Título:

Autor: **Precio:** $1191.00

Editorial: **Año:** 1995

Tema: **Edición:** 1ª

Sinopsis **ISBN:** 9780805311914

This accessible new volume examines and evaluates the principles of programming languages from both their common and language-specific elements. Each chapter is devoted to a particular programming language issue. These issues are illustrated with an example from one of the many programming languages used today.
From the Inside Flap
This book stems in part from courses taught at the University of Kentucky and at the University of Wisconsin-Madison on programming language design. There are many good books that deal with the subject at an undergraduate level, but there are few that are suitable for a one semester graduate-level course. This book is my attempt to fill that gap.

The goal of this course, and hence of this book, is to expose first-year graduate students to a wide range of programming language paradigms and issues, so that they can understand the literature on programming languages and even conduct research in this field. It should improve the students' appreciation of the art of designing programming languages and, to a limited degree, their skill in programming.

This book does not focus on any one language, or even on a few languages; it mentions, at least in passing, over seventy languages, including well-known ones (Algol, Pascal, C, C++, LISP, Ada, FORTRAN), important but less known ones (ML, SR, Modula-3, SNOBOL), significant research languages (CLU, Alphard, Linda), and little-known languages with important concepts (Io, Go .. del). Several languages are discussed in some depth, primarily to reinforce particular programming paradigms. ML and LISP demonstrate functional programming, Smalltalk and C++ demonstrate object-oriented programming, and Prolog demonstrates logic programming.

Students are expected to have taken an undergraduate course in programming languages before using this book. The first chapter includes a review of much of the material on imperative programming languages that would be covered in such a course. This review makes the book self-contained, and also makes it accessible to advanced undergraduate students.

Most textbooks on programming languages cover the well-trodden areas of the field. In contrast, this book tries to go beyond the standard territory, making brief forays into regions that are under current research or that have been proposed and even rejected in the past. There are many fascinating constructs that appear in very few, if any, production programming languages. Some (like power loops) should most likely not be included in a programming language. Others (like Io continuations) are so strange that it is not clear how to program with them. Some (APL arrays) show alternative ways to structure languages. These unusual ideas are important even though they do not pass the test of current usage, because they elucidate important aspects of programming language design, and they allow students to evaluate novel concepts.

Certain themes flow through the entire book. One is the interplay between what can be done at compile time and what must be deferred to runtime. Actions performed at compile time make for more efficient and less error-prone execution. Decisions deferred until runtime lead to greater flexibility. Another theme is how patterns and pattern matching play a large role in many ways in programming languages. Pattern matching is immediately important for string manipulation, but it is also critical in steering logic programming, helpful for extracting data from structures in ML, and for associating caller and callee in CSP. A third theme is the quest for uniformity. It is very much like the mathematical urge to generalize. It can be seen in polymorphism, which generalizes the concept of type, and in overloading, which begins by unifying operators and functions and then unifies disparate functions under one roof. It can be seen in the homoiconic forms of LISP, in which program and data are both presented in the same uniform way.

Two organizing principles suggest themselves for a book on programming languages. The first is to deal separately with such issues as syntax, types, encapsulation, parallelism, object-oriented programming, pattern matching, dataflow, and so forth. Each section would introduce examples from all relevant languages. The other potential organizing principle is to present individual languages, more or less in full, and then to derive principles from them.

This book steers a middle course. I have divided it into chapters, each of which deals primarily with one of the subjects mentioned above. Most chapters include an extended example from a particular language to set the stage. This section may introduce language-specific features not directly relevant to the subject of the chapter. The chapter then introduces related features from other languages.

Because this book covers both central and unusual topics, the instructor of a course using the book should pick and choose whatever topics are of personal interest. In general, the latter parts of chapters delve into stranger and more novel variants of material presented earlier. The book is intended for a one semester course, but it is about 30 percent too long to cover fully in one semester. It is not necessary to cover every chapter, nor to cover every section of a chapter. Only Chapter 1 and the first seven sections of Chapter 3 are critical for understanding the other chapters. Some instructors will want to cover Chapter 4 before the discussion of ML in Chapter 3. Many instructors will decide to omit dataflow (Chapter 6). Others will wish to omit denotational semantics (in Chapter 10).

I have not described complete languages, and I may have failed to mention your favorite language. I have selected representative programming languages that display particular programming paradigms or language features clearly. These languages are not all generally available or even widely known. The appendix lists all the languages I have mentioned and gives you some pointers to the literature and to implementations and documentation available on the Internet through anonymous ftp (file-transfer protocol).

The exercises at the end of each chapter serve two purposes. First, they allow students to test their understanding of the subjects presented in the text by working exercises directly related to the material. More importantly, they push students beyond the confines of the material presented to consider new situations and to evaluate new proposals. Subjects that are only hinted at in the text are developed more thoroughly in this latter type of exercise.

In order to create an appearance of uniformity, I have chosen to modify the syntax of presented languages (in cases where the syntax is not the crucial issue), so that language-specific syntax does not obscure the other points that I am trying to make. For examples that do not depend on any particular language, I have invented what I hope will be clear notation. It is derived largely from Ada and some of its predecessors. This notation allows me to standardize the syntactic form of language, so that the syntax does not obscure the subject at hand. It is largely irrelevant whether a particular language uses begin and end or { and } . On the other hand, in those cases where I delve deeply into a language in current use (like ML, LISP, Prolog, Smalltalk, and C++), I have preserved the actual language. Where reserved words appear, I have placed them in bold monospace. Other program excerpts are in monospace font. I have also numbered examples so that instructors can refer to parts of them by line number. Each technical term that is introduced in the text is printed in boldface the first time it appears. All boldface entries are collected and defined in the glossary. I have tried to use a consistent nomenclature throughout the book.