relevance

clojure

Aaron Bedra
Principal, Relevance Inc.

clojure's four elevators

java interop

lisp

functional

state

1. java interop

java new

java	`new Widget("foo")`
clojure	`(new Widget "foo")`
clojure sugar	`(Widget. "red")`

access static members

java	`Math.PI`
clojure	`(. Math PI)`
clojure sugar	`Math.PI`

access instance members

java	`rnd.nextInt()`
clojure	`(. rnd nextInt)`
clojure sugar	`(.nextInt rnd)`

chaining access

java	`person.getAddress().getZipCode()`
clojure	`(. (. person getAddress) getZipCode)`
clojure sugar	`(.. person getAddress getZipCode)`

parenthesis count

java	( ) ( ) ( ) ( )
clojure	( ) ( ) ( )

atomic data types

type	example	java equivalent
string	`"foo"`	String
character	`\f`	Character
regex	`#"fo*"`	Pattern
a. p. integer	`42`	Integer/Long/BigInteger
double	`3.14159`	Double
a.p. double	`3.14159M`	BigDecimal
boolean	`true`	Boolean
nil	`nil`	`null`
symbol	`foo, +`	N/A
keyword	`:foo, ::foo`	N/A

example:
refactor apache commons isBlank

initial implementation

```
public class StringUtils {
  public static boolean isBlank(String str) {
    int strLen;
    if (str == null || (strLen = str.length()) == 0) {
      return true;
    }
    for (int i = 0; i < strLen; i++) {
      if ((Character.isWhitespace(str.charAt(i)) == false)) {
        return false;
      }
    }
    return true;
  }
}
```

- type decls

```
public class StringUtils {
  public isBlank(str) {
    if (str == null || (strLen = str.length()) == 0) {
      return true;
    }
    for (i = 0; i < strLen; i++) {
      if ((Character.isWhitespace(str.charAt(i)) == false)) {
        return false;
      }
    }
    return true;
  }
}
```

- class

```
public isBlank(str) {
  if (str == null || (strLen = str.length()) == 0) {
    return true;
  }
  for (i = 0; i < strLen; i++) {
    if ((Character.isWhitespace(str.charAt(i)) == false)) {
      return false;
    }
  }
  return true;
}
```

+ higher-order function

```
public isBlank(str) {
  if (str == null || (strLen = str.length()) == 0) {
    return true;
  }
  every (ch in str) {
    Character.isWhitespace(ch);
  }
  return true;
}
```

- corner cases

```
public isBlank(str) {
  every (ch in str) {
    Character.isWhitespace(ch);
  }
}
```

lispify

```
(defn blank? [s]
  (every? #(Character/isWhitespace %) s))
```

clojure is a better java than java

macros capture

common idioms

repeated this

```
(.add frame panel)
(.pack frame)
(.setVisible frame true)
```

repeated this: **doto**

```
(doto frame
  (.add panel)
  (.pack)
  (.setVisible true))
```

say it only once

resource cleanup

```
(let [x (FileInputStream. "datafile")]
  (try
   #_ (... do something with x ...)
  (finally
   (.close x))))
```

resource cleanup: with-open

```
(with-open [x (FileInputStream. "datafile")]
  #_ (... do something with x ...))
```

letting java

call you

implement interface

base class, interfaces

base class cons args

```
(let [r (proxy [Runnable] []
          (run [] (println "Hello")))]
  (doto (Thread. r) (.run)))
```

method bodies

prefer reify (clojure 1.2)

interface

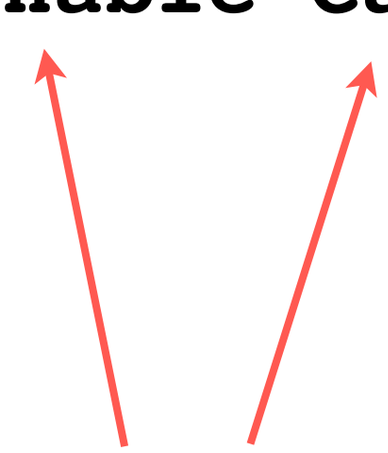
```
(reify Runnable
       (run [] (println "Hello")))
```

method bodies

add more interfaces here

don't need a method?
skip it!

```
(reify Runnable Callable)
```



implements two interfaces, but
all methods will throw
AbstractMethodError

2. lisp

what makes lisp different

feature	industry norm	cool kids	clojure
conditionals	✔	✔	✔
variables	✔	✔	✔
garbage collection	✔	✔	✔
recursion	✔	✔	✔
function type		✔	✔
symbol type		✔	✔
whole language available		✔	✔
everything's an expression			✔
homoiconicity			✔

http://www.paulgraham.com/diff.html

regular code

```
foo.bar(x,y,z);
```

```
foo.bar x y z
```

special forms

imports

scopes

protection

metadata

control flow

anything using a keyword

outside lisp, special forms

look different

may have special semantics unavailable to you

prevent reuse

in a lisp, special forms

look just like anything else

may have special semantics **available** to you

can be augmented with macros

all forms created equal

form	syntax	example
function	list	`(println "hello")`
operator	list	`(+ 1 2)`
method call	list	`(.trim " hello ")`
import	list	`(require 'mylib)`
metadata	list	`(with-meta obj m)`
control flow	list	`(when valid? (proceed))`
scope	list	`(dosync (alter ...))`

clojure is turning the tide in a fifty-year struggle against bloat

3. functional

data literals

type	properties	example
list	singly-linked, insert at front	`(1 2 3)`
vector	indexed, insert at rear	`[1 2 3]`
map	key/value	`{:a 100` `:b 90}`
set	key	`#{:a :b}`

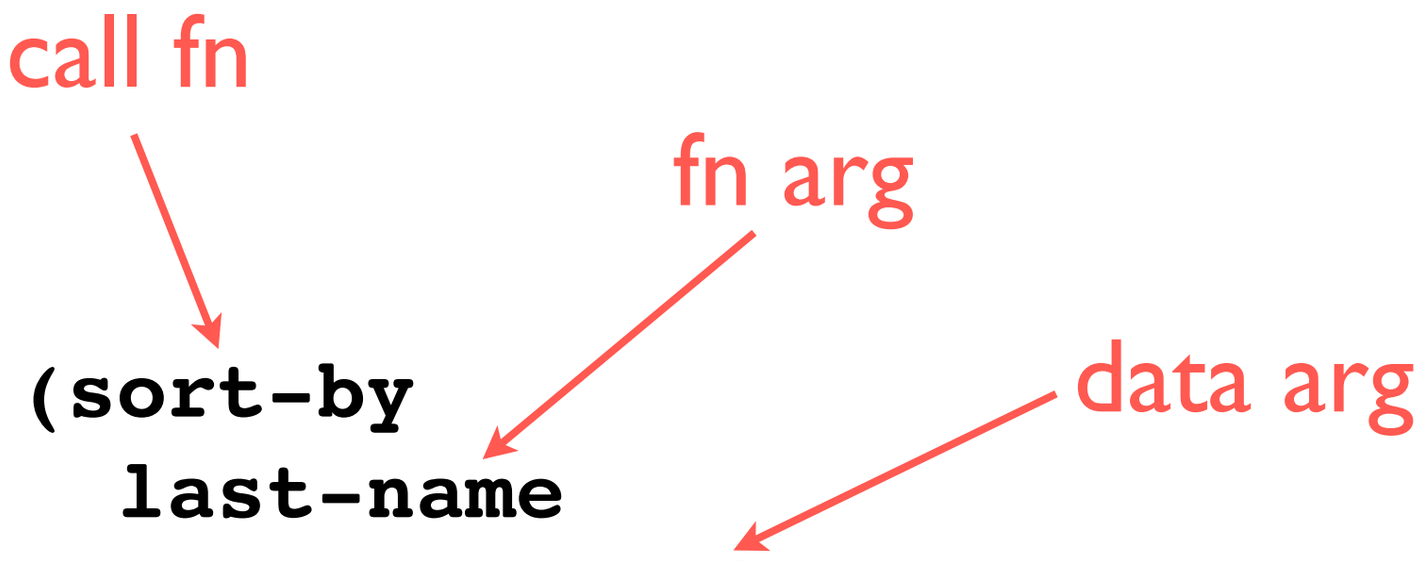
higher-order functions

some data

```
lunch-companions
-> ({:fname "Neal", :lname "Ford"}
    {:fname "Stu", :lname "Halloway"}
    {:fname "Dan", :lname "North"})
```

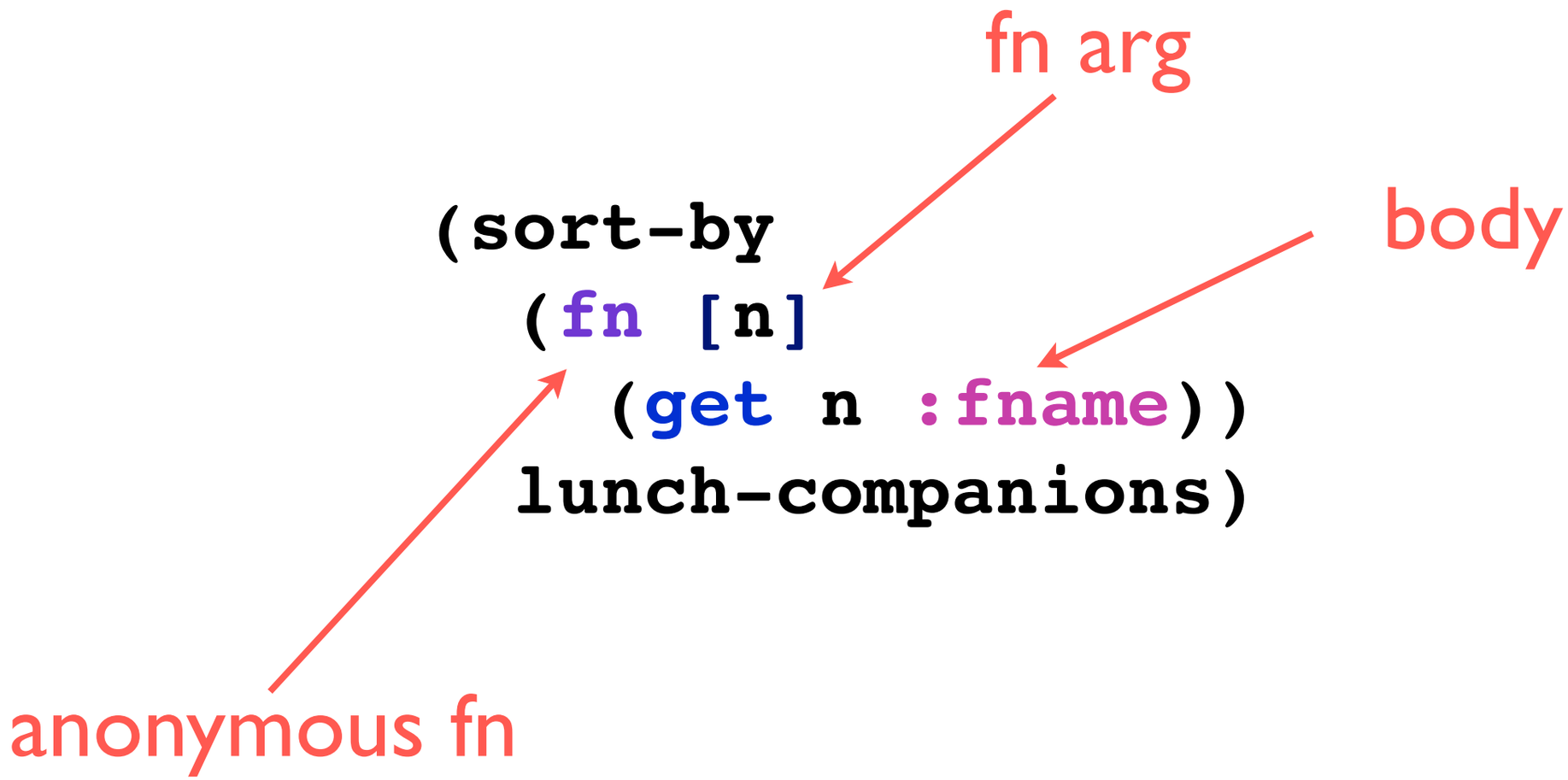
“getter” function

pass fn to fn




```
(sort-by
   last-name
   lunch-companions)
-> ({:fname "Dan", :lname "North"}
    {:fname "Neal", :lname "Ford"}
    {:fname "Stu", :lname "Halloway"})
```

anonymous fn



anonymous #()

