Understanding Core Clojure Functions

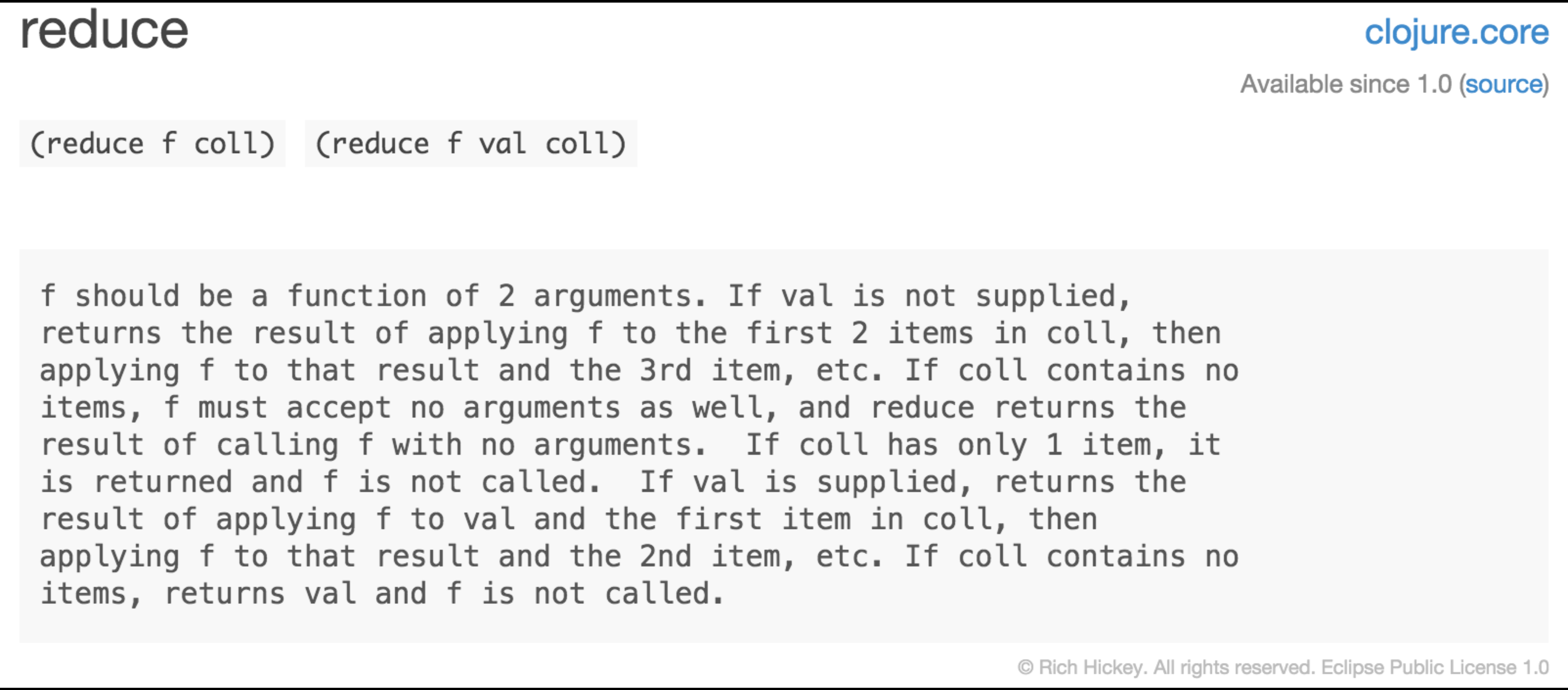
8th Light

Dr. Jonathan Graham

MINED MINDS

http://jonathangraham.github.io twitter: @graham_jp

Are you fully in control when you code?



reduce

clojure.core

Available since 1.0 (source)

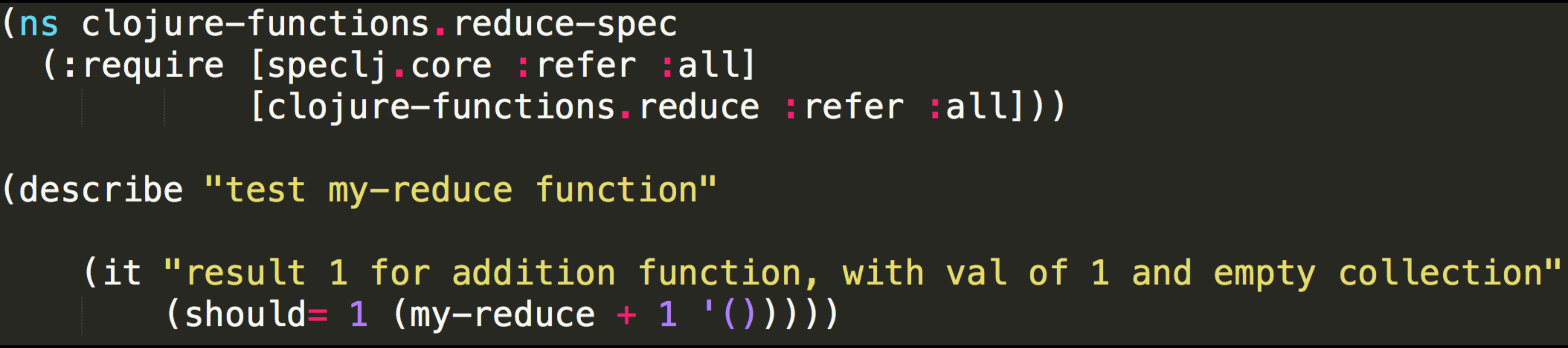
```
(reduce f coll)   (reduce f val coll)
```

```
f should be a function of 2 arguments. If val is not supplied,
returns the result of applying f to the first 2 items in coll, then
applying f to that result and the 3rd item, etc. If coll contains no
items, f must accept no arguments as well, and reduce returns the
result of calling f with no arguments.  If coll has only 1 item, it
is returned and f is not called.  If val is supplied, returns the
result of applying f to val and the first item in coll, then
applying f to that result and the 2nd item, etc. If coll contains no
items, returns val and f is not called.
```

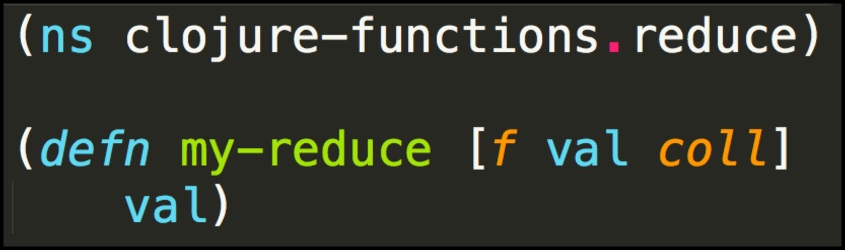
If val is supplied, returns the result of applying f to val and the first item in coll, then applying f to that result and the 2nd item, etc.

If coll contains no items, returns val and f is not called.

Let’s start with a test…

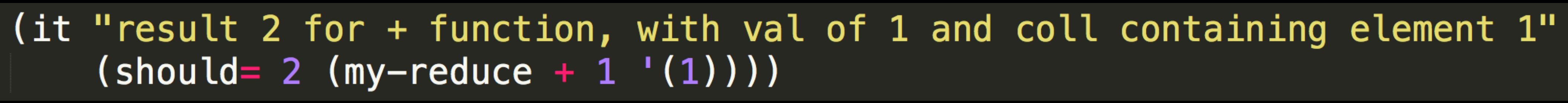


```
(ns clojure-functions.reduce-spec
  (:require [speclj.core :refer :all]
            [clojure-functions.reduce :refer :all]))

(describe "test my-reduce function"

    (it "result 1 for addition function, with val of 1 and empty collection"
        (should= 1 (my-reduce + 1 '()))))
```

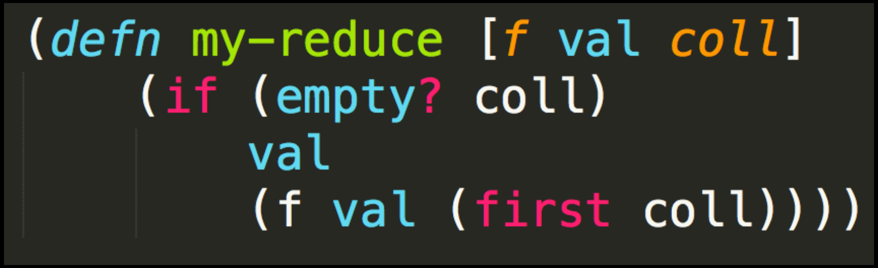


```
(ns clojure-functions.reduce)

(defn my-reduce [f val coll]
    val)
```

A second test could have a single element in the collection.



```clojure
(it "result 2 for + function, with val of 1 and coll containing element 1"
    (should= 2 (my-reduce + 1 '(1))))
```



```clojure
(defn my-reduce [f val coll]
    (if (empty? coll)
        val
        (f val (first coll))))
```

With more than one item in the collection…

```clojure
(it "result 6 for + function, val of 1 and coll with elements 2 and 3"
    (should= 6 (my-reduce + 1 [2 3])))
```

```clojure
(defn my-reduce [f val coll]
    (if (empty? coll)
            val
            (my-reduce f (f val (first coll)) (rest coll))))
```

We can add tests with different functions, collections and initial values.

