Introduction to Clojure

@stuartsierra
#strangeloop

Bullet Points

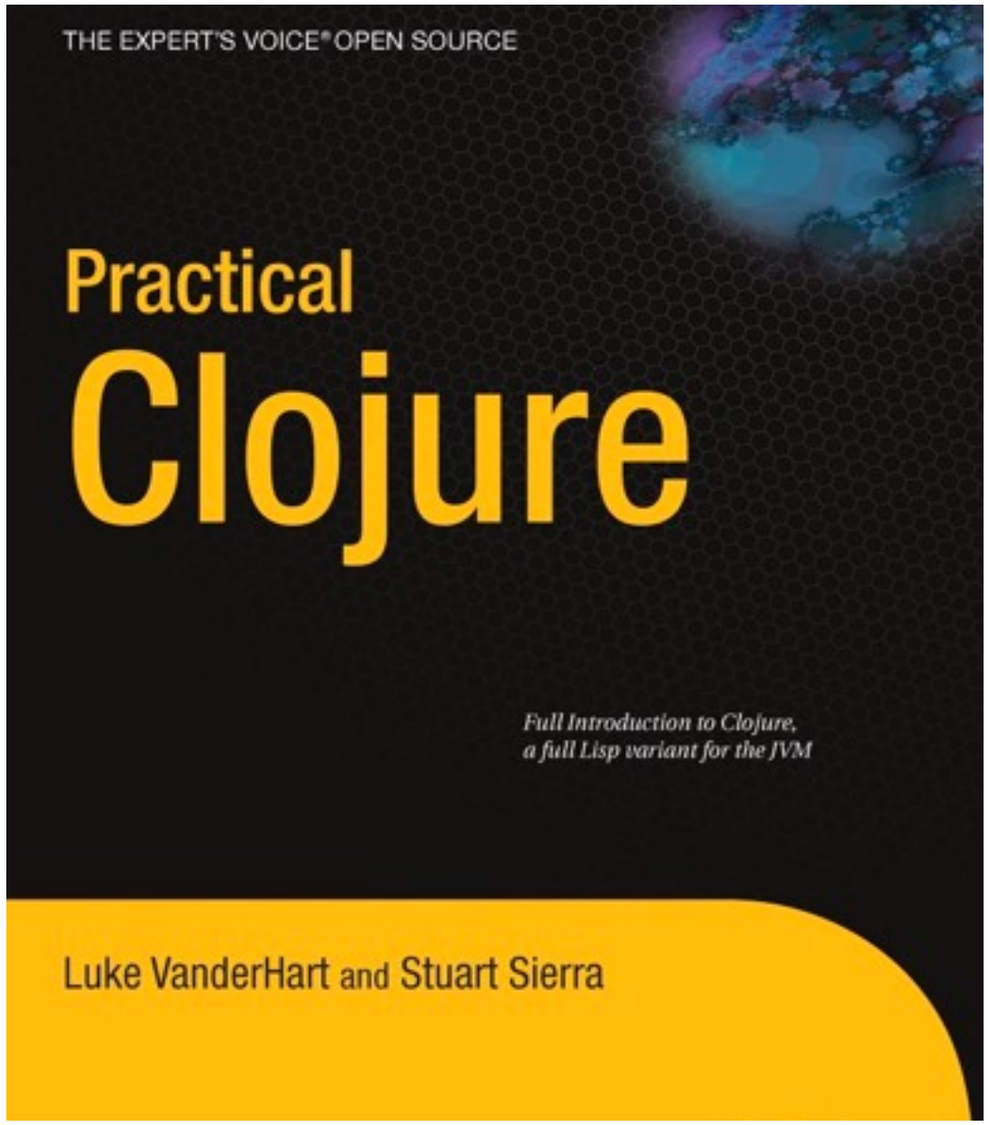
- The REPL
- Data and Code
- Working with Data
- Destructuring
- Higher-Order Functions
- Sequences
- Java Interop
- Libraries
- Namespaces
- Concurrency
- Macros
- Recursion

Stuart Sierra

Relevance, Inc.

Clojure/core

Clojure contributor



Where are you from?

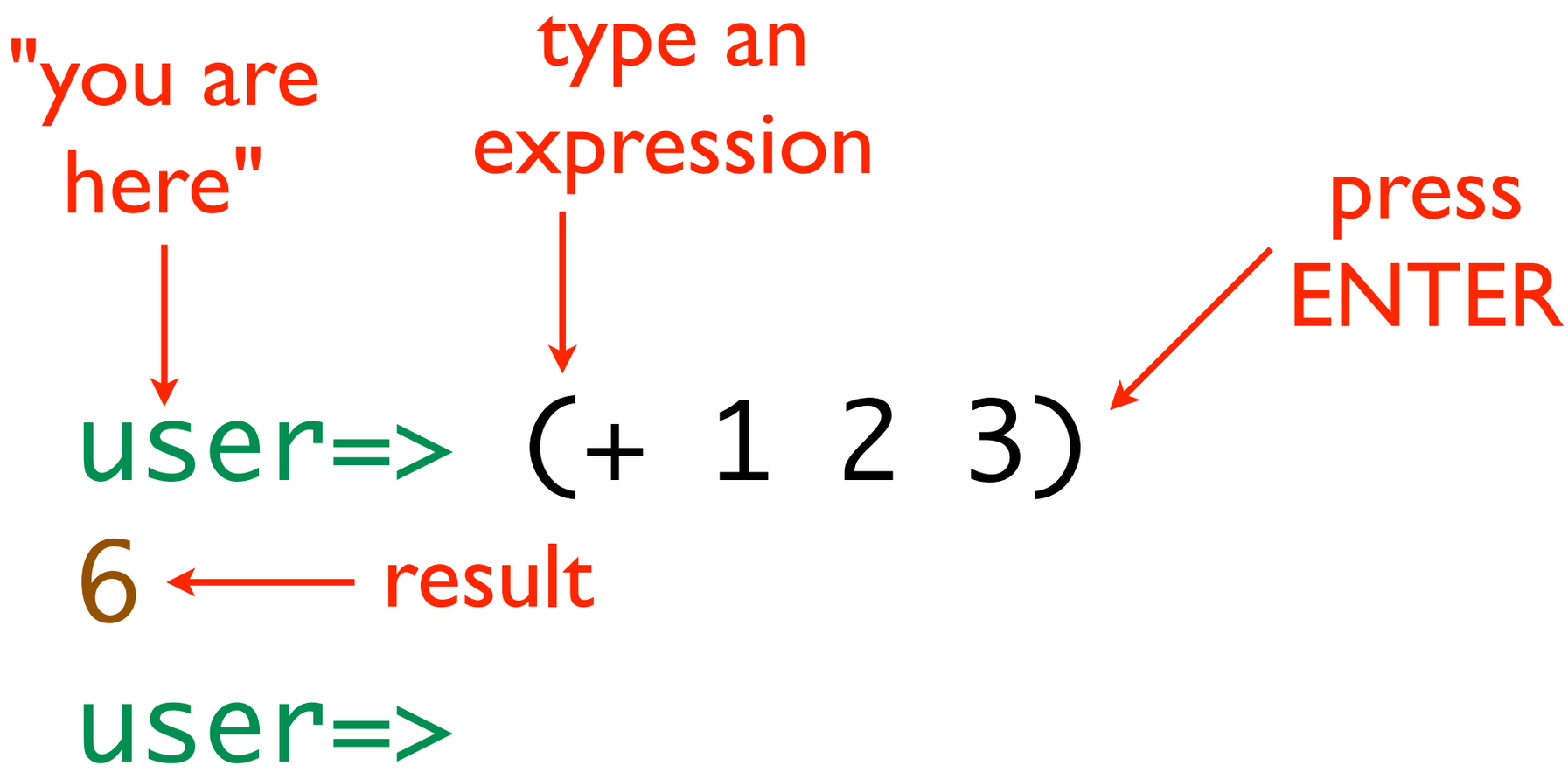
- Lisp?
- Java / C# / C++
- ML / Haskell?
- Python / Ruby / Perl?
- Clojure?
- Multithreaded programming?

The REPL

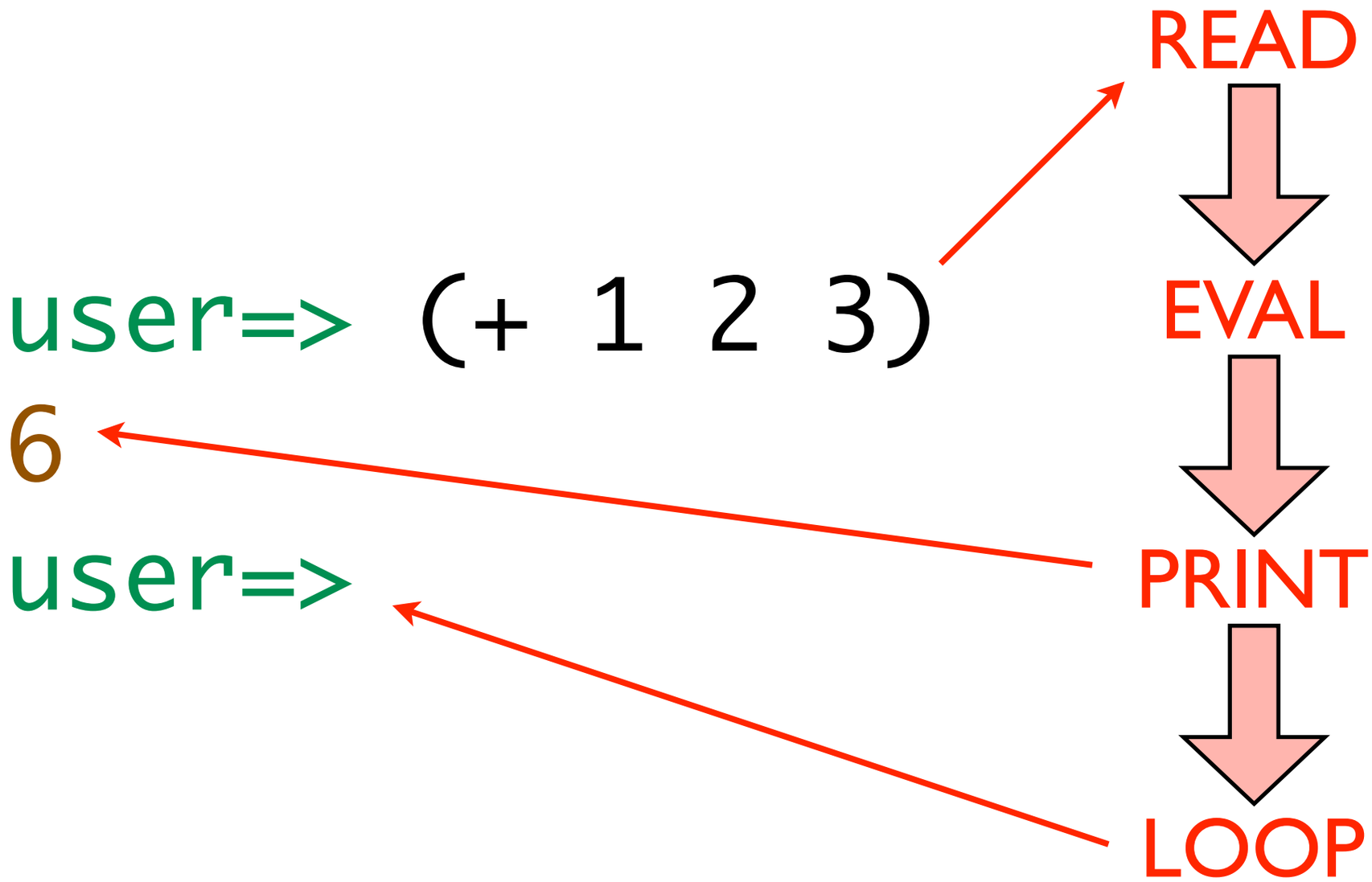
The REPL

- READ
- EVALuate
- PRINT
- LOOP

The REPL


"you are here"
type an expression
press ENTER
user=> (+ 1 2 3)
6
result
user=>

The REPL



The REPL

expression with side effects

```
user=> (println "Hello, World!")
Hello, World!
nil
```

printed output

return value

REPL Helpers: doc

```
user=> (doc when)
-------------------------
clojure.core/when
([test & body])
Macro
  Evaluates test. If logical true,
evaluates body in an implicit do.
nil
```

Fully-qualified name

Arguments

REPL Helpers: find-doc

```
user=> (find-doc "sequence")
... all definitions with "sequence"
in their documentation ...
nil
```

REPL Helpers: apropos

```
user=> (apropos "map")
(sorted-map ns-unmap zipmap map
mapcat sorted-map-by map? amap
struct-map proxy-mappings pmap map-
indexed ns-map array-map hash-map)
```

REPL Helpers: source

```
user=> (source take)
(defn take
  "Returns a lazy sequence of the first n items in coll, or all
items if there are fewer than n."
  {:added "1.0"
   :static true}
  [n coll]
  (lazy-seq
   (when (pos? n)
     (when-let [s (seq coll)]
      (cons (first s) (take (dec n) (rest s)))))))
nil
```

Source code of
the 'take' function

Data and Code

Literals

example	Java type
`"hello"`	String
`\e`	Character
`42`	Long
`6.022e23`	Double
`1.0M`	BigDecimal
`92233720368547758O8N`	clojure.lang.BigInt
`22/7`	clojure.lang.Ratio
`#"hel+o"`	java.util.regex.Pattern

Literals

example	Java type
`nil`	null
`true` `false`	Boolean
`println` `even?` `+`	clojure.lang.Symbol
`:beta` `::gamma`	clojure.lang.Keyword

Data Structures

type	example	properties
list	`(1 2 3)`	singly-linked, grow at front
vector	`[1 2 3]`	indexed, grow at end
map	`{:a 1, :b 20}`	key/value pairs
set	`#{1 7 3}`	unordered, unique keys

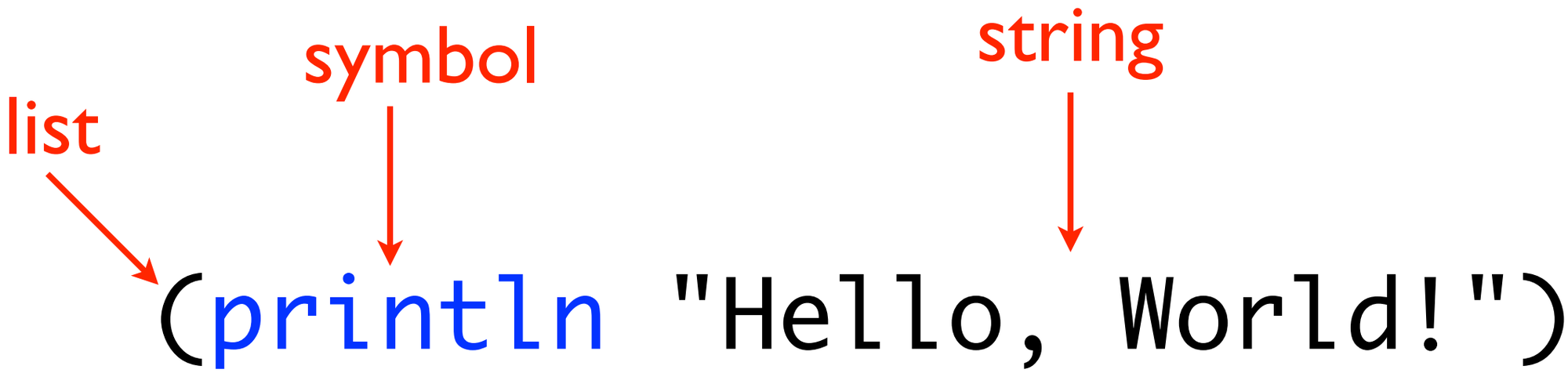
Commas,, are,,,,

,,whitespace,,,,,

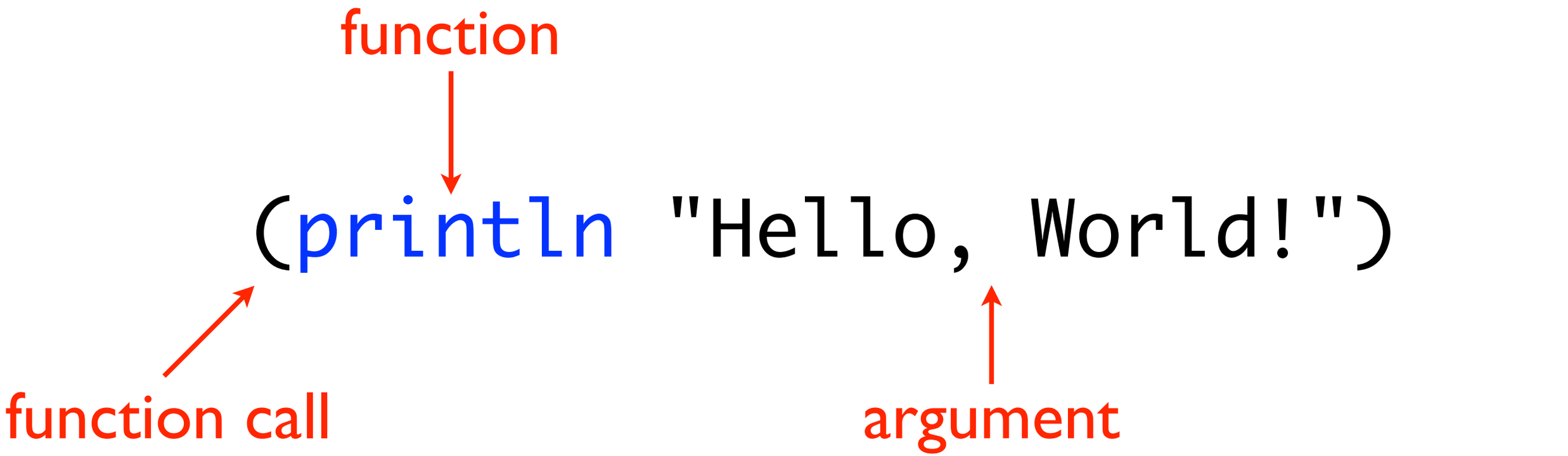
Comments

```
;;; File-level comment

(let [x 6, y 7]
  ;; Block-level comment,
  ;; indented like code
  (println "Hello, World!")
  (* 6 7)) ; end-of-line comment
```