fn arg

```
(sort-by
   #(get %  :fname)
   lunch-companions)
```

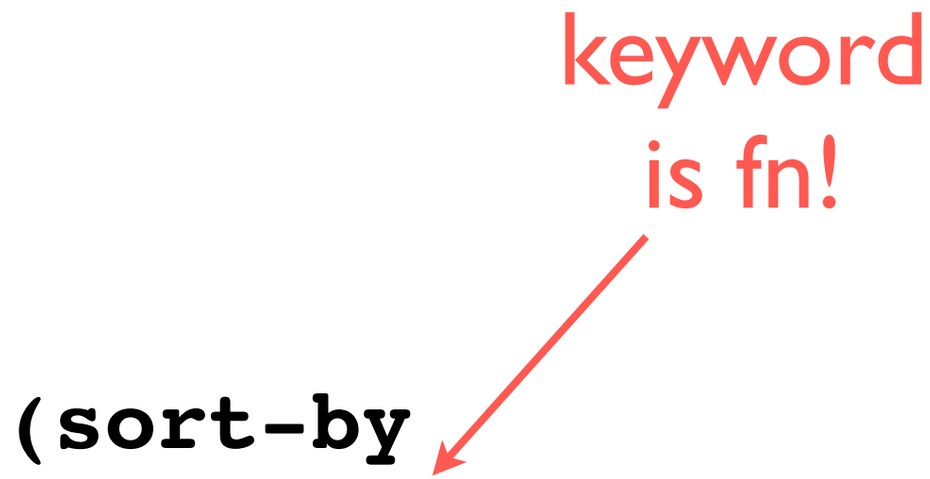
anonymous fn

maps are functions

map is fn!

```
(sort-by
   #(% :fname)
lunch-companions)
```

keywords are functions




```
(sort-by
  #(:fname %)
  lunch-companions)
```

beautiful

```
(sort-by :fname lunch-companions)
```

good implementations have a 1-1 ratio of pseudocode/code

persistent data structures

persistent data structures

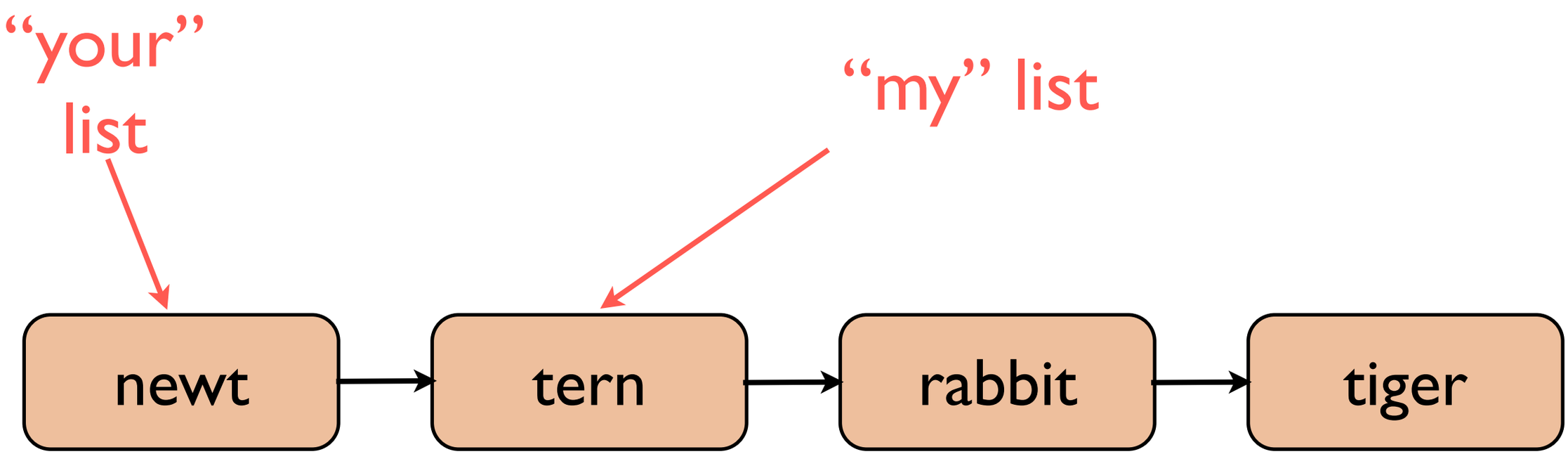
immutable

“change” by function application

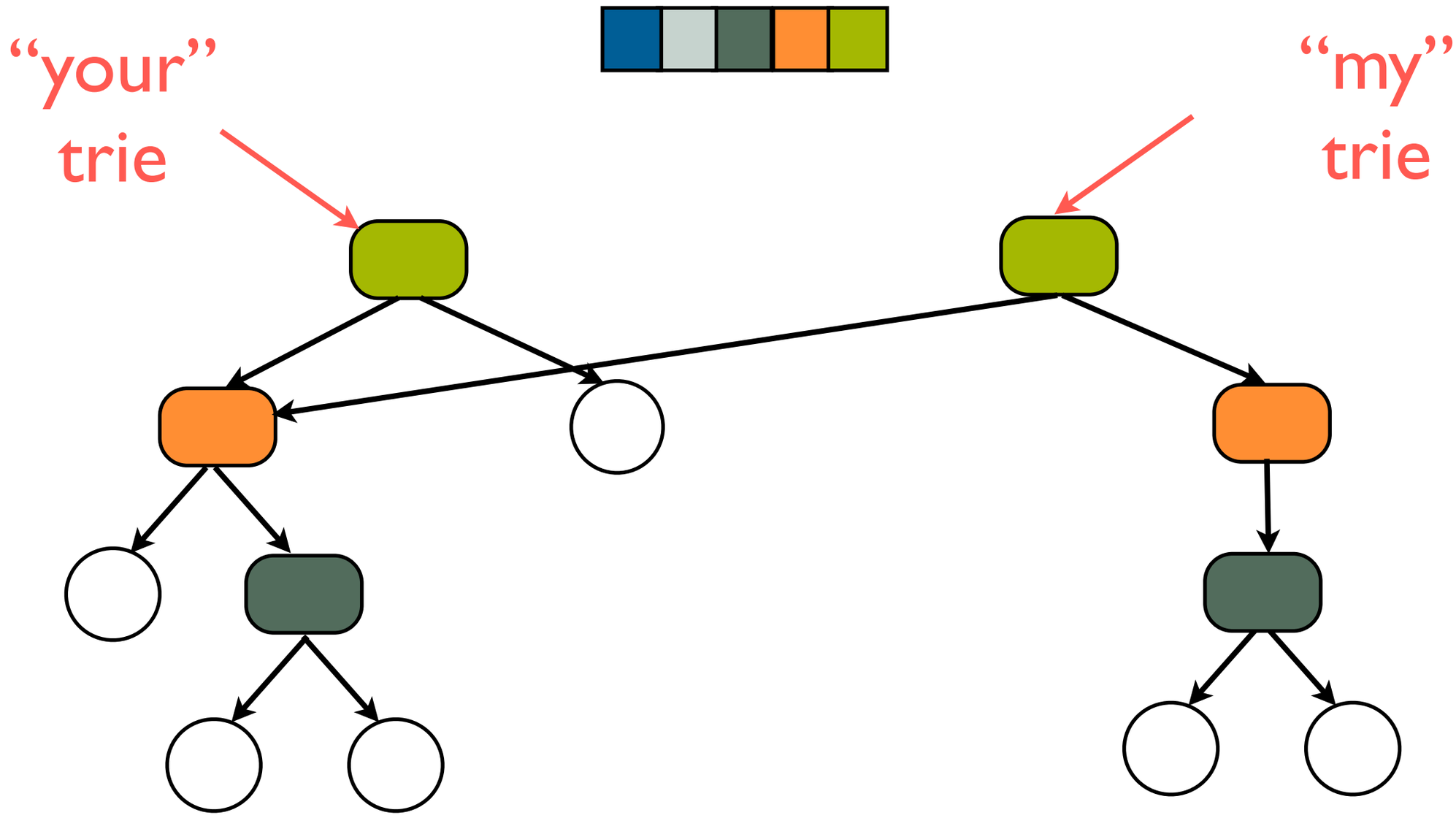
maintain performance guarantees

full-fidelity old versions

persistent example: linked list



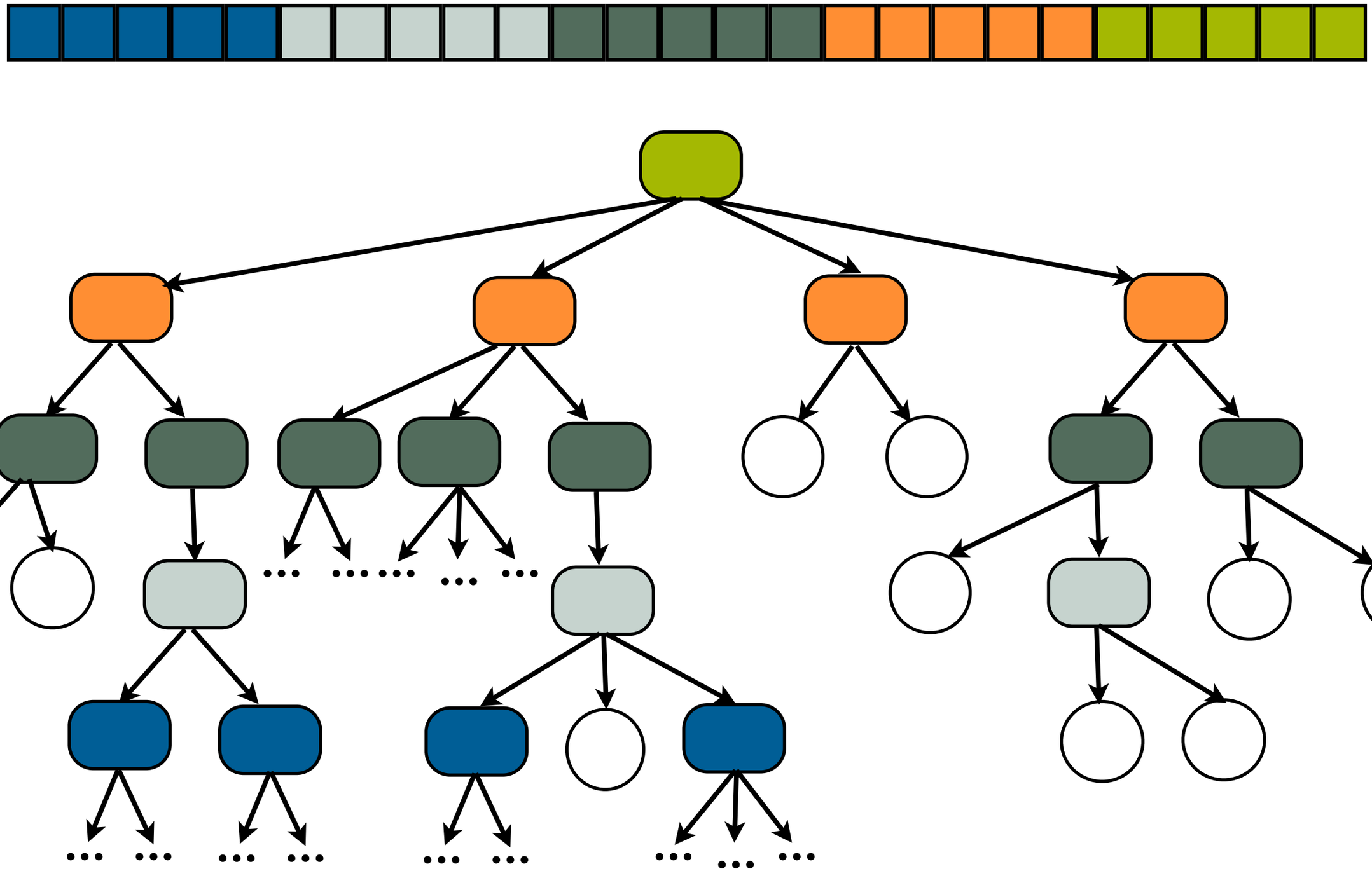
bit-partitioned tries



log2 n:

too slow!

32-way tries



clojure: ‘cause log32 n is fast enough!

sequence library

first / rest / cons

```
(first [1 2 3])
-> 1

(rest [1 2 3])
-> (2 3)

(cons "hello" [1 2 3])
-> ("hello" 1 2 3)
```

take / drop

```
(take 2 [1 2 3 4 5])
-> (1 2)
```

```
(drop 2 [1 2 3 4 5])
-> (3 4 5)
```

map / filter / reduce

```
(range 10)
-> (0 1 2 3 4 5 6 7 8 9)

(filter odd? (range 10))
-> (1 3 5 7 9)

(map odd? (range 10))
-> (false true false true false true
false true false true)

(reduce + (range 10))
-> 45
```

sort

```
(sort [ 1 56 2 23 45 34 6 43])
-> (1 2 6 23 34 43 45 56)

(sort > [ 1 56 2 23 45 34 6 43])
-> (56 45 43 34 23 6 2 1)

(sort-by #(.length %)
  ["the" "quick" "brown" "fox"])
-> ("the" "fox" "quick" "brown")
```

conj / into

```
(conj '(1 2 3) :a)
-> (:a 1 2 3)

(into '(1 2 3) '(:a :b :c))
-> (:c :b :a 1 2 3)

(conj [1 2 3] :a)
-> [1 2 3 :a]

(into [1 2 3] [:a :b :c])
-> [1 2 3 :a :b :c]
```

lazy, infinite sequences

```
(set! *print-length* 5)
-> 5

(iterate inc 0)
-> (0 1 2 3 4 ...)

(cycle [1 2])
-> (1 2 1 2 1 ...)

(repeat :d)
-> (:d :d :d :d :d ...)
```

interpose

```
(interpose \, ["list" "of" "words"])
-> ("list" \, "of" \, "words")

(apply str
  (interpose \, ["list" "of" "words"]))
-> "list,of,words"

(use 'clojure.contrib.str-utils)
(str-join \, ["list" "of" "words"]))
-> "list,of,words"
```

