Extending and Using Specter

Get Going Good

Brian Marick

Extending and Using Specter

Get Going Good

Brian Marick

This book is for sale at http://leanpub.com/specter

This version was published on 2016-06-14

Contents

Preface: Why Specter?

Specter for selecting

Clojure has a a nice function, `get-in`, for looking deep into a data structure:

```
user=> (def x {:a [{:b 3}
                   {:b 4}
                   {:b 5 :c 3}]})
user=> (get-in x [:a 1 :b])
=> 4
```

Let's say that the second argument of `get-in` is a *selector*, which is composed of *selector elements*. For `get-in`, every element must be a key into an associative structure. (Remember that, in Clojure, vectors are associative structures with integer keys, so `1` above is the same sort of element as `:a` and `:b`.)

`get-in` is a fine function, but it's limited to traversal of associative structures to retrieve a single value (such as `4`).

Specter is more powerful than `get-in` because it has more types of selector elements. For example, `ALL` lets you select many leaves of the structure. Here's how you select all values of `:b` from the structure above:

```
user=> (select [:a ALL :b] x)
=> [3 4 5]
```

You can combine `ALL` with arbitrary predicates to reduce the number of selected values to those that satisfy the predicate:

```
user=> (select [:a ALL :b odd?] x)
=> [3 5]
```

As a final teaser, `walker` is a selector for walking code. Here's how you'd select every even integer in a nested expression:

```
user=> (select [(walker integer?) even?]
               '(+ 1 (- 2 3) (+ 4 (- 5 (+ 6 7)) 8)))
=> (2 4 6 8)
```

Specter for updating

`transform` is to `select` as `update-in` is to `get-in`: you provide a function that's given each selected element, and the result is the original input "modified" with the results of the function. The following transforms even integers by multiplying them by 1000:

```
user=> (transform [(walker integer?) even?]    ; selector
                  (partial * 1000)             ; transformer
                  '(+ 1 (- 2 3) (+ 4 (- 5 (+ 6 7)) 8)))
=> (+ 1 (- 2000 3) (+ 4000 (- 5 (+ 6000 7)) 8000))
```

As you can see from above, any of the variety of selector elements you can use with `select` can also be used with `transform`. There are also elements especially useful in `transform`. For example, consider this structure:

```
user=> (def input [ {:a 1 :x {:y 2}}
                    {:a 3 :x {:y 4}} ])
```

We can change the `:a` values with the selector elements you already know:

```
user=> (transform [ALL :a] (partial * 1000) input)
=> [{:a 1000, :x {:y 2}} {:a 3000, :x {:y 4}}]
```

... and do the same to the more-deeply-nested `:y` values:

```
user=> (transform [ALL :x :y] (partial * 1000) input)
=> [{:a 1, :x {:y 2000}} {:a 3, :x {:y 4000}}]
```

But how do we add 1000 times the `:a` values to the corresponding `:y` values? We can do that using the `collect-one` element, which stashes a value for later use. Before we get to its use with `transform`, consider `collect-one` in a `select` expression:

```
user=> (select [ALL (collect-one :a) :x :y] input)
=> [[1 2] [3 4]]
```

The result is not the `:y` values (2 and 4), but rather pairs of `:a` and `:y` values. (If you're familiar with monads, you can think of this as something like the state monad[1].)

In a `transform` expression, the transformation function is applied to each pair:

[1] https://github.com/blancas/morph/wiki/Modeling-Access-to-Global-State

```
user=> (transform [ALL (collect-one :a) :x :y]
                  (fn [a z] (+ (* 1000 a) z))
                  input)
=> [{:a 1, :x {:y 1002}} {:a 3, :x {:y 3004}}]
```

Specter is not as general-purpose as, say, zippers[2], because you can only visit a subtree once. (With zippers, you can navigate to a point in the tree and then, based on the value there, navigate backwards to revisit part of the tree.) However, many of the problems you'd otherwise use zippers for can be solved with Specter, often—I'd argue—more clearly. The above use of `collect-one` is one of many examples.

