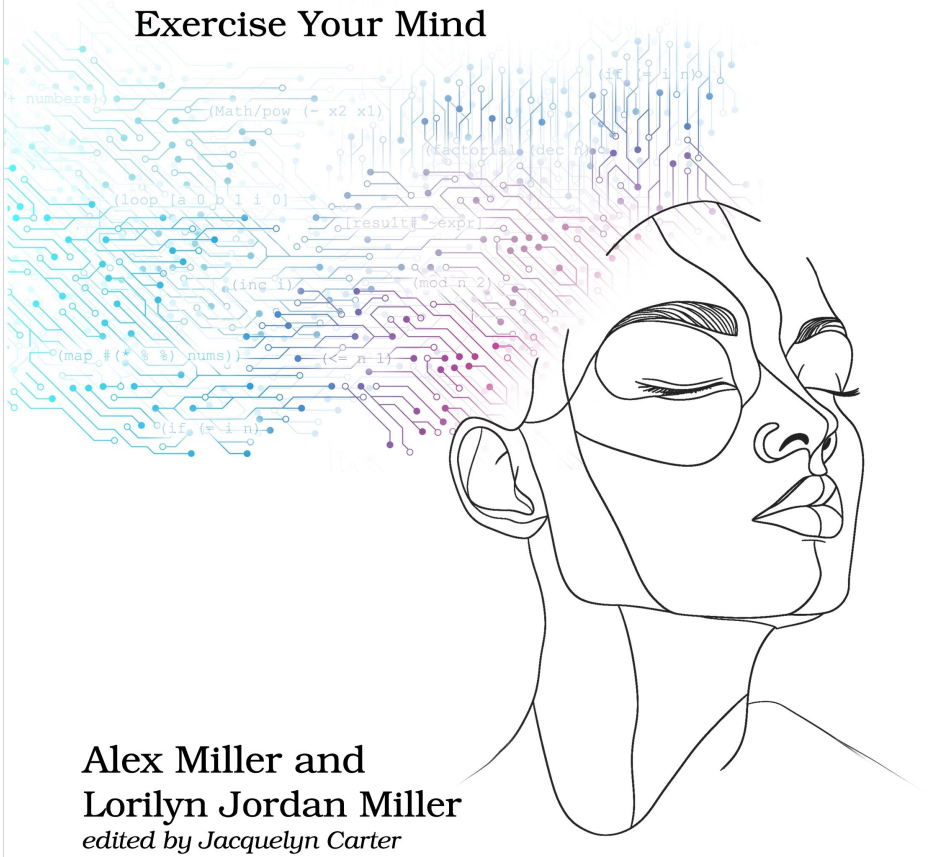
The Pragmatic Programmers

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 7

When in Rome

book/src/when_in_rome.clj

```
(= (conj '(:colosseum :vatican) :pantheon :trevi-fountain)
   (conj  [:colosseum :vatican] :pantheon :trevi-fountain))
```

Guess the Output

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

The program will display the following output:

```
(= (conj '(:colosseum :vatican) :pantheon :trevi-fountain)
   (conj  [:colosseum :vatican] :pantheon :trevi-fountain))
;;=> false
```

Discussion

As we plan our itinerary of sights in Rome, we decide to add the Pantheon and the Trevi Fountain to our stops. But, as we can see, it matters a lot whether we are using lists or vectors to store those sights:

```
(conj '(:colosseum :vatican) :pantheon :trevi-fountain)
;;=> (:trevi-fountain :pantheon :colosseum :vatican)
(conj  [:colosseum :vatican] :pantheon :trevi-fountain)
;;=> [:colosseum :vatican :pantheon :trevi-fountain]
```

The `conj` function (short for "conjoin") is used to add one or more elements to a collection. Each collection type inserts elements in the way that is most efficient for the collection, not necessarily at the beginning or end. In other words, `conj` does not mean "append" or "prepend."

The most common point of confusion for new Clojure users is why lists (or sequences) insert elements at the beginning and vectors insert elements at the end. This all goes back to their implementation and where it is efficient to add elements.

Clojure lists or sequences are conceptually implemented as linked lists, a data structure that can be thought of as a chain where each link in the chain consists of a value and a reference to the next link. To traverse the data structure, we start at the head and walk the chain through the references to get to the end.

