Structured data

Hint

The following web pages are good references to Clojure builtins and data structures:

- ClojureDocs (http://clojuredocs.org)
- Clojure cheatsheet (http://clojure.org/cheatsheet)

Fork this

https://github.com/iloveponies/structured-data (https://github.com/iloveponies/structured-data)

Here (basic-tools.html#how-to-submit-answers-to-exercises) are the instructions if you need them. Be sure to fork the repository behind the link above.

Let there be names

We often want to give a piece of data name, either because the act of naming gives clarity to the code, or because we want to refer to the data many times. As we have seen, namespace global names are declared with `def` . A function or value that is needed only inside one function can be given a *local name* with `let` .

As an example, let's define a function for calculating the length of a triangle's hypotenuse, given the length of its two legs:

```
(defn hypotenuse [x y]
  (let [xx (* x x)
        yy (* y y)]
    (Math/sqrt (+ xx yy))))
```

Here we give the expressions `(* x x)` and `(* y y)` the local names `xx` and `yy` , respectively. They are visible only inside `hypotenuse` .

`let` introduces one or more names and a scope for them:

```
(let [name1 value1
      name2 value2
      ...]
  (expression1)
  (expression2)
  ...)
```

The names introduced by `let` are visible in all the expressions after them, under `let` . A name is not visible to code outside the body of the `let` it is defined in.

```
user=> (let [x 42]
         (+ x x))
;=> 84
user=> x
CompilerException java.lang.RuntimeException:
Unable to resolve symbol: x in this context, compiling:(NO_SOURCE_PAT
H:0)
```

Note the indentation in `let` : the names inside the brackets are all aligned together, and the expressions are indented with two spaces.

```
(let [x 42]
  (indented x))
```

Exercise 1

The following function does a thing:

```
(defn do-a-thing [x]
  (Math/pow (+ x x) (+ x x)))
```

Change the function `do-a-thing` so that it uses `let` to give a name to the common expression `(+ x x)` in its body.

The names declared in a `let` expression can refer to previous names in the same expression:

```
(let [a 10
      b (+ a 8)]
  (+ a b))
;=> 28
```

In the example above, `b` can refer to `a` because `a` is declared before it. On the other hand, `a` can not refer to b:

```
(let [a (+ b 42)
      b 8]
  (+ a b))
; CompilerException java.lang.RuntimeException: Unable to resolve sym
bol:
; b in this context, compiling:(NO_SOURCE_PATH:1)
```

Simple values

Now that we know how to give names to values, let's look at what kind of values Clojure supports.

Scalar values are the regular, singular simple values like `42`, `"foo"` or `true`. The following table describes some of them.

Type	Examples	Description
Numbers	`42`, `3/2`, `2.1`	Numbers include integers, fractions, and floats.
Strings	`"foo"`	Text values.
Characters	`\x`, `\y`, `\√`	A single characer is written with a preceding `\`.
Keywords	`:foo`, `:?`	Values often used as map keys.
Booleans	`true`, `false`	Boolean values.

Vectors

Collections are the other kind of values, in addition to scalars, that are crucial to programming. Clojure has support for a rich set of collection data structures. We'll go over the most important structures in this chapter.

A *vector* is a collection that can be indexed with integers, like an array in other languages. It can contain values of different types.

```
[1 2 3]                 ;=> [1 2 3]
[:foo 42 "bar" (+ 2 3)] ;=> [:foo 42 "bar" 5]
```

A vector is written with surrounding brackets, `[]`, and the elements are written inside, separated by whitespace and optionally commas (`,`).

Vectors are indexed with the `get` function:

```
(get ["a" "b" "c"] 1)  ;=> "b"
(get ["a" "b" "c"] 15) ;=> nil
(get ["x"] 0)          ;=> "x"
```

Trying to index a vector beyond its size does *not* throw an exception. The special value `nil` is returned, instead.

Exercise 2

Write the function `(spiff v)` that takes a vector and returns the sum of the first and third elements of the vector. What happens when you pass in a vector that is too short?

```
(spiff [1 2 3])       ;=> 4
(spiff [1 2 3 4 5 6]) ;=> 4
(spiff [1 2])         ;=> ?
(spiff [])            ;=> ?
```

Basic vector operations

Vectors are immutable: once you have a vector, *you can not change it*. You can, however, easily create new vectors based on a vector:

```
(conj [1 2 3] 4)          ;=> [1 2 3 4]
(assoc [1 2 3 4] 2 "foo") ;=> [1 2 "foo" 4]
```

`conj` adds a value to a collection. Its behaviour depends on the type of collection: with vectors, it adds the value to the end of the vector. To be exact, `conj` does *not* change the given vector. Instead, it returns a new vector, based on the given vector, with the new element appended to the end.