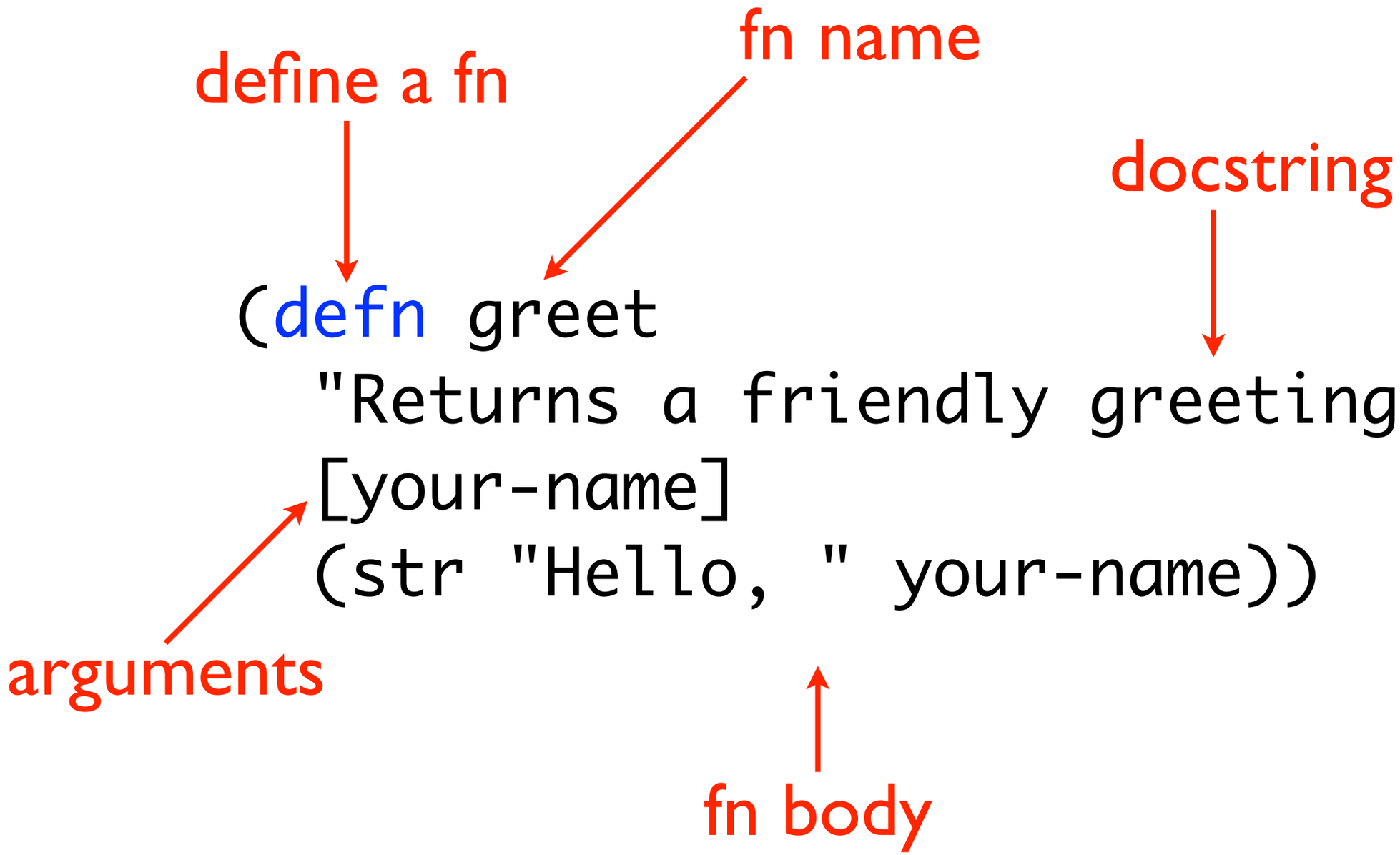
Structure



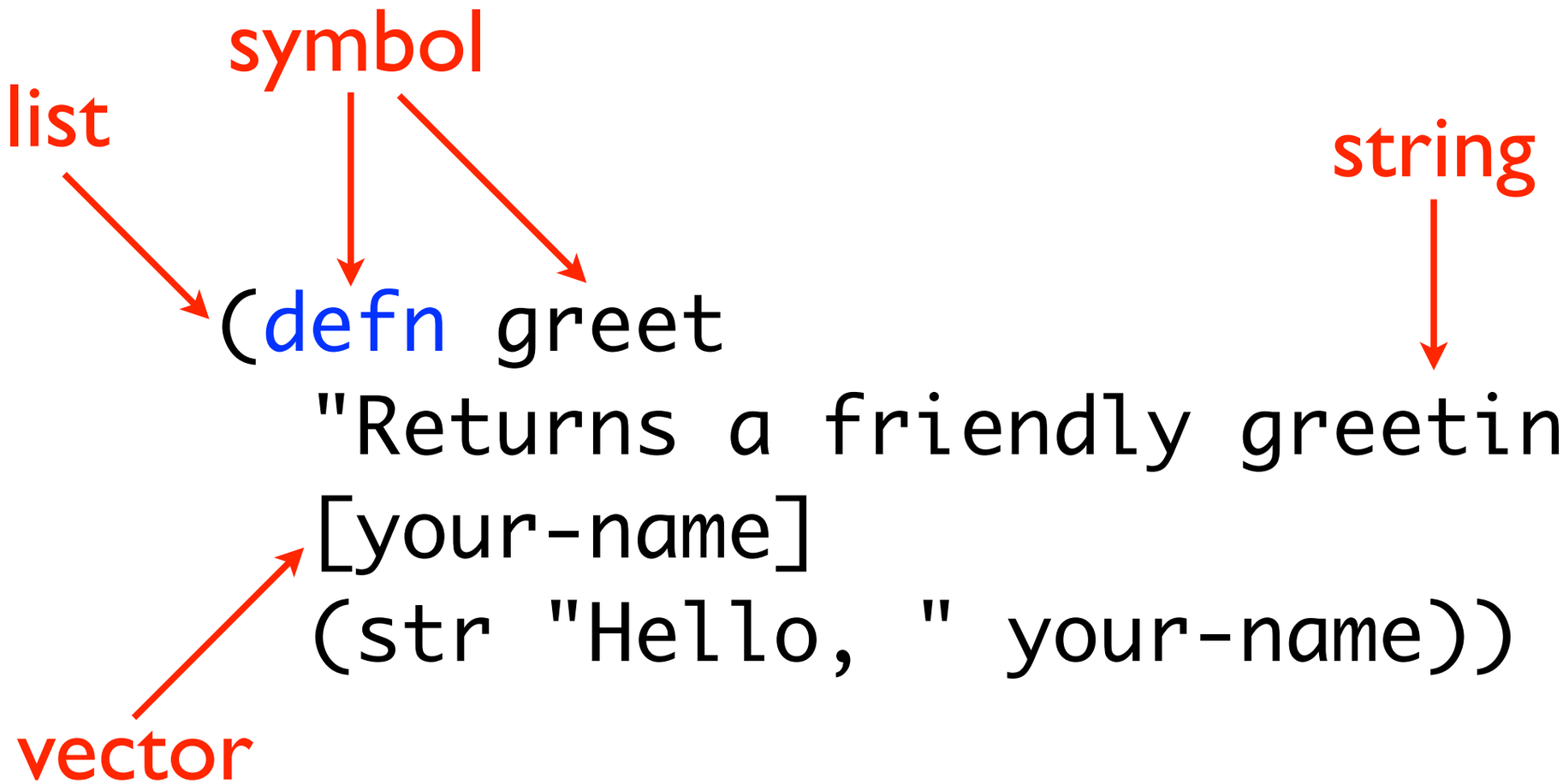
Semantics



defn Semantics



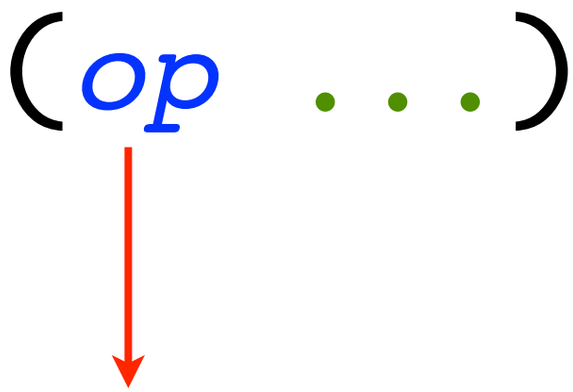
defn Structure



All forms created equal

form	syntax	example
function	list	`(println "hello")`
"operator"	list	`(+ 1 2)`
method call	list	`(.trim " hello ")`
loading	list	`(require 'mylib)`
metadata	list	`(with-meta obj m)`
control flow	list	`(when valid? (proceed))`
scope	list	`(let [x 5] ...)`

Invocation Forms



- Special Form
- Macro
- Function
  or any invocable thing
- Expression which returns
  an invocable thing

Invocation Forms

```
user=> (1 2 3)
ClassCastException java.lang.Long
cannot be cast to clojure.lang.IFn
```

Invocation Forms

not an invocable thing

```
user=> (1 2 3)
ClassCastException java.lang.Long
cannot be cast to clojure.lang.IFn
```

invocable things

14 Special Forms

```
def if fn let
loop recur do
new . throw try
set! quote var
```

Special Form: def

```
(def symbol value-expr?)
```

Special Form: def

```
user=> (def answer (* 6 7))
#'user/answer
user=> answer
42
```

Special Form: if

```
(if condition
    then-expr
    else-expr?)
```

Special Form: if

```
user=> (if (even? 42) "even" "odd")
"even"
user=> (if (even? 7) "even")
nil
```

Truthiness

	Clojure	Java	Common Lisp	Scheme
null value	nil	null	nil or ()	
boolean values	true / false	true / false	t / nil	#t / #f
"falsey"	nil or false	false	nil or ()	#f
"truthy"	everything else	true	everything else	everything else

Special Form: do

```
(do exprs*)
```

Special Form: do

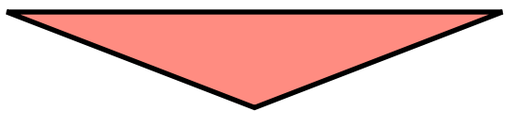
```
user=> (do 1 2 3)
3
user=> (do (println 42 "is even")
           42)
42 is even
42
```

Special Form: do

```
user=> (if (even? 42)
           (do (println 42 "is even")
               42)))
42 is even
42
```

Macro: when

```
(when condition
  exprs*)
```



```
(if condition
  (do exprs*))
```

Macro: when

```
user=> (when (even? 42)
         (println 42 "is even")
         42)
42 is even
42
```

Special Form: quote

```
(quote form)
```

```
'form
```

Special Form: quote

```
user=> (1 2 3)
ClassCastException java.lang.Long
cannot be cast to clojure.lang.IFn
```

```
user=> (quote (1 2 3))
(1 2 3)
user=> '(1 2 3)
(1 2 3)
user=> (list 1 2 3)
(1 2 3)
```

Special Form: let

```
(let [bindings*]
  exprs*)
```

Special Form: let

Binding forms

```
user=> (let [x 6, y 7] (* x y))
42
```

Value expressions

Special Form: let

Earlier bindings available

```
user=> (let [x 6
             y (+ x 1)]
         (* x y))
42
```

Special Form: let

```
user=> (let [x 6
              y (+ x 1)]
          (println (* x y))
          (let [y 10]
            (println (* x y))))
42
60
nil
```

Rebinding is allowed

Special Form: let

```
user=> (let [x 5]
         (let [y x]
           (let [y y]
             (let [z (+ x y)]
               z))))
```

Special Form: fn

```
(fn [params*] exprs*)

(fn ([params*] exprs*)+)
```

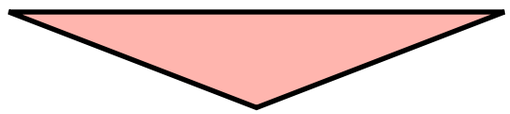
Special Form: fn

compiled fn

```
user=> (fn [x] (+ x 5))
#<user$eval1456$fn__1457>
user=> (def add5 (fn [x] (+ x 5)))
#'user/add5
user=> (add5 10)
15
```

Macro: defn

```
(defn symbol [params*] exprs*)
```



```
(def symbol (fn [params*] exprs*))
```

Arity

```
user=> (defn greet
         ([] (greet "Clojure" "programmer"))
         ([name] (greet name "the programmer"))
         ([first-name last-name]
           (println "Hello," first-name last-name)))
#'user/greet
user=> (greet)
Hello, Clojure programmer
nil
user=> (greet "Stuart")
Hello, Stuart the programmer
nil
user=> (greet "Stuart" "Sierra")
Hello, Stuart Sierra
nil
```

Variadic

```
user=> (defn greet
         ([name] (println "Hello," name))
         ([name & others]
            (println "Hello," name "and friends" others)))
#'user/greet
user=> (greet "Stuart")
Hello, Stuart
nil
user=> (greet "Stuart" "Aaron" "Fogus" "Chouser")
Hello, Stuart and friends (Aaron Fogus Chouser)
nil
```

Working with Data

Value Equality

```
user=> (= [1 2 3] [1 2 3])
true
user=> (= [1 2 3] [1.0 2.0 3.0])
true
user=> (= [1 2 3] '(1 2 3))
true
```

Reference Equality

```
user=> (= (StringBuilder. "a")
          (StringBuilder. "a"))
false
user=> (identical? [1 2 3] [1 2 3])
false
user=> (identical? 2.0 2)
false
```

Reference Equality

```
user=> (= :a :a)
true
user=> (identical? :a :a)
true
user=> (= 'a 'a)
true
user=> (identical? 'a 'a)
false
```

Collections

```
user=> (count [:a :b :c])
3
user=> (coll? [:a :b :c])
true
user=> (vector? [:a :b :c])
true
user=> (list? [:a :b :c])
false
```

Sequential Things

- List
- Vector

Sequential Things

```
user=> (def v [:a :b :c :d])
#'user/v
user=> (first v)
:a
user=> (second v)
:b
user=> (def w (vector 10 11 12 13))
#'user/w
user=> (nth w 3)
13
```

conj

```
user=> (def v [:a :b :c :d])
#'user/v
user=> (conj v :more)
[:a :b :c :d :more]
```

Vectors grow at the end

```
user=> (def lst (list 1 2 3 4))
#'user/lst
user=> (conj lst :more)
(:more 1 2 3 4)
```

Lists grow at the front

Associative Things

- Map
  - Hash map
  - Array map
- Vector (sometimes)

Maps: get

```
user=> (def m {:a 1 :b 2})
#'user/m
user=> (get m :a)
1
user=> (m :a)
1
user=> (:a m)
1
```

Maps are invocable

So are keywords

Maps: assoc

```
user=> (def m (hash-map :a 1 :b 2))
#'user/m
user=> (assoc m :c 3 :d 4)
{:a 1, :c 3, :b 2, :d 4}
user=> (assoc m :a 17)
{:a 17, :b 2}
```

new map
with added
key-value pairs

Maps

```
user=> (def m {:a 1 :b 2})
#'user/m
user=> (dissoc m :a)
{:b 2}
user=> (keys m)
(:a :b)
user=> (conj m [:c 3])
{:a 1, :c 3, :b 2}
```

conj on a map
takes a vector pair

Maps: merge

```
user=> (def m {:a 1 :b 2})
#'user/m
user=> (merge m {:b 17 :c 22})
{:a 1, :c 22, :b 17}
```

Maps: zipmap

```
user=> (zipmap [:a :b :c] [1 2 3])
{:c 3, :b 2, :a 1}
```

Maps: get-in

```
user=> (def person
         {:name {:first "Stuart"
                 :last "Sierra"}
          :address {:city "Brooklyn"
                    :state "New York"}})
#'user/person
user=> (:first (:name person))
"Stuart"
user=> (get-in person [:name :first])
"Stuart"
```

Maps: assoc-in

```
user=> (def person
         {:name {:first "Stuart"
                 :last "Sierra"}
          :address {:city "Brooklyn"
                    :state "New York"}})
#'user/person
user=> (assoc-in person [:name :last] "Halloway")
{:name {:last "Halloway", :first "Stuart"},
 :address {:state "New York", :city "Brooklyn"}}
```

Vectors are Associative

```
user=> (def v [:a :b :c])
#'user/v
user=> (assoc v 1 "Hello")
[:a "Hello" :c]
```

Sets

```
user=> (def s #{1 2 3})
#'user/s
user=> s
#{1 2 3}
user=> (conj s 17)
#{1 2 3 17}
user=> (conj s 2)
#{1 2 3}
```

Sets

```
user=> (def v [1 2 3 1 2 3 4])
#'user/v
user=> (set v)
#{1 2 3 4}
```

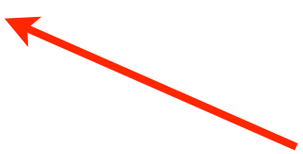
Sets: contains?

```
user=> (def s #{1 2 3})
#'user/s
user=> (contains? s 3)
true
user=> (contains? s 17)
false
```

Checks keys,
not values

Sets: invocation

```
user=> (def s #{1 2 3})
#'user/s
user=> (s 3)
3
user=> (s 17)
nil
```



Returns object
if it's in the set

Destructuring

Destructuring

```
user=> (def stuff [7 8 9 10])
#'user/stuff
```

```
user=> (let [[a b c d] stuff]
         (list (+ a b) (+ c d)))
(15 19)
```

Destructuring

```
(let [bindings*]
  exprs*)
```

Destructuring

```
(let [binding-form value-expr]
  exprs*)
```

Sequential Destructuring

Binding form

Value expression

```
(let [[a b c d] stuff]
  exprs*)
```

Sequential Destructuring

```
(let [[a b c d] stuff] ...
```

Evaluate

```
(let [[a b c d] [7 8 9 10]] ...
```

```
(let [a 7
      b 8
      c 9
      d 10] ...
```

Destructure

Sequential Destructuring

```
user=> (def stuff [7 8 9 10])
#'user/stuff
user=> (let [[a & others] stuff]
         (println a)
         (println others))
7
(8 9 10)
nil
```

Sequential Destructuring

Get the first one or more elements

Get all remaining elements

```
(let [[symbols+ & symbol] expr]
  exprs*)
```

Associative Destructuring

```
user=> (def m {:a 7 :b 4})
#'user/m
user=> (let [{a :a, b :b} m]
         [a b])
[7 4]
```

Associative Destructuring

```
user=> (def m {:a 7 :b 4})
#'user/m
user=> (let [{:keys [a b]} m]
         [a b])
[7 4]
```

Keyword Destructuring

```
user=> (def m {:a 7 :b 4})
#'user/m
user=> (let [{:keys [a b c]} m]
         [a b c])
[7 4 nil]
```

Destructuring with Default Values

```
user=> (def m {:a 7 :b 4})
#'user/m
user=> (let [{:keys [a b c]
              :or {c 3}} m]
          [a b c])
[7 4 3]
```

Destructuring

```
(fn [params*]
  exprs*)
```

Destructuring works here too

```
(defn symbol [params*]
  exprs*)
```

Nested Destructuring

```
user=> (def data {:a [1 2 3] :b {:c 4}})
#'user/data

user=> (let [{[x & y] :a {:keys [c d] :or {d 10}} :b} data]
         (+ x c d))
```

???

Higher-Order Functions

apply

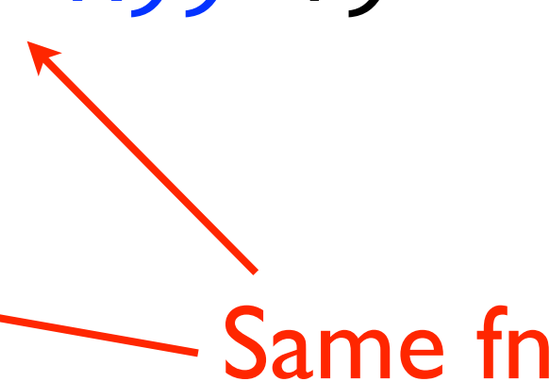
```
user=> (def v [1 2 3 4])
#'user/v
user=> (println v)
[1 2 3 4]
nil
user=> (apply println v)
1 2 3 4
nil
user=> (println 1 2 3 4)
1 2 3 4
nil
```

map

```
user=> (def v [1 2 3 4])
#'user/v
user=> (map (fn [x] (* 5 x)) v)
(5 10 15 20)
```

#() syntax

```
user=> (map (fn [x] (* 5 x)) v)
(5 10 15 20)
user=> (map #(* 5 %) v)
(5 10 15 20)
```



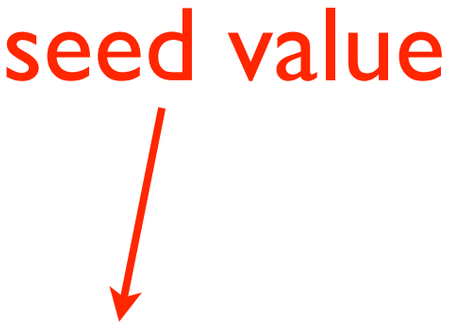
map

```
user=> (def v [:a :b :c])
#'user/v
user=> (def w [10 20 30 40 50])
#'user/n
user=> (map vector v w)
([:a 10] [:b 20] [:c 30])
```

reduce

```
user=> (reduce + [1 2 3 4])
10
user=> (+ (+ (+ 1 2) 3) 4)
10
```

reduce




```
user=> (reduce + 100 [1 2 3 4])
110
user=> (+ (+ (+ (+ 100 1) 2) 3) 4)
110
```

reduce

destructure
key-value pair

seed value

```
user=> (reduce (fn [m [k v]]
                 (assoc m (name k) (+ 100 v)))
               {}
               {:a 1 :b 2})
{"b" 102, "a" 101}
```

filter and remove

```
user=> (filter even? [1 2 3 4 5])
(2 4)
user=> (filter (set "aeiou") "Hello, World!")
(\e \o \o)
user=> (apply str (remove (set "aeiou") "Hello, World!"))
"Hll, Wrld!"
```

Maps: assoc-in

```
user=> (def person
         {:name {:first "Stuart"
                 :last "Sierra"}
          :address {:city "Brooklyn"
                    :state "New York"}})
#'user/person
user=> (assoc-in person [:name :last] "Halloway")
{:name {:last "Halloway", :first "Stuart"},
 :address {:state "New York", :city "Brooklyn"}}
```

