“Getting Clojure” by Russ Olsen

Personal Summary

Patrick Bucher

2023-07-20

Contents

This is a rough summary of Getting Clojure by Russ Olsen. Some examples have been taken from the original, some have been modified, and some have been made up. Instructions for Setup have been added for Arch Linux and Emacs.

1 Hello, Clojure

Install Clojure and Leiningen on Arch Linux:

```
# pacman -S clojure leiningen
```

Start a REPL:

```
$ lein repl
```

Write a "Hello World" program on the REPL:

```
> (println "Hello, World!")
Hello, World!
nil
```

Install Cider and Clojure Mode for Emacs:

```
M-x package-install RET clojure-mode RET
M-x package-install RET cider RET
```

Write the same program (with comments) to `hello.clj` (using Emacs):

```
;; Hello World program in Clojure.
(println "Hello, World!") ; Say hi.
```

Run the REPL in Emacs:

```
M-x cider-jack-in
```

Use `C-c C-k` to evaluate the entire buffer, or just the expression in front of the curser using `C-c C-e`.

Comments start with a semicolon and end with the line. Comments that take up a whole line start with two semicolons by convention.

Run the program:

```
$ clojure -M hello.clj
Hello, World!
```

Or shorter:

```
$ clj -M hello.clj
Hello, World!
```

Use basic arithmetic functions:

```
> (+ 3 2)
5
> (- 100 3 7)
90
> (+ (* 3 6) (/ 12 4))
21
> (/ 8 3) ; produces a ratio
8/3
> (quot 8 3) ; truncates
2
```

Bind a *symbol* to a value:

```
> (def result (* 13 12))
> (println result)
156
```

Concatenate multiple values as a string:

```
> (str 1 "to" 2)
"1to2"
```

Write and call the “Hello World” program as a function:

```
> (defn hello-world [] (println "Hello, World!"))
> (hello-world)
Hello, World!
```

Use a single function parameter:

```
(defn greet [whom]
  (println "Hello," whom))
```

```
> (greet "John")
Hello, John
```

Use multiple function parameters:

```
(defn average [a b]
  (/ (+ a b) 2))
```

```
> (average 10 4)
7
```

Use multiple expressions in the function body:

```
(defn average [a b]
  (def a-plus-b (+ a b))
  (def half-the-sum (/ a-plus-b 2))
  half-the-sum)
```

```
> (average 24 6)
15
```

The last expression of the function body (here: `half-the-sum`) is returned.

Create and run a proper application using Leiningen:

```
$ lein new app hello-world
$ cd hello-world
$ lein run
Hello, World!
```

Extend the example (`src/hello_world/core.clj`) as follows:

```
(ns hello-world.core
  (:gen-class))

(defn greet [app whom]
  (println app "greets" whom))

(defn -main
  [& args]
  (greet "Hello World" "the user"))
```

And run it again:

```
$ lein run
Hello World greets the user
```

2 Vectors and Lists

Create a vector of numbers:

```
> [1 2 3 4]
[1 2 3 4]
```

A vector can hold values of different types:

```
> [1 "two" 3.0 "four"]
[1 "two" 3.0 "four"]
```

Vectors can be nested:

```
> [1 ["foo" "bar"] 2 ["qux" "baz"]]
[1 ["foo" "bar"] 2 ["qux" "baz"]]
```

Vectors can also be created using the `vector` function:

```
> (vector 1 2 3 4)
[1 2 3 4]
> (vector (vector "one" "two") "three" (vector "four" "five"))
[["one" "two"] "three" ["four" "five"]]
```

`count` returns the number of elements in a vector:

```
> (def numbers [1 3 9 27])
> (count numbers)
4
```

`first` returns the first element of a vector:

```
> (first [1 2 3 4])
1
> (first [])
nil
```

`rest` returns all but the first elements of a vector as a *sequence*:

```
> (rest [1 2 3 4])
> (rest [1])
()
> (rest [])
()
```

`nth` returns the element at position `n` (zero-based index):

```
> (nth [1 2 3 4] 2)
3
```

The nth element can also be accessed using the vector's name and an index:

```
> (def numbers [1 2 3 4 5])
> (numbers 3)
4
```

`conj` adds an element *to the end* of a vector:

```
> (def cities ["London", "New York", "Berlin"])
> (conj cities "Moscow")
["London" "New York" "Berlin" "Moscow"]
```

`cons` adds an element *to the front* of a vector:

```
> (def countries ["USA", "Germany", "Turkey"])
> (cons "Russia" countries)
("Russia" "USA" "Germany" "Turkey")
```

Actually, a new vector or sequence, respectively, is created, holding the additional element.

A list can be created as follows:

```
> '("New York" "London" "Berlin")
("New York" "London" "Berlin")
```

Or using the `list` function:

```
> (list "New York" "London" "Berlin")
("New York" "London" "Berlin")
```

The functions `count`, `first`, `rest`, and `nth` can be applied to lists, too:

```
> (def countries '("USA" "Russia" "Germany" "France"))
> (count countries)
4
> (first countries)
"USA"
> (rest countries)
("Russia" "Germany" "France")
> (nth countries 3)
"France"
```

Unlike vectors, a list can *not* be used like a function:

```
> (countries 3)
Execution error (ClassCastException) at user/eval2092 (REPL:1).
clojure.lang.PersistentList cannot be cast to clojure.lang.IFn
```

Vectors are implemented as arrays, lists are implemented as linked lists. This has some implications:

- Appending to the front is fast for lists and slow for vectors.
- Appending to the end is fast for vectors and slow for lists.
- Accessing vector elements happens in constant time.
- Accessing list element n happens in n steps.

The `conj` function therefore adds elements to the front of lists and to the end of vectors:

```
> (conj [1 2 3] "what")
[1 2 3 "what"]
> (conj '(1 2 3) "what")
("what" 1 2 3)
```

3 Maps, Keywords, and Sets

Maps are created using pairs within curly braces:

```
> {"title" "War and Peace" "author" "Lev Tolstoy" "year" 1869}
{"title" "War and Peace", "author" "Lev Tolstoy", "year" 1869}
```

Commas between key-value pairs can be used for better readability, but are optional:

```
> {"title" "War and Peace", "author" "Lev Tolstoy", "year" 1869}
{"title" "War and Peace", "author" "Lev Tolstoy", "year" 1869}
```

Maps can also be created using the `hash-map` function:

```
> (hash-map "title" "War and Peace" "author" "Lev Tolstoy" "year" 1869)
{"author" "Lev Tolstoy", "title" "War and Peace", "year" 1869}
```

The left part of the pair is the *key*, the right part the *value* of the entry.

`get` looks up the value of an entry by its key:

```
> (def book {"title" "War and Peace" "author" "Lev Tolstoy" "year" 1869})
> (get book "author")
"Lev Tolstoy"
```

Like vectors, elements can be accessed without an explicit function call:

```
> (book "title")
"War and Peace"
> (book "year")
1869
> (book "publisher")
nil
```

Idiomatically, *keywords* starting with a colon are used as map keys:

```
> (def book {:title "War and Peace" :author "Lev Tolstoy" :year 1869})
> book
{:title "War and Peace", :author "Lev Tolstoy", :year 1869}
> (book :title)
"War and Peace"
```

Keywords can also be used for map lookups:

```
> (:title book)
"War and Peace"
```

`assoc` returns a map with an element either overwritten or added:

```
> (def book {:title "War and Peace" :author "Lev Tolstoy" :year 1869})
{:title "War and Peace", :author "Lev Tolstoy", :year 1869}
> (assoc book :pages 2000)
{:title "War and Peace", :author "Lev Tolstoy", :year 1869, :pages 2000}
> (assoc book :pages 1987)
{:title "War and Peace", :author "Lev Tolstoy", :year 1869, :pages 1987}
```

Using `assoc`, it's possible to add/modify multiple key-value pairs at once:

```
> (def employee {:name "Dilbert"})
> (assoc employee :job "Engineer" :salary 120000)
{:name "Dilbert", :job "Engineer", :salary 120000}
```

`dissoc` removes a map's entry by its key:

```
> (def employee {:name "Dilbert" :note "smelly"})
> (dissoc employee :note)
{:name "Dilbert"}
```

Like `assoc`, multiple keys can be used at once with `dissoc`:

```
> (def employee {:name "Dilbert" :note "smelly" :terminate "Jan 2023"})
> (dissoc employee :note :terminate)
{:name "Dilbert"}
```

Keys not found in the map will be ignored silently:

```
> (dissoc employee :sex-appeal :girlfriend :hobbies)
{:name "Dilbert", :note "smelly", :terminate "Jan 2023"}
```

`keys` returns the map's keys (in unspecified order):

```
> (def book {:title "War and Peace" :author "Lev Tolstoy" :year 1869})
> (keys book)
(:title :author :year)
```

Use a `sorted-map` for specified key ordering:

```
> (def book (sorted-map :title "War and Peace" :author "Lev Tolstoy" :year 1869))
> (keys book)
(:author :title :year)
```

`keys` returns the map's values in arbitrary order, matching the key's order:

```
> (vals book)
("War and Peace" "Lev Tolstoy" 1869)
> (keys book)
(:title :author :year)
```

A *set* can be created as follows (commas being optional):

```
> #{"Dilbert", "Alice", "Wally"}
#{"Alice" "Wally" "Dilbert"}
```

An element must not occur more than once:

```
> #{"Dilbert", "Alice", "Wally", "Dilbert"}
Syntax error reading source at (REPL:1:42).
Duplicate key: Dilbert
```

`contains?` checks if an element is contained in a set:

```
> (def employees #{"Dilbert", "Alice", "Wally", "Ashok"})
> (contains? employees "Dilbert")
true
> (contains? employees "Pointy Haired Boss")
false
```

This lookup can be done without calling a function, returning the element if it is contained, or `nil` if the element is missing:

```
> (employees "Dilbert")
"Dilbert"
> (employees "Ratbert")
nil
```

When working with keywords, the order can be switched:

```
> (def genres #{:scifi :action :drama :love})
> (genres :scifi)
:scifi
> (:scifi genres)
:scifi
```

Functions like `count`, `first`, and `rest` handle map entries as two-element vectors:

```
> (def employee {:name "Dilbert" :age 42 :job "Engineer"})
> (count employee)
3
> (first employee)
[:name "Dilbert"]
> (rest employee)
([:age 42] [:job "Engineer"])
```

A set can be extendes using `conj`:

```
> (conj genres :western)
#{:western :scifi :drama :action :love}
```

An element won't be added a second time (*without* causing an error):

```
> (conj genres :western)
#{:western :scifi :drama :action :love}
> (conj genres :western)
#{:western :scifi :drama :action :love}
```

`disj` returns a set without the specified element:

```
> (disj genres :western)
#{:scifi :drama :action :love}
```

No error occurs if the element is missing:

```
> (disj genres :comedy)
#{:scifi :drama :action :love}
```

Be aware that `nil` is a valid set entry and map key:

```
> (contains? #{:foo :bar nil} nil)
true
```

4 Logic

Conditional code can be executed using `if`:

```
(if (= guess secret-number)
  (println "You guessed the secret number.")
  (println "Sorry, wrong number guessed..."))
```

If the boolean expression (first argument) holds true, the second argument is evaluated; otherwise, the (optional) third argument is evaluated.

Being an expression, `if` returns a value:

```
(defn yield-rate [balance]
  (if (⩾ balance 0) 0.125 12.5))
```

```
> (yield-rate 300)
0.125
> (yield-rate -150)
12.5
```

`nil` will be returned if the condition evaluates to `false` and if there's no `else` branch.

Comparison operators like `=`, `not=`, and `⩾` are actually functions, which can take two or more arguments:

```
> (= 2 2 2 2 3)
false
> (= 2 2 2 2 2 2)
true

> (not= 1 1 1 1)
false
> (not= 1 1 2 1)
true

> (⩾ 9 6 4 4 1)
true
> (⩾ 9 6 4 5 1)
false
```

Predicate functions return whether or not an expression is of some specific type:

```
> (number? 1987)
true
> (string? "Dilbert")
true
> (keyword? :title)
true
> (map? {:born 1987})
true
> (vector? [1 2 3])
true
```

Multiple conditions can be combined using `and`, `or`, and `not`:

```
> (or (and (> 5 3) (< 1 6)) (not (= 3 1)))
true
```

Both `or` and `and` are *short-circuit* operations (nothing is printed here):

```
> (or (= 1 1) (println "strange"))
true
> (and (not= 1 1) (println "strange"))
false
```

Every value besides `false` and `nil` is treated as *truthy* (i.e. will be evaluated to `true`), even empty collections and the number 0:

```
> (if [] (println "[] is truthy"))
[] is truthy
nil

> (if 0 (println "0 is truthy"))
0 is truthy
nil
```

Multiple expressions can be grouped together using `do`:

```
(if (= guess secret-number)
  (do
    (println "You guessed the secret number.")
    (println "A winner is you.")
    {:points 100})
  (do
    (println "You guessed the wrong number.")
    (println "Shame on you.")
    {:points 0}))
```

The `do` expression evaluates to its last argument.

If no `else` branch is required, `when` can be used instead of `if`, which allows for multiple expressions without using `do`:

```
(when (= guess secret-number)
  (println "You guessed the secret number.")
  (println "A winner is you.")
  {:points 100})
```

If the condition doesn't hold `true`, `nil` is returned (like `if`).

Instead of nesting multiple `if`s, `cond` can be used for handling multiple conditions:

```
(defn check [guess secret-number]
  (cond
    (= guess secret-number) (println "correct")
    (< guess secret-number) (println "too low")
    (> guess secret-number) (println "too high")))

> (check 3 3)
correct
nil
```

```
> (check 4 3)
too high
nil

> (check 3 4)
too low
nil
```

Idiomatically, a catch-all `:else` clause is added to make sure that every condition is handled:

```
(defn check [guess secret-number]
  (cond
    (= guess secret-number) (println "correct")
    (< guess secret-number) (println "too low")
    (> guess secret-number) (println "too high")
    :else (println "You broke the universe")))
```

Since `:else` is truthy, its branch will be evaluated unless any other branch was evaluated before. (Any truthy value could be used instead of `:else`.)