Specter for learning

In order to learn how Specter works, I decided to grow it myself, feature by feature, using the existing implementation as a guide. As I did, I kept having little shocks of recognition: "Ah, continuation-passing style. Clever!" or "Ah, recursively precomputing the call graph—I've seen that before!" I was recognizing design patterns[3], and it occurred to me that this book could *also* introduce those patterns to people who perhaps hadn't seen them before.

Originally, I planned to discuss Specter's design in a Part II of the book, about extending specter with your own selector elements. You need to understand something of the design to do that well. But as I was writing the first chapter, about the most basic Specter selectors, I realized that I could make that chapter a lot less dry by showing the design, and—in fact—by having you implement Specter yourself (or, if you prefer, follow along with a narrated implementation).

That was the moment when the title switched from *Using and Extending Specter* to *Extending and Using Specter*, and when the book became both a book about a useful tool and also a little bit of a book about software design.

[2]http://www.exampler.com/blog/2010/09/01/editing-trees-in-clojure-with-clojurezip

[3]Some say functional languages don't have design patterns. I'd argue that those people are confusing the idea of design patterns with the particular design patterns applicable to object-oriented languages.

How to Use this Book

I read some technical books while riding an exercise bike. I read others by carefully working through their exercises on my laptop. This book is designed to support both styles.

Reading-while-riding is adequate for a broad knowledge of a topic. I read Saša Jurić's *Elixir In Action*[4] on an exercise bike. As a result, I know *what* Elixir can do, but I don't remember much of *how*. To be able to actually program in Elixir, much less build a real system, I'd have to sit down and write code. Typing things at a computer and seeing what it types back really helps memory and comprehension.

Specter API exercises

In order to make Specter features better stick in your mind, I encourage you to try out exercises at the REPL. Such exercises look like this:

This is an exercise

```
(use 'com.rpl.specter)
(use 'com.rpl.specter.macros)
(select [:a] {:a 1}) ; => ??
(select [:a :b] {:a {:b 1}}) ; => ??
...
```

Because you might be reading on an exercise bike, I immediately give you the answers to the exercise. They look like this:

This is an exercise solution

specter-reference.ch1.select-kw[5]

```
(facts "Specter's behavior with keywords"
  (fact "descends a map"
    (s/select [:a] {:a 1}) => [1]
    (s/select [:a :b] {:a {:b 1}}) => [1])
...
```

[4]https://manning.com/books/elixir-in-action
[5]https://github.com/marick/specter-book-code/blob/master/src/specter_reference/ch1/select_kw.clj

That's executable Clojure for the Midje[6] testing tool. Midje is designed so that tests look like examples in Clojure books: an expression you can type on the left-hand side, a separator—very often some sort of arrow—and then the expected value following. You can think of `=>` expressions as very specific facts from which you can readily deduce a more general fact[7].

The namespace at the top of the solution is where the executable version lives in the book's Github repo (https://github.com/marick/specter-book-code[8]). You can run it if you like, using the instructions below.

Specter implementation exercises

As I said in the Preface, I encourage you to learn Specter's design more deeply by implementing it yourself. There are exercises that guide you through that (and, immediately after, my solutions). You can implement your solutions solely in the REPL, but you can also edit files that live under the `exercises` namespace. They have boilerplate code to save you some typing. They also have tests so that you can know you've gotten it right more easily than retyping the same old examples in the REPL.

Because Midje is the bee's knees[9][10], exercise templates use it. That gives you three ways to check your work. The first is to use the command line to check the specific namespace you're working in:

```
$ lein midje exercises.ch1.select-kw
= Namespace exercises.ch1.select-kw

WORK TO DO "Behaves the way specter/select does" at (select_kw.clj:8)
No facts were checked. Is that what you wanted?
```

The `WORK TO DO` message is because tests for unfinished code are marked as facts that aren't yet true:

```
(future-facts "Behaves the way specter/select does"
  (select-kw [:a] nil) => [nil]
```

Just delete the "future-" to see this:

[6]https://github.com/marick/Midje

[7]In mathy terms, the truth statements can contain only constants, like $3+3 = 2×3$. But because you're a human and good at generalizing, you can read sets of such examples and infer universally-quantified facts like $\forall y$: $y+y = 2×y$.

