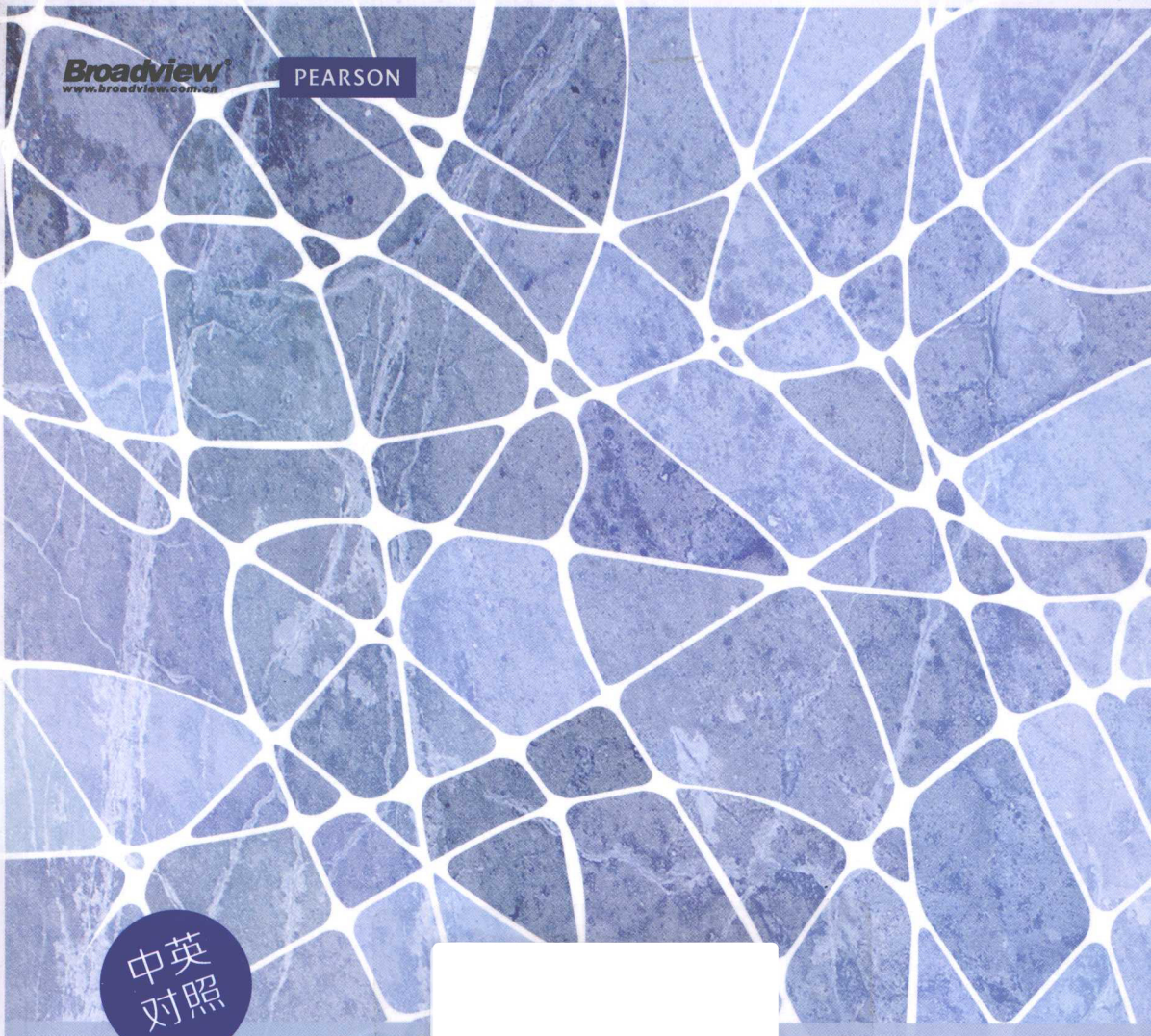
Broadview
www.broadview.com.cn
PEARSON



中英对照

编程的修炼

A Discipline of Programming

[荷] Edsger W.Dijkstra 著
裘宗燕 译

图灵奖获得者Edsger W.Dijkstra是每个在计算机领域学习和工作的人都应该了解和尊重的先驱者，本书为他最重要的著述，堪称编程领域里经典著作中的经典！

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

编程的修炼

A Discipline of Programming

[荷] Edsger W.Dijkstra 著
裘宗燕 译

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书是图灵奖获得者 Edsger W. Dijkstra 在编程领域里的经典著作中的经典。作者基于其敏锐的洞察力和长期的实际编程经验，对基本顺序程序的描述和开发中的许多关键问题做了独到的总结和开发。书中讨论了顺序程序的本质特征、程序描述和对程序行为（正确性）的推理，并通过一系列从简单到复杂的程序的思考和开发范例，阐释了基于严格的逻辑推理开发正确可靠程序的过程。

