Third Edition

Principles Of Programming Languages

Design, Evaluation, and Implementation

Bruce J. MacLennan
University of Tennessee, Knoxville

New York • Oxford
Oxford University Press
1999

To Gail and Kimberly

Oxford University Press

Oxford New York
Athens Auckland Bangkok Bogotá Buenos Aires Calcutta
Cape Town Chennai Dar es Salaam Delhi Florence Hong Kong Istanbul
Karachi Kuala Lumpur Madrid Melbourne Mexico City Mumbai
Nairobi Paris São Paulo Singapore Taipei Tokyo Toronto Warsaw

and associated companies in
Berlin Ibadan

Published by Oxford University Press, Inc.,
198 Madison Avenue, New York, New York, 10016
http://www.oup-usa.org

Library of Congress Cataloging-in-Publication Data
MacLennan, Bruce J.
Principles of programming languages : design, evaluation, and implementation / Bruce J. MacLennan. — 3rd ed.
p. cm.
Includes bibliographical references and index.
ISBN 0-19-511306-3 (cloth)
1. Programming languages (Electronic computers) I. Title.
QA76.7.M33 1999
005.13—dc21 98-27755
CIP

Printing (last digit): 9 8 7 6 5 4 3

Printed in the United States of America
on acid-free paper

Contents

Preface to the Third Edition

Another damned, thick, square book! Always scribble, scribble! Eh! Mr. Gibbon?
—Duke of Gloucester (1743–1805)

There is virtue in small books. There is also a disconcerting tendency for books to get longer with each succeeding edition. I have attempted to combat that tendency in this new third edition of *Principles of Programming Languages*.

In particular, I have resisted the temptation to discuss additional programming languages. The reason is that the languages are the means rather than the end. Every language I have included, in addition to being well known or important in its own right, is intended to illustrate a number of important language design principles. In the interest of economy I have attempted to use the minimum number of languages that cover the relevant topics. Any more would be superfluous.

The new edition of this text remains true to its original conception: a series of case studies to illustrate the principles of programming language design, evaluation, and implementation. To this end, I have attempted to include only those topics that support this goal and to exclude all that are extraneous. No doubt I have erred on both sides, but that has been my aim.

There are several advantages to concentrating on the enduring principles of programming language design rather than on the details of an assortment of fashionable languages. The advantage for the author is that the book is stable in the face of the waxing and waning popularity of the languages. More important, however, is the advantage for students, for this stability means that their knowledge will be useful throughout their careers, not just until the next programming language invention. Nevertheless, after much consideration I have added three design principles, since it has been my experience that their goals should be articulated clearly. Although they were implicit in previous editions, they are now explicit.

Part of the conception of this book is that programming languages can be understood only in their technological and societal context. In this respect, more changes to the book have been necessary, for computer science continues to evolve, and history is always written from the perspective of the present. In this way we refine our understanding of the past and relate it to our current concerns.

There are many ways to organize a programming language textbook, as dictated by the author's vision of the field. I am grateful to the teachers who have found my vision to be compatible with their own and have found this book to be a useful means to their ends. Over the years I have benefitted from the suggestions and criticisms of these teachers and their students, and so I hope this new edition will be an even better tool for the task.

The third edition also includes new exercises. As in the previous editions, some of them are simple tests of comprehension, while others (marked with an asterisk) are design or evaluation problems requiring more time and thought; exercises with two asterisks are major projects. Although I believe students will get more from the more complicated questions. I realize that they may be inappropriate in large classes, for which short-answer and mathematical exercises are more useful.

Thanks are also due to my wife Gail and daughter Kimberly for the patience with which they have so long endured my ranting about "the damned, thick book."

Preface

PURPOSE

The purpose of this book is to teach the skills required to design and implement programming languages. *Design* is an important topic for all computer science students regardless of whether they will ever have to create a programming language. The user who understands the motivation for various language facilities will be able to use them more intelligently. The compiler writer who understands the motivation for these facilities will be able to implement them more reasonably. *Implementation* is also an important topic since the language designer must be aware of the costs of the facilities provided. Both topics are important to all computer scientists because all computer scientists use languages and because there is an increasing number of language-like human interfaces (word processors, command languages, etc.) that require these skills in their development. Thus, this book treats the design and implementation of programming languages as fundamental skills that all computer scientists should possess.

All designers, whether architects, aeronautical engineers, electrical engineers, and so on, require *descriptive skills*, techniques, and notations for communicating their ideas to their clients, to other designers, and to implementers. The ability of descriptive tools to abstract important characteristics of a design and omit irrelevant details makes these tools valuable for *comparing* and *evaluating* designs. Thus, this book aims to teach the descriptive tools important to programming language design and implementation.