Inserting a new element could be done at the end, but only if you start at the head of the chain, walk through every link, and then construct a new link with the element at the end. The time to do this is proportional to the length of the list (1000 units of time for a list with 1000 elements). However, inserting at the beginning always takes a constant amount of time—you make a new link that simply refers to the head of the existing chain. `conj`ing multiple elements will do this repeatedly, so the last thing we added (`:trevi-fountain`) is the first thing in the resulting list.

Vectors, however, are stored as a tree (technically, a "hash array-mapped trie," a data structure invented by Phil Bagwell). We won't go into those details

here, but the tree is arranged in the order of the vector elements. Adding a new value can easily be done at the end by walking the right-most nodes of the tree and appending a new value to the final hash node leaf.

The time it takes to do the tree traversal is proportional to the height of the tree, and the height is the log, base 32, of the number of elements in the vector. Because this data structure uses relatively wide internal nodes, the tree is very shallow, and the height of the tree is usually only a few levels deep, usually no more than 3. Because the height grows so slowly, we sometimes call the traversal time "effectively constant."

Inserting an element at the beginning of the vector could also be relatively fast, but it would require some additional overhead due to the implementation. More importantly, it makes more semantic sense to add elements to the end of the indexed vector than the beginning (or all of the indices would change). For these reasons, vectors insert elements at the end.

Retaining Order While Transforming

Because lists and vectors have different behavior for `conj`, it is important to keep track of which one you are using at any given time in your code. Generally, because vectors are easier to type in (no quoting necessary) and append at the end, data is most commonly built up using vectors and then transformed using sequence functions like `map` and `filter`, which return lazy sequences.

For the specific cases of `map` and `filter`, there are also `mapv` and `filterv` variants of `map` and `filter` which return vectors:

```
;; with map, returns sequence
(map #(* % 3) (range 5))
;;=> (0 3 6 9 12)

;; with mapv, returns vector
(mapv #(* % 3) (range 5))
;;=> [0 3 6 9 12]
```

A more general solution is to use *transducers* and `into`. Most sequence functions provide an additional arity that omits the collection and instead returns a transducer, a function that embodies the function algorithm but decouples from the input and output sources. Transducers can be composed using `comp` into a chain of transformations applied to a source of values, like a collection.

The `into` function takes a target collection, an optional transducer, and a source collection. You can use `into` with a `map` transducer to collect the results into a vector like this:

```
(into [] (map #(* % 3)) (range 5))
;;=> [0 3 6 9 12]
```

Note that `map` here takes only the mapping function. `into` is responsible for applying the resulting transducer function to the source and storing the result into the output vector. This is more complicated than `mapv` but also more flexible (when composing multiple transducers) and potentially faster in better avoiding intermediate collections.

Sets

It's also interesting to consider adding elements to a set:

```
(conj #{:colosseum :vatican} :pantheon :trevi-fountain)
;;=> #{:vatican :trevi-fountain :colosseum :pantheon}
```

The important thing to remember about sets is that they guarantee no particular order. The actual order they are printed in ultimately derives from the hash codes of the elements, which have no obvious relationship to the values (and that order may even differ between Clojure versions). The other interesting aspect of sets is that you cannot add the same element more than once. Mathematically, a set is a set of values (duplicates are not possible):

```
(conj #{1 2 3} 4 4 4)
;;=> #{1 4 3 2}
```

You can try to add the same value multiple times, but it is only added for the first one and makes no modification after that.

Further Reading

ClojureDocs - clojure.core/conj
https://clojuredocs.org/clojure.core/conj

Clojure Reference - Data Structures
https://clojure.org/reference/data_structures

"Ideal Hash Trees" by Phil Bagwell
https://infoscience.epfl.ch/record/64398/files/idealhashtrees.pdf

Clojure Reference - Transducers
https://clojure.org/reference/transducers

Part III

Evaluation

Understanding how Clojure code is executed relies on understanding read and eval. The reader parses characters (text) and creates Clojure data representing the code. The evaluator takes Clojure code (as data) and evaluates the form, producing a result value. These teasers highlight various aspects of read and eval.