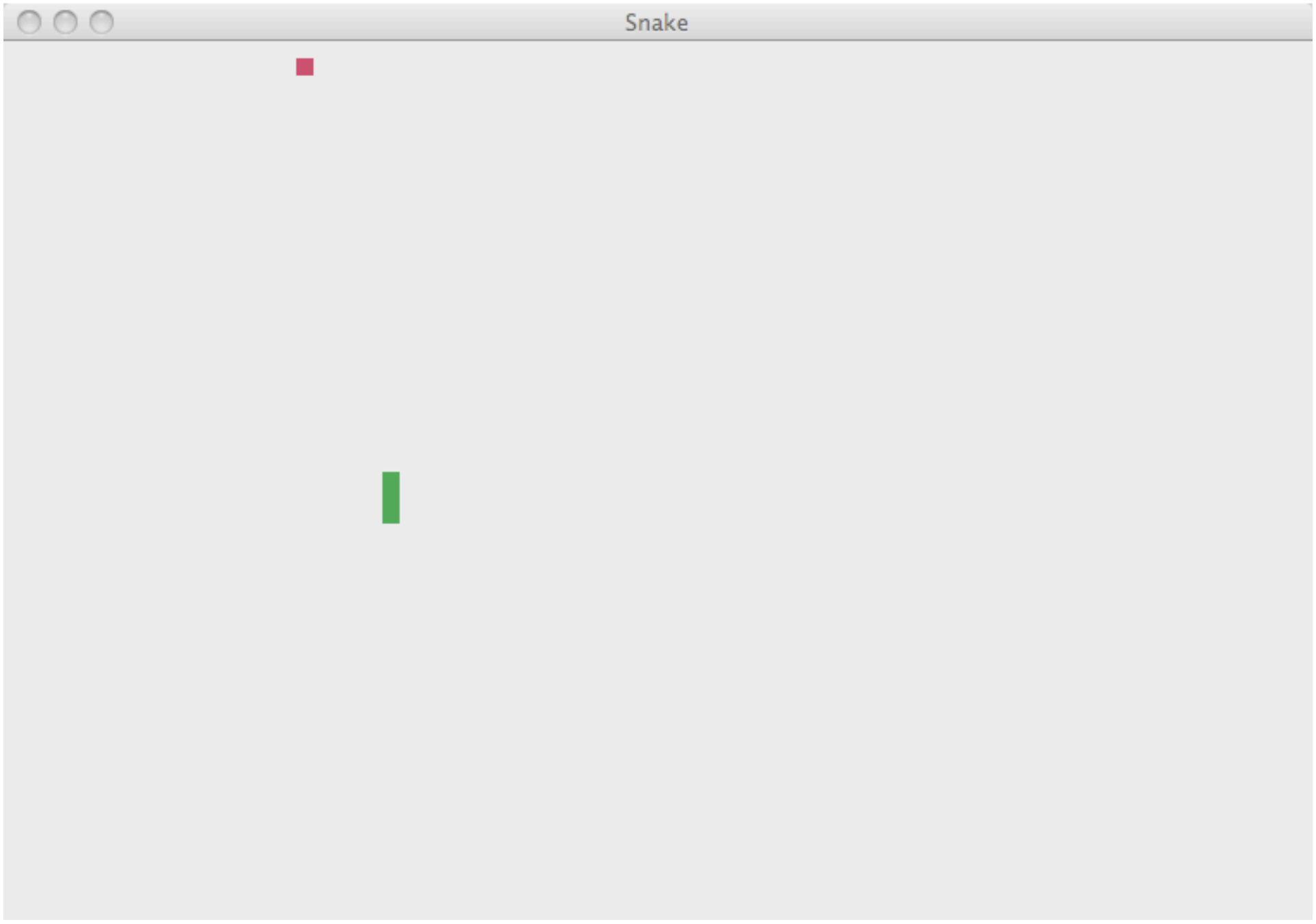
predicates

```
(every? odd? [1 3 5])
-> true

(not-every? even? [2 3 4])
-> true

(not-any? zero? [1 2 3])
-> true

(some nil? [1 nil 2])
-> true
```

nested ops

```
(def jdoe {:name "John Doe",
           :address {:zip 27705, ...}})

(get-in jdoe [:address :zip])
-> 27705

(assoc-in jdoe [:address :zip] 27514)
-> {:name "John Doe", :address {:zip 27514}}

(update-in jdoe [:address :zip] inc)
-> {:name "John Doe", :address {:zip 27706}}
```

destructuring



Sample Code:
http://github.com/stuarthalloway/programming-clojure

early impl:
a snake
is a sequence
of points

first point is
head

```
(defn describe [snake]
  (println "head is " (first snake))
  (println "tail is" (rest snake)))
```

rest is tail

destructure
first element
into `head`

capture
remainder as a
sequence

```
(defn describe [[head & tail]]
  (println "head is " head)
  (println "tail is" tail))
```

destructure remaining
elements into `tail`

snake is more than location

```
(defn create-snake []
  {:body (list [1 1])
   :dir [1 0]
   :type :snake
   :color (Color. 15 160 70)})
```

read attributes

```
(defn describe-snake
 [snake]
 (println "Color is " (:color snake))
 (println "Direction is " (:dir snake))
 (println "Body is " (:body snake)))
```

keyword
attribute
lookup

destructure attributes

let multiple names with map literal

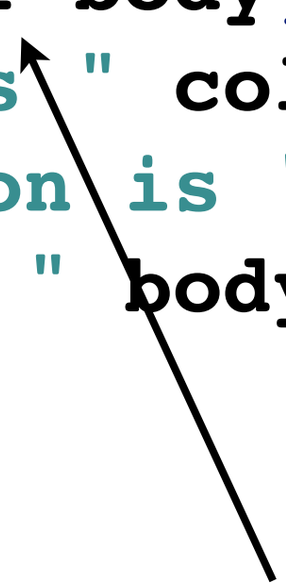
```
(defn describe-snake
    [snake]
    (let [{color :color
           dir :dir
           body :body} snake]
      (println "Color is " color)
      (println "Direction is " dir)
      (println "Body is " body)))
```

simplify with :keys

```
(defn describe-snake
  [snake]
  (let [{:keys [dir color body]}
        snake]
    (println "Color is " color)
    (println "Direction is " dir)
    (println "Body is " body)))
```

destructure in arglist (?)

```
(defn describe-snake
  [{:keys [dir color body]}]
  (println "Color is " color)
  (println "Direction is " dir)
  (println "Body is " body))
```



"I don't care if you are a snake, so long as you have dir, color, and body!"

nesting destructures

2. nested destructure
to pull head and tail from the
:body value

```
(defn describe [{[head & tail] :body}]
  (println "head is " head)
  (println "tail is" tail))
```

1. destructure map,
looking up the `:tail`

losing the game

```
(defn lose? [{[head & tail] :body}]
  (includes? tail head))
```

better

```
(defn lose? [snake]
  (let [{[head & tail] :body} snake]
    (includes? tail head)))
```

where are we?

1. java interop
2. lisp
3. functional

does it work?

example:
refactor apache
commons
indexOfAny

indexOfAny behavior

```
StringUtils.indexOfAny(null, *)                = -1
StringUtils.indexOfAny("", *)                  = -1
StringUtils.indexOfAny(*, null)                = -1
StringUtils.indexOfAny(*, [])                  = -1
StringUtils.indexOfAny("zzabyycdxx",['z','a']) = 0
StringUtils.indexOfAny("zzabyycdxx",['b','y']) = 3
StringUtils.indexOfAny("aba", ['z'])           = -1
```

indexOfAny impl

```
// From Apache Commons Lang, http://commons.apache.org/lang/
public static int indexOfAny(String str, char[] searchChars)
{
  if (isEmpty(str) || ArrayUtils.isEmpty(searchChars)) {
    return -1;
  }
  for (int i = 0; i < str.length(); i++) {
    char ch = str.charAt(i);
    for (int j = 0; j < searchChars.length; j++) {
      if (searchChars[j] == ch) {
        return i;
      }
    }
  }
  return -1;
}
```

simplify corner cases

```
public static int indexOfAny(String str, char[] searchChars)
{
  when (searchChars)
    for (int i = 0; i < str.length(); i++) {
      char ch = str.charAt(i);
      for (int j = 0; j < searchChars.length; j++) {
        if (searchChars[j] == ch) {
          return i;
        }
      }
    }
  }
}
```

- type decls

```
indexOfAny(str, searchChars) {
  when (searchChars)
    for (i = 0; i < str.length(); i++) {
      ch = str.charAt(i);
      for (j = 0; j < searchChars.length; j++) {
        if (searchChars[j] == ch) {
          return i;
        }
      }
    }
  }
}
```

+ when clause

```
indexOfAny(str, searchChars) {
  when (searchChars)
    for (i = 0; i < str.length(); i++) {
      ch = str.charAt(i);
      when searchChars(ch) i;
    }
  }
}
```

+ comprehension

```
indexOfAny(str, searchChars) {
  when (searchChars)
    for ([i, ch] in indexed(str)) {
      when searchChars(ch) i;
    }
  }
}
```

lispify!

```
(defn index-filter [pred coll]
  (when pred
    (for [[idx elt] (indexed coll) :when (pred elt)] idx)))
```

functional

is

simpler

	imperative	**functional**
functions	1	1
classes	1	0
internal exit points	2	0
variables	3	0
branches	4	0
boolean ops	1	0
function calls*	6	3
total	***18***	***4***

functional

is

more general!

reusing index-filter

```
; idxs of heads in stream of coin flips
(index-filter #{:h}
[:t :t :h :t :h :t :t :t :h :h])
-> (2 4 8 9)

; Fibonaccis pass 1000 at n=17
(first
  (index-filter #(> % 1000) (fibo)))
-> 17
```

imperative	**functional**
searches strings	searches *any sequence*
matches characters	matches *any predicate*
returns first match	returns *lazy seq of all matches*

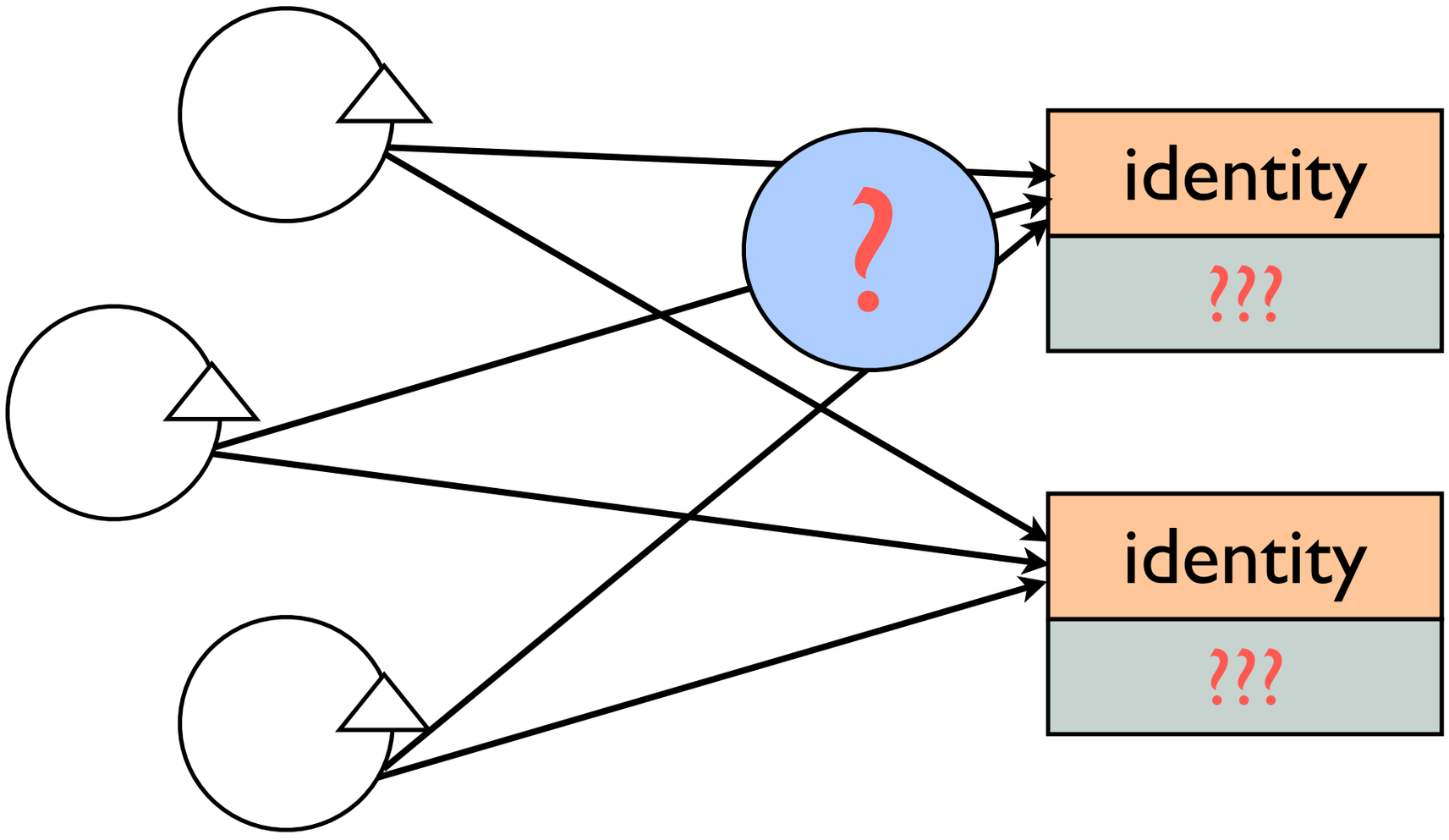
fp reduces incidental complexity by an order of magnitude

