Extracted from:

Mastering Clojure Macros

Write Cleaner, Faster, Smarter Code

The Pragmatic Bookshelf

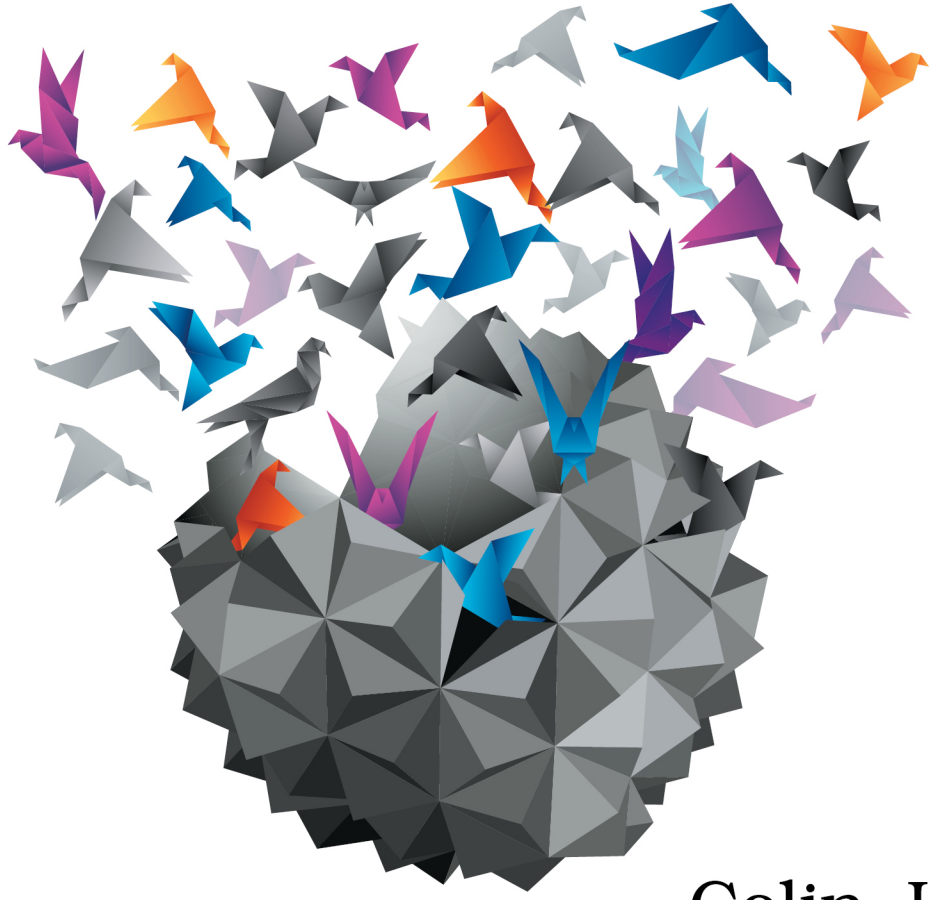
Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Pragmatic exPress

Mastering Clojure Macros

Write Cleaner, Faster, Smarter Code



Colin Jones

Edited by Fahmida Y. Rashid

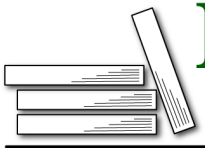
Mastering Clojure Macros

Write Cleaner, Faster, Smarter Code

Colin Jones

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



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:

Fahmida Rashid (editor)
Liz Welch (copyeditor)
Dave Thomas (typesetter)
Janet Furlow (producer)
Ellie Callahan (support)

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

Printed in the United States of America.
ISBN-13: 978-1-941222-22-5
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—August 2014

Most of the macros you've seen so far have been small and straightforward. Wouldn't it be great if they could all be like that? Unfortunately, as you do more and more with macros, the syntax you know so far can get unwieldy.

