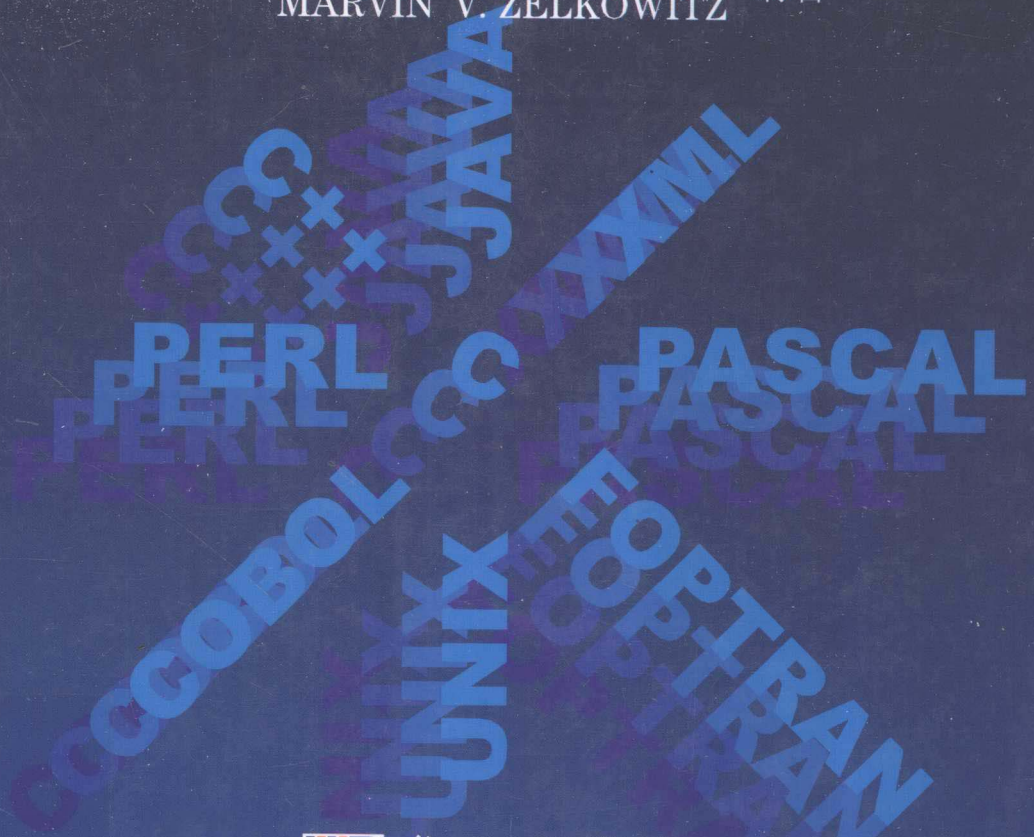
PROGRAMMING LANGUAGES

Design and Implementationen (Fourth edition)

编程语言

——设计与实现（影印版）

(美) TERRENCE W.PRATT
MARVIN V. ZELKOWITZ 编著



科学出版社
www.sciencep.com

编 程 语 言
——设计与实现
（影印版）

Programming Languages
Design and Implementation
Fourth Edition

Terrence W. Pratt
Marvin V. Zelkowitz
编著

科 学 出 版 社
北 京

图字：01-2003-6692 号

内 容 简 介

本书系统地讲述了编程语言，包括 C、C++、Java 和 PERL 等 11 种语言，内容包括编程语言简史、编程环境、编程语言语法、语言模型、基本数据类型、封装、继承、程序控制、子程序控制、存储管理、分布式处理和网络编程等。

本书的范例以多种编程语言表述，显示了编程技巧的通用性。本书内容丰富，适合专、本科学生和程序员使用。

ISBN 0-13-027678-2

图书在版编目(CIP)数据

编程语言：设计与实现=Programming Languages: Design and Implementation /（美） Pratt, T. W. 等著. —影印本. —北京：科学出版社，2004

ISBN 7-03-012473-1

Ⅰ. 编… Ⅱ. P… Ⅲ. 程序语言—程序设计—英文 Ⅳ. TP312

中国版本图书馆 CIP 数据核字（2003）第 099953 号

策划编辑：李佩乾/责任编辑：朱凤成
责任印制：吕春珉/封面设计：北新华文平面设计室

科学出版社 出版
北京东黄城根北街16号
邮政编码:100717
http://www.sciencep.com

双青印刷厂 印刷

科学出版社发行 各地新华书店经销
*
2004 年 1 月第 一 版 开本：787×960 1/16
2004 年 1 月第一次印刷 印张：41 1/2
印数：1—3 000 字数：664 000

定价：68.00 元

（如有印装质量问题，我社负责调换〈环伟〉）

Preface

This fourth edition of *Programming Languages: Design and Implementation* continues the tradition developed in the earlier editions to describe programming language design by means of the underlying software and hardware architecture that is required for execution of programs written in those languages. This provides the programmer with the ability to develop software that is both correct and efficient in execution. In this new edition, we continue this approach, as well as improve on the presentation of the underlying theory and formal models that form the basis for the decisions made in creating those languages.