本书写于 20 世纪 70 年代中后期，但其对编程技术领域的开发、编程语言发展和程序理论研究的深刻影响持续至今。本书值得每个关注计算机科学技术的本质，冀求在程序和软件领域有长远发展的计算机工作者、教师和学生阅读。

版权贸易合同登记号　图字：01-2013-1792

图书在版编目（CIP）数据

编程的修炼：汉英对照 /（荷）戴克斯特拉 (Dijkstra,E.W.) 著；裘宗燕译. —北京：电子工业出版社，2013.7
书名原文：A discipline of programming
ISBN 978-7-121-20250-6

Ⅰ. ①编…　Ⅱ. ①戴… ②裘…　Ⅲ. ①程序设计—汉、英　Ⅳ. ①TP311.1

中国版本图书馆 CIP 数据核字（2013）第 085907 号

策划编辑：符隆美
责任编辑：李利健
印　　刷：三河市鑫金马印装有限公司
装　　订：三河市鑫金马印装有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编 100036
开　　本：720×1000　1/16　印张：28.75　字数：598 千字
印　　次：2013 年 7 月第 1 次印刷
印　　数：4000 册　　定价：79.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

译者序

图灵奖获得者 Edsger W. Dijkstra（1930.5.11—2002.8.6）是每个在计算机领域中学习和工作的人都应该了解和尊重的先驱者，其思想和技术遗产散布在计算机科学技术的众多领域。刚开始学习编程的人们都听过结构化编程的基本思想，在专业学习的早期就会遇到查找图中最短路径的 Dijkstra 算法，在学习并发程序技术时更是无处不见 Dijkstra 的思想和技术贡献。

Dijkstra 是 20 世纪 70 年代前后程序理论研究领域最重要的开拓者之一，也是那个时代的技术英雄。1968 年，Dijkstra 给 CACM 的一篇投稿引起结构化编程的倡导者们与维护 goto 的保守人士的大辩论，最终成就了结构化编程的革命，这是软件开发从个人技艺走向技术和科学的伟大的第一步。Dijkstra 与 Tony Hoare 和 Ole-Johan Dahl 的专著 *Structured Programming* 以其三位作者分别获得图灵奖而永远不可能被超越。其中，Dijkstra 撰写的“Notes on Structured Programming”提出了软件设计的许多指导原则，其深刻影响覆盖了从基础计算机教育直到复杂软件系统设计的广泛领域。Dijkstra 的另一重大贡献是作为最主要奠基人参与了并发程序理论和技术基础的开发。他基于自己的并发编程实践提出了许多概念，开发了许多技术。这里的许多术语出自他的思考，包括互斥、同步、竞态条件、死锁、非确定性、自稳定等。他提出的信号量、P/V 操作、临界区等是今天所有并发机制的基础。他还留给我们许多并发程序范例，如读者/作者问题、生产者/消费者问题、就餐的哲学家问题等，有些是他上课时留给学生的练习。他开发的 THE 多道程序系统为操作系统的理论和实现竖起了第一根标杆。Dijkstra 在程序正确性领域也做出了极为重要的贡献，这是 20 世纪 70 年代之后他最关注的领域，本书就是他一段工作的最重要的总结，下面将详细讨论。有关 Dijkstra 的更多贡献可以参

考介绍他的 WiKi 页和纪念网页（网址为：http://www.utexas.edu/faculty/council/2002-2003/memorials/Dijkstra/dijkstra.html）。

本书的英文书名是 *A Discipline of Programming*，是 Dijkstra 有关程序理论和编程技术的最重要的著作。本书是一本非常独特的编程技术专著。