4. concurrency

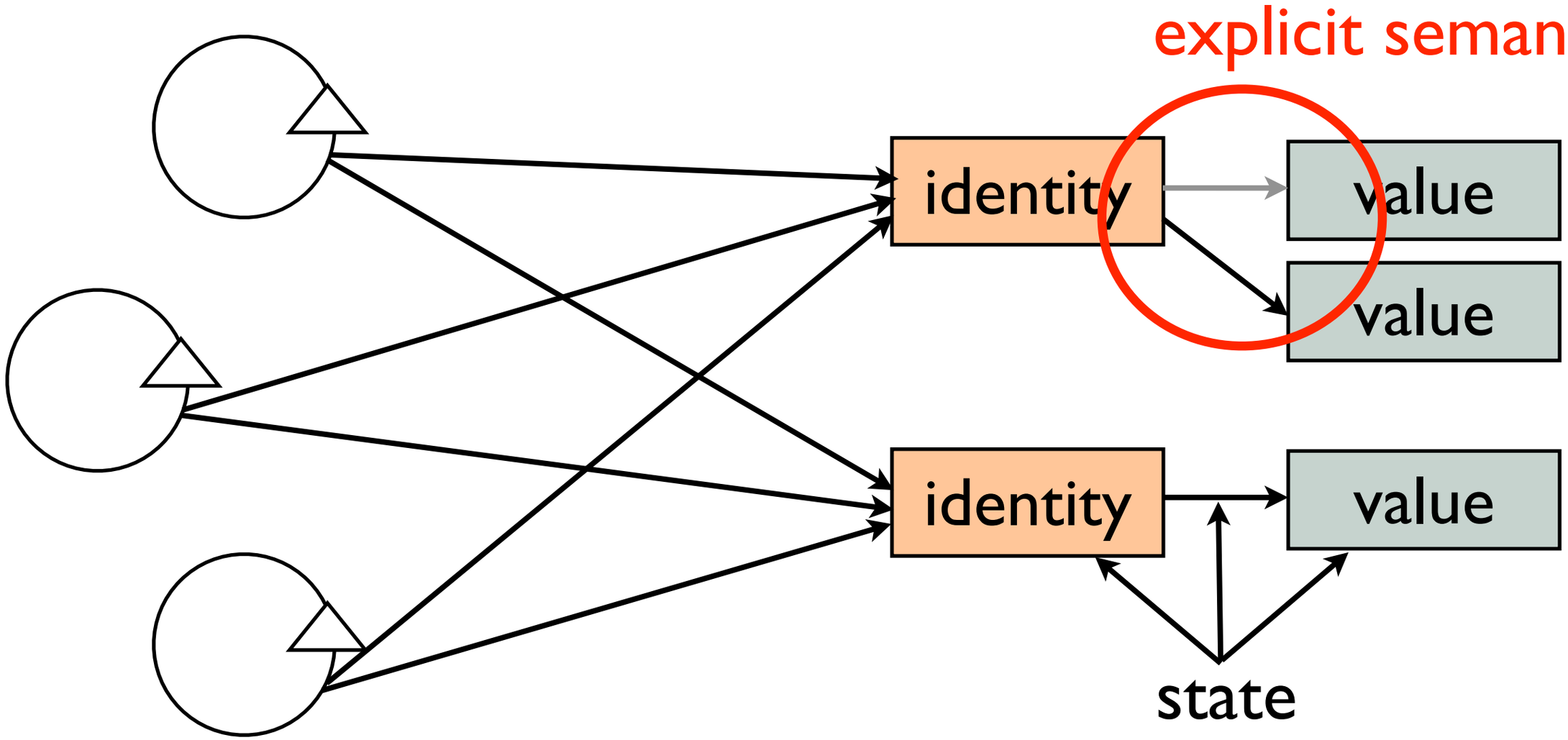
4. state

~~concurrency~~

mutable oo is incoherent



clojure


explicit semantic
identity
value
value
identity
value
state

terms

1. value: immutable data in a persistent data structure

2. identity: series of causally related values over time

3. state: identity at a point in time

identity types (references)

	shared	isolated
synchronous/ coordinated	**refs/stm**	-
synchronous/ autonomous	**atoms**	**vars**
asynchronous/ autonomous	**agents**	-

identity 1 :

refs and stm

ref example: chat

identity

```
(def messages (ref ()))
```

initial value

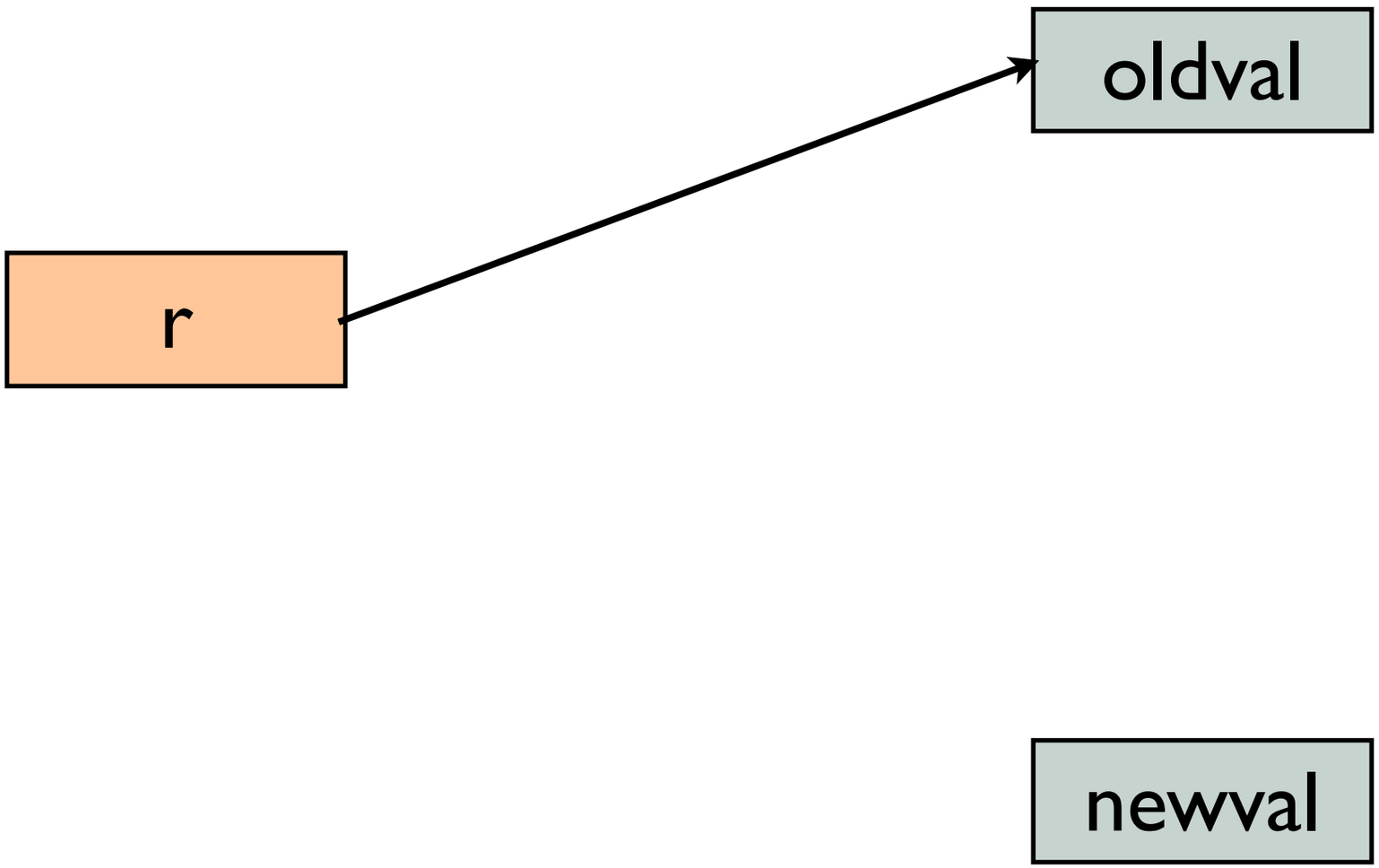
reading value

```
(deref messages)
=> ()
```

```
@messages
=> ()
```

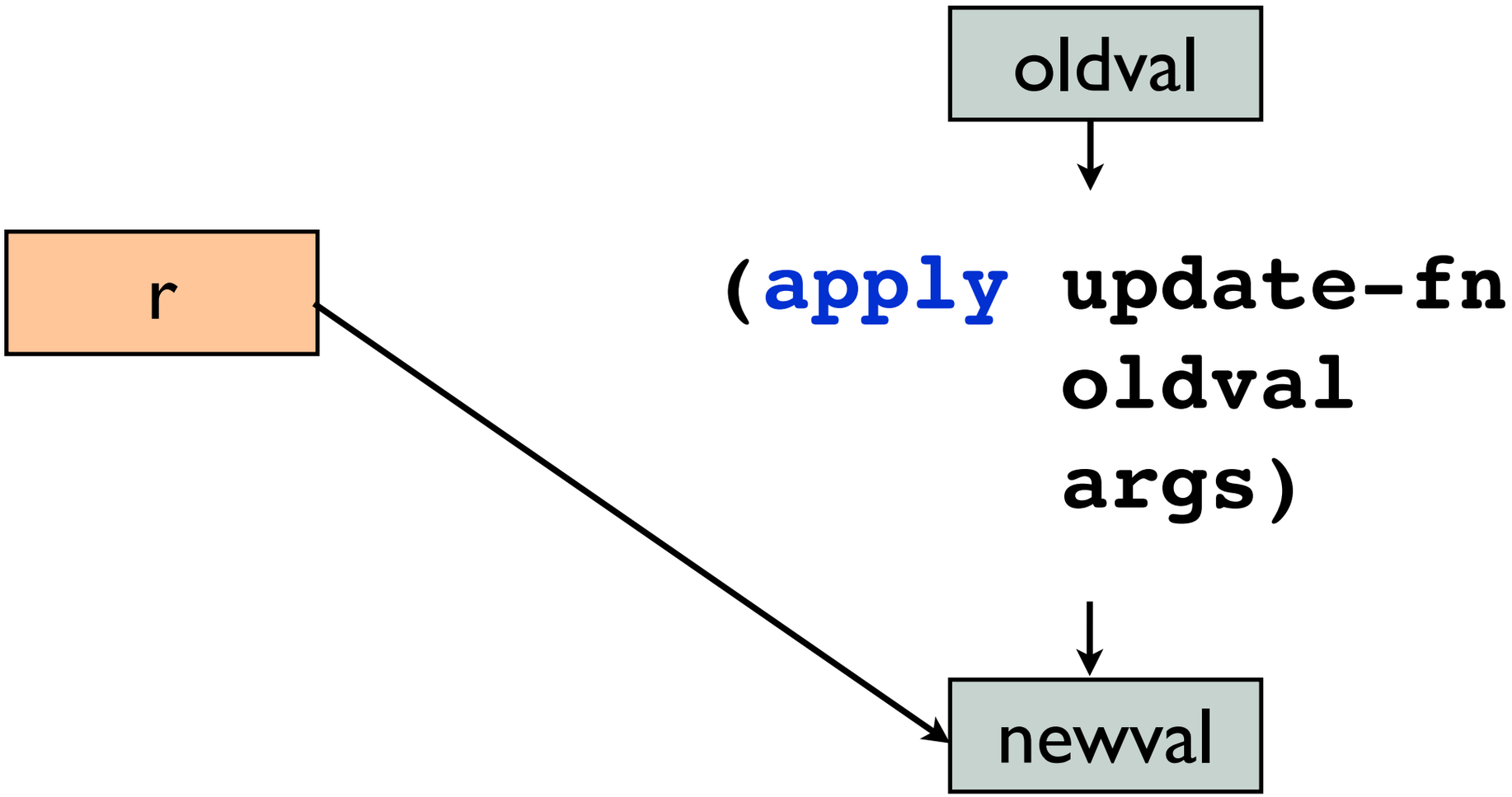
alter

```
(alter r update-fn & args)
```



alter

```
(alter r update-fn & args)
```



updating

apply an...

```
(defn add-message [msg]
  (dosync (alter messages conj msg)))
```

