Extracted from:

Getting Clojure

Build Your Functional Skills One Idea at a Time

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Getting Clojure

Build Your Functional Skills One Idea at a Time

Russ Olsen

edited by Michael Swaine

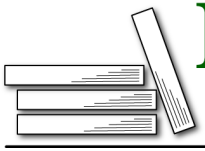
Getting Clojure

Build Your Functional Skills One Idea at a Time

Russ Olsen

The Pragmatic Bookshelf

Raleigh, North Carolina



Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Managing Editor: Brian MacDonald
Supervising Editor: Jacquelyn Carter
Development Editor: Michael Swaine
Copy Editor: Candace Cunningham
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-300-5
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—May 2018

To

Mikey

Felicia

Jackson

Charlie & Jennifer

Tim & Emily & Evan

Nicholas & Jonathan

Meg & Alex & Zachary

and

Scott Downie

The future is in your hands.

CHAPTER 19

Read and Eval

Programming-language syntax is a lot like politics: very few people do it for a living but virtually everyone has an opinion. And many have strong opinions. Certainly it's not hard to find strong opinions about Clojure's syntax. Many of us love what we see as its elegant terseness while some programmers can only see parentheses. Lots and lots of parentheses.

Clojure's somewhat odd syntax is not the shady outcome of a conspiracy of parentheses manufacturers. Nor is it a completely arbitrary esthetic choice. Clojure's syntax is an integral part of how the language works. So in this chapter we're going to look at the two critical functions at the heart of Clojure, `read` and `eval`, and at how they relate to all those parentheses. Along the way we'll take a moment to write our own version of `eval`, essentially building our very own toy Clojure implementation.

If all that sounds intimidating, take heart: like everything else in Clojure, `read` and `eval` are simple. They are also critical to getting to the next level of insight into how the language works. So let's get started.

You Got Data On My Code!

If you have read this far you probably noticed something odd about the syntax of Clojure code: it looks a lot like the syntax of Clojure data literals. If, for example, you started with this bit of nonsensical Clojure data:

read/examples.clj

```
;; Just some data: Note the quote.

'(helvetica times-roman [comic-sans]
  (futura gil-sans
    (courier "All the fonts I have loved!")))
```

you know you have a four-element list that contains a couple of symbols along with a vector and another list. But by swapping out the symbols and changing

contents of the string, you can transform that data into something convincingly codelike:

```
;; Still just data -- note the quote.

'(defn print-greeting [preferred-customer]
  (if preferred-customer
    (println "Welcome back!")))
```

This last example is still just a four-element list of data—notice the quote at the front—but it's also a dead ringer for a Clojure function definition. You can, in fact, turn your data into actual code by removing the quote:

```
;; Now this is code!

(defn print-greeting [preferred-customer]
  (if preferred-customer
    (println "Welcome back!")))
```

This little journey from data to code underlines the fundamental idea of Clojure syntax: Clojure code looks like Clojure data because in Clojure code *is* data. Clojure uses the same syntax and data structures to represent both code and data. So Clojure function calls don't just look like lists; they *are* lists. The arguments to your function definitions don't get wrapped in things that look like vectors; they *are* vectors. Languages that support this sort of *code equals data* equation are said to be *homoiconic*.

Reading and Evaluating

To make all this theory a bit more real, let's dive into the machinery that Clojure uses to read data and execute code. Actually, calling it *machinery* is overstating things since we're talking about two functions, one mundane and one wonderful. The mundane function is `read` and it does exactly what its name suggests: it reads data. Feed `read` a character-producing input stream—perhaps an open file or a network connection or your terminal—and it will read and return one Clojure value. Conveniently, if you just call `read` without any parameters, it will read from standard input. So if you call `(read)` in the REPL like this:

```
user=> (read)
```

then `read` will sit there quietly, waiting for you to type something in. So enter `55`, and `read` will return the number after 54. Alternatively, if you type in `"hello"`, `read` will return a five-character string. And if you type in the following:

```
(defn print-greeting [preferred-customer]
  (if preferred-customer (println "Welcome back!")))
```

you will get a four-element list that looks suspiciously like a Clojure function definition but is nevertheless just some data.