Maps: update-in

```
user=> (def person
         {:name {:first "Stuart"
                 :last "Sierra"}
          :address {:city "Brooklyn"
                    :state "New York"}})
#'user/person
user=> (update-in person [:name :last] #(.toUpperCase %))
{:name {:last "SIERRA", :first "Stuart"},
 :address {:state "New York", :city "Brooklyn"}}
```

identity

```
user=> (doc identity)
-------------------------
clojure.core/identity
([x])
  Returns its argument.
nil
```

filter and identity

```
user=> (def v [1 2 3 nil 4 5 nil 6])
#'user/v
user=> (filter identity v)
(1 2 3 4 5 6)
```

fnil

```
user=> (doc fnil)
-------------------------
clojure.core/fnil
([f x] [f x y] [f x y z])
  Takes a function f, and returns a function that calls f,
replacing a nil first argument to f with the supplied value x.
```

fnil

```
user=> (+ nil 3)
NullPointerException   clojure.lang.Numbers.ops
user=> (def add (fnil + 0))
#'user/add
user=> (add nil 3)
3
```

update-in and fnil

```
user=> (def m {:foo {:a 1 :b 2} :bar {:c 3}})
#'user/m
user=> (update-in m [:foo :a] inc)
{:foo {:a 2, :b 2}, :bar {:c 3}}
user=> (update-in m [:baz :d] inc)
NullPointerException   clojure.lang.Numbers.ops
user=> (update-in m [:baz :d] (fnil inc 0))
{:foo {:a 1, :b 2}, :bar {:c 3}, :baz {:d 1}}
```

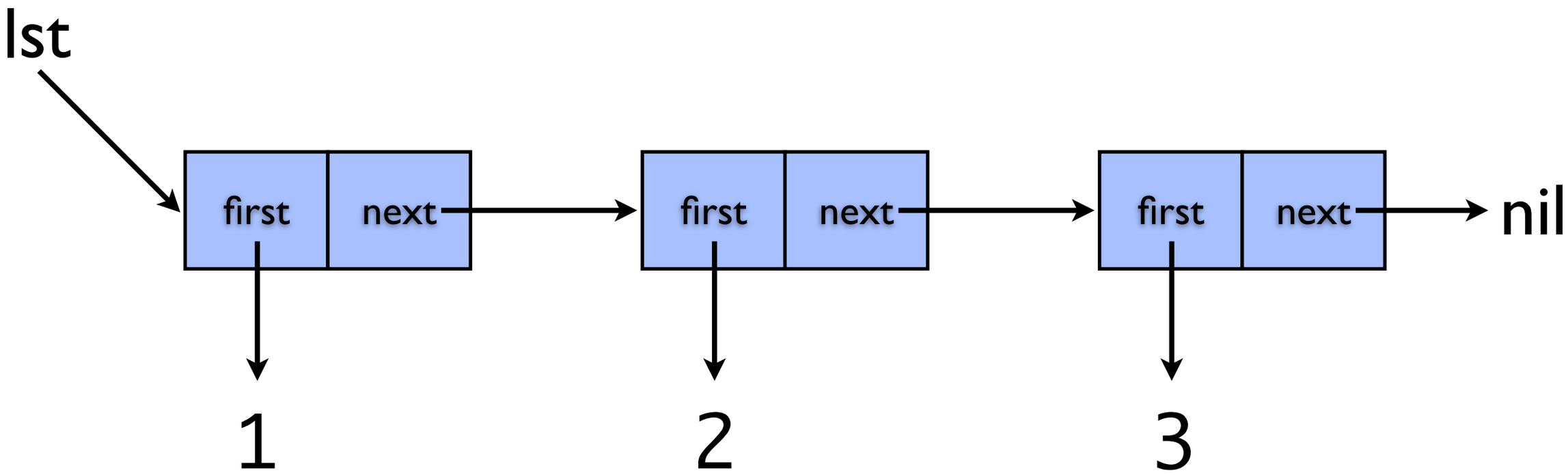
frequencies

```
user=> (reduce (fn [m c]
                 (update-in m [c] (fnil inc 0)))
               {}
               "a short sharp shock")
{\space 3, \a 2, \c 1, \h 3, \k 1, \o 2,
 \p 1, \r 2, \s 3, \t 1}

user=> (frequencies "a short sharp shock")
{\space 3, \a 2, \c 1, \h 3, \k 1, \o 2,
 \p 1, \r 2, \s 3, \t 1}
```

Sequences

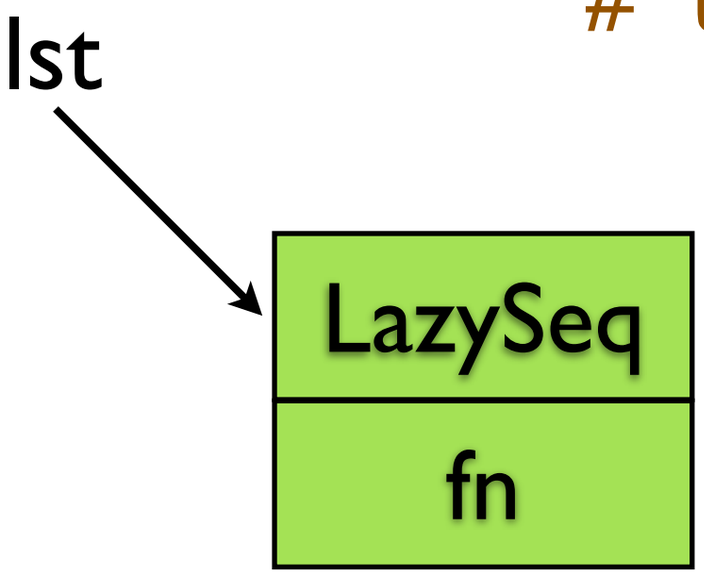
Lists

```
(def lst '(1 2 3))
```



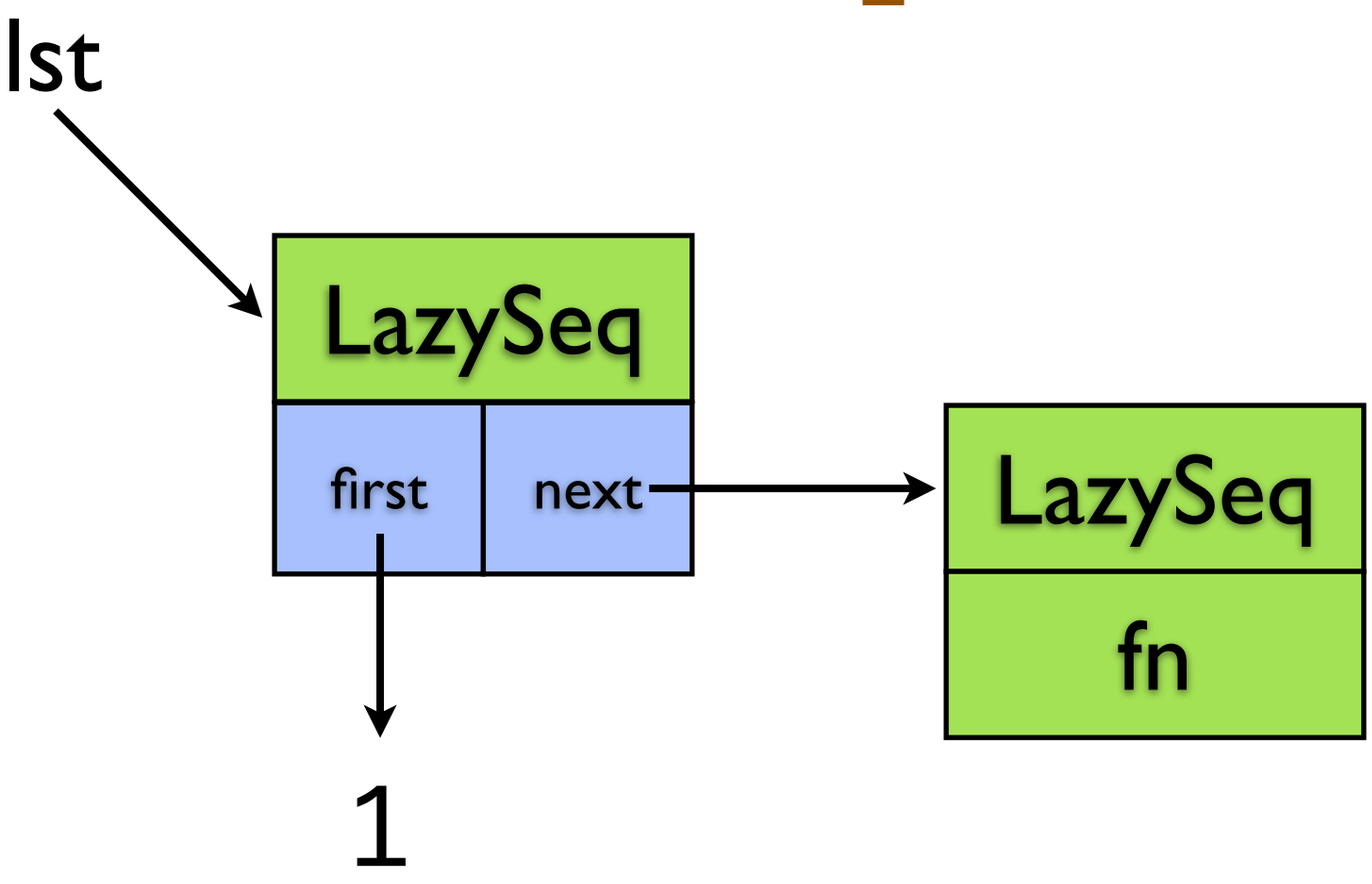
Sequences

```
(def lst (range 1 4))
#'user/lst
```



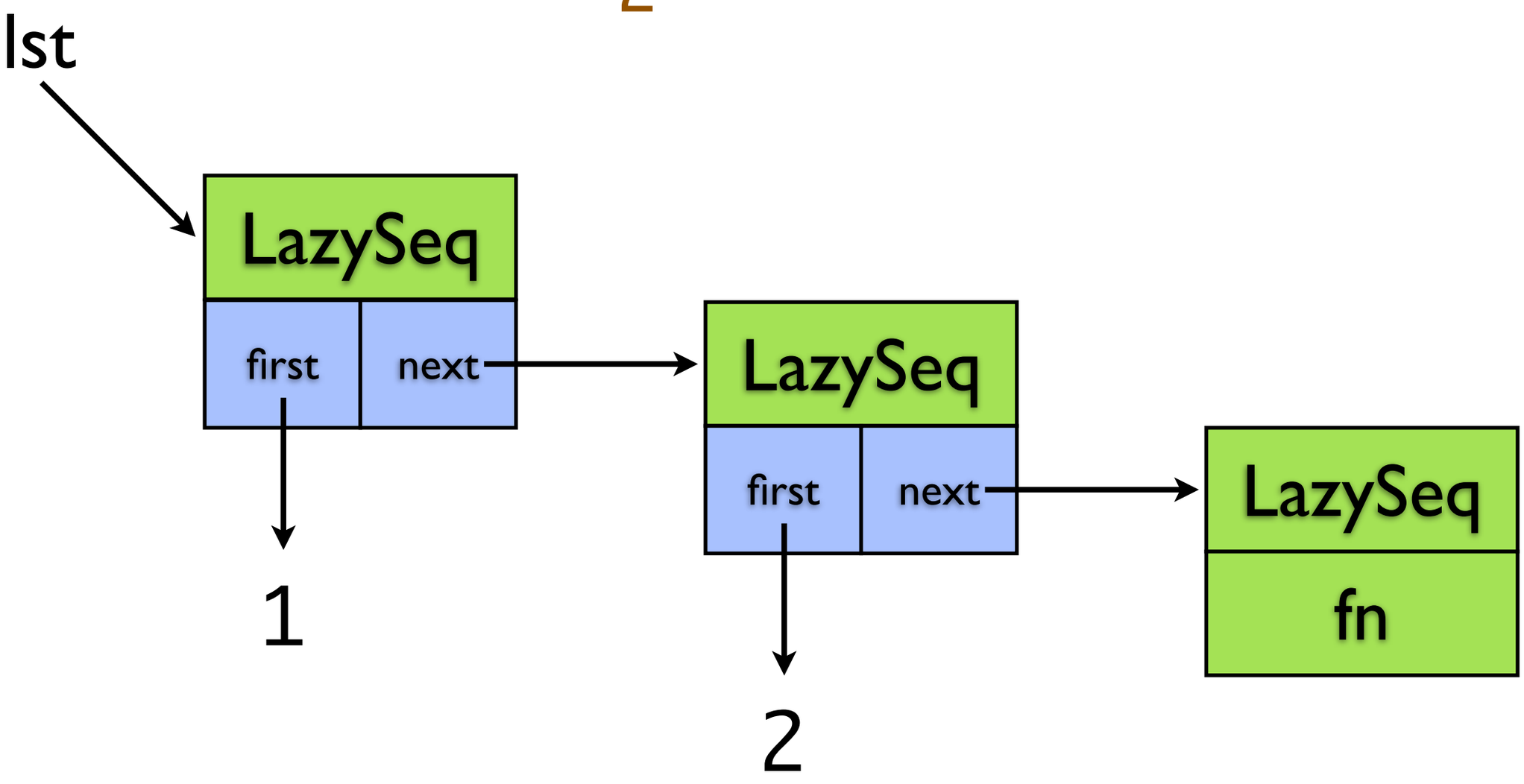
Sequences

```
(first lst)
1
```



Sequences

```
(second lst)
2
```



range

infinite!

```
(range)
```

```
(range end)
```

```
(range start end)
```

```
(range start end step)
```

Infinite Sequences

```
user=> (map vector (range) [:a :b :c])
([0 :a] [1 :b] [2 :c])

user=> (map-indexed vector [:a :b :c])
([0 :a] [1 :b] [2 :c])

user=> (zipmap "abc" (range))
{\c 2, \b 1, \a 0}
```

repeat

```
user=> (repeat 5 :a)
(:a :a :a :a :a)

user=> (take 10 (repeat :b))
(:b :b :b :b :b :b :b :b :b :b)
```

infinite!

seq

```
user=> (seq [1 2 3])
(1 2 3)
user=> (seq {:a 1 :b 2 :c 3})
([:a 1] [:c 3] [:b 2])
user=> (seq #{1 2 3})
(1 2 3)
user=> (seq [])
nil
```

Lazy vs. Strict

```
(with-open [file ...]
  (line-seq file))
```

returns lazy seq of lines in file

file is closed here

Lazy vs. Strict

```
(with-open [file ...]
  (doall (line-seq file)))
```

force entire seq
to be realized

doall, dorun, doseq

```
(doall sequence)
```
returns sequence

```
(dorun sequence)
```
returns nil

```
(doseq [symbol sequence]
  ...body...)
```
returns nil

Java Interop

Java Types

Clojure Literal	Java Type
`"hello"`	java.lang.String
`3.14`	java.lang.Double
`[1 2 3]`	java.util.List
`{"a" 1, "b" 2}`	java.util.Map
`(fn [x] (* x 5))`	java.lang.Runnable

class and ancestors

```
user=> (class "hello")
java.lang.String
user=> (ancestors (class "hello"))
#{java.lang.CharSequence java.lang.Comparable
  java.lang.Object java.io.Serializable}
```

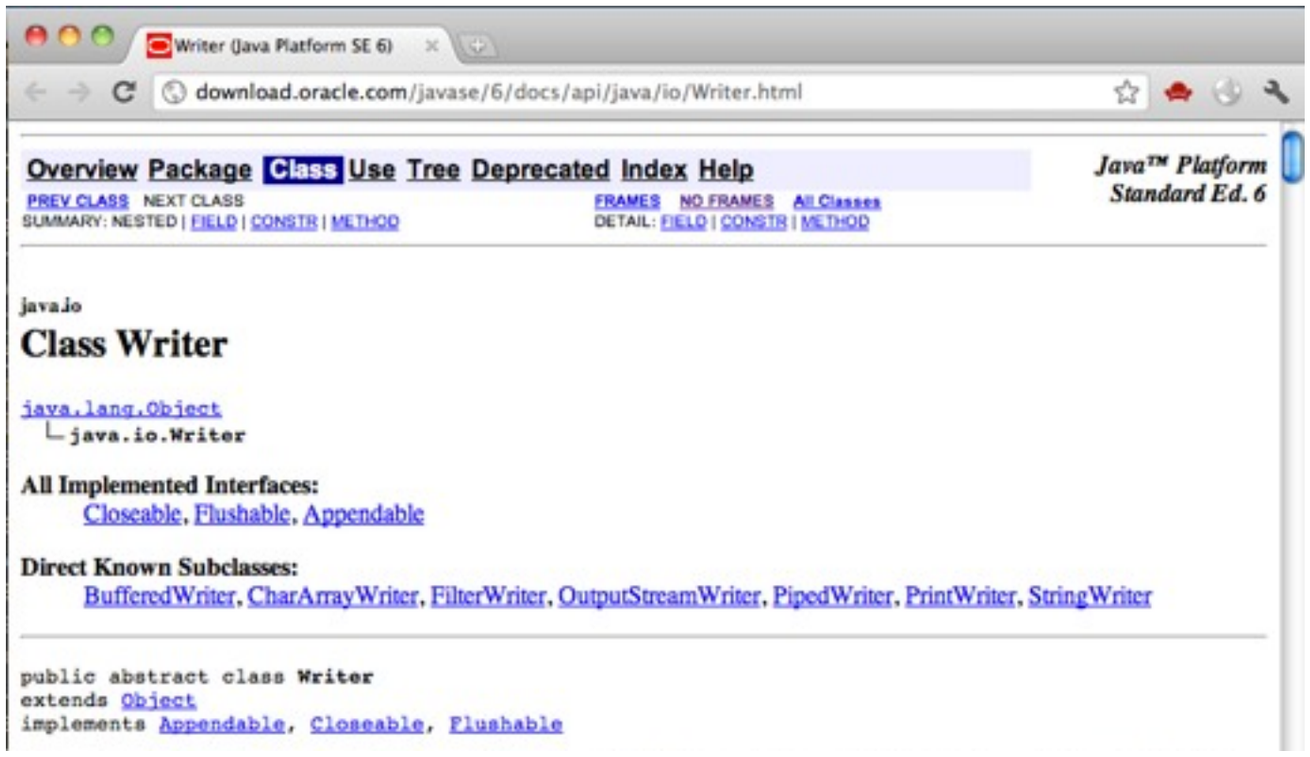
REPL Helpers: javadoc

Java class

```
user=> (javadoc java.io.Writer)
"http://java.sun.com/javase/6/docs/api/java/io/Writer.html"
```

Object

```
user=> (javadoc "hello")
"http://java.sun.com/javase/6/docs/api/java/lang/String.html"
```



Special Form: . (dot)

```
(. object method arguments*)

(. object field)

(. class static-method arguments*)

(. class static-field)
```

Instance Method Calls

object method arguments

```
user=> (. "hello" charAt 0)
\h

user=> (.charAt "hello" 0)
\h
```

Syntactic sugar

Static Method Calls

```
user=> (. Long valueOf "42")
42

user=> (Long/valueOf "42")
42
```

Syntactic sugar

Static Fields

```
user=> (. Math PI)
3.141592653589793
```

```
user=> Math/PI
3.141592653589793
```

Syntactic sugar

Constructors

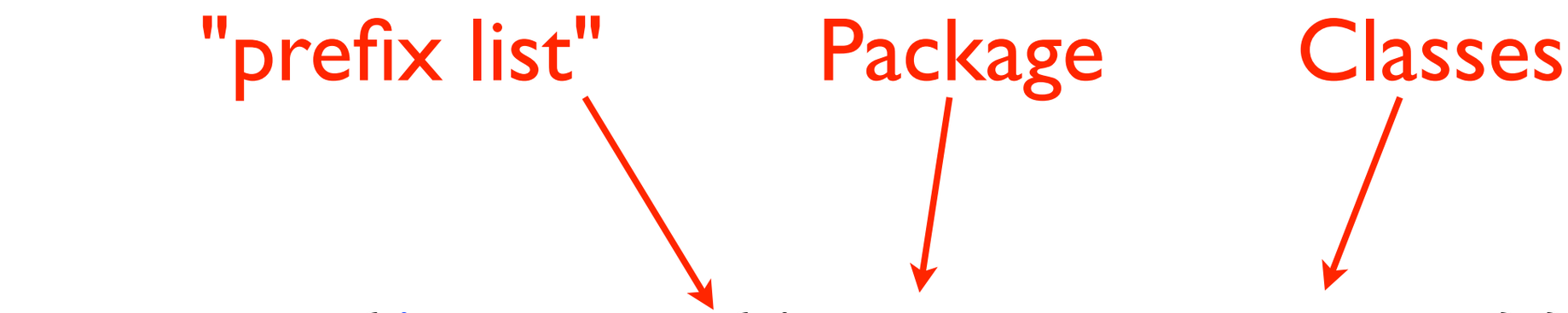
class name

Constructor arguments

```
user=> (new java.io.File "/home")
#<File /home>

user=> (java.io.File. "/home")
#<File /home>
```