首先，编程教科书通常要选一种流行语言作为示例的载体，而本书却没用任何可以由计算机执行的语言，所有示例程序都没运行过，但其正确性却有保证。作者也没用“伪代码”形式，因为那样做会带来不可避免的歧义性，而且伪代码无力作为严格的形式化推理工具。本书中的程序用作者自己设计的一种小语言描述，该语言反映了顺序（非并行）编程语言最本质的性质，还有许多重要特性将在下面讨论。

其次，虽然本书也给出了一系列编程实例，但采用的方式却与其他书籍大相径庭。作者在给出了要解决问题的陈述之后，总是很详细地描述了从问题出发，通过细致的分析思考，逐步深入工作，最后做出所需程序的整个过程。作者通过这种过程反复展示了在编程中应该怎样思考问题，提出合适的概念，规划处理流程，通过自上而下的设计和开发逐步做出程序框架，分析解决遇到的具体问题并做出各部分细节，直至完成完整的程序。作者还特别强调了“关注点分离”的重要性，强调在一个阶段解决一个问题。在一些复杂程序开发完成后还给出了细致的回顾性分析。作者在书中特别注意开发过程的真实性，并不回避其间提出过的错误概念和走过的弯路。这种真实的实践很难在其他编程著作中看到。

虽然上面两点已明显表现出本书的与众不同，但它们还不是本书最重要的特点。Dijsktra 在本书中最重要的关注点是程序的正确性，是关于程序和程序意义的“数学推导”和“证明”。他在前言中写到，我们的目标应该是“设计出具有高度信任水平的程序，而不仅仅是……一些胡乱涂抹出来的，……，随时准备被第一个反例推翻的程序正文”。他认为构造过程的目标应该是直接得到正确的程序，而不是得到一些“大致能用”的代码。目前的编程工作都表示这样，人们在开发中反复测试半成品或成品，找出其中缺陷并予以更正，直至作品或工作过程满足了某些外在“评价标准”，或者由于时间、金钱、人力限制而不得不结束工作。到最后开发者也不敢断定给出的就是所需要的“那个”程序。

有关如何开发出正确的程序，书中强调了多方面的问题。首先是大家都熟悉的问题分析，需要在动手编程之前把问题弄清楚，严格定义好，这件事怎么强调都不过分。这里不过多地讨论，书中有许多具体例子可供参考。更重要的是，Dijkstra 认为保证程序正确的主要武器就是数学意义上的严格描述和推导，他通过

一大批示例展示了严格推导在编程领域里的威力。Dijkstra 认为程序语言（无论哪种语言）本身是一种形式化的严格定义的描述形式，用它们写出的一个程序具有确定的语义。具体计算机运行该程序是实现它的语义，但其语义并不依赖于运行它的计算机，因此需要独立考虑和表达。为此，Dijkstra 基于状态和断言的概念提出了最弱前条件的思想，用谓词转换器作为程序语义的落脚点。由于常规语言不能完美地支持他的想法，他自己开发了一种语言，其中最重要的新概念是卫式命令，语言里的选择和重复结构都基于卫式命令定义。Dijkstra 用书中的实例展示了通过严格推理直接开发出正确程序的过程，展示了不变式、最弱前条件、谓词转换器等重要概念和严格的逻辑推理怎样在程序开发中发挥作用，帮助开发者（这里的实践者是 Dijkstra 本人）理清程序的线索，逐步推导出正确的程序。作者在这里提出的最弱前条件、谓词转换器、非确定性等概念对程序理论和实践的发展都产生了重要影响。

显然，并不是每个人都赞同 Dijkstra 的这种想法，未来世界中所有的程序也未必都采用他的这种理想主义的方式开发。对于数学和逻辑究竟能在编程世界里起多大作用，怀疑者们还在不断地提出各种质疑，而倡导此道的研究者们也在持续不断地工作。可以看到，他们的工作成果正在不断地被纳入越来越多的主要软件公司的开发过程，变成流行软件工具的有机组成部分。从另一个角度看，在流行的编程书籍中，也能越来越多地看到讨论状态、断言、前后条件、不变式、基于协约的编程等内容。这些趋势说明，作为一种复杂的智力劳动，由于其工作成果将变成当前和未来的工业化、信息化社会中所有基础设施的最关键支柱，编程和软件开发工作必将越来越强烈地关注其产品的正确性和可信任问题。本书作者在三十多年前的呼喊并没有过时，也不会被遗忘，他提出的想法和技术在未来软件开发中必定会起越来越重要的作用。原因无二，正如作者在另一处所言：“我早已对计算领域的实践活动有个总结：在那里乱搞的自由度太大，因此，数学的优美绝不是一种可有可无的奢侈，它关系到生命和死亡”。

