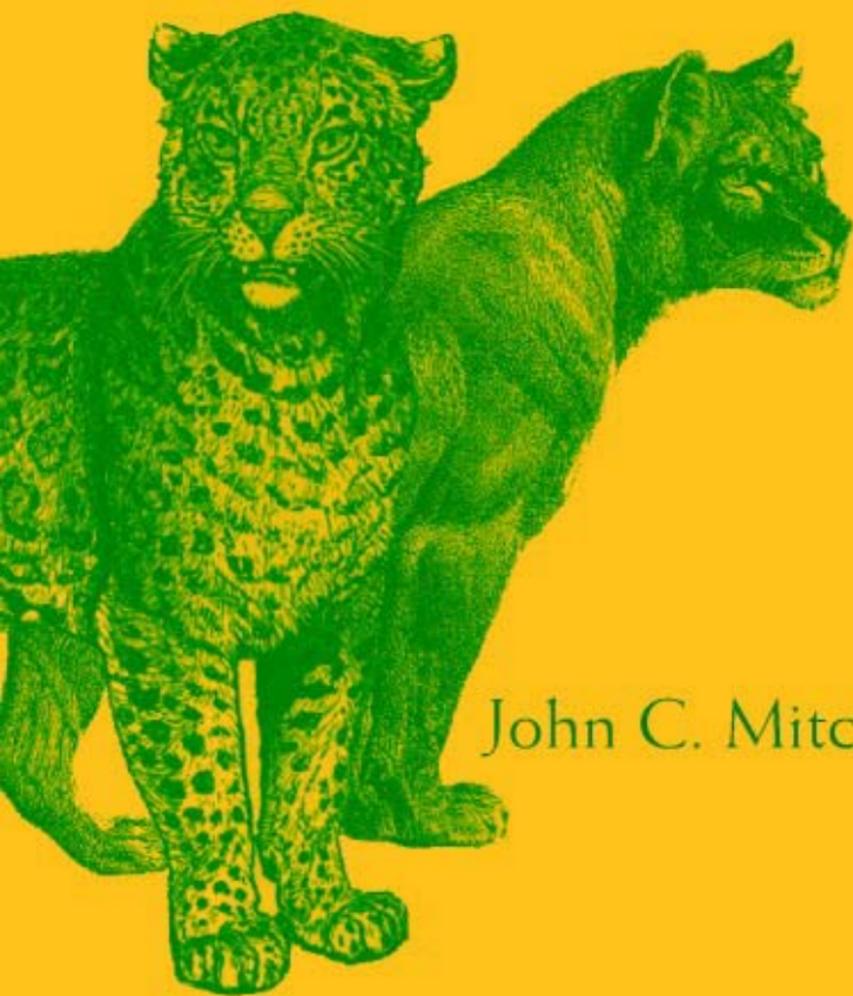


CONCEPTS IN PROGRAMMING LANGUAGES



John C. Mitchell

CAMBRIDGE

more information - www.cambridge.org/0521780985

This page intentionally left blank

Concepts in Programming Languages

This textbook for undergraduate and beginning graduate students explains and examines the central concepts used in modern programming languages, such as functions, types, memory management, and control. The book is unique in its comprehensive presentation and comparison of major object-oriented programming languages. Separate chapters examine the history of objects, Simula and Smalltalk, and the prominent languages C++ and Java.

The author presents foundational topics, such as lambda calculus and denotational semantics, in an easy-to-read, informal style, focusing on the main insights provided by these theories. Advanced topics include concurrency and concurrent object-oriented programming. A chapter on logic programming illustrates the importance of specialized programming methods for certain kinds of problems.

This book will give the reader 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.

John C. Mitchell is Professor of Computer Science at Stanford University, where he has been a popular teacher for more than a decade. Many of his former students are successful in research and private industry. He received his Ph.D. from MIT in 1984 and was a Member of Technical Staff at AT&T Bell Laboratories before joining the faculty at Stanford. Over the past twenty years, Mitchell has been a featured speaker at international conferences; has led research projects on a variety of topics, including programming language design and analysis, computer security, and applications of mathematical logic to computer science; and has written more than 100 research articles. His previous textbook, *Foundations for Programming Languages* (MIT Press, 1996), covers lambda calculus, type systems, logic for program verification, and mathematical semantics of programming languages. Professor Mitchell was a member of the programming language subcommittee of the ACM/IEEE Curriculum 2001 standardization effort and the 2002 Program Chair of the ACM Principles of Programming Languages conference.