Syntactic sugar

Java Import




```
user=> (import (java.net URL URI))
java.net.URI
user=> (URL. "http://clojure.org/")
#<URL http://clojure.org/>
```

try/catch/finally

```
(try
    ... expressions ...
    (catch Class name
        ... handle exception ...)
    (finally
        ... always do this ...))
```

throw

```
(throw exception)
```

throw and *e

constructor

```
user=> (throw (Exception. "Boom!"))
Exception Boom!  user/eval11 (NO_SOURCE_FILE:5)
user=> *e
#<RuntimeException java.lang.RuntimeException:
java.lang.Exception: Boom!>
```

print stack trace

defined in clojure.repl

```
user=> (pst)
Exception Boom!
    user/eval11 (NO_SOURCE_FILE:5)
    clojure.lang.Compiler.eval (Compiler.java:6424)
    clojure.lang.Compiler.eval (Compiler.java:6390)
    clojure.core/eval (core.clj:2795)
    clojure.main/repl/read-eval-print--5967 (main.clj:244)
```

Java Interop Summary

	Special Form	Sugar
Instance Method	`(. obj method args*)`	`(.method obj args*)`
Instance Field	`(. obj field)`	`(.field obj)`
Static Method	`(. Class method args*)`	`(Class/method args*)`
Static Field	`(. Class field)`	`Class/field`
Constructor	`(new Class args*)`	`(Class. args*)`

Java Nested Classes

	Package-Qualified Class Name	Unqualified Class Name
Regular Class	`java.io.File`	`File`
Inner Class	`java.util.Map$Entry`	`Map$Entry`

Libraries

Java Classpath

Clojure source code in ./src/

JAR files in ./lib/

```
java -cp src:lib/* clojure.main
```

classpath

Project Management

- Leiningen
- Cake
- Maven
- Ant

Maven
Directory Structure

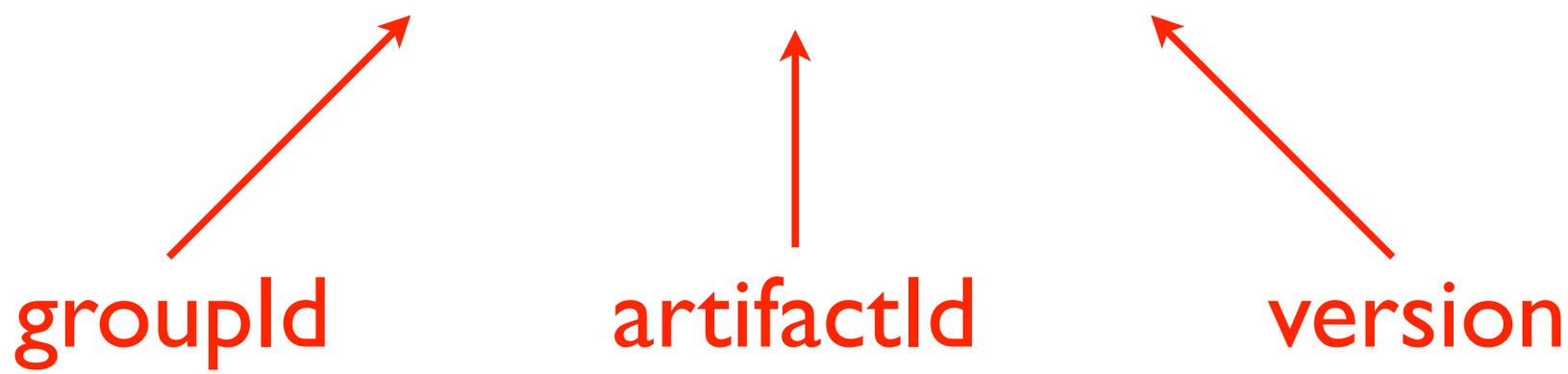
```
pom.xml
src/main/clojure/
     your/project/file.clj
src/test/clojure/
     your/project/file_test.clj
target/classes/
     compiled_file.class
$HOME/.m2/repository/
     org/clojure/clojure/1.3.0/clojure-1.3.0.jar
```

Leiningen
Directory Structure

```
project.clj
src/
    your/project/file.clj
test/
    your/project/file_test.clj
classes/
    compiled_file.class
lib/
    clojure-1.3.0.jar
```

Leiningen project.clj

```
(defproject my.project.name "1.0.0"
  :description "My cool Clojure project."
  :dependencies [[org.clojure/clojure "1.3.0"]
                 [org.clojure/data.zip "0.1.0"]])
```



Leiningen Commands

```
$ lein help
Leiningen is a build tool for Clojure.

Several tasks are available:
  pom
  help
  jar
  test
  deps
  ...
  repl
  new

Run lein help $TASK for details.
See http://github.com/technomancy/leiningen as well.
```

Finding Dependencies

- search.maven.org
- jarvana.com
- clojars.org

clojure-contrib up to 1.2

- One big library
- Version number matches Clojure
- Mixed quality

Clojure	clojure-contrib
1.0.0	1.0.0
1.1.0	1.1.0
1.2.0 1.2.1	1.2.0

clojure-contrib after 1.2

- Many small libraries
- Independent version numbers
- Compatible with Clojure 1.2 and 1.3
- Mixed quality

dev.clojure.org/display/doc/Clojure+Contrib

algo.monads
core.incubator
core.logic
core.match
core.unify
data.csv
data.finger-tree
data.json
data.priority-map
data.xml
data.zip
java.classpath
java.data
java.jdbc
java.jmx
math.combinatorics
math.numeric-tower
test.generative
tools.cli
tools.logging
tools.macro
tools.namespace
tools.nrepl
tools.trace

Included in Clojure

- clojure.inspector
- clojure.java.io
- clojure.java.browse
- clojure.java.shell
- clojure.reflect
- clojure.repl
- clojure.string
- clojure.set
- clojure.test
- clojure.walk
- clojure.xml
- clojure.zip

Namespaces

The REPL

"you are here"

```
user=> (+ 1 2 3)
6
user=>
```

Namespaces

```
user=> (ns my.cool.thing)
nil
my.cool.thing=>
```

"you are here"

Qualified Symbols

```
user=> (ns foo.bar)
nil
foo.bar=> (defn hello []
            (println "Hello, World!"))
#'foo.bar/hello
foo.bar=> (ns baz.quux)
nil
baz.quux=> (foo.bar/hello)
Hello, World!
nil
```

Namespace-qualified symbol

Namespaces and Files

dots become slashes

```
$CLASSPATH/my/cool/thing.clj

(ns my.cool.thing)
```

Names and Hyphens

hyphens become underscores

```
$CLASSPATH/my/cool/web_app.clj
```

```
(ns my.cool.web-app)
```

Macro: ns

```
(ns name references*)
```

ns :require

```
(ns my.cool.project
  (:require [some.ns.foo :as foo]))

(foo/function-in-foo)
```

ns alias

ns :use

```
(ns my.cool.project
  (:use [some.ns.foo :only (a b)]))
```

```
(a)
(b)
```

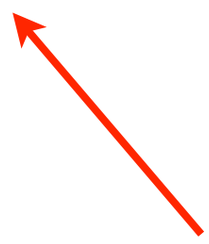
unqualified symbols

list of symbols

ns :use

```
(ns my.cool.project
  (:use some.ns.foo))
```

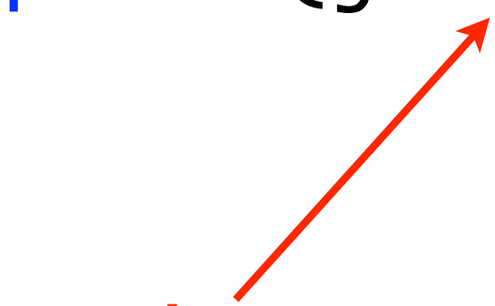
```
(a)
(b)
```



unqualified symbols
from anywhere

ns :import

```
(ns my.cool.project
  (:import (java.io File Writer))
```



package name

class names

Macro: ns

```
(ns name
  (:require [some.ns.foo :as foo]
            [other.ns.bar :as bar])
  (:use [this.ns.baz :only (a b c)]
        [that.ns.quux :only (d e f)])
  (:import (java.io File FileWriter)
           (java.net URL URI)))
```

require in the REPL

arguments must be quoted

```
user=> (require '[clojure.set :as set])
nil
user=> (set/union #{1 2} #{2 3 4})
#{1 2 3 4}
```

use in the REPL

arguments must be quoted

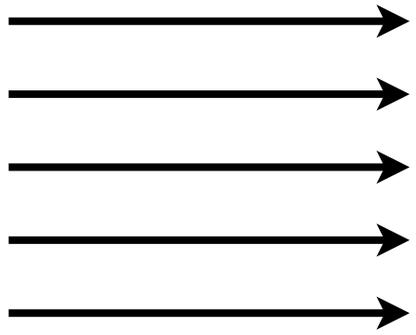
```
user=> (use 'clojure.string)
WARNING: replace already refers to: #'clojure.core/replace in
namespace: user, being replaced by: #'clojure.string/replace
WARNING: reverse already refers to: #'clojure.core/reverse in
namespace: user, being replaced by: #'clojure.string/reverse
nil
user=> (reverse "hello")
"olleh"
```

RELP Helpers: dir

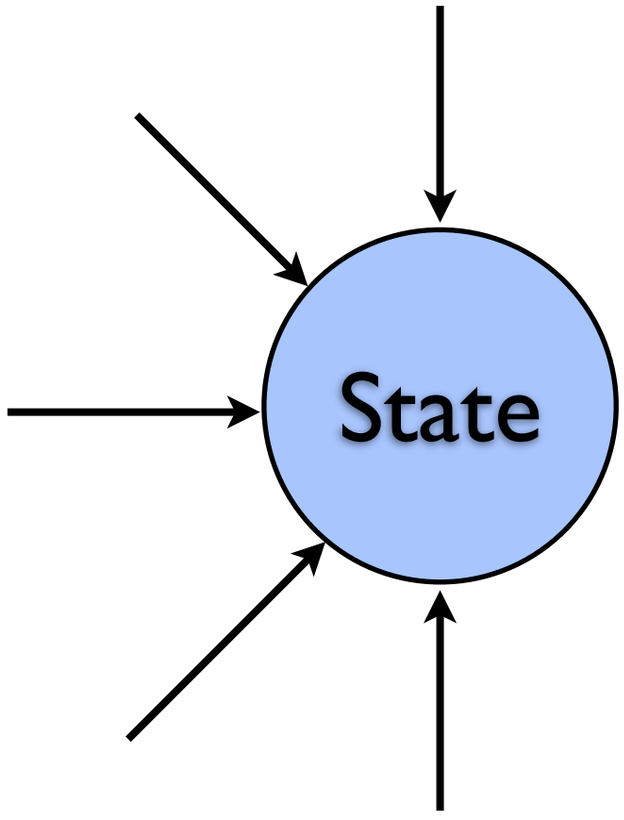
```
user=> (dir clojure.repl)
apropos
demunge
dir
dir-fn
doc
find-doc
pst
root-cause
set-break-handler!
source
source-fn
stack-element-str
thread-stopper
nil
```

Concurrency

Parallelism



Concurrency



```
class Invoice {
  private Date date;

  public Date getDate() {
    return this.date;
  }
  public void setDate(Date date) {
    this.date = date;
  }
}
```

```
class Date {
   public void setDay(int day);
   public void setMonth(int month);
   public void setYear(int year);
}
```

Mutable!

```
class Invoice {
   private Date date;

   public Date getDate() {
      return this.date;
   }
```

Better not change it!

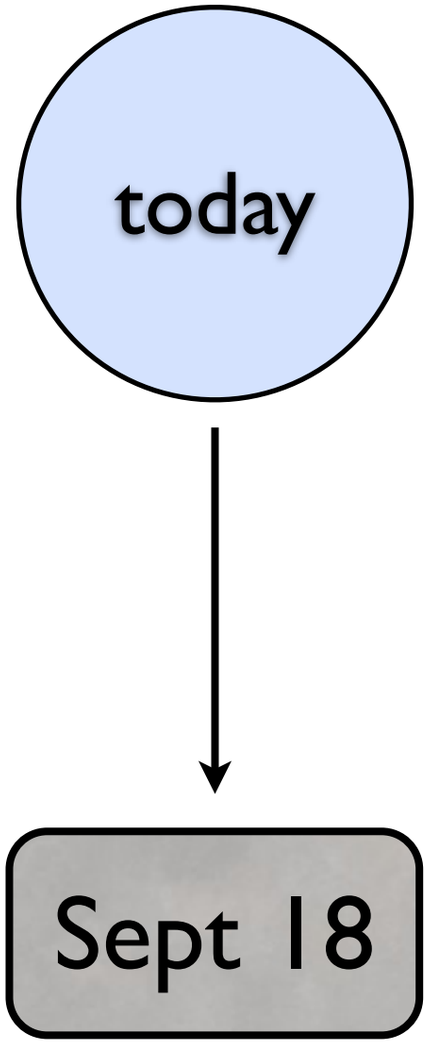
```
   public void setDate(Date date) {
      this.date = date;
   }
}
```

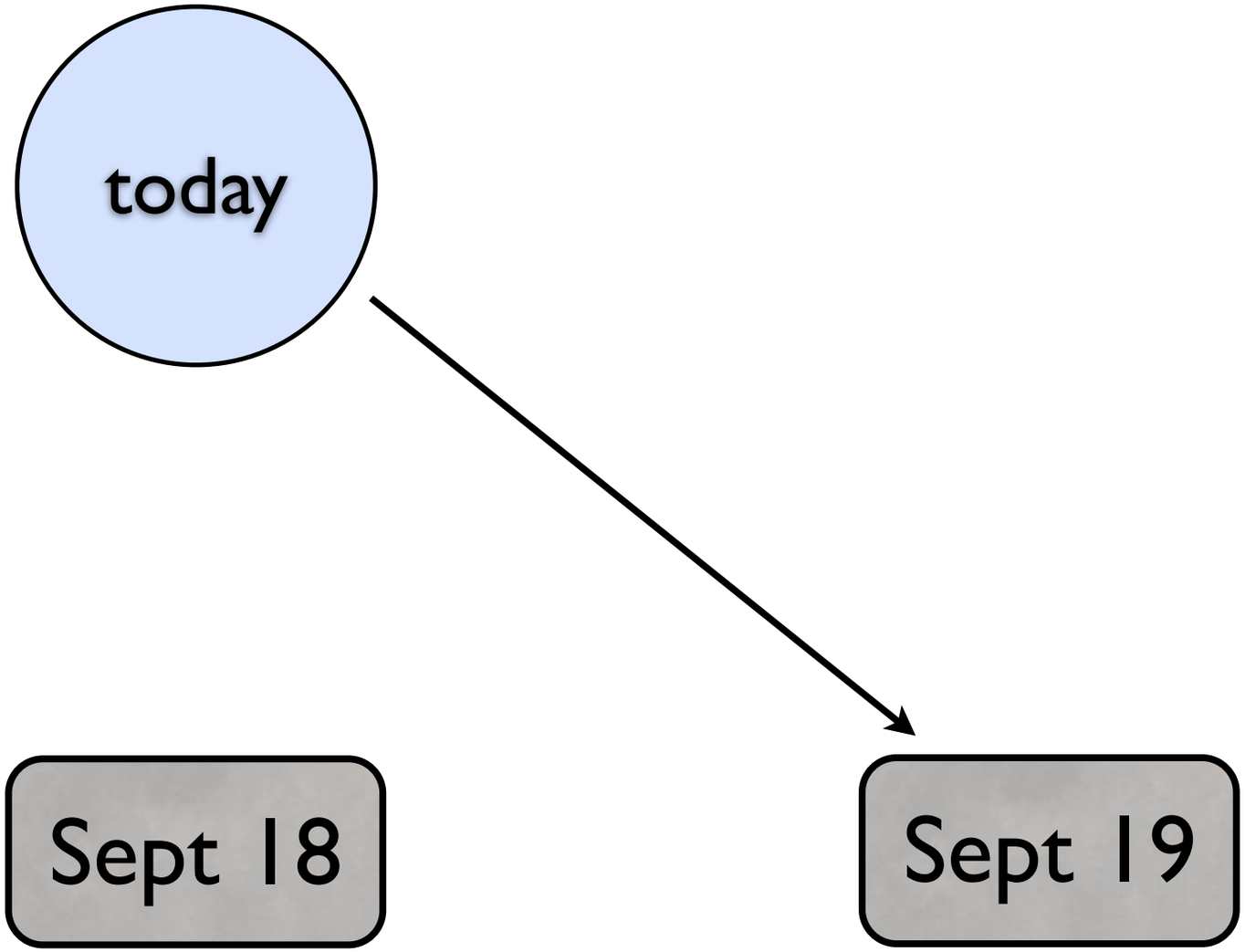
Better not change it!