Exercise 3

Write the function `(cutify v)` that takes a vector as a parameter and adds `"<3"` to its end.

```
(cutify []) => ["<3"]
(cutify [1 2 3]) => [1 2 3 "<3"]
(cutify ["a" "b"]) => ["a" "b" "<3"]
```

Vectors: A Postmodern Deconstruction

Another way of extracting values from a vector is by *destructuring* it:

```
(let [[x y z] [1 2 3 4 5 6]]
  (str x y z))
;=> "123"
```

Here, instead of giving a name to the vector `[1 2 3 4 5 6]`, we indicate with the brackets in `[x y z]` that we want to destructure the vector instead. Inside the brackets, we give names to the first three elements of the vector. `x` will be given the value of the first element, `1`; `b` will be `2` and `c` will be `3`. The concatenation of these values that `str` returns is `"123"`.

Exercise 4

Rewrite our earlier function `spiff` by destructuring its parameter. Call this new function `spiff-destructuring`.

You can destructure function parameters directly. For an example, take the following function:

```
(defn sum-pairs [first-pair second-pair]
  [(+ (first  first-pair) (first  second-pair))
   (+ (second first-pair) (second second-pair))])
```

The function takes two vectors and sums the elements pairwise:

```
(sum-pairs [42 5]   [-42 -5])   ;=> [0 0]
(sum-pairs [64 256] [-51 -219]) ;=> [13 37]
```

`sum-pair` is not very pretty to look at. We can spiff it up by taking out the elements of its parameter vectors by destructuring them:

```
(defn sum-pairs [[x1 y1] [x2 y2]]
  [(+ x1 x2) (+ y1 y2)])
```

`sum-pairs` still takes two parameter vectors, but now it does not give names to its parameters. Instead, it gives names to their first two elements by destructuring the parameters. We could have also destructured the parameters with a `let`.

Thinking With Boxes

Let's define a simple representation for a two-dimensional point. It will simply be a pair (2-element vector) of two numbers.

```
(defn point [x y]
  [x y])
```

And a representation for a rectangle. This will simply be a pair of points, the first being the bottom left corner and the second being the top left corner.

```
(defn rectangle [bottom-left top-right]
  [bottom-left top-right])
```

When you have nested structures where you know their structure in advance, you can destructure multiple levels at a time.

```
(let [[[x1 y1] [x2 y2]] rectangle]
  ... do stuff with coordinates)
```

This should prove to be useful in the following exercises.

Exercise 5

Write the functions `(height rectangle)` and `(width rectangle)` that return the height and width of the given rectangle. Use destructuring.

```
(height (rectangle [1 1] [5 1])) => 0
(height (rectangle [1 1] [5 5])) => 4
(height (rectangle [0 0] [2 3])) => 3

(width (rectangle [1 1] [5 1]))  => 4
(width (rectangle [1 1] [1 1]))  => 0
(width (rectangle [3 1] [10 4])) => 7
```

Exercise 6

Write the function `(square? rectangle)` that returns `true` if `rectangle` is a square and otherwise `false`.

```
(square? (rectangle [1 1] [2 2])) ;=> true
(square? (rectangle [1 1] [2 3])) ;=> false
(square? (rectangle [1 1] [1 1])) ;=> true
(square? (rectangle [3 2] [1 0])) ;=> true
(square? (rectangle [3 2] [1 1])) ;=> false
```

Exercise 7

Write the function `(area rectangle)` that returns the area of the given rectangle.

```
(area (rectangle [1 1] [5 1]))  => 0
(area (rectangle [0 0] [1 1]))  => 1
(area (rectangle [0 0] [4 3]))  => 12
(area (rectangle [3 1] [10 4])) => 21
```

Exercise 8

Write the function `(contains-point? rectangle point)` that returns `true` if `rectangle` contains `point` and otherwise `false`.

Remember that you can give `<=` multiple parameters. `(<= x y z)` returns `true` if $x \leq y \leq z$ holds. Otherwise `false`.

Hint: `and` is useful.

use destructuring.

```
(contains-point? (rectangle [0 0] [2 2])
                 (point 1 1))            ;=> true
(contains-point? (rectangle [0 0] [2 2])
                 (point 2 1))            ;=> true
(contains-point? (rectangle [0 0] [2 2])
                 (point -3 1))           ;=> false
(contains-point? (rectangle [0 0] [2 2])
                 (point 1 3))            ;=> false
(contains-point? (rectangle [1 1] [2 2])
                 (point 1 1))            ;=> true
(contains-point? (rectangle [1 1] [1 1])
                 (point 1 1))            ;=> true
```

Exercise 9

Write the function `(contains-rectangle? outer inner)` that returns `true` if the rectangle `inner` is inside the rectangle `outer` and otherwise `false`.

