 R 的一个状态下终止。这个表示法等价于 $P \Rightarrow wp(S, R)$。

已知规格说明 $\{P\}S\{R\}$，要求推导出 S。wp 的定义使我们企图主要依据于 R 来推出 S，并在这同时推导出 $wp(S, R)$；在完成推导时，我们必须证明 $P \Rightarrow wp(S, R)$。

初看起来，这似乎弄反了；按老的习惯我们认为，应以前提为基础来编制一个程序，因为程序是从头至尾执行的。但是，程序设计是一项"面向目的"的活动，它更为着重于后置条件 R，而不是着重于前提 P。用一个例子可以说明这一点。

假定编写一个程序（程序段）将 x 和 y 两个变量中的最大者存入变量 z。于是，我们要编写一个程序 S，使其满足 $\{P:\text{true}\}S\{R:z=\max(x, y)\}$。后置条件可用下面的形式来表示：

$$R: z \geqslant x \wedge z \geqslant y \wedge (z = x \vee z = y)$$

我们想仅从 R 导出 S：如何对 z 赋值才能满足 R？通过执行 $z:=x$ 能满足 R 的一部分，即 $z=x$，但这只是在某些条件下才能使 R 成立。为了确定那些条件，我们导出

$$wp(\text{“}z:=x\text{”},\ R)=R_x^z$$
$$\equiv x\geqslant x\wedge x\geqslant y\wedge(x=x\vee x=y)$$
$$\equiv x\geqslant y$$

因此，当且仅当一开始 $x\geqslant y$ 时，执行 $z:=x$ 才满足 R。类似地，我们看到，当且仅当 $y\geqslant x$ 时，$z:=y$ 的执行才满足 R。这就引导我们用 E. W. 戴克斯特拉的表示法构造下述选择语句：

if $x\geqslant y\rightarrow z:=x$
□ $y\geqslant x\rightarrow z:=y$
fi

形式的和非形式的两种推理都使我们可以得出如下结论：该程序执行总是能使 R 成立，因此前提是所要求的前提：真。

已知 R，我们编制一个语句 S，然后发现所需要的前提为真，可以被满足。现在要求读者做相反的推理：

只根据前提 P（=真）推导 S；
完成推导后，再证明满足 P 为真的 S 的执行使 R 成立。

显然，用这种方式得出正确语句 S 的可能性是很小的。

程序设计是一项面向目的的活动这个观点，在我的简短发言中不能做更多的解释，但是我希望能使大家相信，这是有些道理的。

4.2 循环不变式

在程序设计的这种训练方法中，另一关键点是涉及循环不变式。[B. 费洛伊德(Bob Floyd)在 1967 年，以及 T. 霍尔在 1969 年的早期开创性工作中，不变式也是一个极为重要的概念。]

让我们分析循环 **do** $B\rightarrow S$ **od**。我所以要使用 E. W. 戴克斯特拉的表示法，是因为到目前为止我认为这是最简单、最简明，也是最灵活的一种表示法。此循环等价于 PL/I 语言中的循环 DO WHILE(B); S END; 关于此循环，我们已证明下列定理。

定理　令 P 为一个满足 $P\Rightarrow wp(S,\ P)$ 的谓词。

令 t 为满足 $((P\wedge B)\Rightarrow t>0$ 和
$(P\wedge B)\Rightarrow wp(\text{“}T:=t;s\text{”},\ t<T)$
的程序变量的一个整数函数，此处 T 是一个新的程序变量。那么
$P\Rightarrow wp(\text{“}$**do** $B\rightarrow S$ **od**$\text{”},\ P\wedge\neg B)$。

第一个假设 $P\Rightarrow wp(S,\ P)$，说明循环体的执行使 P 为真仍成立。因此 P 叫做此循环的一个不变关系式。随后的两个假设确定了整函数 t 的性质，可把 t 看作仍要执行的迭代次数的上界。于是，这些假设表明循环的执行将会终止。其中的一个假设表明，只要另一次迭代是必要的，那么仍要执行的迭代次数的界就要大于零；另一假设表明，该循环体的执行使迭代次数的界至少减一。

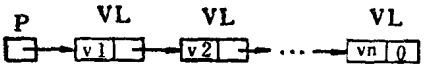
这似乎有些困难吧？起初可能是这样。不过，这是程序设计方面最振奋人心的进展之一，因为它教会了我们怎么理解循环，而循环正是程序设计的最困难的部分之一。此外，它已为教别人理解循环提供了一种办法。对这种形式进展的作用，怎么估计也是不会过高的。

