Clojure For The Brave and True Part – IV Core Functions In Depth

Agung Tuanany

2025-09-02 Tue

if you're fan of the angsty, teenager-centric, quasi-soap opera *the vampire diaries* like i am, you'll remember the episode where the lead protagonist, elena, starts to question her pare, mysterious crush's behavior: ”why did he instantly vanish without a trace when i scraped my knee?” and ”how come his face turned into a grotesque mask of death when i nicked my finger” and so on.

you might be asking yourself similar question if you've started playing with clojure's core functions, ”why did *map* return a list when i gave it a vector?” and ”how come *reduce* treats my map like a list if vectors?” and so on. (with clojure, though, you're at least spared from contemplating the profound existential horror of being 17-year-old for eternity.)

in this chapter, you'll learn about clojure's deep, dark, bloodthirsty, supernatur–**cough**, i mean, in this chapter you'll learn about clojure's underlying concept of *programming to abstractions* and about the sequence and collection abstractions. you'll also learn about *lazy sequences.* this will give you the grounding you need to read the documentation for functions you haven't used before and to understand what's happening when you give them a try.

next, you'll get more experience with the function you'll be reaching for the most. you'll learn how to work with list, vectors, maps, and sets with the function *map*, *reduce*, *into*, *conj*, *concat*, *concat*, *some*, *filter*, *take*, *drop*, *sort*, *sort-by*, and *identity.* you'll also learn how to create new functions with *apply*, *partial*, and *complement.* all this information will help you understand how to do things the clojure way, ans it will give you a solid foundation for writing your own code as well as for reading and learning form others' projects.

finally, you'll learn how to parse and query a csv of vampire data to determine what nosferatu lurk in your hometown.

Programming to Abstractions

To understand programming abstractions, let's compare Clojure to a language that wasn't built with that principle in mind: Emacs (elisp). In elisp, you can use the *mapcar* function to derive a new list, which is similar to how you use *map* in Clojure. However, if you want to map over a hash map (similar to Clojure's map data structure) in elisp, you'll need to use the *maphash* function, whereas in Clojure you can still just use *map.* In other words, elisp uses two different, data structure-specific functions to implement the *map* operation, but Clojure uses only one. You can also call *reduce* on a map in Clojure, whereas elisp doesn't provide a function for reading a hash map.

The reason is that Clojure defines *map* and *reduce* functions in terms of the *sequence abstraction*, not in terms of specific data structures. As long as a data structure responds to the core sequence operations (the functions *first*, *rest*, and *cons*, which we'll look at more closely in a moment), it will work with *map*, *reduce*, and oodles other sequence of other sequence functions for free. This is what Clojurists mean by programming to abstractions, and it's a central tenet of Clojure philosophy.

I think of abstractions as named collections of operations. If you can perform all of an abstraction's operations on an object, then that object is an instance of the abstraction. I think this way even outside of programming. For example, the *battery* abstraction includes the operation "connect a conducting medium to its anode and cathode," and the operation's output is *electrical current.* It doesn't matter if the battery is made out of lithium or out of potatoes. It's a battery as long as it responds to the set of operations that define *battery.*

Similarly, *map* doesn't care about how lists, vectors, sets, and maps are implemented. It only cares about whether it can perform sequence operations on them. Let's look at how *map* is defined in terms of the sequence abstraction so you can understand programming to abstraction in general.

NOTE

- Clojure is built around the idea abstraction –> meanings:

  > "Clojure doesn't care something built, only care what it can do"

- What is a "Sequence abstraction"?

  Think of like a shared language.

  As long as a data structure can answer three simple question:

1. **first** – What's your first element?
2. **rest** – What's left after the first element?
3. **conj** – Can I and something to the front of you?

...then it can be treated as a sequence.

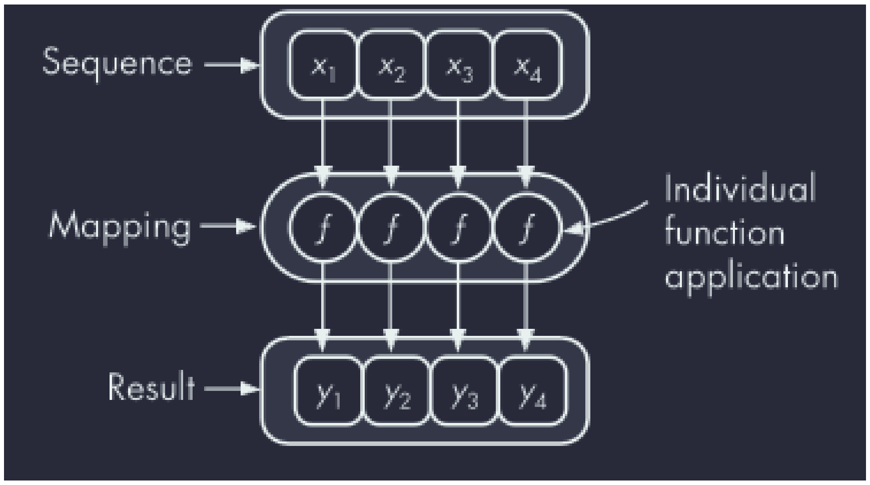
That means any function that works on sequences (*map*, *reduce*, etc.) will automatically work on it.

In very simple words:

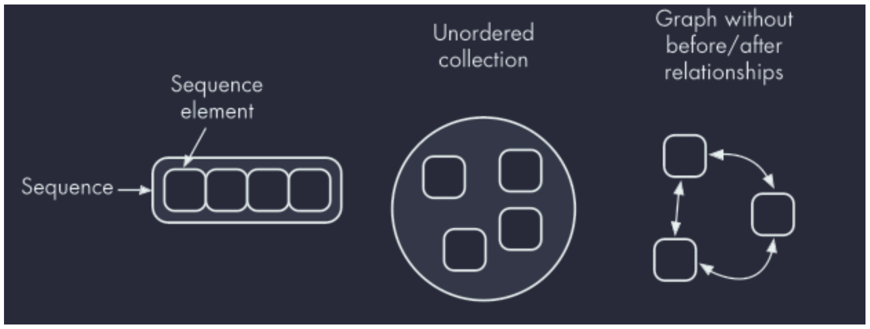
- Elisp: different functions for different data structures (mapcar, maphash).
- Clojure: one function (*map*) works for all, thanks to **sequence abstractions**.
- Abstraction = "If you support operations, I don't care what you are, I'll treat you the same."

