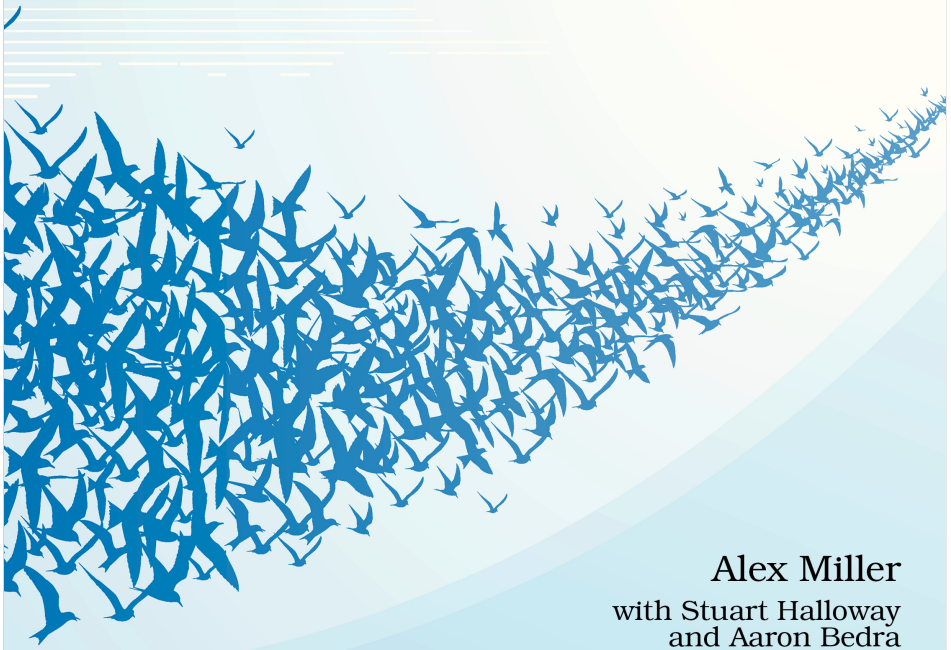
The Pragmatic Programmers

Programming Clojure

Fourth Edition



Alex Miller
with Stuart Halloway
and Aaron Bedra
edited by Jacquelyn Carter

This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit https://www.pragprog.com.

Introduction

Clojure is a dynamic programming language for the Java Virtual Machine (JVM) with a compelling combination of features:

- *Clojure is elegant.* Clojure's clean, careful design lets you write programs that get right to the essence of a problem, without a lot of clutter and ceremony.
- *Clojure is interactive.* Clojure builds on the tradition of Lisp so that you are always working with running code and immediate feedback.
- *Clojure is a functional language.* Data structures are immutable, and most functions are free from side effects. This makes it easier to write correct programs and to compose large programs from smaller ones.
- *Clojure simplifies concurrent programming.* Many languages build a concurrency model around locking, which is difficult to use correctly. Clojure provides several alternatives to locking: software transactional memory, agents, atoms, and dynamic variables.
- *Clojure embraces Java.* Calling from Clojure to Java is direct and fast, with no translation layer.
- *Unlike many popular dynamic languages, Clojure is fast.* Clojure is written to take advantage of the optimizations possible on modern JVMs.

Many other languages cover *some* of the features described in the previous list. Of all these languages, Clojure stands out. These features are not only interesting in themselves, they also reinforce each other. For example, Clojure's use of immutable data and functions without side effects are essential for safe concurrency. We will cover all these features and more in Chapter 1, Getting Started, on page ?.

Who This Book Is For

Clojure is a powerful, general-purpose programming language. As such, this book is for programmers with experience in a programming language like Java, JavaScript, C#, Python, or Ruby, but new to Clojure and looking for a powerful, elegant language.

Clojure is built on top of the Java Virtual Machine, and it is *fast*. This book will be of particular interest to Java programmers who want the expressiveness of a dynamic language without compromising on performance.

Clojure is helping to redefine what features belong in a general-purpose language. Clojure combines ideas from Lisp, functional programming, and concurrent programming and makes them more approachable to programmers seeing these ideas for the first time. Since Clojure was created, many languages like Java, JavaScript, Elixir, and Swift have adopted features inspired by Clojure.

Clojure is part of a larger phenomenon. Languages such as Erlang, F#, Haskell, and Scala have garnered attention recently for their support of functional programming or their concurrency model. Enthusiasts of these languages will find much common ground with Clojure.

What's in This Book

Chapter 1, Getting Started, on page ? demonstrates Clojure's elegance as a general-purpose language, plus the functional style and concurrency model that make Clojure unique. It also walks you through installing Clojure and developing code interactively at the REPL (Read Eval Print Loop).

Chapter 2, Exploring Clojure, on page ? is a breadth-first overview of all of Clojure's core constructs. After this chapter, you'll be able to read most day-to-day Clojure code.

Chapter 3, Developing Interactively, on page ? explores a crucial part of the Clojure experience—interactive development at the REPL, constantly loading and evaluating code as you write it.

The next two chapters cover functional programming. Chapter 4, Unifying Data with Sequences, on page ? shows how all data can be unified under the powerful *sequence* metaphor.

Chapter 5, Functional Programming, on page ? shows you how to write functional code in the same style used by the sequence functions and introduces the concepts of transducers.

Chapter 6, Describing Your Data with Specs, on page ? demonstrates how to write specifications for your data structures and functions and use them to aid in development and testing.

Chapter 7, State and Concurrency, on page ? delves into Clojure's concurrency model. Clojure provides four powerful constructs for dealing with concurrency, plus all of the tools in Java's concurrency libraries.