If val is not supplied, returns the result of applying f to the first 2 items in coll, then applying f to that result and the 3rd item, etc.

No val…

```
(it "result 1 for + function, no val, and coll containing element 1"
    (should= 1 (my-reduce + [1])))
```

```
(defn my-reduce
    ([f coll]
        (my-reduce f (first coll) (rest coll)))
    ([f val coll]
    (if (empty? coll)
           val
           (my-reduce f (f val (first coll)) (rest coll)))))
```

We can add tests with multiple items in coll.

If coll contains no items, f must accept no arguments as well, and reduce returns the result of calling f with no arguments.

No val and empty coll…

```clojure
(it "result 0 for addition on empty list"
    (should= 0 (my-reduce + [])))

(it "result 1 for multiplication on empty list"
    (should= 1 (my-reduce * [])))
```

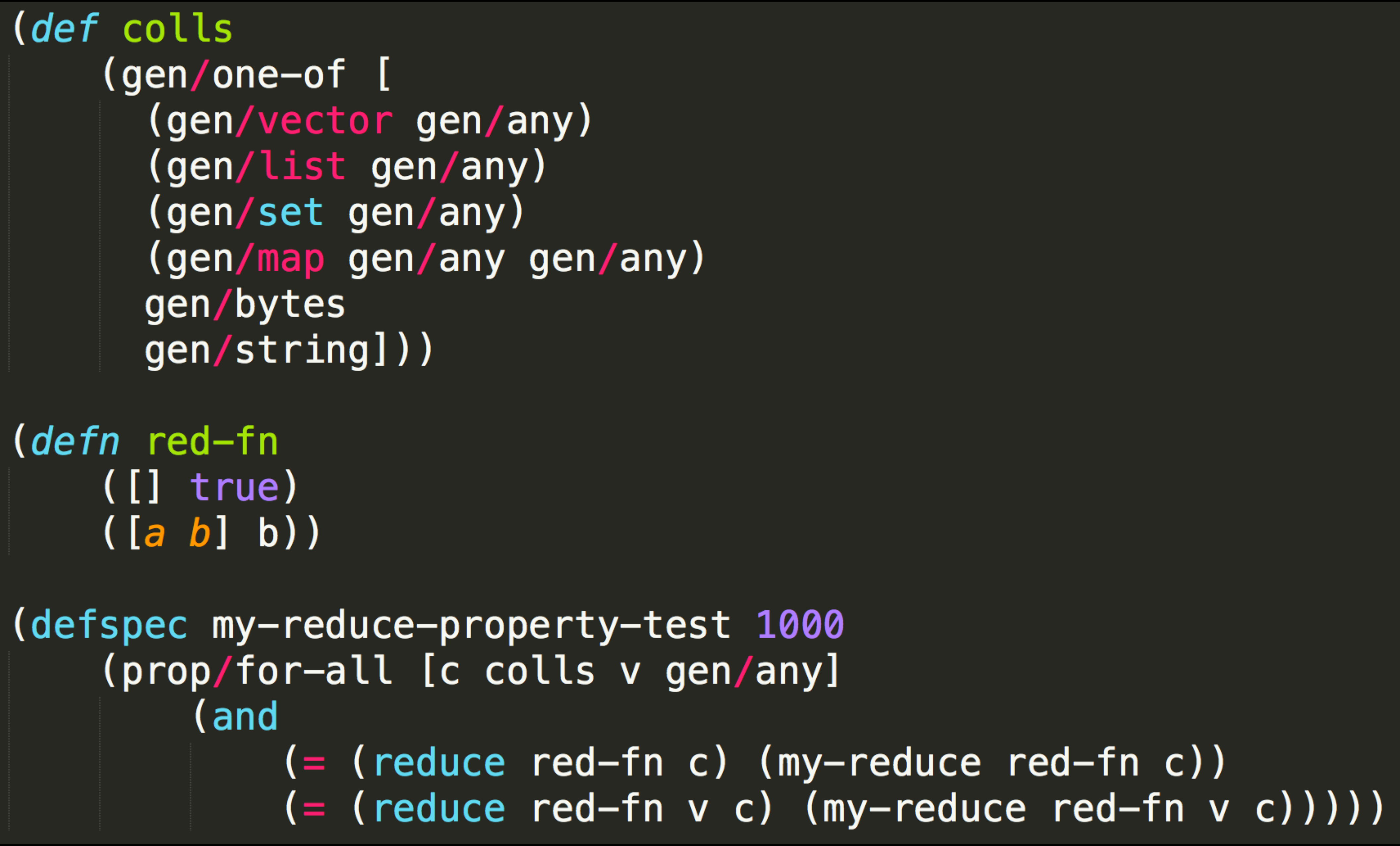
```clojure
(defn my-reduce
    ([f coll]
        (if (empty? coll)
            (f)
            (my-reduce f (first coll) (rest coll))))
    ([f val coll]
        (if (empty? coll)
            val
            (my-reduce f (f val (first coll)) (rest coll)))))
```

How many unit tests are enough to give us confidence that our function behaves in the same was as *reduce*?

Property-based tests

Make statements about the expected behaviour of the code that should hold true for the entire domain of possible inputs. These statements are then verified for many different (pseudo)randomly generated inputs.

Clojure test.check



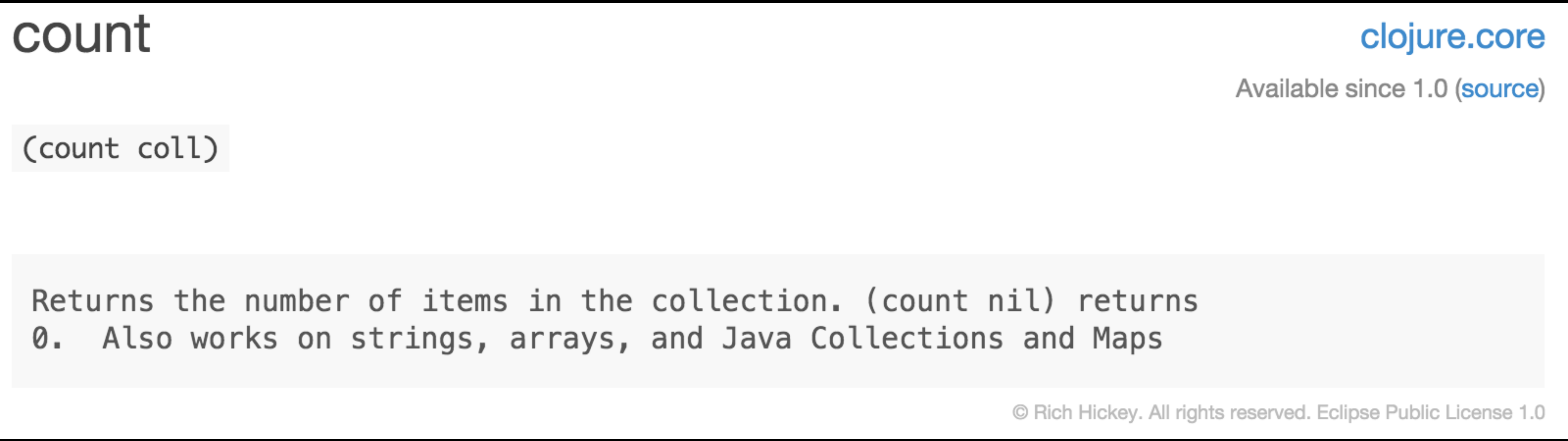
```
(def colls
    (gen/one-of [
      (gen/vector gen/any)
      (gen/list gen/any)
      (gen/set gen/any)
      (gen/map gen/any gen/any)
      gen/bytes
      gen/string]))

(defn red-fn
    ([] true)
    ([a b] b))

(defspec my-reduce-property-test 1000
    (prop/for-all [c colls v gen/any]
        (and
            (= (reduce red-fn c) (my-reduce red-fn c))
            (= (reduce red-fn v c) (my-reduce red-fn v c)))))
```

Functions requiring two arguments can be passed to reduce with only a single argument

```
(defn f [x y] (+ x y))

(reduce f [1])
```