For multiple equality comparisons against *constants*, `case` can be used instead of `cond`:

```
(defn color-hex-code [color]
  (case color
    :red "#ff0000"
    :green "#00ff00"
    :blue "#0000ff"
    "unknown"))

> (color-hex-code :red)
"#ff0000"
> (color-hex-code :black)
"unknown"
```

Exceptions can be handled using `try`/`catch`:

```
(defn safe-divide [dividend divisor]
  (try
    (/ dividend divisor)
    (catch ArithmeticException e
      (println "One does not simply divide by zero."))))
```

```
> (safe-divide 9 3)
3

> (safe-divide 9 0)
One does not simply divide by zero.
nil.
```

Exceptions can be thrown using `throw` and `ex-info`:

```
(defn publish [book]
  (when (< (:pages book) 50)
    (throw
      (ex-info "A book needs fifty pages or more!" book))))

> (publish {:title "Hello" :pages 30})
Execution error (ExceptionInfo) at user/publish (REPL:4).
A book needs fifty pages or more!
```

`ex-info` expects a string message and a map argument, and can be caught as `clojure.lang.ExceptionInfo`.

5 More Capable Functions

Multi-arity functions accept different sets of parameters:

```
(defn greet
  ([to-whom] (println "Hello" to-whom))
  ([message to-whom] (println message to-whom)))

> (greet "John")
Hello John
> (greet "Hi" "John")
Hi John
```

In order to reduce the amount of duplicated code, it's common that lower arity functions call the function with the highest arity by filling in the missing parameters with default values:

```
(defn greet
  ([to-whom] (greet "Hello" to-whom))
  ([message to-whom] (println message to-whom)))
```

Variadic functions accept a variable number of arguments:

```
(defn output-all [& args]
  (println "args" args))

> (output-all "one")
args (one)
> (output-all "one" 2 "three" 4.0)
args (one 2 three 4.0)
```

The arguments left of the ampersand are regular arguments:

```
(defn output-all [x y & args]
  (println "x" x "y" y "args" args))

> (output-all "one" 2 "three" 4.0)
x one y 2 args (three 4.0)
```

Multi-arity and variadic functions are good at dealing with a *different number* of arguments. *Multimethods* are useful to deal with *different characteristics* of arguments. They consist of:

1. A dispatch function (`defn`) that assigns a keyword to an argument.
2. A multimethod (`defmulti`) that groups the implementations and refers to the dispatch function.
3. Multiple methods (`defmethod`), of which each handles one type of argument.

Consider employees being stored in different formats:

```
;; implicit fields: first name, job
(def dogbert ["Dogbert" "Head of Abuse"])
(def ashok ["Ashok" "Technical Intern"])

;; relevant fields: name, position
(def alice {:name "Alice" :position "Engineer" })
(def wally {:name "Wally" :position "Engineer" })

;; relevant fields: first-name, job
(def catbert {:first-name "Catbert" :job "HR Manager"})
(def dilbert {:first-name "Dilbert" :job "Engineer" :department "IT"})
```

The dispatch function figures out which format such an entry has:

```
(defn dispatch-employee-format [employee]
  (cond
    (vector? employee) :vector-employee
    (and (contains? employee :name)
         (contains? employee :position)) :min-employee
    (and (contains? employee :first-name)
         (contains? employee :job)) :max-employee))
```

The multimethod defines a method name and connects it to the dispatcher:

```
(defmulti normalize-employee dispatch-employee-format)
```

The implementations all have the same name, but handle a different keyword, as mapped by the dispatcher function:

```
(defmethod normalize-employee :vector-employee [employee]
  {:first-name (nth employee 0) :role (nth employee 1)})

(defmethod normalize-employee :min-employee [employee]
  {:first-name (:name employee) :role (:position employee)})

(defmethod normalize-employee :max-employee [employee]
  {:first-name (:first-name employee) :role (:job employee)})
```

The different kind of employee data structures are converted to a common format:

```
> (normalize-employee dogbert)
{:first-name "Dogbert", :role "Head of Abuse"}
> (normalize-employee alice)
{:first-name "Alice", :role "Engineer"}
> (normalize-employee dilbert)
{:first-name "Dilbert", :role "Engineer"}
```

If the dispatch method cannot match the argument submitted, an exception will be thrown:

```
> (normalize-employee {:first-name "Topper" :position "Head of Annoyance"})
Execution error (IllegalArgumentException) at user/eval2108 (REPL:1).
No method in multimethod 'normalize-employee' for dispatch value: null
```

This condition can be handled properly by defining a `:default` branch in the dispatcher function.

Implementations for multimethods can be defined in different files, which allows for extensibility. This allows for polymorphism not just based on type, but also based on values.

Some functions are best implemented recursively:

```
(def employees [{:name "Dilbert" :salary 120000}
                {:name "Wally" :salary 130000}
                {:name "Alice" :salary 110000}
                {:name "Boss" :salary 380000}
                {:name "Ashok" :salary 54000}])

(defn sum-payroll
  ([employees] (sum-payroll employees 0))
  ([employees total]
    (if (empty? employees)
      total
      (sum-payroll
        (rest employees)
        (+ total (:salary (first employees)))))))

> (sum-payroll employees)
794000
```

The `sum-payroll` function could run out of stack space if the `employee` vector gets too big. Therefore, Clojure supports *tail call optimization*, by simply replacing the function call with `recur`:

```
(defn sum-payroll
  ([employees] (sum-payroll employees 0))
  ([employees total]
    (if (empty? employees)
      total
      (recur
        (rest employees)
        (+ total (:salary (first employees)))))))
```

The multi-arity function can be simplified to a single-arity function using a `loop` expression:

```
(defn sum-payroll [employees]
  (loop [employees employees total 0]
    (if (empty? employees)
```

```
      total
      (recur
        (rest employees)
        (+ total (:salary (first employees)))))))
```

This construct defines and invokes a pseudo-function, where the `employees` parameter is initialized with the `employees` argument of the `sum-payroll` function; and `total` is initialized to `0`. This values will be re-initialized by `recur` (`employees` to `(rest employees)` and `total` to itself plus the current employees salary).

In practice, higher-ordered functions such as `map` are preferred over `loop`/`recur` constructs.

Since comments are dropped upon compilation, *docstrings* provide a way of documenting code that will be preserved. They are accessible via the `doc` macro:

```
(defn average
  "Computes the average of a and b."
  [a b]
  (/ (+ a b) 2.0))

> (average 3 2)
2.5

> (doc average)
-------------------------
user/average
([a b])
  Computes the average of a and b.
nil
```

Docstrings can also be used for other constructs than functions:

```
> (def dilbert "The smelly IT guy..." {:name "Dilbert" :job "Engineer"})
> (doc dilbert)
-------------------------
user/dilbert
  The smelly IT guy...
nil
```

A map containing `:pre` and `:post` entries can be used to enforce pre- and post-conditions:

```
(defn give-raise [employee amount]
  {:pre  [(⩽ amount 10000) (not= (:name employee) "Ashok")]
   :post [(⩽ (:salary %) 180000)]}
  (assoc employee :salary (+ (:salary employee) amount)))
```

The `:pre` condition makes sure that a raise must not exceed 100000, and that an employee named Ashok will never get a raise.

The `:post` condition makes sure that after a raise, no employee will have a salary of more than 180000. The return value is referred by `%`.

```
> (give-raise {:name "Dilbert" :salary 120000} 5000)
{:name "Dilbert", :salary 125000}

> (give-raise {:name "Wally" :salary 110000} 15000)
Execution error (AssertionError) at user/give-raise (REPL:1).
Assert failed: (⩽ amount 10000)

> (give-raise {:name "Ashok" :salary 45000} 1000)
Execution error (AssertionError) at user/give-raise (REPL:1).
Assert failed: (not= (:name employee) "Ashok")

> (give-raise {:name "Ted" :salary 175000} 8000)
Execution error (AssertionError) at user/give-raise (REPL:1).
Assert failed: (⩽ (:salary %) 180000)
```

6 Functional Things

Functions are values, which can be passed to other functions:

```
(def dilbert {:name "Dilbert" :job "Engineer" :salary 120000})
(def ashok {:name "Ashok" :job "Intern" :salary 45000})

(defn well-paid? [employee]
  (> (:salary employee) 100000))

(defn nerd? [employee]
  (= (:job employee) "Engineer"))

(defn both? [employee pf1 pf2]
  (and (pf1 employee)
       (pf2 employee)))
```

```
> (both? dilbert well-paid? nerd?)
true

> (both? ashok well-paid? nerd?)
false
```

Anonymous functions can be defined using `fn`:

```
(both? dilbert
  (fn [e] (> (:salary e) 100000))
  (fn [e] (= (:name e) "Dilbert")))
```

This can be used to created parametrized functions using a *lexical closure*:

```
(defn cheaper-func [max-salary]
  (fn [employee]
    (< (:salary employee) max-salary)))

(def working-poor? (cheaper-func 50000))
(def cheap-hire? (cheaper-func 100000))

> (working-poor? ashok)
true

> (cheap-hire? dilbert)
false
```

The `apply` function applies a function to each argument:

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

`partial` creates a new function by *partially* filling in the arguments for an existing function. Here, the plus function is partially applied with a single number:

```
> (def increment (partial + 1))
> (increment 1)
2
> (increment 10)
11
```

And here, the `give-raise` function is partially applied to define the `amount` parameter:

```
(def dilbert {:name "Dilbert" :salary 120000 :job "Engineer"})

(defn give-raise [amount employee]
  (assoc employee :salary (+ amount (:salary employee))))

(def small-raise (partial give-raise 1000))

> (small-raise dilbert)
{:name "Dilbert", :salary 121000, :job "Engineer"}
```

`complement` produces a new function by wrapping a function with a `not` call:

```
(defn is-cheap? [employee]
  (⩽ (:salary employee) 100000))

(def is-expensive? (complement is-cheap?))

> (is-cheap? dilbert)
false
> (is-expensive? dilbert)
true
```

`every-pred` combines multiple predicate function with `and`:

```
(defn cheap? [employee]
  (⩽ (:salary employee) 100000))

(defn engineer? [employee]
  (= "Engineer" (:job employee)))

(defn smelly? [employee]
  (= "Dilbert" (:name employee)))

(def fire? (every-pred (complement cheap?) engineer? smelly?))

> (fire? {:name "Dilbert" :salary 120000 :job "Engineer"})
true

> (fire? {:name "Ted" :salary 180000 :job "Marketing"})
false
```

Function literals or *lambdas* can be defined using `#`:

```
> (apply #(+ %1 %2 %3) [1 2 3])
6
```

Since there is no argument list, the arguments are referred to using `%1`, `%2`, etc. The highest-numbered argument defines the number of arguments:

```
> (apply #(+ %5 %6) [1 2 3 4 5 6])
11
```

The arguments one to four (i.e. `[1 2 3 4]`) are ignored.

If only a single argument is needed, it can be referred to by `%` instead of `%1`:

```
> (apply #(* 2 %) [123])
246
```

Use lambdas for very short and simple functions. Use `fn` if named arguments are useful. Use `defn` for lengthy functions with a proper name.

`defn` can be defined in terms of `def` and `fn`:

```
(defn hello [to-whom]
  (println "Hello" to-whom))
```

Has the same effect as:

```
(def hello
  (fn [to-whom]
    (println "Hello" to-whom)))
```

`update` works on a map by applying a function to a map's entry:

```
(def dilbert {:name "Dilbert" :salary 120000 :job "Engineer"})

(defn promote [employee raise-func]
  (update employee :salary raise-func))

> (promote dilbert #(+ % 1000))
{:name "Dilbert", :salary 121000, :job "Engineer"}
```

`update-in` accepts an additional path to locate the field in a nested map to be updated:

```
(def dogbertix {:name "Dogbertix" :ceo {:name "Dogbert" :salary 250000}})

(defn give-bonus [company]
  (update-in company [:ceo :salary] #(* 2 %)))

> (give-bonus dogbertix)
{:name "Dogbertix", :ceo {:name "Dogbert", :salary 500000}}
```

7 Let

`compute-bonus` needs to calculate the same value twice; once for the `if` condition, and once for the return value of the function:

```
(defn compute-bonus [employee bonus-rate max-bonus]
  (if (⩽ (* (:salary employee) bonus-rate) max-bonus)
    (* (:salary employee) bonus-rate)
    max-bonus))

(def dilbert {:name "Dilbert" :salary 120000})

> (compute-bonus dilbert 0.1 5000)
5000

> (compute-bonus dilbert 0.1 25000)
12000.0
```

`let` defines re-usable local bindings:

```
(defn compute-bonus [employee bonus-rate max-bonus]
  (let [bonus (* (:salary employee) bonus-rate)]
    (if (⩽ bonus max-bonus)
      bonus
      max-bonus)))
```

The expression on the right-hand side is assigned to the symbol on the left-hand side of the vector. Multiple local bindings can be created at once:

```
> (let [a 1 b 2 c 3] (println (+ a b c)))
6
```

Later bindings have access to earlier bindings to their left:

```
> (let [a 1 b (* 2 a) c (* 2 b)] (println a b c))
1 2 4
```

The function `compute-bonus` calculates the employee's bonus by looking up their bonus rate in a map:

```
(def employee-bonus-rates
  {"Dilbert" 0.05 "Dogbert" 0.25 "Pointy-Haired Boss" 1.0})

(defn compute-bonus [salary employee-name bonus-rates min-bonus]
  (let [bonus-rate (bonus-rates employee-name)
        bonus (* salary bonus-rate)]
    (if (< bonus min-bonus)
      min-bonus
      bonus)))

> (compute-bonus 120000 "Dilbert" employee-bonus-rates 1000)
6000.0

> (compute-bonus 200000 "Dogbert" employee-bonus-rates 10000)
50000.0
```

The map of `employee-bonus-rates` has to be carried along wherever a bonus has to be calculated. A better approach is to return individual functions by employee that have their bonus rate parametrized:

```
(defn mk-compute-bonus-func [employee-name bonus-rates min-bonus]
  (let [bonus-rate (bonus-rates employee-name)]
    (fn [salary]
      (let [bonus (* bonus-rate salary)]
        (if (< bonus min-bonus)
          min-bonus
          bonus)))))

(def calc-dilbert-bonus
  (mk-compute-bonus-func "Dilbert" employee-bonus-rates 1000))

(def calc-dogbert-bonus
  (mk-compute-bonus-func "Dogbert" employee-bonus-rates 10000))
```

```
> (calc-dilbert-bonus 120000)
6000.0

> (calc-dogbert-bonus 200000)
50000.0
```

`let` is used to bind local variables that are referred to by another function (lexical closure).