What if you had to write an assert macro like the one that comes with Clojure? Given what you know at this point, you'd need to do something like this:

advanced_mechanics/assert_no_syntax_quote.clj

```
(defmacro assert [x]
  (when *assert* ;; check the dynamic var `clojure.core/*assert*` to make sure
                 ;;   assertions are enabled
    (list 'when-not x
       (list 'throw
             (list 'new 'AssertionError
                   (list 'str "Assert failed: "
                         (list 'pr-str (list 'quote x))))))))

user=> (assert (= 1 2))
;=> AssertionError Assert failed: (= 1 2)  user/eval214 (NO_SOURCE_FILE:1)

user=> (assert (= 1 1))
;=> nil
```

And this isn't even a complete solution! We've skipped the arity[1] that takes a failure message string, to keep things from getting too ridiculous. But there's a lot to read and learn here, right?

I don't know about you, but I find it very hard to parse all those nested lists to discover what's going to come out in the macroexpansion. Luckily we know about macroexpand from *Macroexpansion*, on page ?, so it doesn't have to stay a mystery for long:

advanced_mechanics/assert_no_syntax_quote_macroexpanded.clj

```
(macroexpand '(assert (= 1 2)))
;=> (if (= 1 2)
;     nil
;     (do (throw (new AssertionError
;                     (str "Assert failed: "
;                          (pr-str (quote (= 1 2))))))))
;;; [indentation for clarity]
```

How can we do this better? Maybe you already have some ideas about how we can make this code look much more like the code it generates. To do so, we'll use *syntax-quote* for the first time.

1. http://en.wikipedia.org/wiki/Arity

Syntax-Quoting and Unquoting

The big problem with this assert implementation is that it takes a pretty big structural leap to go from the macro implementation to the result of the macroexpansion. This is no problem for the compiler, and our human brains can work it out too, but there's an easier way. The syntax-quote gives us a way to structure a macro's code to look much more like its macroexpansion.

The syntax-quote lets us create lists similar to the way we create them with a normal quote, but it has the added benefit of letting us temporarily break out of the quoted list and interpolate values with an unquote. Think of it as a template, where we can punch holes and insert values wherever we like. For instance, if we had a list where we wanted to insert a value, our normal quote wouldn't fly:

advanced_mechanics/normal_quote_is_stubborn.clj

```
user=> (def a 4)
;=> #'user/a
user=> '(1 2 3 a 5)
;=> (1 2 3 a 5)

user=> (list 1 2 3 a 5)
;=> (1 2 3 4 5)
```

The fourth element in the first list we created is just the symbol a. If we want the value of a, we have to either use the more verbose list construction, or use a syntax-quote with an unquote:

advanced_mechanics/syntax_quote_1.clj

```
user=> (def a 4)
;=> #'user/a
user=> `(1 2 3 ~a 5)
;=> (1 2 3 4 5)
```

If you don't see the difference in these two quote characters at first, look a wee bit closer. The normal quote (') looks like it's on the straight and narrow, and the syntax-quote (`) is a little cockeyed and apparently ready to party. You might also know the syntax-quote as the backtick. The *unquote* (~), well, it unquotes, letting us insert evaluations into the syntax-quoted expression.

If we take a look at how assert is actually implemented, we see that syntax-quote and unquote completely solve the verbosity problem we saw earlier:

advanced_mechanics/assert_syntax_quote.clj

```
(defmacro assert [x]
  (when *assert*
    `(when-not ~x
       (throw (new AssertionError (str "Assert failed: " (pr-str '~x)))))))
```

Wow! That's a lot less code than the nested-list version, and it looks a lot closer to the macroexpansion. As a result, it's easier to understand and maintain. There's one potentially tricky bit left, though: what does it mean when we say '~x within a syntax-quoted expression?

The REPL is a great place to experiment whenever you see something you don't understand. Why don't we give it a try?

advanced_mechanics/syntax_quote_2.clj

```
user=> (def a 4)
;=> #'user/a
user=> `(1 2 3 '~a 5)
;=> (1 2 3 (quote 4) 5)
```

Aha! So this strange '~ dance gives us a way to quote the result of evaluation and plug *that* into a slot in the syntax-quote expression.