4.3 编制循环的一个例子

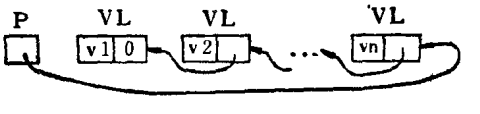
考虑一个链接表反向的问题。两个数组 $V[1:100]$（用于存放值）和 $L[1:100]$（用于放链），以及一个变量 p 用来对某个 n 实现值$(v1,\ \cdots,\ vn)$的链接表（这儿 n 满足 $0\leqslant n\leqslant 100$）：

$v1=V[p]$，$v2=V[L[p]]$，
$v3=V[L[L[p]]]=V[L^2[p]]$，$\cdots$
$vn=V[L^{n-1}[p]]$，$L^n[p]=0$。

用图来表示，我们得到

(1)


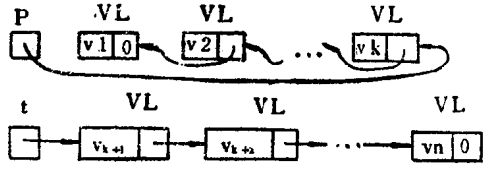
为了使 p 表示表$(vn,\ \cdots,\ v1)$，要把此链接表反向。这样，到终止时应得到

(2)


没有其他数组可用。不应改变数组 V；只可改变数组 L 和变量 p。

因为对表的长度没有限制，看来似乎需要一个循环。而且，编制一个循环，其中每迭代一次改变一个 L 域，看来是合理的。最初，这种不变式必须"看起来象"初始条件，到终止时必须看起来象最后结果。因此，我们寻找一个不变式，初始条件和结果为这个不变式的两个例子。记住这一点，用一个非形式图来表示这个不变式就十分容易了：

(3)



于是，p 表示已被反向过的表的部分；t 表示没有反向过的那部分。然后可以很容易地写出一个算法：

$t:=p$; $p:=0$; （已建立不变式）

do $t\neq 0\rightarrow p, t, L[t]:=t, L[t], p$ **od**

定理中的终止函数 t 这儿是由变量 t 所表示的表中元素的个数。

那不是一个简洁的小程序？我看到过一些程序员对这个小而平常的问题辛苦了半个多小时，仍不知他们的结果是否正确。若先写出不变式，则可使隐藏在循环里面的思想明朗化，具体化，从而几乎能立即推出这个程序。

我得承认在编制这个算法时犯过两个错误。第一，大多数语言没有上面使用的多重赋值语句。因而我必须"串行化"——将一个多重赋值语句写成一串简单赋值语句——而在这样做时，我总是要犯错误。多重赋值表示一种简单的状态改变，它不是只改变一个变量，而是改变一组变量，这在我的思考方法中是相当基本的一种语句，因此，我认为所有的程序设计语言都应包括多重赋值。即使语言中没有多重赋值，至少在程序设计的课程中应讲授多重赋值这个概念。

第二，使用上面那样的图形是很好的，因为与一个复杂的谓词演算语句相比，它能使我们更快地理解所表达的思想。但是，因为要在看起来好象一个个别例子的图中传递全部信息是十分困难的，因此用图也很容易带来错误或低效的结果。为了说明问题，我起初按图写出的循环为：

if $p\neq 0\rightarrow t:=L[p]$; $L[p]:=0$;

do $t\neq 0\rightarrow p, t, L[p]:=t,$

$L[t], p$ **od**

□ $p=0\rightarrow$skip

fi

这是由于不变式的图没有提供 $p=0$（p 表示一个空表时）这一情况的线索，因而我认为 p 至少应当有一个元素。科内尔的一个研究生 G· 莱文(Gary Levin)指出了我的错误。令 $t^{\rightarrow}$ 表示其表头指引元为 t 的链接表中值的表，$t^{\leftarrow}$ 表示同一表的反向。那么输入规格说明可写作 $p^{\rightarrow}=(v1, \cdots, vn)$，所要的结果可写作 $p^{\leftarrow}=(v1, \cdots, vn)$，然后很容易看出一个好的不变式是

$$p^{\leftarrow} o t^{\rightarrow}=(v1, \cdots, vn)。$$

此处 o 表示两个表的连接。由此很容易看到"$t:=p$; $p:=0$"是正确的初始化，从而 $p=0$ 的情况不必单独处理。

4.4 第二个例子

让我们来考虑另一个例子，它很清楚地说明了在程序设计方面的研究是如何教会我们对迭代过程进行推理的。假设有一只瓮，里面装着一些黑球和白球。瓮外也有足够的球，从而能做下面的游戏。通过重复必要次数的下述操作，使瓮中球的数目减少到一。

