Preface

The implementation exercises, for all my frustration while doing them, are tremendously valuable. I find that actually implementing something like type inference or continuations greatly enhances my understanding of it, and testable programs are much easier to play with and build intuition about than are pages of equations.

From anonymous student evaluations

What should it mean to say you're learning about programming languages? Not that you'll see as many languages as possible, and not even that you'll see each major family of languages. Instead, you should master language-design ideas of lasting value. Learn what the great ideas are, in what guises they appear, how you can recognize them, and how you can use them. Great language-design ideas don't appear just in today's programming languages: they came from languages of the past, and they will continue to appear in languages of the future.

Unless you are interested in history, a *comprehensive* study of programming languages and programming-language features is not in your best interest. You have only so much time, and you should insist on learning the very best the field has to offer. You might learn something by enumerating programming languages; by classifying them as "functional," "logic," or "object-oriented"; and perhaps by discussing their "orthogonality," "simplicity," and "readability." But this kind of learning is best left behind, along with old arguments about syntactic forms, parameter-passing mechanisms, and control structures.

Part of the excitement of programming languages is that it's no longer just about designing languages. The field offers rigorous techniques for describing *all* computational processes, for analyzing language features, and for proving properties of programs. A serious introduction to the field must include *formal* modeling and analysis of languages and language features. Practice with formal tools will help you to see past superficial differences in programming languages, to recognize old ideas when they appear in new languages, to evaluate new programming languages, to choose and use programming languages intelligently, and to place the languages of the future into a context that will enable you to use them effectively. But unless you are mathematically inclined, a book that is organized around formalism first and foremost—and there are some great ones—is daunting.

In developing this book, I had three goals:

- To help you learn how to *use* different programming languages effectively
- To help you learn how to *describe* programming languages precisely
- To help you learn to understand and enjoy the *diversity* of programming languages

Because programming languages marry code and mathematics, I also want you to see some ideas about proof. But proving things is a secondary objective; this book is for practitioners.

The book presents case studies of individual languages. Each language is distilled down to a "micro-language" that illustrates essential ideas. Almost all the micro-languages are written in the same concrete syntax (S-expressions). As a result, you are less likely to be distracted by superficial differences in concrete syntax and more likely to understand essential differences in utility and behavior. Finally, and most important, each micro-language comes with a working interpreter. You learn by building: both by writing programs and by exploring and modifying the interpreters.

The book is founded on these principles:

Build, and learn by doing You will learn by building and modifying programs. Each chapter includes many exercises, and each set of exercises is accompanied by a short *guide*. If you are using the book on your own, or if you are using it to teach others, the guide will help you judge which exercises you will find most valuable.

Prove, to keep things simple and precise Although this book is not about proofs, you can get a little experience with proofs by doing some of the exercises in Chapter 2 (language metatheory) and Chapter 3 (equational reasoning). More importantly, the possibility of proof forces *me* to keep things simple and precise. I have described each language carefully, both formally and informally. I have made each language simple enough to be understood *completely*. Finally, I have carefully crafted and documented each interpreter—and the code is worth studying.

Compare, and find several ways to understand It's easier to learn new things if you can compare them with existing things and with each other. To help you learn syntax, for example, I present it in two forms: concrete and abstract. (Concrete syntax says how a language is written; it's something every programmer learns. Abstract syntax, which may be new to you, says what the underlying structure of a language is; it's the best way to think about what you can say in a language.) It will be easy for you to compare not just these two ways of writing one syntax but also to compare the syntaxes of *different* languages. Because each new micro-language uses new syntax only when it is needed to express a new feature, features that are found in multiple micro-languages aren't just based on the same ideas; they also look the same when written down.

As another example, you can learn about the meanings of language constructs by comparing an interpreter with a simple operational semantics (the "big-step" or "natural" variety). It is easier to learn about interpretation and operational semantics together than to learn about each separately.

You can compare example programs both large and small; you'll start by comparing example programs written in a single language, but I hope you will also compare examples written in different languages. From the particular examples, you will learn general concepts: you will learn about recursion by seeing many recursive functions, about higher-order functions by seeing them used in μScheme, about polymorphism by writing programs in three polymorphic languages, and so on.

