OCaml Programming: Correct + Efficient + Beautiful

Michael R. Clarkson et al.

Jun 12, 2021

CONTENTS

Fall 2021 Edition

A textbook on functional programming and data structures in OCaml, with an emphasis on semantics and software engineering.

Based on courses taught by Michael R. Clarkson, Robert L. Constable, Nate Foster, Michael D. George, Dan Grossman, Daniel P. Huttenlocher, Dexter Kozen, Greg Morrisett, Andrew C. Myers, Radu Rugina, and Ramin Zabih.

This work is based on over 20 years worth of course notes and intellectual contributions by the authors named above; teasing out who contributed what is, by now, not an easy task. The primary compiler and author of this work in its form as a unified textbook is Michael R. Clarkson.

For the most recent version of this work, see the most recent CS 3110 course website.

An experimental PDF version of this book is available. It does not contain the embedded videos that the HTML version has. It might also have typesetting errors.

CHAPTER

ONE

BETTER PROGRAMMING THROUGH OCAML

Do you already know how to program in a mainstream language like Python or Java? Good. This book is for you. It's time to learn how to program better. It's time to learn a functional language, OCaml.

Note: The HTML version of this textbook has about 200 videos embedded in it. The first one is below. The videos usually provide an introduction to material, upon which the textbook then expands.

These videos were produced during pandemic when the Cornell course that uses this textbook had to be asynchronous. The student response to them was overwhelmingly positive, so they are now being made public as part of the textbook. But just so you know, they were not produced by a professional A/V team—just a guy in his basement who was learning as he went.

The videos mostly use the versions of OCaml and its ecosystem that were current in Fall 2020. Current versions you are using are likely to look different from the videos, but don't be alarmed: the underlying ideas are the same. The most visible difference is likely to be the VS Code plugin for OCaml. In Fall 2020 the badly-aging "OCaml and Reason IDE" plugin was still being used. It has since been superceded by the "OCaml Platform" plugin.

Functional programming provides a different perspective on programming than what you have experienced so far. Adapating to that perspective requires letting go of old ideas: assignment statements, loops, classes and objects, among others. That won't be easy.

> Nan-in, a Japanese master during the Meiji era (1868-1912), received a university professor who came to inquire about Zen. Nan-in served tea. He poured his visitor's cup full, and then kept on pouring. The professor watched the overflow until he no longer could restrain himself. "It is overfull. No more will go in!" "Like this cup," Nan-in said, "you are full of your own opinions and speculations. How can I show you Zen unless you first empty your cup?"

I believe that learning OCaml will make you a better programmer. Here's why:

- You will experience the freedom of *immutability*, in which the values of so-called "variables" cannot change. Goodbye, debugging.
- You will improve at *abstraction*, which is the practice of avoiding repetition by factoring out commonality. Goodbye, bloated code.
- You will be exposed to a *type system* that you will at first hate because it rejects programs you think are correct. But you will come to love it, because you will humbly realize it was right and your programs were wrong. Goodbye, failing tests.
- You will be exposed to some of the *theory and implementation of programming languages*, helping you to understand the foundations of what you are saying to the computer when you write code. Goodbye, mysterious and magic incantations.

All of those ideas can be learned in other contexts and languages. But OCaml provides an incredible opportunity to bundle them all together. **OCaml will change the way you think about programming.**

> "A language that doesn't affect the way you think about programming is not worth knowing."
>
> ---Alan J. Perlis (1922-1990), first recipient of the Turing Award

Moreover, OCaml is beautiful. OCaml is elegant, simple, and graceful. Aesthetics do matter. Code isn't written just to be executed by machines. It's also written to communicate to humans. Elegant code is easier to read and maintain. It isn't necessarily easier to write.

The OCaml code you write can be stylish and tasteful. At first, this might not be apparent. You are learning a new language after all—you wouldn't expect to appreciate Sanskrit poetry on day 1 of Introductory Sanskrit. In fact, you'll likely feel frustrated for awhile as you struggle to express yourself in a new language. So give it some time. After you've mastered OCaml, you might be surprised at how ugly those other languages you already know end up feeling when you return to them.

1.1 The Past of OCaml

Genealogically, OCaml comes from the line of programming languages whose grandfather is Lisp and includes other modern languages such as Clojure, F#, Haskell, and Racket.