Care needed with functions that cannot be evaluated with no arguments

```
(reduce - ‘())
```


count

clojure.core

Available since 1.0 (source)

```
(count coll)
```

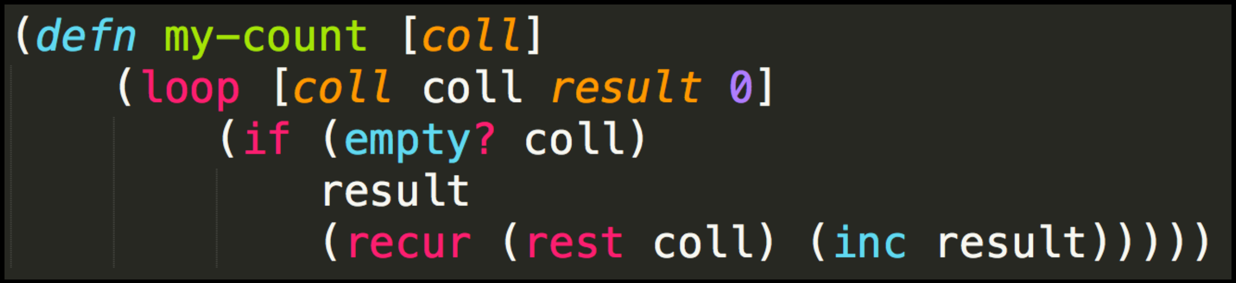
```
Returns the number of items in the collection. (count nil) returns
0.  Also works on strings, arrays, and Java Collections and Maps
```

Taking a TDD approach again…

```
(it "result 0 for an empty list"
    (should= 0 (my-count '())))

(it "result 0 for nil"
    (should= 0 (my-count nil)))

(it "result 1 for a list of one item"
    (should= 1 (my-count '(1))))
```

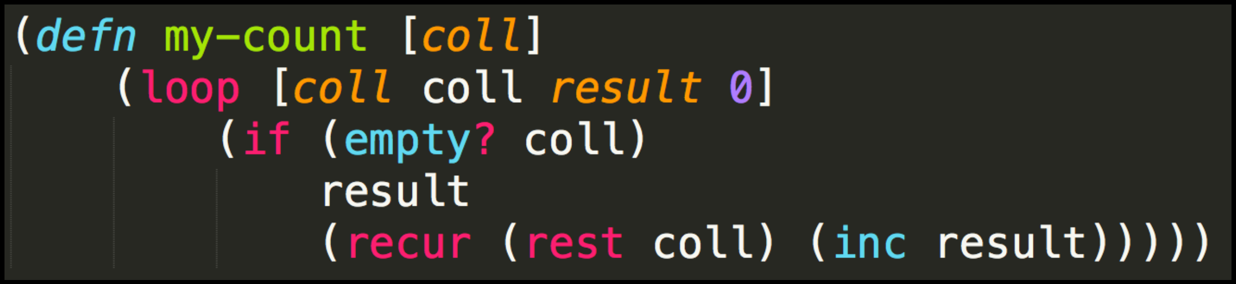
```
(defn my-count [coll]
    (if (empty? coll)
        0
        1))
```

Make tests with more than one element pass

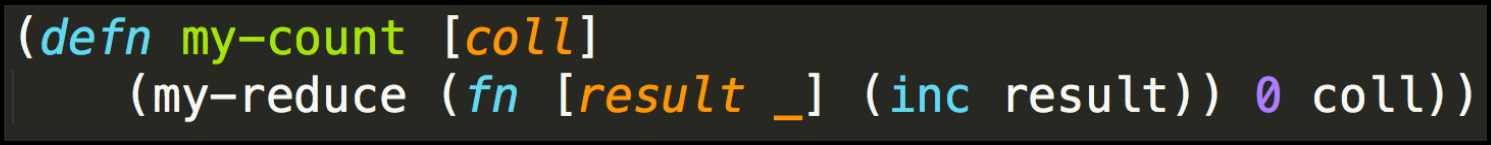


```clojure
(defn my-count [coll]
    (loop [coll coll result 0]
        (if (empty? coll)
            result
            (recur (rest coll) (inc result)))))
```

With a passing test suite, refactor

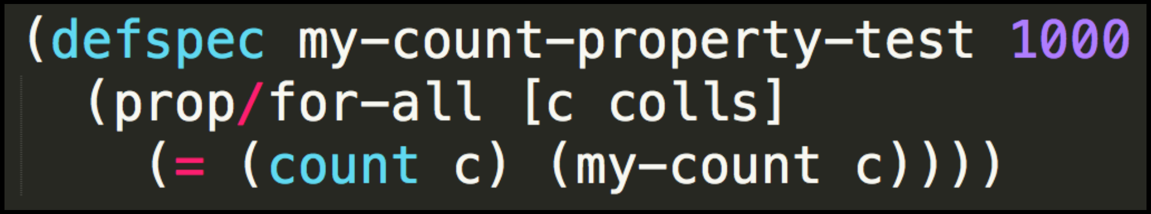


```clojure
(defn my-count [coll]
    (loop [coll coll result 0]
        (if (empty? coll)
            result
            (recur (rest coll) (inc result)))))
```

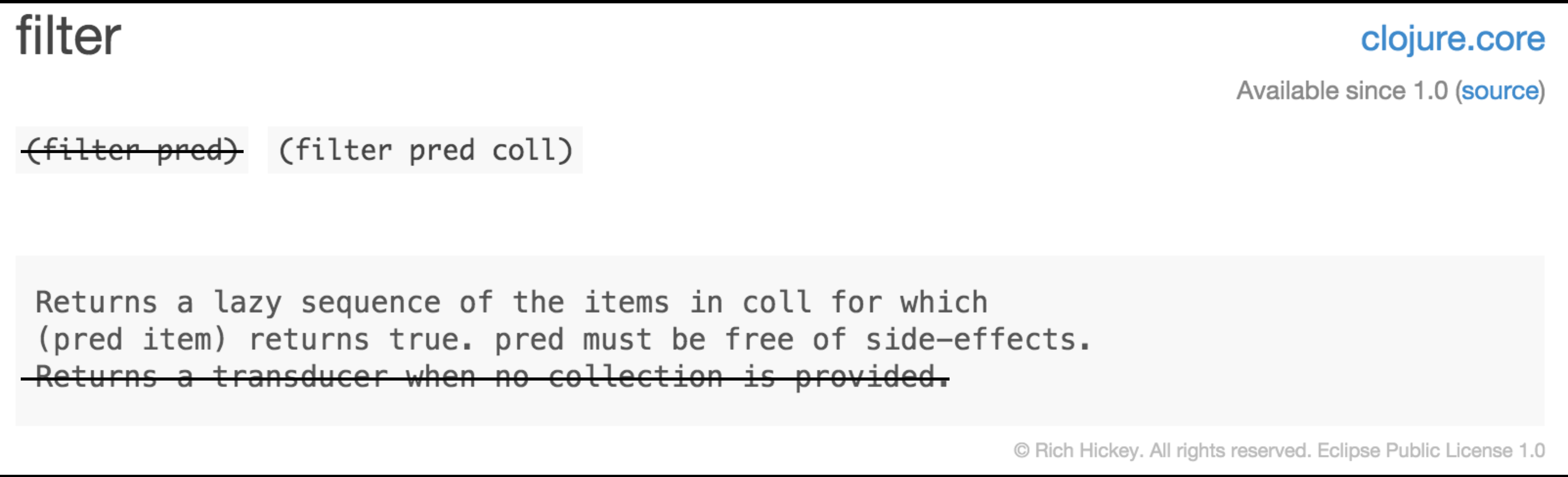


```clojure
(defn my-count [coll]
    (my-reduce (fn [result _] (inc result)) 0 coll))
```

Again, property-based tests can confirm that our function behaves the same as the core function



```
(defspec my-count-property-test 1000
  (prop/for-all [c colls]
    (= (count c) (my-count c))))
```



filter

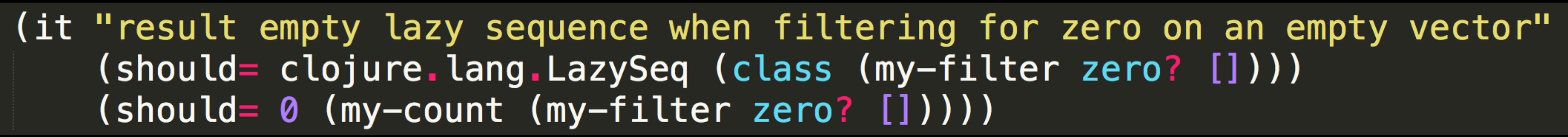
clojure.core

Available since 1.0 (source)

