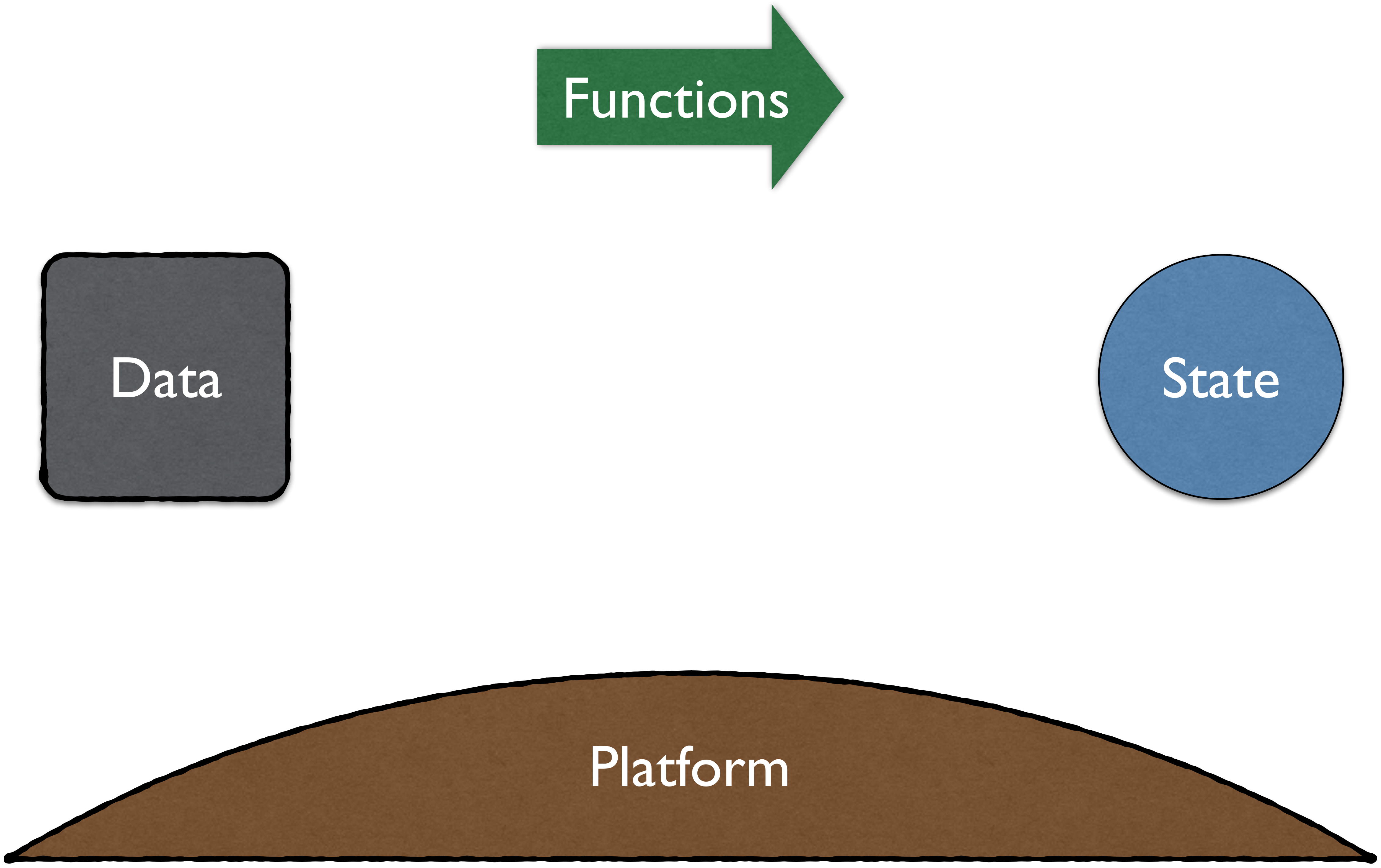
Intro to Clojure

@stuartsierra
DevNexus 2014, Atlanta

cognitect

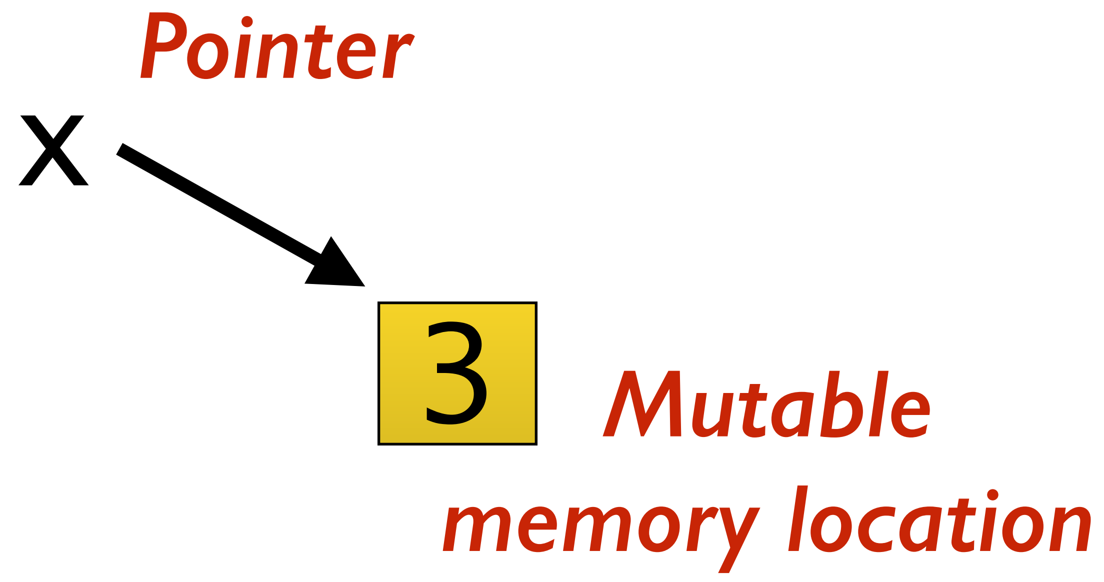

Functions
Data
State
