CS 596 Functional Programming and Design
Fall Semester, 2014
Doc 5 More Functions
Sept 11, 2014

-> Threading macro

(-> x)
(-> x form1 … formN)

Inserts x as second element in form1

Then inserts form1 as second element in form2

etc.

->> Threading macro

(->> x)
(->> x form1 … 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 1)))))
(* 6 (* 5  (* 4 (* 3 2))))
(* 6 (* 5  (* 4 6)))
(* 6 (* 5  24))
(* 6 120)
720
```

Recursive Function

Iterative Process

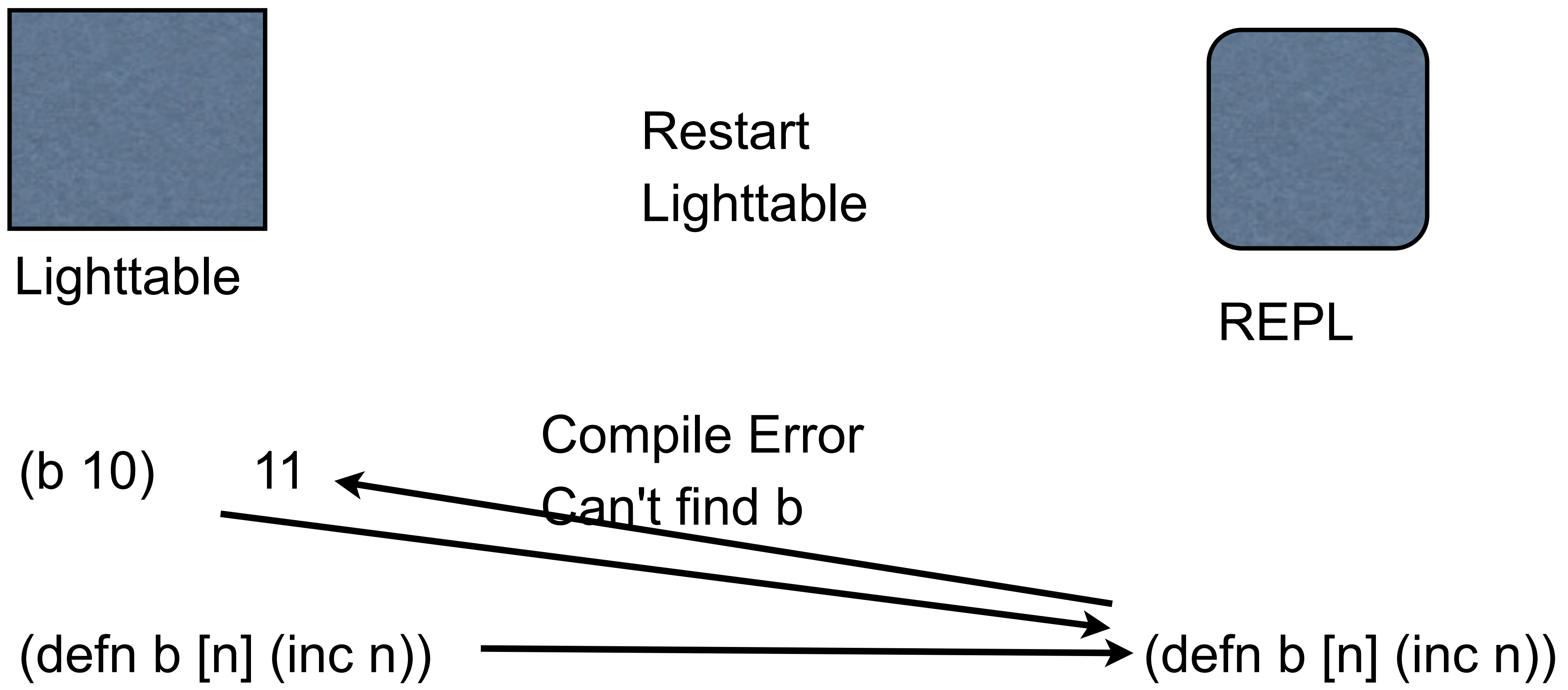
```
(defn factorial
  [n]
  (fact-iter 1 1 n))


