Idiomatic Clojure
A Beginner's Tour of `clojure.core`

Ákos Kiss *Programmer*

June 30, 2016

Grabow & Kiss Software GmbH

Introduction

Preface

"People seldom improve when they have no other model but themselves to copy after."

Oliver Goldsmith

- Examples presented are subjective.
- No claims are made on universality: exceptions prove the rule, yada yada yada...
- Most of the examples aim to reduce nesting of expressions, arguably aiding readability.
- Further improvements or alternative approaches are welcome!

"Example has more followers than reason."

Christian Nestell Bovee

Conditions

do & if

Bad:

```
(if (...)
  (do
    ...))
```

do & if

Bad:

```
(if (...)
  (do
    ...))
```

Better:

```
(when (...)
  ...)
```

do & if

Bad:

```
(if (...)
  (do
    ...))
```

Better:

```
(when (...)
  ...)
```

Fine:

```
(if (...)
  (do
    ...)
  (do
    ...))
```

Say no to `not`

Bad:

```
(not (= ...))
```

Say no to `not`

Bad:

```
(not (= ...))
```

Better:

```
(not= ...)
```

Say no to `not`

Bad:

```
(not (= ...))
```

Better:

```
(not= ...)
```

Bad:

```
(if (not (...))
  ...)
```

Say no to `not`

Bad:

```
(not (= ...))
```

Better:

```
(not= ...)
```

Bad:

```
(if (not (...))
  ...)
```

Better:

```
(if-not (...)
  ...)
```

Say no to `not`

Bad:

```
(not (= ...))
```

Better:

```
(not= ...)
```

Bad:

```
(if (not (...))
  ...)
```

Better:

```
(if-not (...)
  ...)
```

Bad:

```
(when (not (...))
  ...)
```

Say no to `not`

Bad:

```
(not (= ...))
```

Better:

```
(not= ...)
```

Bad:

```
(if (not (...))
  ...)
```

Better:

```
(if-not (...)
  ...)
```

Bad:

```
(when (not (...))
  ...)
```

Better:

```
(when-not (...)
  ...)
```

if, if, if...

Bad:

```
(if expr-1
  (...)
  (if expr-2
    (...)
    (if expr-3
      (...)
      (...))))
```

if, if, if...

Bad:

```
(if expr-1
  (...)
  (if expr-2
    (...)
    (if expr-3
      (...)
      (...))))
```

Better:

```
(cond
  expr-1 (...)
  expr-2 (...)
  expr-3 (...)
  :else (...))
```

`condp` for avoiding repetition

Bad:

```
(cond
  (= x :a) (...)
  (= x :b) (...)
  :else (...))
```

`condp` for avoiding repetition

Bad:

```
(cond
  (= x :a) (...)
  (= x :b) (...)
  :else (...))
```

Better:

```
(condp = x
  :a (...)
  :b (...)
  (...))
```

condp for avoiding repetition

Bad:

```
(cond
  (= x :a) (...)
  (= x :b) (...)
  :else (...))
```

Better:

```
(condp = x
  :a (...)
  :b (...)
  (...))
```

Best?

```
(case x
  :a (...)
  :b (...)
  (...))
```

`let` around `if` & `when`

Bad:

```
(let [x (...)]
  (if x
    (+ x 8)
    (...)))
```

`let` around `if` & `when`

Bad:

```
(let [x (...)]
  (if x
    (+ x 8)
    (...)))
```

Better:

```
(if-let [x (...)]
  (+ x 8)
  (...))
```

`let` around `if` & `when`

Bad:

```
(let [x (...)]
  (if x
    (+ x 8)
    (...)))
```

Better:

```
(if-let [x (...)]
  (+ x 8)
  (...))
```

Bad:

```
(let [x (...)]
  (when x
    (+ x 8)))
```

`let` around `if` & `when`

Bad:

```
(let [x (...)]
  (if x
    (+ x 8)
    (...)))
```

Better:

```
(if-let [x (...)]
  (+ x 8)
  (...))
```

Bad:

```
(let [x (...)]
  (when x
    (+ x 8)))
```

Better:

```
(when-let [x (...)]
  (+ x 8))
```

`complement` complements predicates

Fine:

```
(every? #(not (predicate? %)) (...))
```

`complement` complements predicates

Fine:

```
(every? #(not (predicate? %)) (...))
```

Better:

```
(every? (complement predicate?) (...))
```

“Loops”

`nil` punning

Bad:

```
(defn scheme-on-you
  [s]
  (when-not (empty? s)
    (...)
    (recur (rest s))))
```

`nil` punning

Bad:

```
(defn scheme-on-you
  [s]
  (when-not (empty? s)
    (...)
    (recur (rest s))))
```

Better:

```
(defn the-clj-way
  [s]
  (when (seq s)
    (...)
    (recur (rest s))))
```

Iterate with side effects

Bad:

```
(doall (map (fn [x]
              (prn x))
            (range 10)))
```

Iterate with side effects

Bad:

```
(doall (map (fn [x]
              (prn x))
            (range 10)))
```