Platform

Data

Values

x

```
x = 3

x = 107

x = x + 1
```


Pointer
x
3
Mutable
memory location

3

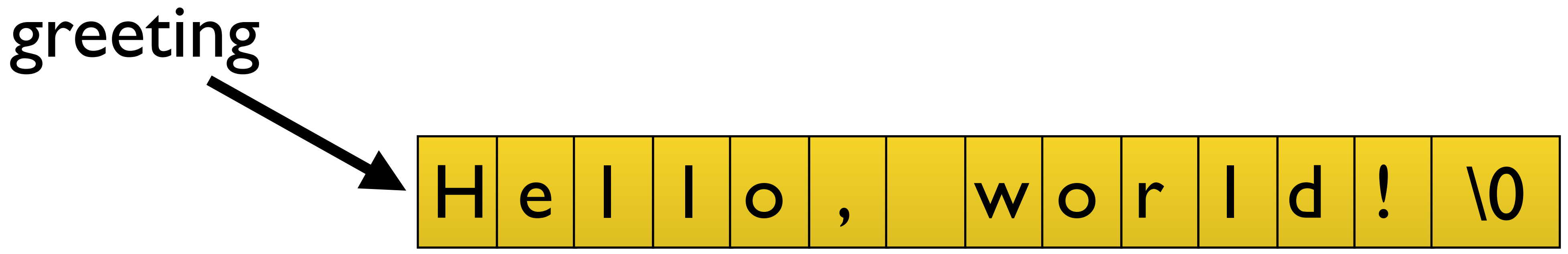
$$3 = 3 + 1$$

3