OCaml originates from work done by Robin Milner and others at the Edinburgh Laboratory for Computer Science in Scotland. They were working on theorem provers in the late 1970s and early 1980s. Traditionally, theorem provers were implemented in languages such as Lisp. Milner kept running into the problem that the theorem provers would sometimes put incorrect "proofs" (i.e., non-proofs) together and claim that they were valid. So he tried to develop a language that only allowed you to construct valid proofs. ML, which stands for "Meta Language", was the result of that work. The type system of ML was carefully constructed so that you could only construct valid proofs in the language. A theorem prover was then written as a program that constructed a proof. Eventually, this "Classic ML" evolved into a full-fledged programming language.

In the early '80s, there was a schism in the ML community with the French on one side and the British and US on another. The French went on to develop CAML and later Objective CAML (OCaml) while the Brits and Americans developed Standard ML. The two dialects are quite similar. Microsoft introduced its own variant of OCaml called F# in 2005.

Milner received the Turing Award in 1991 in large part for his work on ML. The ACM website for his award includes this praise:

> ML was way ahead of its time. It is built on clean and well-articulated mathematical ideas, teased apart so that they can be studied independently and relatively easily remixed and reused. ML has influenced many practical languages, including Java, Scala, and Microsoft's F#. Indeed, no serious language designer should ignore this example of good design.

1.2 The Present of OCaml

OCaml is a functional programming language. The key linguistic abstraction of functional languages is the mathematical function. A function maps an input to an output; for the same input, it always produces the same output. That is, mathematical functions are *stateless*: they do not maintain any extra information or *state* that persists between usages of the function. Functions are *first-class*: you can use them as input to other functions, and produce functions as output. Expressing everything in terms of functions enables a uniform and simple programming model that is easier to reason about than the procedures and methods found in other families of languages.

Imperative programming languages such as C and Java involve *mutable* state that changes throughout execution. *Commands* specify how to compute by destructively changing that state. Procedures (or methods) can have *side effects* that update state in addition to producing a return value.

The **fantasy of mutability** is that it's easy to reason about: the machine does this, then this, etc.

The **reality of mutability** is that whereas machines are good at complicated manipulation of state, humans are not good at understanding it. The essence of why that's true is that mutability breaks *referential transparency*: the ability to replace an expression with its value without affecting the result of a computation. In math, if $f(x) = y$, then you can substitute y anywhere you see $f(x)$. In imperative languages, you cannot: f might have side effects, so computing $f(x)$ at time t might result in a different value than at time t'.

It's tempting to believe that there's a single state that the machine manipulates, and that the machine does one thing at a time. Computer systems go to great lengths in attempting to provide that illusion. But it's just that: an illusion. In reality, there are many states, spread across threads, cores, processors, and networked computers. And the machine does many things concurrently. Mutability makes reasoning about distributed state and concurrent execution immensely difficult.

Immutability, however, frees the programmer from these concerns. It provides powerful ways to build correct and concurrent programs. OCaml is primarily an immutable language, like most functional languages. It does support imperative programming with mutable state, but we won't use those features until many chapters into the book—in part because we simply won't need them, and in part to get you to quit "cold turkey" from a dependence you might not have known that you had. This freedom from mutability is one of the biggest changes in perspective that OCaml can give you.

1.2.1 The Features of OCaml

OCaml is a *statically-typed* and *type-safe* programming language. A statically-typed language detects type errors at compile time, so that programs with type errors cannot be executed. A type-safe language limits which kinds of operations can be performed on which kinds of data. In practice, this prevents a lot of silly errors (e.g., treating an integer as a function) and also prevents a lot of security problems: over half of the reported break-ins at the Computer Emergency Response Team (CERT, a US government agency tasked with cybersecurity) were due to buffer overflows, something that's impossible in a type-safe language.

Some functional languages, like Python and Racket, are type-safe but *dynamically typed*. That is, type errors are caught only at run time. Other languages, like C and C++, are statically typed but not type safe. There's no guarantee that a type error won't occur an run time. And still other languages, like Java, use a combination of static and dynamic typing to achieve type safety.