History is an important aspect of any design discipline. Often the designs used in the past can be understood only in their historical context. Also, it is important that the student be aware of the designs tried in the past and why they succeeded or failed. Thus, this book presents language facilities and styles in their historical context.

SCOPE

In a rapidly moving field such as computer science, there is a danger that any book will be out of date in a few years. Worse, there is the danger that a computer scientist's knowledge

will soon be out of date. It is therefore imperative that we teach principles of enduring value rather than technical details that are soon obsolete. As a result, the implementation techniques discussed in this book are seminal; they form the basis of techniques that are likely to be useful for a long time to come and that can be varied to achieve a wide range of goals. *Principles are emphasized more than details.*

However, this alone is not sufficient to prevent the explosion of knowledge. Rather than trying to present all of the important language styles, language facilities, descriptive tools, and implementation techniques, this book instructs the student in the creation of these things. In the long run this will be much more valuable, since most of the specific techniques we teach will become obsolete. The techniques become obsolete in part because of the activity of those who can create new techniques. *Methods are emphasized more than results.*

Experience with programming languages has shown that although the syntactic form of a language is important, the real effectiveness of a language is determined by its semantics. In other words, what we say is more important than exactly the way we say it. For this reason, this book compares and evaluates languages on the basis of what can be said in the language rather than the details of their syntax. By the same token, the implementation of semantics (i.e., the run-time organization) is stressed at the expense of the implementation of syntax (i.e., parsing). *Semantics is emphasized more than syntax.*

ORGANIZATION

There are two basic ways a programming language text can present the characteristics of a number of languages: horizontally and vertically. In a vertical organization, various language topics are treated one by one, tracing each through several languages. For example, one chapter may treat procedures, discussing their characteristics in FORTRAN, Algol, and Pascal. Another chapter may discuss scope in FORTRAN, Algol, and LISP, and so forth, for all the various facilities. One danger of a vertical organization is that it is apt to degenerate into a catalog of features.

A horizontal organization treats languages as wholes. For example, one chapter would discuss FORTRAN, covering procedures, scope, and other important characteristics. Another chapter would discuss Algol, another Pascal, and so on. A horizontal organization is used in this book because it facilitates discussing the interrelationships between the parts of a programming language. This often overlooked aspect of language design is the cause of many unforeseen complications. The horizontal approach is also necessary if languages are to be considered in their historical context.

The importance of historical context leads to another organizing principle, summarized as *ontogeny recapitulates phylogeny.* This principle means that if in the learning process, the student repeats in summary form the historical learning process in the computer science field, he or she will have a firmer grasp of the subject. This does not mean that the student must be exposed to every mistake and explore every dead end in the field of programming languages; ontogeny *recapitulates* phylogeny, it does not duplicate it. Therefore this book is organized around a stripped down, or pruned, history of programming languages that allows the student to see issues in their historical context and to appreciate the way languages evolve.

Each technical field has its own set of peculiar concepts and terminology; programming language design is no exception. Some of these concepts are of utmost importance; others are details of transient value. It is crucial that the language designer understand the important concepts fully, in all of their ramifications. For this to be the case, each concept must be seen and investigated several times in a number of different contexts. When students finally see a formal definition of a concept in its full generality they will understand its implications; it will not seem arbitrary, as definitions so often do. Thus, an inductive, or bottom-up, approach is used for presenting concepts and abstractions.

Conversely, a top-down approach is used for presenting the structures and facilities that are found in programming languages. The reason is that a programming language facility or feature is best understood in context, that is, in terms of its function with respect to the rest of the language and the goals of the language. Thus, structures and facilities are presented in their functional context. This will help the student to understand the language as a unified whole.

The result of these considerations is that the book as a whole is organized horizontally, and each chapter is organized vertically. That is, each language is analyzed into its major structural subsystems.

In all cases the goal has been to give the student a comprehensive understanding of the most important aspects of programming language design and implementation.

This book is intended for a one-semester course; for shorter courses some of the later chapters (such as Chapters 12 or 13) can be omitted if necessary. It is suggested that, no matter what else is skipped, time be reserved at the end of the course for discussing the content of Chapter 14, "Principles of Language Design."

Most of the problems are practical exercises in language design. They could form the basis for a separate language-design laboratory or for classroom discussion. In a few cases they are potential thesis topics.

In accord with the book's goal of emphasizing broad principles rather than details, no language is presented in full. The student will not find syntax charts for FORTRAN, Pascal, or Ada. Further, it is unlikely that students will be able to program in any of these languages solely on the basis of their description here. Instead, students will learn the fundamental concepts of programming languages, which will simplify their learning the details of whatever languages they may have to use.

To make this book more versatile, a concept directory has been included in addition to the more conventional index and table of contents. This is a vertically organized outline of topics in language design. With it students can find, for example, all of the sections describing parameter passing modes.