Chapter 8, Protocols and Datatypes, on page ? walks through records, types, and protocols in Clojure. Records and types allow you to create your own custom data types, and protocols provide functions that dispatch to implementations based on types.

the (as yet) unwritten Chapter 9, Multimethods, covers one of Clojure's answers to polymorphism. Polymorphism usually means "take the *class* of the *first* argument and dispatch a method based on that." Clojure's multimethods let you choose *any function* of *all* the arguments and dispatch based on that.

the (as yet) unwritten Chapter 10, Java Interop, shows you how to call Java from Clojure and call Clojure from Java. You'll see how to take Clojure straight to the metal and get Java-level performance.

the (as yet) unwritten Chapter 11, Macros, shows off Lisp's signature feature. Macros take advantage of the fact that Clojure code is data to provide metaprogramming abilities that are difficult or impossible in anything but a Lisp.

Chapter 12, Project Tooling, on page ? covers the tools available to manage dependencies, run tests, and debug your code.

Finally, *the (as yet) unwritten Chapter 13, Building an Application,* provides a view into a complete Clojure workflow. You will build an application from scratch, working through solving the various parts to a problem and thinking about simplicity and quality.

How to Read This Book

All readers should begin by reading the first three chapters in order. Pay particular attention to Simplicity and Power in Action, on page ?, which provides an overview of Clojure's advantages.

Experiment continuously. Clojure provides an interactive environment where you can get immediate feedback; see Using the REPL, on page ? for more information.

After you read the first three chapters, skip around as you like. But read Chapter 4, Unifying Data with Sequences, on page ? before you read Chapter 7, State and Concurrency, on page ?. These chapters lead you from Clojure's immutable data structures to a powerful model for writing correct concurrency programs.

For Functional Programmers

- Clojure's approach to FP strikes a balance between academic purity and the realities of execution on the current generation of JVMs. Read Chapter 5, Functional Programming, on page ? carefully to understand how Clojure idioms differ from languages such as Haskell.
- The concurrency model of Clojure (Chapter 7, State and Concurrency, on page ?) provides several explicit ways to deal with side effects and state and will make FP appealing to a broader audience.

For Java/C# Programmers

- Read Chapter 2, Exploring Clojure, on page ? carefully. Clojure has very little syntax (compared to Java or C#), and we cover the ground rules fairly quickly.
- If you wish to make use of Java libraries you already have, read *the (as yet) unwritten Chapter 10, Java Interop,* to learn how to call or extend Java from Clojure, and pass compatible data between them.
- Pay close attention to macros in *the (as yet) unwritten Chapter 11, Macros,* . These are the most alien part of Clojure when viewed from a Java or C# perspective.

For Lisp Programmers

- Some of Chapter 2, Exploring Clojure, on page ? will be review, but read it anyway. Clojure preserves the key features of Lisp, but it breaks with Lisp tradition in several places, and they are covered here.
- Pay close attention to the lazy sequences in Chapter 5, Functional Programming, on page ?.
- If you like Emacs, it has several modes that provide extensive support for Clojure, including REPL support, paredit, static analysis, debugging, and much more.

For Perl/Python/Ruby Programmers

- Read Chapter 7, State and Concurrency, on page ? carefully. Intraprocess concurrency is very important in Clojure.
- Embrace macros (*the (as yet) unwritten Chapter 11, Macros,*). But do not expect to easily translate metaprogramming idioms from your language into macros. Remember always that macros execute at compile time, not runtime.

Notation Conventions

The following notation conventions are used throughout the book.

Literal code examples use the following font:

```
(+ 2 2)
```

The result of executing a code example is preceded by ->.

```
(+ 2 2)
-> 4
```

Where console output cannot easily be distinguished from code and results, it's preceded by a pipe character (|).

```
(println "hello")
| hello
-> nil
```

When introducing a Clojure function for the first time, we'll show the grammar for the function like this:

```
(example-fn required-arg)
(example-fn optional-arg?)
(example-fn zero-or-more-arg*)
(example-fn one-or-more-arg+)
(example-fn & collection-of-variable-args)
```

The grammar is informal, using ? for optional, * for 0 or more, + for 1 or more, and & as a marker before a collection of variable arguments.

Clojure code is organized into namespaces. Where examples in the book depend on a namespace that's not part of the Clojure core, we document that dependency with a require form that loads the namespace:

```
(require '[clojure.java.io :as io])
(io/file "hello.txt")
-> #object[java.io.File 0x11eadcba "hello.txt"]
```

Here, the clojure.java.io namespace is required and aliased as io. Clojure returns nil from a successful call to require—for brevity, this is omitted from the example listings.

While reading the book, you'll enter code in an interactive environment called the REPL. The REPL prompt looks like this:

```
user=>
```

The user in the prompt indicates the namespace you're currently working in. For most of the examples, the current namespace is irrelevant. Where the namespace is irrelevant, we use the following syntax for interaction with the REPL:

```
(+ 2 2)         ; input line without namespace prompt
-> 4            ; return value
```

In those instances where the current namespace is important, we use this:

```
user=> (+ 2 2)  ; input line with namespace prompt
-> 4            ; return value
```

Online Resources

Programming Clojure's official home on the web is the *Programming Clojure* home page[1] at the Pragmatic Bookshelf website. From there, you can order electronic copies of the book or offer feedback by submitting errata entries or posting in the forums.[2]

The sample code for the book is also available from this page and is updated to match each release of the book. Individual examples are in the examples directory, unless otherwise noted.

Throughout the book, listings begin with their filename and a link if reading the book in PDF form. For example, the following listing comes from src/examples/preface.clj:

src/examples/preface.clj
```
(println "hello")
```

With the sample code in hand, you are ready to get started. We'll begin by meeting the combination of features that make Clojure unique.

1. https://www.pragprog.com/titles/shcloj4/
2. https://devtalk.com/books/programming-clojure-fourth-edition/