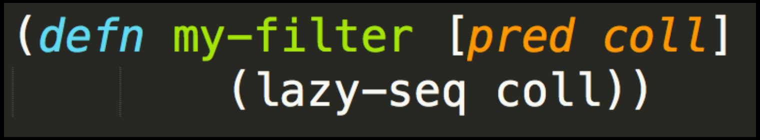
```
~~(filter pred)~~   (filter pred coll)
```

```
Returns a lazy sequence of the items in coll for which
(pred item) returns true. pred must be free of side-effects.
~~Returns a transducer when no collection is provided.~~
```

Filter returns a lazy sequence

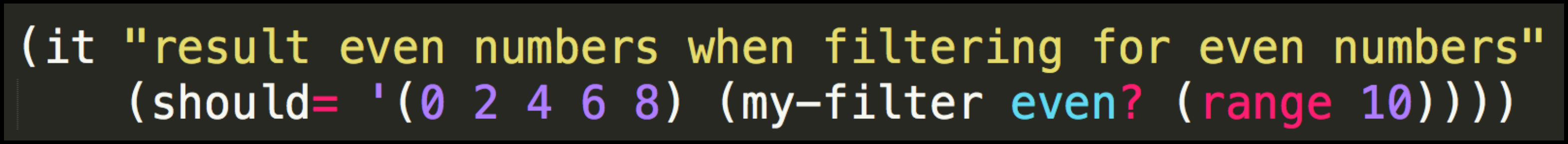


```
(it "result empty lazy sequence when filtering for zero on an empty vector"
    (should= clojure.lang.LazySeq (class (my-filter zero? [])))
    (should= 0 (my-count (my-filter zero? []))))
```

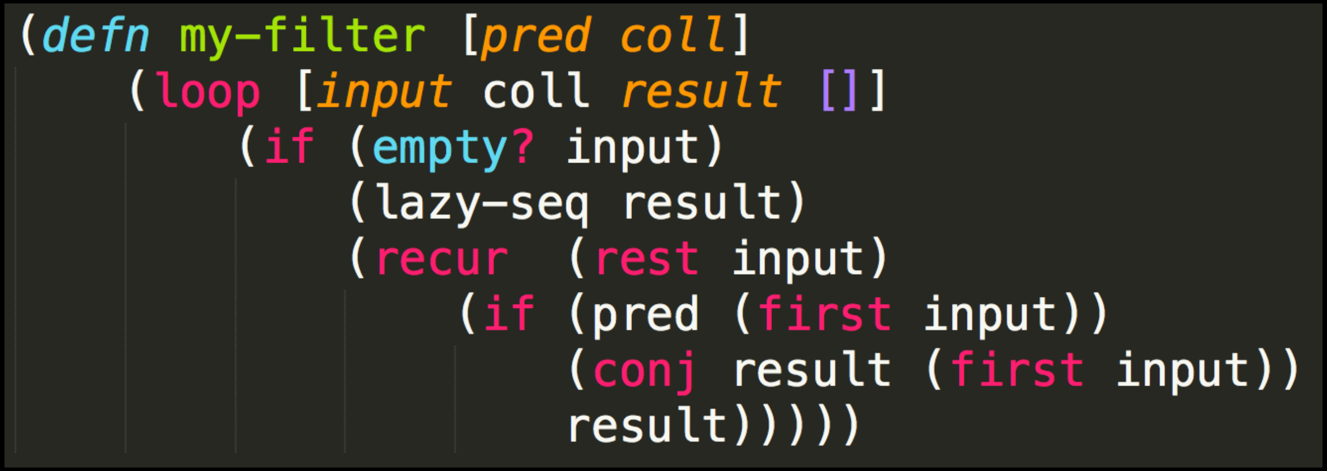


```
(defn my-filter [pred coll]
       (lazy-seq coll))
```

Build function up recursively?



```clojure
(it "result even numbers when filtering for even numbers"
    (should= '(0 2 4 6 8) (my-filter even? (range 10))))
```



```clojure
(defn my-filter [pred coll]
    (loop [input coll result []]
        (if (empty? input)
            (lazy-seq result)
            (recur  (rest input)
                (if (pred (first input))
                    (conj result (first input))
                    result))))))
```

We can add more tests, and these pass

But, our function is not lazy!

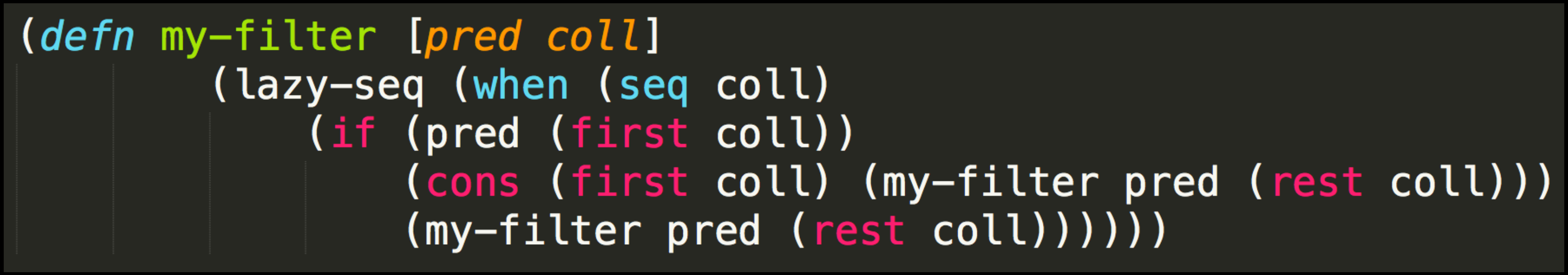
We just convert the result to a lazy sequence

cons is lazy; *conj* is not

conj depends on the collection type, so is realised immediately

cons adds an item to the start of a collection, and can be evaluated lazily

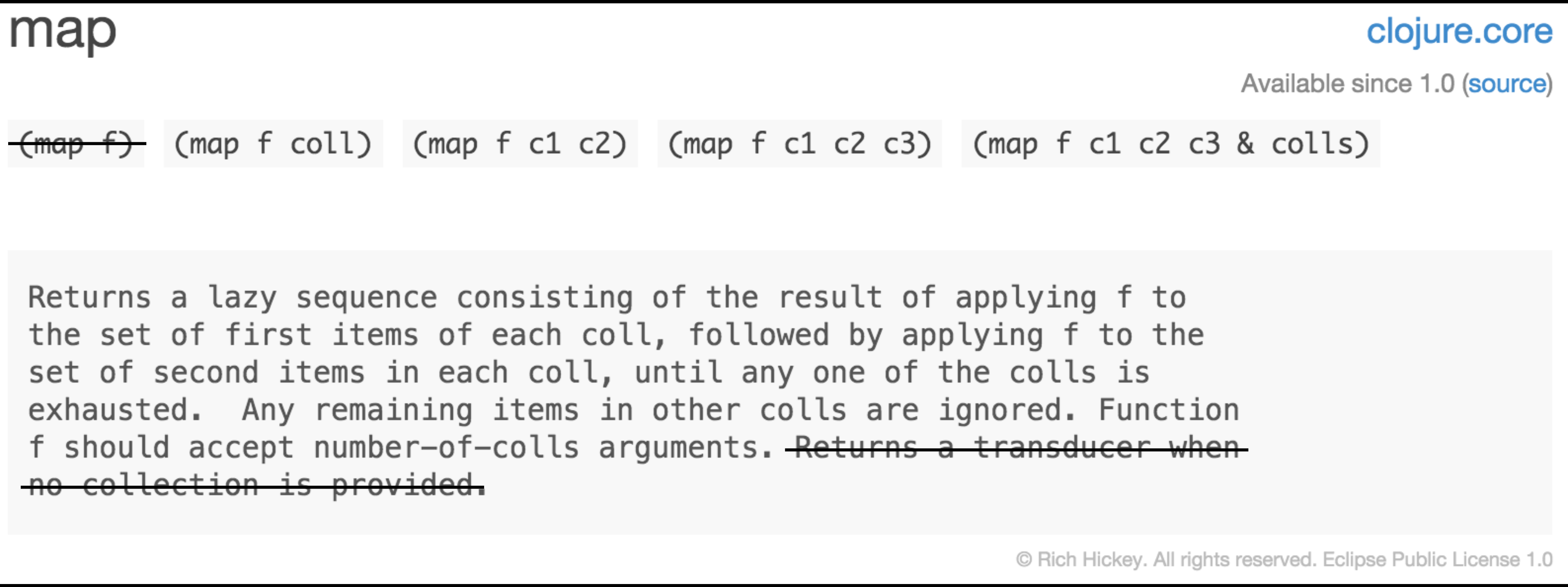
Refactor lazily



```clojure
(defn my-filter [pred coll]
        (lazy-seq (when (seq coll)
            (if (pred (first coll))
                (cons (first coll) (my-filter pred (rest coll)))
                (my-filter pred (rest coll))))))
```