Treating Lists, Vectors, Sets, and Maps as Sequences

If you think about the *map* operation independently of any programming language, or even of programming althogether, its essential behavior is to derive a new sequence y from an existing sequence x using of a function f such that $y_1 = f(x_1), y_2 = f(x_2)...y_n = f(xn)$. Figure 4-1 illustrates how you might visualize a mapping applied a mapping applied to a sequence.



The term *sequence* here refers to a collection of elements organized in linear order, s opposed to, say, an unordered collection or a graph without a *before-and-after* relationship between its nodes. Figure 4-2 shows how you might visualize a sequence, in contrast to the other two collections mentioned.



Absent from this description of mapping and sequence is any mention of lists, vectors, or other concrete data structures. Clojure is designed to allow us to think and program in such abstract terms as much as possible, and it does this by implementing functions in term of data structure abstractions. In this case, *map* is defined in terms of the sequence abstraction. In conversation, you would say *map*, *reduce*, and other sequence functions *take a sequence* or even *take a seq*. In fact, Clojurists usually use *seq* instead of *sequence*, using terms like *seq functions* and the *seq library* to refer to functions that perform sequential operations. Whether you use *sequence* or *seq*, you're indicating that data structure is in its truest heart of hearts doesn't matter in this context.

If the core sequence function *first*, *rest*, and *cons* work on a data structure, you can say the data structure *implements* the sequence abstraction. List vectors, sets, and maps all implement the sequence abstraction, so they all work with *map*, as shown here:

```
(defn titleize
  [topic]
  (str topic " for the Brave and True"))

(map titleize ["Hamsters" "Ragnarok"])
;; => ("Elbows for the Brave and True" "Soap Carving for the Brave and True")
(map titleize ["Empathy" "Decorating"])
;; => ("Empathy for the Brave and True" "Decorating for the Brave and True")
```

```
(map titleize #{"Elbows" "Soap Carving"})
;; => ("Elbows for the Brave and True" "Soap Carving for the Brave and True")
(map #(titleize (second %)) {:uncomfortbale-thing "Winking"})
;; => ("Winking for the Brave and True")
```

first, rest, and cons

In this section, we'll take a quick detour into JavaScript to implement a linked list and three core functions: *first*, *rest*, and *cons*. After those three core functions are implemented, I'll show how to you build *map* with them.

The point is to appreciate the distinction between the *seq* abstraction is Clojure and the concrete implementation of a linked list. It doesn't matter how a particular data structure is implemented: when it comes to using *seq* functions on a data structure, all Clojure asks is "can I *first*, *rest*, and *cons* it?" If the answer is yes, you can use the *seq* library with data structure.

In a linked list, nodes are linked in a linear sequence. Here's how you might create one in JavaScript. In the snippet, *next* is null because this is the last node in the list:

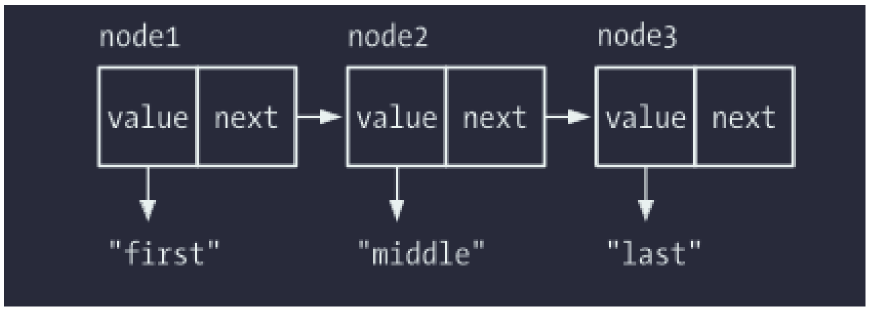
```
var node3 = {
    value: "last",
    next: null
};
```

In this snippet, *node2's* next point to *node3*, and *node1's* next point to *node2*; that the "link" in "linked list":

```
var node2 = {
    value: "middle",
    next: node3
};

var node1 = {
    value: "first",
    next: node2
};
```

Graphically, you could represent this list as show in Figure 4-3.



You can perform three core functions of a linked list: *first*, *rest*, and *cons*. *first* returns the value for the requested node, *rest* returned the remaining values after the requested node,and *cons* adds new node with the given value to the beginning of the list. After those are implemented, you can implement *map*, *reduce*, *filter*, and other *seq* function on top of them.

The following code shows how we would implement and use *first*, *rest*, *cons* with our JavaScript node example, as well as now to use them to return specific nodes and derive a new list. Note that the parameter of *first* and *rest* is named *node*. This might be confusing because you might say, "Ain't I getting first element of a *list*?" Well, you operate on the elements of a list one node at a time!.

```
var node3 = {
    value: "last",
    next: null
};

var node1 = {
    value: "first",
    next: node2
};

var node2 = {
    value: "middle",
    next: node3
};


var first = function(node) {
    return node.value;
};

var rest = function(node) {
    return node.next;
};

var cons = function(newValue, node) {
    return {
        value: newValue,
        next: node
    };
};

// first(node1);
console.log(first(node1))
// => middle
console.log(first(node2))
// => undefined
```

As noted previously, you can implement *map* in terms of *first*, *rest*, and *cons*:

```
var map = function (list, transform) {
    if (list === null) {
        return null;
    } else {
        return cons(transform(first(list)), map(rest(list), transform))
    }
};
```

This function transforms the first element of the list and then calls itself again on the rest of the list until it reaches the end (a null). Let's see it in action! In this example, you're mapping the list that begins with *node1*, returning a new list where the string " mapped!" is appended to each node's value. Then you're using *first* to return the first node's value:

```
first (
    map(node1, function (val) {return val + " mapped!"})
);
```

You can evaluate the complete code is in here.

So here's the cool thing: because *map* is implemented completely in term of *cons*, *first*, and *rest*, you could actually pass it any data structure and it would work as long as *cons*, *first*, and *rest* work on that data structure.