CONCEPTS IN PROGRAMMING LANGUAGES

John C. Mitchell

Stanford University



PUBLISHED BY THE PRESS SYNDICATE OF THE UNIVERSITY OF CAMBRIDGE
The Pitt Building, Trumpington Street, Cambridge, United Kingdom

CAMBRIDGE UNIVERSITY PRESS

The Edinburgh Building, Cambridge CB2 2RU, UK
40 West 20th Street, New York, NY 10011-4211, USA
477 Williamstown Road, Port Melbourne, VIC 3207, Australia
Ruiz de Alarcón 13, 28014 Madrid, Spain
Dock House, The Waterfront, Cape Town 8001, South Africa

<http://www.cambridge.org>

© Cambridge University Press 2004

First published in printed format 2002

ISBN 0-511-03492-X eBook (Adobe Reader)

ISBN 0-521-78098-5 hardback

Contents

<i>Preface</i>	<i>page ix</i>
Part 1 Functions and Foundations	
1 Introduction	3
1.1 Programming Languages	3
1.2 Goals	5
1.3 Programming Language History	6
1.4 Organization: Concepts and Languages	8
2 Computability	10
2.1 Partial Functions and Computability	10
2.2 Chapter Summary	16
Exercises	16
3 Lisp: Functions, Recursion, and Lists	18
3.1 Lisp History	18
3.2 Good Language Design	20
3.3 Brief Language Overview	22
3.4 Innovations in the Design of Lisp	25
3.5 Chapter Summary: Contributions of Lisp	39
Exercises	40
4 Fundamentals	48
4.1 Compilers and Syntax	48
4.2 Lambda Calculus	57
4.3 Denotational Semantics	67
4.4 Functional and Imperative Languages	76
4.5 Chapter Summary	82
Exercises	83

Part 2 Procedures, Types, Memory Management, and Control

5	The Algol Family and ML	93
	5.1 The Algol Family of Programming Languages	93
	5.2 The Development of C	99
	5.3 The LCF System and ML	101
	5.4 The ML Programming Language	103
	5.5 Chapter Summary	121
	Exercises	122
6	Type Systems and Type Inference	129
	6.1 Types in Programming	129
	6.2 Type Safety and Type Checking	132
	6.3 Type Inference	135
	6.4 Polymorphism and Overloading	145
	6.5 Type Declarations and Type Equality	151
	6.6 Chapter Summary	155
	Exercises	156
7	Scope, Functions, and Storage Management	162
	7.1 Block-Structured Languages	162
	7.2 In-Line Blocks	165
	7.3 Functions and Procedures	170
	7.4 Higher-Order Functions	182
	7.5 Chapter Summary	190
	Exercises	191
8	Control in Sequential Languages	204
	8.1 Structured Control	204
	8.2 Exceptions	207
	8.3 Continuations	218
	8.4 Functions and Evaluation Order	223
	8.5 Chapter Summary	227
	Exercises	228
Part 3 Modularity, Abstraction, and Object-Oriented Programming		
9	Data Abstraction and Modularity	235
	9.1 Structured Programming	235
	9.2 Language Support for Abstraction	242
	9.3 Modules	252
	9.4 Generic Abstractions	259
	9.5 Chapter Summary	269
	Exercises	271
10	Concepts in Object-Oriented Languages	277
	10.1 Object-Oriented Design	277
	10.2 Four Basic Concepts in Object-Oriented Languages	278

10.3 Program Structure	288
10.4 Design Patterns	290
10.5 Chapter Summary	292
10.6 Looking Forward: Simula, Smalltalk, C++, Java	293
Exercises	294
11 History of Objects: Simula and Smalltalk	300
11.1 Origin of Objects in Simula	300
11.2 Objects in Simula	303
11.3 Subclasses and Subtypes in Simula	308
11.4 Development of Smalltalk	310
11.5 Smalltalk Language Features	312
11.6 Smalltalk Flexibility	318
11.7 Relationship between Subtyping and Inheritance	322
11.8 Chapter Summary	326
Exercises	327
12 Objects and Run-Time Efficiency: C++	337
12.1 Design Goals and Constraints	337
12.2 Overview of C++	340
12.3 Classes, Inheritance, and Virtual Functions	346
12.4 Subtyping	355
12.5 Multiple Inheritance	359
12.6 Chapter Summary	366
Exercises	367
13 Portability and Safety: Java	384
13.1 Java Language Overview	386
13.2 Java Classes and Inheritance	389
13.3 Java Types and Subtyping	396
13.4 Java System Architecture	404
13.5 Security Features	412
13.6 Java Summary	417
Exercises	420
Part 4 Concurrency and Logic Programming	
14 Concurrent and Distributed Programming	431
14.1 Basic Concepts in Concurrency	433
14.2 The Actor Model	441
14.3 Concurrent ML	445
14.4 Java Concurrency	454
14.5 Chapter Summary	466
Exercises	469

15 The Logic Programming Paradigm and Prolog	475
15.1 History of Logic Programming	475
15.2 Brief Overview of the Logic Programming Paradigm	476
15.3 Equations Solved by Unification as Atomic Actions	478
15.4 Clauses as Parts of Procedure Declarations	482
15.5 Prolog's Approach to Programming	486
15.6 Arithmetic in Prolog	492
15.7 Control, Ambivalent Syntax, and Meta-Variables	496
15.8 Assessment of Prolog	505
15.9 Bibliographic Remarks	507
15.10 Chapter Summary	507
Appendix A Additional Program Examples	509
A.1 Procedural and Object-Oriented Organization	509
<i>Glossary</i>	521
<i>Index</i>	525

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

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 Krzysztof Apt for contributing a chapter on logic programming.