How do we get a vector containing just the even numbers given an input of [0 1 2 3 4 5]?

```
(into [] (my-filter even? [0 1 2 3 4 5]))
```



map

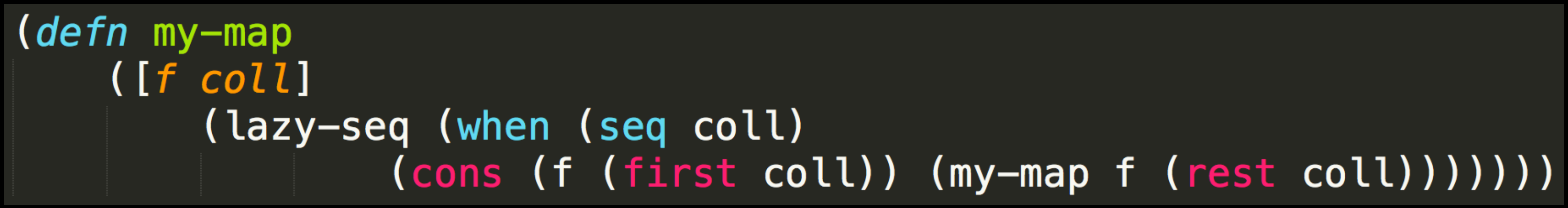
clojure.core

Available since 1.0 (source)

~~`(map f)`~~ `(map f coll)` `(map f c1 c2)` `(map f c1 c2 c3)` `(map f c1 c2 c3 & colls)`

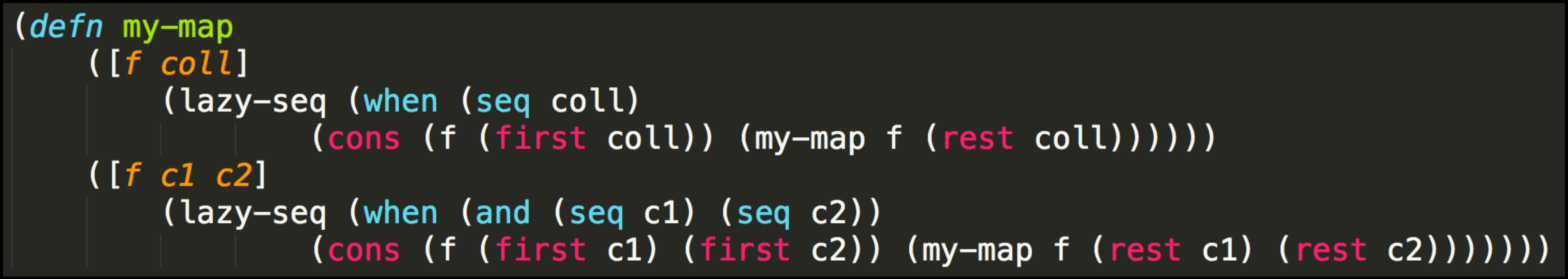
```
Returns a lazy sequence consisting of the result of applying f to
the set of first items of each coll, followed by applying f to the
set of second items in each coll, until any one of the colls is
exhausted.  Any remaining items in other colls are ignored. Function
f should accept number-of-colls arguments.
```
~~Returns a transducer when no collection is provided.~~

For a single collection, write our map function analogously to our filter function



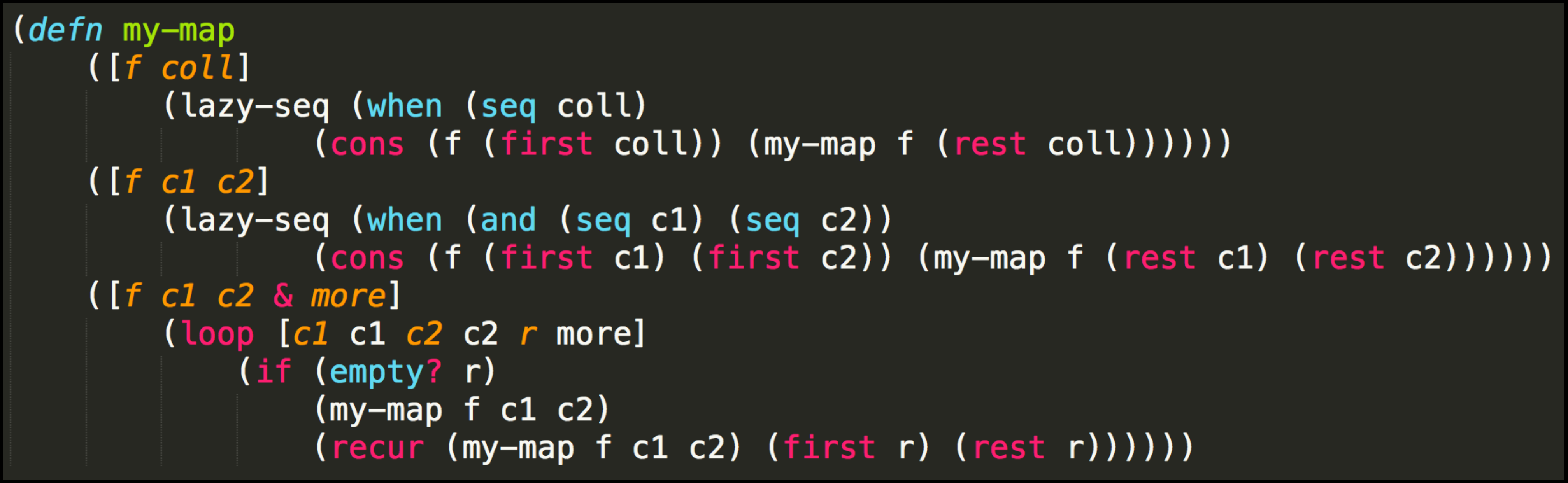
```clojure
(defn my-map
    ([f coll]
        (lazy-seq (when (seq coll)
                (cons (f (first coll)) (my-map f (rest coll)))))))
```

We can extend this approach for two collections



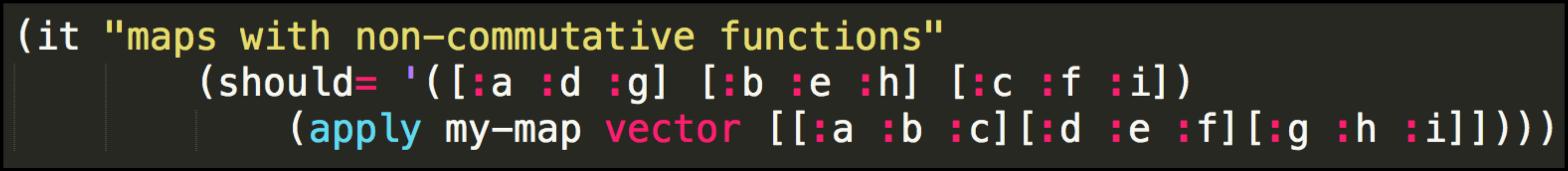
```
(defn my-map
    ([f coll]
        (lazy-seq (when (seq coll)
                (cons (f (first coll)) (my-map f (rest coll))))))
    ([f c1 c2]
        (lazy-seq (when (and (seq c1) (seq c2))
                (cons (f (first c1) (first c2)) (my-map f (rest c1) (rest c2)))))))
```

What about more than two collections? Build up recursively?