本书出版于 1976 年，20 世纪 80 年代曾被影印并在国内计算机科学界流传，当时书名被译为《程序设计训练》，但很奇怪的是没出版过中文译本。20 世纪 90 年代后期，国内计算机领域开始迅猛发展，而这一经典著作却销声匿迹，不为人知，确实很令人遗憾。这次电子工业出版社引进本书是做了一件绝大的好事。我接手时，出版社曾计划以《编程的法则》为书名，我对此有疑。Dijkstra 在本书最后专门讨论了一个科学（或其他）领域的形成，以及其参与者智力活动的情况和性质。本书序言作者（图灵奖获得者 Tony Hoare）将编程与音乐、诗词等领域相提并论，讨论其卓越成就者对后人的启迪和引领。作为一种智力活动的学习和实践者，既需要获取他人经验，也需要内省和自我总结，将其说成是学一些“法则”

当然不妥。作为领域专家的 Dijkstra 虽然锋芒毕露，但其实他是一个谦虚的人，原书名用不定冠词“A”开头，表示这只是他认为在编程领域里有道理的一套智力修炼，未必就是终极和唯一的“那一套”。由于以《编程的一种修炼》作为中文书名很别扭，我选了目前这个名字，还好，它也没有强加于人的意味。

翻译本书就像是在倾听作者缓缓道出其深邃的思考，感觉非常有趣并到处受到启发（虽然我原来已经看过几遍），但是也感到非常累。要想把这本书翻译好确实非常困难，术语是最简单的问题（但也颇费琢磨），更重要的是，既要准确反映原文的字面意思，还要尽可能反映作者的意图和想法。在此基础上还要尽量符合汉语的习惯，语言流畅并容易理解。要把本书译到满意，我觉得至少还需要一两年时间。但出版社不能等，因此，我只能把目前的版本呈献给读者。我当然要对书中的所有缺陷负责，并为这些缺陷表示歉意。我将在 www.math.pku.edu.cn/teachers/qiuzy/为本书创建一个专门页面，做后续的弥补工作。最后，我要衷心感谢出版社的编辑，他们通过认真工作找出了译文里的一些错误。

裘宗燕

北京大学数学学院信息科学系

FOREWORD

In the older intellectual disciplines of poetry, music, art, and science, historians pay tribute to those outstanding practitioners, whose achievements have widened the experience and understanding of their admirers, and have inspired and enhanced the talents of their imitators. Their innovations are based on superb skill in the practice of their craft, combined with an acute insight into the underlying principles. In many cases, their influence is enhanced by their breadth of culture and the power and lucidity of their expression.

This book expounds, in its author's usual cultured style, his radical new insights into the nature of computer programming. From these insights, he has developed a new range of programming methods and notational tools, which are displayed and tested in a host of elegant and efficient examples. This will surely be recognised as one of the outstanding achievements in the development of the intellectual discipline of computer programming.

C.A.R. HOARE

序

在诗歌、音乐、艺术和科学等历史更为悠久的智力修炼领域，历史学家们都把颂词献给那里最卓越的实践者，因为他们的成就拓展了其赞美者的体验和理解，也大大启迪和提升了其追随者们的才华。他们的创新基于其在实践中积累的卓越技艺，并结合了他们对相应领域的基础理论的敏锐洞见。在很多情况下，他们的影响还由于其广博的文化积淀，以及他们在表达方式上的力量和透彻性而得到进一步的提升。

在本书中，作者以其习惯的文字风格，详尽地描述了他对计算机编程的基本性质的激进的新见解。基于这些见解，作者开发了一套编程方法以及与之相适应的记法工具，并用一大批优雅而且高效的实例展示和检验了它们。本书将注定成为在计算机编程的智力修炼领域发展中最杰出的成就之一。

C.A.R.Hoare

PREFACE

For a long time I have wanted to write a book somewhat along the lines of this one: on the one hand I knew that programs could have a compelling and deep logical beauty, on the other hand I was forced to admit that most programs are presented in a way fit for mechanical execution but, even if of any beauty at all, totally unfit for human appreciation. A second reason for dissatisfaction was that algorithms are often published in the form of finished products, while the majority of the considerations that had played their role during the design process and should justify the eventual shape of the finished program were often hardly mentioned. My original idea was to publish a number of beautiful algorithms in such a way that the reader could appreciate their beauty, and I envisaged doing so by describing the —real or imagined— design process that would each time lead to the program concerned. I have remained true to my original intention in the sense that the long sequence of chapters, in each of which a new problem is tackled and solved, is still the core of this monograph; on the other hand the final book is quite different from what I had foreseen, for the self-imposed task to present these solutions in a natural and convincing manner has been responsible for so much more, that I shall remain grateful forever for having undertaken it.