Here's how they might work for an array:

```
    var first = function (array) {
        return array[0];
    }

     var rest = function (array) {
        var sliced = array.slice(1, array.length);
        if (sliced.length == 0) {
            return null;
        } else {
            return sliced;
        }
    }

    var cons = function (newValue, array) {
        return [newValue].concat(array)
    }

    var map = function (list, transform) {
      if (list == null) {
          return null;
      } else {
          return cons(transform(first(list)), map(rest(list), transform));
      }
  };

    var list = ["Translvania", "Forks", "WA"];

    map(list, function (val) {return val + " mapped!"})

console.log(map(list, function (val) {return val + " mapped!"}))
// => : ['Translvania mapped! '(\, 'Forks) mapped! '(\, 'WA) mapped! '(\, null)]
```

You can evaluate above code here.

This code snippet defines *first*, *rest*, *cons* in terms of JavaScript's array functions. Meanwhile, *map* continues referencing functions named *first*, *rest*, and *cons*, so now it works on *array.* So, if you can just implement *first*, *rest*, and *cons*, you get *map* for free along with the aforementioned oodles of other functions.

Abstraction Through Indirection

At this point, you might object that I'm just kicking the can down the road because we're still left with the problem of how a function like *first* is able to work with different data structures. Clojure does this using two forms of indirection. In programming, *indirection* is generic term for the mechanisms a language employs so that one name can have multiple, related meanings. In this case, the name *first* has multiple, data structure-specific meanings. Indirection is what makes abstraction possible.

Polymorphism is one way that Clojure provides indirection. I don't want to get lost in the details, but basically, polymorphic functions dispatch to different function bodies based on the type of the argument supplied. (It's not so different from how multiple-arity functions dispatch to different function bodies based on the number of arguments you provide.)

NOTE:

Clojure has two constructs for defining polymorphic dispatch: the host platform's interface construct and platform-independent protocol. But it's not necessary to understand how these work when you'rejust getting started. I'll cover protocols in Chapter 13.

When it comes to sequences, Clojure also create indirection by doing a kind of lightwight type conversion, producing a data structure that works with an abstraction's functions. Whenever Clojure expects a sequence–for example, when you call *map*, *first*, *rest*, or *cons*–it calls the *seq* function on the data structure in question to obtain a data structure that allows for *first*, *rest*, and *cons*

```
(seq '(1 2 3))
;; => (1 2 3)

(seq [1 2 3])

;; => (1 2 3)

(seq #{1 2 3})
;; (1 3 2)
```

```
(seq {:name "Bill Compton" :ocupation "Dead mopey guy"})

;; => ([:name "Bill Compton"] [:ocupation "Dead mopey guy"])
```

There are two notable details here. First, *seq* always returns a value that looks and behaves like a list; you'd call this value a *sequence* or *seq*. Second, the *seq* of a map consists of two-element key-value vectors. That's why *map* treats your maps like lists of vectors! You can see this in the ”Bill compton” example. I wanted to point out this example in particular because it might be surprising and confusing. It was for me when I first started using Clojure. Knowing these underlying mechanisms will spare you from the kind of frustration and general mopiness[1] often exhibited by male vampires trying to retain and their humanity.

You can convert the *seq* back into a map by using *into* to stick the result into an empty map (you'll look at *into* closely later):

```
(into {} (seq {:a  1 :b 2 :c 3}))

;; => {:a 1, :b 2, :c 3}
```

So, Clojure's sequence function use *seq* on their arguments. The sequence function are defined in terms of the sequence abstraction, using *first*, *rest*, and *cons*. As long as a data structure implements the sequence abstraction, it can use the extensive *seq* library, which includes such superstar function as *reduce*, *filter*, *distinct*, *group-by*, and dozens more.

The takeaway here is that it's powerful to focus on what we can *do* with a data structure and to ignore, as much as possible, its implementation. Implementations rarely matter in and of themselves. They're just a means to and end. In general, programming to abstractions gives you power by letting you use libraries of functions on different data structure regardless of how those data structures are implemented.

Seq Function Examples

Clojure's *seq* libraries is full of useful functions that you'll use all the time. Now that you have a deeper understanding of Clojure's sequence abstraction, let's look at these functions in detail. If you're new to Lisp and functional programming, these examples will be suprising and delightful.

[1]sadness

map

You've seen many examples of *map* by now, but this section shows *map* doing two new tasks: [1] taking multiple collections as arguments and [2] taking a collection of functions as an argument. it also highlights a common *map* pattern: using keywords as the mapping function.

so far, you've only seen examples of *map* operating on one collection. in the following code, the collection is the vector *[1 2 3]*:

```
(map inc [1 2 3])
```

```
(2 3 4)
```

however, you can also give *map* multiple collections. here's a simple example to show how this works:

```
(map str ["a" "b" "c"] ["a" "b" "c"])
```

```
("aa" "bb" "cc")
```

it's a *map* if does the following:

```
(list (str "a" "a") (str "b" "b") (str "c" "c"))
```

```
("aa" "bb" "cc")
```

when you pass *map* multiple collections, the elements of the first collection *(["a" "b" "c"])* will be passed as the first argument of the mapping function *(str)*, the elements of the second collection *(["a" "b" "c"]) will be passed as the second argument, an so on. just be sure that your mapping function can take a number of arguments equal to the number of collections you're passing to /map.*

the following example show how you could use this capability if you were a vampire trying to curb your human consumption. you have two vectors, one representing human intake in liters and another representing critter intake for the past four days. the *unify-diet-data* function takes a single day's data for both human and critter feeding and unifies the two into a single map:

```
(def human-consumption   [8.1 7.3 6.6 5.0])
(def critter-consumption [0.0 0.2 0.3 1.1])

(defn unify-diet-data
  [human critter]
  {:human human
   :critter critter})

(map unify-diet-data human-consumption critter-consumption)
```

```
({:human 8.1, :critter 0.0} {:human 7.3, :critter 0.2} {:human 6.6, :critter 0.3} {:h
```

good job laying off the human!

