Preface

This textbook, suitable for an undergraduate or master's-level course in programming languages, is about great language-design ideas, how to describe them precisely, and how to use them effectively. The ideas revolve around functions, types, modules, and objects. They are described using formal semantics and type theory, and their use is illustrated through programming examples and exercises.

The ideas, descriptive techniques, and examples are conveyed by means of *bridge languages*. A bridge language models a real programming language, but it is small enough to describe formally and to learn in a week or two, yet big enough to write interesting programs in. The bridge languages in this book model Algol, Scheme, ML, CLU, and Smalltalk, and they are related to many modern descendants, including C, C++, OCaml, Haskell, Java, JavaScript, Python, Ruby, and Rust.

Each bridge language is supported by an interpreter, which runs all the examples and supports programming exercises. The interpreters, which are presented in depth in an online Supplement, are carefully crafted and documented. They can be used not only for exercises but also to implement students' own language-design ideas.

The book develops these concepts:

- Abstract syntax and operational semantics
- Definitional interpreters
- Algebraic laws and equational reasoning
- Garbage collection
- Symbolic computing and functional programming
- Parametric polymorphism
- Monomorphic and polymorphic type systems
- Type inference
- Algebraic data types and pattern matching
- Data abstraction using abstract types and modules
- Data abstraction using objects and classes

The concepts are supported by the bridge languages as shown in the Introduction (Table I.2, page 5), which also explains each bridge language in greater detail (pages 3 to 7).

The book calls for skills in both programming and proof:

- As prerequisites, learners should have the first-year, two-semester programming sequence, including data structures, plus discrete mathematics.
- To extend and modify the implementations in Chapters 1 to 4, a learner needs to be able to read and modify C code; the necessary skills have to be learned elsewhere. C is used because it is the simplest way to express programs that work extensively with pointers and memory, which is the topic of Chapter 4.

To extend and modify the implementations in Chapters 5 to 10, a learner needs to be able to read and modify Standard ML code; the necessary skills are developed in Chapters 2, 5, and 8. Standard ML is used because it is a simple, powerful language that is ideally suited to writing interpreters.

- To prove the simpler theorems, a learner needs to be able to substitute equals for equals and to fill in templates of logical reasoning. To prove the more interesting theorems, a learner needs to be able to write a proof by induction.

Designing a course to use this book

Some books capture a single course, and when using such a book, your only choice is to start at the beginning and go as far as you can. But in programming languages, instructors have many good choices, and a book shouldn't make them all for you. This book is designed so you can choose what to teach and what to emphasize while retaining a coherent point of view: how programming languages can be used effectively in practice. If you're relatively new to teaching programming languages and are not sure what to choose, you can't go wrong with a course on functions, types, and objects (Chapters 1, 2, 6, 7, and 10). If you have more experience, consider the ideas below.

Programming Languages: Build, Prove, and Compare gives you interesting, powerful programming languages that share a common syntax, a common theoretical framework, and a common implementation framework. These frameworks support programming practice in the bridge languages, implementation and extension of the bridge languages, and formal reasoning about the bridge languages. The design of your course will depend on how you wish to balance these elements.

- To unlock the full potential of the subject, combine programming practice with theoretical study and work on interpreters. If your students have only two semesters of programming experience and no functional programming, you can focus on the core foundations in Chapters 1 to 3: operational semantics, functional programming, and control operators. You can supplement that work with one of two foundational tracks: If your students are comfortable with C and pointers, they can implement continuation primitives in μScheme+, and they can implement garbage collectors. Or if they can make a transition from μScheme to Standard ML, with help from Chapter 8, they can implement type checkers and possibly type inference.

  If your students have an additional semester of programming experience or if they have already been exposed to functional programming, your course can advance into types, modules, and objects. When I teach a course like this, it begins with four homework assignments that span an introduction to the framework, operational semantics, recursive functions, and higher-order functions. After completing these assignments, my students learn Standard ML, in which they implement first a type checker, then type inference. This schedule leaves a week for programming with modules and data abstraction, a couple of weeks for Smalltalk, and a bit of time for the lambda calculus.

  A colleague whose students are similarly experienced begins with Impcore and μScheme, transitions to Standard ML to work on type systems and type inference, then returns to the bridge languages to explore μSmalltalk, μProlog, and garbage collection.

  If your students have seen interpreters and are comfortable with proof by induction, your course can move much more quickly through the founda-

tional material, creating room for other topics. When I taught a course like this, it explored everything in my other class, then added garbage collection, denotational semantics, and logic programming.

- A second design strategy tilts your class toward programming practice, either de-emphasizing or eliminating theory. To introduce programming practice in diverse languages, *Build, Prove, and Compare* occupies a sweet spot between two extremes. One extreme "covers" N languages in N weeks. This extreme is great for exposure, but not for depth—when students must work with real implementations of real languages, a week or even two may be enough to motivate them, but it's not enough to build proficiency.

  The other extreme goes into full languages narrowly but deeply. Students typically use a couple of popular languages, and overheads are high: each language has its own implementation conventions, and students must manage the gratuitous details and differences that popular languages make inevitable.

  Build, Prove, and Compare offers both breadth and depth, without the overhead. If you want to focus on programming practice, you can aim for "four languages in ten weeks": μScheme, μML, Molecule, and μSmalltalk. You can bring your students up to speed on the common syntactic, semantic, and implementation frameworks using Impcore, and that knowledge will support them through to the next four languages. If you have a couple of extra weeks, you can deepen your students' experience by having them work with the interpreters.

