国外优秀信息科学与技术系列教学用书

程序设计语言概念

（影印版）

CONCEPTS IN PROGRAMMING LANGUAGES

■ John C. Mitchell

高等教育出版社
Higher Education Press

国外优秀信息科学与技术系列教学用书

程序设计语言概念

（影印版）

CONCEPTS IN PROGRAMMING LANGUAGES

John C. Mitchell

高等教育出版社

图字：01-2004-1250 号

Concepts in Programming Languages
John C. Mitchell

本版本仅获准在中华人民共和国大陆地区发行和销售（不包括中国香港、澳门特别行政区和中国台湾以及其他地区）。

Originally published by Cambridge University Press in 2003.
This reprint edition is published with the permission of the Syndicate of the Press of the University of Cambridge, Cambridge, England.
原版由剑桥大学出版社于 2003 年出版。
本影印版由英国剑桥的剑桥大学出版集团授权影印。

Concepts in Programming Languages by John C. Mitchell, Copyright © Cambridge University Press 2003
This book is in copyright. Subject to statutory exception and to the provisions of relevant collective licensing agreements, no reproduction of any part may take place without the written permission of Cambridge University Press.

图书在版编目（CIP）数据

程序设计语言概念＝Concepts in Programming Lang
uages /（美）米切尔（Mitchell,J.C.）.—影印本.
北京：高等教育出版社，2004.8
ISBN 7-04-015779-9

Ⅰ.程... Ⅱ.米... Ⅲ.程序语言-英文
Ⅳ.TP312

中国版本图书馆 CIP 数据核字(2004)第 070685 号

出版发行	高等教育出版社	购书热线	010-64054588
社　　址	北京市西城区德外大街 4 号	免费咨询	800-810-0598
邮政编码	100011	网　　址	http://www.hep.edu.cn
总　　机	010-82028899		http://www.hep.com.cn
经　　销	新华书店北京发行所		
印　　刷	北京外文印刷厂		
开　　本	787×1092　1/16	版　　次	2004 年 8 月第 1 版
印　　张	33.75	印　　次	2004 年 8 月第 1 次印刷
字　　数	650 000	定　　价	42.00 元

本书如有缺页、倒页、脱页等质量问题，请到所购图书销售部门联系调换。
版权所有　侵权必究

出 版 说 明

20 世纪末，以计算机和通信技术为代表的信息科学和技术对世界经济、科技、军事、教育和文化等产生了深刻影响。信息科学技术的迅速普及和应用，带动了世界范围信息产业的蓬勃发展，为许多国家带来了丰厚的回报。

进入 21 世纪，尤其随着我国加入 WTO，信息产业的国际竞争将更加激烈。我国信息产业虽然在 20 世纪末取得了迅猛发展，但与发达国家相比，甚至与印度、爱尔兰等国家相比，还有很大差距。国家信息化的发展速度和信息产业的国际竞争能力，最终都将取决于信息科学技术人才的质量和数量。引进国外信息科学和技术优秀教材，在有条件的学校推动开展英语授课或双语教学，是教育部为加快培养大批高质量的信息技术人才采取的一项重要举措。

为此，教育部要求由高等教育出版社首先开展信息科学和技术教材的引进试点工作。同时提出了两点要求，一是要高水平，二是要低价格。在高等教育出版社和信息科学技术引进教材专家组的努力下，经过比较短的时间，第一批由教育部高等教育司推荐的 20 多种引进教材已经陆续出版。这套教材出版后受到了广泛的好评，其中有不少是世界信息科学技术领域著名专家、教授的经典之作和反映信息科学技术最新进展的优秀作品，代表了目前世界信息科学技术教育的一流水平，而且价格也是最优惠的，与国内同类自编教材相当。这套教材基本覆盖了计算机科学与技术专业的课程体系，体现了权威性、系统性、先进性和经济性等特点。

目前，教育部正在全国 35 所高校推动示范性软件学院的建设，这也是加快培养信息科学技术人才的重要举措之一。为配合软件学院的教学工作，结合各软件学院的教学计划和课程设置，高等教育出版社近期聘请有关专家和软件学院的教师遴选推荐了一批相应的原版教学用书，正陆续组织出版，以方便各软件学院开展双语教学。