another fun thing you can do with *map* is pass it a collection of functions. you could use this if you wanted to perform a set of calculations on different collections of numbers like so:

```
(def sum #(reduce + %))
(def avg #(/ (sum %) (count %)))

(defn stats
  [numbers]
  (map #(% numbers) [sum count avg]))

(stats [3 4 10])
;; => (17 3 17/3)

(stats [80 1 44 13 6])
;; => (144 5 144/5)
```

in this example, the *stats* function iterates over a vector of functions, applying each function to *numbers*.

additionally, clojurists often use *map* to retrieve the value associated with a keyword form a collection of map data structures. because keywords can be used as functions, you can do this succinctly. here's an example:

```
(def identities
  [{:alias "batman"       :real "bruce wayne"}
   {:alias "spider-man"   :real "peter parker"}
   {:alias "santa"        :real "your mom"}
   {:alias "easter bunny" :real "your dad"}])

(map :real identities)


("bruce wayne" "peter parker" "your mom" "your dad")
```

reduce

chapter 3 showed how *reduce* processes each element in a sequence to build a result. this section shows a couple of other ways to use it that might not be obvious.

the first use is to transform a map's values, producing a new map with the same keys but with updated values:

```
(reduce (fn [new-map [key val]]
          (assoc new-map key (inc val)))
        {}
        {:max 30 :min 10})


{:max 31, :min 11}
```

in this example, *reduce* treats the argument *{:max 30 :min 10}* as a sequence of vectors, like *([:max 30] [:min 10]).* then, it starts with an empty map (the second argument) and builds it up using the first argument, an anonymous function. it's as if *reduce* does this:

```
(assoc (assoc {} :max (inc 30))
       :min (inc 10))


{:max 31, :min 11}
```

another use for *reduce* is to filter out keys from a map based on their value. in the following example, the anonymous function checks whether the value of key-value pair is greather than 4. if it isn't, then the key-value pair is filtered out. in the map *{:human 4.1 :critter 3.9}*, 3.9 is less that 4, so the *:critter* key and its 3.9 value are filtered out.

```
(reduce (fn [new-map [key val]]
          (if (> val 4)
            (assoc new-map key val)
            new-map))
        {}
        {:human 4.1
         :critter 3.9})
```

```
{:human 4.1}
```

the takeaway here is that *reduce* is a more flexible function it first appears. whenever you want to derive a new value from a seqable data structure, *reduce* will usually be able to do what you need. if you want an exercise that will really bow your hair back, try implementing *map* using *reduce*, and then do the same for/filter/ and *some* after you read about then in this chapter.

take, drop, take-while, and drop-while

take and *drop* both take two arguments: a number and sequence. *take* returns the first *n* elements of the sequence, whereas *drop* returns the sequence with the first *n* elements removed:

```
(take 3 [1 2 3 4 5 6 7 8 9 10])
```

```
(1 2 3)
```

```
(drop 3 [1 2 3 4 5 6 7 8 9 10])
```

```
(4 5 6 7 8 9 10)
```

their cousins *take-while* and *drop-while* are bit more interesting. each takes *predicate function* (a function whose return is evaluated for truth or falsity) to determine when it should stop taking or dropping.

suppose, for example, that you ad a vector representing entries in your ”food” journal. each entry has the year, month, day, and what you ate. to preserve space, we'll only include a few entries:

```
(def food-journal
  [{:month 1 :day 1 :human 5.3 :critter 2,3}
   {:month 1 :day 2 :human 5.1 :critter 2.0}
   {:month 2 :day 1 :human 4.9 :critter 2.1}
   {:month 2 :day 2 :human 5.0 :critter 2.5}
   {:month 3 :day 1 :human 4.2 :critter 3.3}
   {:month 3 :day 2 :human 4.0 :critter 3.8}
   {:month 4 :day 1 :human 3.7 :critter 3.8}
   {:month 4 :day 2 :human 3.7 :critter 3.6}])
```

with *take-while*, you can give retrieve just january's and february's data. *take-while* traverses the given sequence (in this case, *food-journal*), applying the predicate function to each element.

this example use the anonymous function *#(< (:month %)3)* to test whether the journal entry's month is out of range:

```
(def food-journal1
  [{:month 1 :day 1 :human 5.3 :critter 2.3}
   {:month 1 :day 2 :human 5.1 :critter 2.0}
   {:month 2 :day 1 :human 4.9 :critter 2.1}
   {:month 2 :day 2 :human 5.0 :critter 2.5}
   {:month 3 :day 1 :human 4.2 :critter 3.3}
   {:month 3 :day 2 :human 4.0 :critter 3.8}
   {:month 4 :day 1 :human 3.7 :critter 3.9}
   {:month 4 :day 2 :human 3.7 :critter 3.6}])

(take-while #(< (:month %)3) food-journal1)


({:month 1, :day 1, :human 5.3, :critter 2.3}
{:month 1, :day 2, :human 5.1, :critter 2.0}
{:month 2, :day 1, :human 4.9, :critter 2.1}
{:month 2, :day 2, :human 5.0, :critter 2.5})
```

when *take-while* reaches the first march entry, the anonymous function return false, and *take-while* returns a sequence of every element it tested until that point.

the same idea applies with *drop-while* except that it keeps dropping elements until one test true:

```
  (def food-journal1
    [{:month 1 :day 1 :human 5.3 :critter 2.3}
     {:month 1 :day 2 :human 5.1 :critter 2.0}
     {:month 2 :day 1 :human 4.9 :critter 2.1}
     {:month 2 :day 2 :human 5.0 :critter 2.5}
     {:month 3 :day 1 :human 4.2 :critter 3.3}
     {:month 3 :day 2 :human 4.0 :critter 3.8}
     {:month 4 :day 1 :human 3.7 :critter 3.9}
     {:month 4 :day 2 :human 3.7 :critter 3.6}])

(drop-while #(< (:month %) 3) food-journal1)
```

```
({:month 3, :day 1, :human 4.2, :critter 3.3}
{:month 3, :day 2, :human 4.0, :critter 3.8}
{:month 4, :day 1, :human 3.7, :critter 3.9}
{:month 4, :day 2, :human 3.7, :critter 3.6})
```

