Chapter 1

Introduction

The idea of this book is to learn about programming languages both by programming in them and by studying interpreters for them. For anyone interested in programming languages, experimenting with and comparing similar implementations of different languages is a huge help. But using full implementations of real programming languages doesn't make sense; even if we had the resources to build them, such implementations would be too large and complex to understand. Accordingly, we work with tiny programming languages that I hope capture the spirit of their full-blown cousins. The role of these "micro-languages" is not to help us study entire designs, but to illuminate some of the most essential features you can expect to see repeatedly throughout your career, e.g., first-class functions, objects with inheritance, and static type systems. Putting these features into micro-languages, instead of whole languages, makes them easier for you to learn. This section gives you an overview of all the micro-languages, and thereby an overview of the book.

Impcore (Chapter 2) Impcore includes the imperative constructs that are found at the core of "mainstream" programming languages: loops, conditionals, procedures, and mutable variables. The purpose of Impcore is not to introduce new language features, but rather to introduce the way we think about language features. We use Impcore to understand the central ideas of abstract syntax and operational semantics, and to introduce the software used to build the interpreters.

μScheme (Chapters 3 and 5) μScheme introduces two new ways of programming. It introduces *lists*, a recursive datatype. When processing lists, the natural control structure is recursion, not iteration. This seemingly trivial change has far-reaching effects on programming style. μScheme also introduces *first-class functions*, which are treated as values and can be stored in data structures and passed to or returned from functions. Functions that accept or return functions are called *higher-order functions*, and their use leads to a concise, powerful, and distinctive programming style: *functional programming*.

Garbage collection (Chapter 4) Chapter 4 extends μScheme with automatic memory management through garbage collection. Garbage collection enables programs written in Scheme and other safe languages to allocate new memory as needed, without worrying about where memory comes from or when it is no longer needed. Garbage collection simplifies both programming and interface design, and it has become a hallmark of civilized programming. In Chapter 4, you can implement both mark-scan and copying garbage collectors for μScheme.

Typed Impcore (Chapter 6) Typed Impcore is a statically typed version of Impcore. It illustrates type systems and the implementation of type checkers.

Typed μScheme (Section 6.6) Typed μScheme combines the power of functional programming with a static type system that is more expressive than any in common use. This type system, which type theorists call System F, is the starting point for understanding and expressing polymorphic type systems in real languages, from the simple Hindley-Milner types of Standard ML through the complexities of features like Haskell type classes or Java generics. Typed μScheme will help you learn the difficulties of designing polymorphic languages, and it will give you the opportunity to implement a complete, sophisticated type checker.

μML (Chapter 7) μML, which is derived from μScheme, shows how to substitute type inference for type checking, making things easier for the programmer while still guaranteeing safety. Type inference helps make code shorter, simpler, and more reusable, offering much of the convenience of dynamically typed languages while keeping all the protection of statically typed languages. In Chapter 7, you can implement type inference for yourself.

μHaskell (Chapter 8) μHaskell is a *pure*, *lazy* version of μML. Purity means you can apply all the laws of mathematics—including the algebraic laws of Chapter 3—to a Haskell program, without needing such caveats as "these laws hold only if the program does not print." To ensure that input and output are also subject to algebraic laws, μHaskell uses *monadic I/O*. Laziness means that an expression is evaluated only when its value is needed to show something on the output. Laziness enables feats of function composition that are not possible in a strict language. In Chapter 8, I also add algebraic data types and `case` expressions.

Clu (Chapter 9) Clu demonstrates the ideas that make large systems possible: data abstraction and information hiding. Data abstraction is enforced by compile-time type checking; like Typed μScheme and ML, Clu uses a polymorphic type system. By comparing these three languages, you will get a good idea of the design space of statically typed, polymorphic languages.

Chapter 9 needs major revision; I am working on an appropriate formal model and type system for μClu.

μSmalltalk (Chapter 10) μSmalltalk demonstrates object-orientation in a dynamically typed setting. Unlike hybrid languages such as Ada 95, Java, C#, C++, Modula-3, and Objective C, Smalltalk is *purely* object-oriented: every value is an object, and message-passing is the basic unit of control flow. Any message can be sent to any object. Object types can *inherit* state and implementation from other object types, which enables new forms of code reuse. Although the mechanisms are relatively simple, they offer extraordinary expressive power. In Chapter 10, you'll see how easy it is to harness that power to implement families of related abstractions, including collections and numbers. You can extend the families yourself, for example by implementing integers whose magnitude is limited only by the amount of memory you have available.

μProlog (Chapter 11) Prolog is a simple language inspired by principles of mathematical logic. Its interest lies not in its data types but in its wildly different evaluation model, which uses *backtracking* and *unification*. μProlog uses the Edinburgh Prolog syntax, rather than the Scheme syntax of the first edition; it also includes the cut.

Chapter 11 develops an elegant and compact implementation of μProlog using continuation-passing style. It also discusses *mechanical theorem proving*, to which Prolog traces its origins.

These micro-languages cover a wealth of programming styles and paradigms, and they illustrate some of the greatest ideas in programming: functions, types, and objects. You'll see the syntax and semantics of each micro-language, as well as examples of its use. You'll also see an interpreter that you can use to run programs in the language. These languages are the distinctive feature of this book, and their presentation is intended to help you learn not just to think intelligently about different programming languages, but to develop some intuition about what it is like to *use* different programming languages, and by working with the implementations, to develop a deep understanding of how languages work and what makes them possible. I hope you enjoy the experience.