The following function outputs book entries with their authors, if available:

```
(def books [{:title "War and Peace" :author "Lev Tolstoy"}
            {:title "Beowulf"}
            {:title "The Name of the Rose" :author "Umberto Eco"}
            {:title "Till Eulenspiegel"}])

(defn output [book]
  (let [author (:author book)]
    (if author
      (str (:title book) " by " author)
      (:title book))))

> (map output books)
("War and Peace by Lev Tolstoy" "Beowulf" "The Name of the Rose by Umberto Eco" "Till Eulenspiegel")
```

`if` and `let` can be combined to `if-let`, making the function shorter:

```
(defn output [book]
  (if-let [author (:author book)]
    (str (:title book) " by " author)
    (:title book)))
```

First, the binding with `let` is created; second, the bound value is evaluated using `if` (yielding `true` for any truthy value).

`when-let` combines `when` with `let` in the same way:

```
(defn writtey-by [book]
  (when-let [author (:author book)]
    (str (:title book) " was written by " author)))

> (map writtey-by books)
("War and Peace was written by Lev Tolstoy" nil "The Name of the Rose was written by Umberto Eco" nil)
```

8 Def, Symbols, and Vars

Like keywords, symbols are values. Whereas keywords evaluate to themselves, symbols created with `def` are bound to other values. The symbol itself can be accessed programmatically using the single quote:

```
> (def first-name "Dilbert")
> first-name
"Dilbert"
> 'first-name
first-name
> (str 'first-name)
"first-name"
> (= 'first-name 'last-name)
false
> (= 'first-name 'first-name)
true
```

A symbol and a value are bound together using a *var*, which is accessible through the pound character and the symbol:

```
> (def first-name "Dilbert")
#'user/first-name
> (def the-name #'first-name)
```

Symbol and value then can be accessed as follows:

```
> (.-sym the-name)
first-name
> (.get the-name)
"Dilbert"
```

Vars are *mutable*, so that bindings can be re-defined during development, say, in a REPL session.

Dynamic bindigs can be changed using `binding` and are, by convention, surrounded by asterisks (`*`) or "earmuffs":

```
> (def ^:dynamic *debug-enabled* false)
> *debug-enabled*
false

> (binding [*debug-enabled* true]
    (println *debug-enabled*))
true
```

Vars are *not* supposed to be used like variables in other programming languages. Use `^:dynamic` vars and `binding` sparingly.

The REPL provides some dynamic vars `*[n]` where `[n]` denotes the n-last result:

```
> (+ 2 1)
3
> (+ 5 4)
9
(- *1 *2) ; 9 - 3
6
```

Dynamic vars can be changed using the `set!` function:

```
> (def employees ["Dilbert" "Wally" "Alice" "Ted" "Ashok"])
> employees
["Dilbert" "Wally" "Alice" "Ted" "Ashok"]
> (set! *print-length* 2)
> employees
["Dilbert" "Wally" ...]
```

`*e` denotes the last exception thrown:

```
> (/ 3 0)
Execution error (ArithmeticException) at user/eval2038 (REPL:1).
Divide by zero
> *e
#error {
 :cause "Divide by zero"
 ...
```

9 Namespaces

Vars, which represent the binding between a symbol and a value, live in *namespaces*. There is always one *current* namespace, affected by calls of `def`.

The REPL creates and uses a namespace called `user`:

```
> (def employee "Dilbert")
#'user/employee
```

The `employee` symbol is bound to the value `"Dilbert"` within the `user` namespace.

A new namespace can be created and made the current namespace using `ns`:

```
> (ns dilbertix)
> (def employees [:dilbert :alice :wally])
#'dilbertix/employees
```

Calling `ns` with an existing namespace makes that namespace the current one, without changing it:

```
> (ns user)
> (def today "Sunday")
#'user/today
```

After switching back, the bindings are still available:

```
> (ns dilbertix)
> employees
[:dilbert :alice :wally]
```

Symbols from other namespaces can be referred to using a *fully qualified symbol*:

```
> (ns user)
> dilbertix/employees
```

Namespaces defined in other files need to be loaded before they can be used. The function `clojure.data/diff` is unavailable by default:

```
> (def engineers [:alice :dilbert :wally])
> (def high-performers [:alice :dilbert :topper])
> (clojure.data/diff engineers high-performers)
Execution error (ClassNotFoundException) at java.net.URLClassLoader/findClass (URLClassLoader.java:382).
```

The `clojure.data` namespace can be made available using `require`:

```
> (require 'clojure.data)
> (clojure.data/diff engineers high-performers)
[[nil nil :wally] [nil nil :topper] [:alice :dilbert]]
```

Given a new project skeleton:

```
$ lein new app dilbertix
```

Containing the file `src/dilbertix/core.clj`:

```
(ns dilbertix.core
  (:gen-class))

(defn -main
  "I don't do a whole lot ... yet."
  [& args]
  (println "Hello, World!"))
```

The namespace `dilbertix.core` and the file's location `dilbertix/core.clj` match together: A namespace `foo.bar` is to be found in a file `foo/bar.clj`.

Dashes need to be converted to underscores: The namespace `foo-bar.qux` is to be found in the file `foo_bar/qux.clj`.

Thus, a new namespace `dilbertix.employees` is to be defined within the project folder in the file `src/dilbertix/employees.clj`:

```
(ns dilbertix.employees)

(def job-satisfaction 0.0021)

(def employees [:dilbert :alice :wally])
```

The namespace definition can be supplied with `:require` expressions:

```
(ns dilbertix.core
  (:require dilbertix.employees)
  (:gen-class))

(defn -main
  [& args]
  (println "Our Employees:" dilbertix.employees/employees))
```

Notice the difference between the stand-alone `require`:

```
(require 'dilbertix.employees) ; symbol, quoted
```

And the expression within the `ns` definition:

```
(:require dilbertix.employees) ; keyword, unquoted
```

Aliases can be defined to make imported names shorter:

```
(require '[dilbertix.employees :as employees])
```

Or within the namespace definition:

```
(ns dilbertix.core
    (:require [dilbertix.employees :as employees]))
```

Which allows for shorter references:

```
(println employees/job-satisfaction)
```

Instead of:

```
(println dilbertix.employees/job-satisfaction)
```

Aliases don't mask ordinary bindings, so an `employees` var would still be visible.

Using `:refer`, vars from another namespace are pulled into the current namespace:

```
(require '[dilbertix.employees :refer [employees job-satisfaction])
```

Which would mask an existing `employees` binding in the current namespace. Therefore, `:refer` should be used sparingly.

The current namespace is available through the symbol `*ns*`:

```
> (println *ns*)
#object[clojure.lang.Namespace 0x38158523 user]
```

Existing namespaces can be looked up by their name:

```
> (find-ns 'user)
#object[clojure.lang.Namespace 0x38158523 "user"]
```

Namespaces can be discovered:

```
> (ns-map *ns*)
{primitives-classnames #'clojure.core/primitives-classnames, +' #'clojure.core/+' ...
;; omitted
```

The namespace is part of the symbol and can be extracted using the `namespace` function:

```
> (def hello "world")
> (namespace 'user/hello)
"user"
```

The namespace `clojure.core` provides functions such as `println` or `first` and is made ready automatically:

```
(require '[clojure.core :refer :all])
```

There is *no* hierarchy of namespaces. The dots in `clojure.core.data` are just part of the name.

Using `require`, a namespace only gets loaded once, which is sensible for code within files. In a REPL session, reloading modified code is common, and the `:reload` keyword can be used:

```
(require :reload '[dilbertix.employees :as employees])
```

Symbols no longer needed can be removed using `ns-unmap`:

```
(ns-unmap 'dilbertix.employees)
```

`defonce` makes sure a symbol is only bound to a value *once*, even if the definition is required using `:reload`:

```
(defonce answer-to-everything (summarize-all-books))
```

If the symbol is supposed to be rebound nonetheless, it can be unmapped:

```
(ns-unmap *ns* 'answer-to-everything)
```

10 Sequences

The different collection types (maps, vectors, lists, sets) share a common wrapper interface called a *sequence*. That's why the `count` function (and many others) work on different kinds of collections:

```
> count [1 2 3]) ; vector: number of items
3
> (count {:name "John" :age 29}) ; map: number of key-value pairs
2
```

The `seq` function wraps any collection in a sequence:

```
> (seq {:age 42 :name "Dilbert"})
([:age 42] [:name "Dilbert"]) ; sequence of key-value pairs

> (seq ["Alice", "Dilbert", "Wally"])
("Alice" "Dilbert" "Wally") ; looks like a list, is a sequence
```

`seq` returns `nil` when invoked on an empty collection:

```
> (seq '()) ; empty list
nil
> (seq [])  ; empty vector
nil
> (seq {})  ; empty map
nil
> (seq #{}) ; empty set
nil
```

Like `rest`, `next` returns all but the first elements of a sequence:

```
> (rest [1 2 3])
(2 3)
> (next [1 2 3])
(2 3)
```

Unlike `rest`, `next` returns `nil` if the remainder is an empty sequence:

```
> (rest [1])
()
> (next [1])
nil
```

Always rely on `rest` and `next` returning an empty collection or `nil`, respectively; *never* compare the result of `first` against `nil` for this purpuse:

```
;; bad idea!
(defn is-empty [collection]
  (= (first (seq collection)) nil))

> (is-empty [1 2 3])   ; correct
false
> (is-empty [])        ; correct
true
> (is-empty [nil 1 2]) ; wrong!
true

;; better approach
(defn is-empty [collection]
    (= (next (seq collection)) nil))

> (is-empty [1 2 3])   ; correct
false
> (is-empty [])        ; correct
true
> (is-empty [nil 1 2]) ; correct
false
```

New elements can be added to the front of a sequence using `cons`:

```
> (cons 0 (seq [1 2 3]))
(0 1 2 3)
> (cons [:job "Engineer"] (seq {:name "Dilbert" :age 42}))
([:job "Engineer"] [:name "Dilbert"] [:age 42])
```

`rest`, `next`, `cons` (but *not* `conj`), `sort`, `reverse` all return sequences.

A *seqable* is something that `seq` can turn into a sequence.

`partition` chops a sequence (or *seqable*) into a sequence of smaller junks:

```
> (partition 2 [1 2 3 4 5])
((1 2) (3 4))
```

`interleave` zips two sequences together:

```
> (interleave [1 3 5 7 9] [2 4 6 8])
(1 2 3 4 5 6 7 8)
```

`interpose` adds a separator value in between the elements:

```
> (interpose "and" ["Dilbert" "Wally" "Alice"])
("Dilbert" "and" "Wally" "and" "Alice")
> (interpose '+ [1 2 3])
(1 + 2 + 3)
```

`filter` accepts a predicate function and a sequence, and returns a new sequence holding the elements for which the predicate holds true:

```
> (defn negative? [x] (< x 0))
> (filter negative? [5 -5 10 -10])
(-5 -10)

(defn useful? [employee]
    (not= (:job employee) "Manager"))

> (filter useful? [{:name "Pointy Haired Boss" :job "Manager"}
                   {:name "Dilbert" :job "Engineer"}])
({:name "Dilbert", :job "Engineer"})
```

Like `filter`, `some` applies a predicate to the elements of a sequence. Unlike `filter`, it returns the first truthy value returned by the predicate:

```
> (some neg? [1 2 3])
nil
> (some neg? [1 2 -3])
true

(defn useful-name [employee]
  (when
    (not= (:job employee) "Manager")
    (:name employee)))
```

```
> (some useful-name [{:name "Pointy Haired Boss" :job "Manager"}
                     {:name "Dilbert" :job "Engineer"}])
"Dilbert"
```

`map` transforms the elements of a sequence using a function:

```
(defn raise-salary [employee]
  (assoc employee :salary (* 1.2 (:salary employee))))

> (map raise-salary [{:name "Dilbert" :salary 120000}
                     {:name "Ashok" :salary 10000}])
({:name "Dilbert", :salary 144000.0} {:name "Ashok", :salary 12000.0})

> (map :name [{:name "Dilbert" :salary 120000}
              {:name "Ashok" :salary 10000}])
("Dilbert" "Ashok")
```

`comp` (for "compose") produces a function by applying the argument functions from right to left (first, the salary is raised; second, the `:name` is extracted):

```
> (map (comp :name raise-salary)
    [{:name "Dilbert" :salary 120000}
     {:name "Ashok" :salary 10000}])
(144000.0 12000.0)
```

`for` processes a sequence element by element:

```
> (def employees [{:name "Dilbert" :salary 120000}
                  {:name "Ashok" :salary 10000}])
> (for [e employees] (:name e))
("Dilbert" "Ashok")
```

`reduce` combines the elements of a sequence into a single value. It works with a function that requires *two* values: an accumulator and the current element:

```
> (reduce (fn [acc x] (+ acc x)) [1 2 3 4])
10
> (reduce (fn [acc x] (* acc x)) [1 2 3 4])
24
```

The starting value of the accumulator can be defined, too:

```
> (reduce (fn [acc x] (+ acc x)) 100 [1 2 3 4])
110
```

If the start value is left out, the first element will be used for it.

Since arithmetic operators are functions, this can be simplified:

```
> (reduce + [1 2 3 4])
10
> (reduce * [1 2 3 4])
24
```

Those higher-order functions can be used to compose elegant solutions:

```
(def employees [{:name "Dilbert" :salary 120000}
                {:name "Wally" :salary 130000}
                {:name "Alice" :salary 11000}
                {:name "Dogbert" :salary 180000}
                {:name "Topper" :salary 150000}])

(defn top-earners [n employees]
  (apply
    str
    (interpose
      " ⩾ "
      (map :name (take n (reverse (sort-by :salary employees)))))))

> (top-earners 3 employees)
"Dogbert ⩾ Topper ⩾ Wally"

> (top-earners 5 employees)
"Dogbert ⩾ Topper ⩾ Wally ⩾ Dilbert ⩾ Alice"

> (top-earners 1 employees)
"Dogbert"
```

The definition of `top-earners` needs to be read from the inside out. The pointy arrow function `↠` allows for an easier to read syntax without any runtime performance overhead:

```
(defn top-earners [n employees]
  (->>
    employees
    (sort-by :salary)
    reverse
    (take n)
    (map :name)
    (interpose " ⩾ ")
    (apply str)))
```

The `->>` function uses the result of a function as the *last* argument for the next function all; `->` as the *first* argument for the subsequent function call.

Since Clojure provides so many ways of processing sequences, turning something into a sequence can be a big step to solving that problem.