[8]https://github.com/marick/specter-book-code

[9]http://www.phrases.org.uk/meanings/the-bees-knees.html

[10]Well, I wrote Midje so I *would* say that, wouldn't I?

```
$ lein midje exercises.ch1.select-kw
= Namespace exercises.ch1.select-kw

FAIL "Behaves the way specter/select does" at (select_kw.clj:9)
    Expected: [nil]
      Actual: :unimplemented

FAIL "Behaves the way specter/select does" at (select_kw.clj:10)
    Expected: [nil]
      Actual: :unimplemented
...
```

I personally am not patient enough to enjoy the amount of time it takes both Clojure and Midje to get loaded and start doing work, so I usually use “autotest”. There are two ways to do that. You can do it from the command line:

```
$ lein midje :autotest
```

That will first run all the tests. Thereafter, whenever you save a changed file, it will reload it and thus run the tests again.

In my day-to-day work, I prefer to work in the REPL:

```
$ lein repl
user=> (use 'midje.repl)
user=> (autotest)
```

That starts by running all the tests, but then it gives you an ordinary REPL prompt. You can use the REPL as usual, but a background process monitors changed files and reloads them as necessary.

Thanks for the comments and corrections

- Bahadir Cambel
- Beau Fabry
- Jason Felice

Contacting me

Mail me at marick@exampler.com[11], tweet at @marick, or write to the book discussion page[12].

Please include the book’s version number as shown below.

[11]mailto:marick@exampler.com
[12]https://leanpub.com/specter/feedback

Changelog

Version 2

- Update to Specter 0.11.X (which has API changes)

Understanding Specter By Building It

The simpler selectors contain keywords, predicates, and the special token `ALL`. Here's a sample that extracts all the even values of all the `:a` keys in a vector of maps:

```
user=> (select [ALL :a even?]
               [ {:a 1} {:a 2} {:a 3} {:a 4}])
=> [2 4]
```

In this chapter, you'll see how each of those three kinds of elements work and are implemented.

Keywords

We'll start with keywords. Before you can write code that mimics Specter, you have to know what it does. To find out, start it up and try out the following expressions.

Selectors with keywords

Predict the values of the following uses of `select`.

```
(use 'com.rpl.specter)
(use 'com.rpl.specter.macros)
(select [:a] {:a 1}) ; => ??
(select [:a :b] {:a {:b 1}}) ; => ??

(select [:a] {}) ; => ??
(select [:a] {:not-a 1}) ; => ??
(select [:a :b] {:a {}}) ; => ??

(select [:a] nil) ; => ??
(select [:a] :something-random) ; => ??
(select [:a :b] {:a 1}) ; => ??
```

Alternately, you can just look here:

Selectors with keywords

specter-reference.ch1.select-kw[13]

```
(facts "Specter's behavior with keywords"
  (fact "descends a map"
    (s/select [:a] {:a 1}) => [1]
    (s/select [:a :b] {:a {:b 1}}) => [1])

  (fact "missing keyword has a value of `nil`"
    (s/select [:a] {}) => [nil]
    (s/select [:a] {:not-a 1}) => [nil]
    (s/select [:a :b] {:a {}}) => [nil])

  (fact "as with `get`, bogus structures are silently accepted"
    (s/select [:a] nil) => [nil]                ; (get nil :a) => nil
    (s/select [:a] "no-map") => [nil]           ; (get "no-map" :a) => nil
    (s/select [:a :b] {:a 1}) => [nil]))        ; (get 1 :a) => nil
```

It's interesting that Specter returns a vector of matches even though there can only one for a keyword. That makes selectors like these consistent with ones that use `ALL`.

Specter follows Clojure's somewhat idiosyncratic interpretation of keyword lookup in that you can "look up" a keyword from anything—it's never possible for `get` of a keyword to fail.

You now have a "specification", so…

Mimic `select` for keywords

exercises.ch1.select-kw[14] (See the previous chapter for how to use this.)

Write a function for which the previous facts are true. Call it `select-kw`. Here's its starting definition:

```
(defn select-kw [selector structure]
  :unimplemented)
```

Remember that the return value must be a vector.

