Essentials of Programming Languages

second edition

Daniel P. Friedman

Mitchell Wand

Christopher T. Haynes

The MIT Press
Cambridge, Massachusetts
London, England

Typeset by the authors using $\LaTeX\,2_\varepsilon$.
Printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Information Data

Friedman, Daniel P.
Essentials of programming languages / Daniel P. Friedman, Mitchell Wand, Christopher T. Haynes—2nd ed.
p. cm.
Includes bibliographical references and index.
ISBN 0-262-06217-8 (hc. : alk. paper)
1. Programming Languages (Elecronic computers). I. Wand, Mitchell. II. Haynes, Christopher Thomas. III. Title.

QA76.7.F73 2001
005.13—dc21 00-135246

Contents

Foreword

This book brings you face-to-face with the most fundamental idea in computer programming:

The interpreter for a computer language is just another program.

It sounds obvious, doesn't it? But the implications are profound. If you are a computational theorist, the interpreter idea recalls Gödel's discovery of the limitations of formal logical systems, Turing's concept of a universal computer, and von Neumann's basic notion of the stored-program machine. If you are a programmer, mastering the idea of an interpreter is a source of great power. It provokes a real shift in mindset, a basic change in the way you think about programming.

I did a lot of programming before I learned about interpreters, and I produced some substantial programs. One of them, for example, was a large data-entry and information-retrieval system written in PL/I. When I implemented my system, I viewed PL/I as a fixed collection of rules established by some unapproachable group of language designers. I saw my job as not to modify these rules, or even to understand them deeply, but rather to pick through the (very) large manual, selecting this or that feature to use. The notion that there was some underlying structure to the way the language was organized, and that I might want to override some of the language designers' decisions, never occurred to me. I didn't know how to create embedded sublanguages to help organize my implementation, so the entire program seemed like a large, complex mosaic, where each piece had to be carefully shaped and fitted into place, rather than a cluster of languages, where the pieces could be flexibly combined. If you don't understand interpreters, you can still write programs; you can even be a competent programmer. But you can't be a master.

There are three reasons why as a programmer you should learn about interpreters.

First, you will need at some point to implement interpreters, perhaps not interpreters for full-blown general-purpose languages, but interpreters just the same. Almost every complex computer system with which people interact in flexible ways—a computer drawing tool or an information-retrieval system, for example—includes some sort of interpreter that structures the interaction. These programs may include complex individual operations—shading a region on the display screen, or performing a database search—but the interpreter is the glue that lets you combine individual operations into useful patterns. Can you use the result of one operation as the input to another operation? Can you name a sequence of operations? Is the name local or global? Can you parameterize a sequence of operations, and give names to its inputs? And so on. No matter how complex and polished the individual operations are, it is often the quality of the glue that most directly determines the power of the system. It's easy to find examples of programs with good individual operations, but lousy glue; looking back on it, I can see that my PL/I database program certainly had lousy glue.

Second, even programs that are not themselves interpreters have important interpreter-like pieces. Look inside a sophisticated computer-aided design system and you're likely to find a geometric recognition language, a graphics interpreter, a rule-based control interpreter, and an object-oriented language interpreter all working together. One of the most powerful ways to structure a complex program is as a collection of languages, each of which provides a different perspective, a different way of working with the program elements. Choosing the right kind of language for the right purpose, and understanding the implementation tradeoffs involved: that's what the study of interpreters is about.

The third reason for learning about interpreters is that programming techniques that explicitly involve the structure of language are becoming increasingly important. Today's concern with designing and manipulating class hierarchies in object-oriented systems is only one example of this trend. Perhaps this is an inevitable consequence of the fact that our programs are becoming increasingly complex—thinking more explicitly about languages may be our best tool for dealing with this complexity. Consider again the basic idea: the interpreter itself is just a program. But that program is written in some language, whose interpreter is itself just a program written in some language whose interpreter is itself .... Perhaps the whole distinction between program and programming language is a misleading idea, and future programmers will see themselves not as writing programs in particular, but as creating new languages for each new application.

Friedman, Wand, and Haynes have done a landmark job, and their book will change the landscape of programming-language courses. They don't just *tell* you about interpreters; they *show* them to you. The core of the book is a tour de force sequence of interpreters starting with an abstract high-level language and progressively making linguistic features explicit until we reach a state machine. You can actually run this code, study and modify it, and change the way these interpreters handle scoping, parameter-passing, control structure, etc.

Having used interpreters to study the execution of languages, the authors show how the same ideas can be used to analyze programs without running them. In two new chapters, they show how to implement type checkers and inferencers, and how these features interact in modern object-oriented languages.