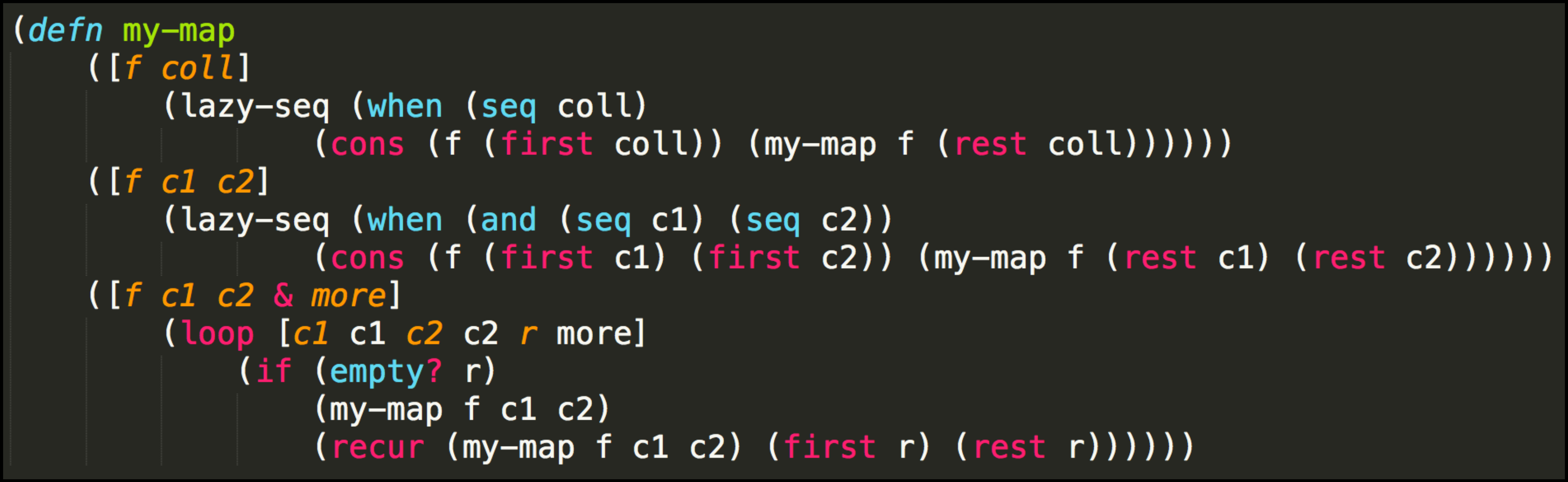


```clojure
(defn my-map
    ([f coll]
        (lazy-seq (when (seq coll)
                (cons (f (first coll)) (my-map f (rest coll))))))
    ([f c1 c2]
        (lazy-seq (when (and (seq c1) (seq c2))
                (cons (f (first c1) (first c2)) (my-map f (rest c1) (rest c2))))))
    ([f c1 c2 & more]
        (loop [c1 c1 c2 c2 r more]
            (if (empty? r)
                (my-map f c1 c2)
                (recur (my-map f c1 c2) (first r) (rest r))))))
```

Does this work for non-commutative functions?



```clojure
(it "maps with non-commutative functions"
        (should= '([:a :d :g] [:b :e :h] [:c :f :i])
            (apply my-map vector [[:a :b :c][:d :e :f][:g :h :i]])))
```

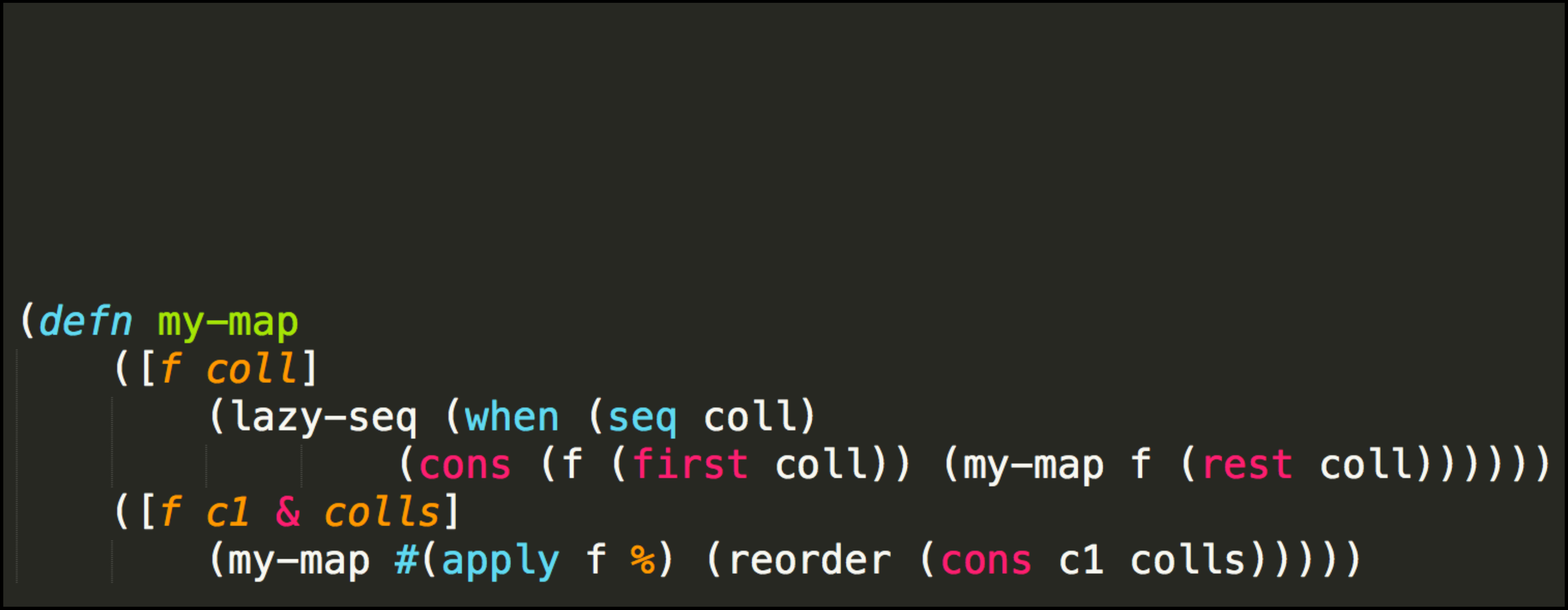


```clojure
(defn my-map
    ([f coll]
        (lazy-seq (when (seq coll)
                (cons (f (first coll)) (my-map f (rest coll))))))
    ([f c1 c2]
        (lazy-seq (when (and (seq c1) (seq c2))
                (cons (f (first c1) (first c2)) (my-map f (rest c1) (rest c2))))))
    ([f c1 c2 & more]
        (loop [c1 c1 c2 c2 r more]
            (if (empty? r)
                (my-map f c1 c2)
                (recur (my-map f c1 c2) (first r) (rest r))))))
```

How can build lazily?

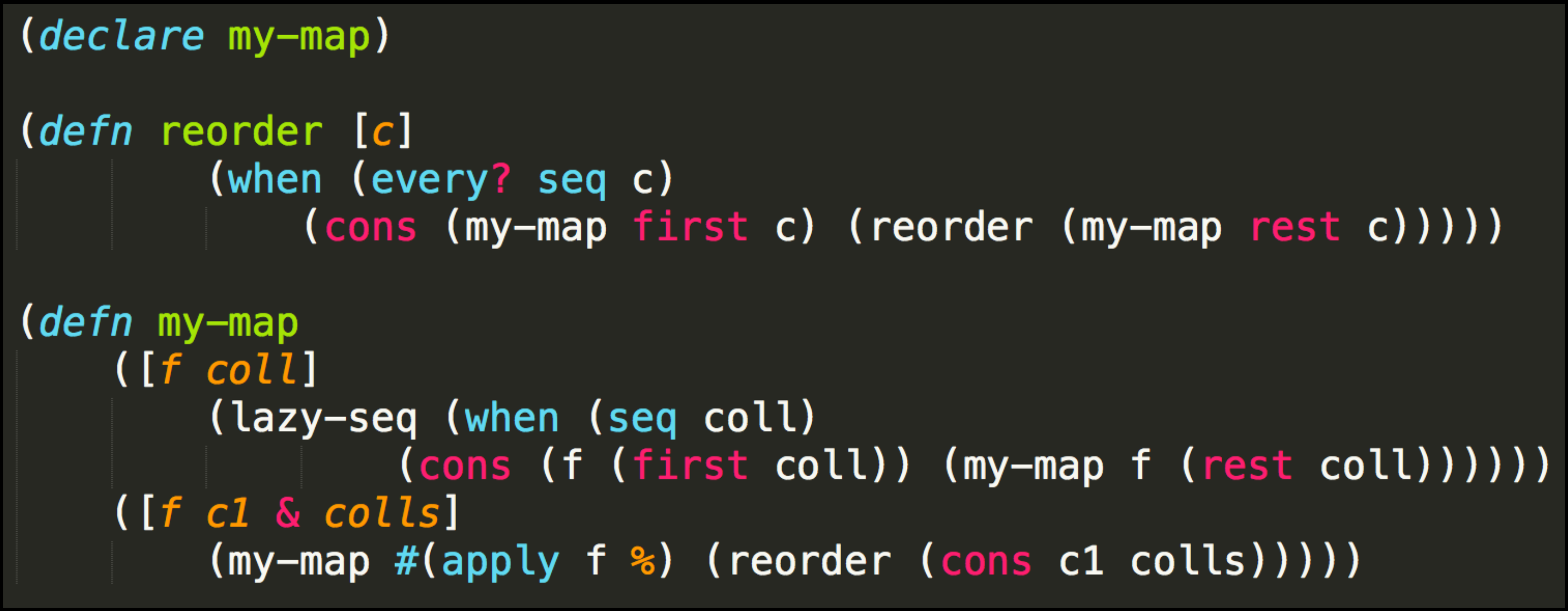
1) Take all of the input collections, and put them into a single sequence.
2) Reorder this sequence, so the first collection is all of the first elements, the second collection is all of the second elements, etc.
3) Map the result of applying the function to each reordered collection in turn