Hint: use `contains-point?`

```
(contains-rectangle? (rectangle [0 0] [3 3])
                     (rectangle [1 1] [2 2])) ;=> true
(contains-rectangle? (rectangle [0 0] [2 2])
                     (rectangle [1 1] [3 3])) ;=> false
(contains-rectangle? (rectangle [0 0] [1 1])
                     (rectangle [0 0] [1 1])) ;=> true
(contains-rectangle? (rectangle [0 0] [1 1])
                     (rectangle [1 1] [2 2])) ;=> false
```

Maps

Where a vector associates integers to values, a *map* is not restricted to integer keys. You can use any kind of value as a key. A map is written with curly brackets, `{}`.

```
{"foo" 42, "bar" 666}
{"mehmeh" (+ 2 5)
 "rupatipor" "ropopo"}
```

A map is indexed with the `get` function:

```
(let [ages {"Juhana" 3
            "Ilmari" 42
            "King of All Cosmos" -6}]
  (get ages "King of All Cosmos"))
;=> -6
```

In idiomatic Clojure programs, the keys of a map are often *keywords*. Keywords are a convenient way of naming keys for values in associative collections such as maps. They are written with a preceding `:`.

```
(def book {:title "The City and the City"
           :authors [{:name "China Miéville", :birth-year 1972}]})

(get book :title) ;=> "The City and the City"
```

Keywords are even more convenient than this. They work as functions that access collections:

```
(:title book) ;=> "The City and the City"
```

When used as a function and given a collection, a keyword looks itself up in the collection and returns the value associated with it.

We are a legion

`count` can be used to find out the amount of elements in a collection.

```
(count [1 2 3]) ;=> 3
(count {:name "China Miéville", :birth-year 1972}) => 2
(count ":)") => 2
```

As we can see, `count` tells the amount of keys for a map and the amount of elements for a vector. It can also be used to find out the length of a string.

Let's define some authors and a couple of books with maps and vectors.

```
(def china {:name "China Miéville", :birth-year 1972})
(def octavia {:name "Octavia E. Butler"
              :birth-year 1947
              :death-year 2006})
(def friedman {:name "Daniel Friedman" :birth-year 1944})
(def felleisen {:name "Matthias Felleisen"})

(def cities {:title "The City and the City" :authors [china]})
(def wild-seed {:title "Wild Seed", :authors [octavia]})
(def embassytown {:title "Embassytown", :authors [china]})
(def little-schemer {:title "The Little Schemer"
                     :authors [friedman, felleisen]})
```

Exercise 10

Write the function `(title-length book)` that counts the length of the book's title.

```
(title-length cities)         ;=> 21
(title-length wild-seed)      ;=> 9
(title-length little-schemer) ;=> 18
```

Exercise 11

Write the function `(author-count book)` that returns the amount of authors that `book` has.

```
(author-count cities)         ;=> 1
(author-count wild-seed)      ;=> 1
(author-count little-schemer) ;=> 2
```

Exercise 12

Write the function `(multiple-authors? book)` that returns `true` if `book` has multiple authors, otherwise `false`.

```
(multiple-authors? cities)         ;=> false
(multiple-authors? wild-seed)      ;=> false
(multiple-authors? little-schemer) ;=> true
```

Adding Values to a Map

`(assoc a-map a-key a-value)` sets the value of `a-key` in `a-map` to be `a-value`.

```
(assoc {:a 1} :b 2) ;=> {:b 2, :a 1}
(assoc {:a 1} :a 2) ;=> {:a 2}
```

Let’s add some information to a book:

```
(assoc cities :awards ["Hugo", "World Fantasy Award",
                       "Arthur C. Clarke Award",
                       "British Science Fiction Award"])
;=> {:awards ["Hugo" "World Fantasy Award" "Arthur C. Clarke Award"
;              "British Science Fiction Award"]
;    :title "The City and the City"
;    :authors [{:birth-year 1972, :name "China Miéville"}]}
```

Vectors are an associative data structure, so `assoc` also works with them.

```
;index: 0 1 2                0   1   2
(assoc [3 2 1] 1 "~o~") ;=> [3 "~o~" 1]
```

Here the key that you give as a parameter is the index that you want to change.

Assoc does not actually change the original data structure, but instead returns an updated version of it.

```
(let [original [1 2 3 4]
      new      (assoc original 2 "foo")]
  original)
;=> [1 2 3 4]
```

Exercise 13

Use `assoc` and `conj` to write the function `(add-author book new-author)` that takes a book and an author as a parameter and adds `author` to `book`s authors.

Hint: use `let` to avoid pain

```
(add-author little-schemer {:name "Gerald J. Sussman"})
;=> {:title "The Little Schemer"
;    :authors [{:birth-year 1944, :name "Daniel Friedman"}
;              {:name "Matthias Felleisen"}
;              {:name "Gerald J. Sussman"}]}
(add-author {:authors [{:name "Juhana"}]} {:name "Jani"})
;=> {:authors [{:name "Juhana"} {:name "Jani"}]}
```

The keys and values of a map can be of any data type, and one map can contain any number of different data types as both keys and values.

`(contains? a-map a-key)` can be used to check if `a-map` has a value for `a-key`.

```
(contains? {"a" 1} "a")   ;=> true
(contains? {"a" 1} 1)     ;=> false
(contains? {"a" nil} "a") ;=> true
(contains? cities :title) ;=> true
(contains? cities :name)  ;=> false
```

Exercise 14

Write the function `(alive? author)` which takes an author map and returns `true` if the `author` is alive, otherwise `false`.