Here's my solution:

[13] https://github.com/marick/specter-book-code/blob/master/src/specter_reference/ch1/select_kw.clj
[14] https://github.com/marick/specter-book-code/blob/master/src/exercises/ch1/select_kw.clj

select-kw

book-code.ch1.select-kw[15]

```
(defn select-kw [selector structure]
  (if (empty? selector)
    (vector structure)
    (select-kw (rest selector) (get structure (first selector)))))
```

This is a recursive solution, which is somewhat un-idiomatic Clojure[16]. The idiomatic `loop ... recur` solution doesn't transform as smoothly into the way Specter actually works.

Predicates

Try out the following examples.

How are predicates different?

Predict the values of the following uses of `select`:

```
(use 'com.rpl.specter)
(use 'com.rpl.specter.macros)
(select [odd?] 1)   ; => ??
(select [even?] 1)  ; => ??

(select [#(get % :a)] {:a 1})   ; => ??

(select [integer? odd?] 1)    ; => ??
(select [integer? even?] 1)   ; => ??
(select [integer? odd?] "hi") ; => ??
```

[15]https://github.com/marick/specter-book-code/blob/master/src/book_code/ch1/select_kw.clj

[16]When a function calls itself as the very last action in a path through its code, that's called *tail-recursive*. In many languages, the compiler automatically converts those calls into more efficient loops. Clojure doesn't do that, which is why it has a special `loop ... recur` construct.

Selectors with predicates

specter-reference.ch1.select-pred[17]

```
(facts "reference behavior for predicates"
  (fact "predicate success: the value is passed through to the result vector"
    (s/select [odd?] 1) => [1])

  (fact "Predicates don't have to be strictly true or false"
    (s/select [#(get % :a)] {:a 1}) => [{:a 1}])

  (fact "predicate failure:, there is no result vector"
    ;; Note that this is different from the behavior of keywords:
    ;; there, a missing key results in `nil` being passed through
    (s/select [even?] 1) => nil)

  (fact "given multiple predicates, each passes judgment in turn"
    (s/select [integer? odd?] 1) => [1]
    (s/select [integer? even?] 1) => nil
    ;; Note that the first predicate's failure prevents the second
    ;; from being checked. (It would throw an exception if it were.)
    (s/select [integer? odd?] "hi") => nil))
```

The interesting difference between predicates and keywords is that keywords *select*, whereas predicates *filter*. As a result of that, a predicate selector can return a "nothing" (`nil`) value. Keywords always return some kind of vector.

Mimic `select` for predicates

exercises.ch1.select-pred[18]

Start from this version:

```
(defn select-pred [selector candidate]
  :unimplemented)
```

This is my solution:

[17]https://github.com/marick/specter-book-code/blob/master/src/specter_reference/ch1/select_pred.clj
[18]https://github.com/marick/specter-book-code/blob/master/src/exercises/ch1/select_pred.clj

`select` for predicates

book-code.ch1.select-pred[19]

```
(defn select-pred [selector candidate]
  (if (empty? selector)
    (vector candidate)
    (if ( (first selector) candidate)
      (select-pred (rest selector) candidate)
      nil)))
```

To illustrate the difference between the two solutions, here they are, combined:

```
(defn select-both [[x & xs :as selector] structure]
  (cond (empty? selector)
        (vector structure)

        (keyword? x)
        (select-both xs (get structure x))

        (fn? x)
        (if (x structure)
          (select-both xs structure)
          nil)))
```

In both cases, the last thing the function does when it runs out of selector is to wrap the final result in a vector. Both variants are recursive, but the `fn?` clause has an extra escape hatch: it returns `nil` on failure (not wrapping that result in a vector). It also just passes the original `structure` along, rather than a piece of it (as the keyword case does).