`line-seq` turns the lines of a text file into a (lazy) sequence. Given this CSV employee data base (`employees.txt`):

```
Pointy Haired Boss;Manager;58;250000
Dilbert;Engineer;42;120000
Alice;Engineer;39;110000
Wally;Engineer;52;130000
Dogbert;Consultant;7;390000
Topper;Salesman;35;850000
Ted;Project Manager;45;280000
```

`read-employee-db` turns it into a sequence of maps:

```
(require '[clojure.java.io :as io])
(require '[clojure.string :as str])

(defn split-by [sep]
  (fn [line]
    (str/split line sep)))

(defn to-employee [values]
  (zipmap [:name :job :age :salary] values))

(defn read-employee-db [filename]
  (with-open [r (io/reader filename)]
    (map
      (comp to-employee (split-by #";"))
```

```
      (doall (line-seq r)))))

> (read-employee-db "employees.txt")
((:name "Pointy Haired Boss" :job "Manager" :age "58" :salary "250000")
 (:name "Dilbert" :job "Engineer" :age "42" :salary "120000")
 (:name "Alice" :job "Engineer" :age "39" :salary "110000")
 (:name "Wally" :job "Engineer" :age "52" :salary "130000")
 (:name "Dogbert" :job "Consultant" :age "7" :salary "390000")
 (:name "Topper" :job "Salesman" :age "35" :salary "850000")
 (:name "Ted" :job "Project Manager" :age "45" :salary "280000"))
```

Which then can be processed using the `top-earners` function from before:

```
> (top-earners 3 (read-employee-db "employees.txt"))
"Topper ⩾ Dogbert ⩾ Ted"
```

Using regular expressions, strings can be turned into sequences (e.g. of words):

```
> (re-seq #"\w+" "this is some sentence to be split") ; split by whitespace
("this" "is" "some" "sentence" "to" "be" "split")
```

Even though sequences are extremely useful, data structures like maps and vectors loose *some* of their power when wrapped as a sequence.

11 Lazy Sequences

Unlike sequences, *lazy sequences* only make up their values as they are actually needed.

`repeat` creates a lazy sequence that contains the given element unlimited times:

```
> (def words (repeat "duck"))
```

There are, of course, not unlimited instances of the string `"duck"` put into memory, which would be impossible to do. The elements of the lazy sequence are only created when used:

```
> (nth words 3)
"duck"
> (nth words 123456)
"duck"

> (take 3 words)
("duck" "duck" "duck")
```

`cycle` creates a lazy sequence by repeating a given sequence endlessly:

```
> (take 7 (cycle [1 2 3]))
(1 2 3 1 2 3 1)
```

`iterate` creates a lazy sequence based on a function. The first argument is a function to be called for each subsequent iteration; the second argument is a starting value:

```
> (def counter (iterate inc 1))
> (take 3 counter)
(1 2 3)
> (take 12 counter)
(1 2 3 4 5 6 7 8 9 10 11 12)
```

Note that the lazy sequence is *not consumed* like an iterator in other programming languages.

This `counter` sequence can be used to enumerate items of a sequence:

```
> (interleave ["Alive" "Dilbert" "Wally"] counter)
("Alive" 1 "Dilbert" 2 "Wally" 3)
```

Or:

```
> (zipmap counter ["Alive" "Dilbert" "Wally"])
{1 "Alive", 2 "Dilbert", 3 "Wally"}
```

`map` is lazy, so it can be applied to unbound lazy sequences:

```
> (defn twice [x] (* 2 x))
> (def doubled (map twice counter))
> (take 7 doubled)
(2 4 6 8 10 12 14)
```

Combined with `cycle`, `map` can be used to combine existing elements in all possible ways (permutations):

```
(def names ["Alice" "Dilbert" "Wally" "Ashok" "Dogbert"])
(def adjectives ["great" "lazy" "nerdy" "evil"])
(def professions ["engineer" "manager" "consultant"])

(defn combine-employees [name adjective profession]
  (str name " the " adjective " " profession))

(def employees
  (map combine-employees
    (cycle names)
    (cycle adjectives)
    (cycle professions)))

> (take 8 employees)
("Alice the great engineer" "Dilbert the lazy manager" "Wally the nerdy consultant"
 "Ashok the evil engineer" "Dogbert the great manager" "Alice the lazy consultant"
 "Dilbert the nerdy engineer" "Wally the evil manager")
```

`lazy-seq` creates a lazy sequence from an existing sequence:

```
> (def numbers (lazy-seq [1 2 3]))
(take 2 numbers)
(1 2)
```

The following function generates an infinite number of Fibonacci numbers as a lazy sequence:

```
(defn fibs
  ([] (fibs 1 1))
  ([a b] (cons a (lazy-seq (fibs b (+ a b))))))
```

The implementation of arity 0 just calls the implementation of arity 2 with the first two Fibonacci numbers. The current element is consed onto the sequence; the next element is then computed using a recursive call by increasing the numbers:

```
> (take 10 (fibs))
(1 1 2 3 5 8 13 21 34 55)
```

Never output lazy sequences as if they were finite:

```
> (def counter (iterate inc 1))
> counter ; bad idea
```

Or at least be sure to set `*print-length*` before doing so:

```
> (set! *print-length* 10)
> counter
(1 2 3 4 5 6 7 8 9 10 ...)
```

`doall` realizes a lazy sequence, which should only be done for *finite* lazy sequences:

```
> (doall counter) ; bad idea, again...
```

`doseq` is similar to `for` and useful if the iteratio step is more interesting than the result:

```
> (def numbers (take 5 counter))
> (doseq [n numbers]
    (println "Current iteration" n))
Current iteration 1
Current iteration 2
Current iteration 3
Current iteration 4
Current iteration 5
```

Infinite sequences should not be sorted or reduced.

Many functions are lazy, such as `take`.

Notice that working with lazy sequences opens a timely gap between when the instruction to do something is given and when it is actually done. This can cause trouble when working with side-effects (e.g. files read/written with `slurp`/`spit` whose content changes in the meantime).

12 Destructuring

Destructuring is a tool for unpacking data structures with little syntax:

```
> (def employees ["Dilbert" "Wally"])
> (let [[nerd lazybone] employees]
    (println nerd "is a nerd")
    (println lazybone "is a lazybone"))
Dilbert is a nerd
Wally is a lazybone
```

The left side vector of `let` describes the data to be extracted. The right side expression is the data structure to be unpacked.

The unpacking need not be exhaustive. Values can be simply dropped on the right side:

```
> (let [[a b c] ["foo" "bar" "baz" "qux"]] ; "qux" ignored
    (println a b c))
foo bar baz
```

And the dummy symbol _ can be used to drop values from the left side:

```
> (let [[_ a _ b] ["foo" "bar" "baz" "qux"]] ; "foo" and "qux" ignored
    (println a b))
bar qux
```

Nested structures can be destructured using a nested pattern on the left side:

```
> (def teams [["Dilbert" "Alice" "Wally"] ["Dogbert" "Ratbert" "Catbert"]])
> (let [[[dilbert _ wally] [dogbert _ catbert]] teams]
    (println dilbert wally dogbert catbert))
```

Anything that can be turned into a sequence can be destructured:

```
> (let [[one _ _ four] '(1 2 3 4)]
    (println one four))
1 4

> (let [[b a r] "bar"]
    (println b a r))
b a r

> (let [[a _ b _ c] (iterate inc 1)]
    (println a b c))
1 3 5
```

Destructuring can not only be used with `let`, but also when calling functions:

```
> (defn fire [[scapegoat-one scapegoat-two]]
    (println scapegoat-one "and" scapegoat-two "are fired!"))
> (fire ["Dilbert" "Alice" "Wally" "Dogbert"])
Dilbert and Alice are fired!
```

When destructuring maps, the variable to be bound stands on the left, and the keyword for the value to be extracted stands on the right:

```
> (def employees {:engineer "Dilbert" :consultant "Dogbert" :slacker "Wally"})
> (let [{scapegoat :engineer} employees]
    (println scapegoat))
Dilbert
```

Nested data structures can be destructured, too:

```
> (def company {:name "Random Inc."
                :employees [{:name "Dilbert" :role "Engineer"}
                            {:name "Dogbert" :role "Consultant"}]})
> (let [{[{looser-job :role} {leech :name}] :employees} company]
    (println looser-job "is the worst job and" leech "is a leech"))
Engineer is the worst job and Dogbert is a leech
```

If all of a map's values are to be bound, listing all the keys is cumbersome:

```
> (def employees [{:name "Dilbert" :role "Engineer" :age 42}
                  {:name "Dogbert" :role "Consultant" :age 7}])
> (defn describe [{name :name role :role age :age}]
    (println name "is a" age "year old" role))
> (map describe employees)
Dilbert is a 42 year old Engineer
Dogbert is a 7 year old Consultant
```

The `:keys` keyword allows for a shorter mapping:

```
> (defn describe [{:keys [name role age]}]
    (println name "is a" age "year old" role))
```

If the passed value should not only be destructured, but also retained in its entirety, the `:as` keyword can be used.

```
> (defn add-greeting [{:keys [name role age] :as employee}]
    (assoc employee
           :greeting
           (str "I'm a " age " year old " role " called " name)))
> (map add-greeting employees)
({:name "Dilbert", :role "Engineer", :age 42,
```

```
  :greeting "I'm a 42 year old Engineer called Dilbert"}
 {:name "Dogbert", :role "Consultant", :age 7,
  :greeting "I'm a 7 year old Consultant called Dogbert"})
```

Default values can be provided using the `:or` keyword:

```
> (defn email [{:keys [user host domain]
                :or {user "root", host "localhost", domain "local"} :as parts}]
    (str user "@" host "." domain))
> (email {:user "john"})
"john@localhost.local"
> (email {:host "dilbertix" :domain "com"})
"root@dilbertix.com"
```

Destructuring can't be used directly with `def`, only within `let`.

13 Records and Protocols

A common tradeoff in programming is between *generic* and *specialized* solutions. Maps are *generic* and very flexible. They can deal with arbitrary keys, which comes with a runtime penalty when dealing with huge amounts of data.

Records are specialized data structures dealing only a set of predefined keys:

```
> (defrecord Employee [name age job salary])
```

`defrecord` creates three `vars`: one for the record type, and two factory functions `→Employee` and `map→Employee` to create instances of the record type:

```
> (def dilbert (->Employee "Dilbert" 42 "Engineer" 120000))

> (def alice (map->Employee
    {:name "Alice"
     :age 37
     :job "Engineer"
     :salary 110000}))
```

An instance of a record can be used like a map:

```
> (:name dilbert)
"Dilbert"

> (:job alice)
"Engineer"

> (keys dilbert)
(:name :age :job :salary)

> (def alice-promoted (assoc alice :salary 120000 :job "Head of Engineering"))
> alice-promoted
#user.Employee{:name "Alice", :age 37, :job "Head of Engineering", :salary 120000}
```

It's also possible to associate *extra fields* with a record; however, those don't get the speed optimization of the record's defined fields:

```
> (def dilbert-secret (assoc dilbert :note "Smelly and ugly guy"))
> dilbert-secret
#user.Employee{:name "Dilbert", :age 42, :job "Engineer",
               :salary 120000, :note "Smelly and ugly guy"}
```

While the speed advantage of records is only noticable for large amounts of data, the documentation provided by records is always helpful:

```
> (defrecord Poet [name century works])
> (defrecord FictionalCharacter [name show traits])

> (def homer-1
    (->Poet "Homer" "8th/7th B.C." ["Iliad" "Odyssey"]))
> (def homer-2
    (->FictionalCharacter "Homer" "The Simpsons" ["lazy" "stupid" "impulsive"]))
```

`class` returns the underlying type of a record instance:

```
> (class homer-1)
user.Poet

> (class homer-2)
user.FictionalCharacter
```

`instance?` (like Java's `instanceof`) checks if an instance if of a specific record type:

```
> (instance? FictionalCharacter homer-1)
false

> (instance? FictionalCharacter homer-2)
true
```

This offers one primitive way to create polymorphic functions:

```
(defn output [x]
  (if (instance? FictionalCharacter x)
    (println (:name x) "the" (:traits x) "character from" (:show x))
    (println (:name x) "the author of" (:works x) "who lived" (:century x))))

> (output homer-1)
Homer the author of [Iliad Odyssey] who lived 8th/7th B.C.

> (output homer-2)
Homer the [lazy stupid impulsive] character from The Simpsons
```

However, *protocols* are a better alternative for this purpose. A protocol (here: `Person`) defines a set of functions (here: `describe`, `greet`) that can be performed on different kinds of records implementing that protocol:

```
(defprotocol Person
  (describe [this])
  (greet [this msg]))
```

The functions of the protocol need to be implemented by the records. The protocol name (here: `Person`) is followed by the *method definitions*:

```
(defrecord Poet [name century works]
  Person
  (describe [this]
    (str
      (:name this)
      ", the author of " (clojure.string/join ", " (:works this))
      " who lived in " (:century this)))
  (greet [this msg]
    (str
      msg " " (:name this)
      ", author of " (clojure.string/join ", " (:works this)))))
```

```
(defrecord FictionalCharacter [name show traits]
  Person
  (describe [this]
    (str
      (:name this)
      ", the " (clojure.string/join ", " (:traits this))
      " character from " (:show this)))
  (greet [this msg]
    (str
      msg " " (:name this)
      ", you " (clojure.string/join ", " (:traits this))
      " character from " (:show this))))
```

The first argument (conventionally called `this`) refers to the instance the function is called on:

```
> (def homer-1
    (->Poet "Homer" "8th/7th B.C." ["Iliad" "Odyssey"]))
> (describe homer-1)
"Homer, the author of Iliad, Odyssey who lived in 8th/7th B.C."
> (greet homer-1 "Greetings")
"Greetings Homer, author of Iliad, Odyssey"

> (def homer-2
    (->FictionalCharacter "Homer" "The Simpsons" ["lazy" "stupid" "impulsive"]))
> (describe homer-2)
"Homer, the lazy, stupid, impulsive character from The Simpsons"
> (greet homer-2 "Hi")
"Hi Homer, you lazy, stupid, impulsive character from The Simpsons"
```

New protocols can be created and implemented for existing types using `extend-protocol`:

```
(defprotocol Greetable
  (say-hi [this]))

(extend-protocol Greetable
  Employee
  (say-hi [employee]
    (str "Hello, I'm " (:name employee) ". I work as a " (:job employee) "."))
  Poet
  (say-hi [poet]
    (str "Greetings, I'm " (:name poet) ", author of "
         (clojure.string/join ", " (:works poet)) ".")))
```

```
  FictionalCharacter
  (say-hi [character]
    (str "Hi, I'm " (:name character) " from " (:show character) ".")))

> (say-hi dilbert)
"Hello, I'm Dilbert. I work as a Engineer."

> (say-hi homer-1)
"Greetings, I'm Homer, author of Iliad, Odyssey"

> (say-hi homer-2)
"Hi, I'm Homer from The Simpsons."
```

It is also possible to implement protocols for existing data types:

```
(extend-protocol Greetable
  String
  (say-hi [string]
    (str "Hi, I'm the String '" string "'."))
  Boolean
  (say-hi [bool]
    (str "Hi, I'm the Boolean '" bool "'.")))

> (say-hi "foobar")
"Hi, I'm the String 'foobar'."

> (say-hi false)
"Hi, I'm the Boolean 'false'."
```

Records and their instances resemble classes and objects, but they don't have hierarchies and are immutable. Protocols are similar to abstract classes or interfaces, but more flexible: They can be implemented without touching the definition of the type the methods are implemented for; and, again, they don't come in hierarchies. Also, most object-oriented programming languages require the programmer to use classes, objects, and interfaces. Records and protocols, however, are optional: Better start without them, and only use them if they bring some tangible benefit.