An author is alive if the author does not have a death year.

```
(alive? china)   ;=> true
(alive? octavia) ;=> false
```

Serial grave digging

We know how to extract information from a single book or author. However, we often want to extract information from a collection of items. As an example, given a collection of books, we want the names of all the authors:

```
(def books [cities, wild-seed, embassytown, little-schemer])

(all-author-names books)
;=> #{"China Miéville" "Octavia E. Butler"
;     "Daniel Friedman" "Matthias Felleisen"}
```

How should we implement `all-author-names`?

We'll give the implementation now, and introduce the new concepts used one by one. The implementation looks like this:

```
(defn author-names [book]
  (map :name (:authors book)))

(defn all-author-names [books]
    (set (apply concat (map author-names books))))
```

Now there's a lot of new stuff there, so we'll take a detour to learn it all before continuing with our book library.

Let's take a look at this `map` function.

Sequences

Before talking about `map`, we need to introduce a new concept: the *sequence*. Many of Clojure's functions that operate on vectors and other collections actually operate on sequences. The `(seq collection)` function returns a sequence constructed from a collection, such as a vector or a map.

Sequences have the following operations:

- `(first sequence)` returns the first element of the sequence.
- `(rest sequence)` returns the sequence without its first element.
- `(cons item sequence)` returns a new sequence where `item` is the first element and `sequence` is the rest.

```
(seq [1 2 3])          ;=> (1 2 3)
(seq {:a 42 :b "foo" :c ["ur" "dad"]})
                       ;=> ([:a 42] [:c ["ur" "dad"]] [:b "foo"])
(first (seq [1 2 3]))  ;=> 1
(rest (seq [1 2 3))    ;=> (2 3)
(cons 0 (seq [1 2 3])) ;=> (0 1 2 3)
```

Here you can see the printed form of sequences, the elements inside `(` and `)`. This has the consequence that copying `(1 2 3)` back to the REPL tries to call `1` as a function. The result is that you can not use the printed form of a sequence as a value like you could with vectors and maps.

Actually, the sequence functions call `seq` on their collection parameters themselves, so we can just write the above examples like this:

```
(first [1 2 3])  ;=> 1
(rest [1 2 3])   ;=> (2 3)
(cons 0 [1 2 3]) ;=> (0 1 2 3)
```

The map function

`(map function collection)` takes two parameters, a function and a sequenceable collection. It calls the function on each element of the sequence and returns a sequence of the return values.

```
(defn munge [x]
  (+ x 42))

(map munge [1 2 3 4])
;=> ((munge 1) (munge 2) (munge 3) (munge 4)) ; [note below]
;=> ( 43        44        45        46)
```

Note: You can’t paste the result line (or the middle one) to the REPL, as it is the printed form of a sequence.

Exercise 15

Write the function `(element-lengths collection)` that returns the lengths of every item in `collection`.

```
(element-lengths ["foo" "bar" "" "quux"])  ;=> (3 3 0 4)
(element-lengths ["x" [:a :b :c] {:y 42}]) ;=> (1 3 1)
```

Earlier, we briefly introduces the `fn` special form that can be used to create functions. This is useful when you want a function that is only visible in the definition of another function. Quite often you want to use `let` to give name to this helper function.

Let’s rewrite the example above in this style:

```
(defn mungefy [a-seq]
  (let [munge (fn [x] (+ x 42))]
    (map munge a-seq)))
```

Now the function `munge` is only visible inside the definition of `mungefy`. It should work like the previous one.

```
(mungefy [1 2 3 4]) ;=> (43 44 45 46)
```

Exercise 16

Use `map` to write the function `(second-elements collection)` that takes a vector of vectors and returns a sequence of the second elements.

Remember that you can use `get` to index a vector.

Use `fn` and `let` to create a helper function and use it with `map`.

```
(second-elements [[1 2] [2 3] [3 4]]) ;=> (2 3 4)
(second-elements [[1 2 3 4] [1] ["a" "s" "d" "f"]])
;=> (2 nil "s")
```

When you have a sequence of maps, the fact that `:keywords` are also functions can be helpful.

```
(:name {:name "MEEEE", :secret "Awesome"}) ;=>  "MEEEE"
```

You can therefore use a `:keyword` as the function parameter of `map`.

```
(let [people [{:name "Juhana", :age 3}
              {:name "Ilmari", :age 42}
              {:name "Jani", :age 72}
              {:name "King of All Cosmos" :age -6}]]
  (map :age people))
;=> (3 42 72 -6)
```

Exercise 17

Write the function `(titles books)` that takes a collection of books and returns their titles.