Finally, doing the exercises will help you develop a deeper understanding of how a semantics is constructed, how an interpreter works, and most important, how to write programs. Each of these avenues to learning reinforces the others, and you can emphasize the one best suited for you.

Overview

If you work through this book, especially the exercises, you will emerge with a deep understanding of a handful of great ideas: functions, types, objects, inference rules, and garbage collection. You will also develop some essential technique: abstract syntax, operational semantics, equational reasoning, type systems, and continuations. But almost nobody understands a great idea by saying "today I will study objects and inheritance." Using this book, you will learn by building programs. (You may also prove the occasional theorem.)

Deep understanding and great ideas emerge from study of concrete designs and from solving concrete problems. I have organized the book around concrete, specific language designs, each of which comes with exercises. Each chapter either presents a new language or presents a new implementation technique in the context of a language presented previously. The presentation of each language includes its complete context-free syntax (easy because the languages are highly simplified), its abstract syntax, its operational semantics, an interpreter, and of course, sample code. You can learn something by reading the chapters, but if you intend to master anything, you must do some exercises. Even if you find them frustrating.

Implementation exercises require a language in which to write implementations. Six of the micro-languages in this book—μScheme, μML, μHaskell, μClu, μSmalltalk, and μProlog[1]—are not only small enough to be studied formally and mastered completely but also big enough that you can write interesting programs.

I have also defined three other languages which are intended primarily for conveying ideas, not for programming. Impcore and Typed Impcore introduce operational semantics and type systems, but neither is expressive enough for writing really interesting programs. Typed μScheme is very expressive, but because you have to instantiate every polymorphic value explicitly at every use, writing even small programs in Typed μScheme quickly becomes tiresome.

Every micro-language is implemented by an interpreter. An interpreter is the best way I know of to help you learn what is going on in the abstract world of operational semantics and type systems—in each chapter, you can see connections between mathematical descriptions of language ideas and code that implements those same ideas. An interpreter also makes it possible for you to learn language design in an eye-opening way—instead of just studying other people's language designs, you can create your own. Whether your own design explores a variation on one of my designs or goes in a completely different direction, the opportunity to try out new design ideas yourself, and to program with the results, will give you a feel for the problems of language design that you can't get just by studying existing languages. Besides, why should other people have all the fun?

One of my most difficult decisions was choosing the language in which the interpreters themselves would be implemented. I decided that if you want to learn programming languages from a standing start, the best choice is not one language but two. Through Chapter 4, the interpreters are written in C. C is well suited to the implementation of garbage collectors, which are the topic of Chapter 4, and I hope you are one of the many programmers already familiar with C or C++. If not, Kernighan and Ritchie (1988) will teach you. The remaining interpreters are written in Standard ML. If you are not familiar with ML, it is a powerful language that is ideally suited to writing interpreters, and there are several fine, open-source implementations available. With the help of the material in Chapter 3, learning ML will be easy.

[1]These micro-languages are distilled from languages whose greatness is widely acknowledged. For example, the designers behind Scheme, ML, Clu, and Smalltalk have all won ACM Turing Awards—the highest professional honor a computer scientist can receive.

This book is narrow and deep. I have deliberately excluded some important topics:

- Modules, units, mixins, and other structures that enable separate compilation and programming at scale are essential ideas in programming languages, but this part of the field is much less mature than the great ideas covered in this book, and I have not identified designs that I am confident will still be important in 20 years.
- Concurrency and parallelism play an important and growing role in the design and implementation of programming languages, but these topics are too difficult (and at present, too ramified) to be handled well in an introductory book. Concurrency brings with it enough new mathematics, new language constructs, and new programming technique that it needs a book of its own.

Organization

Each language chapter has roughly the same structure:

- I introduce the language, its history, and its influence.
- I discuss the basic ideas of the language; I present its syntax and semantics; and I give simple examples.
- I present the abstract syntax and values of the language, usually together with a representation we can use in an interpreter.
- Where practical, I present a formal, operational semantics of the language.
- I present, in detail, an interpreter for the language, and I connect the interpreter to the semantics. In the main text, I present only interesting parts of the interpreters; uninteresting parts are relegated to appendices.