Programming language design is still a very active pursuit in the computer science community as languages are born, age, and eventually die. This fourth edition represents the vital languages of the early 21^{st} century. Postscript, Java, HTML, and Perl have been added to the languages discussed in the third edition to reflect the growth of the World Wide Web as a programming domain. The discussion of Pascal, FORTRAN, and Ada has been deemphasized in recognition of these languages' aging in anticipation of possibly dropping them in future editions of this book.

At the University of Maryland, a course has been taught for the past 25 years that conforms to the structure of this book. For our junior-level course, we assume the student already knows C, Java, or C++ from earlier courses. We then emphasize Smalltalk, ML, Prolog, and LISP, as well as include further discussions of the implementation aspects of C++. The study of C++ furthers the students' knowledge of procedural languages with the addition of object-oriented classes, and the inclusion of LISP, Prolog, and ML provides for discussions of different programming paradigms. Replacement of one or two of these by FORTRAN, Ada, or Pascal would also be appropriate.

It is assumed that the reader is familiar with at least one procedural language, generally C, C++, Java, or FORTRAN. For those institutions using this book at a lower level, or for others wishing to review prerequisite material to provide a framework for discussing programming language design issues, Chapters 1 and 2 provide a review of material needed to understand later chapters. Chapter 1 is a

general introduction to programming languages, while Chapter 2 is a brief overview of the underlying hardware that will execute the given program.

The theme of this book is language design and implementation issues. Chapter 3, and 5 through 12 provide the basis for this course by describing the underlying grammatical model for programming languages and their compilers (Chapter 3), elementary data types (Chapter 5), data structures and encapsulation (Chapter 6), inheritance (Chapter 7), statements (Chapter 8), procedure invocation (Chapter 9), storage management (Chapter 10), distributed processing (Chapter 11) and network programming (Chapter 12), which form the central concerns in language design.

Chapter 4 is a more advanced chapter on language semantics that includes an introduction to program verification, denotational semantics, and the lambda calculus. It may be skipped in the typical sophomore- or junior-level course. As with the previous editions of this book, we include a comprehensive appendix that is a brief summary of the features in the 12 languages covered in some detail in this book.

The topics in this book cover the 12 knowledge units recommended by the 1991 ACM/IEEE Computer Society Joint Curriculum Task Force for the programming languages subject area [TUCKER et al. 1991].

Although compiler writing was at one time a central course in the computer science curriculum, there is increasing belief that not every computer science student needs to be able to develop a compiler; such technology should be left to the compiler specialist, and the hole in the schedule produced by deleting such a course might be better utilized with courses such as software engineering, database engineering, or other practical use of computer science technology. However, we believe that aspects of compiler design should be part of the background for all good programmers. Therefore, a focus of this book is how various language structures are compiled, and Chapter 3 provides a fairly complete summary of parsing issues.

The 12 chapters emphasize programming language examples in FORTRAN, Ada, C, Java, Pascal, ML, LISP, Perl, Postscript, Prolog, C++, and Smalltalk. Additional examples are given in HTML, PL/I, SNOBOL4, APL, BASIC, and COBOL as the need arises. The goal is to give examples from a wide variety of languages and let the instructor decide which languages to use as programming examples during the course.

Although discussing all of the languages briefly during the semester is appropriate, we do not suggest that the programming parts of this course consist of problems in each of these languages. We think that would be too superficial in one course. Ten programs, each written in a different language, would be quite a chore and would provide the student with little in-depth knowledge of any of these languages. We assume that each instructor will choose three or four languages and emphasize those.

All examples in this book, except for the most trivial, were tested on an appropriate translator; however, as we clearly point out in Section 1.3.3, correct execution on our local system is no guarantee that the translator is processing programs according to the language standard. We are sure that Mr. Murphy is at work here,

and some of the trivial examples may have errors. If so, we apologize for any problems that may cause.