When starting on a book like this, one is immediately faced with the question: "Which programming language am I going to use?", and this is *not* a mere question of presentation! A most important, but also a most elusive, aspect of any tool is its influence on the habits of those who train themselves in its use. If the tool is a programming language, this influence is —whether we like it or not— an influence on our thinking habits. Having analyzed that influence to the best of my knowledge, I had come to the conclusion that none of the existing programming languages, nor a subset of them, would suit my purpose; on the other hand I knew myself so unready for the design

前言

很长时间以来，我一直想写一本基本上是按照本书线索的著作，原因是：一方面，我知道程序可以有迷人的形态和深刻的逻辑之美；另一方面，我又不得不接受这样的事实，即绝大部分程序只是以一种适合机器执行的方式表达，完全没有什么美感，也不适合人们欣赏。这种不满意还有第二个原因，那就是各种算法通常总是以一种完成了的产品形式发表，而在设计过程中起着最重要作用的，以及成为证明所完成程序的最终形式的正当性的各种思考的主要部分，通常都完全没有提及。我最初的想法是以读者能欣赏到它们的美的方式发表一系列优美的算法。对于如何做这件事，我当时的想法是描述一些实际的和想象中的设计过程，使其中的每个过程最终都得到了一个所需的程序。我在一定程度上实现了最初的想法，作为这本专著的核心部分是一系列的章节，每一章处理并解决一个新问题。而在另一方面，最终写出的这本书与我早前的期望又有很大不同，由于我特别希望用一种自然而且方便的方式来展现这些内容，因这种追求而强加给自己的任务变成了一种重要的责任。我将永远为自己完成了这一工作而感到欣慰。

在开始写一本像本书这样的著作时，人们立刻会面临一个问题："我准备使用哪一种编程语言？"而实际上这并不仅仅是一个有关展示形式的问题！任何工具的一个最重要的（而且也是最难琢磨的）方面，就是它对于被训练而将使用它的人们的工作习惯的影响，这种影响——无论我们是否喜欢——是对我们的思考习惯的影响。在尽可能地分析了这种影响的各方面情况之后，我得出了一个结论：没有一个现存的语言，也没有一个它们的子集适合我的目标。其次，我也知道自

of a new programming language that I had taken a vow not to do so for the next five years, and I had a most distinct feeling that that period had not yet elapsed! (Prior to that, among many other things, this monograph had to be written.) I have tried to resolve this conflict by only designing a mini-language suitable for my purposes, by making only those commitments that seemed unavoidable and sufficiently justified.

This hesitation and self-imposed restriction, when ill-understood, may make this monograph disappointing for many of its potential readers. It will certainly leave all those dissatisfied who identify the difficulty of programming with the difficulty of cunning exploitation of the elaborate and baroque tools known as "higher level programming languages" or —worse!— "programming systems". When they feel cheated because I just ignore all those bells and whistles, I can only answer: "Are you quite sure that all those bells and whistles, all those wonderful facilities of your so-called "powerful" programming languages belong to the solution set rather than to the problem set?". I can only hope that, in spite of my usage of a mini-language, they will study my text; after having done so, they may agree that, even without the bells and the whistles, so rich a subject remains that it is questionable whether the majority of the bells and the whistles should have been introduced in the first place. And to all readers with a pronounced interest in the design of programming languages, I can only express my regret that, as yet, I do not feel able to be much more explicit on that subject; on the other hand I hope that, for the time being, this monograph will inspire them and will enable them to avoid some of the mistakes they might have made without having read it.