Using our earlier examples:

```
(def china {:name "China Miéville", :birth-year 1972})
(def octavia {:name "Octavia E. Butler"
              :birth-year 1947
              :death-year 2006})
(def friedman {:name "Daniel Friedman" :birth-year 1944})
(def felleisen {:name "Matthias Felleisen"})

(def cities {:title "The City and the City" :authors [china]})
(def wild-seed {:title "Wild Seed", :authors [octavia]})
(def embassytown {:title "Embassytown", :authors [china]})
(def little-schemer {:title "The Little Schemer"
                     :authors [friedman, felleisen]})

(def books [cities, wild-seed, embassytown, little-schemer])
```

`titles` should work like this:

```
(titles [cities]) ;=> ("The City and the City" )
(titles books)
;=> ("The City and the City" "Wild Seed"
;    "Embassytown" "The Little Schemer")
```

Okey, so now that `map` has been gone over, let's see the definition of `all-author-names` again.

```
(defn author-names [book]
  (map :name (:authors book)))

(defn all-author-names [books]
  (set (apply concat (map author-name books))))
```

`author-names` returns the names of the authors of a single book.

```
(author-names cities)         ;=> ("China Miéville")
(author-names little-schemer) ;=> ("Daniel Friedman" "Matthias Fellei
sen")
```

Since this is just a helper function used inside `all-author-names` we can move it inside by using `let` and `fn`.

```
(defn all-author-names [books]
  (let [author-names
         (fn [book] (map :name (:authors book)))]
    (set (apply concat (map author-names books)))))
```

The definition of `all-author-names` still has some mysterious words like `set`, `apply` and `concat` in it. Let's see what would happen without them.

```
(map author-names [cities]) ;=> (("China Miéville"))
(map author-names [cities, wild-seed]) ;=> (("China Miéville") ("Octa
via E. Butler"))
```

So first of all we would get every books authors inside a sequence. To fix this, we need to concatenate the sequences. To do this, there is `concat`.

```
(concat ["China Miéville"] ["Octavia E. Butler"]) ;=> ("China Miévill
e" "Octavia E. Butler")
```

This looks like what we want. However, if we simply try to use `(concat (map author-names books))`, we get the following problem:

```
(concat (map author-names [cities, wild-seed]))
;=> (concat (("China Miéville") ("Octavia E. Butler")))
;=> (("China Miéville") ("Octavia E. Butler"))
```

So we end up only giving `concat` one argument and it simply returns the argument. What we want is to give the elements of `(map author-names books)` to `concat` as arguments.

No worries, there is a way to do this. Let's check out `apply`.

Apply Now, Redux

`(apply function a-seq)` applies `function` to the arguments in `a-seq`. Here's an example:

```
(apply + [1 2 3])
;=> (+ 1 2 3)
;=> 6
```

And here's another with `concat`:

```
(apply concat [["China Miéville"] ["Octavia E. Butler"]])
;=> (concat ["China Miéville"] ["Octavia E. Butler"])
;=> ("China Miéville" "Octavia E. Butler")
```

More generally, `apply` works like this:

```
(apply function [arg1 arg2 arg3 ...]) => (function arg1 arg2 arg3 ...
)
```

Exercise 18

Write the function `(stars n)` that returns a string with `n` aterisks `\*`.

The function `(repeat n x)` returns a sequence with `n` `x`s:

```
(repeat 5 "*") ;=> ("*" "*" "*" "*" "*")
(repeat 3 "~o~") ;=> ("~o~" "~o~" "~o~")
```

Remember that you can use `str` to concatenate strings.

```
(stars 1) ;=> "*"
(stars 7) ;=> "*******"
(stars 3) ;=> "***"
```

Exercise 19

Write the function `(monotonic? a-seq)` that returns `true` if `a-seq` is monotonic and otherwise `false`.

A sequence is monotonic if is either inceasing or decreasing. In a decreasing sequence every element is at most as large as the previous one and in an increasing sequence every member is at least as large as the previous one.

Use `apply`.

Hint: `<=` might be useful

```
(monotonic? [1 2 3])     ;=> true
(monotonic? [0 1 10 11]) ;=> true
(monotonic? [3 2 0 -3])  ;=> true
(monotonic? [3 2 2])     ;=> true    Not strictly monotonic
(monotonic? [1 2 1 0])   ;=> false
```

So now we can put all authors into a single list. There's just one problem left. What is that? Well, let's see what happens if we put together everything seen so far.

```
(apply concat (map author-names books))
;=> ("China Miéville" "Octavia E. Butler"
;    "China Miéville" "Daniel Friedman"
;    "Matthias Felleisen")
```

We had two books by China Miéville, so his name is in the resulting sequence twice. But when we want to see the authors, we are usually not interested in duplicates. So lets turn the sequence into a data structure that supports this.

Set

Our last major data structure is the set. It is an unordered collection of items without duplicates.

```
(set ["^^" "^^" "^__*__^"]) ;=> #{"^__*__^" "^^"}
(set [1 2 3 1 1 1 3 3 2 1]) ;=> #{1 2 3}
```

The textual form of a set is `#{an-elem another-elem ...}` and you can convert another collection into a set with the function `set`.

