Programming Languages: Application and Interpretation

Shriram Krishnamurthi
Brown University

This book is available free-of-cost from the author's Web site.
This version was generated on 2007-04-26.

Preface

The book is the textbook for the programming languages course at Brown University, which is taken primarily by third and fourth year undergraduates and beginning graduate (both MS and PhD) students. It seems very accessible to smart second year students too, and indeed those are some of my most successful students. The book has been used at over a dozen other universities as a primary or secondary text. The book's material is worth one undergraduate course worth of credit.

This book is the fruit of a vision for teaching programming languages by integrating the "two cultures" that have evolved in its pedagogy. One culture is based on interpreters, while the other emphasizes a survey of languages. Each approach has significant advantages but also huge drawbacks. The interpreter method writes *programs* to learn concepts, and has its heart the fundamental belief that by teaching the computer to execute a concept we more thoroughly learn it ourselves.

While this reasoning is internally consistent, it fails to recognize that understanding definitions does not imply we understand consequences of those definitions. For instance, the difference between strict and lazy evaluation, or between static and dynamic scope, is only a few lines of interpreter code, but the *consequences* of these choices is enormous. The survey of languages school is better suited to understand these consequences.

The text therefore melds these two approaches. Concretely, students program with a new set of features first, then try to distill those principles into an actual interpreter. This has the following benefits:

- By seeing the feature in the context of a real language, students can build something interesting with it first, so they understand that it isn't an entirely theoretical construct, and will actually *care* to build an interpreter for it. (Relatively few students are excited in interpreters for their own sake, and we have an obligation to appeal to the remainder too.)

- Students get at least fleeting exposure to multiple languages, which is an important educational attribute that is being crushed by the wide adoption of industrially fashionable languages. (Better still, by experimenting widely, they may come to appreciate that industrial fashions are just that, not the last word in technological progress.)

- Because they have already programmed with the feature, the explanations and discussions are much more interesting than when all students have seen is an abstract model.

- By first building a mental model for the feature through experience, students have a much better chance of actually discovering how the interpreter is supposed to work.

In short, many more humans learn by induction than by deduction, so a pedagogy that supports it is much more likely to succeed than one that suppresses it. The book currently reflects this design, though the survey parts are done better in lecture than in the book.

Separate from this vision is a goal. My goal is to not only teach students new material, but to also change the way they solve problems. I want to show students where languages come from, why we should regard languages as the ultimate form of abstraction, how to recognize such an evolving abstraction, and how to turn what they recognize into a language. The last section of the book, on domain-specific languages, is a growing step in this direction.

Design Principles

- Concepts like design, elegance and artistic sensibility are rarely manifest in computer science courses; in the name of not being judgmental, we may be running the risk of depriving our students of judgment itself. We should reverse this trend. Students must understand that artificial objects have their own aesthetic; the student must learn to debate the tradeoffs that lead to an aesthetic. Programming languages are some of the most thoroughly designed artifacts in computer science. Therefore, the study of programming languages offers a microcosm to study design itself.

- The best means we have to lead students to knowledge is through questions, not answers. The best education prepares them to assess new data by confronting it with questions, processing the responses, and iterating until they have formed a mental model of it. This book is therefore structured more like a discussion than a presentation. It leads the reader down wrong paths (so don't blindly copy code from it!). It allows readers to get comfortable with mistaken assumptions before breaking them down systematically.

- The programming languages course is one of the few places in the curriculum where we can tease out and correct our students' misconceptions about this material. They are often misled on topics such as efficiency and correctness. Therefore, material on compilation, type systems and memory management should directly confront their biases. For instance, a presentation of garbage collection that does not also discuss the trade-offs with manual memory management will fail to address the prejudices students bear.

Background and Prerequisite

This book assumes that students are comfortable reasoning informally about loop invariants, have modest mathematical maturity, and are familiar with the existence of the Halting Problem. At Brown, they have all been exposed to Java but not necessarily to any other languages (such as Scheme).

Supplementary Material

There is some material I use in my course that isn't (currently) in this book:

preparation in Scheme For the first week, I offer supplementary sessions that teach students Scheme. The material from these sessions is available from my course Web pages. In addition, I recommend the use of a simple introduction to Scheme, such as the early sections of *The Little Schemer* or of *How to Design Programs*.

domain-specific languages I discuss instances of real-world domain-specific languages, such as the access-control language XACML. Students find the concepts easy to grasp, and can see why the language is significant. In addition, it is one they may themselves encounter (or even decide to use) in their programming tasks.

garbage collection I have provided only limited notes on garbage collection because I feel no need to offer my own alternative to Paul Wilson's classic survey, *Uniprocessor Garbage Collection Techniques*. I recommend choosing sections from this survey, depending on student maturity, as a supplement to this text.

model checking I supplement the discussion of types with a presentation on model checking, to show students that it is possible to go past the fixed set of theorems of traditional type systems to systems that permit developers to state theorems of interest. I have a pre-prepared talk on this topic, and would be happy to share those slides.

Web programming Before plunging into continuations, I discuss Web programming APIs and demonstrate how they mask important control operators. I have a pre-prepared talk on this topic, and would be happy to share those slides. I also wrap up the section on continuations with a presentation on programming in the PLT Scheme Web server, which natively supports continuations.

articles on design I hand out a variety of articles on the topic of design. I've found Dan Ingalls's dissection of Smalltalk, Richard Gabriel's on Lisp, and Paul Graham's on both programming and design the most useful. Graham has now collected his essays in the book *Hackers and Painters*.

logic programming The notes on logic programming are the least complete. Students are already familiar with unification from type inference by the time I arrive at logic programming. Therefore, I focus on the implementation of backtracking. I devote one lecture to the use of unification, the implications of the occurs-check, depth-first versus breadth-first search, and tabling. In another lecture, I present the implementation of backtracking through continuations. Concretely, I use the presentation in Dorai Sitaram's *Teach Yourself Scheme in Fixnum Days*. This presentation consolidates two prior topics, continuations and macros.