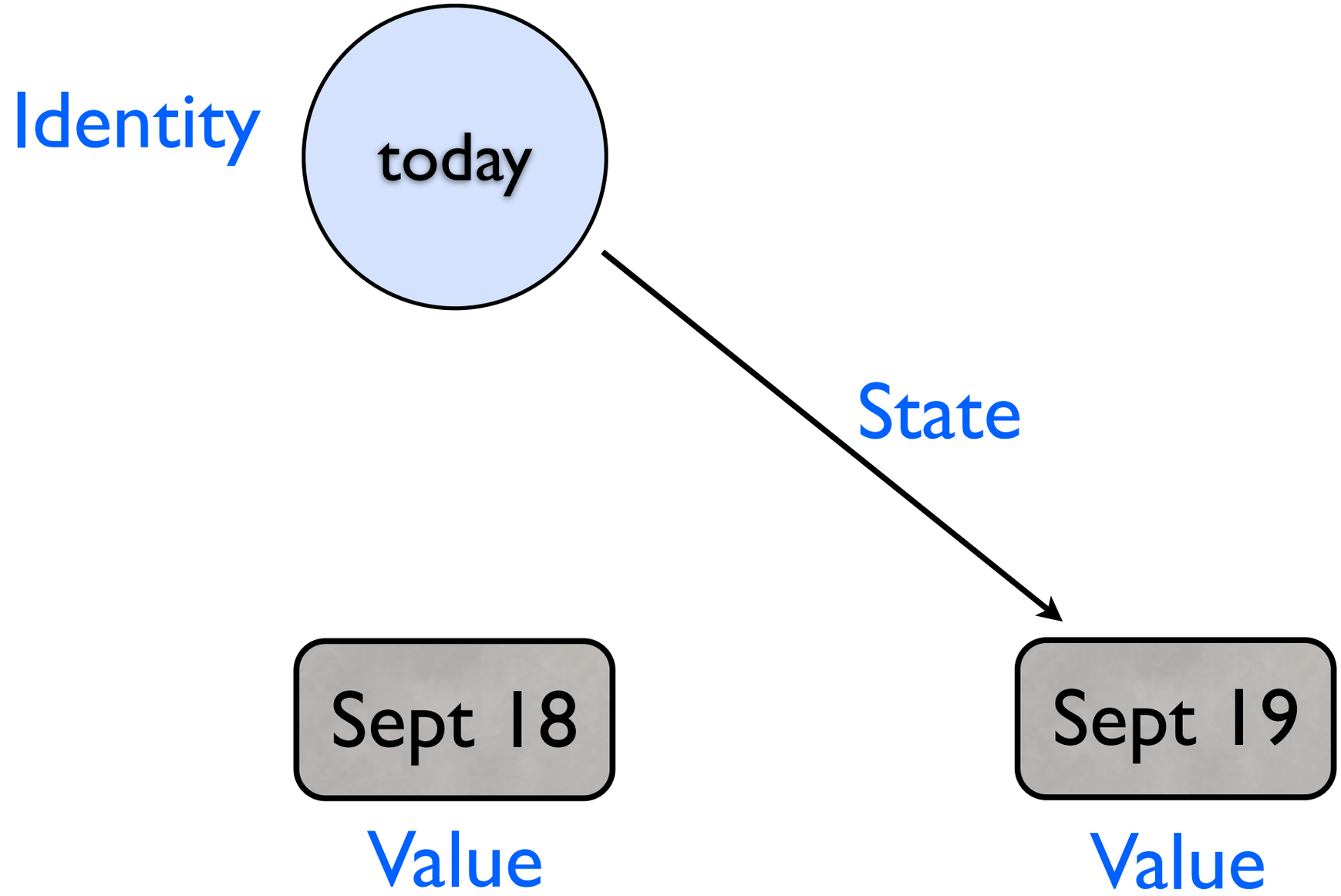
```
class Invoice {
  private Date date;

  public Date getDate() {
    return this.date.clone();
  }

  public void setDate(Date date) {
    this.date = date.clone();
  }
}
```

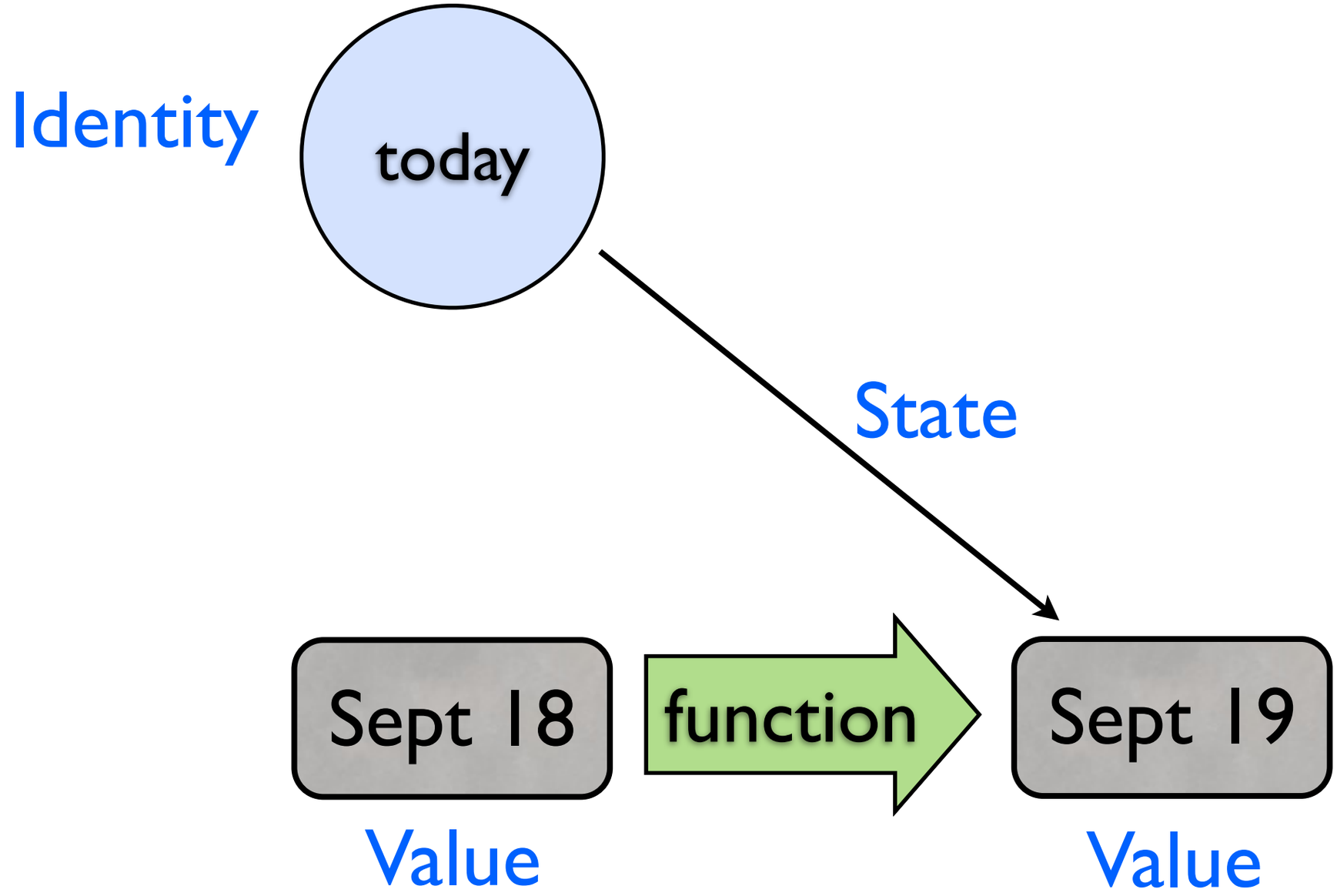
Defensive copying

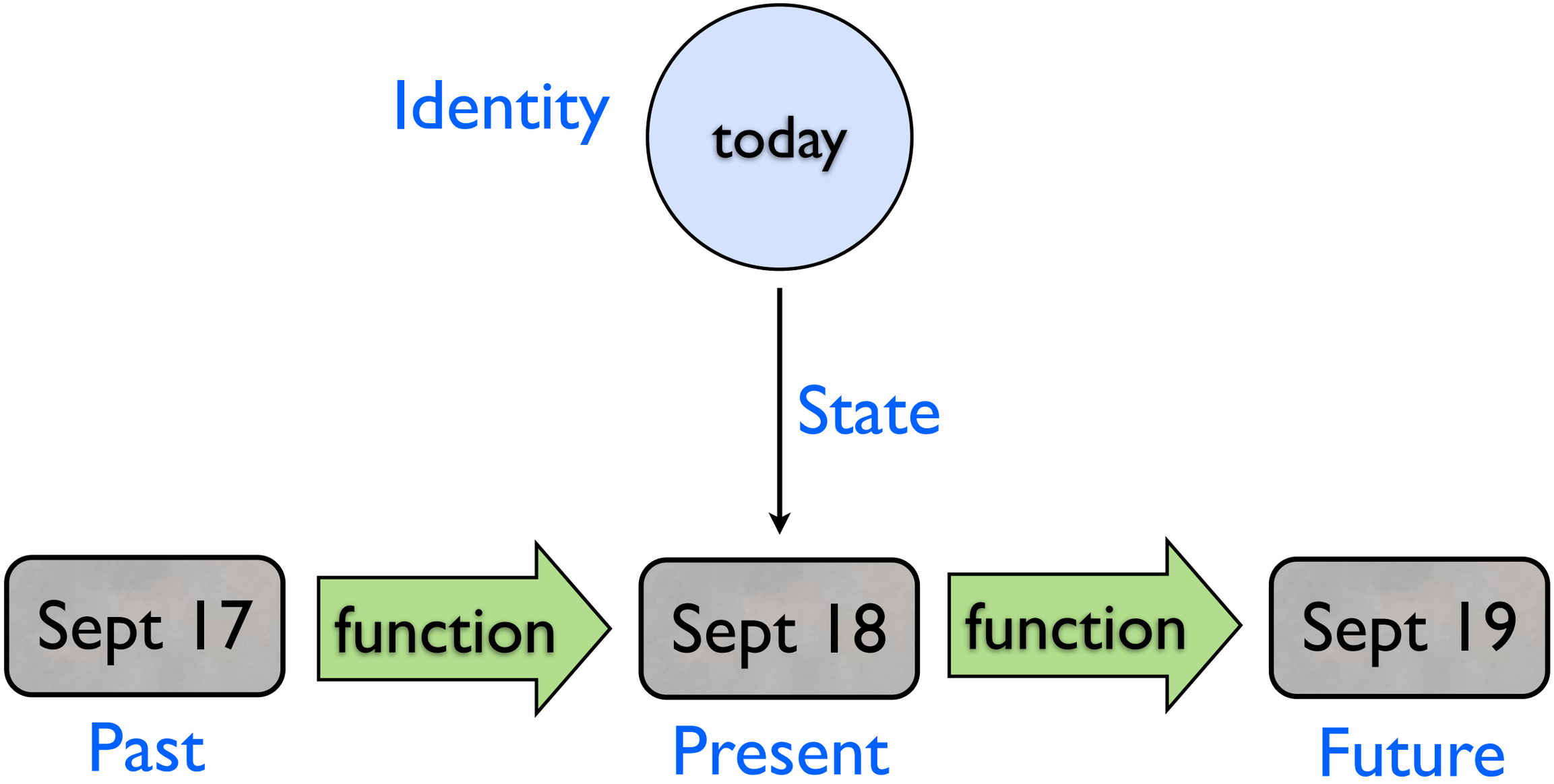
Programming with immutable values means never having to say you're sorry.

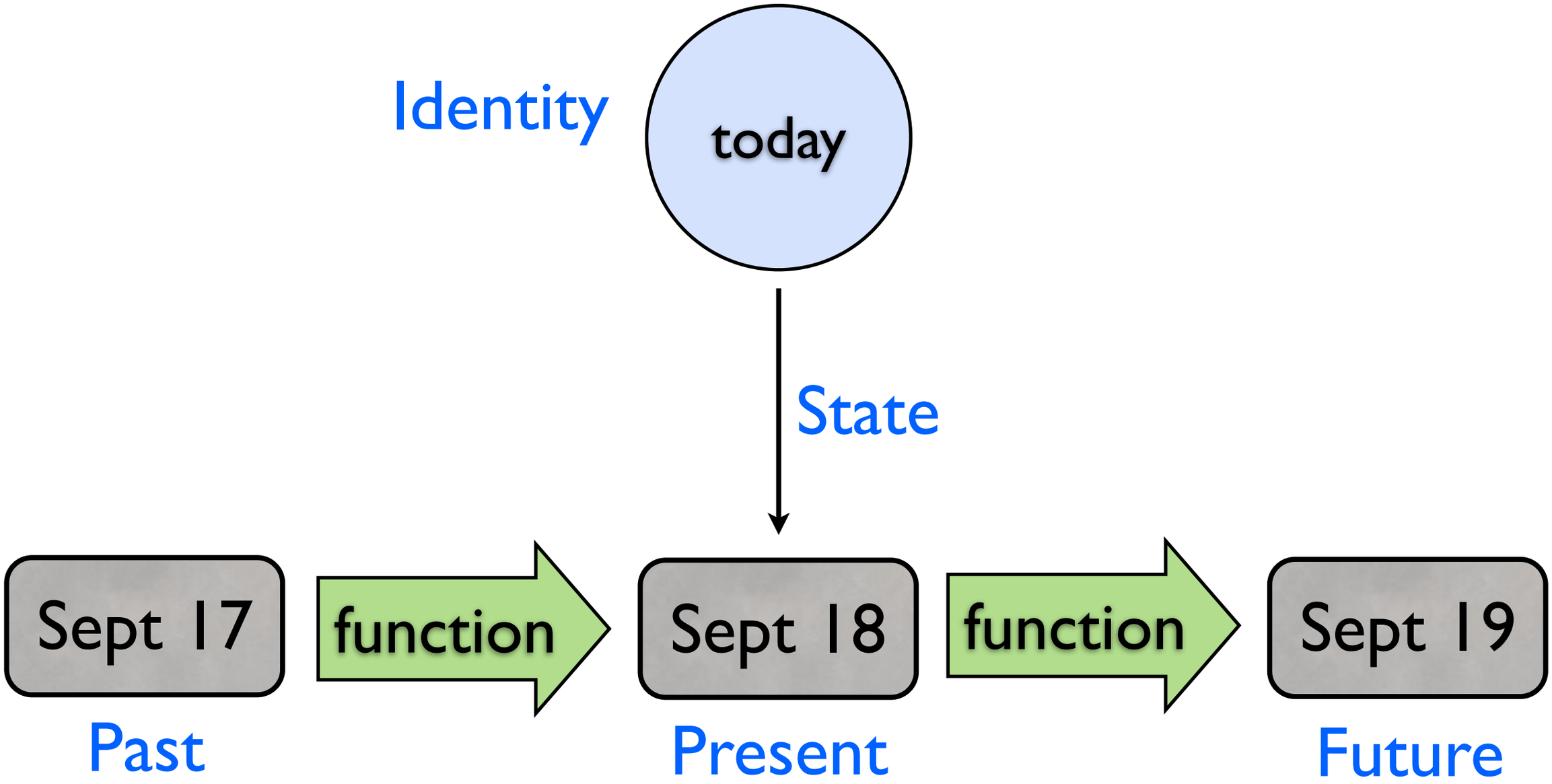

today
Sept 18


today
Sept 18
Sept 19


Identity
today
State
Sept 18
Sept 19
Value
Value

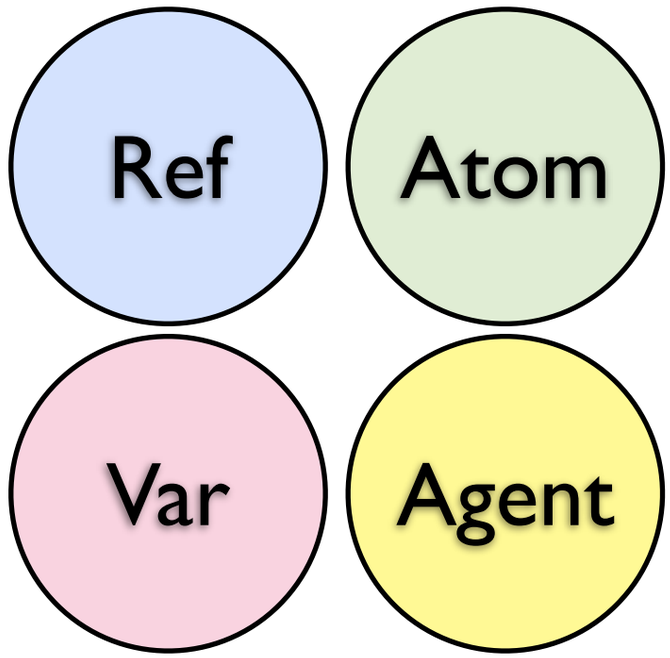

Identity
today
State
Sept 18
function
Sept 19
Value
Value

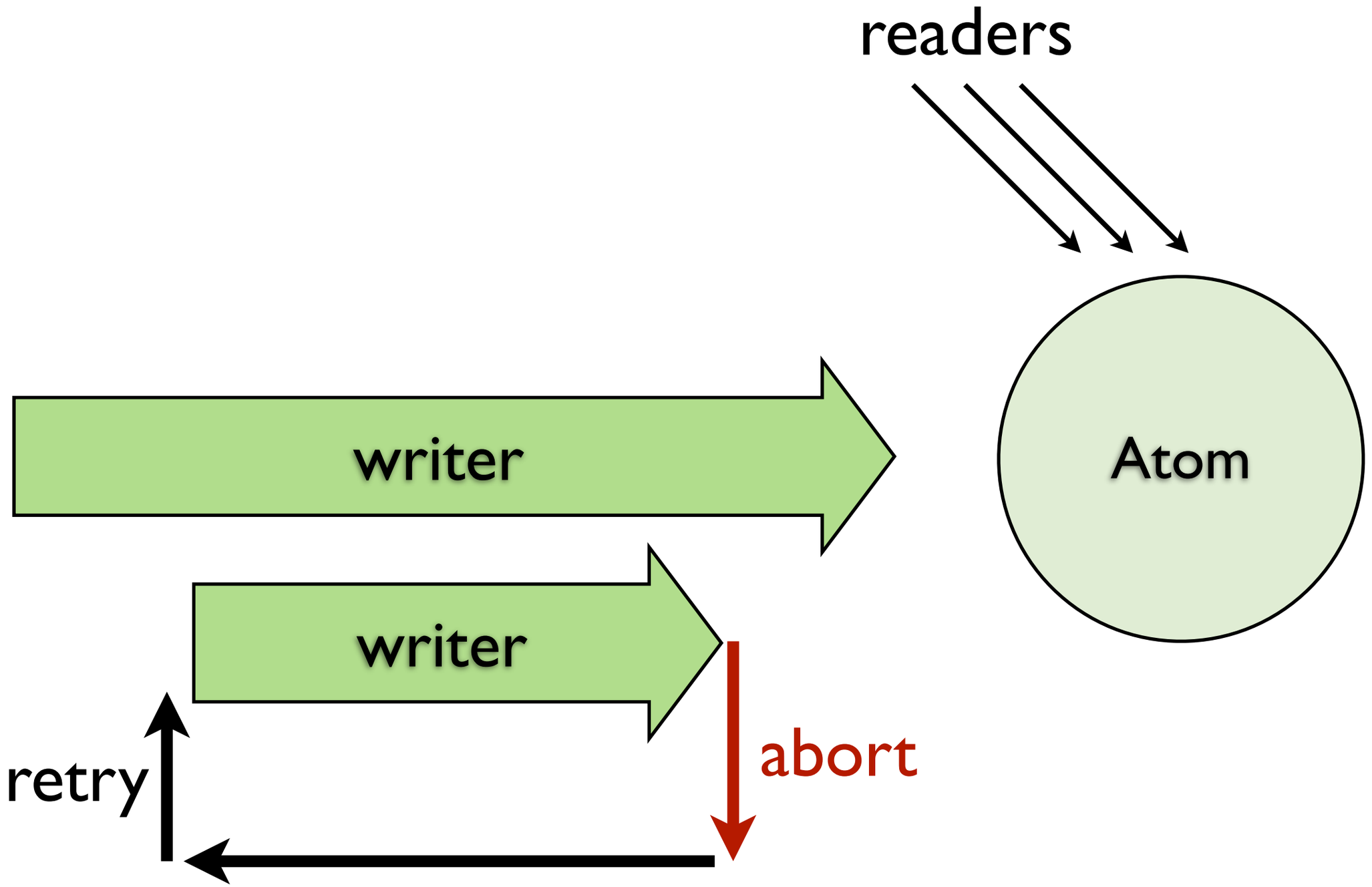

Identity
today
State
Sept 17
function
Sept 18
function
Sept 19
Past
Present
Future



The future is a function of the past.