Extensibility

Our `select-both` isn't a very good design because it makes it awkward to add new kinds of selector elements. The typical Clojure approach to this sort of extensibility is *protocols*. [[[Add short appendix on protocols.]]] Here's a protocol that can be specialized to each kind of selector element:

[19] https://github.com/marick/specter-book-code/blob/master/src/book_code/ch1/select_pred.clj

```
(defprotocol Navigator
  (select* [this remainder structure]))
```

Its signature looks almost the same as that of `select-both`, except that the two parts of the selector are passed in separately instead of destructured with `[x & xs :as selector]` That's required because protocol functions are specialized according to the type of the first argument. Now a generic `select` can be written like this[20]:

```
(defn select [[x & xs :as selector] structure]
  (if (empty? selector)
    (vector structure)
    (select* x xs structure)))
```

… and the protocols can be implemented like this:

```
(extend-type clojure.lang.Keyword
  Navigator
  (select* [this remainder structure]
    (select remainder (get structure this))))

(extend-type clojure.lang.AFn
  Navigator
  (select* [this remainder structure]
    (if (this structure)
      (select remainder structure)
      nil)))
```

That works, but it's ugly:

1. `select*` takes a `remainder` argument that it does nothing with but pass along.
2. `select` knows about `select*`, and `select*` in turn knows about `select` (because it calls it). Smearing information throughout an implementation is generally a poor idea.

Let's start working on these problems by looking at the stack trace for a call to `select` with selector `[:a :b]`. Time flows downward here:

[20]https://github.com/marick/specter-book-code/blob/master/src/book_code/ch1/klunky_protocols.clj

```
(select          [:a    :b]             {:a {:b 1}})
  (select*        :a   [:b]             {:a {:b 1}})
    (select            [:b]                 {:b 1})
      (select*          :b   []             {:b 1})
        (select              []                 1)
          (vector                               1)
```

An interesting thing is that the third argument is irrelevant to how the stack trace evolves. Here's the trace for `select [:a :b] {:a {}}`:

```
(select          [:a    :b]             {:a {   }}
  (select*        :a   [:b]             {:a {   }}
    (select            [:b]                 {   }
      (select*          :b   []             {   }
        (select              []               nil)
          (vector                             nil)
```

That suggests something of a crazy idea. Suppose we convert this two-argument function call:

```
(select [:a :b] {:a {}})
```

... into a *pair* of one-argument calls:

```
( (predict-select-computation [:a :b])
  {:a {}})
```

`predict-select-computation` will convert `[:a :b]` into something with roughly this computational shape:

```
(fn [structure] ... (select* :a ... (select* :b ... (vector ...
```

We can implement that using a design pattern called *continuation-passing style*. It works like this:

Consider the expression `(odd? (inc (+ 2 3)))`. We can express that as a series of `let` steps:

```
(let [result (+ 2 3)
      result (inc result)
      result (odd? result)]
  result)
```

We can generalize that: any computation can be expressed as "evaluate some subexpression, then do the rest of the computation with the result of that subexpression".

Now, `let` isn't a required part of a functional language. You could take it completely out of Clojure and lose no computational power. That's because anything you can do with `let`, you can do with `fn`. The following computes the same result:

```
  (-> (+ a b)
      ((fn [result] (-> (inc result)
                        ((fn [result] (-> (odd? result)
                                          ((fn [result] result))))))))))
```

Alternately, you could also remove the use of the `->` macro and get pure, unadorned, read-them-backwards function calls:

```
  ((fn [result] result)
   ((fn [result] (odd? result))
    ((fn [result] (inc result))
     (+ a b)))))
```

I prefer the first form because it more obviously suggests a simplification. What if, instead of using the `->` macro to feed a result into another function, we gave every `clojure.core` function a variant with an additional argument: a function that represents "the rest of the computation". (Such a function is typically called a *continuation*). If you'll allow me to pretend `+` only takes two arguments, we'd have new functions like these:

```
(defn cont-+ [a b continuation]
  (continuation                   (+ a b)))
(defn cont-inc [n continuation]
  (continuation                   (inc n)))
(defn cont-odd? [n continuation]
  (continuation                   (odd? n)))
```

We can now write `(odd? (inc (+ 2 3)))` like this:

```
(cont-+ 2 3
        (fn [result]
          (cont-inc result
                    (fn [result]
                      (cont-odd? result
                                 identity)))))
```

Well. We're obviously not going to give every `clojure.core` function an alter ego that takes a continuation, so how is this relevant to our goal?

