CS 696 Functional Programming and Design
Fall Semester, 2015
Doc 5 More Functions
Sept 8, 2015

Variable Number of Arguments

```
(defn variable
  [a b & rest]
  (str "a:" a " b:" b " rest:" rest))
```

(variable 1 2)	"a:1 b:2 rest:"
(variable 1 2 3)	"a:1 b:2 rest:(3)"
(variable 1 2 3 4)	"a:1 b:2 rest:(3 4)"
(variable 1)	Error

Changes

assoc
assoc-in
dissoc
merge
merge-with
update
update-with

assoc

(assoc {:a 1 :b 1 :c 1} :a 2 :d 2)	{:a 2, :b 1, :c 1, :d 2}
(assoc-in [{:a 1} {:a 2}] [0 :a] 5)	[{:a 5} {:a 2}]
(assoc-in [{:a 1} {:a 2}] [2 :a] 5)	[{:a 1} {:a 2} {:a 5}]
(assoc-in [{:a 1} {:a 2}] [9 :a] 5)	Error
(assoc-in {:a {:b 1}} [:a :b] 5)	{:a {:b 5}}
(dissoc {:a 1 :b 2 :c 3} :b)	{:a 1, :c 3}
(dissoc {:a 1 :b 2 :c 3} :c :b)	{:a 1}

update

(update-in [{:a 1} {:a 2}] [0 :a] inc)	[{:a 2} {:a 2}]
(update-in [{:a 1} {:a 2}] [0 :a] + 10)	[{:a 11} {:a 2}]
(update-in [{:a 1} {:a 2}] [2 :a] inc)	Error
(update-in {:a {:b 1}} [:a :b] inc)	{:a {:b 2}}
(update {:a 1 :b 1 :c 1} :a inc)	{:a 2, :b 1, :c 1}
(update {:a 1 :b 1 :c 1} :a + 10 2)	{:a 13, :b 1, :c 1}

merge

```
(merge [1 2]  2 3 4)
```
[1 2 2 3 4]

```
(merge '(1 2) 3 4)
```
(4 3 1 2)

```
(merge {:a 1} {:b 2})
```
{:a 1, :b 2}

```
(merge {:a 1} {:b 2 :a 2})
```
{:a 2, :b 2}

```
(merge-with inc [1 2 3])
```
[1 2 3]

```
(merge-with concat
    {:even [2 4] :odd [1 3]}
    {:even [4 6 8]})
```
{:even (2 4 4 6 8), :odd [1 3]}

```
(merge-with +
        {:a 1  :b 2}
        {:a 9  :b 98 :c 0})
```
{:c 0, :a 10, :b 100}

replace

(replace [10 9 8 7 6] [0 2 4]) [10 8 6]

(replace {2 :two, 4 :four} [4 2 3 4 5 6 2]) [:four :two 3 :four 5 6 :two]

Private Functions

```
(defn factorial
  [n]
  (fact-iter 1 1 n))
```

```
(defn- fact-iter
  [product counter max-count]
  (if (> counter max-count)
    product
    (let [next-product (* counter product)]
      (fact-iter next-product (inc counter) max-count))))
```

Control Structures

Loops

Blocks

Branch

Not what you think

loop recur

```
(defn factorial
  [n]
  (loop [count n accumulator 1]
    (if (zero? count)
      accumulator
      (recur (dec count) (* accumulator count)))))
```

doseq

Same options as for

Returns nil

```
(doseq [x [1 2 3]
        y [1 2 3]]
    (prn [x y]))
```

```
(doseq [x [1 2 3]
            y [1 2 3]
            :when (> x y)]
    (prn [x y]))
```

Block - do

```
(do
    form1
    form2
    …
    formN)
```

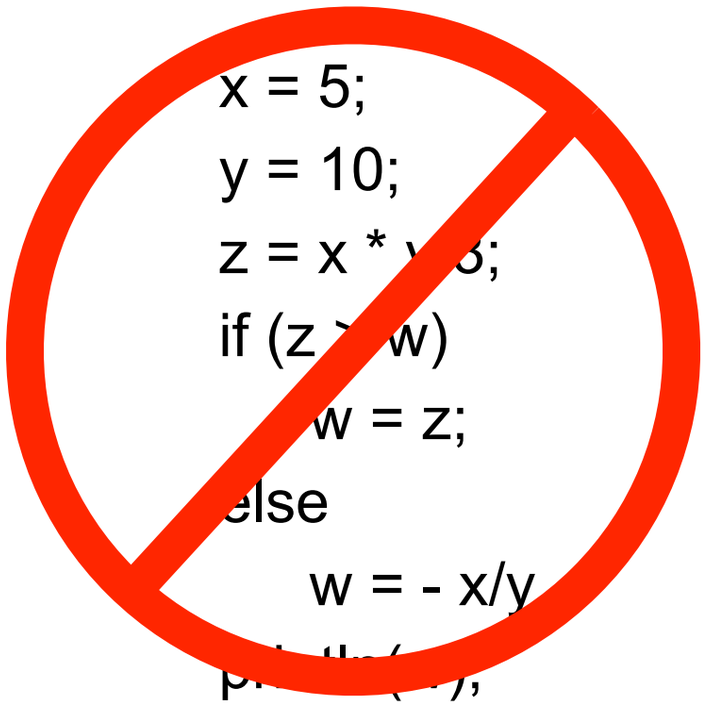
Executes sequence of expressions
Returns the result of last expression