Along with read, Clojure also comes equipped with read-string, a function that parses the characters in a string into a Clojure value. Thus we could get the same four-element list like this:

```
;; A complicated string with some escaped quotes.

(def s
  "(defn print-greeting [preferred-customer]
     (if preferred-customer (println \"Welcome back!\")))")

;; A four-element list.

(read-string s)
```

Which brings us to the wonderful function, eval. If the read function's job is to turn characters into data structures, then it falls to eval to turn data structures into action:

```
;; A three element list.
(def a-data-structure '(+ 2 2))

;; The number 4.

(eval a-data-structure)
```

The wonderful thing about eval is that it takes the data you pass in, which should look like Clojure code, and *compiles and runs it* as Clojure code:

```
;; Bind some-data to a list

(def some-data
  '(defn print-greeting [preferred-customer]
    (if preferred-customer (println "Welcome back!"))))

;; At this point we have some-data defined,
;; but not the print-greeting function.
;; Now let's eval some-data...

(eval some-data)

;; And now print-greeting is defined!

(print-greeting true)
```

Essentially, eval attempts to evaluate whatever data you pass in *as Clojure code*. Sometimes that evaluation is trivial. Numbers, strings, and keywords just evaluate to themselves:

```
(eval 55)      ; Returns the number after 54.
(eval :hello)  ; Returns the keyword :hello
(eval "hello") ; And a string.
```

But you can also have eval evaluate symbols or call functions:

```
(def title "For Whom the Bell Tolls")

;; Get hold of the unevaluated symbol 'title...

(def the-symbol 'title)

;; ...and evaluate it.

(eval the-symbol)

;; While a list gets evaluated as a function call.

(eval '(count title))
```

The key to understanding eval is not so much what it does—it *runs stuff as code*—as what it takes in: ordinary Clojure lists, vectors, keywords, and symbols. You can, for example, construct some code on the fly—using garden-variety Clojure functions like list and vector—and evaluate it with eval. So this is yet another way to define and then call print-greeting:

```
(def fn-name 'print-greeting)
(def args (vector 'preferred-customer))
(def the-println (list 'println "Welcome back!"))
(def body (list 'if 'preferred-customer the-println))

(eval (list 'defn fn-name args body))

(eval (list 'print-greeting true))
```

The Homoiconic Advantage

So there's our answer: Clojure's syntax is the way it is because it's amphibious, equally at home representing code and data. Having a single text format along with a single in-memory representation of both code and data is not just elegant; it also has some serious practical advantages. For example, writing Clojure code-analysis tools is very straightforward. Need to read a file full of Clojure code? No problem:

```
(ns codetool.core
  (:require [clojure.java.io :as io]))

(defn read-source [path]
  (with-open [r (java.io.PushbackReader. (io/reader path))]
    (loop [result []]
      (let [expr (read r false :eof)]
        (if (= expr :eof)
          result
          (recur (conj result expr)))))))
```

Call read-source with the path to a Clojure source file, and you will get back a sequence of all of the expressions in that file, parsed into the lists and vectors that you already know how to use.

If you look closely at read-source, you will see that at its center is a slightly more elaborate call to read: (read r false :eof). The extra arguments tell read to read from somewhere besides standard input (that's the r), and to return the keyword :eof when it hits the end of the file. But the truly remarkable thing about read-source is that most of it is devoted to the mundane tasks of opening the file and managing the results. The actual parsing of the Clojure source is all bundled up in that call to read.

Even more remarkable is that the combination of read and eval makes writing a REPL so easy that sooner or later every Clojure programmer gives it a go. Here's my shot at it:

```
(defn russ-repl []
  (loop []
    (println (eval (read)))
    (recur)))
```

Just read an expression, evaluate it, print the result, and loop. Thanks mostly to eval, the REPL is the rare acronym that is nearly a program.

REPR?

Strictly speaking the loop in russ-repl is not necessary. Remove it and the recur will recursively call the function. It's there because the *L* in REPL demands it.

An Eval of Your Own

The wonderful thing about eval is that it's simultaneously a gateway to the entire Clojure programming language and a very ordinary function. That eval is just an ordinary function raises an interesting question: can we—as an intellectual exercise—implement our own version of eval?

Remarkably, we can. We've already seen that if you hand eval a string or a keyword or a number, you get the same string, keyword, or number back, unchanged. So here's a start on our own toy eval function:

```
(defn reval [expr]
  (cond
    (string? expr) expr
    (keyword? expr) expr
    (number? expr) expr
    :else :completely-confused))
```

Note that the real eval throws an exception when you hand it something it doesn't understand, but to keep the example simple we'll return :completely confused.