Like this: what if we changed the signature of `select*` from this:

```
(select* [this remainder structure]))
```

... to this:

```
(select* [this structure continuation]))
```

Then the result of `(predict-select-computation [:a :b])` could be this:

```
(fn [structure]
  (select* :a structure
           (fn [structure]
             (select* :b structure
                      vector))))
```

Once `predict-select-computation` is implemented, we'll have eliminated the deficiencies of the previous version:

1. There's no need to explicitly pass around the `remainder` of the selector. Indeed the idea of the selector has completely vanished from ("been compiled out of") the code.
2. We've also lost the intermediate call to `select`. Its job—determining if we were at the end of the computation and need to return a vector of the result—has been computed and turned into the use of `vector` as the final continuation.

But wait! There's more:

1. In Specter proper, though perhaps not in this implementation (at least not yet), precomputing a path function and then using it is substantially faster than building the same computational structure at runtime. ("Substantially" here can mean a more than 10X speedup for some paths and some data, and a 3X speedup for simpler cases.)
2. If we're going to reuse the `[:a :b]` path with many structures, we can avoid recomputing it each time. Instead of this:

   ```
   (map (partial select [:a :b]) [...]
   ```

   ... we can have this:

   ```
   (map (predict-select-computation [:a :b]) [...])
   ```

   ... giving even more efficiency gains.

How Specter Does This

Specter has a function akin to `predict-select-computation`. It's called `comp-paths`. Unlike our function, it can be used with both `select` and `transform`. Because of that, it doesn't return a function but rather data. You use it like this:

```
user=> (def ab (comp-paths :a :b)) ; Note the path isn't enclosed in a vector
user=> (type ab)
=> com.rpl.specter.impl.CompiledPath
user=> (compiled-select ab {:a {:b 1}})
=> [1]
user=> (compiled-transform ab (partial * 1000) {:a {:b 1}})
=> {:a {:b 1000}}
```

Implement the `predict-select-computation`

exercises.ch1.continuation-passing[21]

Implement a function that produces a function that takes a structure and executes a predetermined set of calls to `select*`. It will be used by `select` like this:

```
(defn select [selector structure]
  ((predict-select-computation selector) structure))
```

Hint: You'll use `reduce` to convert a set of selector elements into a single (nested) set of continuation-passing functions. Each step of the reduction wraps a "continuation-so-far" into a structure like this:

```
(fn [structure]
  (select* element structure continuation)))
```

A solution:

[21]https://github.com/marick/specter-book-code/blob/master/src/exercises/ch1/continuation_passing.clj

predict-select-computation

predict-select-computation[22]

```
(defn predict-select-computation [selector]
  (reduce (fn [continuation element]
            (fn [structure]
              (select* element structure continuation)))
          vector
          (reverse selector)))
```

Predictive vs. runtime protocol lookup

`predict-select-computation` produces a frozen computation like this:

```
(fn [structure]
  (select* :a
           structure
           (fn [structure]
             (select* even?
                      structure
                      vector)))))
```

`select*` is a function polymorphic in the type of its first argument. That means we can think of it as having, in effect, this implementation:

```
;; Forget about protocol Navigator

(defn selector-function-for [this]
  (cond (keyword? this)
        (fn [this structure continuation]
          (continuation (get structure this)))

        (fn? this)
        (fn [this structure continuation]
          (if (this structure)
            (continuation structure)
            nil))))
```

[22] https://github.com/marick/specter-book-code/blob/master/src/book_code/ch1/continuation_passing.clj

```
;; There's now only one version of select*:
(defn select* [this & args]
  (apply (selector-function-for this) this args))
```

But every call to `selector-function-for` is predetermined by the specific selector given to `predict-select-computation`. Given the selector `[:a even?]`, we know that the first argument to the first `select*` is `:a`, and the first argument to the second `select*` is `even?`. So why shouldn't `selector-function-for` do the computation and save `select*` the trouble?

That is, instead of building this:

```
(fn [structure]
  (select* :a
           structure
           (fn [structure]
             (select* even?
                      structure
                      vector)))))
```

… we want to build something with specific `select*` functions substituted in:

```
(fn [structure]
  (   (fn [this structure continuation]
        (continuation (get structure this)))
      :a structure
      (   (fn [this structure continuation]
            (if (this structure)
              (continuation structure)
              nil))
         even? structure
         vector)))
```

That's certainly more … cryptic … but is it worth the trouble?

As it turns out, yes. In some *extremely* crude tests, I saw a speedup of around 2.8X.

