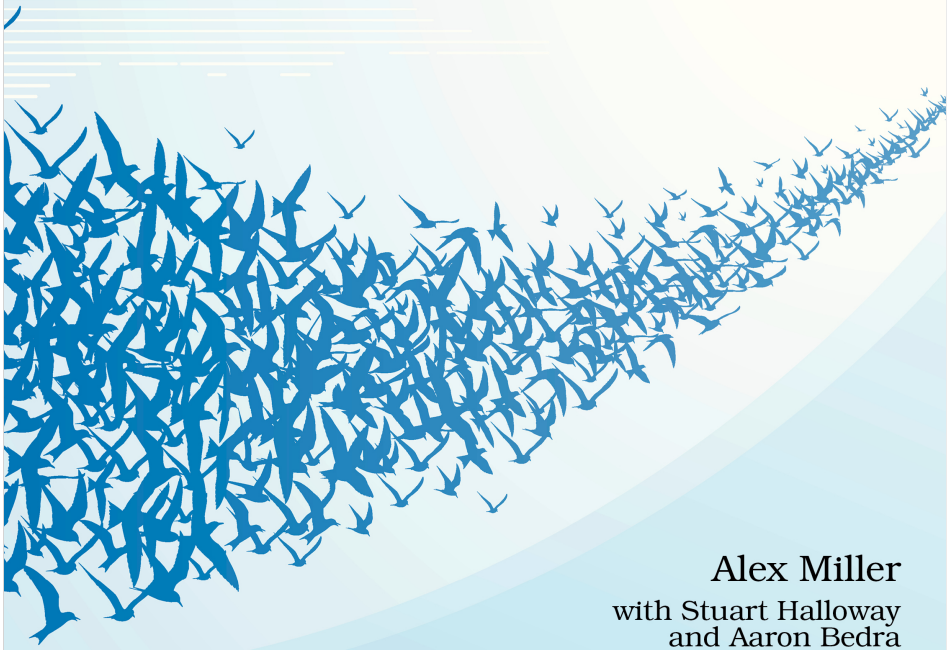
The Pragmatic Programmers

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.

Simplicity and Power in Action

All of the distinctive features in Clojure are there to provide simplicity, power, or both. Some of these features include interactive programming at the REPL, concise and expressive programs, the expressivity of macros, an immutable-first approach to state and concurrency, and an embrace of the JVM host and its ecosystem. Let's look at a few examples that demonstrate these high-level features.

Clojure Is Elegant

Clojure is high signal, low noise. As a result, Clojure programs are short programs. Short programs are cheaper to build, cheaper to deploy, and cheaper to maintain. This is particularly true when the programs are concise rather than merely terse. As an example, consider writing a function to check whether a string is blank:

src/examples/introduction.clj

```
(defn blank? [str]
  (every? Character/isWhitespace str))
```

This function is short, but more importantly, it's *simple*: it has no variables, no mutable state, and no branches. This is possible thanks to *higher-order functions*. A higher-order function is a function that takes functions as arguments and/or returns functions as results. The every? function takes a function and a collection as its arguments and returns true if that function returns true for every item in the collection.

Note also that this definition also works correctly for special cases like nil and the empty string without requiring explicit checks. Both nil and the empty string case are simply treated as sequences of 0 characters and every? is defined to return true for empty sequences.

Because the Clojure version has no branches, it's easy to read and test. These benefits are magnified in larger programs. Also, while the code is concise, it's still readable. In fact, the code reads like a *definition* of blank: a string is blank if every character in it is whitespace.

Clojure has a lot of elegance baked in, but if you find something missing, you can add it yourself, thanks to the power of Lisp.

Clojure Is Lisp Reloaded

Clojure is a Lisp. Lisps have a tiny language core, almost no syntax, and a powerful macro facility. With these features, you can bend Lisp to meet your

design, instead of the other way around. Clojure takes a new approach to Lisp by keeping the essential ideas while embracing a set of syntax enhancements that make Clojure friendlier to non-Lisp programmers.