Protocols and multimethods have a lot in common, but also some differences:

- Multimethods define single, stand-alone operations. Protocols group related operations together.
- Multimethods support an arbitrary dispatch mechanism. Protocols dispatch based on a type.

Simple one-off implementations of a protocol as a single instance (say, for test doubles) can be created using `reify`:

```
(def dirty-harry
  (reify Person
    (describe [_] "Lieutenant Harry Callahan, San Francisco Police Department")
    (greet [_ msg] (str msg ", punk. Feeling lucky today?"))))

> (describe dirty-harry)
"Lieutenant Harry Callahan, San Francisco Police Department"

> (greet dirty-harry "Hi, there")
"Hi there, punk. Feeling lucky today?"
```

Since `this` wasn't used in any of the method bodies, it was replaced by _.

The methods defined for a protocol could pollute the namespace:

```
> (defprotocol Items (count [this]))
Warning: protocol #'user/Items is overwriting function count
```

When in doubt, put protocols in their own namespace.

`deftype` is a more generic version of `defrecord` and requires the programmer to provide all of the behaviour of a new type. This is more work than defining a new record, but allows for more flexibility.

14 Tests

For the following example, a new project `company` is created:

```
$ lein new company
```

A fixed set of employees is defined in `src/company/employees.clj`:

```
(ns company.employees)

(def employees [{:name "Dilbert" :age 42 :job "Engineer" :salary 120000}
                {:name "Alice" :age 37 :job "Engineer" :salary 115000}
                {:name "Wally" :age 47 :job "Engineer" :salary 130000}
                {:name "Pointy Haired Boss" :age 57 :job "Manager" :salary 250000}
```

```
                {:name "Ashok" :age 22 :job "Intern" :salary 18000}
                {:name "Dogbert" :age 7 :job "Consultant" :salary 470000}
                {:name "Catbert" :age 9 :job "Head of HR" :salary 190000}])
```

Some functions to operate on a set of employees are provided in `src/company/core.clj`:

```
(ns company.core)

(defn find-by-name
  "Search for an employee by name (unique result)"
  [employees by-name]
  (first (filter #(= (:name %1) by-name) employees)))

(defn sum-salaries
  "Sum up the salaries of all given employees"
  [employees]
  (reduce #(+ %1 %2) (map #(:salary %1) employees)))
```

Unit tests can be created using the `clojure.test` library. In `src/company/core_test.clj`, test cases for the functions in `company.core` are defined. The test functions from `clojure.core`, the functions to be tested from `company.core`, and the static set of employees in `company.employees` are required:

```
(ns company.core-test
  (:require [clojure.test :refer :all])
  (:require [company.core :as cc])
  (:require [company.employees :as ce]))
```

A simple test is defined using `deftest` and the `is` assertion function:

```
(deftest test-finding-employee-by-name
  (is (not (nil? (cc/find-by-name ce/employees "Dilbert")))))
```

The test can be executed with Leiningen:

```
$ lein test
lein test company.core-test

Ran 1 tests containing 1 assertions.
0 failures, 0 errors.
```

To test another function, another `deftest` is created:

```
(deftest test-sum-up-employee-salaries
  (is (= 1293000 (cc/sum-salaries ce/employees))))
```

Multiple test cases can be grouped together and described using `testing` as so-called *subtests* or *contexts*:

```
(deftest test-finding-employee-by-name
  (testing "Finding employees"
    (is (not (nil? (cc/find-by-name ce/employees "Dilbert"))))
    (is (not (nil? (cc/find-by-name ce/employees "Catbert")))))
  (testing "Not finding employees"
    (is (nil? (cc/find-by-name ce/employees "Sharkbert")))
    (is (nil? (cc/find-by-name ce/employees "Competent Boss")))))
```

In order to test a property of the code, one needs to work with examples that *exercise that property*. Instead of making up examples manually, the `test.check` library can be used for *Property-Based Testing*. First, `test.check` needs to be added as a dependency (`project.clj`):

```
:dependencies [[org.clojure/clojure "1.10.1"]
               [org.clojure/test.check "1.1.0"]]
```

Random data can then be created using *generators*:

```
> (require '[clojure.test.check.generators :as gen])
> (gen/sample gen/string-alphanumeric)
("" "M" "Lw" "cs9" "CtQU" "yOg95" "YE" "3XTEL" "001qEF3w" "ZTwzwZ")
```

`gen/string-alphanumeric` generates an endless stream of alphanumeric strings (including empty ones), `gen/sample` takes a sample of that stream. In order to test the functions for the employee data base, the constrained generators for the following map keywords are needed:

- `:name`: alphanumeric, non-empty
- `:age`: numeric, positive, non-zero
- `:job`: alphanumeric, non-empty
- `:salary`: numeric, positive, non-zero

The constraints can be modeled using `such-that` predicates:

```
(def text-gen
  (gen/such-that not-empty gen/string-alphanumeric))

> (gen/sample text-gen)
("1JB" "91" "t" "FJyD" "34eM" "3h0a5" "9fhzo" "8v31R" "00O83F" "PNEEsZQMe")

(def num-gen
  (gen/such-that (complement zero?) gen/pos-int))

> (gen/sample num-gen)
(1 3 1 3 2 4 1 3 5 5)
```

A single employee map can be created using those functions:

```
(def employee-gen
  (gen/hash-map :name text-gen
                :age num-gen
                :job text-gen
                :salary num-gen))

> (gen/sample employee-gen)
({:name "wlf", :age 3, :job "8", :salary 1}
 {:name "5", :age 1, :job "ZR", :salary 1}
 {:name "Hd", :age 2, :job "L3", :salary 3}) ; output shortened
```

An endless supply of non-empty employee databases (vectors) can be generated:

```
(def payroll-gen
  (gen/not-empty (gen/vector employee-gen)))

> (gen/sample payroll-gen)
([{:name "uQ", :age 2, :job "Zl", :salary 2}]
 [{:name "VR", :age 2, :job "ME", :salary 2}
  {:name "vj", :age 3, :job "8", :salary 1}]
 [{:name "BYsO", :age 2, :job "p7", :salary 2}]) ; output shortened
```

In order to conduct tests, a random example has to be plucked from the test data. Here, `gen/let` is used to create a map containing a payroll together with one single employee taken from that payroll (from the `inventory` built by `inventory-gen`, assign a random, single element to `book`):

```
(def payroll-and-employee-gen
  (gen/let [payroll payroll-gen
            employee (gen/elements payroll)]
    {:payroll payroll :employee employee}))

> (gen/smaple payroll-and-employee-gen)
({:payroll [{:name "Y0", :age 2, :job "K0", :salary 1}],
  :employee {:name "Y0", :age 2, :job "K0", :salary 1}}
  :employee {:name "Q", :age 3, :job "T", :salary 1}}
 {:payroll [{:name "qv", :age 3, :job "2ys", :salary 3}
            {:name "n", :age 2, :job "893", :salary 1}
            {:name "2g", :age 3, :job "cLx", :salary 2}],
  :employee {:name "2g", :age 3, :job "cLx", :salary 2}}) ; output shortened
```

Once the functions to generate the test data are ready, the property needs to be expressed with property test functions, provided by the `test.check` library:

```
> (require '[clojure.test.check.properties :as prop])
```

A *theorem*—e.g. the increment of a number is bigger than that number—can be expressed using `prop/for-all`:

```
(prop/for-all [i gen/pos-int]
  (< i (inc i)))
```

Computers can't prove theorems, but only execute tests, for which `test.check` provides functions:

```
> (require '[clojure.test.check :as tc])
```

To perform a test, a limit (e.g. 50) of examples to be considered needs to be supplied using `tc/quick-check`, which wraps the theorem from above:

```
(tc/quick-check 50
  (prop/for-all [i gen/pos-int]
    (< i (inc i))))
```

This function produces an output describing the tests conducted:

```
{:result true, :pass? true, :num-tests 50, :time-elapsed-ms 4,
 :seed 1622715897727}
```

(The seed value could be used to reproduce the random data that was generated.)

Finally, these tools can be combined to write property test functions:

```
(def tc/quick-check 50
  (prop/for-all [p-and-e payroll-and-employee-gen]
    (= (cc/find-by-name (:payroll p-and-e) (-> p-and-e :employee :name))
       (:employee i-and-e))))
```

For each payroll/employee sample, the function to be tested `cc/find-by-name` is called with the whole payroll and the randomly picked employee's name. The result is than compared to random employee extracted by the generator before.

In order to integrate this property-based test into the native test runner, `defspec` from `test.check` can be used:

```
> (require '[clojure.test.check.clojure-test :as ctest])
```

The test can then be defined using `ctest/decspec` as follows:

```
(ctest/defspec find-by-name-finds-employee 50
  (prop/for-all [p-and-e payroll-and-employee-gen]
    (= (cc/find-by-name (:payroll p-and-e) (-> p-and-e :employee :name))
       (:employee p-and-e))))
```

Which then can be tested using Leiningen:

```
$ lein test

lein test company.core-property-test
{:result true, :num-tests 50, :seed 1622716789601, :time-elapsed-ms 92,
 :test-var "find-by-name-finds-employee"}

lein test company.core-test

Ran 3 tests containing 6 assertions.
0 failures, 0 errors.
```

Here's the whole property-based test for the employee database (`test/company/core_property_test.clj`):

```
(ns company.core-property-test
  (:require [clojure.test :refer :all])
  (:require [company.core :as cc])
  (:require [clojure.test.check :as tc])
  (:require [clojure.test.check.clojure-test :as ctest])
  (:require [clojure.test.check.generators :as gen])
  (:require [clojure.test.check.properties :as prop]))

(def text-gen
  (gen/such-that not-empty gen/string-alphanumeric))

(def num-gen
  (gen/such-that (complement zero?) gen/pos-int))

(def employee-gen
  (gen/hash-map :name text-gen
                :age num-gen
                :job text-gen
                :salary num-gen))

(def payroll-gen
  (gen/not-empty (gen/vector employee-gen)))

(def payroll-and-employee-gen
  (gen/let [payroll payroll-gen
            employee (gen/elements payroll)]
    {:payroll payroll :employee employee}))

(ctest/defspec find-by-name-finds-employee 50
  (prop/for-all [p-and-e payroll-and-employee-gen]
    (= (cc/find-by-name (:payroll p-and-e) (-> p-and-e :employee :name))
       (:employee p-and-e))))
```

While unit tests are easier to implement and understand, they often cover only a few hand-picked examples, leaving much of the input space untested. Property-based testing is harder to implement and understand, but covers a much wider space of possible inputs. However, one might jump to the wrong conclusion that *all* possibilities are covered, where generators probably miss some basic but crucial cases (e.g. picking 0 as a random number for testing division by zero).

Therefore, it's a good idea to start with some unit tests covering the basic cases (say, all combinations of a devision with positive and negative numbers, and zero). Property-based tests can then be introduced to cover more of the input space.

Notice that property-based tests are non-deterministic. Re-starting a failed test pipeline might yield a test run without errors, but the underlying error remains.

Even having only one single trivial test case is way better than having no test cases at all:

```
(require '[clojure.test :refer :all])
(require '[company.core :as cc])
(require '[company.employees :as ce])

(deftest test-sum-up-employee-salaries
  (is (= 1293000 (cc/sum-salaries ce/employees))))
```

This test case reveals a lot about the code base being tested:

- There is a namespace called `company.core`, providing more or less useful functions.
- There is a namespace called `company.employees`, providing an actual database of employee records.
- There is a function `sum-salaries` that calculates the total salaries of an employee database, returning a positive integer.

Simple unit tests can be made more powerful by using parameters with `are`:

```
(deftest test-finding-employee-by-name-parametrized
  (testing "Finding employees by their name, checking their roles"
    (are [actual expected] (= (:job actual) expected)
         (cc/find-by-name ce/employees "Dilbert") "Engineer"
         (cc/find-by-name ce/employees "Ashok") "Intern"
         (cc/find-by-name ce/employees "Dogbert") "Consultant")))
```

Here, three examples are provided, each consisting of two expressions: The left (the actual function call) being mapped to `actual`, the right (the expected result) being mapped to `expected`. The test consists of comparing a property of `actual` (the job of the employee) to the `expected` value. This helps keeping the test definition separate from the test examples, which makes it less effortful to add more test cases.

15 Spec

Clojure programmers are often more concerned with the *shape* of data rather than its *type*. The question is rather "is this a vector of maps each providing a `:name` key?" than "is this a `NamedItemsVector`?".

The shape of data can be verified by providing functions such as this:

```
(defn employee? [x]
  (and
    (map? x)
    (string? (:name x))
    (pos-int? (:age x))
    (string? (:job x))
    (pos-int? (:salary x))))

> (employee? {:name "Dilbert" :age 42 :job "Engineer" :salary 120000})
true
> (employee? {:name "Clint Eastwood" :age 82 :role "Dirty Harry"})
false
```

This manual approach doesn't scale well. Consider matching the shape of strings: Writing a state machine manually for every pattern neither scales well. Instead, *regular expressions* are used to describe those patterns. The `clojure.spec` library provides facilities to define and check the shape of data; it works like regular expressions for data structures:

```
> (require '[clojure.spec.alpha :as s])
```

`clojure.spec` is about to be finished, and therefore is used under the namespace `.alpha` for the time being.