> 从瓮中任取两只球。若两只球的颜色相同，则将这两只球扔掉，而另取一只黑球放入瓮中；若两只球颜色不同，则将白球放回瓮内，而将黑球扔掉。

每"执行"一次上述操作，瓮中球的数目减少一；当仅剩下一只球时，游戏结束。现在的问题是，瓮中最后那只球的颜色跟开始时

瓮中黑球和白球的数目有什么关系(如果有关系的话)?

你们可能立即会依据测试情况来考虑这个问题:如果原先有两只白球会发生什么情况?两只黑球又会发生什么情况?一黑一白呢?两黑一白呢?注意,这样的思考类似于根据测试情况进行程序设计:根据某些测试情况编写程序;寻找更多的测试情况并修改程序,如此等等。经验证明,这种工作方法就是不好。首先,人们很快地产生如此多的测试情况而使结果含混不清。其次,为了应付这些新情况而对程序连续不断的修补,使程序复杂而不清晰。

让我们来试试寻找一个更为有效的解决办法。注意这个游戏是一种迭代过程,并且瓮中的球应当有一些迭代之前和迭代之后均保持的性质,如果你需要的话,这就是循环不变式。上述过程结束后,从这个循环不变式加上最后瓮中只有一只球这个事实,我们就能知道最后那只球的颜色。既然最后只有一只球,而且"1"是个奇数,这就使我们考虑是否黑球(或白球)数目的奇偶性保持不变;奇偶性毕竟是个简单的特性,简单性正是我们始终应努力争取的东西。看一看这个过程就知道,黑球的奇偶性不是个不变式。但是,白球的奇偶性是不变式这一点是一眼就能确定的:白球数每次减二,或根本不变。因此,当且仅当白球的原始数目是奇数时,这最后一个球才是白球。

这个小问题,我自己和其他一些人只用不到十分钟的时间就解决了。然而,我看到其他一些人苦思冥想了半个多小时仍未解决,直到暗示他们寻找一个不变式之后,五分多钟答案就出来了。很清楚,在我们的工具袋中装有不变式这个概念(即寻找在整个过程中都成立的特性)是极为重要的。不过,只有在人们实际使用过这个概念,并有意识地努力应用这个概念时,它才会发挥效力。

往往有人会说,推导不变式,特别是要在编写如一个循环之前推导出一个不变式,太困难了。开始当然是这样,不过一旦人们取得了经验,它就会变得越来越容易,最后就成为一种习惯。可以很早就向新程序员介绍这种概念,而不介绍任何形式系统的东西,犹如下面所做的那样。我告诉我的学生,要按变量的逻辑关系,而不是按其类型对变量说明进行分组。另外,我告诉他们,要先定义变量,然后再编写使用这些变量的语句。此处我使用"定义"这个字意思是"描述他们的逻辑关系"。例如,若数组 T 要包含表示温度的一列数,而 K 是这列数的"计数器"(应当从我们的词汇表中去掉这些不明确的字,因为它们引起不精确性),那么他们应当首先写下语句"$T[1{:}K]$是到此为止所建立的表示温度的表"。一旦这样写了,人们就会立即明白初始化 $K:=0$ 是需要的,并且每当人们必须编写处理 T 或 K 的程序语句时,都可以引用那个定义。该理论在这里确实帮了我不少忙,因为我就把该定义视为一个几乎在程序的所有地方都必须成立的不变式。这就清楚表明,此理论怎样能有助于提高理解能力和怎么可以把这种理论以非形式的方式投入实际使用。