Exercises

Numerous exercises are sprinkled throughout the book. Several more, in the form of homework assignments and exams, are available from my course's Web pages (where ⟨*year*⟩ is one of `2000`, `2001`, `2002`, `2003`, `2004` and `2005`):

`http://www.cs.brown.edu/courses/cs173/`⟨*year*⟩`/`

In particular, in the book I do *not* implement garbage collectors and type checkers. These are instead homework assignments, ones that students generally find extremely valuable (and very challenging!).

Programs

This book asks students to implement language features using a combination of interpreters and little compilers. All the programming is done in Scheme, which has the added benefit of making students fairly comfortable in a language and paradigm they may not have employed before. End-of-semester surveys reveal that students are far more likely to consider using Scheme for projects in other courses after taking this course than they were before it (even when they had prior exposure to Scheme).

Though every line of code in this book has been tested and is executable, I purposely do not distribute the code associated with this book. While executable code greatly enhances the study of programming languages, it can also detract if students execute the code mindlessly. I therefore ask you, Dear Reader, to please type in this code as if you were writing it, paying close attention to every line. You may be surprised by how much many of them have to say.

Course Schedule

The course follows approximately the following schedule:

Weeks	*Topics*
1	Introduction, Scheme tutorials, Modeling Languages
2–3	Substitution and Functions
3	Laziness
4	Recursion
4	Representation Choices
4–5	State
5–7	Continuations
7–8	Memory Management
8–10	Semantics and Types
11	Programming by Searching
11–12	Domain-Specific Languages and Metaprogramming

Miscellaneous “culture lecture” topics such as model checking, extensibility and future directions consume another week.

An Invitation

I think the material in these pages is some of the most beautiful in all of human knowledge, and I hope any poverty of presentation here doesn’t detract from it. Enjoy!

Acknowledgments

This book has a long and humbling provenance. The conceptual foundation for this interpreter-based approach traces back to seminal work by John McCarthy. My own introduction to it was through two texts I read as an undergraduate, the first editions of *The Structure and Interpretation of Computer Programs* by Abelson and Sussman with Sussman and *Essentials of Programming Languages* by Friedman, Wand and Haynes. Please read those magnificent books even if you never read this one.

My graduate teaching assistants, Dave Tucker and Rob Hunter, wrote and helped edit lecture notes that helped preserve continuity through iterations of my course. Greg Cooper has greatly influenced my thinking on lazy evaluation. Six generations of students at Brown have endured drafts of this book.

Bruce Duba, Corky Cartwright, Andrew Wright, Cormac Flanagan, Matthew Flatt and Robby Findler have all significantly improved my understanding of this material. Matthew and Robby's work on DrScheme has greatly enriched my course's pedagogy. Christian Queinnec and Paul Graunke inspired the presentation of continuations through Web programming, and Greg Cooper created the approach to garbage collection, both of which are an infinite improvement over prior approaches.

Alan Zaring, John Lacey and Kathi Fisler recognized that I might like this material and introduced me to it (over a decade ago) before it was distributed through regular channels. Dan Friedman generously gave of his time as I navigated *Essentials*. Eli Barzilay, John Clements, Robby Findler, John Fiskio-Lasseter, Kathi Fisler, Cormac Flanagan, Matthew Flatt, Suresh Jagannathan, Gregor Kiczales, Mira Mezini, Prabhakar Ragde, Marc Smith, and Éric Tanter have provided valuable feedback after using prior versions of this text.

The book's accompanying software has benefited from support by several generations of graduate assistants, especially Greg Cooper and Guillaume Marceau. Eli Barzilay and Matthew Flatt have also made excellent, creative contributions to it.

My chairs at Brown, Tom Dean and Eli Upfal, have permitted me to keep teaching my course so I could develop this book. I can't imagine how many course staffing nightmares they've endured, and ensuing temptations they've suppressed, in the process.

My greatest debt is to Matthias Felleisen. An early version of this text grew out of my transcript of his course at Rice University. That experience made me realize that even the perfection embodied in the books I admired could be improved upon. This result is not more perfect, simply different—its outlook shaped by standing on the shoulders of giants.

Thanks

Several more people have made suggestions, asked questions, and identified errors:

Ian Barland, Hrvoje Blazevic, Daniel Brown, Greg Buchholz, Lee Butterman, Richard Cobbe, Bruce Duba, Stéphane Ducasse, Marco Ferrante, Dan Friedman, Mike Gennert, Arjun Guha, Roberto Ierusalimschy, Steven Jenkins, Eric Koskinen, Neel Krishnaswami, Benjamin Landon, Usman Latif, Dan Licata, Alice Liu, Paulo Matos, Grant Miner, Ravi Mohan, Jason Orendorff, Klaus Ostermann, Pupeno [sic], Manos Renieris, Morten Rhiger, Bill Richter, Peter Rosenbeck, Amr Sabry, Francisco Solsona, Anton van Straaten, Andre van Tonder, Michael Tschantz, Phil Wadler, Joel Weinberger, and Greg Woodhouse.

In addition, Michael Greenberg instructed me in the rudiments of classifying flora.

Contents

Part I

Prelude