The `s/valid?` function expects a predicate function and a value and returns whether or not the value passes the validation:

```
> (s/valid? number? 44)
true
> (s/valid? string? 44)
false
```

Multiple predicates can be combined using `clojure.spec/and`:

```
(def n-leq-100
  (s/and number? #(⩽ % 100)))

> (s/valid? n-leq-100 99)
true

> (s/valid? n-leq-100 101)
false
```

A predicate function `n-leq-100` is called a *spec*. The whole library and concept is called `clojure.spec`.

Predicates can also be combined using `clojure.spec/or`, which requires additional keywords describing the checks:

```
(def far-from-zero?
  (s/or :positive #(> % +10)
        :negative #(< % -10)))

> (s/valid? far-from-zero? 5)
false
> (s/valid? far-from-zero? 15)
true
> (s/valid? far-from-zero? -5)
false
> (s/valid? far-from-zero? -15)
true
```

The keywords `:positive` and `:negative` are useful for providing feedback in case the value fails to match the spec (more of which later).

Multiple predicates can be combined to build new predicates:

```
(def n-pos? #(⩾ % 0))
(def n-leq-100? #(⩽ % 100))
(def n-even? #(= (mod % 2) 0))
(def n-pos-even-leq-100
  (s/and
    n-pos?
    n-leq-100?
    n-even?))

> (s/valid? n-pos-even-leq-100 99)
false
> (s/valid? n-pos-even-leq-100 98)
true
```

`spec/coll-of` can be used to check if something is a collection of something:

```
(def coll-of-strings? (s/coll-of string?))

> (s/valid? coll-of-strings? ["John" "Doe"])
```

```
true
> (s/valid? coll-of-strings? ["one" "two" "three" 4 "five"])
false
```

spec/cat can be used to create *this* should follow *that* specs (descriptive keywords are needed):

```
(def s-n-s-n? (s/cat :s1 string? :n1 number? :s2 string? :n2 number?))

> (s/valid? s-n-s-n? ["Dilbert" 42 "Ashok" 21])
true
> (s/valid? s-n-s-n? ["Dilbert" "Alice" "Dogbert" "Wally"])
false
```

Specs for maps can be written using the keys function (using the employees database from the last chapter src/company/employees.clj):

```
(ns company.employees
  (:require [clojure.spec.alpha :as s]))

(def employee-s?
  (s/keys :req-un [:company.employees/name
                   :company.employees/age
                   :company.employees/job
                   :company.employees/salary]))

> (require '[clojure.spec.alpha :as s])
> (require '[company.employees :as ce])
> (s/valid? ce/employee-s? {:name "Dilbert" :age 42 :job "Engineer" :salary 120000})
true
> (s/valid? ce/employee-s? {:name "Ashok" :age 27 :job "Intern"})
false
```

Here, namespace-qualified keys (:company.employees/name) have been used. However, the -un part of :req-un means *unqualified*, so the keys of a map value don't have to be qualified in order to match.

In order to use specs in different locations, they can be stored in a *global registry* ("global" means JVM-wide) using clojure.spec/def. This registers an employee record as :company.employees/employee:

```
(s/def :company.employees/employee
  (s/keys :req-un [:company.employees/name
```

```
                    :company.employees/age
                    :company.employees/job
                    :company.employees/salary]))
```

The spec can be used globally under its global name:

```
> (s/valid? :company.employees/employee
    {:name "Dilbert" :age 42 :job "Engineer" :salary 120000})
true
```

Keywords must be fully qualified in the spec definition because of the global registry, otherwise they could collide with other keywords. However, from within the namespace `company.employees`, the shortcut `::name` can be used instead of `:company.employees.name`. Thus, the above spec can be simplified:

```
(s/def :company.employees/employee
  (s/keys :req-un [::name
                   ::age
                   ::job
                   ::salary]))
```

`clojure.spec` tries to look up the fully qualified keys in the registry. If a spec is found, the value associated with that key is validated against it. (Otherwise, no validation takes place.)

Let's create additional specs for the map keywords:

```
(s/def ::name string?)

(s/def ::age int?)

(s/def ::job string?)

(s/def ::salary int?)

(s/def ::employee
  (s/keys :req-un [::name
                   ::age
                   ::job
                   ::salary]))

(s/def ::employees (s/coll-of ::employee))
```

```
> (s/valid? :company.employees/employee
    {:name "Dilbert" :age 42 :job "Engineer" :salary 120000})
true
> (s/valid? :company.employees/employee
    {:name "Ashok" :age 27 :job "Intern" :salary "nothing"})
false
```

When using heavily nested specs, it's often unclear *why* a particular value failed to match a spec. In this case, `clojure.spec/explain` can be used (just like `valid`):

```
> (s/explain :company.employees/employee
    {:name "Ashok" :age 27 :job "Intern" :salary "nothing"})
"nothing" - failed: int? in: [:salary] at: [:salary] spec: :company.employees/salary
```

`explain` always returns `nil` and prints its result. The related function `clojure.spec/conform`, on the other side, returns the (positively) matching value, or `:clojure.spec.alpha/invalid` in case of a mismatch:

```
> (s/conform :company.employees/employee
    {:name "Dilbert" :age 42 :job "Engineer" :salary 120000})
{:name "Dilbert", :age 42, :job "Engineer", :salary 120000}
> (s/conform :company.employees/employee
    {:name "Ashok" :age 27 :job "Intern" :salary "nothing"})
:clojure.spec.alpha/invalid
```

Specs can also be used to validate the arguments of a function. One way is to use `:pre` and `:post` conditions with functions (`src/company/core.clj`):

```
(ns company.core
  (:require [company.employees])
  (:require [clojure.spec.alpha :as s]))

(defn find-by-name
  "Search for an employee by name (unique result)"
  [employees by-name]
  {:pre [(s/valid? :company.employees/employees employees)
         (s/valid? :company.employees/name by-name)]}
  (first (filter #(= (:name %1) by-name) employees)))
```

A more convenient way is to define those conditions separately from the function using `clojure.spec/fdef`:

```
(s/fdef find-by-name
        :args (:by-name :company.employees/name))
```

Those checks come with a significant performance penalty and are therefore deactivated by default. They can be activated by explicitly instrumenting a function:

```
> (require '[company.employees])
> (require '[clojure.spec.alpha :as s])
> (require '[clojure.spec.test.alpha :as st])

> (st/instrument 'company.core/find-by-name)
> (find-by-name company.employees/employees "Dilbert")
{:name "Dilbert", :age 42, :job "Engineer", :salary 120000}

> (find-by-name company.employees/employees :dilbert)
Execution error - invalid arguments to company.core/find-by-name at ...
:dilbert - failed: string? at: [:by-name] spec: :company.employees/name
```

This should only be used during development and testing.

Creating specs provides much information that can be used for generating test data. Consider the function `introduce` (`src/company/employees.clj`) and its spec:

```
(defn introduce [employee]
  (str "Hello, my name is "
       (:name employee)
       ", I'm "
       (:age employee)
       " years old."))

(s/fdef introduce :args (s/cat :employee :company.employees/employee))

> (st/instrument 'company.core/introduce)
> (introduce (find-by-name company.employees/employees "Dilbert"))
"Hello, my name is Dilbert, I'm 42 years old."
```

Using the `:ret` keyword, the return value of the function can be checked if it contains the static portion of the text:

```
(s/fdef introduce
        :args (s/cat :employee :company.employees/employee)
        :ret (s/and string?
```

```
                    (partial re-find #"Hello, my name is ")
                    (partial re-find #"I'm ")
                    (partial re-find #" years old.")))
```

The function must be instrumented for testing:

```
> (require '[clojure.spec.test.alpha :as stest])
> (stest/check 'company.core/introduce)
({:spec #object[clojure.spec.alpha$fspec_impl$reify__2524 0x58782ed6
 "clojure.spec.alpha$fspec_impl$reify__2524@58782ed6"],
 :clojure.spec.test.check/ret {:result true, :pass? true, :num-tests 1000,
                               :time-elapsed-ms 431, :seed 1622747931886},
 :sym company.core/introduce})
```

The `:fn` keyword can be used to provide a function for performing additional checks. The `employee-exists` function gets both the arguments (`args`) and the return value (`ret`) of the instrumented function as arguments. The employee's name is extracted, and it is checked, if that name is contained in the return value:

```
(defn employee-exists [{:keys [args ret]}]
  (let [employee (-> args :employee :name)]
    (not (neg? (.indexOf ret employee)))))

(s/fdef introduce
        :args (s/cat :employee :company.employees/employee)
        :ret (s/and string?
                    (partial re-find #"Hello, my name is ")
                    (partial re-find #"I'm ")
                    (partial re-find #" years old."))
        :fn employee-exists)
```

When dealing with keyword specs, double-check that there are no typos. Misnamed keywords won't be validated by a spec.

16 Interoperating with Java

Since Clojure is based on the Java Virtual Machine, Java code can be used directly from Clojure using its interoperation facilities (*interop*). The Clojure REPL is also a great tool to explore Java APIs.

Java offers the class `java.io.File`, which abstracts the concept of a file. The JavaDoc shows that there is a constructor expecting a `pathname`. A file can, thus, created as follows (note the additional dot after `java.io.File`).

```
> (def employees-file (java.io.File. "employees.txt"))
```

Methods of the created `File` instance can be called with a dot in front of the method name:

```
> (.exists employees-file)
false
> (.getAbsolutePath employees-file)
"/home/patrick/employees.txt"
```

Some classes, such as `java.awt.Rectangle`, offer public fields, which can be accessed by prepending a dot and a minus:

```
> (def rect (java.awt.Rectangle. 0 0 15 25))
> (.-width rect)
15
> (.-height rect)
25
```

When refering to a class repeatedly, typing out the fully qualified class name becomes tedious. Therefore, classes can be imported. Use `import` from the REPL:

```
> (import java.io.File)
```

Or `:import` of `ns` within a `.clj` source file:

```
(:ns company.core
  (:import java.io.File))
```

Once imported, `File` can be used as follows:

```
(def backup (File. "backup.txt"))
```

To import multiple classes from the same package, just put them into a list separated by space (which must be quoted in the REPL):

```
> (import '(java.io File InputStream))
```

And without quotation from a `.clj` source file:

```
(:ns company.core
  (:import (java.io File InputStream))
```

Classes from the package `java.lang` are automatically imported.

Static fields and methods can be accessed using a forward slash:

```
> (def dump (File. (str "data" File/separator "dump.txt")))
> (.getAbsolutePath dump)
"/home/patrick/data/dump.txt"

> (def temp (File/createTempFile "employees" ".txt"))
> (.getAbsolutePath temp)
"/tmp/employees6472489788961466629.txt"
```

Java libraries can be used just like Clojure libraries. Let's import the Gson library for reading and writing JSON to the `company` project (`project.clj`):

```
:dependencies [[org.clojure/clojure "1.10.1"]
               [org.clojure/test.check "1.1.0"]
               [com.google.code.gson/gson "2.8.0"]] ; new import
```

The library can be used—and explored—from the REPL:

```
$ lein repl
```

First, the `Gson` class needs to be imported (here, tab completion comes in handy to figure out the package structure):

```
> (import com.google.gson.Gson)
```

Second, a `Gson` object needs to be initialized:

```
> (def gson-obj  (Gson.))
```

Now Clojure values can be turned into JSON strings:

```
> (.toJson gson-obj 42)
"42"

> (.toJson gson-obj [7 14 21 28])
"[7,14,21,28]"

> (def employees [{:name "Dilbert" :age 42 :job "Engineer" :salary 120000}
                  {:name "Alice" :age 37 :job "Engineer" :salary 115000}
                  {:name "Wally" :age 47 :job "Engineer" :salary 130000}])
> (.toJson gson-obj employees)
"[{\":name\":\"Dilbert\",\":age\":42,\":job\":\"Engineer\",\":salary\":120000},
  {\":name\":\"Alice\",\":age\":37,\":job\":\"Engineer\",\":salary\":115000},
  {\":name\":\"Wally\",\":age\":47,\":job\":\"Engineer\",\":salary\":130000}]"
```

Even though interoperability between Clojure and Java works almost seamlessly, a few differences have to be considered.

First, a Java method is not a function, and, thus, cannot be bound like a Clojure functon to a symbol:

```
> (.count [1 2 3])
3
> (def count-method .count)
Syntax error compiling at (/tmp/form-init2910488673606898294.clj:1:1).
Unable to resolve symbol: .count in this context
```

However, it is possible to turn a method into a function using `memfn`:

```
> (def count-method (memfn count))
> (count-method [1 2 3])
3
```

This is helpful when dealing with higher-order functions, such as `map`:

```
> (def files [(File. "source.txt") (File. "target.txt")])
> (map (memfn exists) files)
(false false)
```

Second, many Java objects are mutable, which might be surprising after dealing with Clojure's immutable collections:

```
> (import java.util.Vector)
> (def employees (java.util.Vector.))
> (.addElement employees "Dilbert")
> (.addElement employees "Alice")
> (.addElement employees "Wally")
> employees
["Dilbert" "Alice" "Wally"]
```

Rather than returning a vector with the element added, `nil` is returned, and the element added as a side effect. Use Clojure's collections instead, unless mutability is needed.

17 Threads, Promises, and Futures

Threads are a very powerful, but also dangerous tool, especially when used in the context of mutable values. At runtime, a Clojure program is a Java program, and every Java program runs a single main thread. Additional threads can be started by using Java's `Thread` class:

```
> (defn say-hello [] (println "Hello!"))
> (def thread (Thread. say-hello))
> (.start thread)
Hello!
```

A Clojure function `say-hello` is defined, and a thread is created based on that function (which is a `Runnable`). Once the thread is started, the function is executed.

The effect of multiple threads running can be shown by putting one thread to sleep for a while:

```
> (defn say-hello []
    (println "Hello, once!")
    (Thread/sleep 1000)
    (println "Hello, again!"))
> (def thread (Thread. say-hello))
> (.start thread)
Hello, once!
;; waiting for a second, the REPL is not blocking...
Hello, again!
```

When multiple threads are running, program execution gets non-deterministic, unless control mechanisms are put in place. Let's consider these two functions working on a shared variable:

```
(def employee-of-the-month "Dilbert")
(defn make-alice-eom []
  (def employee-of-the-month "Alice"))
(defn make-wally-eom []
  (def employee-of-the-month "Wally"))
```

The behaviour is completely deterministic when the functions are executed one after another:

```
> (make-alice-eom)
> (make-wally-eom)
employee-of-the-month
"Wally"
```

However, when the functions are executed in separate threads, the result depends on mere chance:

```
(defn race-condition []
    (let [thread-alice (Thread. make-alice-eom)
          thread-wally (Thread. make-wally-eom)]
      (.start thread-alice)
      (.start thread-wally)))

> (race-condition)
> employee-of-the-month
"Wally"
```

Even though the thread started later will usually finish later in this example, no such guarantee can be given. Not modifying shared variables and using immutable data structures helps to avoid such race conditions. Notice that dynamic vars live in thread local storage, and, thus, can be used safely from different threads.