To summarize, our goals in producing this fourth edition were as follows:

- Provide an overview of the key paradigms used in developing modern programming languages;
- Highlight several languages, which provide those features, in sufficient detail to permit programs to be written in each language demonstrating those features;
- Explore the implementation of each language in sufficient detail to provide the programmer with an understanding of the relationship between a source program and its execution behavior;
- Provide sufficient formal theory to show where programming language design fits within the general computer science research agenda;
- Provide a sufficient set of problems and alternative references to allow students the opportunity to extend their knowledge of this important topic.

We gratefully acknowledge the valuable comments received from the users of the third edition of this text and from the hundreds of students of CMSC 330 at the University of Maryland who provided valuable feedback on improving the presentation contained in this book.

Changes to the Fourth Edition. For users familiar with the third edition, the fourth edition has the following changes:

1. A chapter was added (Chapter 12) on the World Wide Web. Java was added as a major programming language, and an overview of HTML and Postscript were added to move the book away from the classical "FORTRAN number–crunching" view of compilers.

2. The material on object-oriented design was moved earlier in the text to indicate its major importance in software design today. In addition, numerous other changes were made by moving minor sections around to better organize the material into a more consistent presentation.

3. We have found that the detailed discussions of languages in Part II of the third edition were not as useful as we expected. A short history of each of the 12 languages was added to the chapter that best represents the major features of that language, and the language summaries in Part II of the third edition were shortened as the appendix. Despite these additions, the size of the book has not increased because we deleted some obsolete material.

Terry Pratt, Howardsville, Virginia

Marv Zelkowitz, College Park, Maryland

Contents

Chapter 1

Language Design Issues

Any notation for the description of algorithms and data structures may be termed a programming language; however, in this book we are mostly interested in those that are implemented on a computer. The sense in which a language may be implemented is considered in the next two chapters. In the remainder of this book, the design and implementation of the various components of a language are considered in detail. The goal is to look at language features, independent of any particular language, and give examples from a wide class of commonly used languages.

Throughout the book, we illustrate the application of these concepts in the design of 12 major programming languages and their dialects: Ada, C, C++, FORTRAN, Java, LISP, ML, Pascal, Perl, Postscript, Prolog, and Smalltalk. In addition, we also give brief summaries about other languages that have made an impact on the field. This list includes APL, BASIC, COBOL, Forth, PL/I, and SNOBOL4. Before approaching the general study of programming languages, however, it is worth understanding why there is value in such a study to a computer programmer.

1.1 WHY STUDY PROGRAMMING LANGUAGES?

Hundreds of different programming languages have been designed and implemented. Even in 1969, Sammet [SAMMET 1969] listed 120 that were fairly widely used, and many others have been developed since then. Most programmers, however, never venture to use more than a few languages, and many confine their programming entirely to one or two. In fact, practicing programmers often work at computer installations where use of a particular language such as Java, C, or Ada is required. What is to be gained, then, by study of a variety of different languages that one is unlikely ever to use?

There are excellent reasons for such a study, provided that you go beneath the superficial consideration of the "features" of languages and delve into the underlying design concepts and their effect on language implementation. Six primary reasons come immediately to mind:

1. *To improve your ability to develop effective algorithms.* Many languages provide features, that when used properly, are of benefit to the programmer but, when used improperly, may waste large amounts of computer time or lead the programmer into time-consuming logical errors. Even a programmer who has used a language for years may not understand all of its features. A typical example is *recursion*—a handy programming feature that, when properly used, allows the direct implementation of elegant and efficient algorithms. When used improperly, it may cause an astronomical increase in execution time. The programmer who knows nothing of the design questions and implementation difficulties that recursion implies is likely to shy away from this somewhat mysterious construct. However, a basic knowledge of its principles and implementation techniques allows the programmer to understand the relative cost of recursion in a particular language and from this understanding to determine whether its use is warranted in a particular programming situation. New programming methods are constantly being introduced in the literature. The best use of concepts like object-oriented programming, logic programming, or concurrent programming, for example, requires an understanding of languages that implement these concepts. New technology, such as the Internet and World Wide Web, change the nature of programming. How best to develop techniques applicable in these new environments depends on an understanding of languages.

2. *To improve your use of your existing programming language.* By understanding how features in your language are implemented, you greatly increase your ability to write efficient programs. For example, understanding how data such as arrays, strings, lists, or records are created and manipulated by your language, knowing the implementation details of recursion, or understanding how object classes are built allows you to build more efficient programs consisting of such components.