5. 寻求简单性

数学家和计算机科学家们常常面临着如何用别人能懂得的方法来表达他们工作的问题,这种表达还要求相当程度的形式化,以使别人能看出他们的工作是完备而正确的。数学家必须在形式和细节这一方面与直观这另一方面之间找到适当的平衡,他必须寻找最简单的方法来确切地表达什么是必需要别人理解的东西。在这方面,程序员甚至碰到更为艰巨的问题,因为程序必然包含大量的细节。因此,寻找适当种类和数量的细节——不多,也不少——这点对程序员来说甚至更加重要,以使读者(以及程序员自己)能尽快地、尽

量容易地懂得和评价他的程序。程序员要寻找适当的表示法和组织形式，以便使复杂性容易处理——或甚至确保复杂性永不出现。

这是如此重要，以致我喜欢把程序设计的这个领域叫做计算的简单性，以与计算机科学方面的一个叫做计算的复杂性的既存领域成对立面。

有关平衡和简单性的需要几乎很少讲授，有些人会争辩说这是不可能讲授的，因为一个人只有通过自己的经验和与别人的交往才能掌握它，否则是不能掌握的。尽管如此，还是应对学生再三强调这种需要，并举例说明之。学生可以从对程序的一些详细的反馈意见得到好处，这种反馈意见可通过加注的方式给出，以说明如何一项稍微不同的技术、一个比较精确的注解、或一种更合适的表示法会有所帮助。这是花费教师的时间和精力的，因为这是不能用程序自动评分装置来做的(按我的意见应当扔掉这类装置)，不过这是值得的。当然，问题在于并不是所有的程序设计教师都懂得不断寻求简单性这种需要。

5.1 例子：取幂

考虑一个老问题，即 b^c 计算问题(b, $c\geqslant 0$, $0^0=1$)。懂得取幂和二进制运算的程序员会注意到

$$b^{13}=b^8b^4b^1,$$

并注意到 $13=(1101)_2$。然后，对于 b^c，他可能得出一个以 c 的二进制表示法为基础的计算公式：

$$c=(c_{n-1}\cdots c_1c_0)_2$$

$$b^c=\prod_{i=0}^{n-1}(\text{if } c_i=1 \text{ then } b^{2**i} \text{ else } 1)$$

此处，$2**i$ 为 2^i，每个 c_i 为 0 或 1。因此，看来似乎合理的是，考虑使用一个带有从 0 变到 n 的"计数器" K 的循环；最初 $z=1$，每次迭代时，将第 k 个因子(若 $c_k=1$ 则 b^{2**k} 否则 1)跟 z 相乘。

考虑到效率，引进一变量 x 以保留 b^{2**k} 值，引进一变量 y 以保留 y 的二进制表示中的尚未"处理"的部分。这样就得到下面的循环不变式 P：

$$P\colon\ 0\leqslant k\leqslant n,$$

$$z=\prod_{i=0}^{k-1}(\text{if } c_i=1 \text{ then } b^{2**i} \text{ else } 1)$$

$$y=(c_{n-1}\cdots c_k)_2$$

$$x=b^{2**k}$$

现在人们可以编制下面的循环：

```
x, y, z, k: = b, c, 1, 0;
do k≠n→
    if even(y)→skip
    □ odd(y)→z: = z*x
    fi;
    k, x, y: = k+1, x*x, floor(y/2)
od
```

不过，让我们回过头来看看是否一切都是尽可能地清楚了。注意，我们不得不引用 c 的二进位数字。此外，这个不变式包括各个项的复杂乘积，它可能是一个主要的绊脚石。它是必需的吗？能否简化？

从某种意义上说，z 含有最后答案的部分：对于某个 Z，$z*Z=b^c$，而 Z 是关于 i 的各个项的乘积(若 $c_i=1$ 则 b^{2**i} 否则 1)，i 在 k 和之间 $n-1$。通过某种简单公式变换，我们可以得到这样简洁的事实：

$$Z=x^y$$

因此，$z*x^y=b^c$，并且不变式 p 可以改写成如下形式，而不必涉及 n 或 k：

$$p1\colon y\geqslant 0\wedge z*x^y=b^c$$

使用 $p1$ 作为不变式可将算法改编如下：

```
z, x, y: = 1, b, c;
do y≠0→if even(y)→y, x: = y/2, x*x
        □odd(y)→y, z: = y-1, z*x
        fi
od
```