Sets have three basic operations:

You can check whether a set contains an element with the function `contains?`:

```
(def games #{"Portal", "Planescape: Torment",
             "Machinarium", "Alpha Protocol"})

(contains? games "Portal") ;=> true
(contains? games "RAGE")   ;=> false
(contains? games 42)       ;=> false
```

`(conj set elem)` adds elem to `set` if it does not already have `elem`:

```
(conj #{:a :b :c} :EEEEE) ;=> #{:a :c :b :EEEEE}
```

Nothing happens if `elem` is already a member of `set`:

```
(conj #{:a :b :c} :a)     ;=> #{:a :c :b}
```

You can also add multiple elements by giving `conj` additional arguments:

```
(conj #{:a :b :c} :d :e)  ;=> #{:a :c :b :d :e}
```

Finally, `(disj set elem)` removes `elem` from `set` if it contains `elem`:

```
(disj #{:a :b :c} :c) ;=> #{:a :b}
(disj #{:a :b :c} :EEEEE) ;=> #{:a :c :b}
(disj #{:a :b :c} :c :a) ;=> #{:b}
```

Exercise 20

Write the function `(toggle a-set elem)` that removes `elem` from `a-set` if `a-set` contains `elem`, and adds it to the set otherwise.

```
(toggle #{:a :b :c} :d) ;=> #{:a :c :b :d}
(toggle #{:a :b :c} :a) ;=> #{:c :b}
```

If you want to know the size of a set, `count` also works with sets.

```
(count #{1 2 3}) ;=> 3
(count (set [1 2])) ;=> 2
```

Exercise 21

Write the function `(contains-duplicates? sequence)` that takes a sequence as a parameter and returns `true` if `sequence` contains some element multiple times. Otherwise it returns `false`.

```
(contains-duplicates? [1 1 2 3 -40]) ;=> true
(contains-duplicates? [1 2 3 -40]) ;=> false
(contains-duplicates? [1 2 3 "a" "a"]) ;=> true
```

Our books looked like this:

```
(def friedman {:name "Daniel Friedman" :birth-year 1944})
(def felleisen {:name "Matthias Felleisen"})

(def little-schemer {:title "The Little Schemer"
                     :authors [friedman, felleisen]})
```

Now we can understand the whole implementation of `all-author-names`. We use

- `fn` to introduce a helper function,
- keywords to index the books,
- map to get all authors from a single book
- `let` to give a name to our helper function,
- `map` to apply the helper function to all the given books, and
- construct a set with the `set` function to get rid of duplicates.

```
(defn all-author-names [books]
  (let [author-names
         (fn [book] (map :name (:authors book)))]
    (set (apply concat (map author-names books)))))
```

Calling our function returns the desired set:

```
(all-author-names books)
;=> #{"Matthias Felleisen" "China Miéville"
;     "Octavia E. Butler" "Daniel Friedman"}
```

Representing Books, Take Two

Now I would like to ask whether `little-schemer` has `felleisen` as an author or not. This turns out to be problematic. There is no function on vectors that can be used to query membership. So how about we change the representation of books? We now have a motivation to put authors into a set instead of a vector. This feels like a more natural fit, since a book never has a single author multiple times and our data doesn't give a natural order for the authors.

New representation:

```
(def little-schemer {:title "The Little Schemer"
                     :authors #{friedman, felleisen}})
```

Exercise 22

Write the function `(old-book->new-book book)` that takes a book with the previous representation (authors in a vector) and returns the same book in the new representation (authors in a set).

Use `assoc` to change the representation. Do not construct a new map using the map literal syntax.

```
(old-book->new-book {:title "The Little Schemer"
                     :authors [friedman, felleisen]})
;=> {:title "The Little Schemer" :authors #{friedman, felleisen}}
(old-book->new-book {:title "Wild Seed", :authors [octavia]})
;=> {:title "Wild Seed", :authors #{octavia}}
```

The reason to use `assoc` is that it allows us to keep any additional key-value pairs intact. Earlier we had an example where we added a list of awards to a book. By using `assoc`, these additional key-value pairs do not disappear anywhere during the transformation.

```
(old-book->new-book
  {:awards ["Hugo" "World Fantasy Award" "Arthur C. Clarke Award"
            "British Science Fiction Award"]
   :title "The City and the City"
   :authors [{:birth-year 1972, :name "China Miéville"}]})
;=> {:awards ["Hugo" "World Fantasy Award" "Arthur C. Clarke Awar
d"
;             "British Science Fiction Award"]
;    :title "The City and the City"
;    :authors #{{:birth-year 1972, :name "China Miéville"}}}
```

Here are all of the books changed to the new representation:

```
(def china {:name "China Miéville", :birth-year 1972})
(def octavia {:name "Octavia E. Butler"
              :birth-year 1947
              :death-year 2006})
(def friedman {:name "Daniel Friedman" :birth-year 1944})
(def felleisen {:name "Matthias Felleisen"})

(def cities {:title "The City and the City" :authors #{china}})
(def wild-seed {:title "Wild Seed", :authors #{octavia}})
(def embassytown {:title "Embassytown", :authors #{china}})
(def little-schemer {:title "The Little Schemer"
                     :authors #{friedman, felleisen}})

(def books [cities, wild-seed, embassytown, little-schemer])
```

Now that the authors are in a set, it is easy to find out whether a book has some author or not.