by using *take-while* and *drop-while* together, you can get data for just february and march:

```
(def food-journal1
  [{:month 1 :day 1 :human 5.3 :critter 2.3}
   {:month 1 :day 2 :human 5.1 :critter 2.0}
   {:month 2 :day 1 :human 4.9 :critter 2.1}
   {:month 2 :day 2 :human 5.0 :critter 2.5}
   {:month 3 :day 1 :human 4.2 :critter 3.3}
   {:month 3 :day 2 :human 4.0 :critter 3.8}
   {:month 4 :day 1 :human 3.7 :critter 3.9}
   {:month 4 :day 2 :human 3.7 :critter 3.6}])

(take-while #(< (:month %) 4)
            (drop-while #(< (:month %) 2) food-journal1))
```

```
({:month 2, :day 1, :human 4.9, :critter 2.1}
{:month 2, :day 2, :human 5.0, :critter 2.5}
{:month 3, :day 1, :human 4.2, :critter 3.3}
{:month 3, :day 2, :human 4.0, :critter 3.8})
```

this example uses *drop-while* to get rid of the january entries, and then it uses *take-while* on the result to keep taking entries until it reaches the first april entry.

filter and some

use *filter* to return all elements of a sequence that test true for a predicate function. here are the journal entries where human consumption is less that five liters:

```
(def food-journal1
  [{:month 1 :day 1 :human 5.3 :critter 2.3}
   {:month 1 :day 2 :human 5.1 :critter 2.0}
   {:month 2 :day 1 :human 4.9 :critter 2.1}
   {:month 2 :day 2 :human 5.0 :critter 2.5}
   {:month 3 :day 1 :human 4.2 :critter 3.3}
   {:month 3 :day 2 :human 4.0 :critter 3.8}
   {:month 4 :day 1 :human 3.7 :critter 3.9}
   {:month 4 :day 2 :human 3.7 :critter 3.6}])

(filter #(< (:human %) 5) food-journal1)
```

```
({:month 2, :day 1, :human 4.9, :critter 2.1}
{:month 3, :day 1, :human 4.2, :critter 3.3}
{:month 3, :day 2, :human 4.0, :critter 3.8}
{:month 4, :day 1, :human 3.7, :critter 3.9}
{:month 4, :day 2, :human 3.7, :critter 3.6})
```

you might be wondering why we didn't just use *filter* in the *take-while* and *drop-while* examples earlier. indeed, *filter* would work for that too. here we're grabbing the january and february data, just like in the *take-while* example:

```
(def food-journal1
  [{:month 1 :day 1 :human 5.3 :critter 2.3}
   {:month 1 :day 2 :human 5.1 :critter 2.0}
   {:month 2 :day 1 :human 4.9 :critter 2.1}
   {:month 2 :day 2 :human 5.0 :critter 2.5}
   {:month 3 :day 1 :human 4.2 :critter 3.3}
   {:month 3 :day 2 :human 4.0 :critter 3.8}
   {:month 4 :day 1 :human 3.7 :critter 3.9}
   {:month 4 :day 2 :human 3.7 :critter 3.6}])

(filter #(< (:month %) 3) food-journal1)
```

```
({:month 1, :day 1, :human 5.3, :critter 2.3}
{:month 1, :day 2, :human 5.1, :critter 2.0}
{:month 2, :day 1, :human 4.9, :critter 2.1}
{:month 2, :day 2, :human 5.0, :critter 2.5})
```

this use is perfectly fine, but *filter* can end up processing all of your data, which isn't always necessary. because the food journal is already sorted by date, we know that *take-while* will return the data we want without having to examine any of the data we won't need. therefore, *take-while* can be more efficient.

often, you want to know whether a collection contains any values that test true for a predicate function. the *some* function does that, returning the first truthy value (any value that's not *false* or *nil*) returned by a predicate function:

```
(def food-journal1
  [{:month 1 :day 1 :human 5.3 :critter 2.3}
   {:month 1 :day 2 :human 5.1 :critter 2.0}
   {:month 2 :day 1 :human 4.9 :critter 2.1}
   {:month 2 :day 2 :human 5.0 :critter 2.5}
   {:month 3 :day 1 :human 4.2 :critter 3.3}
   {:month 3 :day 2 :human 4.0 :critter 3.8}
   {:month 4 :day 1 :human 3.7 :critter 3.9}
   {:month 4 :day 2 :human 3.7 :critter 3.6}])

(some #(> (:critter %) 5) food-journal1)
```

```
nil
```

```
(def food-journal1
  [{:month 1 :day 1 :human 5.3 :critter 2.3}
   {:month 1 :day 2 :human 5.1 :critter 2.0}
   {:month 2 :day 1 :human 4.9 :critter 2.1}
   {:month 2 :day 2 :human 5.0 :critter 2.5}
   {:month 3 :day 1 :human 4.2 :critter 3.3}
   {:month 3 :day 2 :human 4.0 :critter 3.8}
   {:month 4 :day 1 :human 3.7 :critter 3.9}
   {:month 4 :day 2 :human 3.7 :critter 3.6}])

(some #(> (:critter %) 3) food-journal1)
```

```
true
```

you don't have any food journal entries where you consumed more than five liters from critter sources, but you do have at least one where you consumed more that three liters. notice that the return value in the second example is *true* and not the actual entry that produced the true value. the reason is that the anonymous function (> (: $critteryoucouldreturntheentry$:

```
(def food-journal1
  [{:month 1 :day 1 :human 5.3 :critter 2.3}
   {:month 1 :day 2 :human 5.1 :critter 2.0}
   {:month 2 :day 1 :human 4.9 :critter 2.1}
   {:month 2 :day 2 :human 5.0 :critter 2.5}
   {:month 3 :day 1 :human 4.2 :critter 3.3}
   {:month 3 :day 2 :human 4.0 :critter 3.8}
   {:month 4 :day 1 :human 3.7 :critter 3.9}
   {:month 4 :day 2 :human 3.7 :critter 3.6}])

(some #(and (> (:critter %) 3) %) food-journal1)
```

```
{:month 3, :day 1, :human 4.2, :critter 3.3}
```

here, a slightly different anonymous function *uses* and to *first* check whether the condition (> (: $critterwhentheconditionisindeedtrue.$

sort and sort-by

you can *sort* elements in ascending order with *sort*:

```
(sort [3 1 2])
```

```
(1 2 3)
```

if your sorting needs are more complicated, you can use *sort-by*, which allows you to apply a function (sometimes called a *key function*) to the elements of sequence and use the values it returns to determine the sort order. in the following example, which is take from `https://clojuredocs.org/clojure.core/count`, *count* is the key function:

```
(sort-by count ["aaa" "c" "bb"])
```

```
("c" "bb" "aaa")
```

```
(sort ["aaa" "c" "bb"])
```

```
("aaa" "bb" "c")
```

if you were sorting using *sort*, the elements would be sorted in alphabetical order, returning *(”aaa” ”bb” ”c”).* instead, the result is *(”c” ”bb” ”aaa”)* because you're sorting by *count* and the count of *”c”* is 1, ”bb” is 2, and ”aaa” is 3.