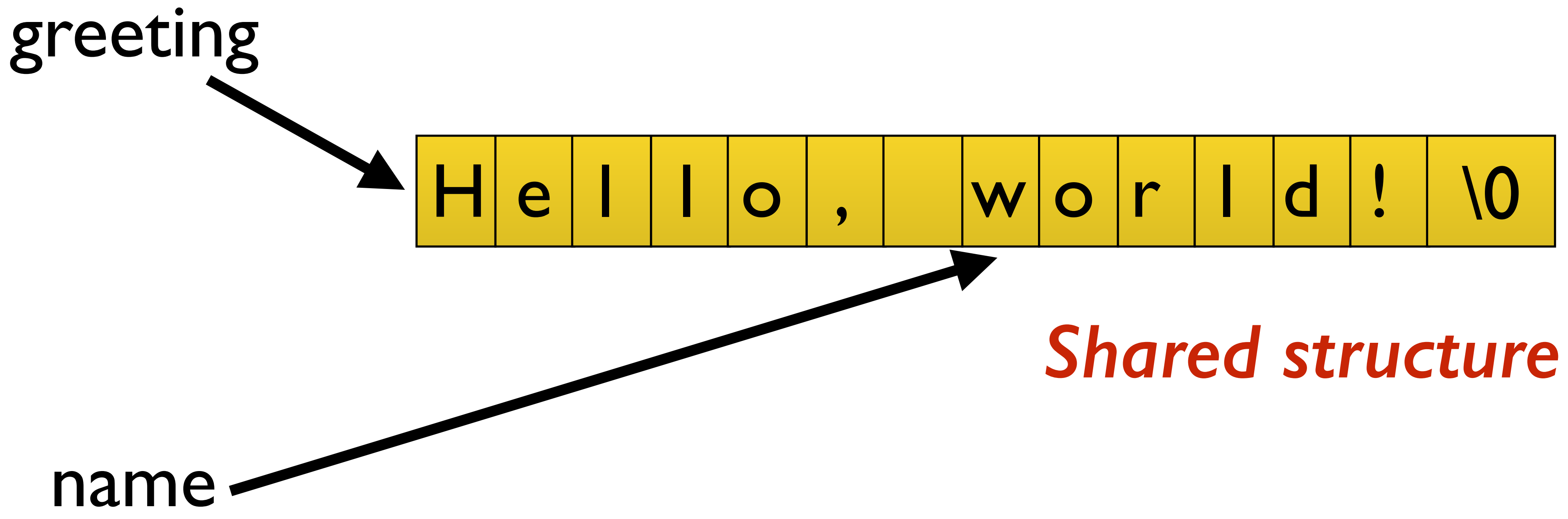
Immutable value

" H e l l o ,   w o r l d ! "

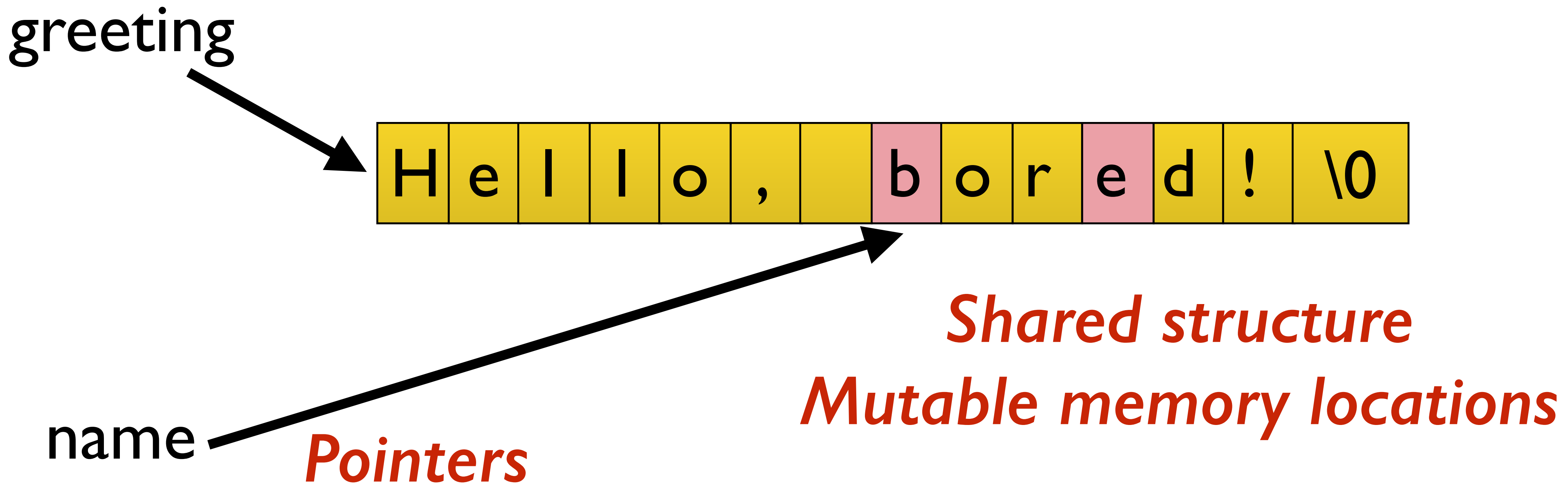
```
char greeting[14];
strcpy(greeting, "Hello, world!");
```