We can make the confusion less likely by handling symbols and vectors and lists. These are all a bit more complex, so let's delegate them to separate functions:

```
(defn reval [expr]
  (cond
    (string? expr) expr
    (keyword? expr) expr
    (number? expr) expr
    (symbol? expr) (eval-symbol expr)
    (vector? expr) (eval-vector expr)
    (list? expr) (eval-list expr)
    :else :completely-confused))
```

Actually evaluating symbols isn't too difficult: just look them up in the current namespace:

```
(defn eval-symbol [expr]
  (.get (ns-resolve *ns* expr)))
```

Vectors are also straightforward. We just need to recursively evaluate the contents:

```
(defn eval-vector [expr]
  (vec (map reval expr)))
```

Things only get interesting when we evaluate lists. First we need to evaluate the contents of the list in exactly the same way that we did with vectors. Once we've done that we just need to call the function, which we do with apply:

```
(defn eval-list [expr]
  (let [evaled-items (map reval expr)
        f (first evaled-items)
        args (rest evaled-items)]
    (apply f args)))
```

We could go on, perhaps adding support for maps and if expressions and ns and so forth, but let's pause here and take stock.

Go!

I encourage you to go on and see how much Clojure you can implement. There is nothing like building your own to get a clearer idea of how the real thing works.

The first thing to note about our excursion into programming-language implementation is that we're cheating. We're relying on all the glories that Clojure provides to implement a slow, partial subset of Clojure. What we get out of reval is not a practical programming language but *insight*—insight into how Clojure works—all courtesy of the homoiconic power of the language.

Second, it's important to keep in mind that since Clojure—real Clojure—is a compiled language, the details of the real eval are rather more complicated. Instead of saying, *Oh, this is a list. Treat it as a function call*, the real eval says, *Oh, this is a list. Generate some code to call the function and then run that code*.

Nevertheless, building toy versions of eval is so much fun and so illuminating that it has been a cottage industry among programmers using LISP-based languages for decades. Welcome to the club.

In the Wild

If you are interested in the ins and outs of implementing Clojure, you should definitely check out the MAL project.[1] MAL, short for *Make a Lisp*, defines a simple Clojure-like language and then proceeds to implement it in (as of this writing) 68 languages, everything from Ada to Visual Basic.

You can find a great example of the power of using plain old Clojure values for both code and data in metadata. Metadata is extra data that you can hang on Clojure symbols and collections, data that in some ways enhances your value without being an official part of the value.

There are two ways you can hang some metadata on a value. You can do it either explicitly with the with-meta function:

```
(def books1 (with-meta ["Emma" "1984"] {:favorite-books true}))
```

or by using the ^:keyword syntactical sugar:

```
(def books1 ^:favorite-books ["Emma" "1984"])
```

Having applied some metadata to your value, you can get it back with the meta function:

```
;; Gives you the {:favorite-books true} map.
(meta books1)
```

1. https://github.com/kanaka/mal

The key thing about metadata is that it is *extra* data: metadata doesn't affect the actual value. That means that two otherwise equal values are still equal even if they have different metadata:

```
;; Otherwise identical vectors with different metadata...
(def books2 (with-meta ["Emma" "1984"] {:favorite-books true}))
(def books3 (with-meta ["Emma" "1984"] {:favorite-books false}))
;; Are still equal.
(= books2 books3)   ; True!
```

If the metadata syntax looks familiar, it should. We already came across metadata when we were dealing with dynamic vars:

```
(def ^:dynamic *print-length* nil)
```

There are also less obvious uses of metadata. For example, when you define a function with a docstring, Clojure stashes the docstring of the function in the metadata of the symbol.

But don't take my word for it. Define a function with a docstring:

```
(defn add2
  "Return the sum of two numbers"
  [a b]
  (+ a b))
```

and then look at the metadata on the `add2` var:

```
(meta #'add2)
```

You will see something like this:

```
{:doc "Return the sum of two numbers",
 :arglists ([a b]),
 :name add2,
 :ns #object[clojure.lang.Namespace 0xa55c011 "user"]
 :line 1
 :column 1
 ...
}
```

And there is the docstring, along with all sorts of useful information about our function, all stashed in the metadata.