Some threads do their work in the background, but we need to make sure that they can finish their work before the main thread finishes using `join`:

```
(defn delete-cache []
  (.delete (java.io.File. "/tmp/cache.txt")))

> (def delete-thread (Thread. delete-cache))
> (.start delete-thread)
> (.join delete-thread)
```

`join` waits until the thread is finished and then returns `nil`.

It is often useful to get back a result from a computation performed in a thread. A *promise* is some kind of a value trap that will deliver a value when demanded. A promise is created using the `promise` function:

```
> (def the-result (promise))
```

A value is delivered using the `deliver` function:

```
> (deliver the-result "Dilbert")
```

The value then can be grabbed using `deref` or using `@`:

```
> (println "The result is:" (deref the-result))
The result is: Dilbert
> (println "The result is:" @the-result)
The result is: Dilbert
```

Once set, the value of a promise can't be changed again; the value trap was shut!

Promises are useful in the context of multiple threads. Consider this employee database, to which two functions shall be applied concurrently:

```
(def employees [{:name "Dilbert" :age 42 :salary 120000}
                {:name "Ashok"   :age 27 :salary  18000}
                {:name "Topper"  :age 37 :salary 250000}])

(defn average-age [employees]
  (float (/ (reduce + (map :age employees)) (count employees))))

(defn average-salary [employees]
  (float (/ (reduce + (map :salary employees)) (count employees))))
```

The two calculations can be executed in separate threads and deliver their results using a promise. The parallel processing (given multiple CPUs) might come in handy as the database grows:

```
(defn perform-calculations [employees]
  (let [age-prom (promise)
        pay-prom (promise)]
    (.start (Thread. #(deliver age-prom (average-age employees))))
    (.start (Thread. #(deliver pay-prom (average-salary employees))))
```

```
    (println "Average age:" @age-prom)
    (println "Average salary:" @pay-prom)))

> (perform-calculations employees)
Average age: 35.333332
Average salary: 129333.336
```

No `join` is needed, since the threads are done after delivering their values.

This whole process can be simplified by using a *future*, which is a promise that brings its own thread along:

```
(defn perform-calculations [employees]
  (let [age-future (future (average-age employees))
        pay-future (future (average-salary employees))]
    (println "Average age:" @age-future)
    (println "Average salary:" @pay-future)))

> (perform-calculations employees)
Average age: 35.333332
Average salary: 129333.336
```

A function call wrapped using `future` will be executed in its own thread and deliver its return value to be grabbed using `deref` or `@`.

In general, prefer futures over promises, because they don't require dealign with threads directly. For more fine-grained control, consider using Java's thread-pool facilities (`java.util.concurrent.Executors`).

In practice, it is often a sensible approach to provide a timeout (here: 500 milliseconds) and a fallback value (here: the keyword `:timeout` with a promise):

```
(deref calc-prom 500 :timeout)
```

`pmap` is a function that, from the outside, works like `map`, but, on the inside, uses multiple threads to process the elements in parallel (hence the name; *parallel* map). Parallel processing comes with some performance overhead and should only be used if it brings a net performance win.

18 State

The less mutable state a program has, the easier it is to understand. However, modeling things that do change over time requires state.

Consider this employee database with a `hire` to add new employees to it:

```
(def employees [{:name "Dilbert" :job "Engineer" :salary 120000}
                {:name "Alice" :job "Engineer" :salary 110000}
                {:name "Dogbert" :job "Consultant" :salary 250000}])

(defn hire [employee]
  (conj employees employee))

> (hire {:name "Wally" :job "Engineer" :salary 130000})
[{:name "Dilbert", :job "Engineer", :salary 120000}
 {:name "Alice", :job "Engineer", :salary 110000}
 {:name "Dogbert", :job "Consultant", :salary 250000}
 {:name "Wally", :job "Engineer", :salary 130000}]

> employees
[{:name "Dilbert", :job "Engineer", :salary 120000}
 {:name "Alice", :job "Engineer", :salary 110000}
 {:name "Dogbert", :job "Consultant", :salary 250000}]
```

The function produces a new list with the additional employee, but the original list of employees remains the same.

The state of the `employees` vector *could* be changed using `def` (notice the exclamation mark in `hire!` to denote the side-effect):

```
(defn hire! [employee]
  (def employees (conj employees employee)))

> (hire! {:name "Wally" :job "Engineer" :salary 130000})
> employees
[{:name "Dilbert", :job "Engineer", :salary 120000}
 {:name "Alice", :job "Engineer", :salary 110000}
 {:name "Dogbert", :job "Consultant", :salary 250000}
 {:name "Wally", :job "Engineer", :salary 130000}]
```

The `employees` vector was updated, however, *not in a thread-safe manner* (see previous chapter).

Thread-safe state change can be achieved by using *atoms*, which wrap the value to be changed:

```
(def employees
  (atom [{:name "Dilbert" :job "Engineer" :salary 120000}
         {:name "Alice" :job "Engineer" :salary 110000}
         {:name "Dogbert" :job "Consultant" :salary 250000}]))
```

```
(defn hire! [employee]
  (swap! employees #(conj % employee)))
```

The `atom` function wraps the `employees` vector, which then can be updated in a thread-safe manner using `swap!`, which takes two arguments: first, the atom to be updated (`employees`), second, a function to be applied to produce the new value to be stored in the atom. Note that `employees` is no longer a vector, but a vector wrapped in an atom.

The employees database can now be updated thread-safely in place:

```
> (hire! {:name "Wally" :job "Engineer" :salary 130000})
```

To access the value wrapped in the atom, use `deref` or `@` (as with promises and futures):

```
> (deref employees)
[{:name "Dilbert", :job "Engineer", :salary 120000}
 {:name "Alice", :job "Engineer", :salary 110000}
 {:name "Dogbert", :job "Consultant", :salary 250000}
 {:name "Wally", :job "Engineer", :salary 130000}] ; new entry

> @employees
[{:name "Dilbert", :job "Engineer", :salary 120000}
 {:name "Alice", :job "Engineer", :salary 110000}
 {:name "Dogbert", :job "Consultant", :salary 250000}
 {:name "Wally", :job "Engineer", :salary 130000}] ; new entry
```

Any value can be wrapped in an atom, consider this counter:

```
(def counter (atom 0))

(defn increase-counter! [amount]
  (swap! counter + amount))
```

The `swap!` function applies the `+` function to the `counter` atom. The `amount` value is handed over to the `+` function by `swap!`:

```
> (increase-counter! 1)
> @counter
1
> (increase-counter! 5)
> @counter
6
```

When an atom is updated using `swap!`, it performs the following steps to guarantee thread safety:

1. The current value of the atom is read.
2. The update function is called to produce the new value.
3. The current value of the atom is read *again*, and compared to the value read previously.
    - If the value *did not change* in the meantime, the value is updated.
    - If the value *did change* in the meantime, the whole process is repeated from the first step.

Note that the update function can be called multiple times for a single update! Therefore it's important, that the update function has no side effects!

Sometimes, there are multiple values that need to be synchronized. Consider this empty employee database, for which also the total number of employees, and the total salary needs to be kept track of:

```
(def employees (atom []))

(def total-payroll (atom 0))

(def total-staff (atom 0))
```

The `hire!` function tries to keep track of those three atoms:

```
(defn hire! [employee]
  (swap! employees #(conj % employee))
  (swap! total-payroll #(+ (:salary employee)))
  (swap! total-staff inc))

> (hire! {:name "Dogbert" :salary 250000})
> (hire! {:name "Dilbert" :salary 120000})

> @employees
[{:name "Dogbert", :salary 250000}
 {:name "Dilbert", :salary 120000}]
> @total-payroll
370000
> @total-staff
2
```

This seems to work fine, but the atoms are out of sync *between* the calls to `swap!`:

```
(defn hire! [employee]
  (swap! employees #(conj % employee))
  ; out of sync
  (swap! total-payroll #(+ (:salary employee)))
  ; out of sync
  (swap! total-staff inc))
```

The three values must be updated either *all together* or *not at all*, like an *atomic* database transaction.

Such groups of atoms can be managed as *refs*, which are a lot like atoms, but use different functions. First, `ref` is used instead of `atom` for the wrapping:

```
(def employees (ref []))

(def total-payroll (ref 0))

(def total-staff (ref 0))
```

Second, `alter` is used instead of `swap!`. And, third, all the updates belonging to the same transaction are grouped together using `dosync`:

```
(defn hire! [employee]
  (dosync
    (alter employees #(conj % employee))
    (alter total-payroll #(+ % (:salary employee)))
    (alter total-staff inc)))
```

Now all the values are kept perfectly in sync. When read, all the three values being altered within the same call to `dosync` will either yield all the old or all the new values:

```
> (hire! {:name "Catbert" :salary 180000})
> (hire! {:name "Alice" :salary 120000})
> @employees
[{:name "Catbert", :salary 180000}
 {:name "Alice", :salary 120000}]
> @total-payroll
300000
> @total-staff
2
```

If the old value of a `ref` is not of interest, use `ref-set` instead of `alter`, providing just the new value.

Sometimes, modifications to values should be accompanied by some side effects, say, writing changes to a file when new items are added. Consider the function `notify-new-hire` together with the old version of `hire!` working on a single atom. Every time a new employee is hired, `notify-new-hire` is called:

```
(def employees (atom []))

(defn notify-new-hire [employee]
  (println "Watch out for" (:name employee)))

(defn hire! [employee]
  (notify-new-hire employee)
  (swap! employees #(conj % employee)))

> (hire! {:name "Alice" :salary 120000})
Watch out for Alice
```

This works fine—until an update has to be retried because the values have been modified in between. In this case, `notify-new-hire` would be executed multiple times, which is not wanted.

An *agent* is an atom that can be combined with side-effects. Instead of calling `swap!`, the function `send` is used to both update the agent, and to produce the side effect:

```
(def employees (agent []))

(defn notify-new-hire [employee]
  (println "Watch out for" (:name employee)))

(defn hire! [employee]
  (send employees
        (fn [old-employees]
          (notify-new-hire employee)
          (conj old-employees employee))))
```

Notice that an anonymous function (`fn`) has been used instead of a lambda expression. Every agent has its own queue of functions. When `hire!` gets called, the call to the anonymous function gets queued up. `send` works asynchronously, i.e. it returns immediately after the function was put into the queue. The agent pops an outstanding function call from the queue and executes it. The side effect (calling `notify-new-hire`) and the update are then performed:

```
> (hire! {:name "Ratbert" :salary 0})
Watch out for Ratbert
> (hire! {:name "Alice" :salary 115000})
Watch out for Alice
> @employees
[{:name "Ratbert", :salary 0}
 {:name "Alice", :salary 115000}]
```

Since agents perform their updates in their own thread, exceptions caused by a failed update are not reported immediately. Consider this agent for tracking donations:

```
(def donations (agent 0))

(defn praise-donor [amount donor]
  (println "Praise" donor "for donating" amount "coins!"))

(defn donate! [amount donor]
  (send donations
        (fn [old-donations]
          (praise-donor amount donor)
          (+ old-donations amount))))

> (donate! 100 "John")
Praise John for donating 100 coins!
> @donations
0
```

If the arguments `amount` (integer) and `donor` (string) are swapped, the operation cannot be completed:

```
> (donate! "Jane" 200)
Praise 200 for donating Jane coins!
> @donations
100
```

The message from the side effect looks suspicious, and the `donations` haven't been increased. But the error can go unnoticed until the next update is done:

```
> (donate! 300 "Jim")
Execution error (ClassCastException) at user/donate!$fn (REPL:5).
java.lang.String cannot be cast to java.lang.Number
```

In this case, the agent has to be re-started using `agent-restart`. The error condition can be detected using `agent-error`:

```
(if (agent-error donations)
  (restart-agent donations 0 :clear-actions true))
```

However, the value managed by the agent is reset to 0 in the process.

Make sure to call `shutdown-agents` at the end of the `-main` function of your program, so that the agent's threads get properly terminated:

```
(defn -main []
  ;; working with agents
  (shutdown-agents))
```

Use the following guidelines to pick the proper mechanism for your state changes:

1. If a value remains *mostly stable*, put it into a `var`.
2. If a number of values need to be updated together without side effects, use `refs`.
3. If the update of values is to be accompanied by side effects, or if the update function takes a lot of time, use an `agent`.
4. If you have a single value that changes without additional side effects, us an `atom`.

Instead of using `refs`, the values to be updated could also be grouped into a single data structure, which is then wrapped in an `atom`:

```
(def payroll (atom {:total-staff 0
                    :total-payroll 0
                    :employees []}))

(defn hire! [employee]
  (swap! payroll
         (fn [old]
           (assoc old
                  :total-staff (inc (:total-staff old))
                  :total-payroll (+ (:total-payroll old) (:salary employee))
                  :employees (conj (:employees old) employee)))))

> (hire! {:name "Alice" :salary 115000})
> (hire! {:name "Dogbert" :salary 250000})
> @payroll
{:total-staff 2,
 :total-payroll 365000,
 :employees [{:name "Alice", :salary 115000}
             {:name "Dogbert", :salary 250000}]}
```

There are various functions in Clojure making use of atoms, e.g. `memoize`, which wraps a function with a cache that maps the arguments to the computed return values. Consider this Fibonacci function:

```
(defn fib [n]
  (println "Computing Fibonacci number for" n)
  (cond
    (= n 0) 1
    (= n 1) 1
    (> n 1) (+ (fib (- n 1)) (fib (- n 2)))
    :else (throw (ex-info "fib(n) only defined for n ⩾ 0"))))

> (fib 1)
Computing Fibonacci number for 1
1

> (fib 2)
Computing Fibonacci number for 2
Computing Fibonacci number for 1
Computing Fibonacci number for 0
2

> (fib 3)
Computing Fibonacci number for 4
Computing Fibonacci number for 3
Computing Fibonacci number for 2
Computing Fibonacci number for 1
Computing Fibonacci number for 0
Computing Fibonacci number for 1
Computing Fibonacci number for 2
Computing Fibonacci number for 1
Computing Fibonacci number for 0
5
```

The function is called with the same argument multiple times, which slows down the computation for bigger `n`.