  Also relegated to appendices is a topic of some intellectual interest but of little importance in a book about programming languages: parsing. In the main text, I assume that all programs have been parsed into abstract-syntax trees, and our code uses abstract syntax, not concrete syntax. Abstract syntax is not just a useful representation for interpreters; it is the correct way to think about the syntactic structure of languages.
- I use the language in one or more larger examples.
- For enrichment, I present aspects of the parent language as it is in real life, including its true concrete syntax and interesting language features not included in our micro-version.
- I conclude by summarizing the chapter, suggesting material for further study, and providing a glossary of terms commonly associated with the language.
- Each chapter ends with exercises, which are divided in three parts: programs in the language, changes to or properties of the semantics, and modifications to the interpreter. Each chapter also includes a guide to the exercises; the guide explains the idea behind each exercise, and it will help you choose which exercises you want to do.

Some chapters have additional sections exploring concepts pertinent to that language or describing different languages based on similar concepts.

In three chapters, I focus not on a language but on an implementation technique. Chapter 4 extends the interpreter of Chapter 3 by adding garbage collection. Chapter 5 introduces ML by providing a new implementation, written in ML, of the language described in Chapter 3. Chapter 6 presents type checkers for typed variants of the languages described in Chapters 2 and 3.

All the interpreters are available in machine-readable form. Moreover, the same machine-readable form is used to present the example code shown in the text, using the Noweb tool for literate programming. This tool guarantees that what is printed on paper is consistent with the running code. Noweb has also made it possible to extract all examples marked ⟨*transcript*⟩ and to check their accuracy mechanically.

Solutions to the exercises, including garbage collectors, are available to qualified instructors. For details, please see `http://www.cs.tufts.edu/~nr/build-prove-compare`.

Acknowledgments

This book is inspired by Sam Kamin's 1990 book *Programming Languages: An Interpreter-Based approach*. Sam gave me his blessing and encouragement, and he contributed the section on Prolog and logic programming, as well as a substantial number of exercises and examples. I owe him a great debt.

Russ Cox helped bootstrap the early chapters, especially the C code. His work was supported by an Innovation Grant from the Dean for Undergraduate Education at Harvard.

Matthew Fluet and I spent two days working nearly nonstop on the design and type theory of μClu. Matthew's insights and oversight were invaluable; without him, Chapter 9 would never have been revised. I'm also grateful to several members of IFIP Working Group 2.8, especially Stephanie Weirich, who helped me identify type-theoretic techniques for handling data abstraction.

David Chase suggested the debugging technique described in Section 4.6.

Ben Weitzman found the loophole exploited in Exercise 14 on page 279.

Christian Lindig wrote, in Objective Caml, a prettyprinter from which I derived the prettyprinter in Appendix L.

Sam Guyer helped me articulate my thoughts on why we study programming languages.

Dan Grossman read an early version of the manuscript, and he not only commented on every detail but also made me think hard about what I was doing. Jeremy Condit, Ralph Corderoy, Allyn Dimock, Lee Feigenbaum, Luiz de Figueiredo, Tony Hosking, Scott Johnson, and Juergen Kahrs also reviewed parts of the manuscript. Gregory Price suggested ways to improve the wording of several problems. Penny Anderson, Jon Berry, Richard Borie, Allyn Dimock, William Harrison, David Hemmendinger, Joel Jones, Giampiero Pecelli, and Jan Vitek bravely used preliminary versions in their classes. Penny found far more errors and suggested many more improvements than anyone else; she has my profound thanks.

My students, who are too numerous to mention by name, found many errors in earlier drafts. Each was paid one dollar per error, from which an elite minority earned enough to pay for their books.

Mike Hamburg and Inna Zakharevich spurred me to improve the concrete syntax of μSmalltalk and to provide better error messages.

Cory Kerens, despite spending many hours alone while I was visiting the land of ideas, cheered me on.

Norman Ramsey
Medford, Massachusetts
October, 2013