Recall from our introduction to quoting on page ? that the normal quote is a reader macro expanding to (quote ...). Well, it turns out that the unquote ~ is another reader macro. So an expanded version of this would look like:

advanced_mechanics/syntax_quote_3.clj

```
user=> `(1 2 3 (quote (clojure.core/unquote a)) 5)
;=> (1 2 3 (quote 4) 5)
```

Internally, Clojure's reader has some special code to walk through the syntax-quote form looking for clojure.core/unquote occurrences and unquoting those things. I wouldn't try to use clojure.core/unquote outside the scope of a syntax-quote, though—it won't work unless you've written a macro that makes it work. Leiningen[2] (as of version 2.3.4) actually allows the unquote to be used in project.clj for evaluation, but it's now discouraged in favor of *read-eval*. That var (clojure.core/unquote) is unbound by default, and it's unclear whether it's strictly needed by the language, since everything using it looks for a *symbol*, not a var.

Joe asks:

What's Read-eval?

Read-eval is the name of the Clojure reader macro that allows you to evaluate code during a read. For instance, (read-string "#=(+ 1 2)") returns 3, not (+ 1 2) as you'd get without the #=. This has security implications for read and anything that depends on it, so you should always be careful not to read user input, preferring something like clojure.edn/read instead. Generally you should make it a goal to steer clear of read-eval whenever possible.

2. https://leiningen.org

There are other strange ways we can mix and match these quotes and unquotes as well, but first let's see a special kind of unquote called *unquote-splicing*. Let's say we have a list in hand of an unknown length, and we want to insert all the elements from that list into another list. Using the unquote that we already know about won't fly here, but we can use the usual concat to get the result we want. Always remember that when you hit a wall in writing macros, you can fall back on all your normal Clojure-writing experience, since when you write macros you're manipulating normal data structures like lists.

advanced_mechanics/unquote_splicing_1.clj

```
user=> (def other-numbers '(4 5 6 7 8))
;=> #'user/other-numbers
user=> `(1 2 3 ~other-numbers 9 10)
;=> (1 2 3 (4 5 6 7 8) 9 10)
user=> (concat '(1 2 3) other-numbers '(9 10))
;=> (1 2 3 4 5 6 7 8 9 10)
```

This concat version is a little unsatisfying. It works fine for this use case, but it might not be so great if we needed to inject the values into a syntax-quoted expression. Luckily, the unquote-splicing reader macro, ~@, gives us a succinct and fully syntax-quote-compatible way of doing the same thing:

advanced_mechanics/unquote_splicing_2.clj

```
user=> (def other-numbers '(4 5 6 7 8))
;=> #'user/other-numbers
user=> `(1 2 3 ~@other-numbers 9 10)
;=> (1 2 3 4 5 6 7 8 9 10)
```

And there are hooks in Clojure's syntax-quote implementation in the reader that look for clojure.core/unquote-splicing occurrences, just like with the normal unquote, to make this behavior possible.

Syntax-quote As a Macro?

Believe it or not, it's also possible for syntax-quote to be written as a macro, and at least two people have created projects doing just that:

- https://github.com/brandonbloom/backtick
- https://github.com/hiredman/syntax-quote

So one day we might very well see the syntax-quote arise from the dark depths of the Reader into the light of macro-land.

Besides the ability to interpolate values when syntax-quoting, there's an additional implication for symbols that occur within the syntax-quoted form. Take a look at the difference between this normal-quoted expression and its syntax-quoted sibling:

advanced_mechanics/syntax_quote_4.clj

```
user=> '(a b c)
;=> (a b c)
user=> `(a b c)
;=> (user/a user/b user/c)
```

With the syntax-quoted version, the symbols all include the namespaces where the syntax-quote appears! We say that these symbols are *namespace-qualified*. At first this might seem strange, but there's a good reason for it. Imagine you had this macro (which, as you'll see in Chapter 3, *Use Your Powers Wisely*, on page ?, is a bad idea to begin with, but we'll use it here for clarity):

advanced_mechanics/syntax_quote_5.clj

```
user=> (defmacro squares [xs] (list 'map '#(* % %) xs))
;=> #'user/squares
user=> (squares (range 10))
;=> (0 1 4 9 16 25 36 49 64 81)
```

Easy enough—we just map over the input collection, squaring each element.