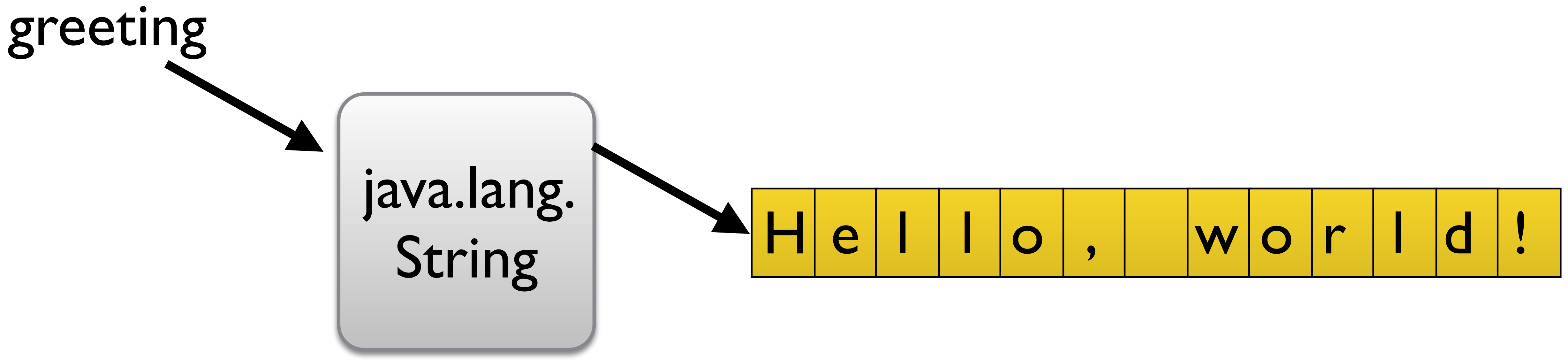
```
char *name = greeting + 7;
```



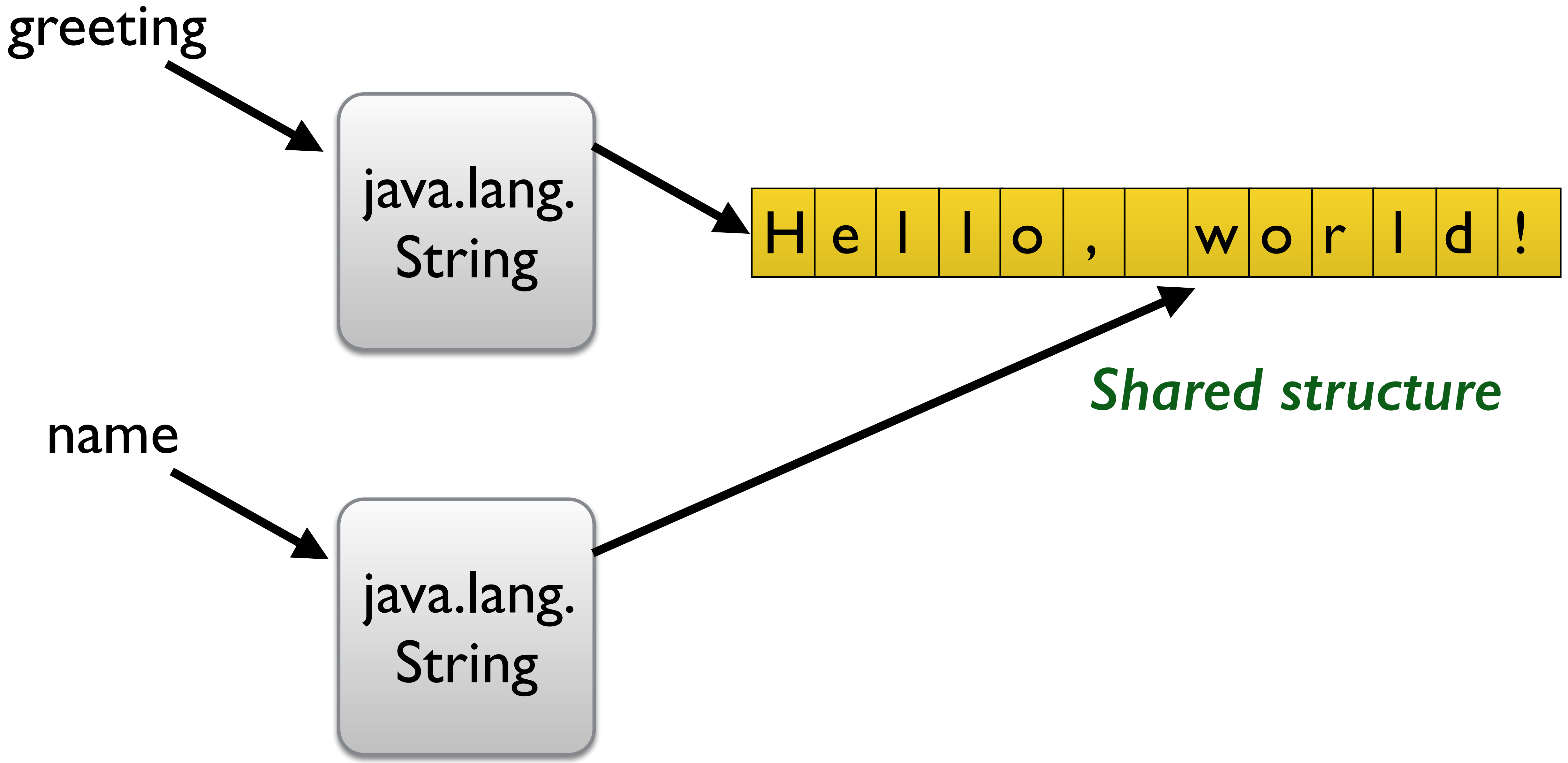
```
name[0] = 'b';
name[3] = 'e';
```



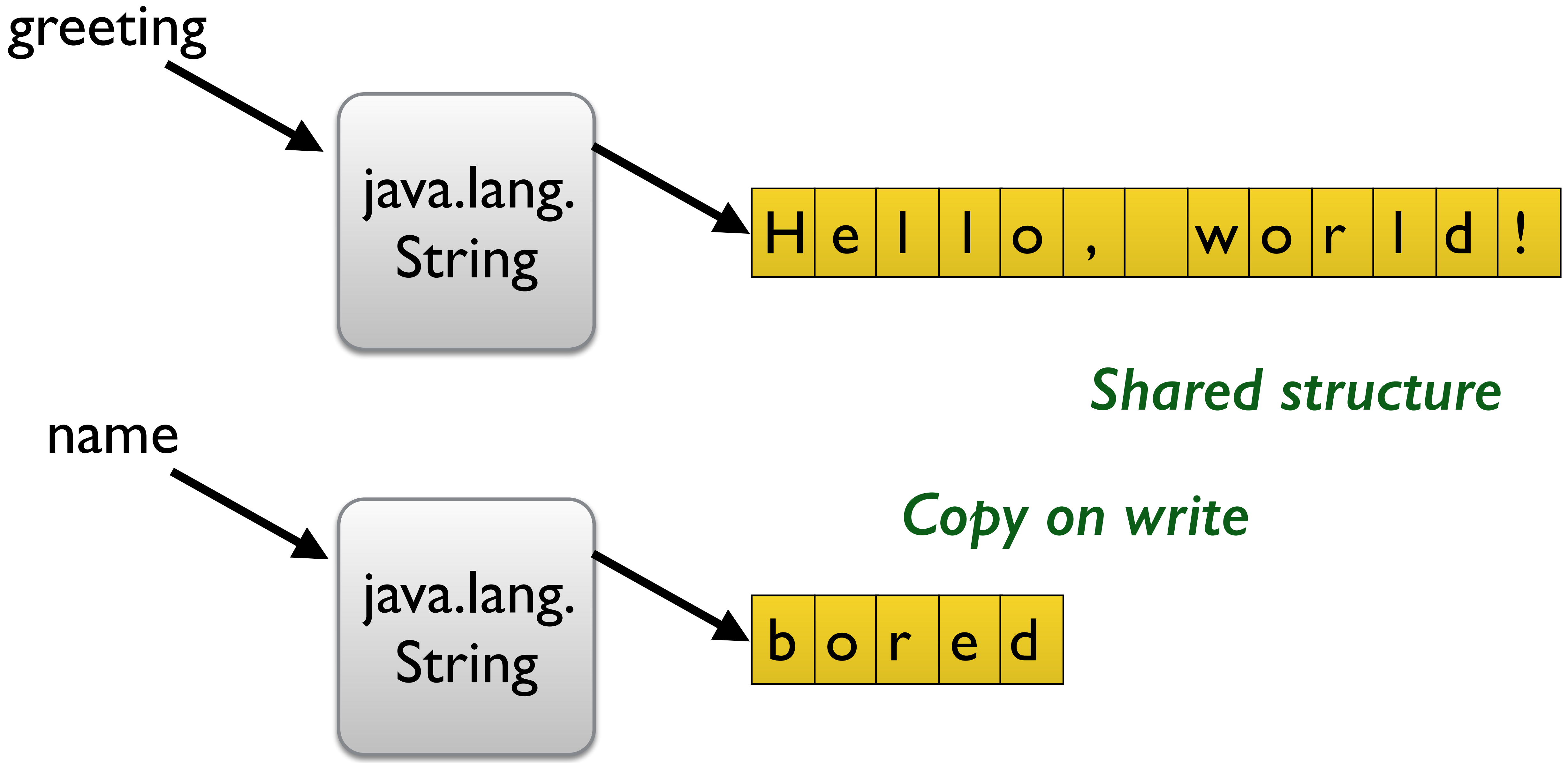
```
String greeting =
    new String("Hello, world!");
```



```
String name =
   greeting.substring(7,12);
```



```
name = name.replace('w', 'b')
           .replace('l', 'e')
```