ACKNOWLEDGMENTS

Several generations of students in my introductory programming languages course at the Naval Postgraduate School have lived through early drafts of this book. Their comments, criticisms, and tolerance are gratefully acknowledged.

The late R. W. Hamming contributed substantially to the completion of this book through

his encouragement, insightful reviews of early drafts, and many discussions of the philosophy of book writing. I take this opportunity to acknowledge this debt publicly. Any errors or inadequacies are, of course, my own responsibility.

Most important, I thank my wife, Gail, without whose constant support this book might not have been completed.

Concept Directory

History, Motivation, and Evaluation

Note: Page numbers denote the entire subsection that begins on the specified page.

Design and Implementation

Principles

Implementation

0 INTRODUCTION

WHAT IS A PROGRAMMING LANGUAGE?

The subject of this book is programming languages, specifically, the principles for their design, evaluation, and implementation. Thus, we must begin by saying what a programming language is.

A programming language is a language intended for the description of programs. Often a program is expressed in a programming language so that it can be executed on a computer. This is not the only use of programming languages, however. They may also be used by people to describe programs to other people. Indeed, since much of a programmer's time is spent reading programs, the understandability of a programming language to people is often more important than its understandability to computers.

There are many languages that people use to control and interact with computers. These can all be referred to as *computer languages.* Many of these languages are used for special purposes, for example, for editing text, conducting transactions with a bank, or generating reports. These special-purpose languages are not *programming* languages because they cannot be used for general programming. We reserve the term *programming language* for a computer language that can be used, at least in principle, to express any computer program.[1] Thus, our final definition is

A programming language is a language that is intended for the expression of computer programs and that is capable of expressing any computer program.

[1] This is not a vague notion. There is a precise theoretical way of determining whether a computer language can be used to express any program, namely, by showing that it is equivalent to a universal Turing machine. This topic is outside the scope of this book.

THE DIFFERENCES AMONG PROGRAMMING LANGUAGES

Since, by definition, any programming language can be used to express any program, it follows that all programming languages are equally powerful—any program that can be written in one can also be written in another. Why, then, are there so many programming languages? And why should one study their differences, when in this very fundamental sense they are all the same? The reason is that, although it's *possible* to write any program in any programming language, it's not equally *easy* to do so. Thus, in this book, we will not be very concerned with the *theoretical* power of programming languages (they're all the same). Rather, we concentrate on their *practical* power, as real tools used by real people. In this regard they will be seen to differ in many important respects. But why devote so much time to just one kind of tool?

IMPORTANCE OF THE STUDY OF PROGRAMMING LANGUAGES

Programming languages are important for students in all disciplines of computer science because they are the primary tools of the central activity of computer science: programming. As a result, the progress of computer science can be traced in the progress of programming languages, and many issues of computer science manifest themselves as programming language issues. This is particularly true in programming methodology, where advances in languages and programming techniques have gone hand in hand. The reason is simple: Programming languages remain the central tool for problem solving in computer science.

INFLUENCE OF LANGUAGES ON PROBLEM SOLVING

The Sapir–Whorf hypothesis is a (still controversial) linguistic theory that states that the structure of language defines the boundaries of thought. Although there is no evidence that the use of a particular language will prevent us from thinking certain thoughts, it is the case that a given language can facilitate or impede certain modes of thought and that languages embody characteristic ways of dealing with the world and other people. When applied to programming languages, the analogous statement is that although no programming language can prevent us from finding certain solutions to a problem, a given language can influence the class of solutions we are likely to see and the frame of mind with which we approach programming, thus subtlely influencing the quality of our programs.

BENEFITS FOR ALL COMPUTER SCIENTISTS

The study of programming languages is important to anyone who uses them, that is, anyone who programs. The reason is that from this study you will learn the motivation for and the

use of the most important facilities found in modern programming languages. You will learn the benefits of these facilities, as well as their costs, by studying the techniques used to implement them. This will provide you with a basis for evaluating languages, which will aid you in choosing the best language for your application. The understanding acquired of the motivations for the facilities in a language will enable you to use those facilities to their fullest potential. The repertoire of language mechanisms with which you are familiar will have been increased so that even if the language you must use does not provide the facilities you need, you will be able to simulate them through your knowledge of their implementation.

There are many programming languages now in widespread use—many more than can be taught to you as part of your computer science education. This means that in your computer science career you will be required frequently to learn a new programming language and to put it to effective use. Your speed in learning new languages is one aspect of your versatility as a computer scientist. Fortunately, underneath the surface details most languages are very similar. Therefore, the study of programming languages, by increasing the range of facilities in which you are fluent, will enable you to see more that is familiar in any new language that you encounter. This will speed your learning of new languages.