Consider the following snippet of Java code:

```
public class Person {
  private String firstName;
  public String getFirstName() {
  // continues
```

In this code, getFirstName() is a method. Methods are polymorphic and give the programmer control over meaning, but the interpretation of *every other word* in the example is *fixed by the language*. Sometimes you really need to change what these words mean. So, for example, you might do the following:

- Redefine private to mean "private for production code but public for serialization and unit tests."
- Redefine class to automatically generate getters and setters for private fields, unless otherwise directed.
- Create a subclass of class that provides callback hooks for life-cycle events. For example, a life-cycle–aware class could fire an event whenever an instance of the class is created.

These kinds of needs are commonplace. In most languages, you would have to petition the language implementer to add the kinds of features mentioned here. In Clojure, you can add your own language features with *macros* (Chapter 11, Macros, on page ?). Clojure itself is built out of macros such as defrecord:

```
(defrecord name [field1 field2 field3])
```

If you need different semantics, write your own macro. If you want a variant of records with strong typing and configurable nil-checking for all fields, you can create your own defrecord macro, to be used like this:

```
(defrecord name [Type :field1 Type :field2 Type :field3]
  :allow-nils false)
```

This ability to reprogram the language from within the language is the unique advantage of Lisp. You will see facets of this idea described in various ways:

- Lisp is homoiconic.[2] That is, Lisp code is also Lisp data. This makes it easy for programs to write and transform other programs.
- The whole language is there, all the time. Paul Graham's essay "Revenge of the Nerds"[3] explains why this is so powerful.

Clojure offers a combination of features that make Lisp more approachable:

- Clojure generalizes Lisp's physical list into an abstraction called a *sequence*. This preserves the power of lists, while extending that power to a variety of other data structures, including ones you make yourself.
- Clojure's hosted approach includes access to the broad and portable Java standard library, and a deployment platform with great reach.
- Clojure's symbol resolution and syntax quoting makes it easier to write many common macros.

Many Clojure programmers will be new to Lisp, and they've probably heard bad things about all those parentheses. Clojure keeps the parentheses (and the power of Lisp!) but improves on traditional Lisp syntax in several ways:

- Clojure provides a convenient literal syntax for a variety of data structures besides just lists: regular expressions, maps, sets, vectors, and metadata. These features make Clojure code less *list-y* than other Lisps. For example, function parameters are specified in a vector [] instead of a list ().

  src/examples/introduction.clj

  ```
  (defn hello-world [username]
    (println (format "Hello, %s" username)))
  ```

  The vector makes the argument list jump out visually and makes Clojure function definitions easy to read.

- In Clojure, unlike most Lisps, commas to separate elements are optional —this provides concise literal collections. In fact, Clojure literally treats commas as whitespace and ignores them.

  ```
  ; make vectors look like arrays in other languages
  [1, 2, 3, 4]
  -> [1 2 3 4]
  ```

- Idiomatic Clojure primarily uses parentheses to signal *function invocation*, not grouping, whereas other Lisps use them for both. Consider the `cond` macro, present in both Common Lisp and Clojure. `cond` evaluates a set of

2. https://en.wikipedia.org/wiki/Homoiconicity
3. https://www.paulgraham.com/icad.html

test/result pairs, returning the first result for which a test form yields true. Each test/result pair is grouped with parentheses:

```
; Common Lisp cond
(cond ((= x 10) "equal")
      ((> x 10) "more"))
```

Common Lisp uses parentheses here sometime to mean invocation, and sometimes to group the parts of a case. Clojure uses parentheses in its `cond` only for invocation, and uses position to imply grouping instead:

```
; Clojure cond
(cond (= x 10) "equal"
      (> x 10) "more")
```

This is an aesthetic decision, and both approaches have their supporters. The important thing is that Clojure takes the opportunity to improve on Lisp traditions when it can do so without compromising Lisp's power.

Clojure is an excellent Lisp, for both Lisp experts and Lisp beginners.

Clojure Is a Functional Language

Clojure is a functional language but not a pure functional language like Haskell. Functional languages have the following properties:

- Functions are *first-class objects*. That is, functions can be created at runtime, passed around, returned, and in general, used like any other datatype.
- Data is immutable.
- Functions are *pure*; that is, they have no side effects.