Shared structure

Copy on write

java.lang.
String

Immutable value

```java
class Invoice {
  private Date date;

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

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

Mutable!

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

```
class Invoice {
  private Date date;

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

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

```
class Invoice {
  private Date date;

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

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

```
1, 2, 3, 4, 5
```

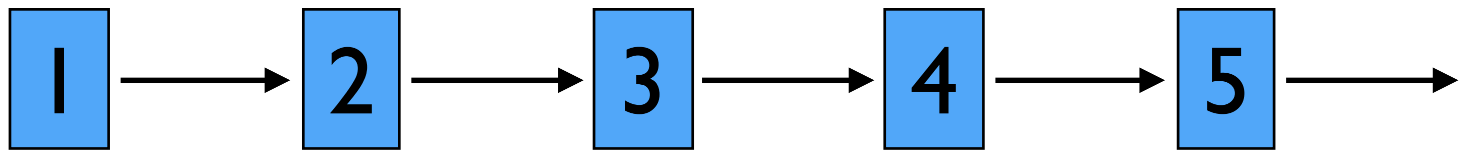
```
( 1, 2, 3, 4, 5 )
```

List

```
( 1  2  3  4  5 )
```

List

```
( 1  2  3  4  5 )
```



List

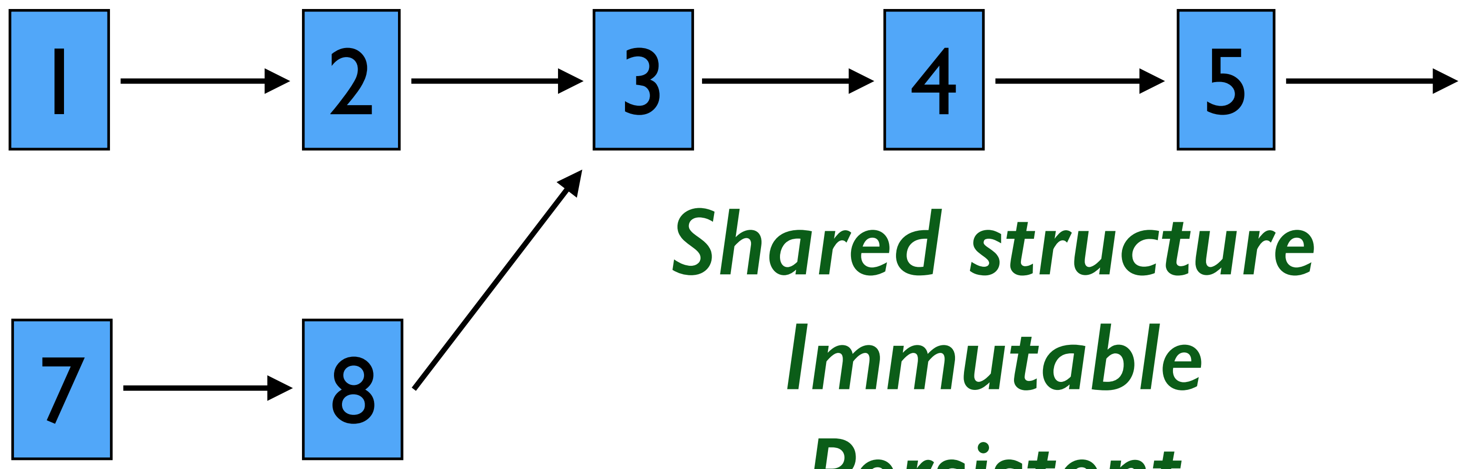
(1 2 3 4 5)

$$\begin{tikzcd}
\boxed{1} \arrow[r] & \boxed{2} \arrow[r] & \boxed{3} \arrow[r] & \boxed{4} \arrow[r] & \boxed{5} \arrow[r] & {} \\
\boxed{7} \arrow[r] & \boxed{8} \arrow[ur] & & & &
\end{tikzcd}$$

(7 8 3 4 5)

List

(1 2 3 4 5)



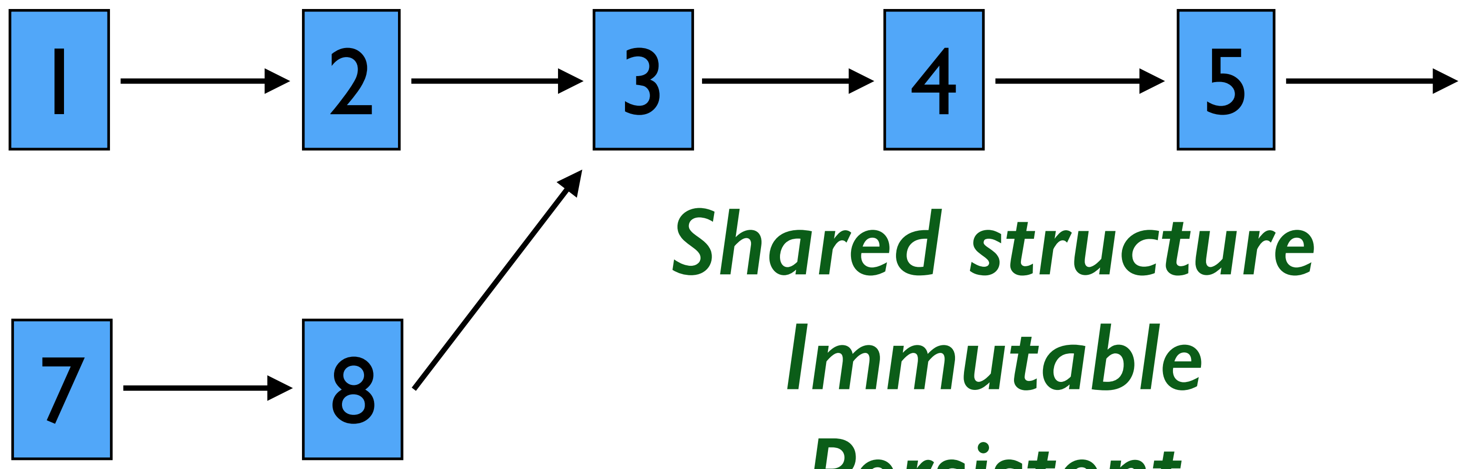
Shared structure
Immutable
Persistent

(7 8 3 4 5)

List

O(1) at the head
O(n) at the tail