我们希望这些教学用书的引进出版，对于提高我国高等学校信息科学技术的教学水平，缩小与国际先进水平的差距，加快培养一大批具有国际竞争力的高质量信息技术人才，起到积极的推动作用。同时我们也欢迎广大教师和专家们对我们的教材引进工作提出宝贵的意见和建议。联系方式：hep.cs@263.net。

高等教育出版社

二〇〇二年九月

Preface

> A good programming language is a conceptual universe for thinking about programming.
>
> Alan Perlis, NATO Conference on Software Engineering Techniques, Rome, 1969

Programming languages provide the abstractions, organizing principles, and control structures that programmers use to write good programs. This book is about the concepts that appear in programming languages, issues that arise in their implementation, and the way that language design affects program development. The text is divided into four parts:

- *Part 1:* Functions and Foundations
- *Part 2:* Procedures, Types, Memory Management, and Control
- *Part 3:* Modularity, Abstraction, and Object-Oriented Programming
- *Part 4:* Concurrency and Logic Programming

Part 1 contains a short study of Lisp as a worked example of programming language analysis and covers compiler structure, parsing, lambda calculus, and denotational semantics. A short Computability chapter provides information about the limits of compile-time program analysis and optimization.

Part 2 uses procedural Algol family languages and ML to study types, memory management, and control structures.

In Part 3 we look at program organization using abstract data types, modules, and objects. Because object-oriented programming is the most prominent paradigm in current practice, several different object-oriented languages are compared. Separate chapters explore and compare Simula, Smalltalk, C++, and Java.

Part 4 contains chapters on language mechanisms for concurrency and on logic programming.

The book is intended for upper-level undergraduate students and beginning graduate students with some knowledge of basic programming. Students are expected to have some knowledge of C or some other procedural language and some

此为试读，需要完整PDF请访问：www.ertongbook.com

acquaintance with C++ or some form of object-oriented language. Some experience with Lisp, Scheme, or ML is helpful in Parts 1 and 2, although many students have successfully completed the course based on this book without this background. It is also helpful if students have some experience with simple analysis of algorithms and data structures. For example, in comparing implementations of certain constructs, it will be useful to distinguish between algorithms of constant-, polynomial-, and exponential-time complexity.

After reading this book, students will have a better understanding of the range of programming languages that have been used over the past 40 years, a better understanding of the issues and trade-offs that arise in programming language design, and a better appreciation of the advantages and pitfalls of the programming languages they use. Because different languages present different programming concepts, students will be able to improve their programming by importing ideas from other languages into the programs they write.

Acknowledgments

This book developed as a set of notes for Stanford CS 242, a course in programming languages that I have taught since 1993. Each year, energetic teaching assistants have helped debug example programs for lectures, formulate homework problems, and prepare model solutions. The organization and content of the course have been improved greatly by their suggestions. Special thanks go to Kathleen Fisher, who was a teaching assistant in 1993 and 1994 and taught the course in my absence in 1995. Kathleen helped me organize the material in the early years and, in 1995, transcribed my handwritten notes into online form. Thanks to Amit Patel for his initiative in organizing homework assignments and solutions and to Vitaly Shmatikov for persevering with the glossary of programming language terms. Anne Bracy, Dan Bentley, and Stephen Freund thoughtfully proofread many chapters.

Lauren Cowles, Alan Harvey, and David Tranah of Cambridge University Press were encouraging and helpful. I particularly appreciate Lauren's careful reading and detailed comments of twelve full chapters in draft form. Thanks also are due to the reviewers they enlisted, who made a number of helpful suggestions on early versions of the book. Zena Ariola taught from book drafts at the University of Oregon several years in a row and sent many helpful suggestions; other test instructors also provided helpful feedback.

Finally, special thanks to Krzystof Apt for contributing a chapter on logic programming.