For many tasks, functional programs are easier to understand, less error prone, and *much* easier to reuse. For example, the following short program searches a database of compositions for every composer who has written a composition named "Requiem":

```
(for [c compositions :when (= (:name c) "Requiem")] (:composer c))
-> ("W. A. Mozart" "Giuseppe Verdi")
```

The name `for` does not introduce a loop but a *list comprehension*. Read the code as, "For each `c` in `compositions`, where the name of `c` is `"Requiem"`, yield the composer of `c`." List comprehension is covered more fully in Transforming Sequences, on page ?.

This example has four desirable properties:

- It is *simple*; it has no loops, variables, or mutable state.

- It is *thread safe*; no locking is needed.
- It is *parallelizable*; you could farm out individual steps to multiple threads without changing the code for each step.
- It is *generic*; `compositions` could be a plain set, XML data, or a database result set.

Contrast functional programs with *imperative* programs, where explicit statements alter program state. Most object-oriented programs are written in an imperative style and have *none* of the advantages listed here; they are unnecessarily complex, not thread safe, not parallelizable, and difficult to generalize. (For a head-to-head comparison of functional and imperative styles, skip forward to Where's My for Loop?, on page ?.)

Clojure's approach to changing state enables concurrency without explicit locking and complements Clojure's functional core.

Clojure Simplifies Concurrent Programming

Clojure's support for functional programming makes it easy to write thread-safe code. Since immutable data structures cannot *ever* change, there's no danger of data corruption based on another thread's activity.

However, Clojure's support for concurrency goes beyond just functional programming. When you need references to mutable data, Clojure provides both atoms and software transactional memory (STM). Atoms provide safe updates to a single value in memory via an update function. STM is a higher-level approach to thread safety than the locking mechanisms that Java provides. Coordinated updates are described in transactions, similar to database transactions.

For example, the following code creates a working, thread-safe, in-memory database of accounts:

```
(def accounts (atom #{}))
```

The `atom` function creates a protected reference to the set of accounts. Account updates are described as functions that update the state. For example, adding a new account is just using `conj` to update the set:

```
(swap! accounts conj {:id "CLJ" :balance 1000.0})
```

The `swap!` ensures that the set is safely updated even if multiple threads add an account simultaneously. The primary limitation of `swap!` is that only one stateful value can be updated at a time. Clojurists have been surprised to discover that relying on immutable data and atomic values is sufficient to

satisfy most applications. When you really need it, refs and the STM are available to coordinate updates across multiple stateful values.

Although the example here is trivial, the technique is general, and it works on real-world problems. See Chapter 7, State and Concurrency, on page ? for more on concurrency and state in Clojure.

Clojure Embraces the Java Virtual Machine

Clojure gives you clean, simple, direct access to Java. You can call any Java API directly:

```
(System/getProperties)
-> {"java.specification.version" "21",
   ... many more ...
```

Clojure adds a lot of syntactic sugar for calling Java. We won't get into the details here (see Calling Java, on page ?), but notice that in the following code, the Clojure version has *fewer parentheses* than the Java version:

```
// Java
"hello".getClass().getName()

; Clojure
(.. "hello" getClass getName)
```

Clojure provides simple functions for implementing Java interfaces and subclassing Java classes. Also, all Clojure functions implement Callable and Runnable. This makes it trivial to pass the following anonymous function to the constructor for a Java Thread.

```
(.start (Thread. (fn [] (println "Hello" (Thread/currentThread)))))
-> Hello #object[java.lang.Thread 0x2057ff1f Thread[Thread-0,5,main]]
```

The funny output here is Clojure's way of printing a Java instance. java.lang.Thread is the class name of the instance, 0x2057ff1f is the hash code of the instance, and Thread[Thread-0,5,main] is the instance's toString representation.

(Note that in the preceding example, the new thread will run to completion, but its output may interleave in some strange way with the REPL prompt. This is not a problem with Clojure but simply the result of having more than one thread writing to an output stream.)

Because the Java invocation syntax in Clojure is clean and simple, it's idiomatic to use Java directly, rather than to hide Java behind Lispy wrappers.

Now that you've seen a few of the reasons to use Clojure, it's time to start writing some code.