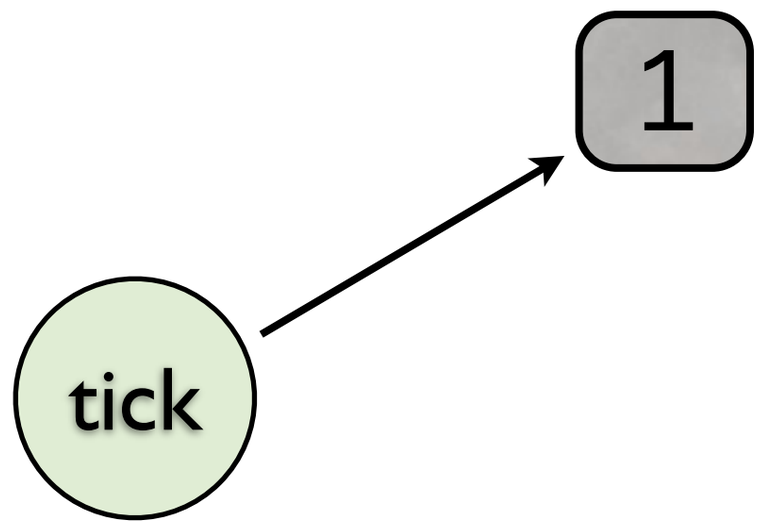
Mutable References




readers
writer
Atom
writer
retry
abort

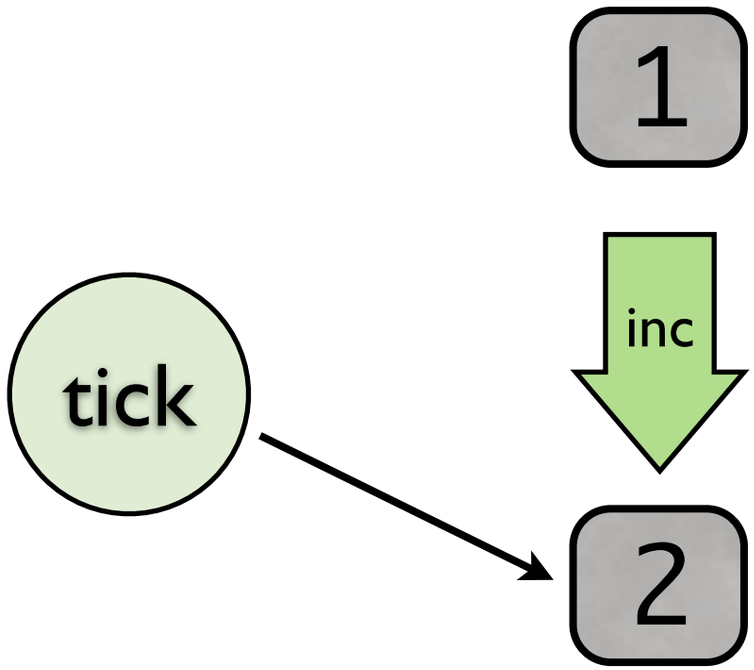
Atom

```
(def tick (atom 1))
(deref tick)    ⇨1
```



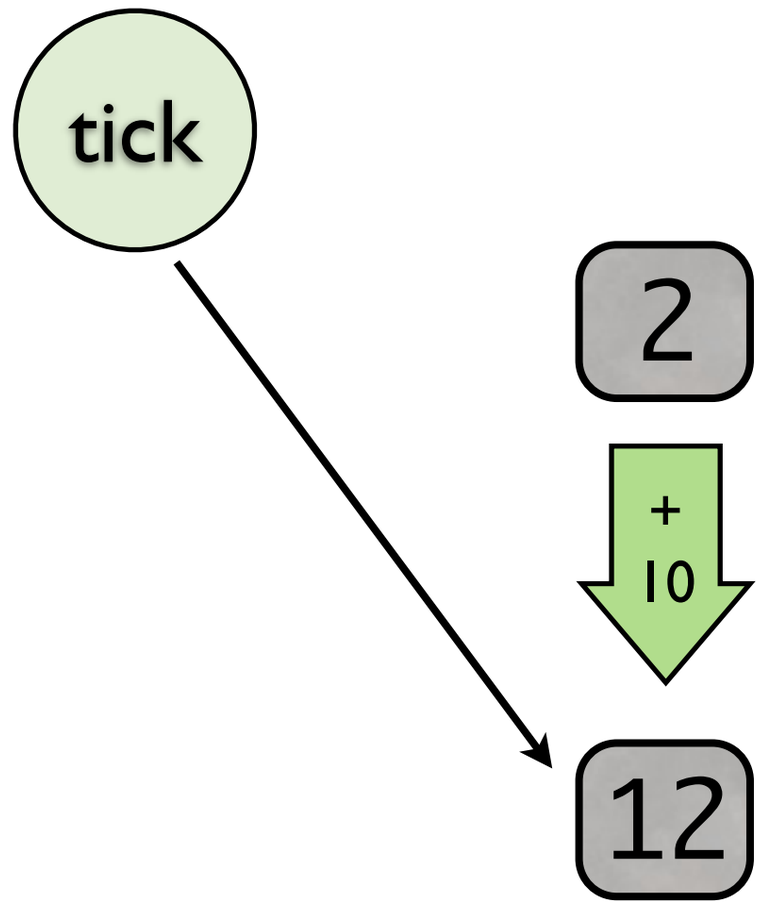
Atom

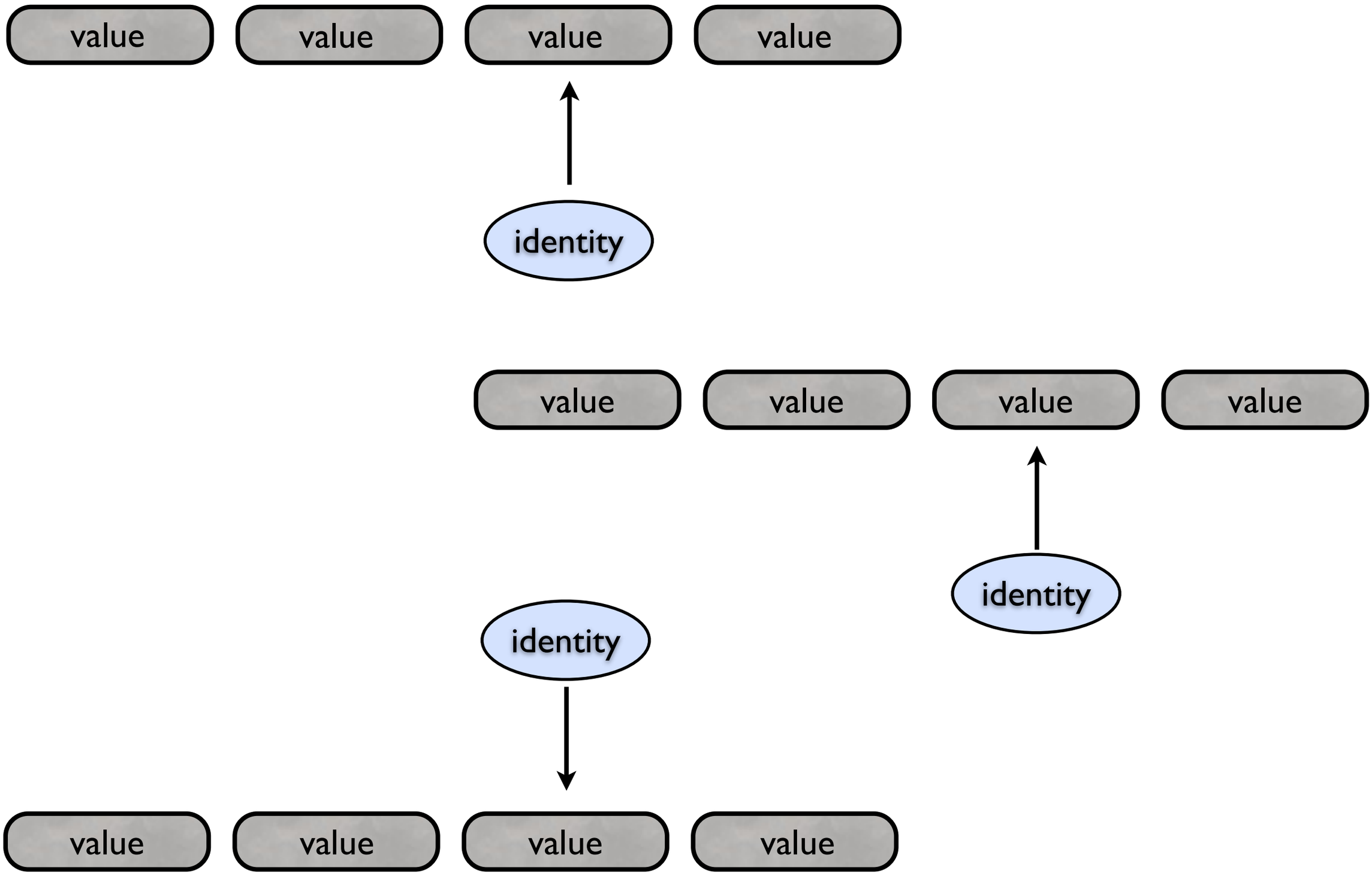
```
(def tick (atom 1))
(deref tick)    ⇨1

(swap! tick inc)
@tick           ⇨2
```

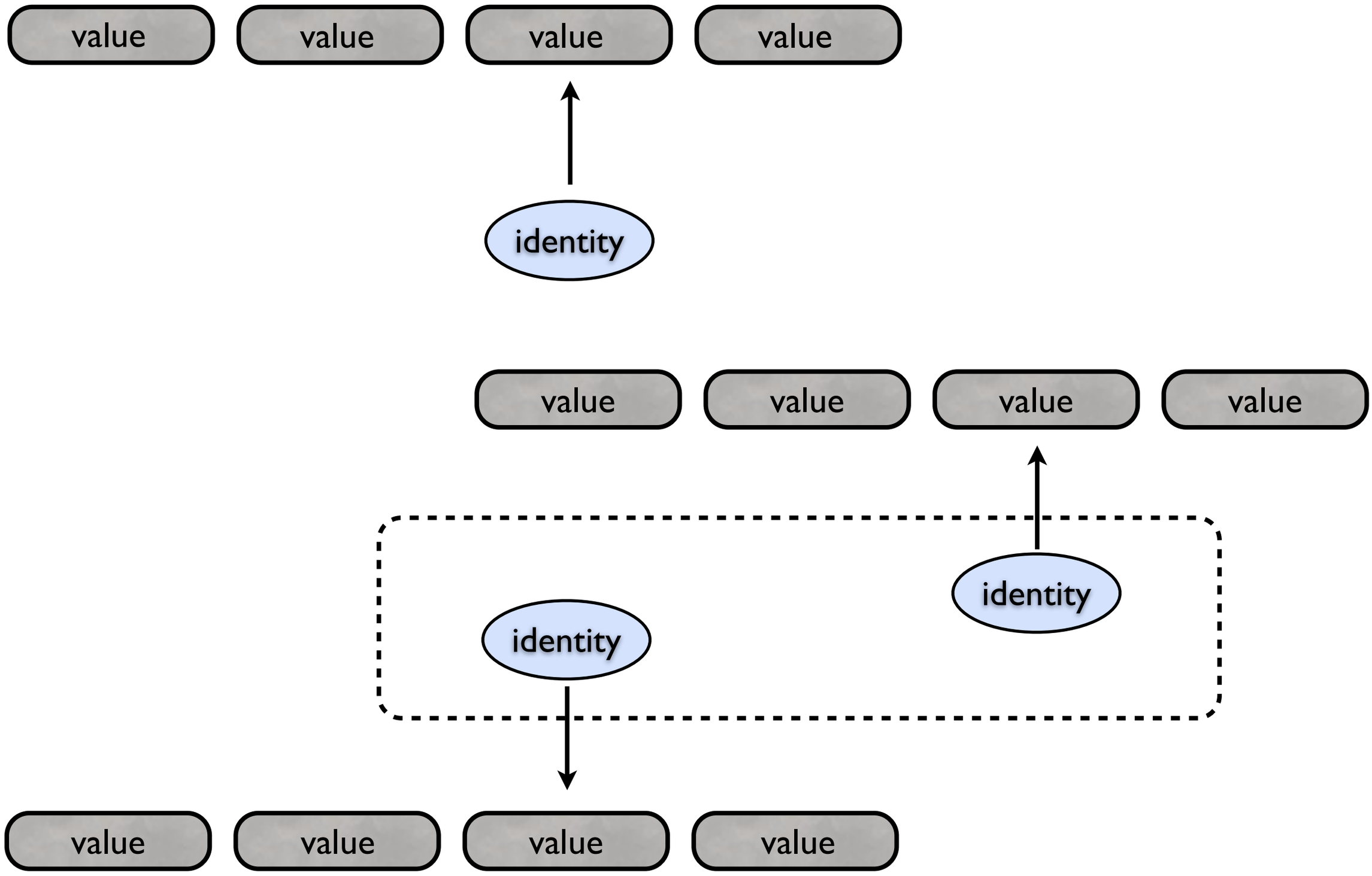


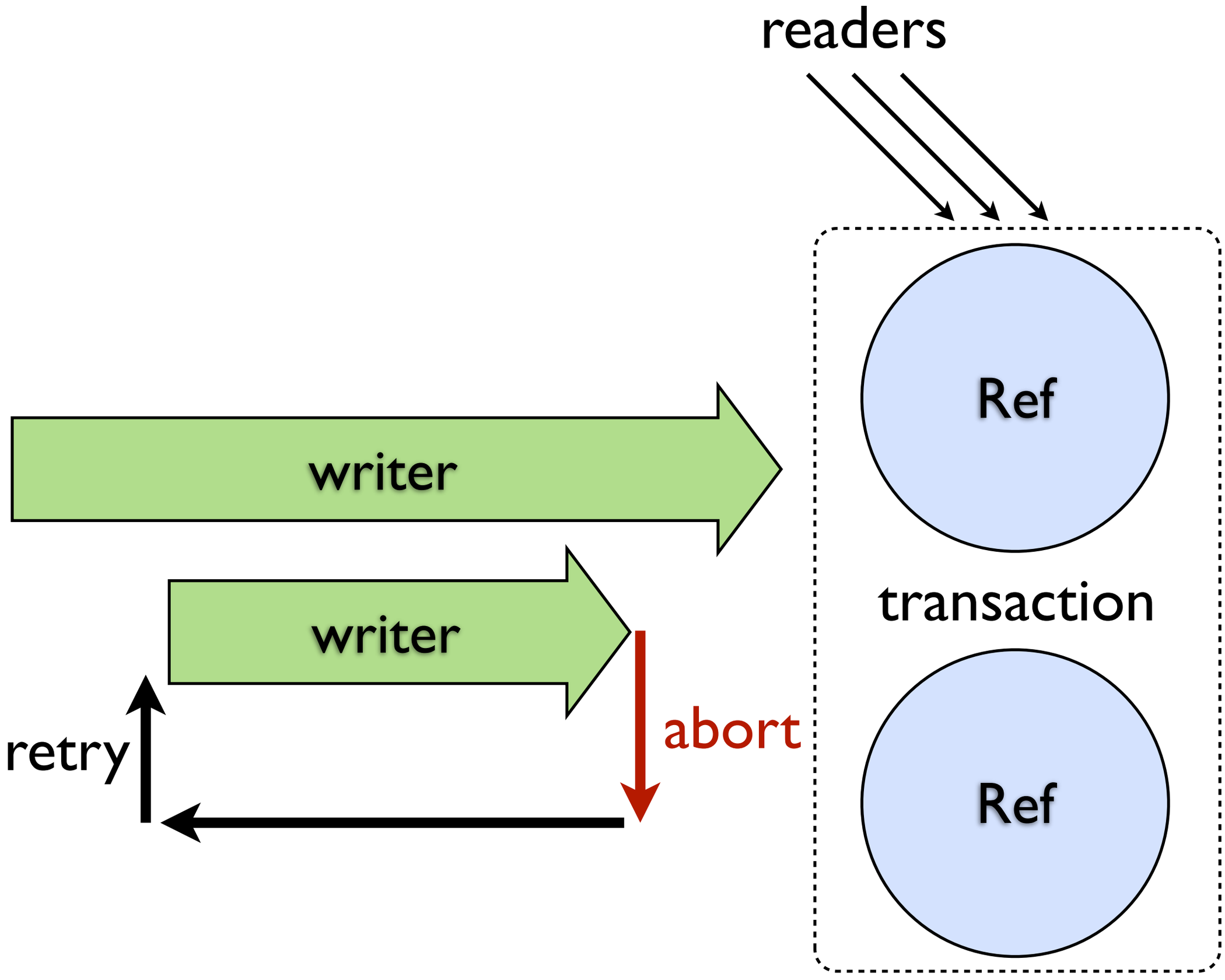
Atom

```
(def tick (atom 1))
(deref tick)          ⇒ 1

(swap! tick inc)
@tick                 ⇒ 2

(swap! tick + 10)
@tick                 ⇒ 12
```



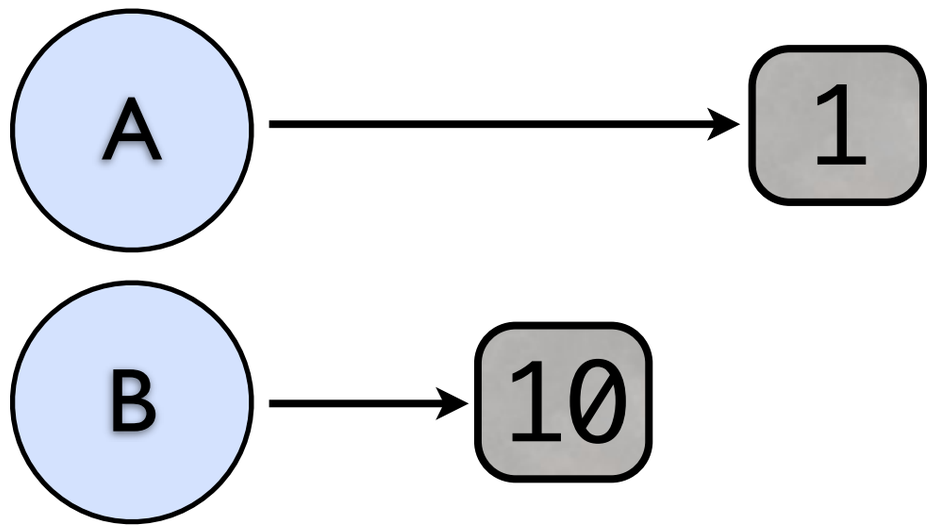

value
value
value
value
identity
value
value
value
value
identity
identity
value
value
value
value


value
value
value
value
identity
value
value
value
value
identity
identity
value
value
value
value


readers
Ref
transaction
Ref
writer
writer
retry
abort

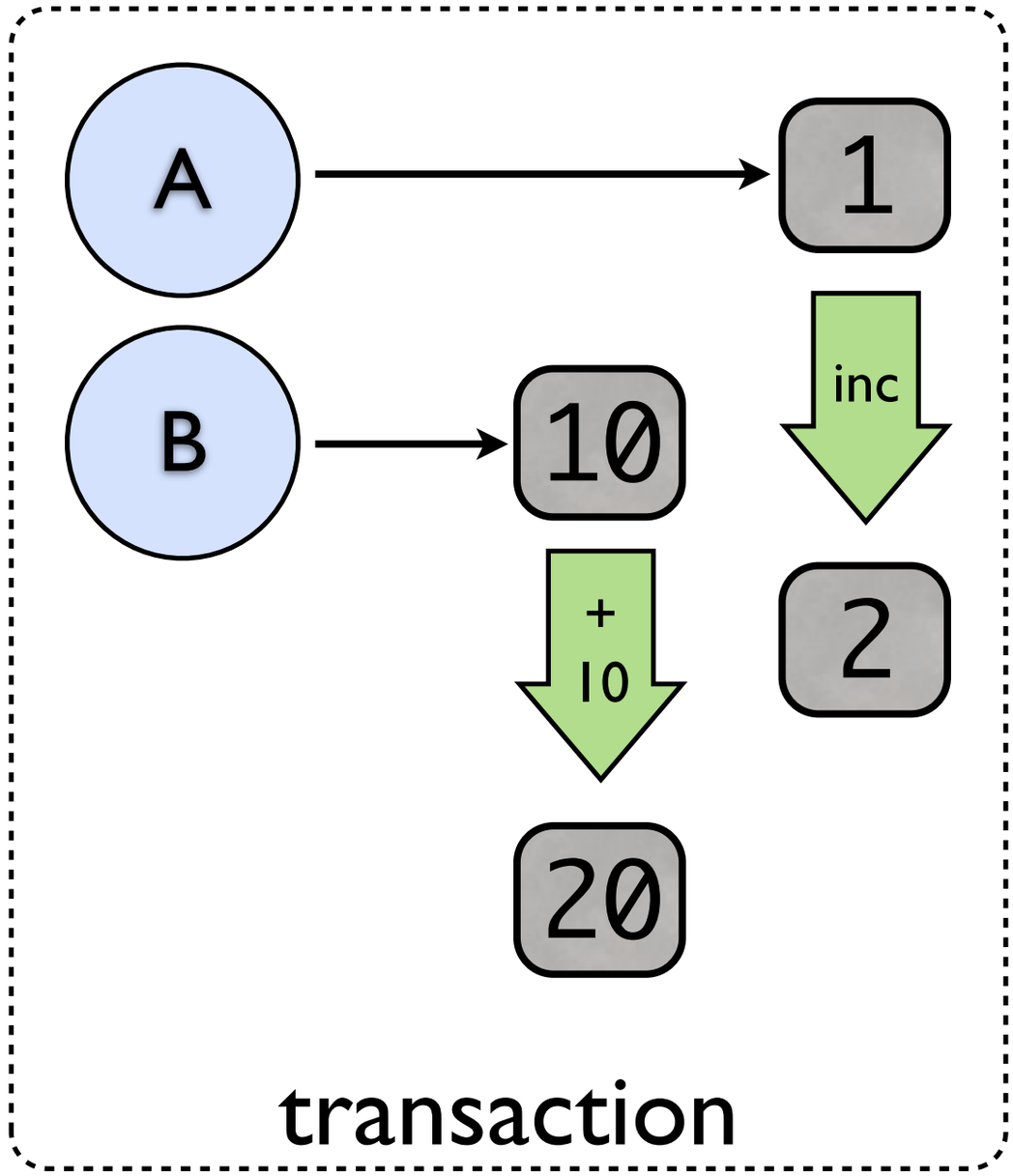
Ref

```
(def A (ref 1))
(def B (ref 10))
```



