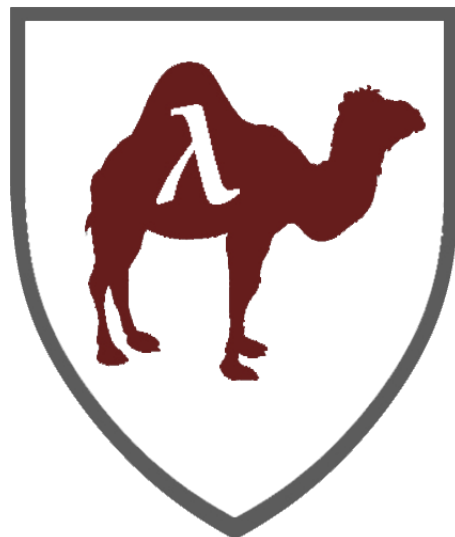


STUART M. SHIEBER

PROGRAMMING
WELL:
ABSTRACTION AND
DESIGN IN
COMPUTATION



©2025 Stuart M. Shieber. All rights reserved for the time being, though the intention is for this document to eventually be licensed under a CC license. In the meantime, please do not cite, quote, or redistribute.

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	13
1 Introduction	19
1.1 An extended example: greatest common divisor	21
1.2 Programming as design	24
1.3 The OCaml programming language	26
1.4 Tools and skills for design	28
2 A Cook's tour of OCaml	29
3 Expressions and the linguistics of programming languages	31
3.1 Specifying syntactic structure with rules	31
3.2 Disambiguating ambiguous expressions	34
3.3 Abstract and concrete syntax	36
3.4 Expressing your intentions	37
3.4.1 Commenting	38
4 Values and types	41
4.1 OCaml expressions have values	41
4.1.1 Integer values and expressions	41
4.1.2 Floating point values and expressions	42
4.1.3 Character and string values	43
4.1.4 Truth values and expressions	43
4.2 OCaml expressions have types	44
4.2.1 Type expressions and typings	46
4.3 The unit type	48
4.4 Functions are themselves values	48
5 Naming and scope	51
5.1 Variables are names for values	51
5.2 The type of a let-bound variable can be inferred	52
5.3 let expressions are expressions	52
5.4 Naming to avoid duplication	53
5.5 Scope	55

5.6	Global naming and top-level let	57
6	Functions	59
6.1	Function application	60
6.2	Multiple arguments and currying	61
6.3	Defining anonymous functions	62
6.4	Named functions	63
6.4.1	Compact function definitions	64
6.4.2	Providing typings for function arguments and outputs	65
6.5	Function abstraction and irredundancy	67
6.6	Defining recursive functions	69
6.7	Unit testing	72
6.8	Supplementary material	76
7	Structured data and composite types	77
7.1	Tuples	77
7.2	Pattern matching for decomposing data structures . . .	79
7.2.1	Advanced pattern matching	82
7.3	Lists	83
7.3.1	Some useful list functions	85
7.4	Records	90
7.4.1	Field selection	92
7.5	Comparative summary	92
8	Higher-order functions and functional programming	95
8.1	The map abstraction	95
8.2	Partial application	97
8.3	The fold abstraction	100
8.4	The filter abstraction	102
8.5	Problem section: Credit card numbers and the Luhn check	103
8.6	Supplementary material	105
9	Polymorphism and generic programming	107
9.1	Polymorphism	108
9.2	Polymorphic map	109
9.3	Regaining explicit types	110
9.4	The List library	112
9.5	Problem section: Function composition	113
9.6	Weak type variables	114
9.7	Supplementary material	115
10	Handling anomalous conditions	117