During the act of writing —which was a continuous source of surprise and excitement— a text emerged that was rather different from what I had originally in mind. I started with the (understandable) intention to present my program developments with a little bit more formal apparatus than I used to use in my (introductory) lectures, in which semantics used to be introduced intuitively and correctness demonstrations were the usual mixture of rigorous arguments, handwaving, and eloquence. In laying the necessary foundations for such a more formal approach, I had two surprises. The first surprise was that the so-called "predicate transformers" that I had chosen as my vehicle provided a means for directly defining a relation between initial and final state, without any reference to intermediate states as may occur during program execution. I was very grateful for that, as it affords a clear separation between two of the programmer's major concerns, the mathematical correctness concerns (viz. whether the program defines the proper relation between initial and final state—and the predicate transformers give us a formal tool for that investigation without bringing computational processes into the picture) and the engineering concerns about efficiency (of which it is now clear that they are only defined in relation to an implementation). It turned out to

己完全没有为设计一种新的编程语言做好准备，因此，我曾发誓在随后的五年里不去做这件事。而且我有一种非常清晰的感觉：这个时期还没有过去！（但还有一个前提，就是除了其他事情外，这本书必须要写。）我试着消解这一矛盾的方式是设计了一个适合我的具体目标的小型语言，只做出一些看起来不可能避免，而且其正当性也得到了充分证实的承诺。

这种犹豫和自我强加的约束如果被错误地理解，有可能使本书的许多潜在读者对它感到很失望。那些把编程的困难等同于老练地利用那些精细而花哨的称为“高级编程语言”的工具，或者（更糟糕的！）“编程系统”的困难的人们注定会对这本书不满意。如果因为我忽略了所有那些诱人的花哨玩意儿而使他们感到受了骗，我只能回答说：“你真能确定所有那些诱人的花哨玩意儿，以及那些你所谓‘强大的’编程语言的美妙功能确实属于解集合，而不属于问题的集合吗？”我只是希望，即便我用的是一种小型语言，他们也能看看这本书。在做完这件事之后，他们有可能同意，虽然没有那些诱人的花哨玩意儿，仍然有非常丰富的问题需要讨论。因此，是否在一开始就介绍大部分花哨玩意儿，确实是应该质疑的。还有，对于那些明显是对编程语言的设计有兴趣的读者，我只能表示我的歉意。正如我已经做的那样，我没办法在这个问题上做更明确的事情。但在另一方面，我也希望随着时间的推移，这一专著会对他们有所启示，而且能帮助他们避免一些如果没有读过它有可能犯的错误。

在写作的过程中——它持续不断地让我感到惊喜和激动——逐渐呈现的文本与我初始时头脑中的想象大不相同。我一开始设想以一种（易理解的）方式去展示程序的开发过程，带上比我在（引论）课程中更多一点的形式化设施，其中以直观的方式介绍所使用的语义，有关正确性的论证采用通常的严格论述，手工编排，再加上有说服力的文字。在为更形式化的方法建立了必要的基础后，我得到了两个惊喜。第一个惊喜就是所谓的“谓词转换器”，作为我选用的工具，它提供了一种方法，使我们可以直接定义初始状态与最终状态之间的关系，不需要参考在程序实际执行中可能经历的中间状态。我对这种情况感到非常欣慰，因为这清晰地区分了程序员的两个主要关注点：数学的正确性（即程序是否定义了初始状态与最终状态之间所需的正确关系——谓词转换器是我们研究这一问题的一种形式化工具，研究中不需要考虑实际的计算进程），以及工程上对于效率的关注（现

be a most helpful discovery that the same program text always admits two rather complementary interpretations, the interpretation as a code for a predicate transformer, which seems the more suitable one for us, versus the interpretation as executable code, an interpretation I prefer to leave to the machines! The second surprise was that the most natural and systematic "codes for predicate transformers" that I could think of would call for nondeterministic implementations when regarded as "executable code". For a while I shuddered at the thought of introducing nondeterminacy already in uniprogramming (the complications it has caused in multiprogramming were only too well known to me!), until I realized that the text interpretation as code for a predicate transformer has its own, independent right of existence. (And in retrospect we may observe that many of the problems multiprogramming has posed in the past are nothing else but the consequence of a prior tendency to attach undue significance to determinacy.) Eventually I came to regard nondeterminacy as the normal situation, determinacy being reduced to a —not even very interesting— special case.

