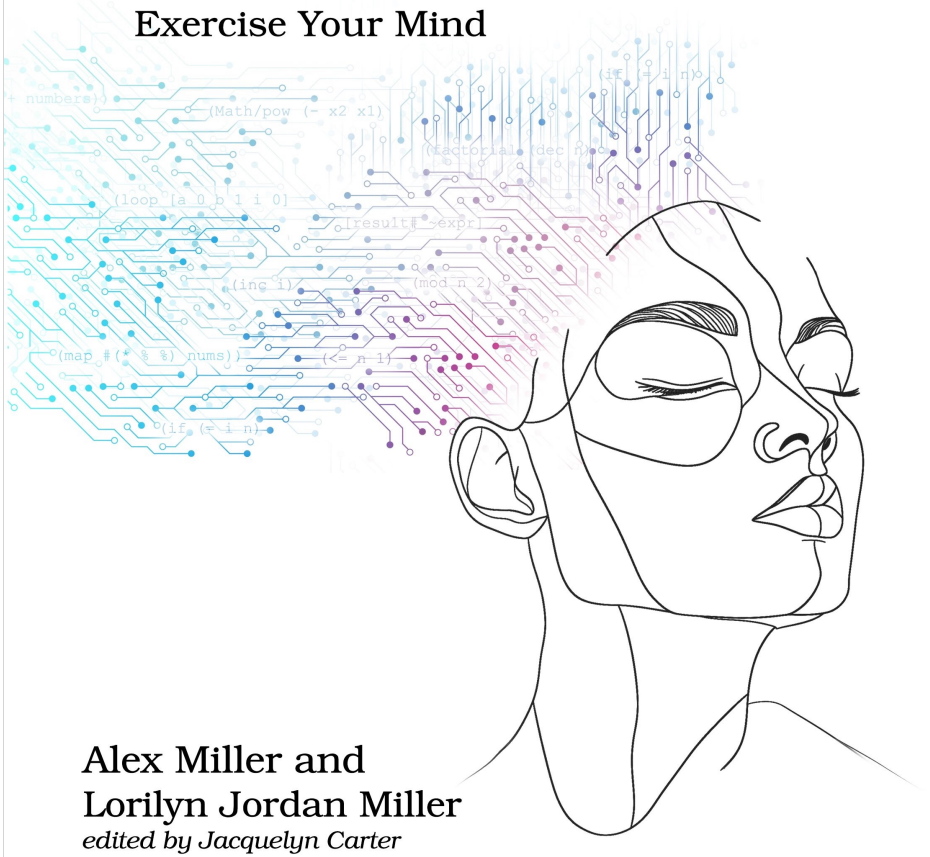
The Pragmatic Programmers

Clojure Brain Teasers

Exercise Your Mind



Alex Miller and
Lorilyn Jordan Miller
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.

Puzzle 16

Quoth the Raven

book/src/quoth.clj

```
(def raven "nevermore")

(= raven "nevermore")
(= 'raven (symbol "raven"))
(= `raven (symbol "user/raven"))
```

Guess the Output

Try to guess what the output is before moving to the next page.

The program will display the following output for all expressions:

```
(def raven "nevermore")

(= raven "nevermore")
;;=> true

(= 'raven (symbol "raven"))
;;=> true

(= `raven (symbol "user/raven"))
;;=> true
```

Discussion

All three forms return true, demonstrating several versions of symbol evaluation and quoting.

In Clojure, all code is made up of expressions, which may be arbitrarily nested. The evaluation phase processes each top-level expression to determine a result, which typically involves recursively evaluating nested expressions. However, sometimes we want to read a form but NOT evaluate it, thus leaving it as data. That brings us to *quoting*.

The first form is just the symbol raven:

```
(= raven "nevermore")
;;=> true
```

Symbols are names that resolve to something. Here, the raven symbol is a name referring to the var that was created with def in the first line. The var is resolved in the namespace and evaluated to its current value, the string "nevermore".

The second form quotes the symbol with a single quote using a single quote character:

```
(= 'raven (symbol "raven"))
;;=> true
```

The ' is expanded in the reader to (quote raven). quote is a special form implemented by the compiler—it will cause its value to be read as data by the reader but NOT evaluated.

For example, this code creates a literal list with symbols in it, without looking up the symbols or invoking the first element of the list as an operation.

```
(a b c)    ;; tries to evaluate by invoking a
;;=> Syntax error compiling at (REPL:1:1).
;;=> Unable to resolve symbol: a in this context

'(a b c)   ;; suppress evaluation
;;=> (a b c)
```

The third form uses a backtick character (called *syntax quote* in Clojure):

```
(= `raven (symbol "user/raven"))
;;=> true
```

Like a quote, a syntax quote avoids evaluation, but it has some additional features that make it ideal for use in macros. Macros are a special kind of function—they take code as input and return replacement code as output. In a sense, macros let users provide code that runs at compilation time (perhaps Poe would call it a "Dream Within a Dream").

The teaser was run in the `user` namespace. Because the `` `raven `` symbol is unqualified and does not refer to a local binding, it is resolved here in the "current" namespace. The syntax quoted symbol is thus resolved to `user/raven`.

Because macros run in the context where they are invoked, they need to return code with names that have meaning in that context, not within the namespace of the macro itself. Thus, macro writers can use a syntax quote with unqualified symbols to resolve to qualified symbols in the context of the expanded code.

In this teaser we've seen how symbols, quoted symbols, and syntax quoted symbols vary in their evaluation. While you won't need to quote very often in your code, it is an essential tool when writing macros and some kinds of metaprogramming.

Further Reading

Clojure Reference - Evaluation
https://clojure.org/reference/evaluation

Clojure Reference - Special Form quote
https://clojure.org/reference/special_forms#quote