- Threading macro

(- x)
(- x form1 ... formN)

Inserts x as second element in form1

Then inserts form1 as second element in form2

etc.
-\rightarrow \text{ Threading macro}

(-\rightarrow x)
(-\rightarrow x \text{ form1 } \ldots \text{ formN})

 Inserts x as last element in form1

 Then inserts form1 as last element in form2

 etc.
as-> Allow Threading in different locations

(as-> 5 c)  bind 5 to c
(+ 3 c)    (+ 3 5)  bind 8 to c
(/ c 2)    (/ 8 2)  bind 4 to c
(- c 1))   (- 4 1)  return 3
Recursive Function

(defn factorial
  [n]
  (if (= n 1)
    1
    (* n (factorial (dec n)))))

Recursive Process

(factorial 6)
(* 6 (factorial 5))
(* 6 (* 5 (factorial 4)))
(* 6 (* 5 (* 4 (factorial 3))))
(* 6 (* 5 (* 4 (* 3 (factorial 2)))))
(* 6 (* 5 (* 4 (* 3 (* 2 (factorial 1)))))
(* 6 (* 5 (* 4 (* 3 2))))
(* 6 (* 5 (* 4 6)))
(* 6 (* 5 24))
(* 6 120)
720
Recursive Function

(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))))
Order Matters

(declare fact-iter)

(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))))
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))

Thursday, September 11, 14
(defn b [n] (inc n))

(b 10) 11

Lighttable  REPL
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)))))
(defn factorial
  ([n]
   (factorial 1 1 n))

  ([product counter max-count]
   (if (> counter max-count)
    product
    (let [next-product (* counter product)]
      (factorial next-product (inc counter) max-count)))))

Multiple Arities
recur - Tail Recursion

(defn factorial
  ([n]
   (factorial 1 1 n))

  ([product counter max-count]
   (if (> counter max-count)
     product
     (let [next-product (* counter product)]
       (recur next-product (inc counter) max-count))))
Testing recur

(defn recursive-sum [a b]
  (if (= 0 b)
    a
    (recursive-sum (inc a) (dec b))))

(recursive-sum 0 20000)

StackOverflowError

(defn recur-sum [a b]
  (if (= 0 b)
    a
    (recur (inc a) (dec b))))

(recur-sum 0 50000000)

50000000
(defn factorial
    [n]
    (loop [count n accumulator 1]
        (if (zero? count)
            accumulator
            (recur (dec count) (* accumulator count)))))
recur - Iterative Processes only

(defn factorial [n]
  (if (= n 1)
    1
    (* n (recur (dec n))))

Compile Error
Lazy Evaluation

```java
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 nest 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 sequence-producing expression

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

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
(def d (next example)) ;; n= 4
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

<table>
<thead>
<tr>
<th>Expression</th>
<th>Evaluated Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo</td>
<td>&quot;hi&quot;</td>
</tr>
<tr>
<td>'foo</td>
<td>foo</td>
</tr>
<tr>
<td>bar</td>
<td>fn</td>
</tr>
<tr>
<td>(bar 12)</td>
<td>13</td>
</tr>
</tbody>
</table>
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=")
## Bindings, Shadowing & Functions

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

(def dec "December")

(clojure.core/dec 10)

(def + -)
(+ 4 3) 1
```
(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>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>(reduce + [1 2 3 4])</td>
<td>10</td>
</tr>
<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

(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))))))

x = 5;
y = 10;
z = x * y/3;
if (z > w)
  w = z;
else
  w = - x/y
println(w);
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

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&gt; 3)</td>
<td>true</td>
</tr>
<tr>
<td>(&gt; 8 5)</td>
<td>true</td>
</tr>
<tr>
<td>(=)</td>
<td>true</td>
</tr>
<tr>
<td>(&gt; 8 5 3)</td>
<td>true</td>
</tr>
<tr>
<td>(==)</td>
<td>true</td>
</tr>
<tr>
<td>(&gt; 8 5 3 1)</td>
<td>true</td>
</tr>
<tr>
<td>(not=)</td>
<td>false</td>
</tr>
<tr>
<td>(&gt; 8 5 6 1)</td>
<td>false</td>
</tr>
<tr>
<td>(&lt;)</td>
<td>1</td>
</tr>
<tr>
<td>(&gt;)</td>
<td>-1</td>
</tr>
<tr>
<td>(&lt;=)</td>
<td>0</td>
</tr>
<tr>
<td>(&gt;=)</td>
<td>0</td>
</tr>
<tr>
<td>(compare [1 2 3] [1 2 3])</td>
<td>0</td>
</tr>
<tr>
<td>(compare [1 2 3] #{1 2 3})</td>
<td>Error</td>
</tr>
<tr>
<td>(compare [1 2 3] [0 4 5])</td>
<td>1</td>
</tr>
<tr>
<td>(compare [1 2 3] [0 4 5 6])</td>
<td>-1</td>
</tr>
<tr>
<td>(compare [1 2 3] nil)</td>
<td>1</td>
</tr>
<tr>
<td>(compare nil [1 2 3])</td>
<td>-1</td>
</tr>
<tr>
<td>(compare &quot;abc&quot; &quot;def&quot;)</td>
<td>-3</td>
</tr>
<tr>
<td>(compare &quot;abc&quot; &quot;abe&quot;)</td>
<td>-2</td>
</tr>
<tr>
<td>Test</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>-------------</td>
</tr>
<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]) 
(some {2 "two" 3 "three"} [nil 3 2]) 3
(some [2 "two" 3 "three"] [nil 3 2]) IllegalArgumentException

Thursday, September 11, 14
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

\[
\text{(if (not (empty? (rest x)))
  \{value (reduce + (rest x))\}
  \{value :empty\})} \\
\text{(if-let [tail (seq (rest x))]}
  \{value (reduce + tail)}
  \{value :empty\})
\]

\[
\text{(let [tail (rest x)]
  (if (not (empty? tail))
    \{value (reduce + tail)}
    \{value :empty\}))} \\
\text{(if-let [binding-form test]}
  \text{then}
  \text{else)}
\]

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

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)

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

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

(if-let [a false]
:true
: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 condition
  expression1
  expression2
  ...
  expressionN)

(if condition
  (do
    expression1
    expression2
    ...
    expressionN))
(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?