Part of the reason for the appeal of this approach is that the authors have chosen a good tool—the Scheme language, which combines the uniform syntax and data-abstraction capabilities of Lisp with the lexical scoping and block structure of Algol. But a powerful tool becomes most powerful in the hands of masters. The sample interpreters in this book are outstanding models. Indeed, since they are *runnable* models, I'm sure that these interpreters and analyzers will find themselves at the cores of many programming systems over the coming years.

This is not an easy book. Mastery of interpreters does not come easily, and for good reason. The language designer is a further level removed from the end user than is the ordinary application programmer. In designing an application program, you think about the specific tasks to be performed, and consider what features to include. But in designing a language, you consider the various applications people might want to implement, and the ways in which they might implement them. Should your language have static or dynamic scope, or a mixture? Should it have inheritance? Should it pass parameters by reference or by value? Should continuations be explicit or implicit? It all depends on how you expect your language to be used, on which kinds of programs should be easy to write, and which you can afford to make more difficult.

Also, interpreters really *are* subtle programs. A simple change to a line of code in an interpreter can make an enormous difference in the behavior of the resulting language. Don't think that you can just skim these programs—very few people in the world can glance at a new interpreter and predict from that how it will behave even on relatively simple programs. So study these programs. Better yet, *run* them—this is working code. Try interpreting some simple expressions, then more complex ones. Add error messages. Modify the interpreters. Design your own variations. Try to really master these programs, not just get a vague feeling for how they work.

If you do this, you will change your view of your programming, and your view of yourself as a programmer. You'll come to see yourself as a designer of languages rather than only a user of languages, as a person who chooses the rules by which languages are put together, rather than only a follower of rules that other people have chosen.

Hal Abelson
Cambridge, MA
August, 2000

Preface

Goal

This book is an analytic study of programming languages. Our goal is to provide a deep, working understanding of the essential concepts of programming languages. These essentials have proved to be of enduring importance; they form a basis for understanding future developments in programming languages.

Most of these essentials relate to the semantics, or meaning, of program elements. Such meanings reflect how program elements are interpreted as the program executes. Programs called interpreters provide the most direct, executable expression of program semantics. They process a program by directly analyzing an abstract representation of the program text. We therefore choose interpreters as our primary vehicle for expressing the semantics of programming language elements.

The most interesting question about a program as object is, "What does it do?" The study of interpreters tells us this. Interpreters are critical because they reveal nuances of meaning, and are the direct path to more efficient compilation and to other kinds of program analyses.

Interpreters are also illustrative of a broad class of systems that transform information from one form to another based on syntax structure. Compilers, for example, transform programs into forms suitable for interpretation by hardware or virtual machines. Though general compilation techniques are beyond the scope of this book, we do develop several elementary program translation systems. These reflect forms of program analysis typical of compilation, such as control transformation, variable binding resolution, and type checking.

The following are some of the strategies that distinguish our approach.

1. Each new concept is explained through the use of a small language. These languages are often cumulative: later languages may rely on the features of earlier ones.

2. Language processors such as interpreters and type checkers are used to explain the behavior of programs in a given language. They express language design decisions in a manner that is both formal (unambiguous and complete) and executable.

3. When appropriate, we use interfaces and specifications to create data abstractions. In this way, we can change data representation without changing programs. We use this to investigate alternative implementation strategies.

4. Our language processors are written both at the very high level needed to produce a concise and comprehensible view of semantics and at the much lower level needed to understand implementation strategies.

5. We show how simple algebraic manipulation can be used to predict the behavior of programs and to derive their properties. In general, however, we make little use of mathematical notation, preferring instead to study the behavior of programs that constitute the implementations of our languages.

6. The text explains the key concepts, while the exercises explore alternative designs and other issues. For example, the text deals with static binding, but dynamic binding is discussed in the exercises. One thread of exercises applies the concept of lexical addressing to the various languages developed in the book.

We provide several views of programming languages using widely varying levels of abstraction. Frequently our interpreters provide a very high-level view that expresses language semantics in a very concise fashion, not far from that of formal mathematical semantics. At the other extreme, we demonstrate how programs may be transformed into a very low-level form characteristic of assembly language. By accomplishing this transformation in small stages, we maintain a clear connection between the high-level and low-level views.

Organization

The first two chapters provide the foundations for a careful study of programming languages. Chapter 1 emphasizes the connection between inductive data specification and recursive programming and introduces several notions related to the scope of variables. Chapter 2 introduces a data type facility. This leads to a discussion of data abstraction and examples of representational transformations of the sort used in subsequent chapters.