Better:

```
(mapv (fn [x]
        (prn x))
      (range 10))
```

Iterate with side effects

Bad:

```
(doall (map (fn [x]
              (prn x))
            (range 10)))
```

Better:

```
(mapv (fn [x]
        (prn x))
      (range 10))
```

Best?

```
(doseq [x (range 10)]
  (prn x))
```

`for` is not just like a `map`

Bad:

```
(filter even?
        (map (fn [x]
               (let [y (* x 11)]
                 (inc y)))
             (range 10)))
```

`for` is not just like a `map`

Bad:

```
(filter even?
        (map (fn [x]
               (let [y (* x 11)]
                 (inc y)))
             (range 10)))
```

Better:

```
(filter even?
        (for [x (range 10)
              :let [y (* x 11)]]
          (inc y)))
```

`for` is not just like a `map`

Bad:

```
(filter even?
        (map (fn [x]
               (let [y (* x 11)]
                 (inc y)))
             (range 10)))
```

Better:

```
(filter even?
        (for [x (range 10)
              :let [y (* x 11)]]
          (inc y)))
```

Best?

```
(for [x (range 10)
      :let [y (* x 11)]
      :when (odd? y)] ; check before increment
  (inc y))
```

There's more to `for`: Nesting

Bad:

```
(apply concat (map (fn [x]
                     (map (fn [y]
                            (+ x y))
                          (range x)))
                   (range 5)))
```

There's more to `for`: Nesting

Bad:

```
(apply concat (map (fn [x]
                     (map (fn [y]
                            (+ x y))
                          (range x)))
                   (range 5)))
```

Better:

```
(mapcat (fn [x]
          (map (fn [y]
                 (+ x y))
               (range x)))
        (range 5))
```

There's more to `for`: Nesting

Bad:

```
(apply concat (map (fn [x]
                     (map (fn [y]
                            (+ x y))
                          (range x)))
                   (range 5)))
```

Better:

```
(mapcat (fn [x]
          (map (fn [y]
                 (+ x y))
               (range x)))
        (range 5))
```

Best?

```
(for [x (range 5)
      y (range x)]
  (+ x y))
```

There's even more to `for`: Terminating

Fine:

```
(map #(* % 2) (take-while #(< % 10) (range 20)))
```

There's even more to `for`: Terminating

Fine:

```
(map #(* % 2) (take-while #(< % 10) (range 20)))
```

Better:

```
(for [x (range 20)
      :while (< x 10)]
  (* x 2))
```

Higher order `fn`s

Be partial

Fine:

```
(map #(clojure.string/join " " %)
     [["Ketchup" "is" "a" "Vegetable"]
      ["Get" "out" "of" "my" "yard"]])
```

Be partial

Fine:

```
(map #(clojure.string/join " " %)
     [["Ketchup" "is" "a" "Vegetable"]
      ["Get" "out" "of" "my" "yard"]])
```

Better:

```
(map (partial clojure.string/join " ")
     [["ITS" "TERRIBLE" "SECRETS" "BLIGHT" "YOUR" "MIND"]
      ["SEEK" "THE" "TOTALITY" "OF" "FOUR" "RUNES!"]])
```

`compose` like it's 1748–49

Bad:

```
(map (fn [x] (even? (* (val x) 3))) (...))
```

compose like it's 1748–49

Bad:

```
(map (fn [x] (even? (* (val x) 3))) (...))
```

Better:

```
(map (comp even? (partial * 3) val) (...))
```

Bad:

```
(reduce (fn [acc entry]
          (assoc acc (:key entry) (:value entry)))
        {}
        [{:key :hu :value "Sajt"}
         {:key :de :value "Käse"}
         {:key :en :value "Cheese"}])
```

juxtify

Bad:

```
(reduce (fn [acc entry]
          (assoc acc (:key entry) (:value entry)))
        {}
        [{:key :hu :value "Sajt"}
         {:key :de :value "Käse"}
         {:key :en :value "Cheese"}])
```

Better:

```
(into {}
      (map (juxt :key :value)
           [{:key :hu :value "Sajt"}
            {:key :de :value "Käse"}
            {:key :en :value "Cheese"}]))
```

Threading

Don’t thread lightly

Bad:

```
(char (* (Integer/parseInt (str \2)) 6))
```

Don't thread lightly

Bad:

```
(char (* (Integer/parseInt (str \2)) 6))
```

Better:

```
(-> \2 str Integer/parseInt (* 6) char)
```

Don’t thread lightly

Bad:

```
(reduce (...)
        (filter (...)
                (map (...) (range 10))))
```

Don't thread lightly

Bad:

```
(reduce (...)
        (filter (...)
                (map (...) (range 10))))
```

Better:

```
(->> (range 10)
     (map (...))
     (filter (...))
     (reduce (...)))
```

do things to thing (then use it)

Bad:

```
(let [pants (management.FancyPantsFactory/getTrousersInstance)]
  (.setColor pants "Khaki")
  (.setStyle pants "smart-casual")
  (.setMaterial pants "cotton")
  (.putOn (...) pants))
```

do things to thing (then use it)