OCaml supports a number of advanced features, some of which you will have encountered before, and some of which are likely to be new:

- **Algebraic datatypes:** You can build sophisticated data structures in OCaml easily, without fussing with pointers and memory management. *Pattern matching*—a feature we'll soon learn about that enables examining the shape of a data structure—makes them even more convenient.
- **Type inference:** You do not have to write type information down everywhere. The compiler automatically figures out most types. This can make the code easier to read and maintain.
- **Parametric polymorphism:** Functions and data structures can be parameterized over types. This is crucial for being able to re-use code.
- **Garbage collection:** Automatic memory management relieves you from the burden of memory allocation and deallocation, a common source of bugs in languages such as C.
- **Modules:** OCaml makes it easy to structure large systems through the use of modules. Modules are used to encapsulate implementations behind interfaces. OCaml goes well beyond the functionality of most languages with modules by providing functions (called *functors*) that manipulate modules.

1.2.2 OCaml in Industry

OCaml and other functional languages are nowhere near as popular as Python, C, or Java. OCaml's real strength lies in language manipulation (i.e., compilers, analyzers, verifiers, provers, etc.). This is not surprising, because OCaml evolved from the domain of theorem proving.

That's not to say that functional languages aren't used in industry. There are many industry projects using OCaml and Haskell, among other languages. Yaron Minsky (Cornell PhD '02) even wrote a paper about using OCaml in the financial industry. It explains how the features of OCaml make it a good choice for quickly building complex software that works.

1.3 Look to Your Future

General-purpose languages come and go. In your life you'll likely learn a handful. Today, it's Python and Java. Yesterday, it was Pascal and Cobol. Before that, it was Fortran and Lisp. Who knows what it will be tomorrow? In this fast-changing field you need to be able to rapidly adapt. A good programmer has to learn the principles behind programming that transcend the specifics of any specific language. There's no better way to get at these principles than to approach programming from a functional perspective. Learning a new language from scratch affords the opportunity to reflect along the way about the difference between *programming* and programming *in a language.*

If after OCaml you want to learn more about functional programming, you'll be well prepared. OCaml does a great job of clarifying and simplifying the essence of functional programming in a way that other languages that blend functional and imperative programming (like Scala) or take functional programming to the extreme (like Haskell) do not.

And even if you never code in OCaml again after learning it, you'll still be better prepared for the future. Advanced features of functional languages have a surprising tendency to predict new features of more mainstream languages. Java brought garbage collection into the mainstream in 1995; Lisp had it in 1958. Java didn't have generics until version 5 in 2004; the ML family had it in 1990. First-class functions and type inference have been incorporated into mainstream languages like Java, C#, and C++ over the last 10 years, long after functional languages introduced them.

News Flash!

Python just announced plans to support pattern matching in February 2021.

1.4 A Brief History of CS 3110

This book is the primary textbook for CS 3110 at Cornell University. The course has existed for over two decades and has always taught functional programming, but it has not always used OCaml.

Once upon a time, there was a course at MIT known as 6.001 *Structure and Interpretation of Computer Programs* (SICP). It had a textbook by the same name, and it used Scheme, a functional programming language. Tim Teitelbaum taught a version of the course at Cornell in Fall 1988, following the book rather closely and using Scheme.

CS 212. Dan Huttenlocher had been a TA for 6.001 at MIT; he later became faculty at Cornell. In Fall 1989, he inaugurated CS 212 Modes of Algorithm Expression. Basing the course on SICP, he infused a more rigorous approach to the material. Huttenlocher continued to develop CS 212 through the mid 1990s, using various homegrown dialects of Scheme.

Other faculty began teaching the course regularly. Ramin Zabih had taken 6.001 as a first-year student at MIT. In Spring 1994, having become faculty at Cornell, he taught CS 212. Dexter Kozen (Cornell PhD 1977) first taught the course in Spring 1996. The earliest surviving online record of the course seems to be Spring 1998, which was taught by Greg Morrisett in Dylan; the name of the course had become Structure and Interpretation of Computer Programs.

By Fall 1999, CS 212 had its own lecture notes. As CS 3110 still does, that instance of CS 212 covered functional programming, the substitution and environment models, some data structures and algorithms, and programming language implementation.

CS 312. At that time, the CS curriculum had two introductory programming courses, CS 211 Computers and Programming, and CS 212. Students took one or the other, similar to how students today take either CS 2110 or CS 2112. Then they took CS 410 Data Structures. The earliest surviving online record of CS 410 seems to be from Spring 1998. It covered many data structures and algorithms not covered by CS 212, including balanced trees and graphs, and it used Java as the programming language.

