

Programming Languages: Application and Interpretation

Shriram Krishnamurthi
Brown University

Copyright © 2003, Shriram Krishnamurthi

This work is licensed under the
Creative Commons Attribution-NonCommercial-ShareAlike 3.0 United States License.
If you create a derivative work, please include the version information below in your attribution.

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

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:

<i>Weeks</i>	<i>Topics</i>
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 Buttermann, Richard Cobbe, Bruce Duba, Stéphane Ducasse, Marco Ferrante, Dan Friedman, Mike Gennert, Arjun Guha, Roberto Ierusalimsky, 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

Preface	iii
Acknowledgments	vii
I Prelude	1
1 Modeling Languages	3
1.1 Modeling Meaning	4
1.2 Modeling Syntax	5
1.3 A Primer on Parsers	6
1.4 Primus Inter Parsers	9
II Rudimentary Interpreters	11
2 Interpreting Arithmetic	13
3 Substitution	15
3.1 Defining Substitution	16
3.2 Calculating with <code>with</code>	20
3.3 The Scope of <code>with</code> Expressions	21
3.4 What Kind of Redundancy do Identifiers Eliminate?	23
3.5 Are Names Necessary?	24
4 An Introduction to Functions	27
4.1 Enriching the Language with Functions	27
4.2 The Scope of Substitution	29
4.3 The Scope of Function Definitions	30
5 Deferring Substitution	33
5.1 The Substitution Repository	34
5.2 Deferring Substitution Correctly	35

5.3	Fixing the Interpreter	36
6	First-Class Functions	41
6.1	A Taxonomy of Functions	41
6.2	Enriching the Language with Functions	42
6.3	Making <code>with</code> Redundant	45
6.4	Implementing Functions using Deferred Substitutions	45
6.5	Some Perspective on Scope	48
6.5.1	Filtering and Sorting Lists	48
6.5.2	Differentiation	49
6.5.3	Callbacks	50
6.6	Eagerness and Laziness	52
6.7	Standardizing Terminology	53
III	Laziness	57
7	Programming with Laziness	59
7.1	Haskell	59
7.1.1	Expressions and Definitions	59
7.1.2	Lists	61
7.1.3	Polymorphic Type Inference	62
7.1.4	Laziness	64
7.1.5	An Interpreter	68
7.2	Shell Scripting	69
8	Implementing Laziness	73
8.1	Implementing Laziness	73
8.2	Caching Computation	77
8.3	Caching Computations Safely	79
8.4	Scope and Evaluation Regimes	82
IV	Recursion	87
9	Understanding Recursion	89
9.1	A Recursion Construct	90
9.2	Environments for Recursion	91
9.3	An Environmental Hazard	95
10	Implementing Recursion	97

V	Intermezzo	103
11	Representation Choices	105
11.1	Representing Environments	105
11.2	Representing Numbers	106
11.3	Representing Functions	106
11.4	Types of Interpreters	107
11.5	Procedural Representation of Recursive Environments	109
VI	State	115
12	Church and State	117
13	Mutable Data Structures	119
13.1	Implementation Constraints	120
13.2	Insight	121
13.3	An Interpreter for Mutable Boxes	123
13.3.1	The Evaluation Pattern	124
13.3.2	The Interpreter	126
13.4	Scope versus Extent	129
14	Variables	133
14.1	Implementing Variables	134
14.2	Interaction Between Variables and Function Application	135
14.3	Perspective	137
VII	Continuations	145
15	Some Problems with Web Programs	147
16	The Structure of Web Programs	149
16.1	Explicating the Pending Computation	150
16.2	A Better Server Primitive	150
16.3	Testing Web Transformations	153
16.4	Executing Programs on a Traditional Server	154
17	More Web Transformation	157
17.1	Transforming Library and Recursive Code	157
17.2	Transforming Multiple Procedures	159
17.3	Transforming State	161
17.4	The Essence of the Transformation	162
17.5	Transforming Higher-Order Procedures	163