scope a transaction

...update fn

unified update model

update by function application

readers require no coordination

readers never block anybody

writers never block readers

a sane approach
to local state
permits coordination,
but does not require it

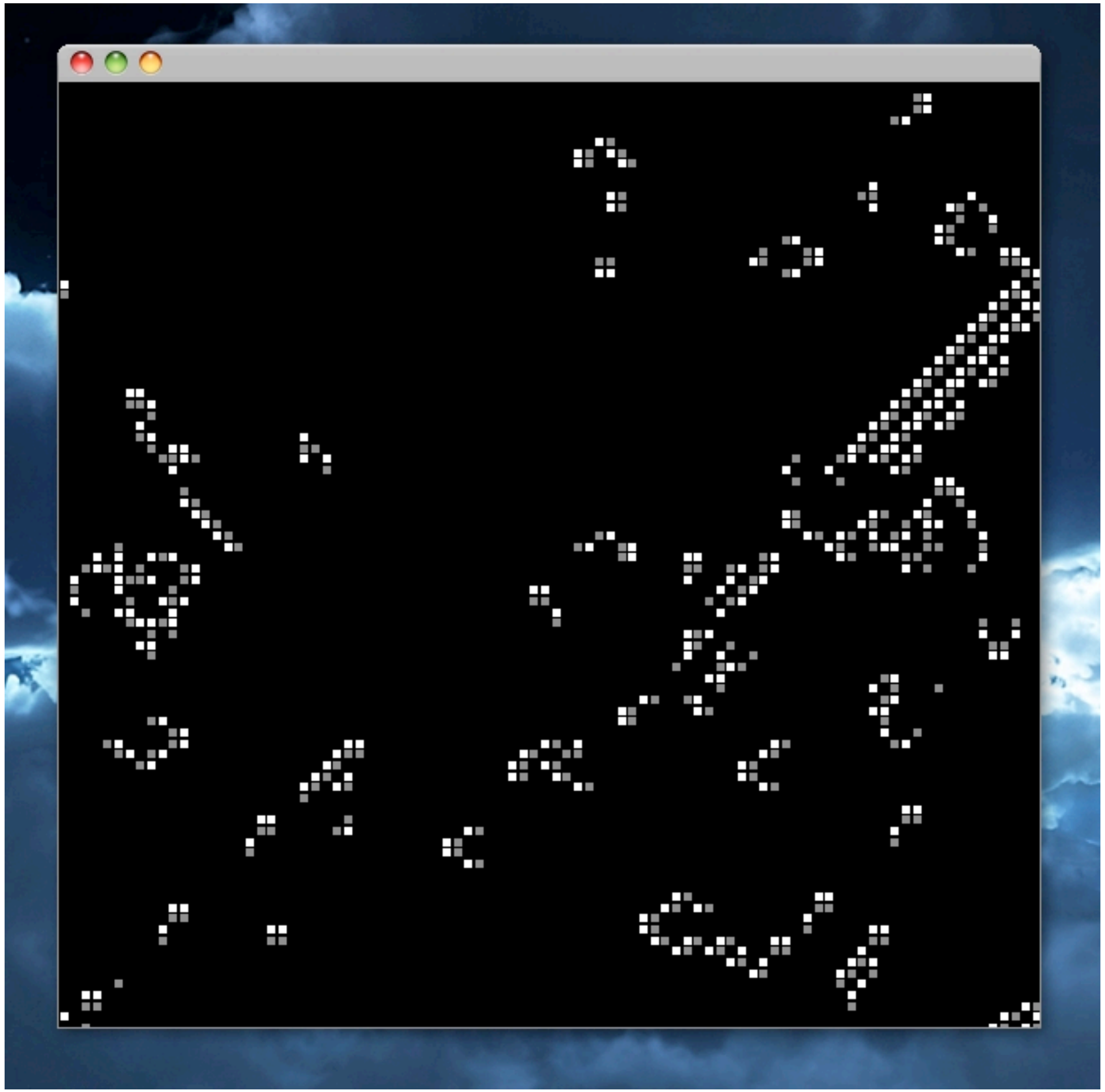
unified update model

	ref	**atom**	**agent**	**var**
create	**`ref`**	**`atom`**	**`agent`**	**`def`**
deref	**`deref/@`**	**`deref/@`**	**`deref/@`**	**`deref/@`**
update	**`alter`**	**`swap!`**	**`send`**	**`alter-var-root`**

identity 2:

atoms

http://blog.bestinclass.dk/index.php/2009/10/brians-functional-brain/



board is just a value

```
(defn new-board
  "Create a new board with about half the cells set
   to :on."
  ([] (apply new-board dim-board))
  ([dim-x dim-y]
     (for [x (range dim-x)]
       (for [y (range dim-y)]
         (if (< 50 (rand-int 100)) :on :off))))))
```

distinct bodies by arity

update is just a function

```
(defn step
  "Advance the automation by one step, updating all
   cells."
  [board]
  (doall
   (map (fn [window]
          (apply #(doall (apply map rules %&))
                 (doall (map torus-window window))))
        (torus-window board))))
```

rules

cursor over previous, me, next

state is trivial

identity

initial value

```
(let [stage (atom (new-board))]
  ...)
```

```
(defn update-stage
  "Update the automaton."
  [stage]
  (swap! stage step))
```

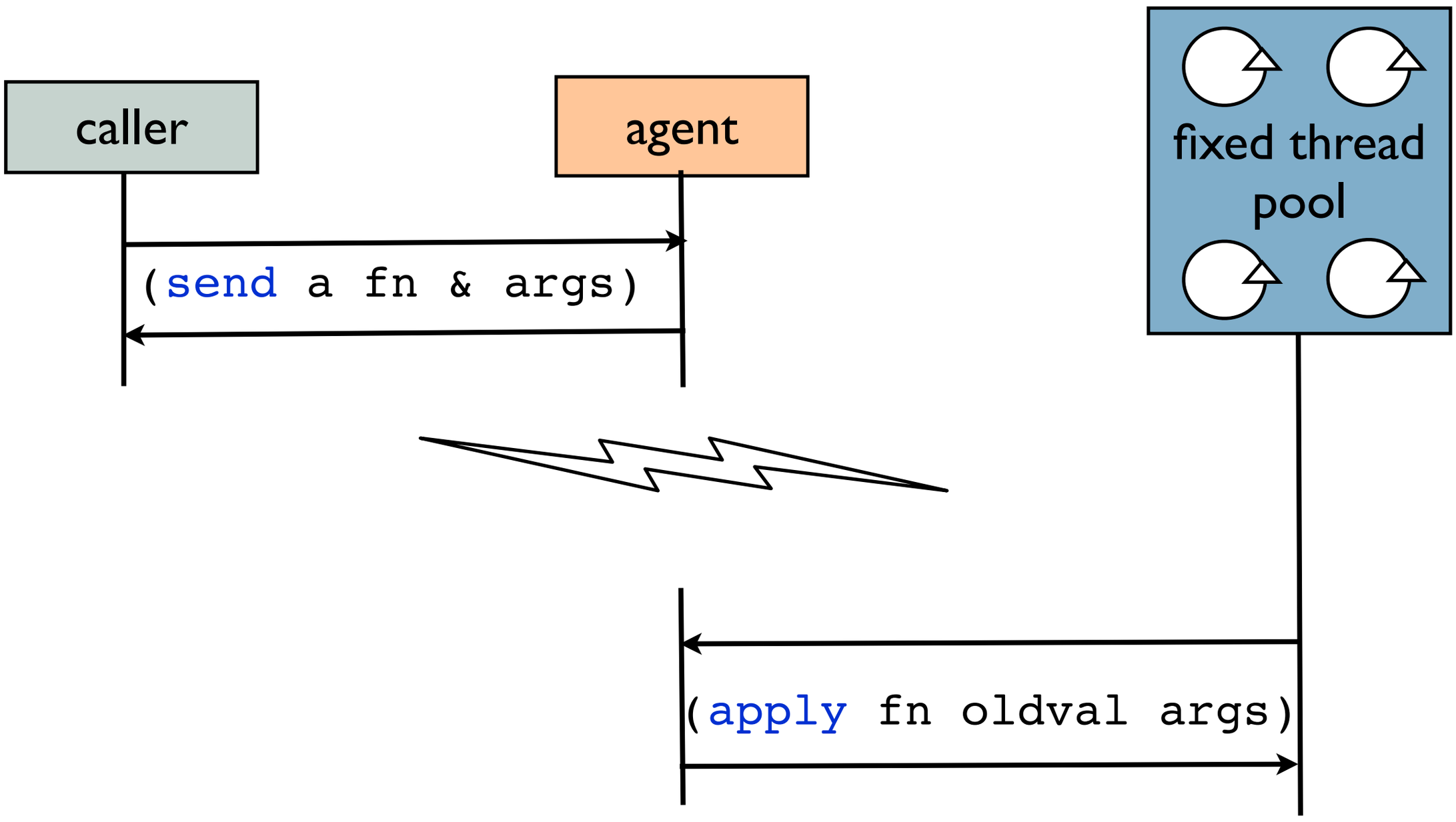
apply a fn

update fn

identity 3:

agents

send


caller
agent
fixed thread pool
(send a fn & args)
(apply fn oldval args)

example: deferred logging

identity

initial value

```
(def *logging-agent* (agent nil))

(if log-immediately?
  (impl-write! log-impl level msg err)
  (send-off *logging-agent*
            agent-write! log level msg err))
```

apply a fn

"update" fn

identity 4:

vars

def forms create vars

```
(def greeting "hello")

(defn make-greeting [n]
  (str "hello, " n)
```

vars can be rebound

api	scope
`alter-var-root`	root binding
`set!`	thread-local, permanent
`binding`	thread-local, dynamic

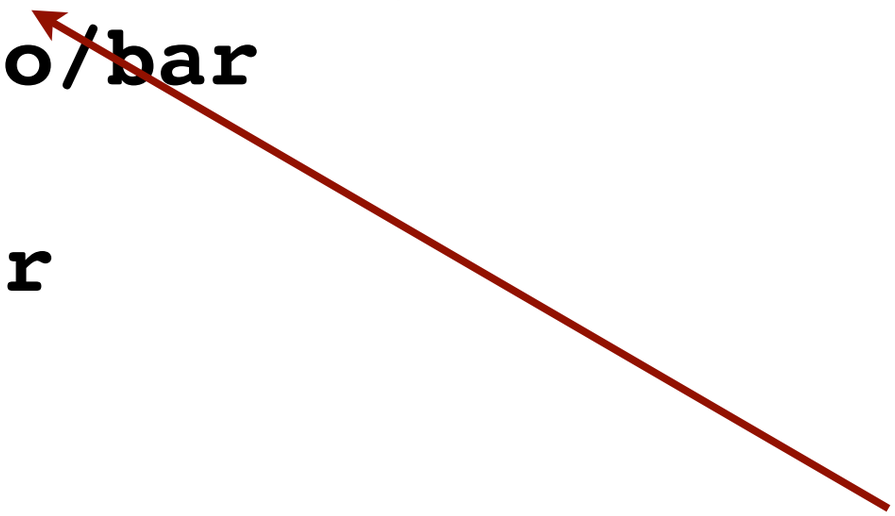
system settings

```
(set! *print-length* 20)
=> 20

primes
=> (2 3 5 7 11 13 17 19 23 29 31 37 41
    43 47 53 59 61 67 71 ...)

(set! *print-length* 5)
=> 5

primes
=>(2 3 5 7 11 ...)
```

var	usage
`*in*, *out*, *err*`	standard streams
`*print-length*, *print-depth*`	structure printing
`*warn-on-reflection*`	performance tuning
`*ns*`	current namespace
`*file*`	file being evaluated
`*command-line-args*`	*guess*

with-... helper macros

```
(def bar 10)
-> #'user/bar

(with-ns 'foo (def bar 20))
-> #'foo/bar

user/bar
-> 10

foo/bar
-> 20
```



bind a var
for a dynamic
scope

other def forms

form	usage
`defonce`	set root binding once
`defvar`	var plus docstring
`defunbound`	no initial binding
`defstruct`	map with slots
`defalias`	same metadata as original
`defhinted`	infer type from initial binding
`defmemo`	defn + memoize

many of these are in clojure.contrib.def...

unified update, revisited

update mechanism	ref	atom	agent
pure function application	`alter`	`swap!`	`send`
pure function (commutative)	`commute`	-	-
pure function (blocking)	-	-	`send-off`
setter	`ref-set`	`reset!`	-

send-off to *agent* for background iteration

monte carlo via ongoing agent

```
(defn background-pi [iter-count]
  (let
    [agt (agent {:in-circle 0 :total 0})
     continue (atom true)
     iter (fn sim [a-val]
            (when continue (send-off *agent* sim))
            (run-simulation a-val iter-count))]
    (send-off agt iter)
    {:guesser agt :continue atom})
```

queue more work

escape hatch

do the work

(not (= agents actors))

agents	actors
in-process only	oop
no copying	copying
no deadlock	can deadlock
no workflow	workflow

validation

create a function

that checks every item...

```
(def validate-message-list
  (partial
    every?
    #(and (:sender %) (:text %))))
```

for some criteria

```
(def messages
  (ref
    ()
    :validator validate-message-list))
```

and associate fn with updates to a ref

agent error handling

```
(def counter (agent 0 :validator integer?))
-> #'user/counter

(send counter (constantly :fail))
-> #<Agent 0>

(agent-errors counter)
-> (#<IllegalStateException
     java.lang.IllegalStateException:
     Invalid reference state>)

(clear-agent-errors counter)
-> nil

@counter
-> 0
```

will fail soon

list of errors

reset and move on

agents and transactions

tying agent to a tx

```
(defn add-message-with-backup [msg]
  (dosync
   (let [snapshot (alter messages conj msg)]
     (send-off backup-agent (fn [filename]
                              (spit filename snapshot)
                              filename))
     snapshot)))
```

exactly once if tx succeeds

where are we?

1. java interop
2. lisp
3. functional
4. value/identity/state

does it work?

a workable approach to state

good values: persistent data structures

good identities: references

mostly functional?

usable by mortals?

mostly

functional?