The specific details of how Specter goes from a particular value (like the keyword `:a` or the function `even?`) to a raw selection function aren't particularly relevant here, and they use some undocumented (but public) `clojure.core` functions. If you still want to see them, I've written a simplified version of Specter's implementation[23].

The important thing here is that the protocol and signature of `select*` haven't changed, but Specter has still managed to squeeze out a substantial speedup.

[23]https://github.com/marick/specter-book-code/blob/master/src/book_code/ch1/realistic.clj

ALL

Specter also supplies an `ALL` selector element. Try out these examples to see how it works:

ALL

```clojure
(use 'com.rpl.specter)
(use 'com.rpl.specter.macros)
(select [ALL] [1 2 3 4]) ; => ??

(select [ALL ALL] [[1] [2 3]] ; => ??
(select [ALL ALL] [[0] [[1 2] 3]]) ; => ??
(select [ALL ALL] [[1] nil [] [2]]) ; => ??

(select [ALL :a] [{:a 1} {:a 2}]) ; => ??
(select [ALL even?] [1 2 3 4]) ; => ??
(select [ALL :a even?] [{:a 1} {:a 2}]) ; => ??
```

ALL

specter-reference.ch1.select-kw[24]

```
(fact "all by itself is a no-op"
  (s/select [s/ALL] [1 2 3 4]) => [1 2 3 4])

(fact "Since ALL 'spreads' the elements, it can be used to flatten"
  (s/select [s/ALL s/ALL] [ [1] [2 3] ])
  =>                      [  1   2 3   ]

  (fact "... but it won't flatten deeper than the level of nesting"
    (s/select [s/ALL s/ALL] [[0] [[1 2] 3]])
    =>                      [ 0   [1 2] 3])

  (fact "both nil and an empty vector are flattened into nothing"
    (s/select [s/ALL s/ALL] [[1] nil [] [2]])
    =>                      [ 1          2]))


(fact "ALL applies the rest of the selector to each element"
  (s/select [s/ALL :a] [{:a 1} {:a 2} {   }])
  =>                   [    1      2   nil ]
  (s/select [s/ALL even?] [1 2 3 4])
  =>                      [  2   4]
  (s/select [s/ALL :a even?] [{:a 1} {:a 2}])
  =>                         [           2 ])
```

I think of `ALL` as mapping the `rest` of the selector over each element of the structure it receives, then glueing all the results together. It seems like that could be easily done: just add a `select*` protocol function to `ALL`, much as we've done on keywords and functions. And indeed it is, once a few formalities are taken care of:

1. Since protocols are selected by the type of an value, we must arrange for `ALL` to be of a distinct type (as keywords and functions are). That can be done like this:

   ```
   (def ALL (->AllType)) ; `ALL` is the only instance of `AllType`
   ```

2. We need to define `ALL`'s type like this:

[24] https://github.com/marick/specter-book-code/blob/master/src/specter_reference/ch1/select_all.clj

```
(deftype AllType [])
```

3. Finally, we need to add the `AllType` implementation of `select*`

   ```
   (extend-type AllType
     Navigator
     (select* [this structure continuation]
       ...))
   ```

   (If you're fluent with protocols, you may wonder why we don't define `select*` when we define the type:

   ```
   (deftype AllType []
     Navigator
     (select* [this structure continuation]
       ...))
   ```

   The reason is that the implementation-dependent lookup of specific protocol functions doesn't work for functions defined that way.)

All that given, …

Implement ALL

exercises.ch1.all[25]

Be sure that the results of a `select` using `ALL` returns a vector, not a lazy sequence.

Here's my solution:

The implementation of ALL

book-code.ch1.all[26]

```
(extend-type AllType
  Navigator
  (select* [this structure continuation]
    (into [] (mapcat continuation structure))))
```

Specter takes another stab at increasing performance by using reducers[27] for performance. We won't worry about that in this book.

[25] https://github.com/marick/specter-book-code/blob/master/src/exercises/ch1/all.clj
[26] https://github.com/marick/specter-book-code/blob/master/src/book_code/ch1/all.clj
[27] http://blog.guillermowinkler.com/blog/2013/12/01/whats-so-great-about-reducers/