After having laid the foundations, I started with what I had intended to do all the time, viz. solve a long sequence of problems. To do so was an unexpected pleasure. I experienced that the formal apparatus gave me a much firmer grip on what I was doing than I was used to; I had the pleasure of discovering that explicit concerns about termination can be of great heuristic value—to the extent that I came to regret the strong bias towards partial correctness that is still so common. The greatest pleasure, however, was that for the majority of the problems that I had solved before, this time I ended up with a more beautiful solution! This was very encouraging, for I took it as an indication that the methods developed had, indeed, improved my programming ability.

How should this monograph be studied? The best advice I can give is to stop reading as soon as a problem has been described and to try to solve it yourself before reading on. Trying to solve the problem on your own seems the only way in which you can assess how difficult the problem is; it gives you the opportunity to compare your own solution with mine; and it may give you the satisfaction of having discovered yourself a solution that is superior to mine. And, by way of a priori reassurance: be not depressed when you find the text far from easy reading! Those who have studied the manuscript found it quite often difficult (but equally rewarding!); each time, however, that we analyzed their difficulties, we came together to the conclusion that not the text (i.e. the way of presentation), but the subject matter itself was "to blame". The moral of the story can only be that a nontrivial algorithm is just nontrivial, and that its final description in a programming language is highly compact compared to the considerations that justify its design: the shortness of the final text should not mislead us! One of my assistants made the suggestion —which I faithfully transmit, as it could be a valuable one—

在也很清楚，这件事只与实现有关）。这已经成为一种最有帮助的发现，因为同一个程序正文总是有两种相当互补的解释：它可以解释为一个谓词转换器的编码，这样做看起来更适合我们的需要；或者解释为可执行代码，我宁愿把这种解释留给机器去做。第二个惊喜是，我能够想象到的最自然而且最系统化的“谓词转换器的编码”，在被看作“可执行代码”时，将要求一种非确定性的实现。有一段时间，我对把非确定性引进单道编程感到不寒而栗（我对它给多道编程带来的复杂性知道得太多了！），直到我认识到，将程序正文解释为一个谓词转换器的编码有其自身的存在理由。（回顾往事，我们可以看到，过去提出的有关多道编程的许多问题并不是别的什么，只不过是不适当地过分强调了确定性的重要性而带来的后果。）我最终认识到，应该把非确定性看作正常情况，这样，确定性将变成一种——并不很有趣的——特例了。

在打好了有关的基础之后，我将所有的时间都投入了想做的事情上，也就是说，去解决一系列的问题。做这件事使我得到了未曾预料的快乐。与我以前的工作方式相比，形式化的设施使我能更牢固地把握所做的工作。我很高兴地发现，明确地关注终止性问题能带来许多富于启发性的看法，以至于使我觉得偏向于考虑部分正确性的观点如此常见是非常令人遗憾的。然而，最大的快乐是，对于大部分我原来做过的问题，这次都得到了更漂亮的解答！这是非常令人鼓舞的事情，我将它看作是一种指示剂，说明我所开发的方法确实提升了我的编程能力。

应该怎样学习这本专著呢？我能给出的最佳建议就是，一旦看完了问题的描述，就停止阅读，转去试着自己解决它。尝试自己解决问题，是你能自己认识和评价问题的困难程度的唯一方法，它也使你有机会去比较你的解和我给出的解，还给你得到满足的机会，即看到你给出的解比我给出的更好。还是要先说一下，当你发现这里的内容远不是非常容易读的时候，请不要沮丧。研读过这本专著的人都觉得它的内容通常是很难的（但收获也同样很多！）。然而，每次我们分析遇到的困难时，得到的结论都是应该将这种困难“归咎于”实际讨论的问题，而不是有关的文字本身（即它的表达方式）。它的寓意是，一个非平凡的算法本身就是非平凡的，而与论证其设计的正确性的思考相比，在一个编程语言里做出的算法描述是高度紧凑的：不要受到最后的程序正文长度的误导！我的一个助理给出的建议——这也是我忠实地采纳的，因为它可以很有价值——是让学生分为一些小

that little groups of students should study it together. (Here I must add a parenthetical remark about the "difficulty" of the text. After having devoted a considerable number of years of my scientific life to clarifying the programmer's task, with the aim of making it intellectually better manageable, I found this effort at clarification to my amazement (and annoyance) repeatedly rewarded by the accusation that "I had made programming difficult". But the difficulty has always been there, and only by making it visible can we hope to become able to design programs with a high confidence level, rather than "smearing code", i.e., producing texts with the status of hardly supported conjectures that wait to be killed by the first counterexample. None of the programs in this monograph, needless to say, has been tested on a machine.)