1 line in 1000 creates a reference

project	loc	calls to ref	calls to agent	calls to atom
clojure	7232	3	1	2
clojure-contrib	17032	22	2	12
compojure	1966	1	0	0
incanter	6248	1	0	0

usable by mortals?

```clojure
; compojure session management
(def memory-sessions (ref {}))

(defmethod read-session :memory
  [repository id]
  (@memory-sessions id))

(defmethod write-session :memory
  [repository session]
  (dosync
    (alter memory-sessions
      assoc (session :id) session)))
```

multimethod dispatch

read

update

```
; from clojure core
(defn memoize [f]
  (let [mem (atom {})]
    (fn [& args]
      (if-let [e (find @mem args)]
        (val e)
        (let [ret (apply f args)]
          (swap! mem assoc args ret)
          ret))))))
```

cache previous results

cache hit

cache miss: call f, add to cache

clojure

values are

- immutable, persistent

identities are

- well-specified, consistent

state is

- mostly functional
- usable by mortals

languages that emphasize

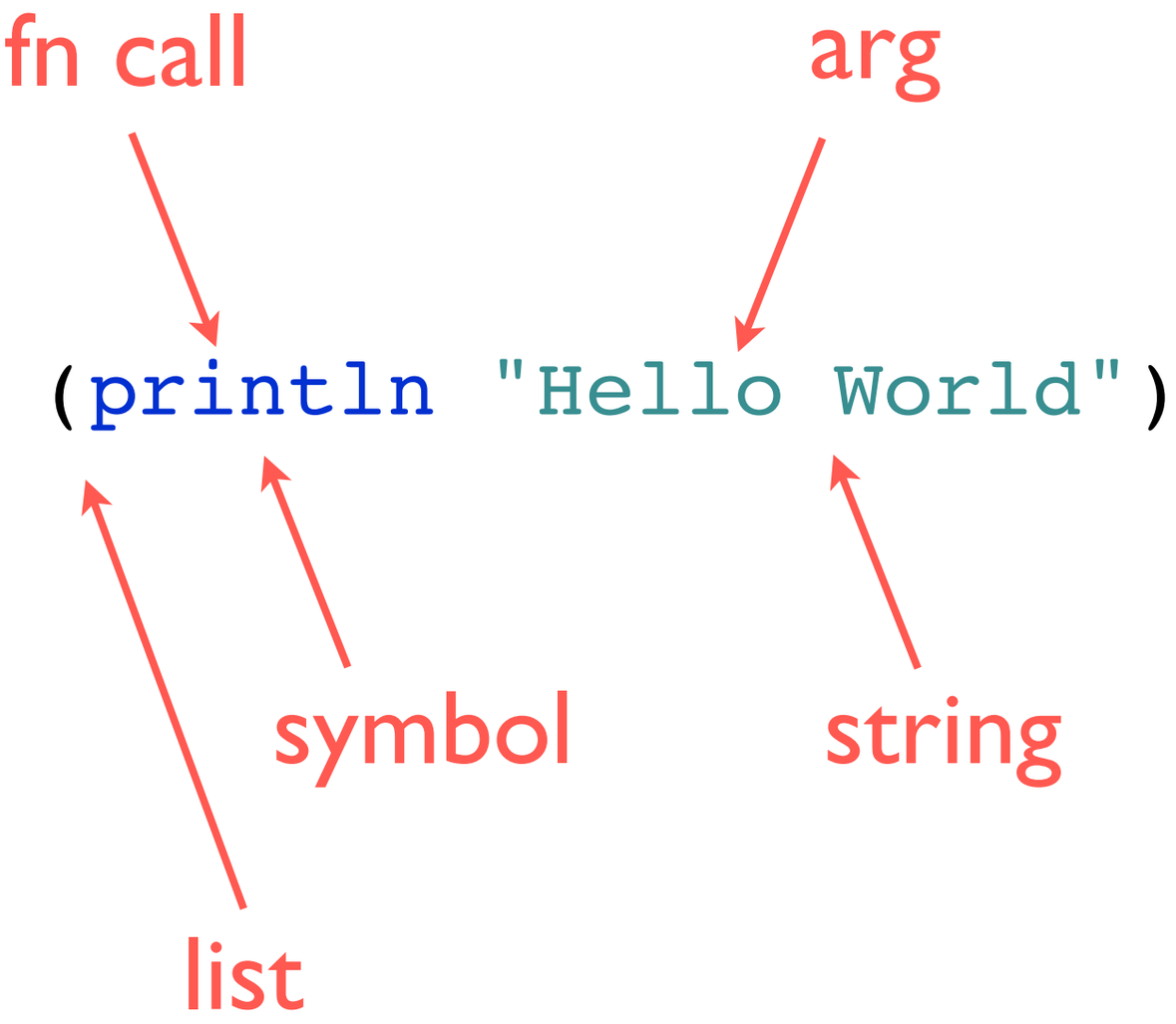
immutability are *better at mutation*

exploring

reading code

semantics:

structure:



defn semantics

define a fn

fn name

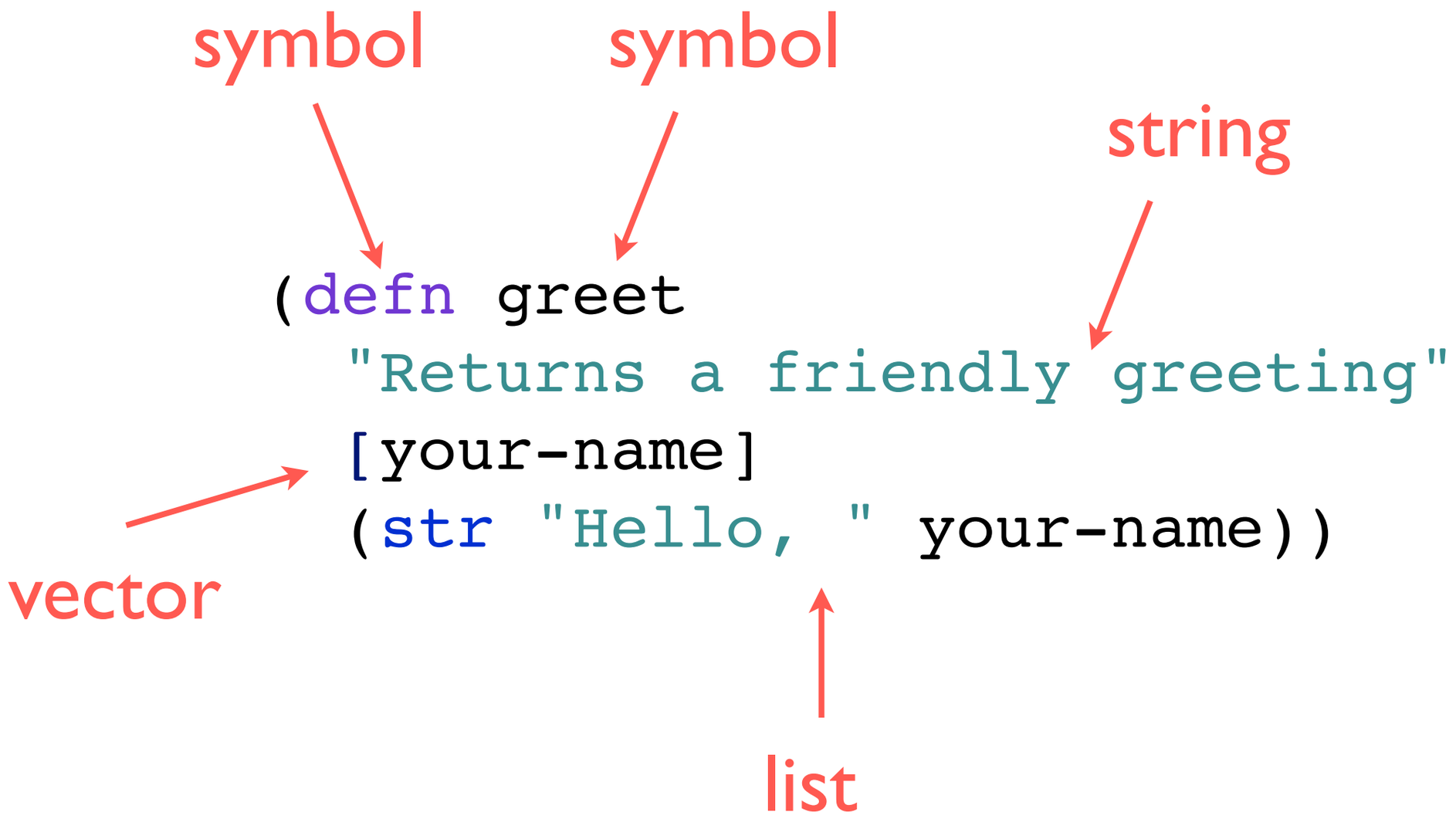
docstring

```
(defn greet
  "Returns a friendly greeting"
  [your-name]
  (str "Hello, " your-name))
```

arguments

fn body

defn structure



doc

```
(doc name)
-------------------------
clojure.core/name
([x])
  Returns the name String of a symbol
or keyword.
```

find-doc

```
(find-doc "pmap")
-------------------------
clojure.core/pmap
([f coll] [f coll & colls])
  Like map, except f is applied in parallel. Semi-lazy in that the
  parallel computation stays ahead of the consumption, but doesn't
  realize the entire result unless required. Only useful for
  computationally intensive functions where the time of f dominates
  the coordination overhead.
-------------------------
clojure.core/zipmap
([keys vals])
  Returns a map with the keys mapped to the corresponding vals.
```

source

```
(use '[clojure.contrib.repl-utils :only (source)])


(source odd?)
(defn odd?
  "Returns true if n is odd, throws an exception if
   n is not an integer"
  [n] (not (even? n)))
```

javadoc

```
(javadoc "foo")
```

Overview Package **Class** Use Tree Deprecated Index Help

Java™ Platform Standard Ed. 6

PREV CLASS NEXT CLASS FRAMES NO FRAMES All Classes

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

java.lang

Class String

java.lang.Object
  └ **java.lang.String**

All Implemented Interfaces:

Serializable, CharSequence, Comparable<String>

```
public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence
```

The `String` class represents character strings. All string literals in Java programs, such as `"abc"`, are implemented as instances of this class.

Strings are constant; their values cannot be changed after they are created. String buffers support mutable strings. Because String objects are immutable they can be shared. For example:

```
String str = "abc";
```

dir

```
(use '[clojure.contrib.ns-utils :only (dir)

(dir clojure.contrib.java-utils)
as-file
as-properties
as-str
as-url
file
get-system-property
read-properties
relative-path-string
set-system-properties
with-system-properties
write-properties
```

show (java)

```
(use '[clojure.contrib.repl-utils :only (show)])

(show 1)
===  public final java.lang.Integer  ===
[ 0] static MAX_VALUE : int
[ 1] static MIN_VALUE : int
[ 2] static SIZE : int
[ 3] static TYPE : Class
[ 4] static bitCount : int (int)
[ 5] static decode : Integer (String)
[ 6] static getInteger : Integer (String)
[ 7] static getInteger : Integer (String,Integer)
[ 8] static getInteger : Integer (String,int)
[ 9] static highestOneBit : int (int)
[10] static lowestOneBit : int (int)
...
```

show (clojure)

```
(use '[clojure.contrib.repl-utils :only (show)])

user=> (show [1 2 3])
===  public clojure.lang.PersistentVector  ===
[ 0] static EMPTY : PersistentVector
[ 1] static applyToHelper : Object (IFn,ISeq)
[ 2] static create : PersistentVector (ISeq)
[ 3] static create : PersistentVector (List)
[ 4] static create : PersistentVector (Object[])
[ 5] add : boolean (Object)
[ 6] add : void (int,Object)
[ 7] addAll : boolean (Collection)
[ 8] addAll : boolean (int,Collection)
[ 9] applyTo : Object (ISeq)
[10] arrayFor : Object[] (int)
...
```

pprint

```
(use '[clojure.contrib.pprint :only (pprint)])
-> nil

(pprint
 (for [rank (range 8 0 -1)]
   (for [file "abcdefgh"]
     (str file rank))))
(("a8" "b8" "c8" "d8" "e8" "f8" "g8" "h8")
 ("a7" "b7" "c7" "d7" "e7" "f7" "g7" "h7")
 ("a6" "b6" "c6" "d6" "e6" "f6" "g6" "h6")
 ("a5" "b5" "c5" "d5" "e5" "f5" "g5" "h5")
 ("a4" "b4" "c4" "d4" "e4" "f4" "g4" "h4")
 ("a3" "b3" "c3" "d3" "e3" "f3" "g3" "h3")
 ("a2" "b2" "c2" "d2" "e2" "f2" "g2" "h2")
 ("a1" "b1" "c1" "d1" "e1" "f1" "g1" "h1"))
```

more concurrency options

prepare to parallelize

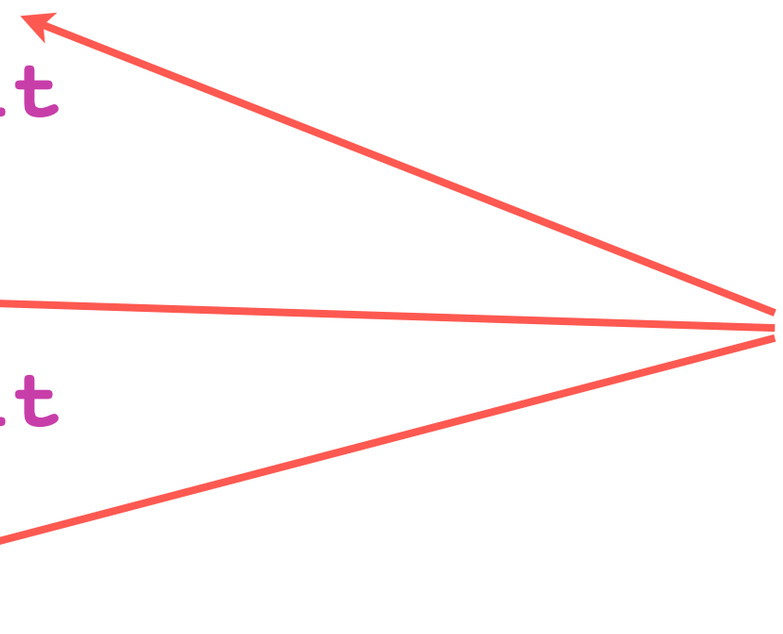
```
(defn step
  "Advance the automation by one step, updating all
   cells."
  [board]
  (doall
   (map (fn [window]
          (apply #(doall (apply map rules %&))
                 (doall (map torus-window window))))
        (torus-window board))))
```

done

```
(defn step
  "Advance the automation by one step, updating all
   cells."
  [board]
  (doall
   (pmap (fn [window]
           (apply #(doall (apply map rules %&))
                  (doall (map torus-window window))))
         (torus-window board))))
```

delay

```
(def e (delay (expensive-calculation)))
-> #'demo.delay/e

(delay? e)
-> true

(force e)
-> :some-result

(deref e)
-> :some-result

@e
-> :some-result
```



first call blocks until work completes on **this** thread, later calls hit cache

future

```
(def e1 (future (expensive-calculation)))
-> #'demo.future/e1

(deref e1)
-> :some-result

@e1
-> :some-result
```