What do you think would happen if we were to use this macro from a namespace where `map` meant something different?

advanced_mechanics/syntax_quote_6.clj

```
user=> (ns foo (:refer-clojure :exclude [map]))
;=> nil
foo=> (def map {:a 1 :b 2})
;=> #'foo/map
foo=> (user/squares (range 10))
;=> (0 1 2 3 4 5 6 7 8 9)
foo=> (user/squares :a)
;=> :a
foo=> (first (macroexpand '(user/squares (range 10))))
;=> map
foo=> ({:a 1 :b 2} :nonexistent-key :default-value)
;=> :default-value
```

So in a situation like this, since the verb `map` is an unqualified symbol, the `squares` macro call in the namespace `foo` will cause `foo/map` to be used as a function. The squaring function gets passed in *as the value to look up*, and the not-found default gets returned as the result in these cases. Well, this clearly isn't what we wanted when we wrote that macro, and that's where

syntax-quote's namespace qualification leaps to the rescue. If we instead define the squares macro using a syntax-quote (or if we're into making things cumbersome, namespace-qualifying the map symbol ourselves), we don't have this problem:

advanced_mechanics/syntax_quote_7.clj

```
user=> (defmacro squares [xs] `(map #(* % %) ~xs))
;=> #'user/squares
user=> (squares (range 10))
;=> (0 1 4 9 16 25 36 49 64 81)
user=> (ns foo (:refer-clojure :exclude [map]))
;=> nil
foo=> (def map {:a 1 :b 2})
;=> #'foo/map
foo=> (user/squares (range 10))
;=> (0 1 4 9 16 25 36 49 64 81)
foo=> (first (macroexpand '(user/squares (range 10))))
;=> clojure.core/map
```

This is a great example of how Clojure's macro system tries to help you avoid shooting yourself in the foot. Use and master the syntax-quote, and your macro-writing life will be much easier. But we do need to go just a bit further to get you all the syntax-quoting knowledge you need.

If we had chosen to write the squares macro in a slightly different way, using the fn special form instead of the shortcut syntax (#(* % %)), we would have seen an error:

advanced_mechanics/gensym_1.clj

```
user=> (defmacro squares [xs] `(map (fn [x] (* x x)) ~xs))
;=> #'user/squares
user=> (squares (range 10))
;=> CompilerException java.lang.RuntimeException:
;  Can't use qualified name as parameter: user/x, compiling: (NO_SOURCE_PATH:1:1)
```

Ack! The namespace expansion has bitten us in this case, rather than helping. We could roll back to the list approach without too much pain here, but we'd really like to take advantage of syntax-quote's benefits. Given what you know about syntax-quoting and unquoting, you can extrapolate to a solution that puts a non-namespaced symbol in the macroexpansion:

advanced_mechanics/unhygienic_1.clj

```
user=> `(* ~'x ~'x)
;=> (clojure.core/* x x)
```

This solution would work fine for our use case: we could slap ~' in front of all the xs in the definition of squares and have a working implementation in no time:

advanced_mechanics/unhygienic_2.clj
```
user=> (defmacro squares [xs] `(map (fn [~'x] (* ~'x ~'x)) ~xs))
;=> #'user/squares
user=> (squares (range 10))
;=> (0 1 4 9 16 25 36 49 64 81)
```

And because the scope of x is limited to that anonymous squaring function, we don't have the namespace-related problems we saw before. However, let's expand our worldview to include macros *other* than the now-worn-out squares:

advanced_mechanics/unhygienic_3.clj
```
user=> (defmacro make-adder [x] `(fn [~'y] (+ ~x ~'y)))
;=> #'user/make-adder
user=> (macroexpand-1 '(make-adder 10))
;=> (clojure.core/fn [y] (clojure.core/+ 10 y))
```

So make-adder expands into a function definition, and that function adds the macro's argument to the function's argument. This has a strange result when we try to use it:

advanced_mechanics/unhygienic_4.clj
```
user=> (defmacro make-adder [x] `(fn [~'y] (+ ~x ~'y)))
;=> #'user/make-adder
user=> (def y 100)
;=> #'user/y
user=> ((make-adder (+ y 3)) 5)
;=> 13
```

