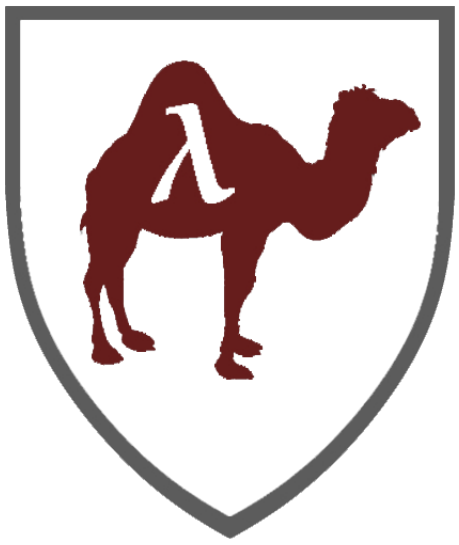
STUART M. SHIEBER

PROGRAMMING WELL: ABSTRACTION AND DESIGN IN COMPUTATION



CI Build: 89-1eee1c9 (Mon Jan 13 22:22:06 UTC 2025)

Commit 1eee1c9 from Mon Jan 13 17:08:39 2025 -0500 by CS51 Bot.

Contents

Preface

This book began as the notes for Computer Science 51, a second semester course in programming at Harvard College, which follows the legendary CS50 course that ably introduces some half of all Harvard undergraduate students to computer programming, and in its online HarvardX version CS50x has benefited hundreds of thousands of other students.

Students just learning to program, like those in CS50, typically view the end product of programming as a program that works – that "gets the right answer". Once such a program is in hand, the student thinks, the programmer's job is done. This book was developed to move students past this view of programming, to focus on programming *well*, regarding programming not as a transaction but as an art and a craft.

The book emphasizes the role of abstraction and abstraction mechanisms in engendering a design space in which good programs can be constructed. These abstraction mechanisms are associated with and enable the major programming paradigms – first- and higher-order functional programming, structure-driven programming, generic programming, modular programming, imperative programming, procedural programming, lazy programming, object-oriented programming, and concurrent programming. By expanding the student's armamentarium of abstraction mechanisms, this design space grows as well, making possible programs that are better along multiple dimensions – readability, maintainability, succinctness, efficiency, testability, and, most importantly but ineffably, beauty.

The programming edicts:

- *Edict of intention:* Make your intentions clear.
- *Edict of irredundancy:* Never write the same code twice.
- *Edict of decomposition:* Carve software at its joints.
- *Edict of prevention:* Make the illegal inexpressible.
- *Edict of compartmentalization:* Limit information to those with a need to know.

Aims

In developing the book, I had in mind several aims.

Explicit presentation of general principles. I introduce a small set of very general software engineering principles – presented as "edicts" in the text – and make frequent reference to them throughout the text to tie together more particular software engineering ideas.

I emphasize other general principles, such as the separation of concepts and paradigms from languages, and programming as art and craft, not a science.

Use of formal methods and notations. Facility with notation is the essence of mathematical maturity, and a strong correlate to computational thinking. I explicitly motivate the use of formal notation, and introduce notations for many of the core ideas in the book – syntax, semantics, complexity – both to emphasize rigorous thinking and to provide practice in handling notations. Use of this kind of notation is ubiquitous in computer science (Guy Steele has referred to this kind of notation, which he calls "computer science metanotation", as "the most popular programming language in computer science") though it is rarely introduced explicitly. For that reason alone, an introductory presentation of these notations is valuable for the early computer science student.

Provenance of ideas. Rather than presenting computational ideas or techniques as disconnected from history, I emphasize the provenance of these ideas, highlighting the role of real people in their development and promulgation and providing acculturation into some of the intellectual history of computer science. Special attention is given wherever appropriate to the role of women in developing the ideas.

Emphasis on reliable methods. Emphasis is placed on using modern methods for generating reliable programs by having the computer take on much of the work, in particular, strong static typing (and the polymorphic type inference that makes it practical), unit testing, and compartmentalization.

Pedagogical structure. The textbook contains a variety of components in keeping with its pedagogical goals.

- My intention is for the text to be self-contained. Little background is assumed beyond basic programming of the sort learned in a first-semester programming course. Any mathematical ideas that arise in examples or assignments are explained in an appendix.
- Code examples in the text are often developed step-wise, rather than being presented as whole and complete, reflecting how code is typically constructed. Similarly, examples are often revisited as new concepts are introduced that can be used to implement the examples in novel ways.