John Mitchell

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

covers Lisp, with a shorter discussion of Algol and subsequent related languages. A brief account of some earlier languages is given here for those who may be curious about programming language prehistory.

In the 1950s, a number of languages were developed to simplify the process of writing sequences of computer instructions. In this decade, computers were very primitive by modern standards. Most programming was done with the native machine language of the underlying hardware. This was acceptable because programs were small and efficiency was extremely important. The two most important programming language developments of the 1950s were Fortran and Cobol.

Fortran was developed at IBM around 1954–1956 by a team led by John Backus. The main innovation of Fortran (a contraction of formula translator) was that it became possible to use ordinary mathematical notation in expressions. For example, the Fortran expression for adding the value of i to twice the value of j is $i + 2*j$. Before the development of Fortran, it might have been necessary to place i in a register, place j in a register, multiply j times 2 and then add the result to i . Fortran allowed programmers to think more naturally about numerical calculation by using symbolic names for variables and leaving some details of evaluation order to the compiler. Fortran also had subroutines (a form of procedure or function), arrays, formatted input and output, and declarations that gave programmers explicit control over the placement of variables and arrays in memory. However, that was about it. To give you some idea of the limitations of Fortran, many early Fortran compilers stored numbers 1, 2, 3 . . . in memory locations, and programmers could change the values of numbers if they were not careful! In addition, it was not possible for a Fortran subroutine to call itself, as this required memory management techniques that had not been invented yet (see Chapter 7).

Cobol is a programming language designed for business applications. Like Fortran programs, many Cobol programs are still in use today, although current versions of Fortran and Cobol differ substantially from forms of these languages of the 1950s. The primary designer of Cobol was Grace Murray Hopper, an important computer pioneer. The syntax of Cobol was intended to resemble that of common English. It has been suggested in jest that if object-oriented Cobol were a standard today, we would use “add 1 to Cobol giving Cobol” instead of “C++”.

The earliest languages covered in any detail in this book are Lisp and Algol, which both came out around 1960. These languages have stack memory management and recursive functions or procedures. Lisp provides higher-order functions (still not available in many current languages) and garbage collection, whereas the Algol family of languages provides better type systems and data structuring. The main innovations of the 1970s were methods for organizing data, such as records (or structs), abstract data types, and early forms of objects. Objects became mainstream in the 1980s, and the 1990s brought increasing interest in network-centric computing, interoperability, and security and correctness issues associated with active content on the Internet. The 21st century promises greater diversity of computing devices, cheaper and more powerful hardware, and increasing interest in correctness, security, and interoperability.

1.4 ORGANIZATION: CONCEPTS AND LANGUAGES

There are many important language concepts and many programming languages. The most natural way to summarize the field is to use a two-dimensional matrix, with languages along one axis and concepts along the other. Here is a partial sketch of such a matrix:

Language	Expressions	Functions	Heap storage	Exceptions	Modules	Objects	Threads
Lisp	x	x	x				
C	x	x	x				
Algol 60	x	x					
Algol 68	x	x	x				x
Pascal	x	x	x				
Modula-2	x	x	x		x		
Modula-3	x	x	x	x	x	x	
ML	x	x	x	x	x		
Simula	x	x	x			x	x
Smalltalk	x	x	x	x		x	x
C++	x	x	x	x	x	x	
Objective C	x	x	x			x	
Java	x	x	x	x	x	x	x

Although this matrix lists only a fraction of the languages and concepts that might be covered in a basic text or course on the programming languages, one general characteristic should be clear. There are some basic language concepts, such as expressions, functions, local variables, and stack storage allocation that are present in many languages. For these concepts, it makes more sense to discuss the concept in general than to go through a long list of similar languages. On the other hand, for concepts such as objects and threads, there are relatively few languages that exhibit these concepts in interesting ways. Therefore, we can study most of the interesting aspects of objects by comparing a few languages. Another factor that is not clear from the matrix is that, for some concepts, there is considerable variation from language to language. For example, it is more interesting to compare the way objects have been integrated into languages than it is to compare integer expressions. This is another reason why competing object-oriented languages are compared, but basic concepts related to expressions, statements, functions, and so on, are covered only once, in a concept-oriented way.

Most courses and texts on programming languages use some combination of language-based and concept-based presentation. In this book a concept-oriented organization is followed for most concepts, with a language-based organization used to compare object-oriented features.

The text is divided into four parts:

- Part 1: Functions and Foundations* (Chapters 1–4)
- Part 2: Procedures, Types, Memory Management, and Control* (5–8)
- Part 3: Modularity, Abstraction and Object-Oriented Programming* (9–13)