`memoize` wraps the function so that it caches its return values by argument:

```
> (def fib (memoize fib))
> (fib 1)
Computing Fibonacci number for 1
1
```

```
> (fib 2)
Computing Fibonacci number for 2
Computing Fibonacci number for 0
2
> (fib 3)
Computing Fibonacci number for 3
3
> (fib 4)
Computing Fibonacci number for 4
5
> (fib 10)
Computing Fibonacci number for 10
Computing Fibonacci number for 9
Computing Fibonacci number for 8
Computing Fibonacci number for 7
Computing Fibonacci number for 6
Computing Fibonacci number for 5
89
```

The function returned by `memoize` has its own atom, which serves as a cache for the results having been computed.

19 Read and Eval

Clojure's syntax might look strange on the first sight, but is crucial for how the language works. There are two critical functions in Clojure—`read` and `eval`—that relate to Clojure's syntax.

Clojure code looks a lot like data literals:

```
'(dilbert pointy-haired-boss [alice]
   (wally ratbert
     (dogbert "some characters from dilbert...")))
```

Just replace the Dilbert character names by some other symbols, and almost have Clojure code:

```
'(defn say-hello [friendly]
   (if friendly
     (println "Hello, my dear!")))
```

Just remove the quote, and you have an executable Clojure function.

```
(defn say-hello [friendly]
  (if friendly
    (println "Hello, my dear!")))

> (say-hello true)
Hello, my dear!
```

Clojure code *is* just Clojure data, and Clojure function calls *are* lists. Clojure is *homoiconic*, which means, code and data are the same thing.

The `read` function reads data, by default from the REPL (`stdin`), until `[Return]` is entered:

```
> (read)
1
1
> (read)
"hello"
"hello"
> (read)
(defn say-hi []
  (println "Hi!"))
(defn say-hi [] (println "Hi!"))
```

The `read` function just returns returns the expression it read.

`read-string` reads a string and turns it into a Clojure value (notice the escaped double quotes around the text `Hi`):

```
> (read-string "(defn say-hi [] (println \"Hi!\"))"
(defn say-hi [] (println "Hi!"))
```

Once data has been read (from whatever source), it can be evaluated as Clojure code using the `eval` function:

```
> (def some-function-call '(+ 2 3))
> (eval some-function-call)
5
```

The code in the quoted list `(+ 2 3)` is compiled and run as Clojure code.

Some data types, like numbers, strings, and keywords, just evaluate to themselves:

```
> (eval 42)
42
> (eval "Dilbert")
"Dilbert"
> (eval :salary)
:salary
```

A Clojure function consists of different data structures—lists, vectors—which can all be made up as data, and then be combined to a list, which makes up an actual function that can be evaluated:

```
> (def function-name 'say-hi)
> (def args (vector 'to-whom))
> (def output (list 'println "Hi," 'to-whom))
> (def whole-function (list 'defn function-name args output))

> (eval whole-function) ; creates function say-hi
> (say-hi "Joe")
Hi, Joe
```

The two functions `read` and `eval` combined can be used to read Clojure code from an external source, say, user-specific settings from a config file. No extra language or syntax has to be made up: The full power of Clojure is supported out of the box.

It's also possible to define a very simple REPL using `read` and `eval`:

```
(defn my-repl []
  (loop []
    (println (eval (read)))
    (recur)))

> (my-repl)
(println "Hello")
Hello
(println (+ 3 2))
5
; Hit Ctrl-D to finish
```

A toy version of the function `eval` can be implemented as an ordinary Clojure function:

```
(defn my-eval [expr]
  (cond
```

```
    (string? expr) expr
    (keyword? expr) expr
    (number? expr) expr
    (symbol? expr) (my-eval-symbol expr)
    (vector? expr) (my-eval-vector expr)
    (list? expr) (my-eval-list expr)
    :else :unknown-expression))
```

Strings, keywords, and numbers just evaluate to themselves, so the `expr` is just returned.

Symbols need to be looked up in the current environment:

```
(defn my-eval-symbol [expr]
  (.get (ns-resolve *ns* expr)))
```

Vectors needs to be processed recursively:

```
(defn my-eval-vector [expr]
  (vec (map my-eval expr)))
```

And lists, which can be function calls, need to be separated into the function name `f` and the argument list `args`, which are then applied using `apply`:

```
(defn my-eval-list [expr]
  (let [evaled-items (map my-eval expr)
        f (first evaled-items)
        args (rest evaled-items)]
    (apply f args)))

> (my-eval '(println "Hello"))
Hello
```

A real `eval` function *compiles* the given expression to Clojure code, which then can be executed a lot faster. Therefore, `eval` is not to be used as an everyday tool, but only for special cases: it is just *too powerful* and therefore *too dangerous*. It also is slower than regular Clojure code, due to the additional compilation step.

Reading in code from arbitrary sources can also be very dangerous. Use the `read` function from `clojure.edn` if you don't trust the source. But `read` is also not a tool for everyday use, so only use it if really needed.

20 Macros

Code is both useful and painful: it solves problems, but also creates new problems of its own. Writing more *expressive* code leads to *less* code. The usefulnes of code is kept, but the pain is reduced; *less code, less pain*.

Macros are a powerful tool in LISP-like languages (such as Clojure) to automate some part of code writing.

Consider a rating system in which numbers are interpreted in three categories:

- positive numbers: good
- zero: indifferent
- negative numbers: bad

This rating system could be used to print a depiction of the rating in English:

```
(defn print-rating [rating]
  (cond
    (pos? rating) (println "good")
    (zero? rating) (println "indifferent")
    :else (println "bad")))

> (print-rating 3)
good
> (print-rating 0)
indifferent
> (print-rating -5)
bad
```

Another implementation could be required to turn the rating number into a keyword for further programmatic processing:

```
(defn evaluate-rating [rating]
  (cond
    (pos? rating) :good
    (zero? rating) :indifferent
    :else :bad))

> (evaluate-rating 3)
:good
> (evaluate-rating 0)
:indifferent
> (evaluate-rating -1)
:bad
```

Structurally, the two usages of `cond` are identical, they just have different consequences. The commonalities of the two functions could be factored out into a new function, `arithmetic-if`:

```
(defn arithmetic-if [n pos zero neg]
  (cond
    (pos? n) pos
    (zero? n) zero
    (neg? n) neg))
```

The function accepts both the number `n` to be categorized, and the three possible *consequences* of a match: `pos`, `zero`, and `neg`.

This works great, if a value has to be returned, as in `evaluate-rating`, which can be refactored in terms of `arithmetic-if`:

```
(defn evaluate-rating [rating]
  (arithmetic-if rating :good :indifferent :bad))

> (evaluate-rating 5)
:good
> (evaluate-rating 0)
:indifferent
> (evaluate-rating -7)
:bad
```

The same refactoring applied to `print-rating`, however, produces surprising results:

```
(defn print-rating [rating]
  (arithmetic-if rating
                 (println "good")
                 (println "indifferent")
                 (println "bad")))

> (print-rating 4)
good
indifferent
bad
> (print-rating 0)
good
indifferent
bad
> (print-rating -2)
```

```
good
indifferent
bad
```

All three `println` function calls are executed! When `arithmetic-if` is invoked, the arguments get evaluated. This is unproblematic for `n`, which is a number and, therefore, evaluates to itself. Function calls like `println`, however, are evaluated by their actual execution.

If `arithmetic-if` expected functions as its last three parameters, `print-rating` could be implemented based on the former:

```
(defn arithmetic-if [n pos-f zero-f neg-f]
  (cond
    (pos? n) (pos-f)
    (zero? n) (zero-f)
    (neg? n) (neg-f)))

(defn print-rating [rating]
  (arithmetic-if rating
                 #(println "good")
                 #(println "indifferent")
                 #(println "bad")))

> (print-rating 4)
good
> (print-rating 0)
indifferent
> (print-rating -2)
bad
```

However, the implementation of `evaluate-rating` now becomes more complicated, thwarting the gains made using `arithmetic-if` as an abstraction:

```
(defn evaluate-rating [rating]
  (arithmetic-if rating
                 #(identity :good)
                 #(identity :indifferent)
                 #(identity :bad)))

> (evaluate-rating 7)
:good
> (evaluate-rating 0)
:indifferent
```

```
> (evaluate-rating -5)
:bad
```

What `arithmetic-if` is really supposed to do is to transform code written like this:

```
;; evaluate-rating
(arithmetic-if rating
  :good
  :indifferent
  :bad)

;; print-rating
(arithmetic-if rating
  (println "good")
  (println "indifferent")
  (println "bad"))
```

Into code being executed like this:

```
;; evaluate-rating
(cond
  (pos? rating) :good
  (zero? rating) :indifferent
  :else :bad)

;; print-rating
(cond
  (pos? rating) (println "good")
  (zero? rating) (println "indifferent")
  :else (println "bad"))
```

Since Clojure code *is* just data, this transformation can be made using a function building up another function:

```
(defn arithmetic-if-to-cond [n pos zero neg]
  (list 'cond (list 'pos? n) pos
              (list 'zero? n) zero
              :else neg))
```

Fed with parameters protected with a single quote from evaluation, this function produces the desired `cond` forms:

```
> (arithmetic-if-to-cond 'rating
                         '(println "good")
                         '(println "indifferent")
                         '(println "bad"))
(cond
  (pos? rating) (println "good")
  (zero? rating) (println "indifferent")
  :else (println "bad"))

> (arithmetic-if-to-cond 'rating
                         ':good
                         ':indifferent
                         ':bad)
(cond
  (pos? rating) :good
  (zero? rating) :indifferent
  :else :bad)
```

However, those `cond` forms are *still just data*, and not compiled code that actually can be used. Here, macros come into play, which are defined using `defmacro`:

```
(defmacro arithmetic-if [n pos zero neg]
  (list 'cond (list 'pos? n) pos
              (list 'zero? n) zero
              :else neg))
```

`print-rating` can now be implemented without using lambdas or quotation:

```
(defn print-rating [rating]
  (arithmetic-if rating
                 (println "good")
                 (println "indifferent")
                 (println "bad")))

> (print-rating 4)
good
> (print-rating 0)
indifferent
> (print-rating -2)
bad
```

Which is also the case for `evaluate-rating`:

```
(defn evaluate-rating [rating]
  (arithmetic-if rating
                 :good
                 :indifferent
                 :bad))

> (evaluate-rating 7)
:good
> (evaluate-rating 0)
:indifferent
> (evaluate-rating -5)
:bad
```

The Clojure compilation process works as follows: First, source code is read, i.e. turned into data structures (lists, vectors, etc.). Second, *macro expansion* is performed, modifying those data structures. Third, those modified data structures with expanded macros are turned into byte code by the actual compilation step.

Unlike C, Clojure macros work on the code as a *data structures*, not on code as mere *program text*, which makes Clojure macros more powerful and *less* dangerous than C macros.

The `arithmetic-if` macro from above requires a lot of quotes to prevent the expressions from being evaluated. Clojure provides a templating system called *syntax quoting*, which makes macros more readable:

```
(defmacro arithmetic-if [n pos zero neg]
  `(cond
     (pos? ~n) ~pos
     (zero? ~n) ~zero
     :else ~neg))
```

Syntax quoting starts with a backquote (before `cond`). Within the quoted form, expressions from the outside are referred to using a tilde prefix, which prevents them from being evalauted when the macro is expanded before compilation.

There are a few other syntax specialities when it comes to using macros with syntax quoting. Consider the `conjunction` macro, which works like `and`—"conjunction" just being a fancy word for "and":

```
(defmacro conjunction
  ([] true)
  ([x] x)
  ([x & next]
   `(let [current# ~x]
      (if current# (conjunction ~@next) current#))))
```

The macro works as follows:

1. For an empty list of conditions, `true` is returned (first base case).
2. For a single condition, the evaluated condition is returned (second base case).
3. For a list of more than one condition (general case), the following logic is applied:
    1. The first condition is bound to `current#`; the suffix `#` being used to guarantee a unique symbol.
    2. If the `current#` condition evaluates to `true`, the `conjunction` macro is "called" recursively with the next condition in the list. This `next` symbol is prefixed both by `~` for syntax quoting, and an `@`—more of which later.
    3. Otherwise, if `current#` evaluates to `false`, `current#` itself is returned, which terminates the recursive process.

In order to understand the `@` prefix, consider this alternative implementation of `defn` as a macro called `my-defn`:

```
(defmacro my-defn [name args & body]
  `(def ~name (fn ~args ~body)))
```

A function to add two numbers is created using `my-defn`:

```
(my-defn add-two-numbers [a b] (+ a b))
```

Unfortunately, the function won't work:

```
> (add-two-numbers 3 4)
Execution error (ClassCastException) at user/add-two-numbers (REPL:1).
java.lang.Long cannot be cast to clojure.lang.IFn
```

Let's see what code the macro actually generated using `macroexpand-1`, the first tool to be grabbed if macros not behave as intended:

```
> (macroexpand-1 '(my-defn add-two-numbers [a b] (+ a b)))
(def add-two-numbers (clojure.core/fn [a b] ((+ a b))))
```

Notice the `((+ a b))` expression being wrapped in two sets of parentheses. Expression was created by `~body` in the macro template, which is itself a collection because it was declared as a variadic parameter (`& body`). Thus, `body` is a list of one element containing another list: `((+ a b))`.

The `@` prefix makes sure that the collection is expanded before being written into the code. Here's a working version of `my-defn` with this expansion:

```
(defmacro my-defn [name args & body]
  `(def ~name (fn ~args ~@body)))
```

This creates the working code as intended:

```
> (macroexpand-1 '(my-defn add-two-numbers [a b] (+ a b)))
(def add-two-numbers (clojure.core/fn [a b] (+ a b)))

> (my-defn add-two-numbers [a b] (+ a b))
> (add-two-numbers 1 2)
3
```

Notice that macros are processed in a two-step process: First, they are *expanded*, second, the generated code is *evaluated*:

```
(defmacro two-step-process []
  (println "This code is run upon macro expansion.")
  `(fn [] (println "This code is run with the generated code.")))

> (def generated-code (two-step-process))
This code is run upon macro expansion.

> (generated-code)
This code is run with the generated code.
```

Also notice that macros do not exist at runtime, so their names can't be found in a stack trace. Macros also can't be used like a function given to a higher-order function such as `filter` or `map`. Only use macros if the code to be expressed is at odds with Clojure's evaluation rules. Stick to functions otherwise.