(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))))
```

```
(factorial 6)
(fact-iter  1  1  6)
(fact-iter  1  2  6)
(fact-iter  2  3  6)
(fact-iter  6  4  6)
(fact-iter  24  5  6)
(fact-iter  120  6  6)
(fact-iter  720  7  6)
720
```

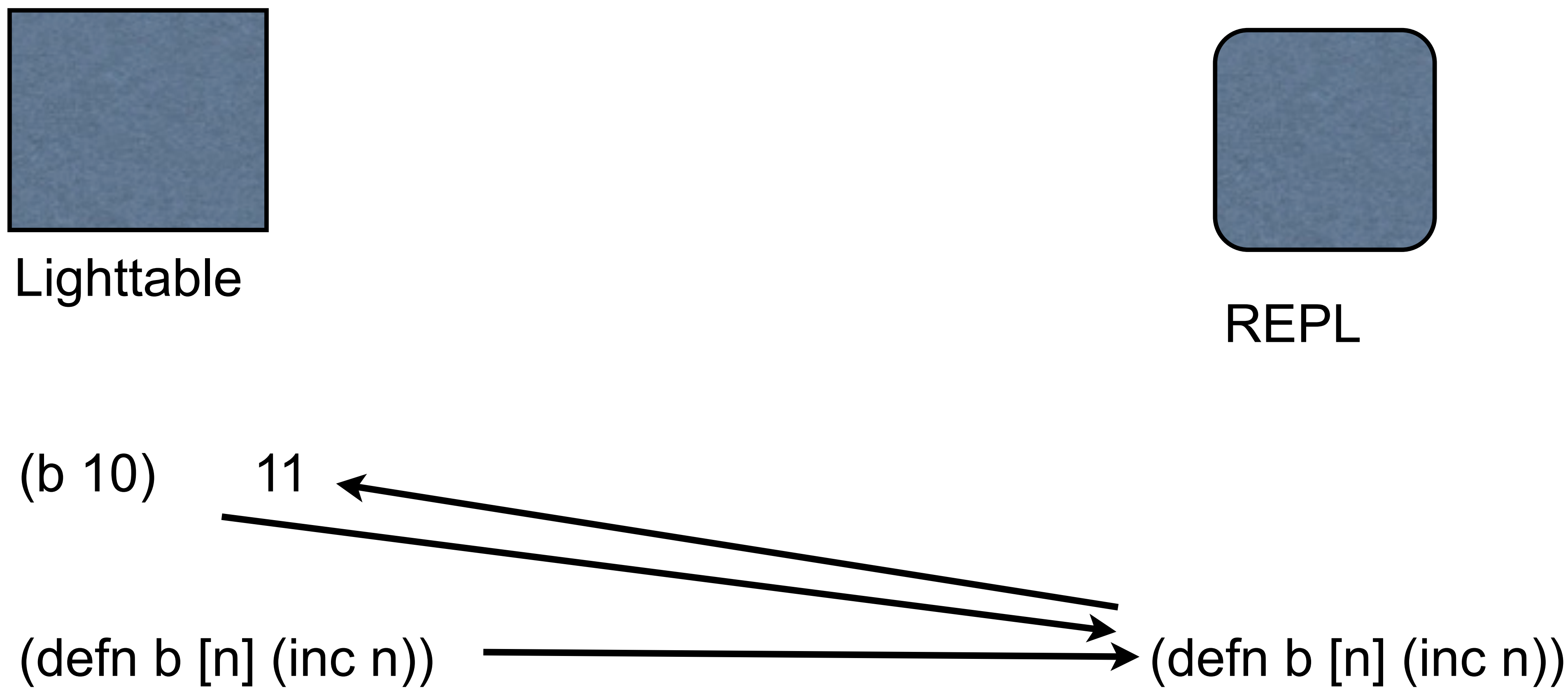
Order Matters

```
(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
Compile Error
Can't find b
(b 10)
11
(defn b [n] (inc n))
(defn b [n] (inc n))

REPL State



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))))
```

Multiple Arities

```
(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)))))
```

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

loop recur

```
(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

```
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))
```

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))
```

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

def defines global names/values

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

(average 1 2 3)	2
sum	Error
size	Error

let defines local names/values

Don't use def inside functions

Symbols, Values & Binding

Symbols reference a value

(def foo "hi")

foo & bar are symbols

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

They are bound to values

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

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)
(reduce f val coll)

Applies f to coll

(reduce + [1 2 3 4])	10
(reduce + [])	0
(reduce + 1 [])	1
(reduce + 1 [2 3])	6
(reduce + '(1 2 3))	6
(reduce str ["a" "b" "c"])	"abc"
(reduce conj #{} [1 2 3])	#{1 3 2}

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?

```
x = 5;
y = 10;
z = x * [illegible]3;
if (z [illegible] w)
    w = z;
[illegible]lse
    w = - x/y
p[illegible]ln([illegible]);
```

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

```
(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 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
(> 8 5 6 1)	false

- =
- ==
- not=
- <
- >
- <=
- >=
- 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
#{2 "two" 3 "three"} [nil 3 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                (if condition
    expresssion1                   (do
    expresssion2      <-->        expresssion1
    …                             expresssion2
    expresssionN)                 …
                                  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?