I owe the reader an explanation why I have kept the mini-language so small that it does not even contain procedures and recursion. As each next language extension would have added a few more chapters to the book and, therefore, would have made it correspondingly more expensive, the absence of most possible extensions (such as, for instance, multiprogramming) needs no further justification. Procedures, however, have always occupied such a central position and recursion has been for computing science so much the hallmark of academic respectability, that some explanation is due.

First of all, this monograph has not been written for the novice and, consequently, I expect my readers to be familiar with these concepts. Secondly, this book is not an introductory text on a specific programming language and the absence of these constructs and examples of their use should therefore not be interpreted as my inability or unwillingness to use them, nor as a suggestion that anyone else who can use them well should not do so. The point is that I felt no need for them in order to get my message across, viz. how a carefully chosen separation of concerns is essential for the design of in all respects, high-quality programs: the modest tools of the mini-language gave us already more than enough latitude for nontrivial, yet very satisfactory designs.

The above explanation, although sufficient, is, however, not the full story. In any case I felt obliged to present repetition as a construct in its own right, as such a presentation seemed to me overdue. When programming languages emerged, the "dynamic" nature of the assignment statement did not seem to fit too well into the "static" nature of traditional mathematics. For lack of adequate theory mathematicians did not feel too easy about it, and, because it is the repetitive construct that creates the need for assignment to variables, mathematicians did not feel too easy about repetition either. When programming languages without assignments and without repetition —such as pure LISP— were developed, many felt greatly relieved. They were back on the familiar grounds and saw a glimmer of hope of making programming an activity with a firm and respectable mathematical basis. (Up to this very day

组一起学习这本书。(在这里，我必须对书中正文的“困难程度”加一点附带的说明。我本人科学生涯中的许多年一直在致力于弄清楚程序员的任务，目标是设法使它成为智力上可以管理的工作。从事了多年的这种澄清性的工作之后，我感到好玩，也很吃惊地发现，反复出现的回馈是“我把编程弄得更困难了”。但困难始终在那里，只有将其变为明显可见的，我们才有希望设计出具有高度信任水平的程序，而不仅仅是做出了一些“胡乱涂抹出的代码”，即那种基于根本无法得到支持的假设，准备着被第一个反例推翻的程序正文。不用说，本书的任何一个程序都没有在机器上测试过。)

我还要给读者解释为什么我把这里的小型语言弄得这么小，小到没有包含过程和递归。由于任何一点语言扩充都会给这本书增加几章内容，并因此使它也相应地变得更昂贵，所以对于大部分可能的扩充(例如多道编程)，我都不需要更多的解释。而过程总是在编程中居于核心的地位，递归对于计算科学而言也是最受学术界重视的标志，因此，我必须给出一些解释。

首先，这本专著不是为新手写的，因此，我期望本书的读者熟悉这些概念。其次，本书不是某个特殊编程语言的引论教材，缺乏这些结构和使用它们的例子，不会被解释为我不能或者不希望使用它们，也不会被作为是建议那些有能力使用这些结构的人不要用它们。这里的关键是我觉得在传递想给出的信息时我并不需要这些结构。我想讨论的是应该仔细地分离各种关注，以及为什么从各个方面看这种做法都是设计出高质量程序的最重要的基础。以这里的小型语言作为一种有节制的工具，对于给出各种非平凡而且非常令人满意的设计而言，已经能给我们足够的行动自由了。

前面的解释虽然已经很充分，但还不是故事的全部。在任何情况下，我都觉得必须把重复结构本身作为语言中的一种结构，因为在我看来，这样的阐释是早就应该有的东西。当编程语言诞生时，其赋值语句的“动态”性质看起来与传统数学的“静态”性质很不吻合。由于没有合适的理论，数学家就觉得很不喜欢它。而且，由于重复结构是需要变量赋值的最根本原因，数学家也很不喜欢重复结构。当人们开发出没有赋值，也没有重复结构的编程语言——例如，纯 LISP——时，许多人都大大地松了一口气。这使他们又可以回到自己熟悉的场地里，看到了一点希望的微光：有可能将编程变成一种具有坚实的而且得到广泛尊重的数学基础