Finally—and returning to the topic at hand—in exactly the same way that a function call is just a list and the parameters in a `defn` are just a vector, metadata is just a map. So if you pick up some metadata from a value:

```
(def md (meta books3))
```

you have a plain old map:

```
;; Do mapish things to the metadata

(count md) ; Returns 1
(vals md)  ; Returns (false)
```

Remember, Clojure code is just Clojure data, all the way down.

Staying Out of Trouble

Having introduced you to the eval function, let me now give you a critical bit of advice: *don't use it.* At least not for real, at least not now—not while you are just learning Clojure. The eval function is an incredibly useful tool to have lying around, but it's also something of a blunt instrument. Just how blunt? Think about the difference between writing this into your Clojure program:

```
(+ 1 1)
```

and this:

```
(eval '(+ 1 1))
```

The first, straightforward, bit of code adds two numbers together. If it's part of a Clojure application then it will get compiled once and it's the compiled version that will get fired off at runtime. The second rendition of the code will also get compiled once, but it's the call to eval, along with some data, that will get compiled. At runtime that call to eval will crank up the Clojure compiler *again*, this time to turn the list (+ 1 1) into something executable. And that will happen every time you want to add 1 and 1.

There are also some subtleties involved in using eval that will eventually rise up and bite you if you get too enthusiastic. To take just one example, since eval doesn't know about your local *generated by let* bindings, this:

```
(def x 1)

(let [x 10000000]
  (eval '(+ x 1)))
```

may not give you the result you are expecting.

In real life people tend to reserve eval for development tools like the REPL and for those rare moments when you need to programmatically generate and execute code. Reserve eval for those moments when you have no idea what runtime code you want to execute until some runtime data tells you. For the other 99.99 percent of the programming problems that you are likely to face, the ordinary tools of functional programming, together with macros—which we'll talk about in the next chapter—will suffice. And yes, we're now at the

point where we can describe the amazing bag of tricks that is functional programming as *ordinary*.

Compared to eval, the read function is both less cool and more useful on an everyday basis. The main danger lurking inside of read is that under some circumstances it will execute arbitrary code as specified by the data you're reading. For example, try evaluating this:

```
(def s "#=(println \"All your bases...\")")
(read-string s)
```

The bottom line is that you only want to use read to read data that you trust. If you want to read some Clojure-style data from an untrusted source, use the read function from the clojure.edn namespace (the official name for the Clojure-style data format is EDN):

```
(require '[clojure.edn :as edn])
;; Read some data that I don't necessarily trust.
(def untrusted (edn/read))
```

And one final word on Clojure's syntax: as we've seen in this chapter, there is a method to the madness of Clojure's syntax. Yes, putting the function name on the inside—(rest some-seq)—looks a little strange. But at the cost of adapting to a slightly odd syntax, we get the ability to read and manipulate Clojure code as data and we get the seamless integrations of read and eval. And, as we'll see in the next chapter, we get macros.

If the Clojure syntax still bugs you, keep in mind programming-language syntax is largely a matter of taste. Nobody is born understanding what this means:

```
if (x == 0) {
  System.out.println("The number is zero");
}
```

Nor this:

```
if x == 0
  puts "The number is zero"
end
```

And certainly not this:

```
<?xml version="1.0" encoding="utf-8"?>
<books>
  <book>
    <name>Pride and Prejudice</name>
    <author>Austen</author>
  </book>
```

```
  <book>
    <name>Death Comes to Pemberley</name>
    <author>James</author>
  </book>
  <book>
    <name>Pride and Prejudice and Zombies</name>
    <author>Grahame-Smith</author>
  </book>
</books>
```

And yet we adapted. If Clojure's syntax is still giving you headaches, stick with it. It is an integral part of what makes the language go.

Wrapping Up

In this chapter we explored the ideas behind Clojure's somewhat odd but *it grows on you after a while* syntax. We saw how Clojure syntax is a bit odd because it's a compromise between being a good programming-language syntax and a good syntax for data. But that compromise pays off in a big way: by using the same syntax and data structures for code and data, we can roll the entire Clojure programming language into two simple functions: `read` and `eval`.

Now that you understand the ideas behind Clojure's syntax, it's time to turn to the final topic of this book: macros.