concat

finally, *concat* simply appends the members of one sequence to the end of another:

```
(concat [1 2] [3 4])
```

```
(1 2 3 4)
```

lazy seqs

as you saw earlier, *map* first calls *seq* on the collection you pass to it. but that's not the whole story. many function, including *map* and *filter*, return a *lazy seq.* a lazy seq is a seq whose members aren't computed until you try to access them. computation until the moment it's needed makes your programs more efficient, and it has the surprising benefit of allowing you to construct infinite sequences.

demonstrating lazy seq efficiency

to see lazy seqs in action, pretend that you're part of modern-day task force whose purpose is to identify vampires. your intelligence agents tell you that there is only one active vampire in your city, and they've helpfully narrowed down the list of suspects to a million people. your boss gives you a list of one million social security numbers and shouts, "get it done, mcfishwich!".

thankfully, you are in possesion of a vampmatic 3000 computifier, the state-of-the-art device for vampire identification. because the source code for this vampire-hunting technology is proprietary. i've stubbed it out to simulate the time it would take to perform this task. here is a subset of a vampire database:

```
(def vampire-database
  {0 {:makes-blood-puns? false, :has-pulse? true  :name "mcfishwich"}
   1 {:makes-blood-puns? false, :has-pulse? true  :name "mcmackson"}
   2 {:makes-blood-puns? true,  :has-pulse? false :name "damon salvatore"}
   3 {:makes-blood-puns? true,  :has-pulse? true  :name "mickey mouse"}})

(defn vampire-related-details
  [social-security-number]
  (thread/sleep 1000)
  (get vampire-database social-security-number))

(defn vampire?
  [record]
  (and (:makes-blood-puns? record)
       (:not (:has-pulse? record))
       record))

(defn identify-vampire
  [social-security-number]
  (first (filter vampire?
                 (map vampire-related-details social-security-number))))
```

you have a function, *vampire-related-details*, which takes one second to look up an entry from the database. next, you have a function *vampire?*, which returns a record if it passes the vampire test; otherwise, it returns *false*. finally, *identify-vampire* maps the first record that indicates vampirism.

to show how much time it takes to run these functions, you can use the time operation. when you use *time*, your code behaves exactly as it would if you didn't use *time*, but with one exception: a report of the elapsed time is printed. here's an example:

```
(def vampire-database
  {0 {:makes-blood-puns? false, :has-pulse? true  :name "mcfishwich"}
   1 {:makes-blood-puns? false, :has-pulse? true  :name "mcmackson"}
   2 {:makes-blood-puns? true,  :has-pulse? false :name "damon salvatore"}
   3 {:makes-blood-puns? true,  :has-pulse? true  :name "mickey mouse"}})

(defn vampire-related-details
  [social-security-number]
  (thread/sleep 1000)
  (get vampire-database social-security-number))

(defn vampire?
  [record]
  (and (:makes-blood-puns? record)
       (:not (:has-pulse? record))
       record))

(defn identify-vampire
  [social-security-number]
  (first (filter vampire?
                 (map vampire-related-details social-security-number))))

(time (vampire-related-details 0))


"elapsed time: 1000.640251 msecs"
{:makes-blood-puns? false, :has-pulse? true, :name "mcfishwich"}
```

the first printed line reports the time taken by the given operation–in this case, 1000.640251 milliseconds. the second is the return value, which is your database record in this case. the return value is exactly the same as it would been if you hadn't used *time*.

a nonlazy implementation of *map* would first have to apply *vampire-related-details* to every member of *social-security-number* before passing the result to *filter*. because you have one million suspects, this would take one million seconds, or 12 days, and half your city would be dead by then! of course, if it turns out that the only vampire is the last suspect in the record, it will still take that much time with the lazy version, but at least there's a good chance that it won't.

because *map* is lazy, it doesn't actually apply *vampire-related-details* to social security numbers until you try to access the mapped element. in fact, *map* returns a value almost instantly:

```
(def vampire-database
  {0 {:makes-blood-puns? false, :has-pulse? true  :name "mcfishwich"}
   1 {:makes-blood-puns? false, :has-pulse? true  :name "mcmackson"}
   2 {:makes-blood-puns? true,  :has-pulse? false :name "damon salvatore"}
   3 {:makes-blood-puns? true,  :has-pulse? true  :name "mickey mouse"}})

(defn vampire-related-details
  [social-security-number]
  (thread/sleep 1000)
  (get vampire-database social-security-number))

(defn vampire?
  [record]
  (and (:makes-blood-puns? record)
       (:not (:has-pulse? record))
       record))

(defn identify-vampire
  [social-security-number]
  (first (filter vampire?
                 (map vampire-related-details social-security-number))))

(time (def mapped-details (map vampire-related-details (range 0 1000000))))


"elapsed time: 0.030287 msecs"
#'user/mapped-details
```

in this example, *range* returns a lazy sequence consisting of the integers from 0 to 999.999. then, *map* returns a lazy sequence that is associated with the name *mapped-details*. because *map* didn't actually apply *vampire-related-details* to any of the elements returned by *range*, the entire operation took barely any time–certainly less than 12 days.

you can think of a lazy seq as a consisting of two parts: a recipe for how to realize the elements of a sequence and the elements that have been realized so far. when you use *map*, the lazy seq it returns doesn't include any realized elements yet, but it does have the recipe for generating its elements. every time you try to access an unrealized element, the lazy seq will use its recipe to generate the requested element.