- A third design strategy tilts your class toward applied theory. *Build, Prove, and Compare* is not suitable for a class in pure theory—the bridge languages are too big, the reasoning is informal, and the classic results are missing. But it is suitable for a course that is primarily about using formal notation to explain precisely what is going on in whole programming languages, reinforced by experience implementing that notation. Your students can do metatheory with Impcore, Typed Impcore, Typed μScheme, and nano-ML; equational reasoning with μScheme; and type systems with Typed Impcore, Typed μScheme, nano-ML, μML, and Molecule. They can compare how universally quantified types are used in three different designs (Typed μScheme, nano-ML/μML, and Molecule).

- What about a course in interpreters? If you are interested in *definitional* interpreters, *Build, Prove, and Compare* presents many well-crafted examples. And the online Supplement presents a powerful infrastructure that your students can use to build more definitional interpreters (Appendices F to I). But apart from this infrastructure, the book does not discuss what a definitional interpreter is or how to design one. For a course on interpreters, you would probably want an additional book.

All of these potential designs are well supported by the exercises (345 in total), which fall into three big categories. For insight into how to use programming languages effectively, there are programming exercises that use the bridge languages. For insight into the workings of the languages themselves, as well as the formalism that describes them, there are programming exercises that extend or modify the interpreters. And for insight into formal description and proof, there are theory exercises. Model solutions for some of the more challenging exercises are available to instructors.

A few exercises are simple enough and easy enough that your students can work on them for 10 to 20 minutes in class. But most are intended as homework.

- To introduce a new language like Impcore, μScheme, μML, Molecule, or μSmalltalk, think about assigning from a half dozen to a full dozen programming exercises, most easy, some of medium difficulty.
- To introduce proof technique, think about assigning around a half dozen proof problems, maybe one or two involving some form of induction (some metatheory, or perhaps an algebraic law involving lists).
- To develop a deep understanding of a single topic, assign one exercise or a group of related exercises aimed at that topic. Such exercises are provided for continuations, garbage collection, type checking, type inference, search trees, and arbitrary-precision integers.

Contents and scope

Because this book is organized by language, its scope is partly determined by what the bridge languages do and do not offer relative to the originals on which they are based.

- μScheme offers `define`, a `lambda`, and three "let" forms. Values include symbols, machine integers, Booleans, `cons` cells, and functions. There's no numeric tower and there are no macros.
- μML offers type inference, algebraic data types, and pattern matching. There are no modules, no exceptions, no mutable reference cells, and no value restriction.
- Molecule offers a procedural, monomorphic core language with mutable algebraic types, coupled to a module language that resembles OCaml and Standard ML.
- μSmalltalk offers a pure object-oriented language in which everything is an object; even classes are objects. Control flow is expressed via message passing in a form of continuation-passing style. μSmalltalk provides a modest class hierarchy, which includes blocks, Booleans, collections, magnitudes, and three kinds of numbers. And μSmalltalk includes just enough reflection to enable programmers to add new methods to existing classes.

Each of the languages supports multiple topical themes; the major themes are programming, semantics, and types.

- *Idiomatic programming* demonstrates effective use of proven features that are found in many languages. Such features include functions (μScheme, Chapter 2), algebraic data types (μML, Chapter 8), abstract data types and modules (Molecule, Chapter 9), and objects (μSmalltalk, Chapter 10).
- *Big-step semantics* expresses the meaning of programs in a way that is easily connected to interpreters, and which, with practice, becomes easy to read and write. Big-step semantics are given for Impcore, μScheme, nano-ML, μML, and μSmalltalk (Chapters 1, 2, 7, 8, and 10).
- *Type systems* guide the construction of correct programs, help document functions, and guarantee that language features like polymorphism and data abstraction are used safely. Type systems are given for Typed Impcore, Typed μScheme, nano-ML, μML, and Molecule (Chapters 6 to 9).

In addition the major themes and the concepts listed above, the book addresses, to varying degrees, these other concepts:

- Subtype polymorphism, in μSmalltalk (Chapter 10)
- Light metatheory for both operational semantics and type systems (Chapters 1, 5, and 6)
- Free variables, bound variables, variable capture, and substitution, in both terms and types (Chapters 2, 5, and 6)
- Continuations for backtracking search, for small-step semantics, and for more general control flow (Chapters 2, 3, and 10)
- The propositions-as-types principle, albeit briefly (Chapter 6 Afterword)

A book is characterized not only by what it includes but also by what it omits. To start, this book omits the classic theory results such as type soundness and strong normalization; although learners can prove some simple theorems and look for interesting counterexamples, theory is used primarily to express and communicate ideas, not to establish facts. The book also omits lambda calculus, because lambda calculus is not suitable for programming.

The book omits concurrency and parallelism. These subjects are too difficult and too ramified to be handled well in a broad introductory book.

And for reasons of space and time, the book omits three engaging programming models. One is the pure, lazy language, as exemplified by Haskell. Another is the prototype-based object-oriented language, made popular by JavaScript, but brilliantly illustrated by Self. The third is logic programming, as exemplified by Prolog—although Prolog is explored at length in the Supplement (Appendix D). If you are interested in μHaskell, μSelf, or μProlog, please write to me.

Software and other supplements

The software described in the book is available from the book's web site, which is `build-prove-compare.net`. The web site also provides a "playground" that allows you to experiment with the interpreters directly in your browser, without having to download anything. And it holds the book's PDF Supplement, which includes additional material on multiprecision arithmetic, extensions to algebraic data types, logic programming, and longer programming examples. The Supplement also describes all the code: both the reusable modules and the interpreter-specific modules.