10.1	A non-solution: Error values	118
10.2	Option types	119
10.2.1	Option poisoning	121
10.3	Exceptions	122
10.3.1	Handling exceptions	125
10.3.2	Zippping lists	126
10.3.3	Declaring new exceptions	130
10.4	Options or exceptions?	131
10.5	Unit testing with exceptions	132
10.6	Supplementary material	134
11	Algebraic data types	137
11.1	Built-in composite types as algebraic types	139
11.2	Example: Boolean document search	140
11.3	Example: Dictionaries	146
11.4	Example: Arithmetic expressions	149
11.5	Problem section: Binary trees	151
11.6	Supplementary material	153
12	Abstract data types and modular programming	155
12.1	Modules	158
12.2	A queue module	159
12.3	Signatures hide extra components	162
12.4	Modules with polymorphic components	165
12.5	Abstract data types and programming for change	166
12.5.1	A string set module	169
12.5.2	A generic set signature	172
12.5.3	A generic set implementation	176
12.6	A dictionary module	181
12.7	Alternative methods for defining signatures and modules	185
12.7.1	Set and dictionary modules	186
12.8	Library Modules	188
12.9	Problem section: Image manipulation	189
12.10	Problem section: An abstract data type for intervals	190
12.11	Problem section: Mobiles	191
12.12	Supplementary material	194
13	Semantics: The substitution model	195
13.1	Semantics of arithmetic expressions	197
13.2	Semantics of local naming	201
13.3	Defining substitution	204
13.3.1	A problem with variable scope	204
13.3.2	Free and bound occurrences of variables	205
13.3.3	Handling variable scope properly	206

13.4	Implementing a substitution semantics	207
13.4.1	Implementing substitution	208
13.4.2	Implementing evaluation	209
13.5	Problem section: Semantics of booleans and conditionals	212
13.6	Semantics of function application	212
13.6.1	More on capturing free variables	214
13.7	Substitution semantics of recursion	218
13.8	Supplementary material	221
14	Efficiency, complexity, and recurrences	223
14.1	The need for an abstract notion of efficiency	224
14.2	Two sorting functions	225
14.3	Empirical efficiency	227
14.4	Big- O notation	229
14.4.1	Informal function notation	231
14.4.2	Useful properties of O	232
14.4.3	Big- O as the metric of relative growth	233
14.5	Recurrence equations	234
14.5.1	Solving recurrences by unfolding	236
14.5.2	Complexity of reversing a list	237
14.5.3	Complexity of reversing a list with accumulator	239
14.5.4	Complexity of inserting in a sorted list	240
14.5.5	Complexity of insertion sort	241
14.5.6	Complexity of merging lists	242
14.5.7	Complexity of splitting lists	243
14.5.8	Complexity of divide and conquer algorithms	243
14.5.9	Complexity of mergesort	244
14.5.10	Basic Recurrence patterns	245
14.6	Problem section: Complexity of the Luhn check	246
14.7	Supplementary material	246
15	Mutable state and imperative programming	247
15.1	References	249
15.1.1	Reference operator types	250
15.1.2	Boxes and arrows	251
15.1.3	References and pointers	252
15.2	Other primitive mutable data types	254
15.2.1	Mutable record fields	254
15.2.2	Arrays	255
15.3	References and mutation	255
15.4	Mutable lists	258
15.5	Imperative queues	260
15.5.1	Method 1: List references	262
15.5.2	Method 2: Two stacks	262

15.5.3	Method 3: Mutable lists	264
15.6	Hash tables	266
15.7	Conclusion	270
15.8	Supplementary material	270
16	Loops and procedural programming	271
16.1	Loops require impurity	272
16.2	Recursion versus iteration	273
16.2.1	Saving stack space	273
16.2.2	Tail recursion	274
16.3	Saving data structure space	275
16.3.1	Problem section: Metering allocations	276
16.3.2	Reusing space through mutable data structures	277
16.4	In-place sorting	278
16.5	Supplementary material	283
17	Infinite data structures and lazy programming	285
17.1	Delaying computation	285
17.2	Streams	287
17.2.1	Operations on streams	288
17.3	Lazy recomputation and thunks	291
17.3.1	The Lazy Module	293
17.4	Application: Approximating π	294
17.5	Problem section: Circuits and boolean streams	296
17.6	A unit testing framework	297
17.7	A brief history of laziness	301
17.8	Supplementary material	302
18	Extension and object-oriented programming	303
18.1	Drawing graphical elements	304
18.2	Objects introduced	308
18.3	Object-oriented terminology and syntax	311
18.4	Inheritance	313
18.4.1	Overriding	315
18.5	Subtyping	316
18.6	Problem section: Object-oriented counters	319
18.7	Supplementary material	320
19	Semantics: The environment model	321
19.1	Review of substitution semantics	321
19.2	Environment semantics	322
19.2.1	Dynamic environment semantics	323
19.2.2	Lexical environment semantics	330
19.3	Conditionals and booleans	331