BENEFITS FOR LANGUAGE DESIGNERS

Although, as indicated above, the study of programming languages is important to all students of computer science, it is especially important to certain disciplines. Obviously, it is important if you are a student of language design. All engineering design is a cumulative process; we learn from the successes and failures of the designs of the past. To this end it is necessary to be familiar with the history of programming languages. As George Santayana said, "Those who cannot remember the past are condemned to repeat it." An understanding of the reasons why certain designs have been tried in the past and later abandoned will help you to develop a sense of good language design and to become skillful in making design trade-offs. As R.W. Hamming has said, "We would know what they thought when they did it." To help you remember the lessons of the past, we have formulated and illustrated a number of *maxims or principles* of good programming language design. The central role these play has dictated the book's title: *Principles of Programming Languages.*

BENEFITS FOR LANGUAGE IMPLEMENTERS

If you are interested in language implementation, you will gain insight into the motivations for various language facilities, thus allowing you to make reasonable implementation trade-offs. Although language implementation is a complicated subject, requiring one or more courses, this book presents the most useful and important techniques for implementing a number of common programming language facilities. These are seminal techniques that can be elaborated to satisfy more stringent requirements or varied to solve related problems; they are a basis for further studies in language implementation.

BENEFITS FOR HARDWARE ARCHITECTS

By understanding the requirements of programming language implementation, hardware architects will gain insight into the ways machines may better support languages. More important, you will learn to design a semantically coherent machine—a machine with complete and coherent sets of data types and operations on those data types. The reason for this is simple. Just as a programming language can be considered a *virtual computer*, that is, a computer implemented in software, so a computer can be considered a programming language implemented in hardware. This view suggests that many of the principles of programming language design can be equally well applied to computer architecture, and indeed they can.

BENEFITS FOR SYSTEM DESIGNERS

Designers of all sorts of software systems (e.g., operating systems and database systems) will learn principles and techniques applicable to all human interfaces. Many software tools including operating system command languages, database systems, editors, text formatters, and debuggers, have many of the characteristics of programming languages, and so the principles you learn here will be applicable to much of your future software design. The study of both language design and implementation is obviously valuable here. Knowledge of programming languages is more directly necessary for designers of file systems, linkage editors, and other software that must interface with programming languages.

BENEFITS FOR SOFTWARE MANAGERS

Finally, if you manage software development efforts, then you will benefit in several ways from the study of programming languages. The project manager often makes decisions regarding the language to be used on a given project, or whether an existing language should be used or extended, or whether a completely new language should be designed. You will be better able to do this if you know what common languages can and cannot do, and if you know the current direction and state of the art of programming language research. You will be better able to make these decisions if you know the costs of designing or extending a language, the costs of implementing a language, and the benefits of various language facilities.

PLAN OF THE BOOK

In 1965 an American Mathematical Association Prospectus estimated that 1700 programming languages were then in use.[2] In the intervening years, many more have been invented.

[2] Quoted in P.J. Landin, "The Next 700 Programming Languages." *Commun. ACM 9*, 3 (March 1966), p. 157.

Clearly, it is impossible to discuss every language, or even a sizable fraction of them. How have we chosen the languages to present in this book?

Certainly, we have chosen languages of actual or potential importance. All other things being equal, we have chosen languages that you are likely to encounter in your career as a computer scientist. But there are other, more important factors in our selection.

An understanding of these factors is implicit in the purpose of this book—which is not to teach you to program in half a dozen programming languages.[3] As noted before, there is little chance that we could teach you just those languages you will later need to know. Rather, our goal is to present the most important principles for the design, evaluation, and implementation of programming languages. To this end, we have chosen languages that will serve as good *case studies* to illustrate these principles.

These principles have developed in a series of historical stages, each being a reaction to the perceived problems and opportunities discovered in the previous stage. For this reason, we have chosen programming languages that are illustrative of the major generations of programming language evolution. Thus, FORTRAN, Algol-60, Pascal, and Ada are representatives of the first, second, third, and fourth programming language generations.[4] We have picked these particular languages because they form a single evolutionary line in the family tree of programming languages.

Since language development is now entering the fifth generation, it is too early to predict what the next stage in programming language evolution will be. Therefore, we have illustrated the fifth generation with representatives of three important new programming paradigms: function-oriented programming (LISP), object-oriented programming (Smalltalk), and logic-oriented programming (PROLOG). It is likely that all three of these paradigms will be important in the years to come.

---

[3] Thus, there are few exercises in this book that require you to do significant programming in a language under study. Attempting to evaluate a language on the basis of writing a few short programs in it is as misleading as evaluating an automobile by driving it once or twice around the block. Meaningful evaluation requires experience with large programs maintained over long periods, which is impractical in a course such as this. As is often the case, if we are careful, we will learn more by vicarious experience than by direct experience.

[4] Note that some authors use the term *fourth-generation language* to refer to various application generation packages. These are not *programming* languages in the sense referred to previously; we are discussing fourth-generation *programming* languages.