- The text is tightly connected to a series of pedagogical activities for students. Throughout the text, exercises test understanding of the just presented material; solutions to the exercises, often with extensive further explanations and descriptions of alternatives, are available in an appendix. Supplementary materials tightly connected with the book include labs, problem sets, and a project. Labs, intended to be done individually or synchronously in pairs or groups, provide a series of small and carefully graduated problems that build up practice with the programming concepts introduced in the texts. Lab solutions, again providing alternatives and cross-references to previous and upcoming discussions, are provided. Problem sets provide for more open-ended work on larger-scale but still self-contained problems, and relate to topical issues such as public-key encryption, symbolic math, artificial intelligence search, music composition, and epidemic simulation. The culmination is a project implementing a small run-time-typed subset of OCaml, synthesizing ideas from throughout the book, especially the presentations of formal syntax and semantics.

Openness. The text and related materials are intended to be openly available, allowing widespread adoption, including in venues, like MOOCs, where closed materials aren't appropriate.

Use of OCaml

It is typical in courses that introduce multiple programming paradigms to introduce different programming languages geared towards one or another of the paradigms. This language profligacy has the effect of dramatically increasing the amount of language syntax that needs to be introduced and misleadingly implies that the paradigms are coincident with or require different languages. By contrast, I make use of a single well-designed and well-supported language, OCaml, whose relatively simple core allows development and exposition of all of these paradigms and the abstraction mechanisms they rely on. OCaml is introduced and used not for its own sake but as a vehicle for conveying the wide range of programming and computational concepts.

OCaml is an ideal language for pedagogical purposes for the following reasons:

Simple core. The language is designed based on a relatively simple core set of orthogonal constructs, which are extended via syntactic sugar. This spareness means that students can get to the level of implementing an interpreter for a nontrivial subset of the language

by the end of the book.

Clean semantics. The language has quite clean semantics, which aids understanding.

Type discipline. Programs are strongly statically typed, so that students are confronted from the start with thinking in terms of always and only using values consistently with their types. Experience with reasoning about the types of expressions can inform better programming practice even when programming later in languages with weaker type systems or dynamic typing.

Multi-paradigm. Although the core of the language is relatively spare, built on top of the core is syntactic support for multiple paradigms including functional, modular, imperative, lazy, and object-oriented programming.

Nonproprietary. The language is supported by an open-source, non-proprietary, cross-platform toolset.

The primary disadvantage of using OCaml is that the language is little known and not widely used in the software industry. It is generally viewed as an "academic language", of interest to computer scientists rather than mainstream software developers. Nonetheless, the general approach of strongly statically typed languages based on a functional foundation is gaining currency through languages like F#, Reason, Rust, and Elm. More importantly, the goal of the textbook is not to teach a particular language so as to improve employability; rather, it is to teach a range of programming concepts that will be of use whatever language one programs in.

Limitations

The book is intentionally limited in certain ways.

- It does not cover the OCaml language exhaustively, and does not serve as a language reference. This is in keeping with the use of OCaml as a vehicle for presenting concepts. Just enough OCaml is presented to make possible the implementations of the presented concepts. (Cf. Minsky et al.'s *Real World OCaml*.)
- It does not cover formal proofs of correctness (though there is limited and informal discussion of invariants). The importance of correct code is highlighted in a focus on unit testing. (Indeed, a recurring thematic example is the building up of a simple unit testing framework for OCaml.)

- There is no coverage of interactive systems, graphics, or user interface design and implementation. (Cf. Stein's text *Interactive Programming In Java*.)
- No large application examples are given in their entirety. (Cf. the Whitington or Cousineau texts.) However, the problem sets provide opportunity for working with larger-scale examples.

Acknowledgements

The nature of the course – introducing a wide range of programming abstractions and paradigms with an eye toward developing a large design space of programs, using functional programming as its base and OCaml as the delivery vehicle – is shared with similar courses at a number of colleges. The instructors in those courses have for many years informally shared ideas, examples, problems, and notes in an open and free-flowing manner. When I took over the course from Greg Morrisett (now Dean and Vice Provost of Cornell Tech), I became the beneficiary of all of this collaboration, including source materials from these courses – handouts, notes, lecture material, and problem sets – which have been influential in the structure and design of these notes, and portions of which have thereby inevitably become intermixed with my own contributions in a way that would be impossible to disentangle. I owe a debt of gratitude to all of the faculty who have been engaged in this informal sharing, especially,

- Dan Grossman, University of Washington
- Michael Hicks, University of Maryland
- Greg Morrisett, Cornell University
- Benjamin Pierce, University of Pennsylvania
- David Walker, Princeton University
- Stephanie Weirich, University of Pennsylvania
- Steve Zdancewic, University of Pennsylvania

All of these faculty have kindly agreed to allow their contributions to be used here and distributed openly.

In addition, the course and this text have benefited immensely from the large crew of teaching staff of CS51 throughout the years. These include the head teaching fellows(list goes here tbd) as well as Sam Green and Serina Hu for help developing the `caml-tex` system that allows running the code examples as part of the typesetting process.