19.4	Recursion	332
19.5	Implementing environment semantics	334
19.6	Semantics of mutable storage	335
19.6.1	Lexical environment semantics of recursion . . .	339
19.7	Supplementary material	340
20	Concurrency	341
20.1	Sequential, concurrent, and parallel computation	342
20.2	Dependencies	343
20.3	Threads	344
20.4	Interthread communication	347
20.5	Futures	350
20.6	Futures are not enough	352
20.7	Locks	356
20.7.1	Abstracting lock usage	358
20.8	Deadlock	359
A	Final project: Implementing MiniML	361
A.1	Overview	361
A.1.1	Grading and collaboration	362
A.1.2	A digression: How is this project different from a problem set?	362
A.2	Implementing a substitution semantics for MiniML . . .	363
A.3	Implementing an environment semantics for MiniML .	368
A.4	Extending the language	371
A.4.1	Extension ideas	371
A.4.2	A lexically scoped environment semantics	372
A.4.3	The MiniML parser	375
A.5	Submitting the project	375
A.6	Alternative final projects	376
A	Problem sets	377
A.1	The prisoners' dilemma	377
A.2	Higher-order functional programming	378
A.3	Bignums and RSA encryption	379
A.4	Symbolic differentiation	380
A.5	Ordered collections	381
A.6	The search for intelligent solutions	382
A.6.1	Search problems	382
A.7	Refs, streams, and music	384
A.8	Force-directed graph drawing	384
A.8.1	Background	385
A.9	Simulating an infectious process	387
A.9.1	The simulation	387

B	Mathematical background and notations	389
B.1	Functions	389
B.1.1	Defining functions with equations	389
B.1.2	Notating function application	390
B.1.3	Alternative mathematical notations for functions and their application	390
B.1.4	The lambda notation for functions	393
B.2	Summation	394
B.3	Logic	395
B.4	Geometry	395
B.5	Sets	396
B.6	Equality and identity	397
C	A style guide	399
C.1	Formatting	400
C.1.1	No tab characters	400
C.1.2	80 column limit	400
C.1.3	No needless blank lines	400
C.1.4	Use parentheses sparingly	400
C.1.5	Delimiting code used for side effects	401
C.1.6	Spacing for operators and delimiters	402
C.1.7	Indentation	403
C.2	Documentation	404
C.2.1	Comments before code	404
C.2.2	Comment length should match abstraction level	405
C.2.3	Multi-line commenting	405
C.3	Naming and declarations	405
C.3.1	Naming conventions	405
C.3.2	Use meaningful names	406
C.3.3	Constants and magic numbers	407
C.3.4	Function declarations and type annotations	407
C.3.5	Avoid global mutable variables	408
C.3.6	When to rename variables	408
C.3.7	Order of declarations in a module	408
C.4	Pattern matching	409
C.4.1	No incomplete pattern matches	409
C.4.2	Pattern match in the function arguments when possible	409
C.4.3	Pattern match with as few match expressions as necessary	410
C.4.4	Misusing match expressions	410
C.4.5	Avoid using too many projection functions	411
C.5	Verbosity	411

C.5.1	Reuse code where possible	411
C.5.2	Do not abuse if expressions	412
C.5.3	Don't rewrap functions	412
C.5.4	Avoid computing values twice	413
C.6	Other common infelicities	413
D	Solutions to selected exercises	415
	Bibliography	475
	Index	479
	Image Credits	484

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.

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.

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.

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 sparseness 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 Whittington 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.