in the previous example, *mapped-details* is unrealized. once you try to access a member of *mapped-details*, it will use its recipe to generate the element you've requested, and you'll incur the *one-second-per-database-lookup* cost:

```
(def vampire-database
  {0 {:makes-blood-puns? false, :has-pulse? true  :name "mcfishwich"}
   1 {:makes-blood-puns? false, :has-pulse? true  :name "mcmackson"}
   2 {:makes-blood-puns? true,  :has-pulse? false :name "damon salvatore"}
   3 {:makes-blood-puns? true,  :has-pulse? true  :name "mickey mouse"}})

(defn vampire-related-details
  [social-security-number]
  (thread/sleep 1000)
  (get vampire-database social-security-number))

(defn vampire?
  [record]
  (and (:makes-blood-puns? record)
       (:not (:has-pulse? record))
       record))

(defn identify-vampire
  [social-security-number]
  (first (filter vampire?
                 (map vampire-related-details social-security-number))))

(def mapped-details (map vampire-related-details (range 0 1000000)))

(time (first mapped-details))


"elapsed time: 32010.491347 msecs"
{:makes-blood-puns? false, :has-pulse? true, :name "mcfishwich"}
```

this operation took about 32 seconds. that's much better than one million seconds, but this is still 31 seconds more than we would have expected. after all, you're only trying to access the very first element, so it should have taken only one second.

the reason it took 32 seconds is that clojure *chunks* its lazy sequences, which just means that whenever clojure has to realize an element, it preemptively realize some of the next element of *mapped-details*, but clojure went ahead and prepared the next 31 as well. clojure does this because it almost always result in better performance.

thankfully, lazy seq elements need to be realized only once. accessing the first element of *mapped-details* again takes almost no time

```
(time (first mapped-details))

"elapsed time: 0.028961 msecs"
=>{:makes-blood-puns? false, :has-pulse? true, :name "mcfishwich"}
```

with all this newfound knowledge, you can efficiently mine the vampire database to find the fanged culprit:

```
(time (identify-vampire (range 0 1000000)))

"elapsed time: 32019.912 msecs"
=>{:makes-blood-puns? true, :has-pulse? false, :name "damon salvatore"}
```

infinite sequences

one cool, useful capability that lazy seqs give you is the ability to construct infinite sequences. so far, you've only worked with lazy sequences generated from vectors or lists that terminated. however, clojure comes with a few functions to create infinite sequences. one easy way to create an infinite sequence is with *repeat*, which creates a sequence whose every members is the argument you pass:

```
(concat (take 8 (repeat "na")) ["batman!"])


("na" "na" "na" "na" "na" "na" "na" "na" "batman!")
```

in this case, you create an infinite sequence whose every element is the string *"na"*, then use that to construct a sequence that may or not provoke nostalgia.

you can also use *repeatedly*, which will call the provided function to generate each element in the sequence:

```
(take 3 (repeatedly (fn [] (rand-int 10))))
```

```
(2 0 3)
```

here, the lazy sequence returned by *repeatedly* generates every new element by calling the anonymous function $(fn[](rand-int10))$, which returns a random integer between 0 and 9. if you run this in your repl, your result will most likely be different from this one.

a lazy seq's recipe doesn't have to specify an endpoint. functions like *first* and *take* which realize the lazy seq, have no way of knowing what will come next in a seq, and if the seq keep providing elements, well, they'll just keep taking them. you can see this if you construct your own infinite sequence.

```
(defn even-numbers
  ([] (even-numbers 0))
  ([n] (cons n (lazy-seq (even-numbers (+ n 2))))))

(take 10 (even-numbers))
```

```
(0 2 4 6 8 10 12 14 16 18)
```

this example is a bit mind-bending because of its use of recursion. it helps to remember that *cons* returns a new list with an element appended to he given list:

```
(cons 0 '(2 4 6))
```

```
(0 2 4 6)
```

(incidentally, lisp programmers call it *consing*[2] when they use the *cons* function.)

in *even-numbers*, you're consing to a lazy list, which includes a recipe (a function) for the next element (as opposed to consing to a fully realized list).

and that covers lazy seq! now you know everything there is to know about sequence abstraction, and we can turn to the collection abstraction.

[2]In Lisp, "consing" refers to the fundamental operation of allocating a cons cell, which is a memory object capable of holding two values or pointers. The term comes from the cons function itself, used to "consecute" or link these cells together to build basic data structures like lists. Therefore, "consing" means creating new cons cells.

the collection abstraction

the collection abstraction is closely related to the sequence abstraction. all of clojure's core data structures–vectors, maps, lists, and sets–take part in both abstraction.

the sequence abstraction is about operating on members individually, whereas the collection abstraction is about the data structure as a whole. for example, the collection function *count*, *empty?*, and *every?* aren't about any individual element; they're about the whole:

```
(empty? [])
```

```
true
```

```
(empty? ["no! "])
```

```
false
```

practically speaking, you'll rarely consciously say, "okay, self! you're working with the collection as a whole now. think in terms of the collection abstraction!" neverthenless, it's useful to know these concepts that underline the functions and data structures you're using.

now we'll examine two common collection functions–*into* and *conj*–whose similarities ca be a bit confusing.

into

one of the most important collection function is *into.* as you now know, many seq functions return a seq rather than the original data structure. you'll probably want to convert the return value back into the original value, and *into* lets you do that:

```
(map identity {:sunlight-reaction "glitter"})
```

```
([:sunlight-reaction "glitter"])
```

```
(into {} (map identity {:sunlight-reaction "glitter"}))
```

```
{:sunlight-reaction "glitter"}
```

here, the *map* function returns a sequential data structure after being given a map data structure, and into converts the seq back into a map.

this will work with other data structure as well:

```
(map identity [:garlic :sesame-oil :fried-eggs])
```

```
(:garlic :sesame-oil :fried-eggs)
```

```
(into [] (map identity [:garlic :sesame-oil :fried-eggs]))
```

```
[:garlic :sesame-oil :fried-eggs]
```

here, the first line, *map* return a seq, and we use *into* in the second line to covert the result back to a vector.