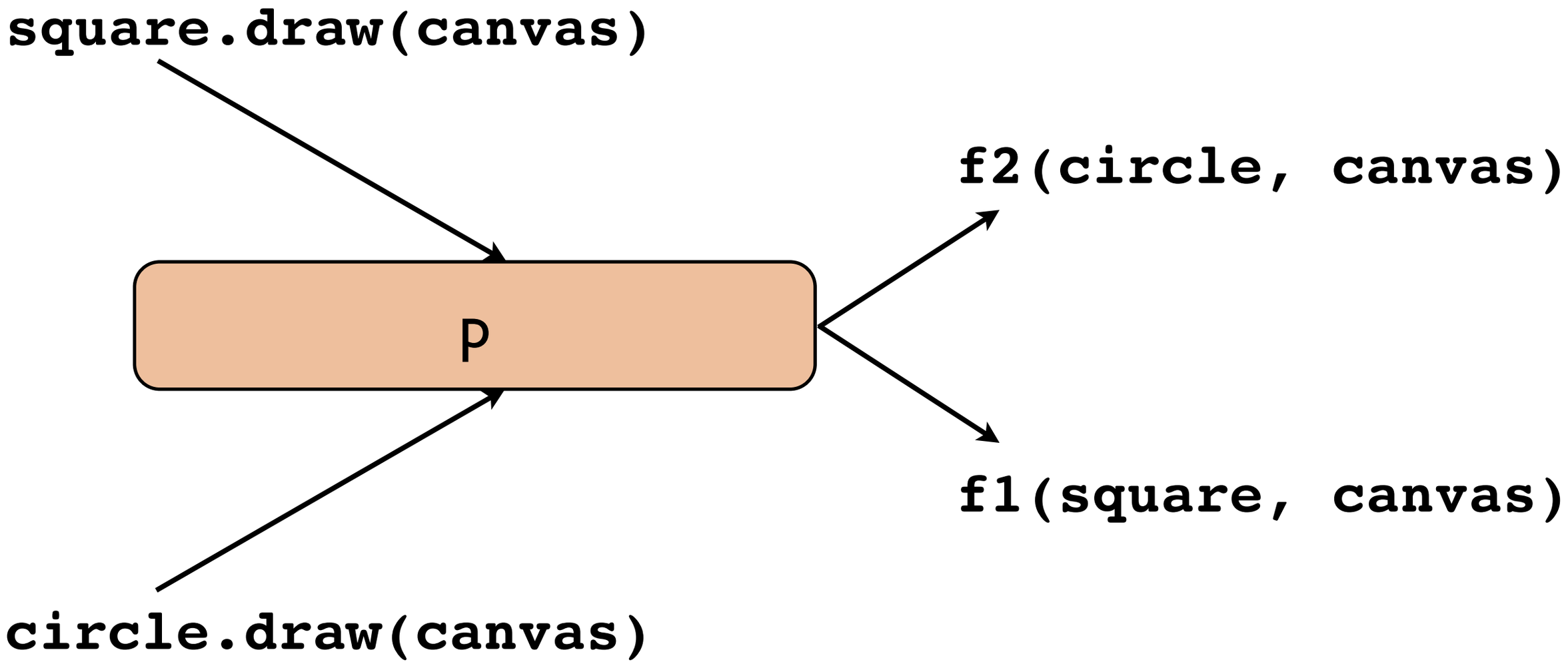
first call blocks until work completes on **other** thread, later calls hit cache

cancelling a future

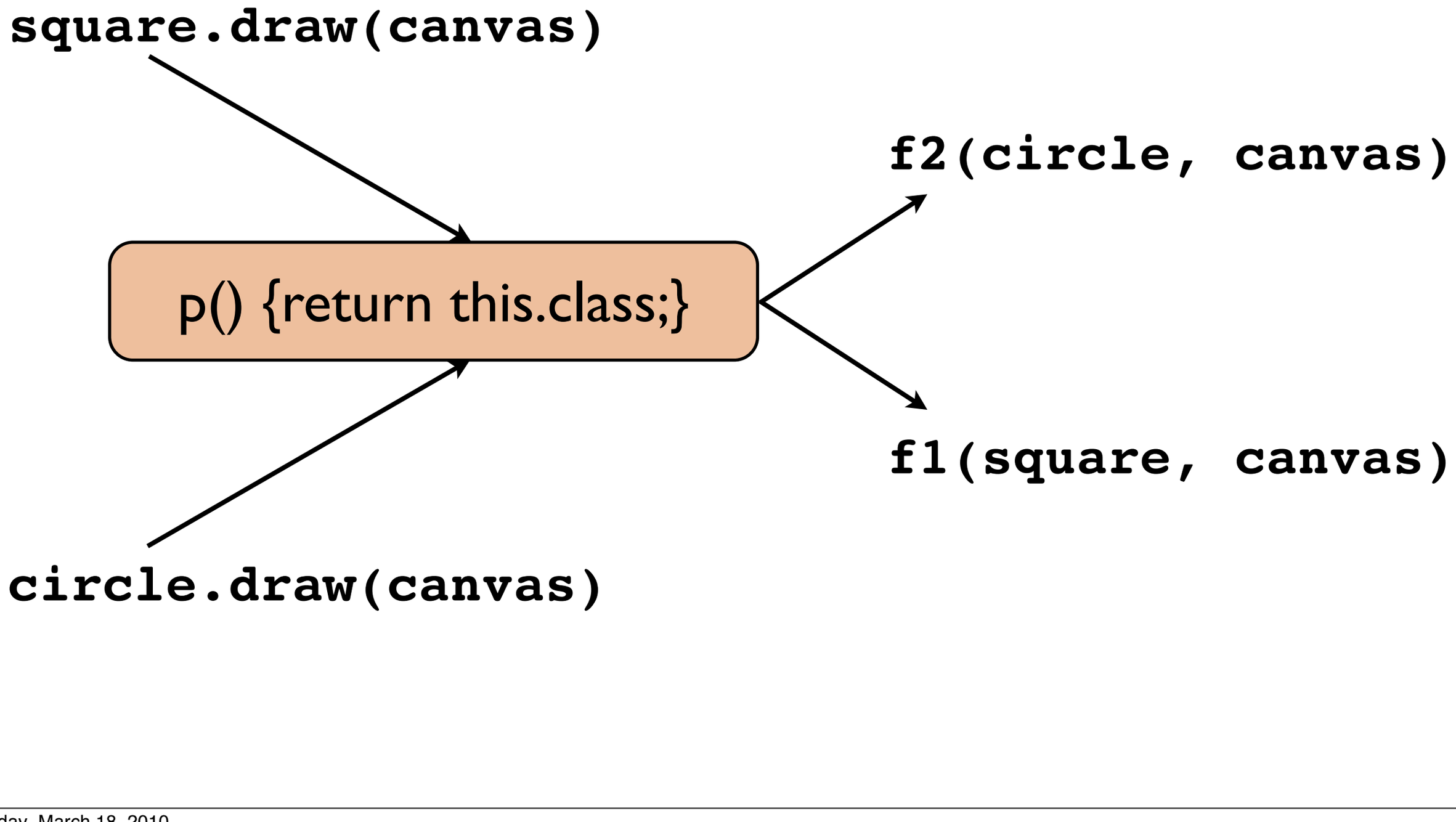
```
(def e2 (future (expensive-calculation)))
-> #'demo.future/e2

(future-cancel e2)
-> true

(future-cancelled? e2)
-> true

(deref e2)
-> java.util.concurrent.CancellationException
```

multimethods

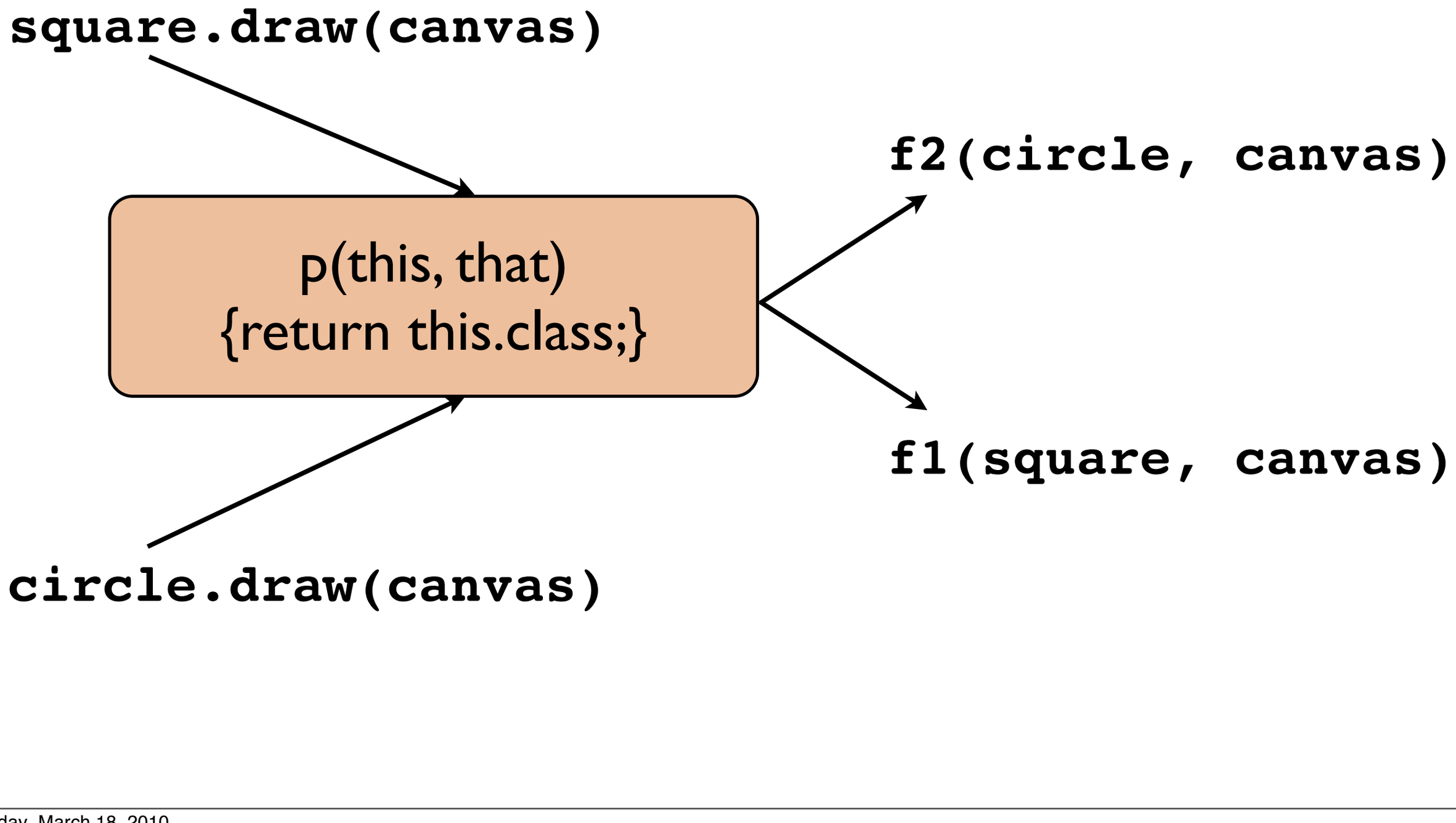
polymorphism



p is just a function



this isn't special



check all args

```
(fn [this, that]
  [(class this)
   (class that)])
```

```
(fn [square, canvas])

(fn [circle, canvas])

(fn [square, surface])

(fn [circle, surface])
```