17.6	Perspective on the Web Transformation	166
18	Conversion into Continuation-Passing Style	169
18.1	The Transformation, Informally	169
18.2	The Transformation, Formally	172
18.3	Testing	175
19	Programming with Continuations	177
19.1	Capturing Continuations	178
19.2	Escapers	178
19.3	Exceptions	179
19.4	Web Programming	181
19.5	Producers and Consumers	181
19.6	A Better Producer	186
19.7	Why Continuations Matter	192
20	Implementing Continuations	193
20.1	Representing Continuations	193
20.2	Adding Continuations to the Language	196
20.3	On Stacks	199
20.4	Tail Calls	200
20.5	Testing	202
VIII	Memory Management	207
21	Automatic Memory Management	209
21.1	Motivation	209
21.2	Truth and Provability	211
IX	Semantics	215
22	Shrinking the Language	217
22.1	Encoding Lists	218
22.2	Encoding Boolean Constants and Operations	219
22.3	Encoding Numbers and Arithmetic	220
22.4	Eliminating Recursion	223
23	Semantics	231

X	Types	235
24	Introduction	237
24.1	What Are Types?	239
24.2	Type System Design Forces	240
24.3	Why Types?	240
25	Type Judgments	243
25.1	What They Are	243
25.2	How Type Judgments Work	246
26	Typing Control	249
26.1	Conditionals	249
26.2	Recursion	250
26.3	Termination	252
26.4	Typed Recursive Programming	253
27	Typing Data	255
27.1	Recursive Types	255
27.1.1	Declaring Recursive Types	255
27.1.2	Judgments for Recursive Types	256
27.1.3	Space for Datatype Variant Tags	258
28	Type Soundness	261
29	Explicit Polymorphism	265
29.1	Motivation	265
29.2	Solution	266
29.3	The Type Language	269
29.4	Evaluation Semantics and Efficiency	270
29.5	Perspective	271
30	Type Inference	273
30.1	Inferring Types	273
30.1.1	Example: Factorial	274
30.1.2	Example: Numeric-List Length	275
30.2	Formalizing Constraint Generation	276
30.3	Errors	277
30.4	Example: Using First-Class Functions	279
30.5	Solving Type Constraints	280
30.5.1	The Unification Algorithm	280
30.5.2	Example of Unification at Work	280
30.5.3	Parameterized Types	281
30.5.4	The “Occurs” Check	282

30.6 Underconstrained Systems	282
30.7 Principal Types	283
31 Implicit Polymorphism	285
31.1 The Problem	285
31.2 A Solution	286
31.3 A Better Solution	287
31.4 Recursion	288
31.5 A Significant Subtlety	288
31.6 Why Let and not Lambda?	289
31.7 The Structure of ML Programs	289
31.8 Interaction with Effects	290
XI Programming by Searching	291
32 Introduction	293
33 Programming in Prolog	295
33.1 Example: Academic Family Trees	295
33.2 Intermission	301
33.3 Example: Encoding Type Judgments	302
33.4 Final Credits	305
34 Implementing Prolog	307
34.1 Implementation	307
34.1.1 Searching	308
34.1.2 Satisfaction	308
34.1.3 Matching with Logic Variables	310
34.2 Subtleties and Compromises	311
34.3 Future Directions	311
XII Domain-Specific Languages and Metaprogramming	313
35 Domain-Specific Languages	315
35.1 Language Design Variables	315
35.2 Languages as Abstractions	315
35.3 Domain-Specific Languages	316
36 Macros as Compilers	319
36.1 Language Reuse	319
36.1.1 Example: Measuring Time	319
36.1.2 Example: Local Definitions	322

36.1.3	Example: Nested Local Definitions	323
36.1.4	Example: Simple Conditional	324
36.1.5	Example: Disjunction	325
36.2	Hygiene	327
36.3	More Macrology by Example	328
36.3.1	Loops with Named Iteration Identifiers	329
36.3.2	Overriding Hygiene: Loops with Implicit Iteration Identifiers	330
36.3.3	Combining the Pieces: A Loop for All Seasons	333
36.4	Comparison to Macros in C	334
36.5	Abuses of Macros	334
36.6	Uses of Macros	335
37	Macros and their Impact on Language Design	337
37.1	Language Design Philosophy	337
37.2	Example: Pattern Matching	338
37.3	Example: Automata	341
37.3.1	Concision	345
37.3.2	Efficiency	347
37.4	Other Uses	348
37.5	Perspective	348
XIII	What's Next?	349
38	Programming Interactive Systems	351
39	What Else is Next	355

Part I
Prelude

Thank You for previewing this eBook

You can read the full version of this eBook in different formats:

- HTML (Free /Available to everyone)
- PDF / TXT (Available to V.I.P. members. Free Standard members can access up to 5 PDF/TXT eBooks per month each month)
- Epub & Mobipocket (Exclusive to V.I.P. members)

To download this full book, simply select the format you desire below