这就是众所周知的对数(关于 c)算法的通常表示形式。重新组织一下，减少不必要的测试，可使算法更有效些。我试图说明的是，一个典型的程序员，假使他懂得使用不变

式和形式正确性概念，并且假使他能坚韧不拔地寻求简单性，并在形式化与常识两者之间寻求良好的平衡，他就必然能把它推导出来。如果要理解最后这个算法，读者也必须懂得正确性这个概念。

注意，没有给出关于这个算法在所有细节上的正确性之形式证明，这样做反而会带来模糊。相反，只给出有助于理解所必需的形式化部分——特别是给出了循环不变式，而把其他看来很明显或很容易的部分留给读者。

必须坚韧不拔地寻求简单性、适当数量的细节，以及形式化与直观两者之间的平衡。程序员每走一步，都必须问问自己：我找到表达事物的正确方法了没有？随之而来的还有对于风格和优美——这是甚至很少谈及的另一个问题——的需要。优美是极重要的；只有当我们不断努力寻找最简洁的，最简单的论据(argument)或表示法，我们才能期望搞好程序设计。必须不断地向学生指出优美的程序和论证的例子。

6. 使用合适的表示法

在引论中，我对BASIC和FORTRAN做了相当刻薄的评论。让我作点解释以便多少缓和一下这个评论。三十年来，我们一直在为寻找表达各种算法的合适表示法而奋斗。我们想要在计算机上执行我们的程序，这就意味着必须实现有关的表示法。这个事实增强了我们的这种奋斗。实现一种表示法要花很大的力量，因此当一个新的、也许更好的表示法出现时，我们完全会由于惰性而拒绝使用它。困难不仅仅在于要改变我们的思考习惯，而且还在于要改变我们所有的旧程序。所以我(几乎)能理解对于改变的阻力。

在历史上，一直存在有关表示法的斗争，在这方面，现在跟过去没有什么差别。例如，在十七世纪和十八世纪，用下面的记号代表等号：

$$= \quad \infty \quad || \quad | \quad 2|2 \quad) = (\quad [。$$

十八世纪初，只剩下了两个记号，R·雷科德(Robert Recorde)的"="和R·德斯卡特斯(René Descartes)的"∞"，彼此经历了一场旷日持久的斗争，"="号最后获胜。从这个观点看，在一些语言中把这个普遍采用的记号"="用于其他目的(即赋值)是很不好的。[在关于表示法历史的一本书[6]中，卡乔里(Cajori)提到，∞可能是侧着的金牛座符号。另外，听了我有关这个课题的发言之后，W·特希(Wlad Turski)猜测"="来自代表平分点(春分、秋分时，昼与夜相等)的天文记号"≏"。因为在春分点期间的某个时刻，太阳位于金牛座，那么"="和"∞"可能具有相同的起源！]

人们也记得十七世纪时导数的两种表示法之间的斗争：牛顿用$\dot{y}$，而莱布尼兹(Leibnitz)用dy/dx，后来莱布尼兹的表示法赢得胜利。

最后，在那些年代里数学上出现了一种截然相反的有趣现象(Ken Iverson)，一个叫W·奥特里德(William Oughtred)的人使用了一百五十多个数学符号，而在这同时R·西蒙(Robert Simon)仍坚持只使用自然语言进行证明和讲解，在1756年他编辑的欧几里德教本中，没有使用一个数学符号。

卡乔里在他的书中得出结论说，有关符号表示(即表示法)的整个斗争，对数学家来说，是一种很好的殷鉴，就是说没有一样东西是绝对好或绝对不好的。在适当的时候适度地使用符号，就会恰到好处。

这恰恰是我们在前一节所讨论的问题：适当的平衡是需要的。

因此，当人们使用FORTRAN和BASIC时，我是不会感到过分不安的，我只是为他们惋惜而已。

但是，确实使我不安的是，听说有些程序员只知道FORTRAN或BASIC语言。B·

沃夫(Benjamin Whorf)说,语言影响思想和意识。程序设计语言(表示法)起着相同的作用。一个人如果只知道FORTRAN或BASIC,他对程序设计的思考就会受到严重的妨碍,因为他必须把他所有的算法思想放进一种表示法,而且是一种不合适的表示法。

业已证明,有些程序就是不能用FORTRAN的do-loop编写,因此只知道用那样一种表示法编写循环的人们,就会处于严重不利的条件。最近有个人告诉我,他把一个局限于只知道FORTRAN的人比做一个一直被关在一个黑暗的小房子里的孩子。