Exercise 23

Write the function `(has-author? book author)` that returns `true` if `author` is in the authors of `book` and otherwise `false`.

```
(has-author? cities china)             ;=> true
(has-author? cities felleisen)         ;=> false
(has-author? little-schemer felleisen) ;=> true
(has-author? little-schemer friedman)  ;=> true
(has-author? little-schemer octavia)   ;=> false
```

Does our previous definition for `all-author-names` still work? It does, but let's take another look at it.

```
(defn all-author-names [books]
  (let [author-names
         (fn [book] (map :name (:authors book)))]
    (set (apply concat (map author-names books)))))
```

Here we first turn each book into a sequence of names. Concatenate the sequences and finally turn this sequence into a set. Let's break this into two steps. First, let's define a function that returns all authors in a set. Then use this set to get the names.

For sets, there is a special function `(clojure.set/union set1 set2 ...)` that returns a new set that has all the elements of its parameters.

```
(clojure.set/union #{1 2} #{2 3} #{1 2 3 4} #{7 8}) ;=> #{1 2 3 4 7 8
}
(apply clojure.set/union [#{1 2} #{5} #{7 8}])      ;=> #{1 2 5 7 8}
```

That is, `union` works like `concat` but is specialized for sets. Let's put this into good use:

Exercise 24

Write the function `(authors books)` that returns the authors of every book in `books` as a set.

```
(authors [cities, wild-seed])              ;=> #{china, octavia}
(authors [cities, wild-seed, embassytown]) ;=> #{china, octavia}
(authors [little-schemer, cities])         ;=> #{china, friedman,
 felleisen}
```

Now that we have all of our authors, defining `all-author-names` should be simple.

Exercise 25

Write the function `(all-author-names books)` that works like the previous one and uses `authors`.

```
(all-author-names books)
;=> #{"Matthias Felleisen" "China Miéville"
;     "Octavia E. Butler" "Daniel Friedman"}
(all-author-names [cities, wild-seed])
;=> #{"China Miéville" "Octavia E. Butler"}
(all-author-names []) ;=> #{}
```

String Representation for Books

Now that we have defined these books, I would like to have a readable string representation for them. Let's start by defining a representation for a single author.

Exercise 26

Write the function `(author->string author)` that returns a string representation of `author` as follows:

You can assume that every author with a `:death-year` also has a `:birth-year`.

```
(author->string felleisen) ;=> "Matthias Felleisen"
(author->string friedman)  ;=> "Daniel Friedman (1944 - )"
(author->string octavia)   ;=> "Octavia E. Butler (1947 - 2006)"
```

Hint: you probably want to split this string into two parts: name and years. Use `let` to form these and use `str` to create the final string.

Now we have a string representation for a single author. Some of our books had multiple authors, so we need to figure out a way to give a string representation for multiple authors. To do this, we need a handy helper function.

Sometimes you want to add something in between the elements of a sequence. For that, there is `(interpose separator a-seq)`, which returns a new sequence that has `separator` between each element of `a-seq`.

```
(interpose ":" [1 2 3])        ;=> (1 ":" 2 ":" 3)
(interpose " and " ["a", "b"]) ;=> ("a" " and " "b")
(interpose ", " [])            ;=> ()

(apply str (interpose " and " ["a", "b"])) ;=> "a and b"
```

With this, it shouldn't be too hard to get a nice representation for a sequence of authors.

Exercise 27

Write the function `(authors->string authors)` which takes a sequence of authors as a parameter and returns a string representation of `authors` in the following manner:

```
(authors->string (:authors little-schemer))
;=> "Daniel Friedman (1944 - ), Matthias Felleisen"
(authors->string #{octavia})          ;=> "Octavia E. Butler (194
7 - 2006)"
(authors->string #{})                 ;=> ""
(authors->string #{octavia, friedman})
;=> "Octavia E. Butler (1947 - 2006), Daniel Friedman (1944 - )"
;   order doesn't matter
```

Since the authors are in a set, which doesn't have a predefined order, the resulting string can have the authors in any order.

Now that we can handle the case of multiple authors, we can move on to the string representation of a single book.

Exercise 28

Write the function `(book->string book)` takes a single book as a parameter and returns a string representation of `book` as follows:

```
(book->string wild-seed) ;=> "Wild Seed, written by Octavia E. Bu
tler"
(book->string little-schemer)
;=> "The Little Schemer, written by Daniel Friedman (1944 - ), Ma
tthias Felleisen"
;                                    ^-- order doesn't matter
```

Again, the order of authors in the string doesn't matter.