3. *To increase your vocabulary of useful programming constructs.* Language serves both as an aid and a constraint to thinking. People use language to express thoughts, but language also serves to structure how one thinks, to the extent that it is difficult to think in ways that allow no direct expression in words. Familiarity with a single programming language tends to have a similar constraining effect. In searching for data and program structures suitable to the solution of a problem, one tends to think only of structures that are immediately expressible in the languages with which one is familiar. By studying the constructs provided by a wide range of languages, a programmer increases his programming vocabulary. The understanding of implementation techniques is particularly important because, to use a construct while programming in a language that does not provide it directly, the programmer must provide an implementation of the construct in terms of the primitive elements actually provided by the language. For example, the subprogram control structure known as a *coroutine* is useful in many programs, but few

languages provide a coroutine feature directly. A C or FORTRAN programmer, however, may readily design a program to use a coroutine structure and then implement it as a C or a FORTRAN program if familiar with the coroutine concept and its implementation.

4. *To allow a better choice of programming language.* A knowledge of a variety of languages may allow the choice of just the right language for a particular project, thereby reducing the required coding effort. Applications requiring numerical calculations can be easily designed in languages like C, FORTRAN, or Ada. Developing applications useful in decision making, such as in artificial-intelligence applications, would be more easily written in LISP, ML, or Prolog. Internet applications are more readily designed using Perl and Java. Knowledge of the basic features of each language's strengths and weaknesses gives the programmer a broader choice of alternatives.

5. *To make it easier to learn a new language.* A linguist, through a deep understanding of the underlying structure of natural languages, often can learn a new foreign language more quickly and easily than struggling novices who understand little of the structure even of their native tongue. Similarly, a thorough knowledge of a variety of programming language constructs and implementation techniques allows the programmer to learn a new programming language more easily when the need arises.

6. *To make it easier to design a new language.* Few programmers ever think of themselves as language designers, yet many applications are really a form of programming language. A designer of a user interface for a large program such as a text editor, an operating system, or a graphics package must be concerned with many of the same issues that are present in the design of a general-purpose programming language. Many new languages are based on C or Pascal as implementation models. This aspect of program design is often simplified if the programmer is familiar with a variety of constructs and implementation methods from ordinary programming languages.

There is much more to the study of programming languages than simply a cursory look at their features. In fact, many similarities in features are deceiving. The same feature in two different languages may be implemented in two very different ways, and thus the two versions may differ greatly in the cost of use. For example, almost every language provides an addition operation as a primitive, but the cost of performing an addition in C, COBOL, or Smalltalk may vary by an order of magnitude.

In this book, numerous language constructs are discussed, accompanied in almost every case by one or more designs for the implementation of the construct on a conventional computer. However, no attempt has been made to be comprehensive in covering all possible implementation methods. The same language or construct,

if implemented on the reader's local computer, may differ radically in cost or detail of structure when different implementation techniques have been used or when the underlying computer hardware differs from the simple conventional structure assumed here.

1.2 A SHORT HISTORY OF PROGRAMMING LANGUAGES

Programming language designs and implementation methods have evolved continuously since the earliest high-level languages appeared in the 1950s. Of the 12 languages described in some detail, the first versions of FORTRAN and LISP were designed during the 1950s; Ada, C, Pascal, Prolog, and Smalltalk date from the 1970s; C++, ML, Perl, and Postscript date from the 1980s; and Java dates from the 1990s. In the 1960s and 1970s, new languages were often developed as part of major software development projects. When the U.S. Department of Defense did a survey as part of its background efforts in developing Ada in the 1970s, it found that over 500 languages were being used on various defense projects.

1.2.1 Development of Early Languages

We briefly summarize language development during the early days of computing, generally from the mid-1950s to the early 1970s. Later developments are covered in more detail as each new language is introduced later in this book.

Numerically based languages. Computer technology dates from the era just before World War II through the early 1940s. Determining ballistics trajectories by solving the differential equations of motion was the major role for computers during World War II, which led to them being called *electronic calculators*.

In the early 1950s, symbolic notations started to appear. Grace Hopper led a group at Univac to develop the A-0 language, and John Backus developed Speedcoding for the IBM 701. Both were designed to compile simple arithmetic expressions into executable machine language.

The real breakthrough occurred in 1957 when Backus managed a team to develop FORTRAN, or FORmula TRANslator. As with the earlier efforts, FORTRAN data were oriented around numerical calculations, but the goal was a full-fledged programming language including control structures, conditionals, and input and output statements. Because few believed that the resulting language could compete with hand-coded assembly language, every effort was put into efficient execution, and various statements were designed specifically for the IBM 704. Concepts like the three-way arithmetic branch of FORTRAN came directly from the hardware of the 704, and statements like *READ INPUT TAPE* seem quaint today. It wasn't very elegant, but in those days, little was known about *elegant* programming, and the language was fast for the given hardware.

FORTRAN was extremely successful and was the dominant language through