Depending on which course they took, CS 211 or 212, students were entering upper-level courses with different skill sets. After extensive discussions, the faculty chose to make CS 211 required, to rename CS 212 into CS 312 Data Structures and Functional Programming, and to make CS 211 a prerequisite for CS 312. At the same time, CS 410 was eliminated from the curriculum and its contents parceled out to CS 312 and CS 482 Introduction to Analysis of Algorithms. Dexter Kozen taught the final offering of CS 410 in Fall 1999.

Greg Morrisett inaugurated the new CS 312 in Spring 2001. He switched from Scheme to Standard ML. Kozen first taught it in Fall 2001, and Andrew Myers in Fall 2002. Myers began to incorporate material on modular programming from another MIT textbook, *Program Development in Java: Abstraction, Specification, and Object-Oriented Design* by Barbara Liskov and John Guttag. Huttenlocher first taught the course in Spring 2006.

CS 3110. In Fall 2008 two big changes came: the language switched to OCaml, and the university switched to four-digit course numbers. CS 312 became CS 3110. Myers, Huttenlocher, Kozen, and Zabih first taught the revised course in Fall 2008, Spring 2009, Fall 2009, and Fall 2010, respectively. Nate Foster first taught the course in Spring 2012; and Bob Constable and Michael George co-taught for the first time in Fall 2013.

Michael Clarkson (Cornell PhD 2010) first taught the course in Fall 2014, after having first TA'd the course as a PhD student back in Spring 2008. He began to revise the presentation of the OCaml programming material to incorporate ideas by Dan Grossman (Cornell PhD 2003) about a principled approach to learning a programming language by decomposing it into syntax, dynamic, and static semantics. Grossman uses that approach in CSE 341 Programming Languages at the University of Washington and in his popular Programming Languages MOOC.

In Fall 2018 the compilation of this textbook began. It synthesizes the work of over two decades of functional programming instruction at Cornell. In the words of the Cornell Evening Song,

> 'Tis an echo from the walls Of our own, our fair Cornell.

1.5 Summary

This book is about becoming a better programmer. Studying functional programming will help with that. The biggest obstacle in our way is the frustration of speaking a new language, particularly letting go of mutable state. But the benefits will be great: a discovery that programming transcends programming in any particular language or family of languages, an exposure to advanced language features, and an appreciation of beauty.

1.5.1 Terms and concepts

- dynamic typing
- first-class functions
- functional programming languages
- immutability
- Lisp
- ML

- OCaml
- referential transparency
- side effects
- state
- static typing
- type safety

1.5.2 Further reading

- Introduction to Objective Caml, chapters 1 and 2, a freely available textbook that is recommended for this course
- OCaml from the Very Beginning, chapter 1, a relatively inexpensive PDF textbook that is very gentle and recommended for this course
- A guided tour [of OCaml]: chapter 1 of *Real World OCaml*, a book written by some Cornellians that some students might enjoy reading
- The history of Standard ML: though it focuses on the SML variant of the ML language, it's relevant to OCaml
- The value of values: a lecture by the designer of Clojure (a modern dialect of Lisp) on how the time of imperative programming has passed
- The perils of JavaSchools: an essay by the CEO of Stack Overflow on why (my words here) CS 2110 is not enough, and why you need both CS 3110 and CS 3410.
- Teach yourself programming in 10 years: an essay by a Director of Research at Google that puts the time required to become an educated programmer into perspective

1.6 Exercises

Future chapters of this textbook contain exercises as the final section. The exercises are annotated with a difficulty rating:

- One star [★]: easy exercises that should take only a minute or two.
- Two stars [★★]: straightforward exercises that should take a fewminutes.
- Three stars [★★★]: exercises that might require anywhere from five to twenty minutes or so.
- Four [★★★★] or more stars: challenging or time-consuming exercises provided for students who want to dig deeper into the material.

Some exercises are annotated "advanced" or "optional". It's possible we've misjudged the difficulty of a problem from time to time. Let us know if you think an annotation is off.

Please do not post your solutions to the exercises anywhere, especially not in public repositories where they could be found by search engines. A repository of solutions is available to current students in the course. Instructions for how to access it will be provided elsewhere. Instructors from other universities may also request access.