同样明显的是,有些表示法比其他表示法粗陋,有些表示法表达我们的意图要比其他一些表示法好,而有些表示法比其他表示法更易出错。

因而,对专业人员来说,应当学会几种表示法,并应根据技术考虑,而不只是根据熟悉与否选择他想用于工作的一种(或数种)表示法。程序员有时可能被迫使用某种表示法,但是,那并不意味着他必须用它来进行思考。正如我多年前所说,人们设计程序是把设计转换到某种语言,而不是用那种语言来进行程序的设计。

7. 总　结

上面是我发言的主要部分,现在来总结一下。我讨论了程序员应当掌握的两个主要的技术领域:逻辑和程序设计理论。还讨论了一些比较难于理解的,但应当牢牢掌握的概念:对简单性的需要、对优美的需要、对形式和直观两者之间平衡的需要和不断寻找合适表示法的需要。尽管我不能担保,但是我的观点是,一个程序员只要能认真地接受我的劝告,那么他一定会提高他的程序设计效率。

不过,要对我的发言加一点说明,以免引起误解;学习程序设计是一件艰苦的工作。这是不可能一蹴而就的。它需要研究和实践。多年来由于计算机所造成的,我们的一些原来行得通的老的思考习惯,必须在某种程度上加以克服。此外,仅知道技术资料是不够的,必须有意识地去努力应用它。许多资料初看起来似乎相当简单,但是只有当你努力应用它,它才能得到应用。因此,我喜欢下面的格言——我不知道这是谁说的:

> 永远不要把一个基本原理看作是显而易见的东西,因为只有通过有意识地应用这种原理,你才会取得成功。

不过,从长远的观点考虑,我认为人们为了学习这样的资料而化些必要的工夫是完全值得的。

参考文献6篇(略)

刘先德　蔡林希译

朱三元校

计算机硬件结构的主要倾向

Alice Recoque

本文用由顶向下的方法来阐明计算机硬件结构的主要倾向。本文的第一部份提出以速度、使用方便、可用性、可适性为电子数据处理机/最终用户鉴定系统优劣的项目，然后用系统结构来说明这些项目的特性定义，描述系统结构采用诸如多道处理、速度的效率、寻址形式、指令码进展、平衡输入/输出等概念。根据硬件和软件的当前水平和今后的进展来说明这些特性的实现。叙述了冯·诺依曼结构(串行语言、可编址的存贮器等等)对目前技术水平的影响。另一方面也分析了引脚-门比率、磁盘访问时间和新存贮器技术的影响。本文还简要地描述了如本地分布、数据库、高平行性的结构作为综合用户需要和工艺手段间的例子。

1. 序　　言

本文并不想全面描述计算机结构主要的先进技术方案。部份原因是由于篇幅有限，而更重要的是我们对如何判断"分布结构"、"数据库处理机"和其它目前试图发展的结构的优劣更感兴趣。

所以本文首先从最终用户观点出发来分析如何得到一个"好的系统"，然后把这些准则变为与结构有关的特征，以便从工艺进展观点来考虑如何作进一步改进以得到最终的结构。

最后给出与硬件工艺发展和用户应用相适应的一些结构。这些结构在不远的将来是必需或者可以做到的。

2. 好系统的准则

首先必需作出与当前所需改进项目有关的定义，然后把它归结为与结构有关的项目。很明显好系统准则应从最终用户的观点出发来概括，因为只有他们才是最后的判断者。

粗略地说，除要求价格尽可能低外，用户希望系统具有：快速、使用方便、可用性好、容易适应今后发展的需要。

以下将说明这些准则对设计人员的要求。

2.1 系统必需快速

对于最终点用户，速度可表示为：

① 开始工作和结束工作之间化费的时间；

② 交互式处理的响应时间。

有大量的参数与这个题目有关，但它们之间的相互关系却不易搞清。

(1) 处理机的内部速度通常用时钟时间来表示。这个参数表示处理机内部处理过程的最终数量，而这个参数则与工艺、组装、一个时钟周期内依次执行的逻辑操作数有关。

(2) 内部速度的效率是指在同一脉冲期间所平行访问的门数目，更重要的是在这个时间中起作用的门。

重要的是将这一概念进一步引伸，我们可以设想一台最简单的计算机，例如一台微型计算机。

所有微指令均以纯串行执行，因此处理机所有的门可能在同一时钟时间中都起作用。