Extracted from:

Programming Clojure

Second Edition

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

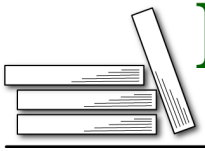
Programming Clojure

Second Edition

Stuart Halloway
Aaron Bedra

Foreword by Rich Hickey, creator of Clojure

Edited by Michael Swaine



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

The team that produced this book includes:

Michael Swaine (editor)
Potomac Indexing, LLC (indexer)
Kim Wimpsett (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Printed in the United States of America.
ISBN-13: 978-1-934356-86-9
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—April 2012

All of the distinctive features in Clojure are there to provide simplicity, power, or both. Here are a few examples:

- Functional programming is simple, in that it isolates calculation from state and identity. Benefits: functional programs are easier to understand, write, test, optimize, and parallelize.
- Clojure's Java interop forms are powerful, giving you direct access to the semantics of the Java platform. Benefits: you can have performance and semantic equivalence to Java. Most importantly, you will never need to "drop down" to a lower-level language for a little extra power.
- Lisp is simple in two critical ways: it separates reading from evaluation, and the language syntax is made from a tiny number of orthogonal parts. Benefits: syntactic abstraction captures design patterns, and S-expressions are XML, JSON, and SQL as they should have been.
- Lisp is also powerful, providing a compiler and macro system at runtime. Benefits: Lisp has late-bound decision making and easy DSLs.
- Clojure's time model is simple, separating values, identities, state, and time. Benefits: programs can perceive and remember information, without fear that somebody is about to scribble over the past.
- Protocols are simple, separating polymorphism from derivation. Benefits: you get safe, ad hoc extensibility of type and abstractions, without a tangle of design patterns or fragile monkey patching.

This list of features acts as a road map for the rest of the book, so don't worry if you don't follow every little detail here. Each feature gets an entire chapter later.

Let's see some of these features in action by building a small application. Along the way, you will learn how to load and execute the larger examples we will use later in the book.

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.[1] This is particularly true when the programs are concise rather than merely terse. As an example, consider the following Java code, from Apache Commons:

1. *Software Estimation: Demystifying the Black Art* [McC06] is a great read and makes the case that smaller is cheaper.

data/snippets/isBlank.java
```
public class StringUtils {
    public static boolean isBlank(String str) {
        int strLen;
        if (str == null || (strLen = str.length()) == 0) {
            return true;
        }
        for (int i = 0; i < strLen; i++) {
            if ((Character.isWhitespace(str.charAt(i)) == false)) {
                return false;
            }
        }
        return true;
    }
}
```

The isBlank() method checks to see whether a string is *blank*: either empty or consisting of only whitespace. Here is a similar implementation in Clojure:

src/examples/introduction.clj
```
(defn blank? [str]
  (every? #(Character/isWhitespace %) str))
```

The Clojure version is shorter. But even more important, it is *simpler*: 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.

Because the Clojure version has no branches, it is easier to read and test. These benefits are magnified in larger programs. Also, while the code is concise, it is still readable. In fact, the Clojure program reads like a *definition* of blank: a string is blank if every character in it is whitespace. This is much better than the Commons method, which hides the definition of blank behind the implementation detail of loops and if statements.

As another example, consider defining a trivial Person class in Java:

data/snippets/Person.java
```
public class Person {
    private String firstName;
    private String lastName;

    public Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
```

```
    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
}
```

In Clojure, you would define Person with a single line:

```
(defrecord Person [first-name last-name])
```

and work with the record like so:

```
(def foo (->Person "Aaron" "Bedra"))
-> #'user/foo
foo
-> #:user.Person{:first-name "Aaron", :last-name "Bedra"}
```

defrecord and related functions are covered in Section 6.3, *Protocols*, on page ?.

Other than being an order of magnitude shorter, the Clojure approach differs in that a Clojure Person is *immutable*. Immutable data structures are naturally thread safe, and update capabilities can be layered when using Clojure's references, agents, and atoms, which are covered in Chapter 5, *State*, on page ?. Because records are immutable, Clojure also provides correct implementations of hashCode() and equals() automatically.

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. For decades, Lisp advocates have pointed out the advantages that Lisp has over, well, everything else. At the same time, Lisp's world domination plan seems to be proceeding slowly.

Like any other Lisp, Clojure faces two challenges:

- Clojure must succeed as a Lisp by persuading Lisp programmers that Clojure embraces the critical parts of Lisp.
- At the same time, Clojure needs to succeed *where past Lisps have failed* by winning support from the broader community of programmers.

Clojure meets these challenges by providing the metaprogramming capabilities of Lisp and at the same time embracing a set of syntax enhancements that make Clojure friendlier to non-Lisp programmers.

Why 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. By contrast, 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 can bend to meet your needs. 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.

We have seen programs that needed all these features. Without them, programmers resort to repetitive, error-prone workarounds. Literally *millions* of lines of code have been written to work around missing features in programming languages.