Ref

```
(def A (ref 1))
(def B (ref 10))


(dosync
  (alter A inc)
  (alter B + 10))
```

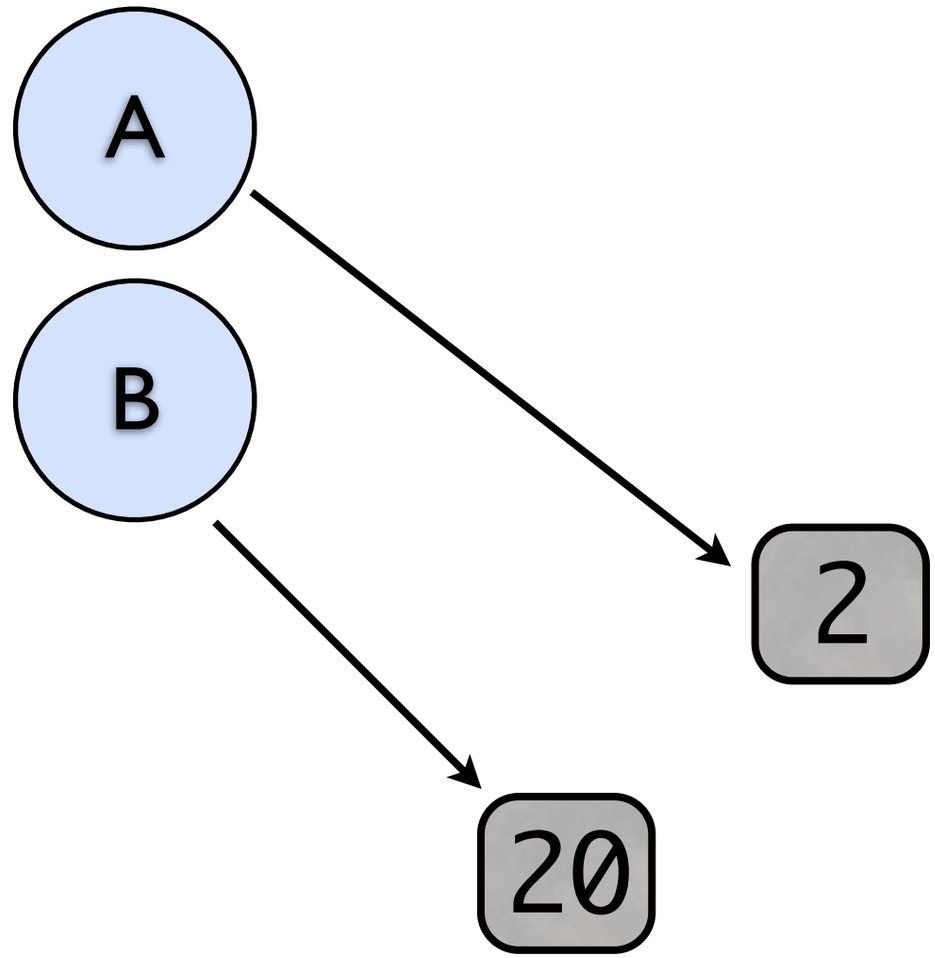


Ref

```
(def A (ref 1))
(def B (ref 10))
```

```
(dosync
  (alter A inc)
  (alter B + 10))
```

```
@A ⇨2
@B ⇨20
```



Database Transactions

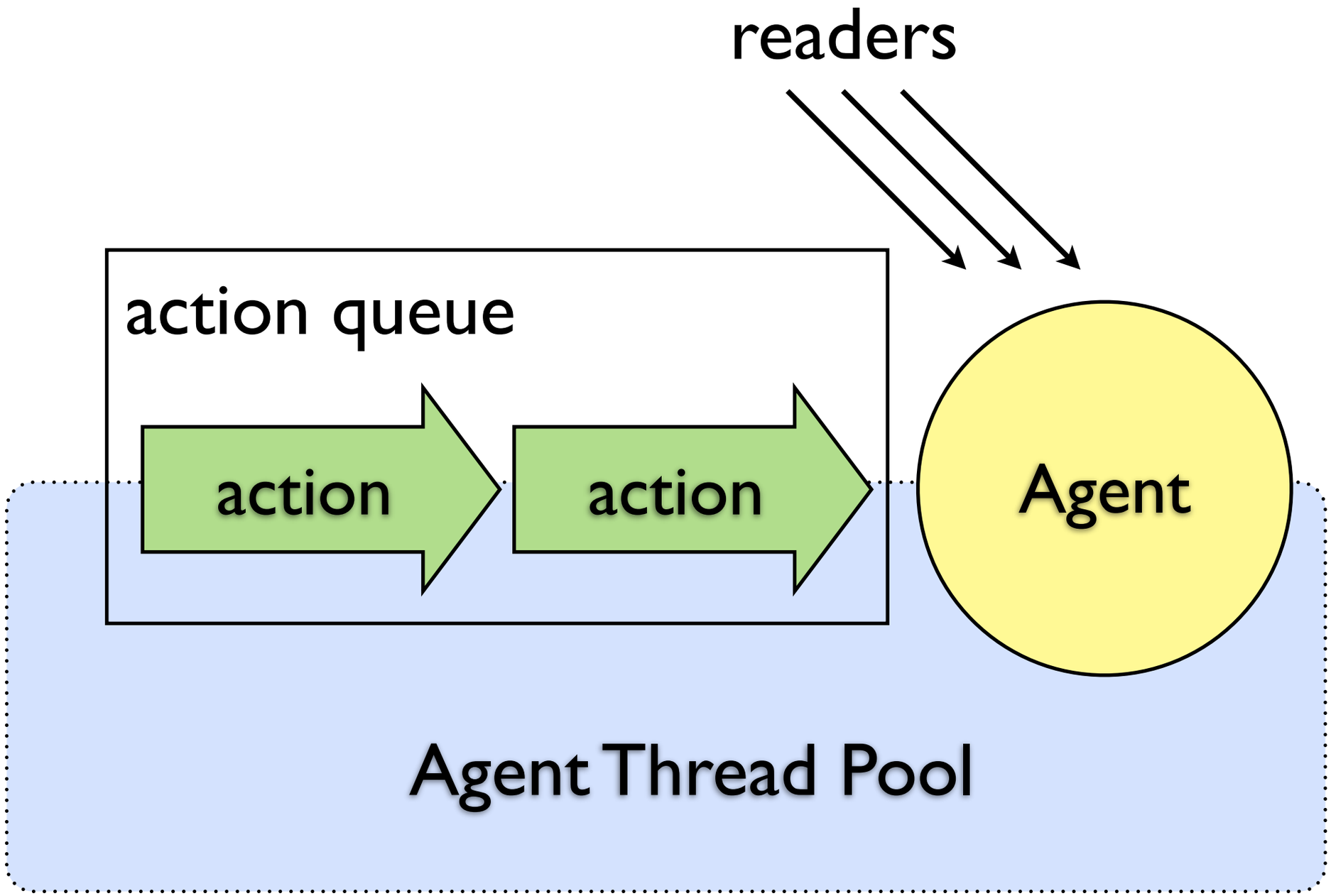
Atomic
Consistent
Isolated
Durable

Clojure Transactions

Atomic

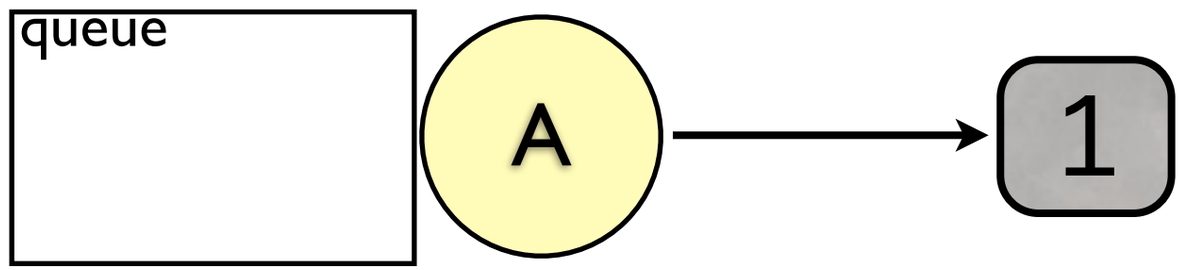
Consistent

Isolated


readers
action queue
action
action
Agent
Agent Thread Pool

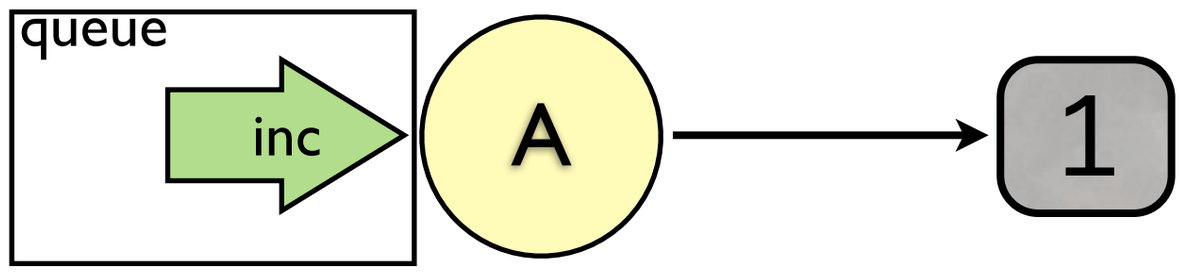
Agent

```
(def A (agent 1))
```



Agent

```
(def A (agent 1))

(send A inc)
```



@A ⇨ 1

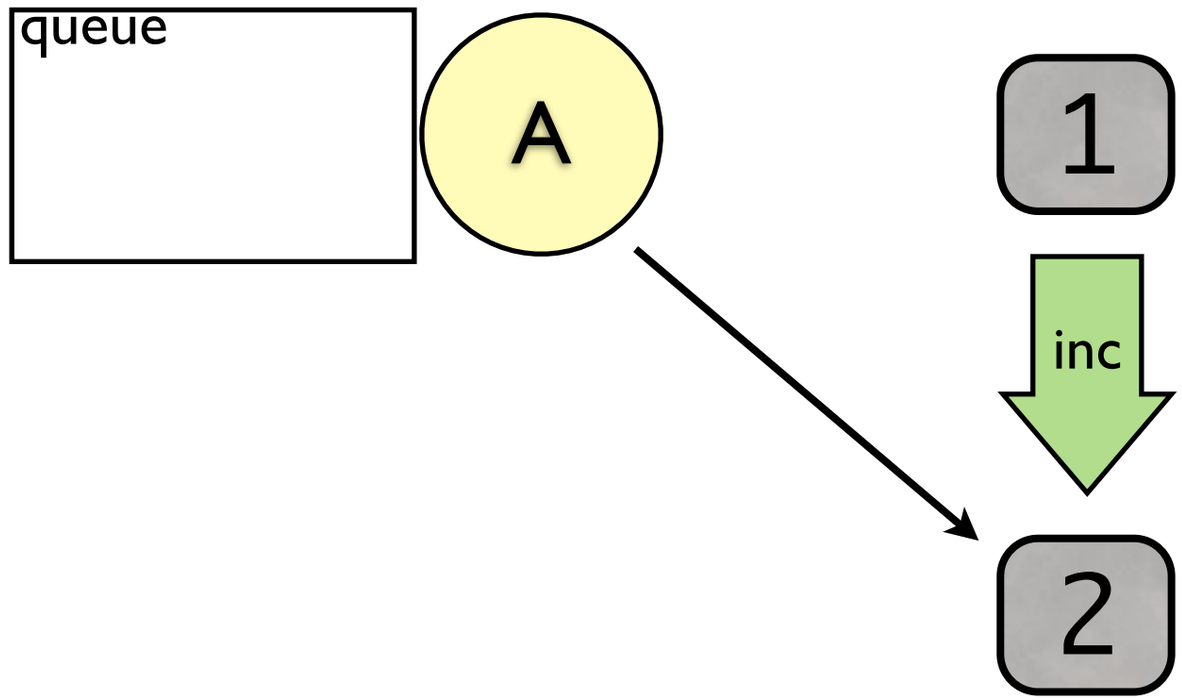
Agent

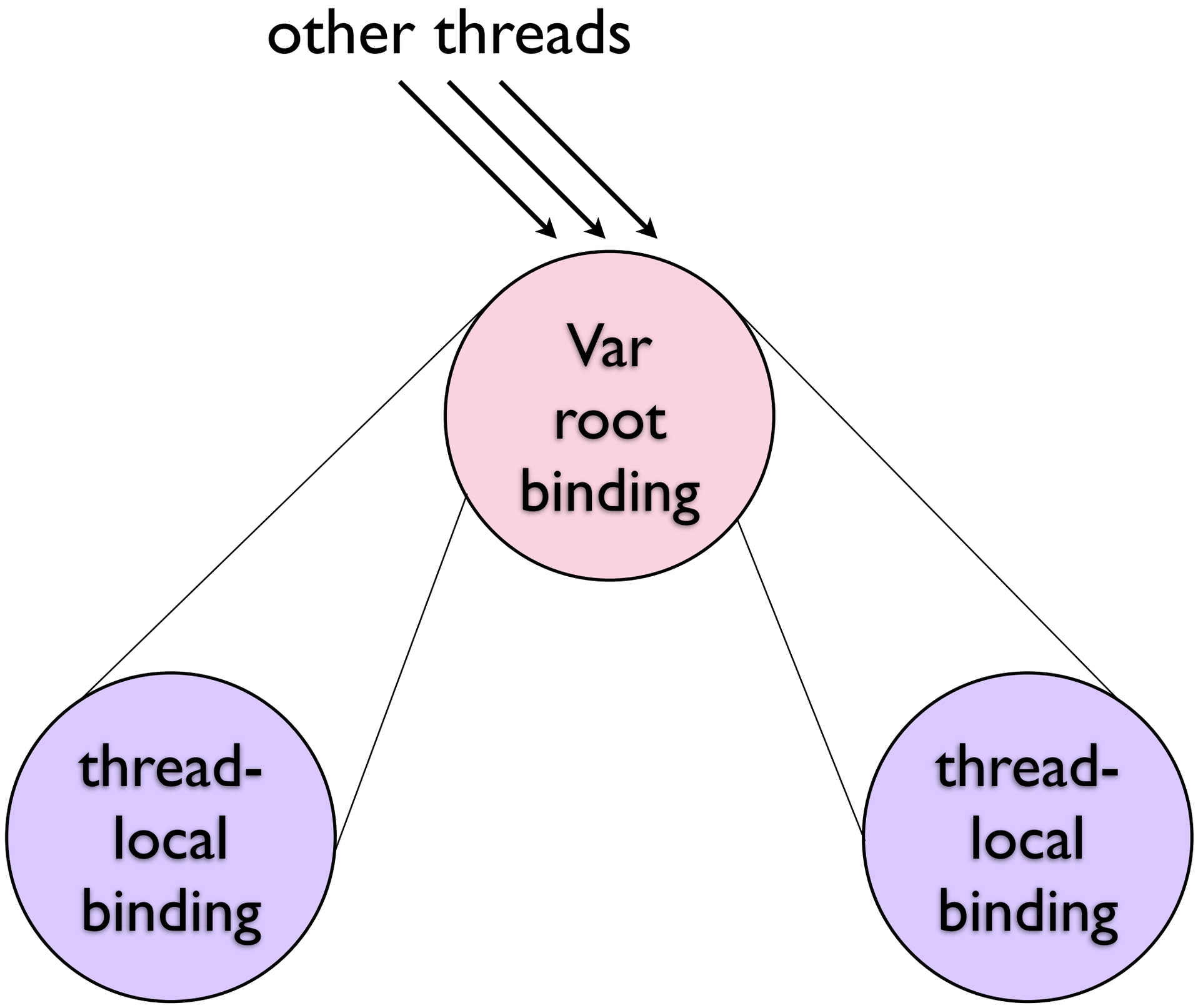
```
(def A (agent 1))
```

```
(send A inc)
```