No way to pass results between expressions

```
(do
    (println "starting do")
    (spit "log.txt" "in do")
    (+ 10 x))
```

Used to evaluate forms with side effects
I/O
Setting globals

Execute a sequence of statements?



Can't stack statements

Compose functions

let helps

```
(defn foo
    [x y w]
    (let [z  (/ (* x y ) 3)]
        (println
    (if (> z w)
        z
        (- (/ x y)))))
```

Branching

if
if-not
if-let
if-some
when
when-not
when-let
when-first
when-some
cond
condp

if

(if test then)
(if test then else)

if test is true then execute then

(if-not test then)
(if-not test then else)

if test is true then execute then

if is a form so returns a value

```
(defn middle
  [a b c]
  (if (or (<= a b c) (<= c b a))
    b
    (if (or (<= a c b) (<= b c a))
      c
      a)))
```

(middle 3 1 2) ⟶ 2

Comparing

	(> 3)	true
	(> 8 5)	true
=	(> 8 5 3)	true
==	(> 8 5 3 1)	true
not=	(> 8 5 6 1)	false
<		
>		
<=		
>=		
compare		

```
-1
1
0
0
Error
1
-1
1
-1
-3
-2
```

Tests

nil?	Returns true if the argument is nil, false otherwise
identical?	Tests if the two arguments are the same object
zero?	Returns true if the argument is zero, else false
pos?	Returns true if the argument is greater than zero
neg?	Returns true if the argument is less than zero, else false
even?	Returns true if the argument is even, throws an exception if the argument is not an integer
odd?	Returns true if n is odd, throws an exception if the argument is not an integer
coll?	Returns true if the argument implements IPersistentCollection
seq?	Return true if the argument implements ISeq
vector?	Return true if the argument implements IPersistentVector
list?	Returns true if the argument implements IPersistentList
map?	Return true if the argument implements IPersistentMap
set?	Returns true if the argument implements IPersistentSet
contains?	Returns true if key is present in the given collection, else false
distinct?	Returns true if no two of the arguments are =
empty?	Returns true if the collection argument has no items same as (not (seq coll))

Naming Convention

Tests

- Return true/false
- end in ?

So why not

- compare?

Truthiness

Things that are false

- false
- nil

Things that are true

- Everything else

some

(some predicate collection)
(some pred coll)

Returns first true value of (predicate x) for any x in collection

(some even? [1 2 3])	true
(some even? [1 3 5])	nil
(some #(if (even? %) %) [1 2 3 4])	2
	3
(some {2 "two" 3 "three"} [nil 3 2])	3
(some [2 "two" 3 "three"] [nil 3 2])	IllegalArgumentException

Idiomatic Clojure

Using collections as functions

Very odd to non-clojure programmers

Done a lot

Testing Collections

Is a collection

- nil
- empty
- has elements

```
(empty? nil)          true
(empty? [])           true
(empty? [1 2 3])      false
(seq nil)             nil
(seq [])              nil
(seq [1 2 3])         (1 2 3)
```

if-let

```
(if (not (empty? (rest x)))
   {:value (reduce + (rest x))}
  {:value :empty})
```

```
(let [tail  (rest x)]
  (if (not (empty? tail))
     {:value (reduce + tail)}
    {:value :empty}))
```

```
(let [tail (seq (rest x))]
  (if tail
     {:value (reduce + tail)}
    {:value :empty}))
```

```
(if-let [tail (seq (rest x))]
  {:value (reduce + tail)}
  {:value :empty})
```

```
(if-let [binding-form test]
     then
     else)
```

binding-form = result of test
Then do if on binding-form

if-let

```
(def personA {:name "Roger" :illness "flu"})
(def personB {:name "Roger"})


(defn example
  [person]
  (if-let [disease (:illness person)]
        disease
        "Well"))
```