check arg twice

```
(fn [this, that]
  [(class this)
   (opaque? this)
   (class that)])
```

```
fn1
fn2
fn3
fn4
fn5
fn6
fn7
fn8
```

example: coerce

define a multimethod

```
(defmulti coerce
  (fn [dest-class src-inst]
    [dest-class (class src-inst)]))
```

based on dest (a class)

and src (an inst)

method impls

dispatch value to match

```
(defmethod coerce
  [java.io.File String]
  [_ str]
  (java.io.File. str))
```

args

body

```
(defmethod coerce
  [Boolean/TYPE String] [_ str]
  (contains?
   #{"on" "yes" "true"}
   (.toLowerCase str)))
```

defaults

```
(defmethod coerce
  :default
  [dest-cls obj]
  (cast dest-cls obj))
```

dispatch comparison

language	java	ruby	clojure
basic polymorphism?	✔	✔	✔
change "methods" at runtime?		✔	✔
change "types" at runtime?		✔	✔
dispatch based on all arguments?		*	✔
arbitrary fn dispatch?		*	✔
pattern matching		*	*

dispatch workarounds are the middle management of the design patterns movement

class inheritance

```
(defmulti whatami? class)

(defmethod whatami? java.util.Collection
  [_] "a collection")

(whatami? (java.util.LinkedList.))
-> "a collection"

(defmethod whatami? java.util.List
  [_] "a list")

(whatami? (java.util.LinkedList.))
-> "a list"
```

add methods anytime

most derived type wins

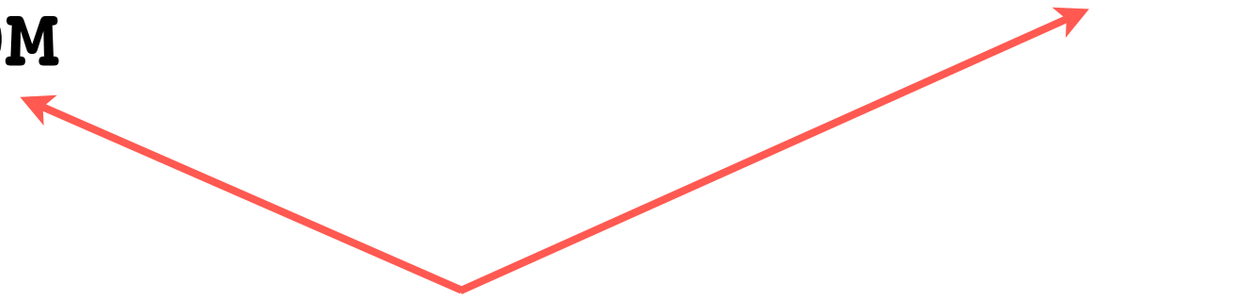
name inheritance

```
(defmulti interest-rate :type)
(defmethod interest-rate ::account
  [_] 0M)
(defmethod interest-rate ::savings
  [_] 0.02)
```

double colon (::) is shorthand for resolving keyword into the current namespace, e.g. ::savings == :my.current.ns/savings

deriving names


```
(derive ::checking ::account)
(derive ::savings ::acount)

(interest-rate {:type ::checking})
-> 0M
```



there is no ::checking method, so select method for base name ::account

multimethods

function	notes
`prefer-method`	resolve conflicts
`methods`	reflect on {dispatch, meth} pairs
`get-method`	reflect by dispatch
`remove-method`	remove by dispatch
`prefers`	reflect over preferences

macros

the **if** special form

```
(if test
    then
    else?)
```

evaluate only
if `test` is
logical true

evaluate only
if `test` is
logical false

if-like things cannot be functions!

function calls

evaulate their args

pass args to implementation

if

evaluates an arg

decides which other args to evaluate, and when

lisp macros

get access to source forms

after they are read

before compile/interpret

macroexpand forms into other forms

choose when/how to evaluate each argument

example: **when**

```
(when x
  (println "x is true")))
```

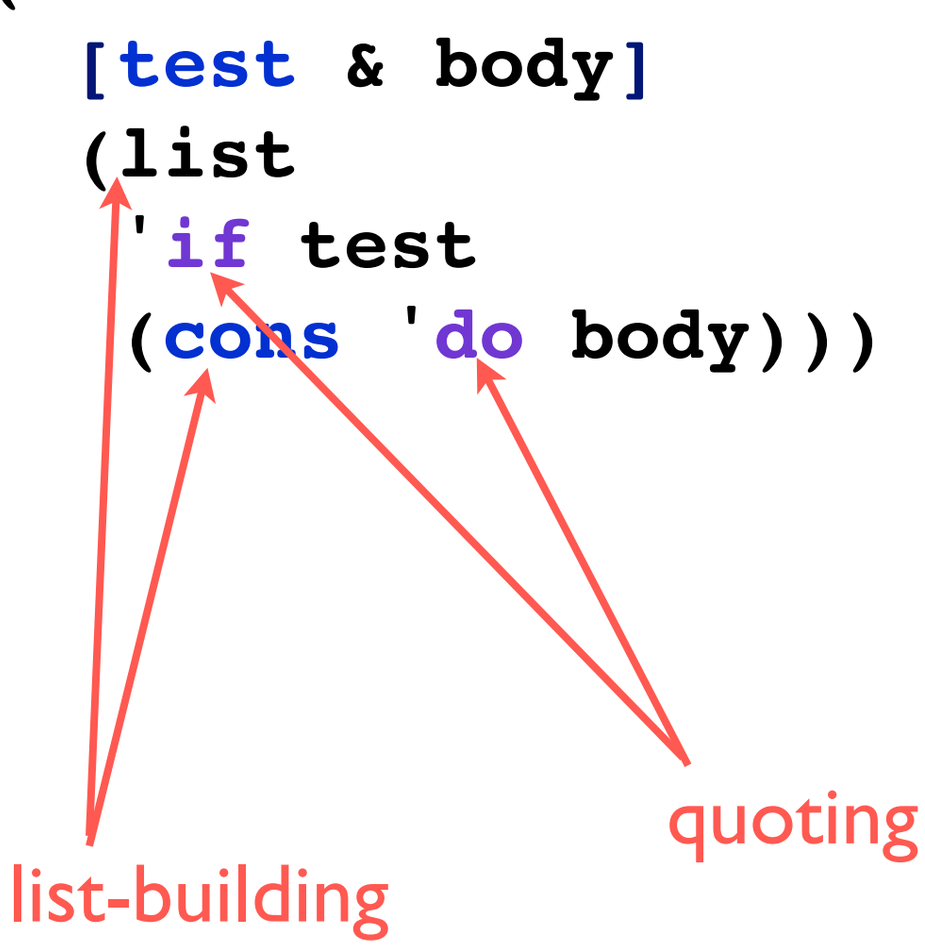
macroexpansion

```
(defmacro when
[test & body]
(list
 'if test
 (cons 'do body)))
```

```
(if x
  (do (println "x is true")))
```

quoting and list-building

```
(defmacro when
  [test & body]
  (list
   'if test
   (cons 'do body)))
```



syntax-quoting

```
(defmacro when
  [test & body]
  (list
   'if test
   (cons 'do body)))
```

=

```
(defmacro when
  [test & body]
  `(if ~test
     (do ~@body)))
```

unquote

syntax-quote

unquote-splicing

test your macros!

```
(macroexpand-1
   '(when x
      (println "x is true")))

-> (if x
       (do (println "x is true")))
```

a bench macro

```
(defmacro bench [expr]
  `(let [start (System/nanoTime)
         result ~expr]
     {:result result
      :elapsed (- (System/nanoTime)
                  start)}))
```

not done yet...

start/result can **capture** caller bindings

```
(defmacro bench [expr]
  `(let [start (System/nanoTime)
         result ~expr]
     {:result result
      :elapsed (- (System/nanoTime)
                  start)}))
```

not done yet...

avoiding accidental capture

```
(defmacro bench [expr]
  `(let [start (System/nanoTime)
         result ~expr]
     {:result result
      :elapsed (- (System/nanoTime)
                  start)}))
```

```
(bench (x))
-> java.lang.Exception:
   Can't let qualified name: user/start
```

not done yet...

use auto-gensyms

```
(defmacro bench [expr]
  `(let [start# (System/nanoTime)
         result# ~expr]
     {:result result#
      :elapsed (- (System/nanoTime)
                  start#)}))
```

suffix generates unique symbol within a quoted form

generated symbols

```
(macroexpand-1 '(bench (x)))

-> (clojure.core/let
     [start__37__auto__ (java.lang.System/nanoTime)
      result__38__auto__ (x)]
    {:result result__38__auto__,
     :elapsed (clojure.core/-
                (java.lang.System/nanoTime)
                 start__37__auto__)})
```

common macro types

type	examples
control flow	`when when-not and or`
vars	`defn defmacro defmulti`
java interop	`.. doto deftype proxy`
rearranging	`-> ->> -?>`
scopes	`dosync time with-open`
“special form”	`fn lazy-seq let`

metadata

metadata:
data that is
orthogonal to the
value of an object

metadata uses

documentation

serialization

protection

optimization

relationships (e.g. test -> testee)

grouping/typing (?)

add & retrieve metadata

add
metadata

data

```
(def x (with-meta
         {:password "swordfish"}
         {:secret true}))
-> #'user/x
```

metadata

```
x
-> {:password "swordfish"}

(meta x)
-> {:secret true}
```

retrieve
metadata

sugar: #^

add metadata

metadata first!

```
(def y #^{:secret true}
           {:password "swordfish"}))
-> #'user/y
```

```
(meta y)
-> {:secret true}
```

retrieve metadata

subtleties

metadata can be on data, **or on a var**

to place metadata on a var:

put it on the symbol when defing the var

compiler will copy it to the var

metadata on functions added Jan 19, 2010

var: add metadata

add metadata

to symbol

```
(def #^{:secret true} z
  {:password "swordfish"})
-> #'user/z
```

```
(meta z)
-> nil
```

z's data has no metadata

var: retrieve metadata

#' is
var-quote

```
(meta #'z)
-> {:ns #<Namespace user>,
    :name z,
    :file "NO_SOURCE_PATH",
    :line 34,
    :secret true}
```

implicit
metadata

explicit
metadata

add

type hints

to improve performance

type metadata example

```
(defn capitalize
  "Upcase the first character of a string,
   lowercase the rest."
  [s]
  (if (.isEmpty s)
    s
    (let [up (.. s
                 (substring 0 1)
                 (toUpperCase))
          down (.. s
                   (substring 1)
                   (toLowerCase))]
      (.concat up down))))
```

warn-on-reflection

```
(set! *warn-on-reflection* true)
-> true

(require :reload 'demo.capitalize)
Reflection warning, demo/capitalize.clj:6 -
  reference to field isEmpty can't be resolved.
Reflection warning, demo/capitalize.clj:8 -
  call to substring can't be resolved.
Reflection warning, demo/capitalize.clj:8 -
  call to toUpperCase can't be resolved.
Reflection warning, demo/capitalize.clj:11 -
  call to substring can't be resolved.
Reflection warning, demo/capitalize.clj:11 -
  call to toLowerCase can't be resolved.
Reflection warning, demo/capitalize.clj:14 -
  call to concat can't be resolved.
-> nil
```

add type metadata

```
(defn capitalize
  "Upcase the first character of a string,
   lowercase the rest."
  [#^String s]
  (if (.isEmpty s)
    s
    (let [up (.. s
                 (substring 0 1)
                 (toUpperCase))
          down (.. s
                   (substring 1)
                   (toLowerCase))]
      (.concat up down))))
```

s is known to be a String

no more warnings

```
(set! *warn-on-reflection* true)
-> true

(require :reload 'demo.capitalize)
-> nil
```

more idiomatic

```
(defn capitalize
  "Upcase the first character of a string,
   lowercase the rest."
  [#^String s]
  (if (.isEmpty s)
    s
    (.concat
     (.toUpperCase (subs s 0 1))
     (.toLowerCase (subs s 1)))))
```

still non-reflective,
if subs is non-reflective

compojure routes

```
(defroutes snippet-app
  "Create and view snippets."
  (GET "/"
     (new-snippet))

  (GET "/:id"
     (show-snippet (params :id)))

  (POST "/"
    (create-snippet (:body params)))

  (GET "/public/*"
    (or (serve-file (params :*)) :next))

  (ANY "*"
    (page-not-found)))
```

compojure html

```
(defn show-snippet [id]
  (layout
   (str "Snippet " id)
   (let [snippet (select-snippet (Integer/parseInt id))]
     (html
      [:div [:pre [:code.clojure (escape-html snippet)]]]
      [:div.date (:created_at snippet)]))))
```

compojure handler

```
(defn hello-world [request]
  {:status  200
   :headers {}
   :body    "Hello World"})
```

compojure middleware

```
(defn with-header
  [handler header value]
  (fn [request]
    (let [response (handler request)]
      (assoc-in
       response
       [:headers header]
       value))))
```

```
(decorate
 hello-world
 (with-header "X-Lang"
              "Clojure")
 (with-header "X-Framework"
              "Compojure"))
```

clojure's four elevators

java interop

lisp

functional

state

Email: aaron@thinkrelevance.com
Office: 919-442-3030
Twitter: twitter.com/abedra
Github: abedra
Blog: http://aaronbedra.com
Book: http://tinyurl.com/clojure

The Pragmatic Programmers

Programming Clojure

Stuart Halloway

Edited by Susannah Davidson Pfalzer