And finally, we can define a string representation for a sequence of books.

Exercise 29

Write the function `(books->string books)` that takes a sequence of books as a parameter and returns a string representation of `books` like this:

```
(books->string []) ;=> "No books."
(books->string [cities])
;=> "1 book. The City and the City, written by China Miéville (19
72 - )."
(books->string [little-schemer, cities, wild-seed])
;=> "3 books. The Little Schemer, written by Daniel Friedman (194
4 - ), Matthias Felleisen. The City and the City, written by Chin
a Miéville (1972 - ). Wild Seed, written by Octavia E. Butler (19
47 - 2006)."
```

Filtering sequences

Another common function besides `map` is `filter`. It is used to select some elements of a sequence and disregard the rest:

```
(filter pos? [-4 6 -2 7 -8 3])
;=>        (      6        7          3  )
; value    -4    6    -2   7     -8   3
; pos?   false true false true false true

(filter (fn [x] (> (count x) 2)) ["ff" "f" "ffffff" "fff"])
;=> ("ffffff" "fff")
```

`(filter predicate collection)` takes two parameters, a function and a sequencable collection. It calls `predicate` (the function) on each element of `collection` and returns a sequence of elements of `collection` for which `predicate` returned a truthy value. In the above example the values `(6 7 3)` were selected because for them `pos?` returned `true`; for the others it returned `false`, a falsey value, and they were filtered out.

Exercise 30

Write the function `(books-by-author author books)`.

Hint: `has-author?`

```
(books-by-author china books)   ;=> (cities embassytown)
(books-by-author octavia books) ;=> (wild-seed)
```

```
(def authors #{china, felleisen, octavia, friedman})
```

Exercise 31

Write the function `(author-by-name name authors)` that takes a string `name` and a sequence of authors and returns an author with the given name if one is found. If one is not found, then `nil` should be returned.

Hint: remember `first`

```
(author-by-name "Octavia E. Butler" authors)                ;=> octavia
(author-by-name "Octavia E. Butler" #{felleisen, friedman}) ;=> nil
(author-by-name "China Miéville" authors)                   ;=> china
(author-by-name "Goerge R. R. Martin" authors)              ;=> nil
```

Exercise 32

Write the function `(living-authors authors)` that takes a sequence of authors and returns those that are alive. Remember `alive?`.

```
(living-authors authors)             ;=> (china, felleisen, friedman)
(living-authors #{octavia})          ;=> ()
(living-authors #{china, felleisen}) ;=> (china, felleisen)
```

The order in the results doesn't matter.

Here's another book. This one has both living and dead authors, which is a useful test case for the following exercises.

```
(def jrrtolkien {:name "J. R. R. Tolkien" :birth-year 1892 :death-year 1973})
(def christopher {:name "Christopher Tolkien" :birth-year 1924})
(def kay {:name "Guy Gavriel Kay" :birth-year 1954})

(def silmarillion {:title "Silmarillion"
                   :authors #{jrrtolkien, christopher, kay}})
```

And here's another with multiple dead authors:

```
(def dick {:name "Philip K. Dick", :birth-year 1928, :death-year 1982})
(def zelazny {:name "Roger Zelazny", :birth-year 1937, :death-year 1995})

(def deus-irae {:title "Deus Irae", :authors #{dick, zelazny}})
```

If you want to know whether a collection is empty or not, you can use `(empty? coll)` to do that.

```
(empty? [])  ;=> true
(empty? #{}) ;=> true
(empty? [1]) ;=> false
```

Exercise 33

Write the function `(has-a-living-author? book)` that returns `true` if `book` has a living author, and otherwise `false` .

```
(has-a-living-author? wild-seed)      ;=> false
(has-a-living-author? silmarillion)   ;=> true
(has-a-living-author? little-schemer) ;=> true
(has-a-living-author? cities)         ;=> true
(has-a-living-author? deus-irae)      ;=> false
```

Exercise 34

Write the function `(books-by-living-authors books)` that takes a sequence of books as a parameter and returns those that have a living author.

```
(books-by-living-authors books) ;=> (little-schemer cities embass
ytown)
(books-by-living-authors (concat books [deus-irae, silmarillion])
)
;=> (little-schemer cities embassytown silmarillion)
```

Keeping your vectors

`map` and `filter` always return sequences, regardless of the collection type given as a parameter. Sometimes, however, you want the result to be a vector. For an example, you may want to index the vector afterwards. In this situation, you can use `mapv` and `filterv` , which are variants of `map` and `filter` that always return vectors.

```
(mapv ... [...])    ;=> [...]
(filterv pos? [-4 6 -2 7 -8 3])  ;=> [6 7 3]
(filterv pos? #{-4 6 -2 7 -8 3}) ;=> [3 6 7]
(mapv ... #{...})   ;=> [...]
```

Done!

Phew, that was quite a lot of stuff.

Clojure with Style → (style.html)

 questions? (mailto:juhana.laurinharju@cs.helsinki.fi)