Bad:

```
(let [pants (management.FancyPantsFactory/getTrousersInstance)]
  (.setColor pants "Khaki")
  (.setStyle pants "smart-casual")
  (.setMaterial pants "cotton")
  (.putOn (...) pants))
```

Better:

```
(->> (doto (management.FancyPantsFactory/getTrousersInstance)
       (.setColor "Khaki")
       (.setStyle "smart-casual")
       (.setMaterial "cotton"))
     (.putOn (...)))
```

Destructuring

getting things

Bad:

```
(defn remaining-vacation-days [person]
  (let [age (get person :age)
        days-on-vacation (get person :days-on-vacation)]
    (- (+ 20 (* age 0.2))
       (count days-on-vacation))))
```

getting things

Bad:

```
(defn remaining-vacation-days [person]
  (let [age (get person :age)
        days-on-vacation (get person :days-on-vacation)]
    (- (+ 20 (* age 0.2))
       (count days-on-vacation))))
```

Better:

```
(defn remaining-vacation-days
  [{age :age
    days-on-vacation :days-on-vacation}]
  (- (+ 20 (* age 0.2))
     (count days-on-vacation)))
```

getting things

Bad:

```
(defn remaining-vacation-days [person]
  (let [age (get person :age)
        days-on-vacation (get person :days-on-vacation)]
    (- (+ 20 (* age 0.2))
       (count days-on-vacation))))
```

Better:

```
(defn remaining-vacation-days
  [{age :age
    days-on-vacation :days-on-vacation}]
  (- (+ 20 (* age 0.2))
     (count days-on-vacation)))
```

Best?

```
(defn remaining-vacation-days [{:keys [age days-on-vacation]}]
  (- (+ 20 (* age 0.2))
     (count days-on-vacation)))
```

vectors too

Bad:

```
(defn add-length-to-first-two-numbers
  [numbers]
  (let [a (first numbers)
        b (second numbers)]
    (+ a b (count numbers))))
```

vectors too

Bad:

```
(defn add-length-to-first-two-numbers
  [numbers]
  (let [a (first numbers)
        b (second numbers)]
    (+ a b (count numbers))))
```

Better:

```
(defn add-length-to-first-two-numbers
  [[a b
    :as numbers]]
  (+ a b (count numbers)))
```

Addendum

A Short Ballad Dedicated to the Growth of Programs

This is a tale of a sorry quest
To master pure code at the T guru's behest
I enrolled in a class that appealing did seem
For it promised to teach fine things like T3 and Scheme

The first day went fine; we learned of cells
And symbols and lists and functions as well
Lisp I had mastered and excited was I
For to master T3 my hackstincts did cry

I sailed through the first week with no problems at all
And I even said "closure" instead of "function call"
Then said the master that ready were we
To start real hacking instead of simple theory

A Short Ballad Dedicated to the Growth of Programs

Will you, said he, write me a function please
That in lists would associate values with keys
I went home and turned on my trusty Apollo
And wrote a function whose definition follows:

```
(cdr (assq key a-list))
```

A one-liner I thought, fool that I was
Just two simple calls without a `COND` clause
But when I tried this function to run
`CDR` didn't think that `NIL` was much fun

So I tried again like the good King of yore
And of code I easily generated some more:

```
(cond ((assq key a-list) => cdr))
```

A Short Ballad Dedicated to the Growth of Programs

It got longer but purer, and it wasn't too bad
But then `COND` ran out and that was quite sad

Well, that isn't hard to fix, I was told
Just write some more code, my son, be bold
Being young, not even a moment did I pause
I stifled my instincts and added a clause

```
(cond ((assq key a-list) => cdr)
      (else nil))}
```

Sometimes this worked and sometimes it broke
I debugged and prayed and even had a stroke
Many a guru tried valiantly to help
But undefined datums their efforts did squelch.

A Short Ballad Dedicated to the Growth of Programs

I returneth once more to the great sage of T
For no way out of the dilemma I could see
He said it was easy – more lines must I fill
with code, for `FALSE` was no longer `NIL`.

```
(let ((val (assq key a-list)))
        (cond (val (cdr val))
         (else nil)))
```

You'd think by now I might be nearing the end
Of my ballad which seems bad things to portend
You'd think that we could all go home scot-free
But `COND` eschewed `VAL`; it wanted `#T`

A Short Ballad Dedicated to the Growth of Programs

So I went back to the master and appealed once again
I said, pardon me, but now I'm really insane
He said, no you're not really going out of your head
Instead of just `VAL`, you must use `NOT NULL` instead

```
(let ((val (assq key a-list)))
        (cond ((not (null? val)) (cdr val))
         (else nil)))
```

A Short Ballad Dedicated to the Growth of Programs

My song is over and I'm going home to bed
With this ineffable feeling that I've been misled
And just in case my point you have missed
Somehow I preferred

```
(CDR (ASSQ KEY A-LIST))
```

by Ashwin Ram