(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
Bindings, Shadowing & Functions

(decl 10)

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

  Compile Error

(decl 10)

(def dec "December")

(decl 10)

(clojure.core/dec 10)

(def + -)

(+ 4 3) 1
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
reduce

(reduce f coll)  Applies f to coll
(reduce f val coll)

<table>
<thead>
<tr>
<th>(reduce + [1 2 3 4])</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>(reduce + [])</td>
<td>0</td>
</tr>
<tr>
<td>(reduce + 1 [])</td>
<td>1</td>
</tr>
<tr>
<td>(reduce + 1 [2 3])</td>
<td>6</td>
</tr>
<tr>
<td>(reduce + '(1 2 3))</td>
<td>6</td>
</tr>
<tr>
<td>(reduce str [&quot;a&quot; &quot;b&quot; &quot;c&quot;])</td>
<td>&quot;abc&quot;</td>
</tr>
<tr>
<td>(reduce conj #{} [1 2 3])</td>
<td>#{1 3 2}</td>
</tr>
</tbody>
</table>
Better Average

(defn average
    [& numbers]
    (let [sum (reduce + numbers)
           size (count numbers)]
        (if (> size 0)
            (/ sum size)))))

(average)                      nil
(average 1)                     1
(average 1 2)                   3/2
(average 1 2 3 4 5 6)           7/2
But + works on multiple values - Why Reduce?

(+ 1 2 3) 6
(+ [1 2 3]) Error
(reduce + [1 2 3]) 6
(reduce + 1 2 3) Error
Control Structures

Block
Branch
Loops

Not what you think


**Block - do**

\[(\text{do} \quad \text{form1} \quad \text{form2} \quad \ldots \quad \text{formN})\]

- Executes sequence of expressions
- Returns the result of last expression
- No way to pass results between expressions

\[(\text{do} \quad \text{(println "starting do")} \quad \text{(spit "log.txt" "in do")} \quad \text{(+ 10 x))}\]

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

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

if test is true then execute then
if test is true then execute then
if is a form so returns a value

(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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nil?</td>
<td>Returns true if the argument is nil, false otherwise</td>
</tr>
<tr>
<td>identical?</td>
<td>Tests if the two arguments are the same object</td>
</tr>
<tr>
<td>zero?</td>
<td>Returns true if the argument is zero, else false</td>
</tr>
<tr>
<td>pos?</td>
<td>Returns true if the argument is greater than zero</td>
</tr>
<tr>
<td>neg?</td>
<td>Returns true if the argument is less than zero, else false</td>
</tr>
<tr>
<td>even?</td>
<td>Returns true if the argument is even, throws an exception if the argument is not an integer</td>
</tr>
<tr>
<td>odd?</td>
<td>Returns true if n is odd, throws an exception if the argument is not an integer</td>
</tr>
<tr>
<td>coll?</td>
<td>Returns true if the argument implements IPersistentCollection</td>
</tr>
<tr>
<td>seq?</td>
<td>Return true if the argument implements ISeq</td>
</tr>
<tr>
<td>vector?</td>
<td>Return true if the argument implements IPersistentVector</td>
</tr>
<tr>
<td>list?</td>
<td>Returns true if the argument implements IPersistentList</td>
</tr>
<tr>
<td>map?</td>
<td>Return true if the argument implements IPersistentMap</td>
</tr>
<tr>
<td>set?</td>
<td>Returns true if the argument implements IPersistentSet</td>
</tr>
<tr>
<td>contains?</td>
<td>Returns true if key is present in the given collection, else false</td>
</tr>
<tr>
<td>distinct?</td>
<td>Returns true if no two of the arguments are =</td>
</tr>
<tr>
<td>empty?</td>
<td>Returns true if the collection argument has no items same as (not (seq coll))</td>
</tr>
</tbody>
</table>
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
"two" 3 "three") [nil 3 2]) #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

<table>
<thead>
<tr>
<th>Is a collection</th>
<th>(empty? nil)</th>
<th>true</th>
</tr>
</thead>
<tbody>
<tr>
<td>nil</td>
<td>(empty? [])</td>
<td>true</td>
</tr>
<tr>
<td>empty</td>
<td>(empty? [1 2 3])</td>
<td>false</td>
</tr>
<tr>
<td>has elements</td>
<td>(seq nil)</td>
<td>nil</td>
</tr>
<tr>
<td></td>
<td>(seq [])</td>
<td>nil</td>
</tr>
<tr>
<td></td>
<td>(seq [1 2 3])</td>
<td>(1 2 3)</td>
</tr>
</tbody>
</table>
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 :true
  :false)

(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 (> x 2)
  4)

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

(when (seq collection)
  ;do something with collection
)
(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?
(defn pos-neg-or-zero
 [n]
 (cond
   (< n 0) "negative"
   (> n 0) (str n "is positive")
   :else "zero"))

(pos-neg 1)
positive

(pos-neg 0)
il

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 - 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
Loops

loop
for
doseq
For

(for [x (range 2)
    y (range 3)]
  [x y])

Returns a lazy sequence

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

(for [x (range 5)
    y (range 5)
  :while (< y x)]
  [x y])

([1 0] [2 0] [2 1] [3 0] [3 1] [3 2] [4 0] [4 1] [4 2] [4 3])

(for [x [0 1 2 3 4 5]
    :let [y (* 3 x)]
    :when (even? y)]
  y)

(0 6 12)
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]))
Destructuring - Positional

(let [[a b c] (range 5)]
  (println "a b c are: " a b c))

(let [[a b c :as all] [1 2 3 4 5]]
  (println "a b c are:" a b c)
  (println "all is:" all))

(let [[a b c & more :as all] (range 5)]
  (println "a b c are:" a b c)
  (println "more is:" more)
  (println "all is:" all))
Destructuring - Positional

(defn destructuring
    [[a b c & more :as all] z]
    (println "a b c are:" a b c)
    (println "more is:" more)
    (println "all is:" all)
    (println "z is:" z))

(destructuring [1 2 3 4 5] "cat")

a b c are: 1 2 3
more is: (4 5)
all is: [1 2 3 4 5]
z is: cat
Associative Destructuring

(let [{first 0, third 2, last 4} [1 2 3 4 5]]
  [first third last])
Destructuring - Maps

(def guys-name-map {:first-name "Guy" :middle-name "Lewis" :last-name "Steele"})

(let [{l-name :last-name, f-name :first-name} guys-name-map]
  (str f-name " " l-name))

(let [{:keys [last-name first-name]} guys-name-map]
  (str first-name " " last-name))
Destructuring - :keys, :strs, :syms

[{:keys [a b c]} map]  a, b, c get values at keys :a :b :c in map

[{:strs [a b c]} map]  a, b, c get values at keys "a" "b" "c" in map

[{:syms [a b c]} map]  a, b, c get values at keys 'a 'b 'c in map
Destructuring :as - The Entire map

(def guys-name-map {:first-name "Guy" :middle-name "Lewis" :last-name "Steele"})

(let [{l-name :last-name, f-name :first-name :as whole-name} guys-name-map]
  (println f-name " " l-name)
  whole-name)

;; Guy Steele
;;{:first-name "Guy", :middle-name "Lewis", :last-name "Steele"}
Destructuring :or - Default Values

```(def guys-name-map {:first-name "Guy" :middle-name "Lewis"
                     :last-name "Steele"})

(let [{l-name :last-name, title :title,
       :or {title "Mr."} guys-name-map}
   (str title " " f-name " " l-name))```
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