CS 696 Functional Design & Programming
Fall Semester, 2015
Doc 4 Functions
Sep 3, 2015

REPL State

Lighttable

Restart
Lighttable

REPL

(b 10) 11

(defn b [n] (inc n)) (defn b [n] (inc n))

REPL State

Lighttable

Restart
Lighttable

REPL

(b 10)

11

Compile Error
Can't find b

(defn b [n] (inc n))

(defn b [n] (inc n))

Common Operations on Collections

Combine elements into one result

sum all elements,

min

Transform each element

add 10 to each element

Pass each element as argument to function

Print each element to standard out

Select all elements that meet a condition

all elements greater than 10

Select one elements that meet a condition

First element greater than 10

Group elements by some criteria

group strings by size

Map, Reduce, Filter

Higher order functions

Very important

Map

Apply a function to each element of a collection, return resulting collection

Ruby - collect, map

Smalltalk - collect

Filter

Returns elements of collection that make

Reduce

Applies function

Reduce

(reduce + [1 2 3 4])	10
(reductions + [1 2 3 4])	(1 3 6 10)
(reduce small-add [1 2 3 4 5 6])	6

```
(defn small-add
  [subresult x]
  (if (< x 4)
    (+ subresult x)
    (reduced subresult)))
```

Map

Map - the noun

{:a 1 :c 10}

Map - the verb

(map inc [1 2 3]) (2 3 4)

Map - the Verb

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

(map inc [1 2 3])	(2 3 4)
(map + [1 2 3] [4 5 6])	(5 7 9)
(map + [1 2 3 4 5] [4 5 6])	(5 7 9)
(map inc #{1 2 3})	(2 4 3)
(map + [1 2 3] #{4 5 6})	(5 8 8)

map	Returns lazy sequence
mapv	Returns vector
pmap	Done in parallel, semi-lazy
map-indexed	f gets index & element

map-indexed

```
(map-indexed (fn [index item] {:index index :item item})  [1 2 3])
```

```
({:index 0, :item 1} {:index 1, :item 2} {:index 2, :item 3})
```

pmap

Distributes work among cores, not separate processors/machines

Operation needs to be computationally intense

(time (doall (map inc (range 10000)))) "Elapsed time: 4.73 msecs"

(time (doall (pmap inc (range 10000)))) "Elapsed time: 529.905 msecs"

Parallel Example

```
(defn long-running-job [n]
    (Thread/sleep 3000) ; wait for 3 seconds
    (+ n 10))
```

```
(time (doall (map long-running-job (range 4))))        12.005 secs

(time (doall (map long-running-job (range 8))))        24.005 secs
```

```
(time (doall (pmap long-running-job (range 4))))       3.01 secs

(time (doall (pmap long-running-job (range 8))))       3.01 secs

(time (doall (pmap long-running-job (range 64))))      6.01 secs
```


Since the job is not doing any real work pmap performs very well. It can use multiple threads on one processor and the threads can all perform at the same time

Slightly More Realistic Example

```
(defn long-running-job
  [n]
  (reduce + (take 10000000 (iterate #(Math/sin %) n))))
```

```
(time (doall (map long-running-job (range N))))
(time (doall (pmap long-running-job (range N))))
```

N	map time secs	pmap time secs
2	7.5	4.8
4	15.3	10.1

2.13 GHz Intel Core 2 Duo

Partition Size

One can control the size of data send to each thread

partition-all

filter

(filter even? [1 2 3 4 5 6 7]) (2 4 6)

(remove even? [1 2 3 4 5 6 7]) (1 3 5 7)

(keep even? [1 2 3 4 5 6 7]) (false true false true false true false)

(first (filter even? [1 2 3 4 5 6 7])) 2

(filter #{3 5 9 12} [1 2 3 4 5 6 7]) (3 5)

fliterv returns vector of results instead of lazy sequence

Specialized filter functions

(take-while neg? [-2 -1 0 1 2 3])	(-2 -1)
(take-while neg? [-2 -1 0 -1 -2 3])	(-2 -1)
(drop-while neg? [-1 -2 -6 -7 1 2 3 4 -5 -6 0 1])	(1 2 3 4 -5 -6 0 1)
(split-with #(< % 3) [1 2 3 4 5 1])	[(1 2) (3 4 5 1)]
(split-with pred coll)	[(take-while pred coll) (drop-while pred coll)]

Tests

(every? even? '(2 4 6)) true

(every? even? '(2 4 7)) false

(every? #{1 2} [1 2 1]) true

(some even? '(2 4 7)) true

(some even? '(1 5 7)) nil

not-every?
not-any?

partition

(partition n coll)
(partition n step coll)
(partition n step pad coll)

(partition 4 (range 20))
((0 1 2 3) (4 5 6 7) (8 9 10 11) (12 13 14 15) (16 17 18 19))

(partition 9 (range 20))
((0 1 2 3 4 5 6 7 8) (9 10 11 12 13 14 15 16 17))

(partition 5 3 (range 20))
((0 1 2 3 4) (3 4 5 6 7) (6 7 8 9 10) (9 10 11 12 13) (12 13 14 15 16) (15 16 17 18 19))

(partition 9 9 [1 1 1] (range 20))
((0 1 2 3 4 5 6 7 8) (9 10 11 12 13 14 15 16 17) (18 19 1 1 1))