in the following example, we start with a vector with two identical entries, *map* converts it to a list, and then we use *into* to stick the value into a set.

```
(map identity [:garlic-clove :garlic-clove])
```

```
(:garlic-clove :garlic-clove)
```

```
(into #{} (map identity [:garlic-clove :garlic-clove]))
```

```
#{:garlic-clove}
```

because sets only contain unique values, the set ends up with just one value.

the first argument of *into* doesn't have to be empty. here, the first example shows how you can use *into* to add elements to a map, and second shows how you can add elements to a vector.

```
(into {:favorite-emotion "gloomy"} [[:sunlight-reaction "glitter!"]])
```

```
{:favorite-emotion "gloomy", :sunlight-reaction "glitter!"}
```

```
(into ["cherry"] '("pine" "spruce"))
```

```
["cherry" "pine" "spruce"]
```

and, of course, both arguments can be the same type. in this next example, both arguments are maps, whereas all the previous examples had arguments of different types. it works as you'd expect, returning a new map with the elements of the second map added to the first:

```
(into {:favorite-animal "kitty"} {:least-favorite-smell "dog"
                                  :relationship-with-teenager "creepy"})
```

```
{:favorite-animal "kitty",
:least-favorite-smell "dog",
:relationship-with-teenager "creepy"}
```

if *into* asked to describe its strengths at a job interview, it would say. "i'm great at taking two collection and adding all the elements from the second to the first."

conj

conj also adds elements to a collection, but it does it in a slightly different way:

```
(conj [0] [1])
```

```
[0 [1]]
```

whoopsie! looks like it added the entire vector [1] to [0]. compare this with *into*:

```
(into [0] [1])
```

```
[0 1]
```

notice that the number 1 is passed as a scalar (singular, non-collection) value, whereas *into*'s second argument must be a collection.

you can supply as many elements to add with *conj* as you want, and you can also add to other collections like maps:

```
(conj [0] 1 2 3 4)
```

```
[0 1 2 3 4]

(conj {:time "midnight"} [:place "ye olde cemetarium"])


{:time "midnight", :place "ye olde cemetarium"}
```

conj and *into* are similar that you could even define *conj* in term of *into*:

```
(defn my-conj
  [target & additions]
  (into target additions))

(my-conj [0] 1 2 3)

;; => [0 1 2 3]
```

This kind of pattern isn't that uncommon. You'll often see two functions that do the same thing, except one takes a rest parameter (*conj*) and one take seqable data structure (*into*).

```
(defn my-conj
  [target & additions]  ;; see the '&'
  (into target additions))

(my-conj [0 1])

;; => [0 1]
```

In Clojure, the symbol is used in a function definition to indicate that the parameter following it can accepts a variable number of arguments. This known as a "variadic[3]" function.

In the function definition [*targetadditions*], the number of arguments is counted as follows:

1. *target*: This is a required argument. You must provide one value for *target* when callind the function.

2. *additions*: This allows for a variable number of additional arguments. You can provide zero or more arguments for *additions*.

[3]Is a function that accepts a variable number of arguments, meaning the number of parameters passed to it can change with each call. This contrasts with traditional functions, which are defined with a fixed number and type of parameters in their prototype.

Function Functions

Learning to take advantage of Clojure's ability to accept functions as arguments and return functions as value is really fun, even if it takes some getting used to.

Tow of Clojure functions, *apply* and *partial*, might seem especially weird because they both accept *and* return functions. Let's unweird them.

apply

apply expolodes a seqable data structure so it can be passed to a function that expects a rest parameter. For example, *max* takes any number of arguments and returns the greatest of all the arguments. Here's how you'd find the greatest number:

```
(max 0 1 2)

;; => 2
```

But what if you want to find the greatest element of a vector? You can't just pass the vector to *max*:

```
(max [0 1 2])

;; => [0 1 2]
```

This doesn't return the greatest element in the vector because *map* returns the greatest of all the arguments passed to it, and in this case you're only passing it a vector containing all the numbers you want to compare, rather than passing in the numberrs as separate arguments. *apply* is perfect for this situation.

```
(apply max [0 1 2])

;; => 2
```

By using *apply*, it's as if you called ($max012$). You'll often use *apply* like this, exploding the elements of a collection so that they get passed to a function as separate arguments.

Remember how we defined *conj* in terms of *into* earlier? Well, we can also define *into* in terms of *conj* by using *apply*:

```
(defn my-into
  [target additions]
  (apply conj target additions))

(my-into [0] [1 2 3])

;; => [0 1 2 3]
```

This call to *my-into* is equivalent to calling (*conj*[0]123).

```
(conj [0] 1 2 3)

;; => [0 1 2 3]
```

partial

partial takes a function and any number of arguments. It then returns a new function. When you call the returned function, it calls the original function with the original arguments you supplied it along with the new arguments.

Here's an example:

```
(def add10 (partial + 10))

(add10 3)

;; => 13
```

```
(def add10 (partial + 10))

(add10 5)

;; => 15
```

```
(def add-missing-elements
  (partial conj ["water" "earth" "air"]))

(add-missing-elements "unobtainium" "adamantium")

;; => ["water" "earth" "air" "unobtainium" "adamantium"]
```

So, when you call *add*10, it calls the original function and arguments (+10) and tacks on whichever arguments you call *add*10 with. To help clarify how *partial* works, here's how you might define it:

```
(defn my-partial
  [partialized-fn & args]
  (fn [& more-args]
    (apply partialized-fn (into args more-args))))

(def add20 (my-partial + 20))
(add20 3)

;; => 23
```

In this example, the value of *add20* is the anonymous function returned by *my-partial*. The anonymous function is defined like this:

```
(fn [$ more-args]
  (apply + (into [20] more-args)))
```

In general, you want to use partials when you find you're repeating the same combination of function and arguments in many different contexts. This toy example shows how you could use *partial* to specialize a logger, creating a *warn* function:

```
(defn lousy-logger
  [log-level message]
  (condp = log-level
    :warn (clojure.string/lower-case message)
    :emergency (clojure.string/upper-case message)))

(def warn (partial lousy-logger :warn))

(warn "Red light ahead")
;; => red light ahead
```

Calling *(warn "Red light")* here is identical to calling *(lousy-logger :warn "Red light ahead")*

complement

A Vampire Data Analysis Program for the FWPD

Summary

Exercises