Our map function

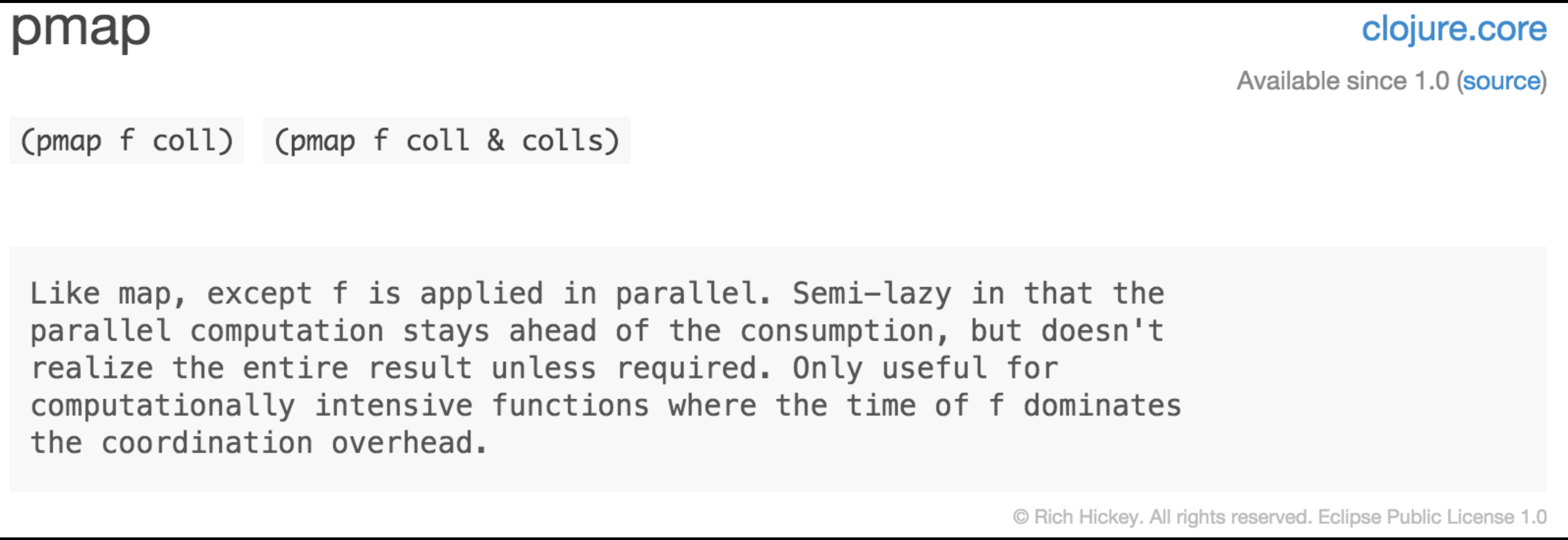


```clojure
(defn my-map
    ([f coll]
        (lazy-seq (when (seq coll)
                (cons (f (first coll)) (my-map f (rest coll))))))
    ([f c1 & colls]
        (my-map #(apply f %) (reorder (cons c1 colls)))))
```

Our map function



```clojure
(declare my-map)

(defn reorder [c]
        (when (every? seq c)
            (cons (my-map first c) (reorder (my-map rest c)))))

(defn my-map
    ([f coll]
        (lazy-seq (when (seq coll)
                (cons (f (first coll)) (my-map f (rest coll))))))
    ([f c1 & colls]
        (my-map #(apply f %) (reorder (cons c1 colls)))))
```



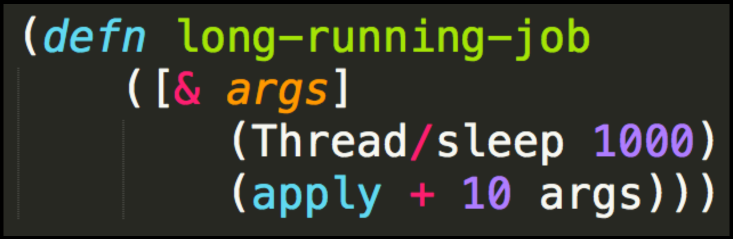
pmap

clojure.core

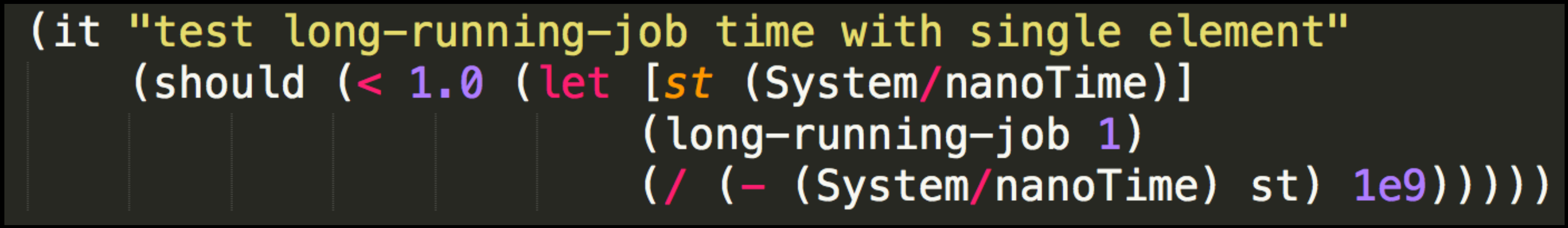
Available since 1.0 (source)

`(pmap f coll)` `(pmap f coll & colls)`

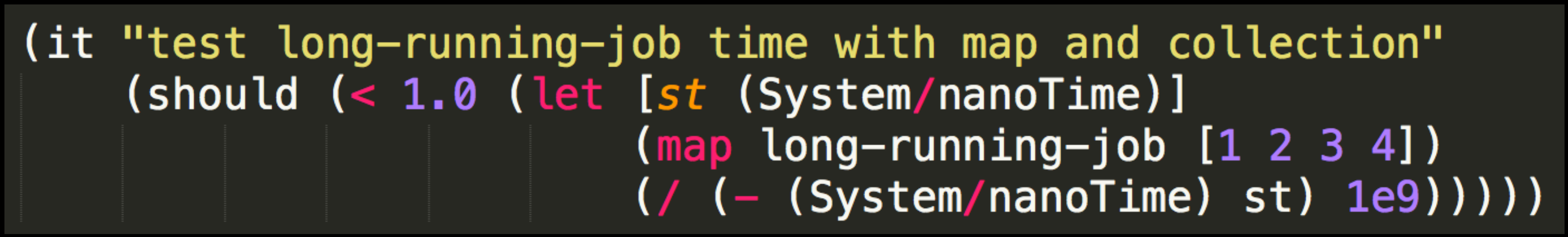
```
Like map, except f is applied in parallel. Semi-lazy in that the
parallel computation stays ahead of the consumption, but doesn't
realize the entire result unless required. Only useful for
computationally intensive functions where the time of f dominates
the coordination overhead.
```



```clojure
(defn long-running-job
  ([& args]
    (Thread/sleep 1000)
    (apply + 10 args)))
```

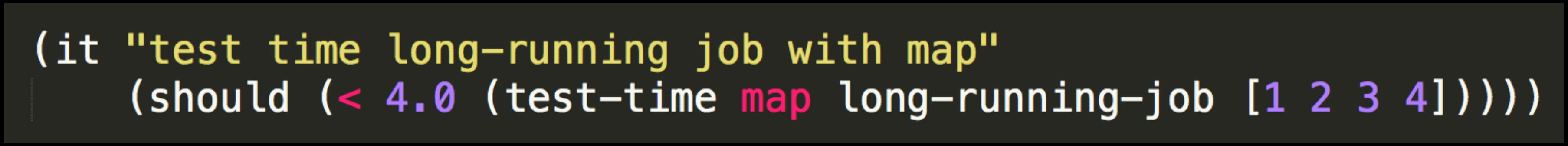


```clojure
(it "test long-running-job time with single element"
  (should (< 1.0 (let [st (System/nanoTime)]
                   (long-running-job 1)
                   (/ (- (System/nanoTime) st) 1e9)))))
```