In most languages, you would have to petition the language implementer to add the kinds of features mentioned earlier. In Clojure, you can add your own language features with *macros* (Chapter 7, *Macros*, on page ?). Clojure itself is built out of macros such as defrecord:

```
(defrecord name [arg1 arg2 arg3])
```

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

```
(defrecord name [Type :arg1 Type :arg2 Type :arg3]
  :allow-nulls 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 just Lisp data. This makes it easy for programs to write 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.

Lisp syntax also eliminates rules for operator precedence and associativity. You will not find a table documenting operator precedence or associativity anywhere in this book. With fully parenthesized expressions, there is no possible ambiguity.

The downside of Lisp's simple, regular syntax, at least for beginners, is Lisp's fixation on parentheses and on lists as the core datatype. Clojure offers an interesting combination of features that makes Lisp more approachable for non-Lispers.

Lisp, with Fewer Parentheses

Clojure offers significant advantages for programmers coming to it from other Lisps:

- 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.
- Clojure's reliance on the JVM provides a standard library and a deployment platform with great reach.
- Clojure's approach to symbol resolution and syntax quoting makes it easier to write many common macros.

Many Clojure programmers will be new to Lisp, and they have 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:

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

- Clojure provides a convenient literal syntax for a wide variety of data structures besides just lists: regular expressions, maps, sets, vectors, and metadata. These features make Clojure code less "listy" than most 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 are whitespace.

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

- Idiomatic Clojure does not nest parentheses more than necessary. Consider the `cond` macro, present in both Common Lisp and Clojure. `cond` evaluates a set of test/result pairs, returning the first result for which a test form yields true. Each test/result pair is grouped with parentheses, like so:

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

  Clojure avoids the extra parentheses:

  ```
  ; 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 be less Lispy when it can do so without compromising Lisp's power.

Clojure is an excellent Lisp, both for Lisp experts and for 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 (= "Requiem" (:name c))] (:composer c))
-> ("W. A. Mozart" "Giuseppe Verdi")
```

The name `for` does not introduce a loop but a *list comprehension*. Read the earlier 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 or XML 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 earlier; 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 Section 2.7, *Where's My for Loop?*, on page ?.)

People have known about the advantages of functional languages for a while now. And yet, pure functional languages like Haskell have not taken over the world, because developers find that not everything fits easily into the pure functional view.

There are four reasons that Clojure can attract more interest now than functional languages have in the past:

- Functional programming is more urgent today than ever before. Massively multicore hardware is right around the corner, and functional languages provide a clear approach for taking advantage of it. Functional programming is covered in Chapter 4, *Functional Programming*, on page ?.

- Purely functional languages can make it awkward to model state that really needs to change. Clojure provides a structured mechanism for working with changeable state via software transactional memory and refs (on page ?), agents (on page ?), atoms (on page ?), and dynamic binding (on page ?).
- Many functional languages are statically typed. Clojure's dynamic typing makes it more accessible for programmers learning functional programming.
- Clojure's Java invocation approach is *not* functional. When you call Java, you enter the familiar, mutable world. This offers a comfortable haven for beginners learning functional programming and a pragmatic alternative to functional style when you need it. Java invocation is covered in Chapter 9, *Java Down and Dirty*, 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 is 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 protects them via software transactional memory (STM). STM is a higher-level approach to thread safety than the locking mechanisms that Java provides. Rather than creating fragile, error-prone locking strategies, you can protect shared state with transactions. This is much more productive, because many programmers have a good understanding of transactions based on experience with databases.

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

```
(def accounts (ref #{}))
(defrecord Account [id balance])
```

The ref function creates a transactionally protected reference to the current state of the database. Updating is trivial. The following code adds a new account to the database:

```
(dosync
  (alter accounts conj (->Account "CLJ" 1000.00)))
```

The dosync causes the update to accounts to execute inside a transaction. This guarantees thread safety, and it is easier to use than locking. With transactions, you never have to worry about which objects to lock or in what order. The transactional approach will also perform better under some common usage scenarios, because (for example) readers will never block.

Although the example here is trivial, the technique is general, and it works on real-world problems. See Chapter 5, *State*, on page ? for more on concurrency and STM 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.runtime.name=Java(TM) SE Runtime Environment
... many more ...
```

Clojure adds a lot of syntactic sugar for calling Java. We won't get into the details here (see Section 2.5, *Calling Java*, on page ?), but notice that in the following code the Clojure version has both fewer dots *and fewer parentheses* than the Java version:

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

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

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

```
(.start (new Thread (fn [] (println "Hello" (Thread/currentThread)))))
-> Hello #<Thread Thread[Thread-0,5,main]>
```

The funny output here is Clojure's way of printing a Java instance. Thread is the class name 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 is idiomatic to use Java directly, rather than to hide Java behind Lispy wrappers.

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