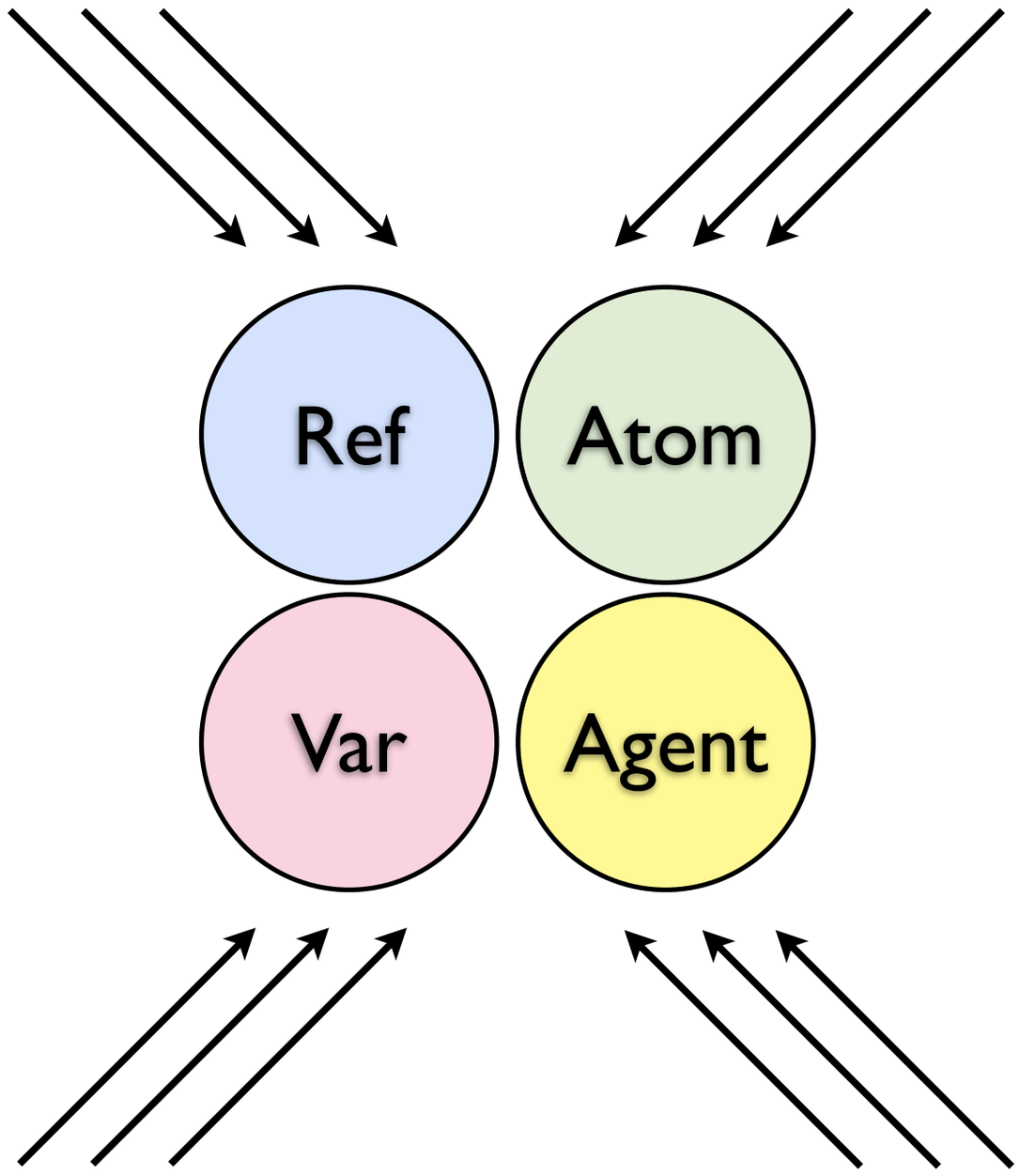
@A ⇨ 1

@A ⇨ 2

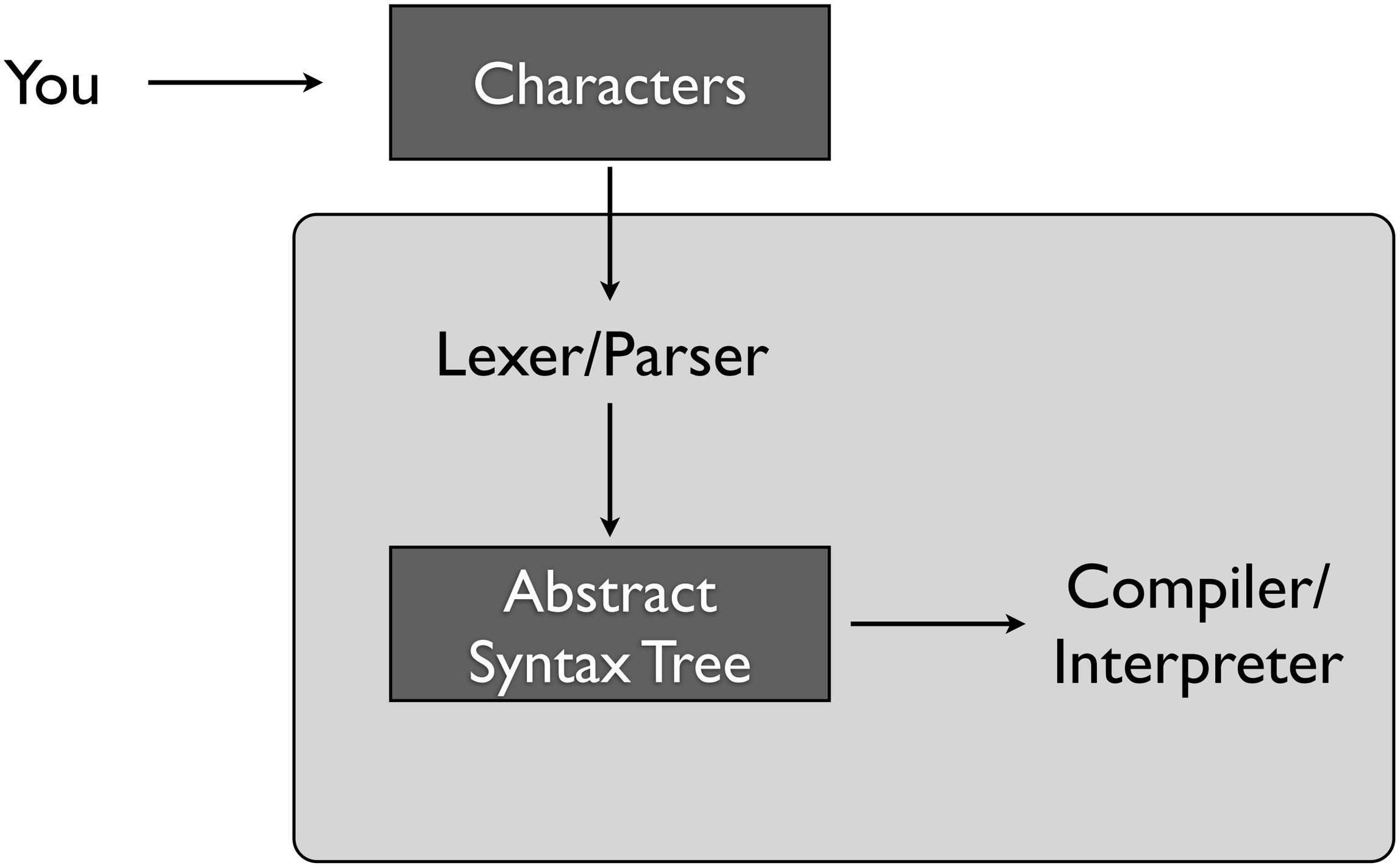


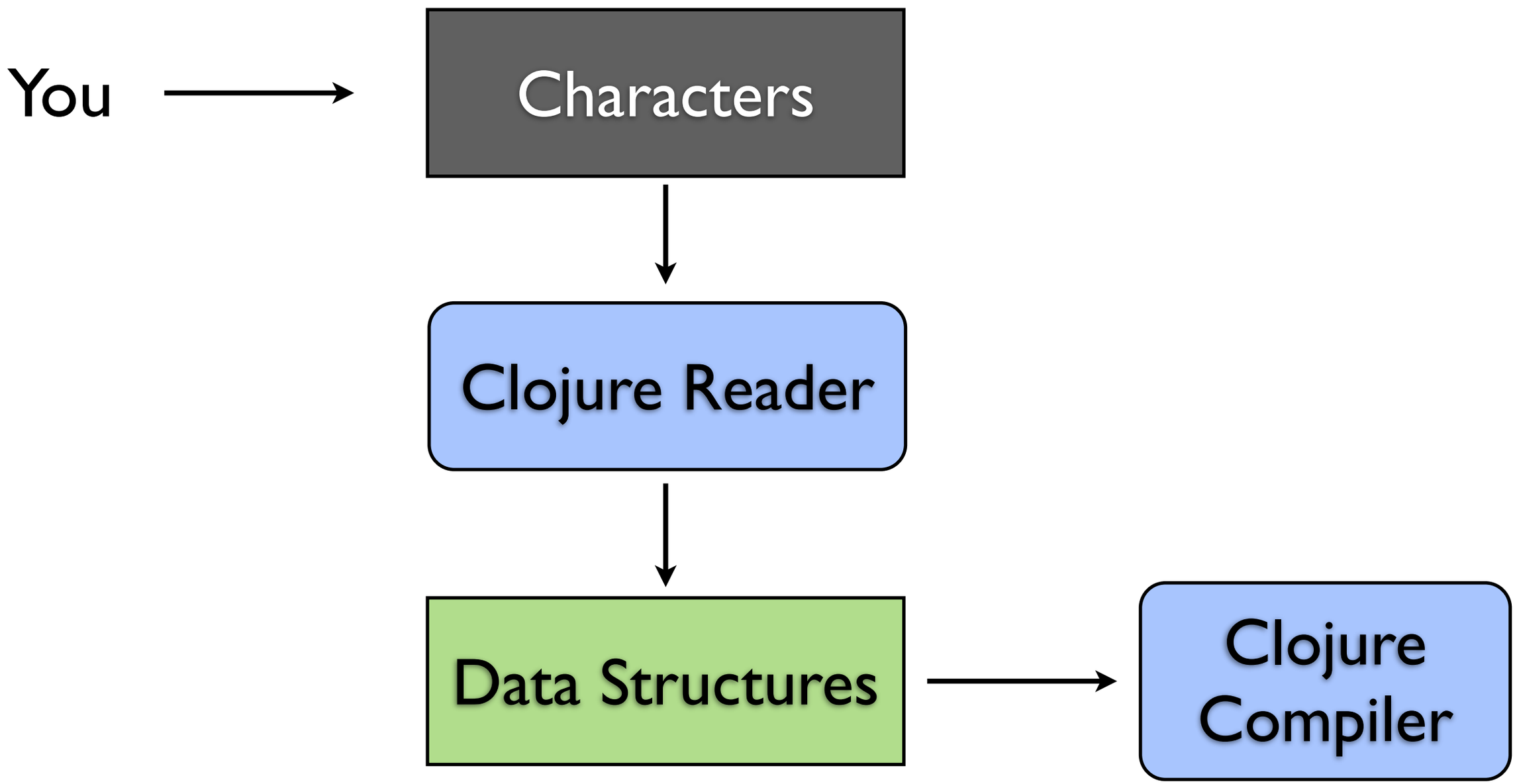

other threads
Var
root
binding
thread-
local
binding
thread-
local
binding

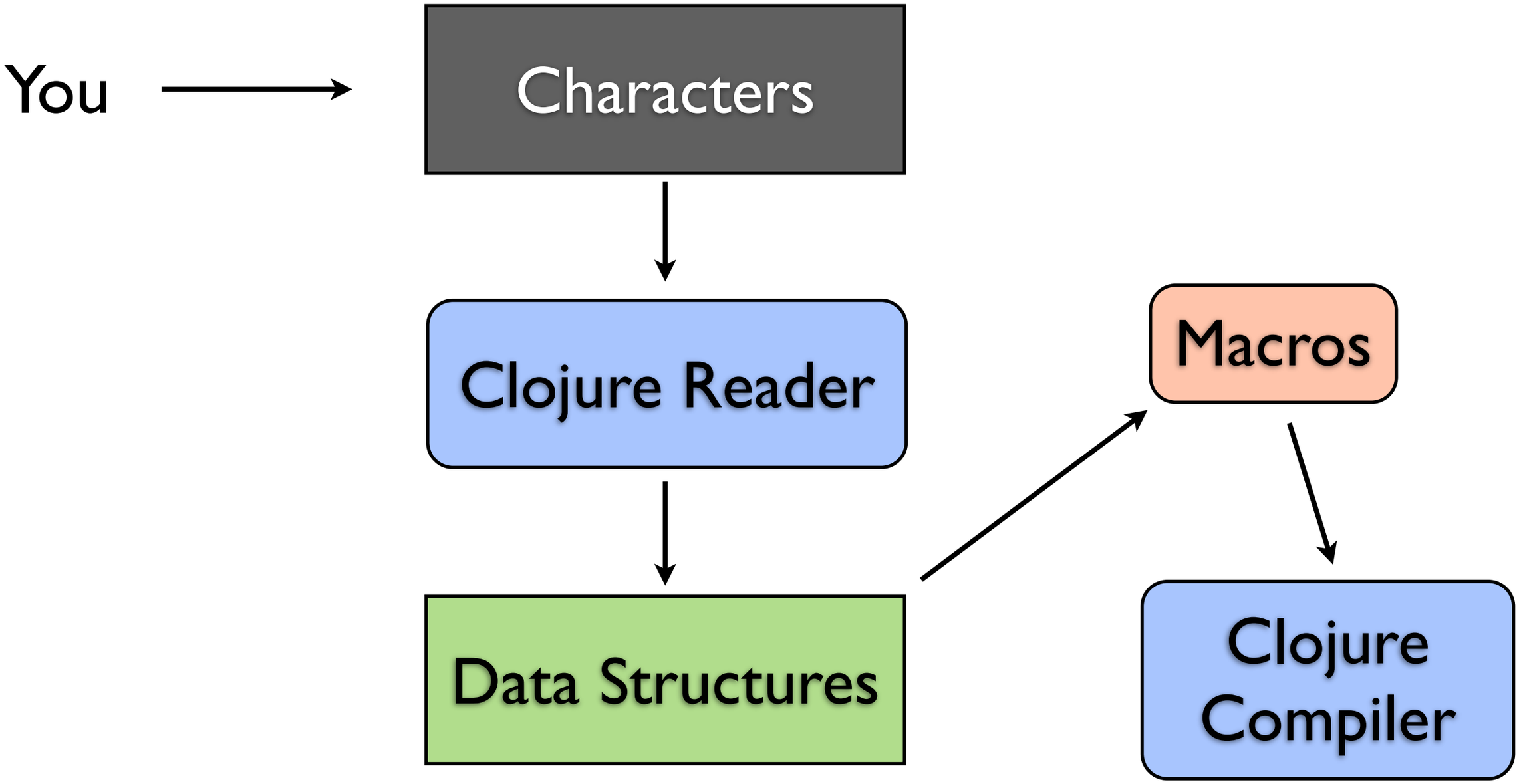
Concurrency



Macros


You
Characters
Lexer/Parser
Abstract
Syntax Tree
Compiler/
Interpreter


You
Characters
Clojure Reader
Data Structures
Clojure
Compiler

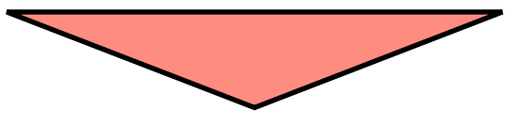

You
Characters
Clojure Reader
Macros
Data Structures
Clojure
Compiler

read-string and eval

```
user=> (read-string "(println 1 2 3)")
(println 1 2 3)

user=> (eval *1)
1 2 3
nil
```

Macro: when

```
(when condition
  exprs*)
```



```
(if condition
  (do exprs*))
```

defmacro

name

parameters

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

construct code to return

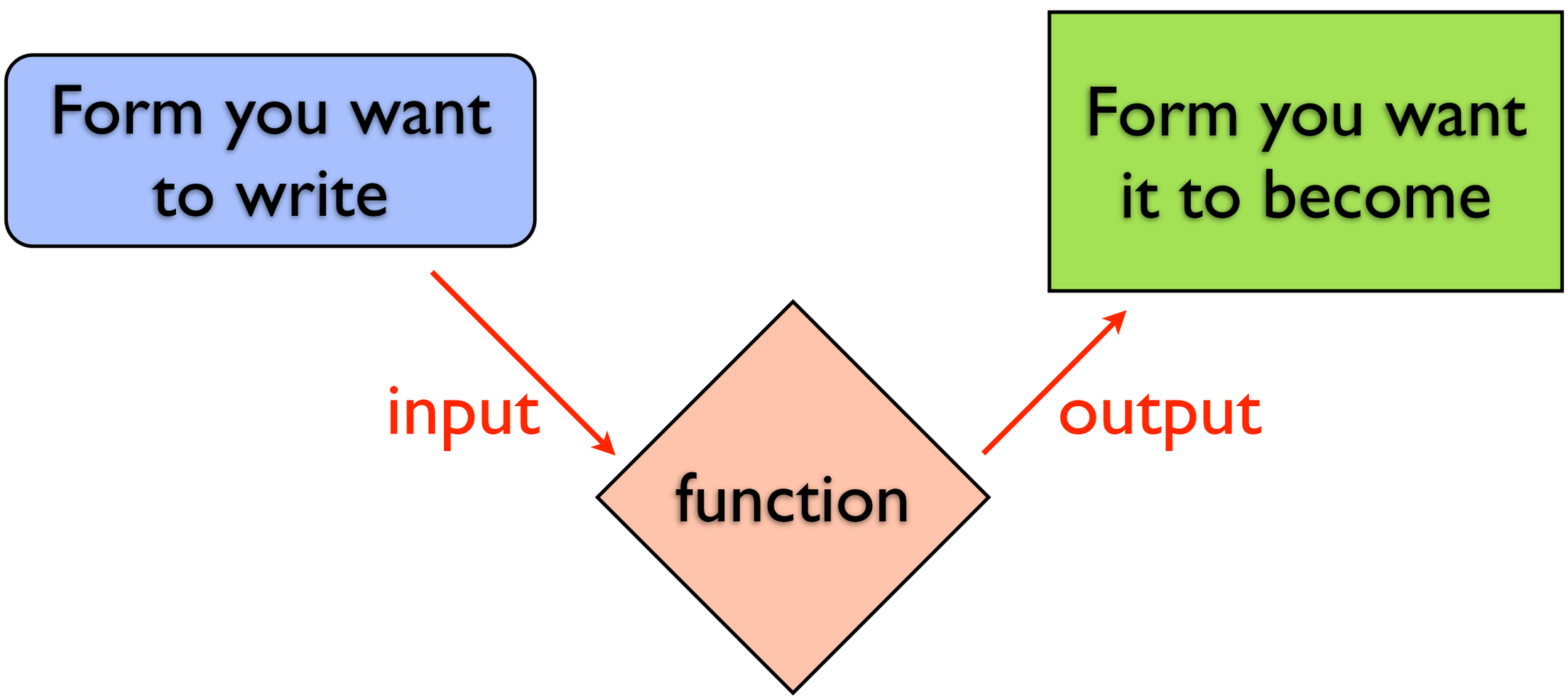
macroexpand

quoted expression

```
user=> (macroexpand '(when a b c d))
(if a (do b c d))
```

expanded result

How to Write a Macro



How to Write a Macro

```
;; we want to write:
(when foo
  (dothis)
  (dothat))
```

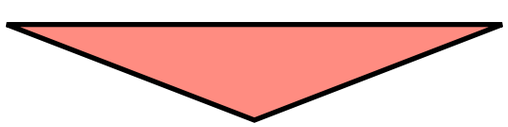
```
;; and have it become:
(if foo
  (do
    (dothis)
    (dothat)))
```

```
;; from core.clj
(defmacro when
  [test & body]
  (list 'if test (cons 'do body)))
```

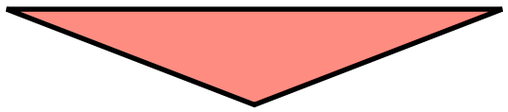
```
;; use macroexpand to try it
user=> (macroexpand '(when foo (dothis) (dothat)))
(if foo (do (dothis) (dothat)))
```

Macro Expansion

```
(when a b c d)
```



Macro: when

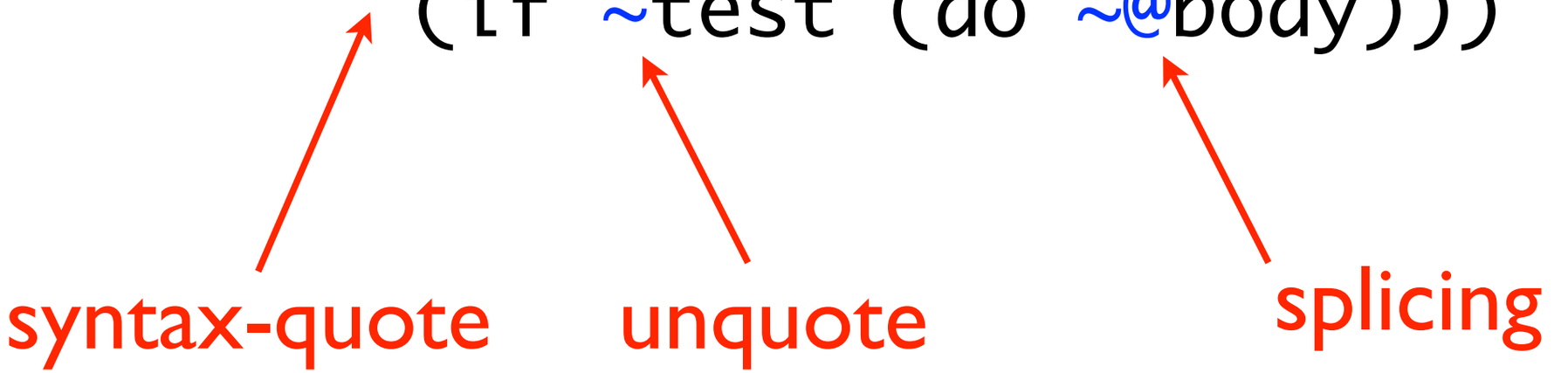


```
(if a (do b c d))
```

arguments are not evaluated

syntax-quote

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



syntax-quote

```
user=> (def x 5)
#'user/x
user=> (def lst '(:a :b :c))
#'user/lst
user=> `(x ~x ~lst)
(user/x 5 (:a :b :c))
user=> `(x ~x ~@lst)
(user/x 5 :a :b :c)
```

gensym

```
user=> (gensym "foo")
foo39
user=> (gensym "foo")
foo42
```

auto-gensym

qualified symbol
not allowed here

```
user=> `(let [x 1] x)
(clojure.core/let [user/x 1]  user/x)
```

```
user=> `(let [x# 1] x#)
(clojure.core/let [x__29__auto__ 1]  x__29__auto__)
```

generated
symbol

Recursion

Recursive GCD

```
user=> (defn gcd [x y]      ; x >= y
         (if (zero? y) x
           (gcd y (rem x y))))
#'user/gcd
user=> (gcd 100 35)
5
user=> (gcd 37 22)
1
```

Recursive GCD

```
user=> (defn gcd [x y]          ; x >= y
         (if (zero? y) x
           (gcd y (rem x y))))
```

GCD with recur

```
user=> (defn gcd [x y]          ; x >= y
         (if (zero? y) x
           (recur y (rem x y))))
```

tail position

Recursive Fibonacci

```
user=> (defn fib [n]
         (cond (zero? n) 0
               (= 1 n) 1
               :else (+ (fib (dec n)) (fib (- n 2)))))
#'user/fib
user=> (fib 10)
55
user=> (fib 50)  ; wait a really long time
```

Fibaonacci Helper

```
user=> (defn fib-step [a b n]
         (if (= 1 n) b
           (recur b (+ a b) (dec n))))
#'user/fib-step
user=> (defn fib [n]
         (fib-step 0 1 n))
#'user/fib
user=> (fib 50)
12586269025
```

loop / recur

```
user=> (defn fib [n]
         (loop [a 0, b 1, i n]
           (if (= 1 i) b
             (recur b (+ a b) (dec i)))))
#'user/fib
user=> (fib 50)
12586269025
```

loop / recur

```
(loop [initialization]
  (if termination-condition
      return-value
      (recur updated-variables)))
```

More

Clojure

- clojure.org
- clojure.blip.tv
- dev.clojure.org
- groups.google.com/group/clojure
- #clojure on irc.freenode.net

Fun

- github.com/relevance/labrepl
- 4clojure.com
- github.com/functional-koans/clojure-koans