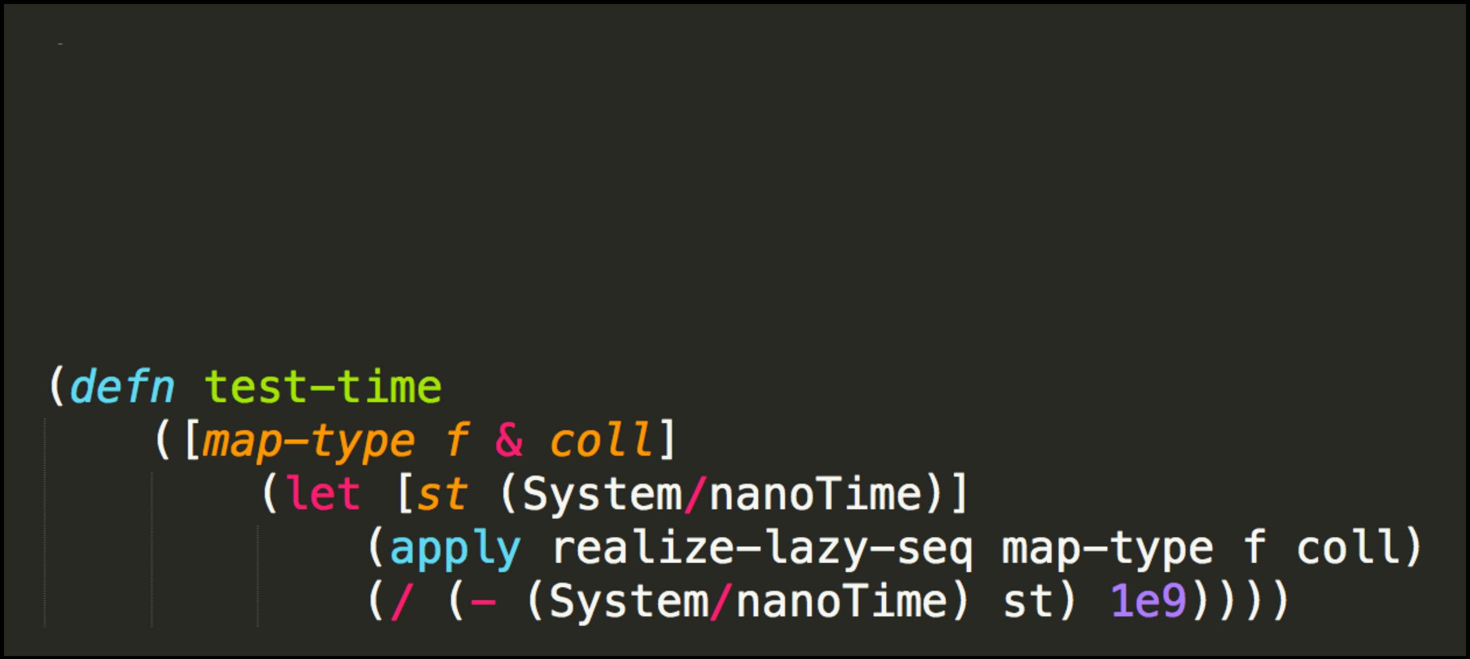


```clojure
(it "test long-running-job time with map and collection"
  (should (< 1.0 (let [st (System/nanoTime)]
                   (map long-running-job [1 2 3 4])
                   (/ (- (System/nanoTime) st) 1e9)))))
```

map is lazy, and the time we are measuring is the time to make a new lazy-seq, not the time to actually execute it



```
(it "test time long-running job with map"
    (should (< 4.0 (test-time map long-running-job [1 2 3 4]))))
```



```
(defn test-time
    ([map-type f & coll]
        (let [st (System/nanoTime)]
            (apply realize-lazy-seq map-type f coll)
            (/ (- (System/nanoTime) st) 1e9))))
```

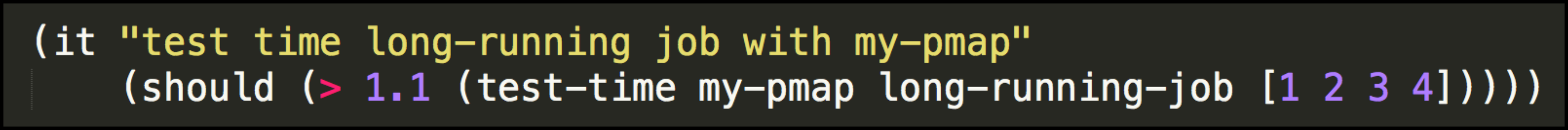
```clojure
(it "test time long-running job with map"
    (should (< 4.0 (test-time map long-running-job [1 2 3 4]))))
```

```clojure
(defn realize-lazy-seq
    ([map-type f & args]
        (loop [res (apply map-type f args)]
            (when res
                (recur (next res))))))

(defn test-time
    ([map-type f & coll]
        (let [st (System/nanoTime)]
            (apply realize-lazy-seq map-type f coll)
            (/ (- (System/nanoTime) st) 1e9))))
```

```
(it "test time long-running job with my-pmap"
    (should (> 1.1 (test-time my-pmap long-running-job [1 2 3 4]))))
```

```
(defn my-pmap
    ([f coll]
        (let [results (my-map #(future (f %)) coll)]
            (my-map deref results))))
```

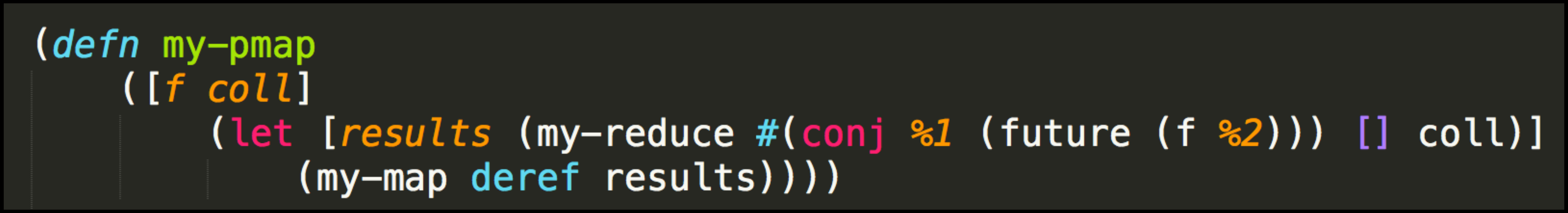
Fails



```clojure
(it "test time long-running job with my-pmap"
    (should (> 1.1 (test-time my-pmap long-running-job [1 2 3 4]))))
```

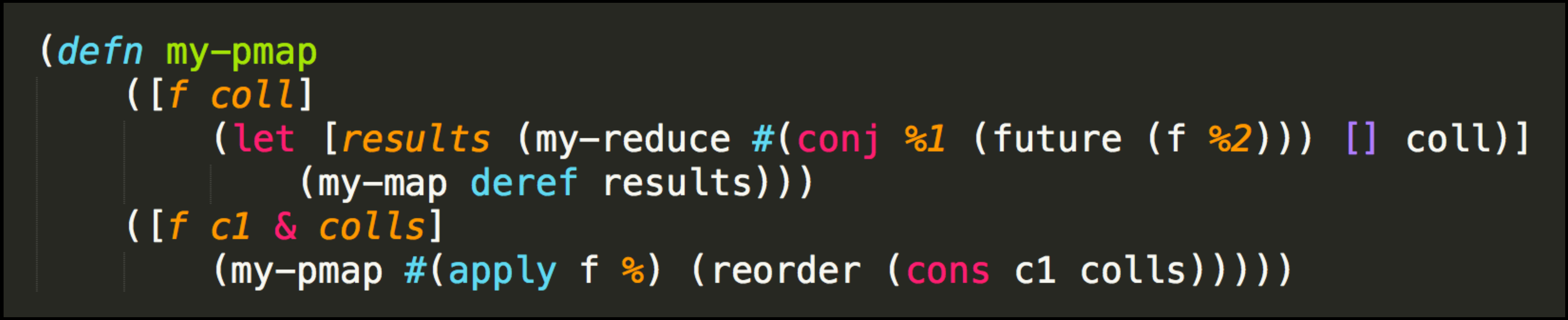
We need to generate all of the futures before we start to deref

Remember that *conj* is not lazy…



```clojure
(defn my-pmap
    ([f coll]
        (let [results (my-reduce #(conj %1 (future (f %2))) [] coll)]
            (my-map deref results))))
```

Extend to multiple collections as we did with *my-map*



```
(defn my-pmap
    ([f coll]
        (let [results (my-reduce #(conj %1 (future (f %2))) [] coll)]
            (my-map deref results)))
    ([f c1 & colls]
        (my-pmap #(apply f %) (reorder (cons c1 colls)))))
```

The behavior of all the functions can be confirmed to be the same as the core functions using property-based tests

http://jonathangraham.github.io twitter: @graham_jp

Understanding Core Clojure Functions

8th Light

Dr. Jonathan Graham

MINED MINDS

http://jonathangraham.github.io twitter: @graham_jp