(example personA) "flu"

(example personB) "Well"

if-some

Added Clojure 1.6
Like if-let
tests for not nilness

```
(if-some [a nil]
  :true
  :false)
```
:false

```
(if-some [a false]
  :true
  :false)
```
:true

```
(if-let [a nil]
  :true
  :false)
```
:false

```
(if-let [a false]
  :true
  :false)
```
:false

when, when-not, when-let, when-some

if with only the true condition
Returns nil when condition is false

```
(when condition
    expresssion1
    expresssion2
    …
    expresssionN)
```

```
(if condition
  (do
    expresssion1
    expresssion2
    …
    expresssionN))
```

```
(when (> x 2)
  4)
```

```
(when (> x 2)
  (println "foo")
  4)
```

```
(when (seq collection)
    ;do something with collection
)
```

Idiomatic Clojure

```
(when (seq collection)
      ;do something with collection
)
```

Body only executed if collection has elements

```
(when (seq [1 2]) :body-executed)          :body-executed

(when (seq []) :body-executed)             nil

(when (seq nil)  :body-executed)           nil
```

when verses if

when is an if without branch

What is the point of when?

cond

```
(defn pos-neg-or-zero
  [n]
  (cond
    (< n 0) "negative"
    (> n 0) (str n "is positive")
    :else "zero"))
```

```
(defn pos-neg
  [n]
  (cond
    (< n 0) "negative"
    (> n 0) "positive"))
```

```
positive
nil
```

Find first condition that is true

Return the result of that condition's expression

condp

(condp function expression
    test-expression1 result-expression1
    …
    test-expressionN result-expressionN
    optional-default)

Return result-expressionK for first K where

    (function test-expressionK expression) evaluates to true

If no such K return default

Runtime exception if no match

Example - With default

```
(defn example
  [value]
  (condp = value
    1 "one"
    2 "two"
    3 "three"
    (str "unexpected value, " value)))
```

```
(example 2)        "two"
(example 9)        "unexpected value, 9"
```

Example - Without default

```
(defn example
  [value]
  (condp = value
    1 "one"
    2 "two"
    3 "three"))
```

```
(example 2)        "two"
(example 9)        IllegalArgumentException
```

condp - Complex version

(condp function expression
    test-expression1 :>> result-fn1
    …
    test-expressionN :>> result-fnN
    optional-default)

Find first (lowest) K where

    (function test-expressionK expression) evaluates to true

    then return (result-fnK function)

If no such K return default

Runtime exception if no match

Lazy Evaluation

```
if (object != null && object.isGreen() ) {
     //do something
}
```

object.isGreen() only evaluated if object not null

Common form of lazy evaluation

Example

Take a sequence and nests the elements

(steps [1 2 3 4])

[1 [2 [3 [4 []]]]]

```
(defn rec-steps
  [[x & xs]]
  (if x
    [x (rec-steps xs)]
    []))
```

(rec-steps (range 2106))

java.lang.StackOverflowError

Using lazy evaluation

```
(defn lazy-rec-steps
  [s]
  (lazy-seq
   (if (seq s)
    [(first s) (lazy-rec-steps (rest s))]
    [])))
```

```
(lazy-rec-steps [1 2 3])                    (1 (2 (3 ())))

(class (lazy-rec-steps [1 2 3]))            clojure.lang.LazySeq

(dorun (lazy-rec-steps (range 1000000)))    nil
```

Lazy Sequences & REPL

When you display a lazy sequence in REPL the entire sequence is evaluated

(lazy-rec-steps (range 3000)) Stack Overflow

This will cause problems

- Stack overflows
- Code that works in REPL not working in program

Works but slow

```
(defn print-seq
 [s]
 (println "start " (first s))
 (if (seq s)
     (recur (first (next s)))))


(print-seq (lazy-rec-steps (range 3000)) )
```

Rules for Lazy

Use lazy-seq at outermost level of lazy squence-producing expression

Use **rest** instead of **next** if consuming another sequece

Use higher-order functions when processing sequences

Don't hold on to the **head**

rest verses next

next has to look at the next element, causing it to be computed

rest does not look at the next element

Example

```
(defn lazy-test
  [n]
  (lazy-seq
   (println "n= " n)
   (if (> n 0)
     (cons n (lazy-test (dec n))))))
```

```
(def example (lazy-test 5))
(def a (rest example))          ;;n= 5
(def b (rest example))
```

```
def example (lazy-test 5))
(def c (next example))          ;;n= 5
                                ;;n= 4
(def d (next example))
```