For

```
(for [x [2 3 4]]
  x)
```

(2 3 4)

```
(for [x [2 3 4]
      y [:a :b]]
  [x y])
```

([2 :a] [2 :b] [3 :a] [3 :b] [4 :a] [4 :b])

```
(for [x [2 4 6]
      y [5 9]]
  (* x y))
```

(10 18 20 36 30 54)

```
(for [x [0 1 2 3 4]
      :let [y (* x 3)]
      :when (even? y)]
  y)
```

(0 6 12)

For - :while & :when

```
(for [x [0 1 2]
      y [0 1 2]
      :when (not= x y)]
  [x y])
```

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

```
(for [x [0 1 2]
      y [0 1 2]
      :while (not= x y)]
  [x y])
```

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

iterate

(take 5 (iterate inc 2)) (2 3 4 5 6)

(take 4 (iterate (partial + 2) 0)) (0 2 4 6)

When Processing Collections Consider Using

map
reduce
filter
for
some
repeatedly
sort-by
keep
take-while
drop-while

Common Operations on Collections

Combine elements into one result	reduce
Transform each element	map
Pass each element as argument to function	for, doseq
Select all elements that meet a condition	filter, take-while, drop-while
Select one elements that meet a condition	(first (filter condition xs))
Group elements by some criteria	group-by, partition-by partition

Read from inside out

```
(defn calculate                    let
      [a b c d]                    ->
  (+ (/ (+ a b)  c) d))            ->>
```

let

Allows you to

- compute partial results
- give results names

Compute average of three numbers

```
(defn average
  [a b c]
  (/ (+ a b c) 3))
```

```
(defn average
  [a b c]
  (let [sum (+ a b c)
        size 3]
    (/ sum size)))
```

Using let

```
(defn calculate
      [a b c d]
   (+ (/ (+ a b)  c) d))
```

```
(defn calculate-2
  [a b c d]
  (let [a+b (+ a b)
        divide-c (/ a+b c)
        plus-d (+ divide-c d)]
    plus-d))
```

-> Threading macro

(-> x)
(-> x form1 … formN)

Inserts x as second element in form1

Then inserts form1 as second element in form2

etc.

-> Example

```
(def c 5)

(-> c
    (+ 3)
    (/ 2)
    (- 1))
```

```
(+ c 3)
(/ 8 2)
(- 4 1)
```

-> Example

```
(def c 5)

(-> c
    (+ 3)
    (/ 2)
    dec)
```

```
(+ c 3)
(/ 8 2)
(dec 4)
```

-> Example

```
(-> "a b c d"
    .toUpperCase
    (.replace "A" "X")
    (.split " ")
    first)
```

```
(.toUpperCase "a b c d")
(.replace "A B C D"  "A" "X")
(.split "X B C D"  " ")
(first {"X", "B", "C", "D"} )
```


From Clojure Docs

-> Example

```
(-> person :employer :address :city)

(def person
    {:name "Mark Volkmann"
     :address {:street "644 Glen Summit"
                       :city "St. Charles"
                       :state "Missouri"
                       :zip 63304}
              :employer {:name "Object Computing, Inc."
                         :address {:street "12140 Woodcrest Dr."
                                   :city "Creve Coeur"
                                   :state "Missouri"
                                   :zip 63141}}})
```


From Clojure Docs

->> Threading macro

(->> x)
(->> x form1 … formN)

Inserts x as last element in form1

Then inserts form1 as last element in form2

etc.

->> Example

(def c 5)

(->> c

(+ 3)

(/ 2)

(- 1))

(+ 3 c)

(/ 2 **8**)

(- 1 **1/4**)

as-> Allow Threading in different locations

```
(as-> 5 c
    (+ 3 c)
    (/ c 2)
    (- c 1))
```

bind 5 to c

```
(+ 3 5)
(/ 8 2)
(- 4 1)
```

bind 8 to c

bind 4 to c

return 3

Multiple lines

```
(defn average
      [a b c]
   (println (str "a is " a)
      (+ 1 3)
      (/ (+ a b c) 3))
```

```
(average 1 2 3)
```

returns 2

prints on standard out

a is 1

Why not use def & multiple lines?

```
(defn average-bad
  [a b c]
  (def sum (+ a b c))
  (def size 3)
  (/ sum size))
```

```
(defn average
  [a b c]
  (let [sum (+ a b c)
        size 3]
    (/ sum size)))
```

(average-bad 1 2 3)	2
sum	6
size	3

(average 1 2 3)	2
sum	Error
size	Error

def defines global names/values

let defines local names/values

Don't use def inside functions

Symbols, Values & Binding

Symbols reference a value

foo & bar are symbols

They are bound to values

```
(def foo "hi")

(def bar (fn [n] (inc n)))
```

Expession	Evaluated Result
foo	"hi"
'foo	foo
bar	fn
(bar 12)	13

Binding & Shadowing

```
→ (def x 1)

  (defn shadow
    [x]
  ● (println "Start function x=" x)
    (let [x 20]
      (println "In let x=" x))
    (println "After let x=" x))

(println "Before function x=" x)
(shadow 10)
(println "After function x=")
```

```
Before function x= 1

Start function x= 10

In let x= 20

After let x= 10

After function x= 1
```

Bindings, Shadowing & Functions

```
(dec 10)

(let [dec "December"
      test (dec 10)]
  test)
```

Compile Error

```
(dec 10)

(def dec "December")

(dec 10)
```

Compile Error

```
(clojure.core/dec 10)
```

```
(def + -)
(+ 4 3)        1
```