Chapter 3 uses these foundations to describe the behavior of programming languages. It introduces interpreters as mechanisms for explaining the run-time behavior of languages and develops an interpreter for a simple, lexically scoped language with first-class procedures, recursion, and assignment to variables. This interpreter is the basis for much of the material in the remainder of the book. The chapter then explores call-by-reference, call-by-need, and call-by-name parameter-passing mechanisms, and culminates with a sketch of an interpreter for an imperative language.

Chapter 4 extends the language of chapter 3 with type declarations. First we implement a type checker. Next we show how to use the types to enforce abstraction boundaries. Finally we show how the types in a program can be deduced by a unification-based type inference algorithm.

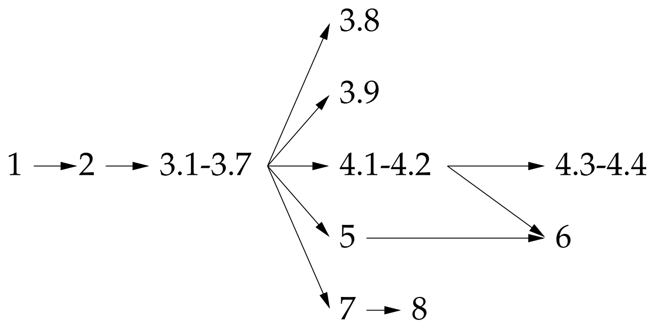
Chapter 5 presents the basic concepts of object-oriented languages, centered on classes (but ignoring types, which are deferred to chapter 6). We develop an efficient runtime architecture, which is used as the basis for the material in chapter 6.

Chapter 6 combines the ideas of the type checker of chapter 4 with those of the object-oriented language of chapter 5, leading to a conventional typed object-oriented language. This requires introducing new concepts including abstract classes, abstract methods, and casting.

Chapter 7 rewrites our basic interpreter in continuation-passing style. The control structure that is needed to run the interpreter thereby shifts from recursion to iteration. This exposes the control mechanisms of the interpreted language, and strengthens one's intuition for control issues in general. It also provides the means for extending the interpreter with exception-handling and multi-threading mechanisms. Finally, we use continuation-passing style to present logic programming.

Chapter 8 is the companion to the previous chapter. There we show how to transform our familiar interpreter into continuation-passing style; here we show how to accomplish this for a much larger class of programs. Continuation-passing style is a powerful programming tool, for it allows any sequential control mechanism to be implemented in almost any language. The algorithm is also a fine example of an abstractly specified source-to-source program transformation.

The dependencies of the various chapters are shown in the figure below.



Finally, appendix A describes our SLLGEN parsing system.

Usage

This material has been used in both undergraduate and graduate courses. In addition, it has been used in continuing education courses for professional programmers. We assume background in data structures and experience both in a procedural language such as C, C++, or Java, and in Scheme.

Exercises are a vital part of the text and are scattered throughout. They range in difficulty from being trivial if related material is understood [⋆], to requiring many hours of thought and programming work [⋆⋆⋆]. A great deal of material of applied, historical, and theoretical interest resides within them. We recommend that each exercise be read and some thought be given as to how to solve it. Although we write our program interpretation and transformation systems in Scheme, any language that supports both first-class procedures and assignment (ML, Common Lisp, etc.) is adequate for working the exercises.

Exercise 0.1 [$\star$] We often use phrases like "some languages have property *X*." For each such phrase, find one or more languages that have the property and one or more languages that do not have the property. Feel free to ferret out this information from any descriptive book on programming languages (say (Scott, 2000), (Sethi, 1996), or (Pratt & Zelkowitz, 1996)).

Exercise 0.2 [$\star$] Determine the rationale for the existence of index items, such as `cons-prim`, that do not appear in the body of the book.

This is a hands-on book: everything discussed in the book may be implemented within the limits of a typical university course. Because the abstraction facilities of functional programming languages are especially suited to this sort of programming, we can write substantial language-processing systems that are nevertheless compact enough that one can understand and manipulate them with reasonable effort.

The web site, available through the publisher, includes complete Scheme code for all of the interpreters and analyzers in this book. The code is as compliant with R^{5}RS (Kelsey *et al.*, 1998) as we could make it. The site includes pointers to several Scheme implementations (some of which are freely available) and compatibility files that should allow our code to run without change on these implementations or any Scheme implementation that is R^{5}RS-compliant.