John Mitchell

郑 重 声 明

高等教育出版社依法对本书享有专有出版权。任何未经许可的复制、销售行为均违反《中华人民共和国著作权法》，其行为人将承担相应的民事责任和行政责任，构成犯罪的，将被依法追究刑事责任。为了维护市场秩序，保护读者的合法权益，避免读者误用盗版书造成不良后果，我社将配合行政执法部门和司法机关对违法犯罪的单位和个人给予严厉打击。社会各界人士如发现上述侵权行为，希望及时举报，本社将奖励举报有功人员。

反盗版举报电话：(010) 58581897/58581896/58581879

传　　真：(010) 82086060

E - mail：dd@hep.com.cn

通信地址：北京市西城区德外大街 4 号
高等教育出版社打击盗版办公室

邮　　编：100011

购书请拨打电话：(010)64014089　64054601　64054588

策划编辑　康兆华
责任编辑　康兆华
市场策划　陈　振
封面设计　张　楠
责任印制　陈伟光

Contents

此为试读,需要完整PDF请访问: www.ertongbook.com

PART 1

Functions and Foundations

PART 1

Functions and Foundations

1

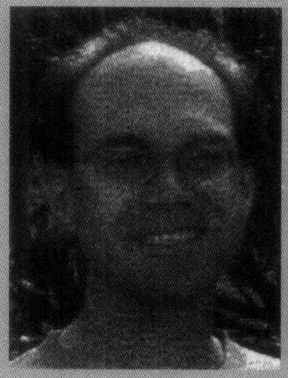
Introduction

"The Medium Is the Message"
Marshall McLuhan

1.1 PROGRAMMING LANGUAGES

Programming languages are the medium of expression in the art of computer programming. An ideal programming language will make it easy for programmers to write programs succinctly and clearly. Because programs are meant to be understood, modified, and maintained over their lifetime, a good programming language will help others read programs and understand how they work. Software design and construction are complex tasks. Many software systems consist of interacting parts. These parts, or software components, may interact in complicated ways. To manage complexity, the interfaces and communication between components must be designed carefully. A good language for large-scale programming will help programmers manage the interaction among software components effectively. In evaluating programming languages, we must consider the tasks of designing, implementing, testing, and maintaining software, asking how well each language supports each part of the software life cycle.

There are many difficult trade-offs in programming language design. Some language features make it easy for us to write programs quickly, but may make it harder for us to design testing tools or methods. Some language constructs make it easier for a compiler to optimize programs, but may make programming cumbersome. Because different computing environments and applications require different program characteristics, different programming language designers have chosen different trade-offs. In fact, virtually all successful programming languages were originally designed for one specific use. This is not to say that each language is good for only one purpose. However, focusing on a single application helps language designers make consistent, purposeful decisions. A single application also helps with one of the most difficult parts of language design: leaving good ideas out.



THE AUTHOR

I hope you enjoy using this book. At the beginning of each chapter, I have included pictures of people involved in the development or analysis of programming languages. Some of these people are famous, with major awards and published biographies. Others are less widely recognized. When possible, I have tried to include some personal information based on my encounters with these people. This is to emphasize that programming languages are developed by real human beings. Like most human artifacts, a programming language inevitably reflects some of the personality of its designers.

As a disclaimer, let me point out that I have not made an attempt to be comprehensive in my brief biographical comments. I have tried to liven up the text with a bit of humor when possible, leaving serious biography to more serious biographers. There simply is not space to mention all of the people who have played important roles in the history of programming languages.

Historical and biographical texts on computer science and computer scientists have become increasingly available in recent years. If you like reading about computer pioneers, you might enjoy paging through *Out of Their Minds: The Lives and Discoveries of 15 Great Computer Scientists* by Dennis Shasha and Cathy Lazere or other books on the history of computer science.

John Mitchell

Even if you do not use many of the programming languages in this book, you may still be able to put the conceptual framework presented in these languages to good use. When I was a student in the mid-1970s, all "serious" programmers (at my university, anyway) used Fortran. Fortran did not allow recursion, and recursion was generally regarded as too inefficient to be practical for "real programming." However, the instructor of one course I took argued that recursion was still an important idea and explained how recursive techniques could be used in Fortran by managing data in an array. I am glad I took that course and not one that dismissed recursion as an impractical idea. In the 1980s, many people considered object-oriented programming too inefficient and clumsy for real programming. However, students who learned about object-oriented programming in the 1980s were certainly happy to know about

these "futuristic" languages in the 1990s, as object-oriented programming became more widely accepted and used.