Despite the fact that we defined y to be 100, it looks like the value being used for y is 5! Why do you think this is? By macroexpanding the make-adder call in question, we can see that we're creating a function that takes one argument named y, which shadows the (def y 100) definition:

advanced_mechanics/unhygienic_5.clj
```
user=> (macroexpand-1 '(make-adder (+ y 3)))
;=> (clojure.core/fn [y] (clojure.core/+ (+ y 3) y))
```

If you were trying to use this version of make-adder without digging into the macro definition, you'd think this was horribly broken, wouldn't you? We've accidentally allowed what's called *symbol capture*, where the macro has internally shadowed, or captured, some symbols that users of this macro might expect to be available when their expression is evaluated. There is a solution here, though, and it's called the *gensym*.

Approaching Hygiene with the Gensym

In order to avoid symbol capture issues like the one we just saw, Clojure gives us a few tools, all related to the gensym function. gensym's job is simple: it pro-

duces a symbol with a unique name. The names will look funny because the name needs to be unique for the application, but that's OK because we never need to type them into code:

advanced_mechanics/gensym_2.clj

```
user=> (gensym)
;=> G__671
user=> (gensym)
;=> G__674
user=> (gensym "xyz")
;=> xyz677
user=> (gensym "xyz")
;=> xyz680
```

As you can see, any given invocation of gensym gives a unique value back—so if you want to refer to the same one twice, you'll need to hold onto the value with a let binding or something similar. These generated symbols (gensyms) are extremely useful for macros, but because they're normal data, you can use them anywhere you'd use a symbol. In our make-adder macro earlier, we can't have user/y as a function argument, and we just saw that we don't want plain old y as a function argument, but we *can* use a gensym as a function argument:

advanced_mechanics/gensym_3.clj

```
(defmacro make-adder [x]
  (let [y (gensym)]
    `(fn [~y] (+ ~x ~y))))

user=> y
100
user=> ((make-adder (+ y 3)) 5)
108
```

Now this version uses the value of y that we'd expect as users of this macro. Notice that the let and gensym here are *outside* of the syntax-quote. It's a bit unfortunate that this is so verbose—let's use the more concise and built-in version. We'll use a feature called the *auto-gensym*, which just looks like a normal symbol with a pound sign (#) at the end, like a reverse hashtag:

advanced_mechanics/gensym_4.clj

```
(defmacro make-adder [x]
  `(fn [y#] (+ ~x y#)))

user=> y
100
user=> ((make-adder (+ y 3)) 5)
108
```

This, and not any of the previous ways we've done it, should be the tool you reach for when you need to bind a name within a macro. There are several other very similar cases of binding symbols to values where we also need to use gensyms to construct macros safely:

advanced_mechanics/gensym_5.clj

```
(defmacro safe-math-expression? [expression]
  `(try ~expression
        true
        (catch ArithmeticException e# false)))

;; clojure.core/and
(defmacro and
  ([] true)
  ([x] x)
  ([x & next]
   `(let [and# ~x]
      (if and# (and ~@next) and#))))
```

Bindings set up by special forms like let, letfn, and try's catch clause have the same requirement as function arguments, so you should typically use the auto-gensym for these situations, too.

A lot of care has been taken in Clojure to make macro construction less error-prone. These variable-capture problems, along with the ability to get gensyms explicitly, have been around for a long time in Common Lisp, but it takes a bit of voodoo (see Doug Hoyte's *Let Over Lambda [Hoy08]*) to get something like Clojure's auto-gensym feature. It's worth noting that if you wander into the dark forests of nesting syntax-quotes, you (a) may never return, and (b) may want to take a look at unify-gensyms from Zach Tellman's Potemkin.[3]

Of course, anyone with a Scheme background is probably howling at this point because they have a *hygienic* macro system that makes unintended variable capture impossible. Allowing variable capture when we really, really want it makes Clojure's macro system technically more dangerous than hygienic systems. By getting us most of the way there, Clojure gives us more safety than Common Lisp's macro system, along with the power to do variable capture when it makes for an elegant solution.

3. https://github.com/ztellman/potemkin