Although this is not a book about the history of programming languages, there is some attention to history throughout the book. One reason for discussing historical languages is that this gives us a realistic way to understand programming language trade-offs. For example, programs were different when machines were slow and memory was scarce. The concerns of programming language designers were therefore different in the 1960s from the current concerns. By imaging the state of the art in some bygone era, we can give more serious thought to why language designers made certain decisions. This way of thinking about languages and computing may help us in the future, when computing conditions may change to resemble some past situation. For example, the recent rise in popularity of handheld computing devices and embedded processors has led to renewed interest in programming for devices with limited memory and limited computing power.

When we discuss specific languages in this book, we generally refer to the original or historically important form of a language. For example, "Fortran" means the Fortran of the 1960s and early 1970s. These early languages were called Fortran I, Fortran II, Fortran III, and so on. In recent years, Fortran has evolved to include more modern features, and the distinction between Fortran and other languages has blurred to some extent. Similarly, Lisp generally refers to the Lisps of the 1960s, Smalltalk to the language of the late 1970s and 1980s, and so on.

1.2 GOALS

In this book we are concerned with the basic concepts that appear in modern programming languages, their interaction, and the relationship between programming languages and methods for program development. A recurring theme is the trade-off between language expressiveness and simplicity of implementation. For each programming language feature we consider, we examine the ways that it can be used in programming and the kinds of implementation techniques that may be used to compile and execute it efficiently.

1.2.1 General Goals

In this book we have the following general goals:

- To understand the *design space* of programming languages. This includes concepts and constructs from past programming languages as well as those that may be used more widely in the future. We also try to understand some of the major conflicts and trade-offs between language features, including implementation costs.
- To develop a better understanding of the languages we currently use by comparing them with other languages.
- To understand the programming techniques associated with various language features. The study of programming languages is, in part, the study of conceptual frameworks for problem solving, software construction, and development.

Many of the ideas in this book are common knowledge among professional programmers. The material and ways of thinking presented in this book should be useful to you in future programming and in talking to experienced programmers if you work for a software company or have an interview for a job. By the end of the course, you will be able to evaluate language features, their costs, and how they fit together.

1.2.2 Specific Themes

Here are some specific themes that are addressed repeatedly in the text:

- *Computability:* Some problems cannot be solved by computer. The undecidability of the halting problem implies that programming language compilers and interpreters cannot do everything that we might wish they could do.
- *Static analysis:* There is a difference between compile time and run time. At compile time, the program is known but the input is not. At run time, the program and the input are both available to the run-time system. Although a program designer or implementer would like to find errors at compile time, many will not surface until run time. Methods that detect program errors at compile time are usually conservative, which means that when they say a program does not have a certain kind of error this statement is correct. However, compile-time error-detection methods will usually say that some programs contain errors even if errors may not actually occur when the program is run.
- *Expressiveness versus efficiency:* There are many situations in which it would be convenient to have a programming language implementation do something automatically. An example discussed in Chapter 3 is memory management: The Lisp run-time system uses garbage collection to detect memory locations no longer needed by the program. When something is done automatically, there is a cost. Although an automatic method may save the programmer from thinking about something, the implementation of the language may run more slowly. In some cases, the automatic method may make it easier to write programs and make programming less prone to error. In other cases, the resulting slowdown in program execution may make the automatic method infeasible.

1.3 PROGRAMMING LANGUAGE HISTORY

Hundreds of programming languages have been designed and implemented over the last 50 years. As many as 50 of these programming languages contained new concepts, useful refinements, or innovations worthy of mention. Because there are far too many programming languages to survey, however, we concentrate on six programming languages: Lisp, ML, C, C++, Smalltalk, and Java. Together, these languages contain most of the important language features that have been invented since higher-level programming languages emerged from the primordial swamp of assembly language programming around 1960.

The history of modern programming languages begins around 1958–1960 with the development of Algol, Cobol, Fortran, and Lisp. The main body of this book

此为试读,需要完整PDF请访问: www.ertongbook.com