```
( 1  2  3  4  5 )
```



Shared structure
Immutable
Persistent

```
( 7  8  3  4  5 )
```

Vector

```
[ 1  2  3  4  5 ]
```

Vector

```
[ 1  2  3  4  5 ]
```

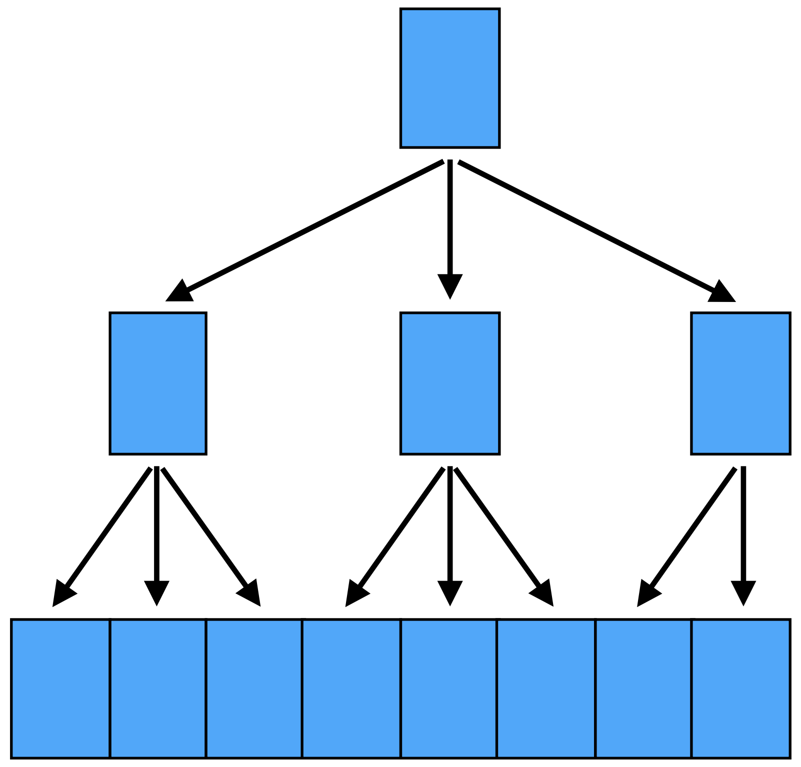
Shared structure
Immutable
Persistent

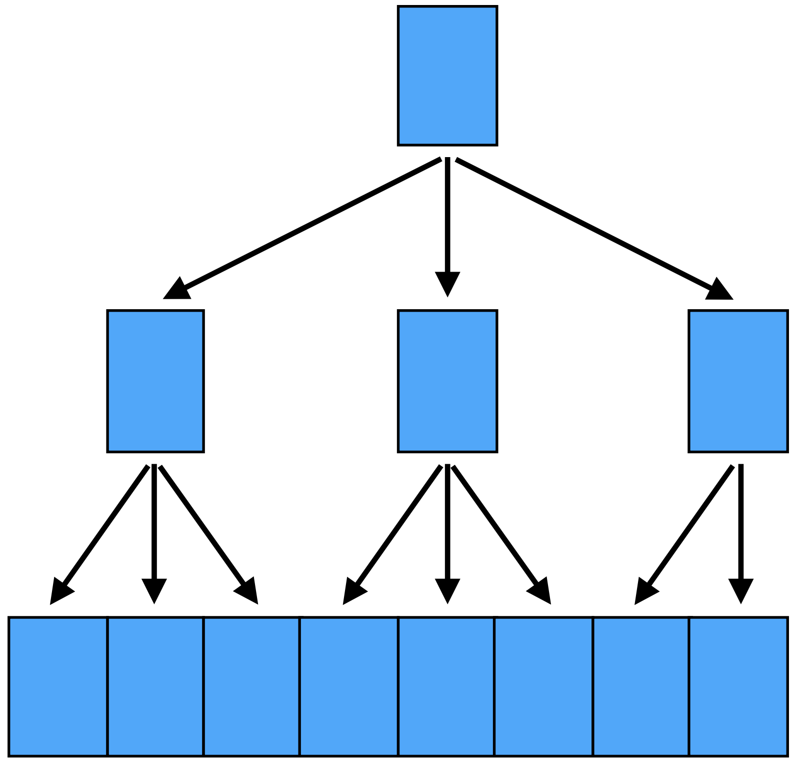
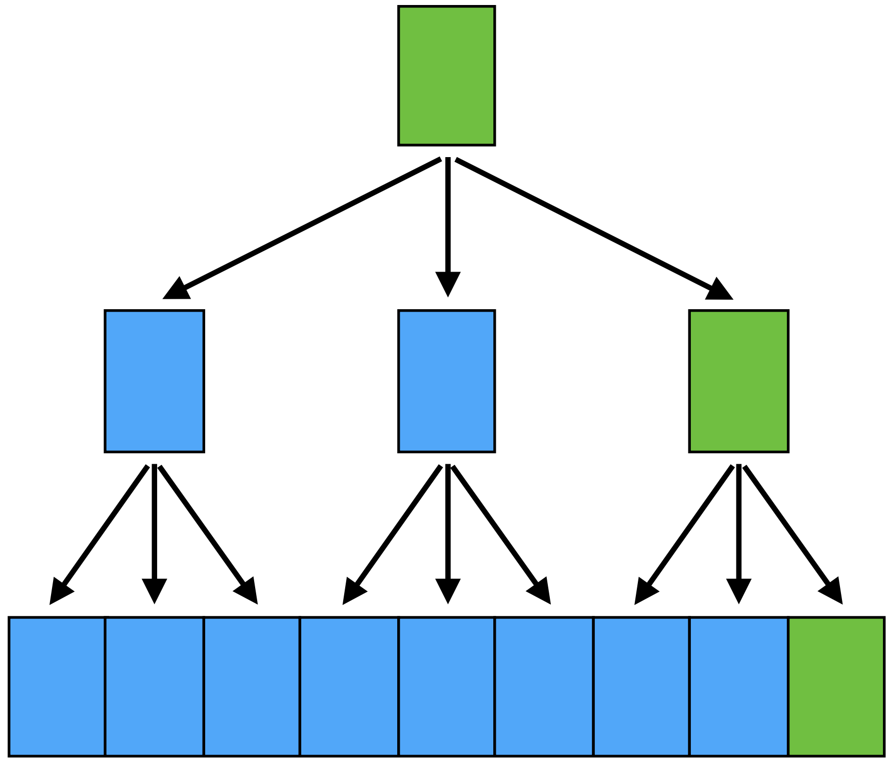
Vector

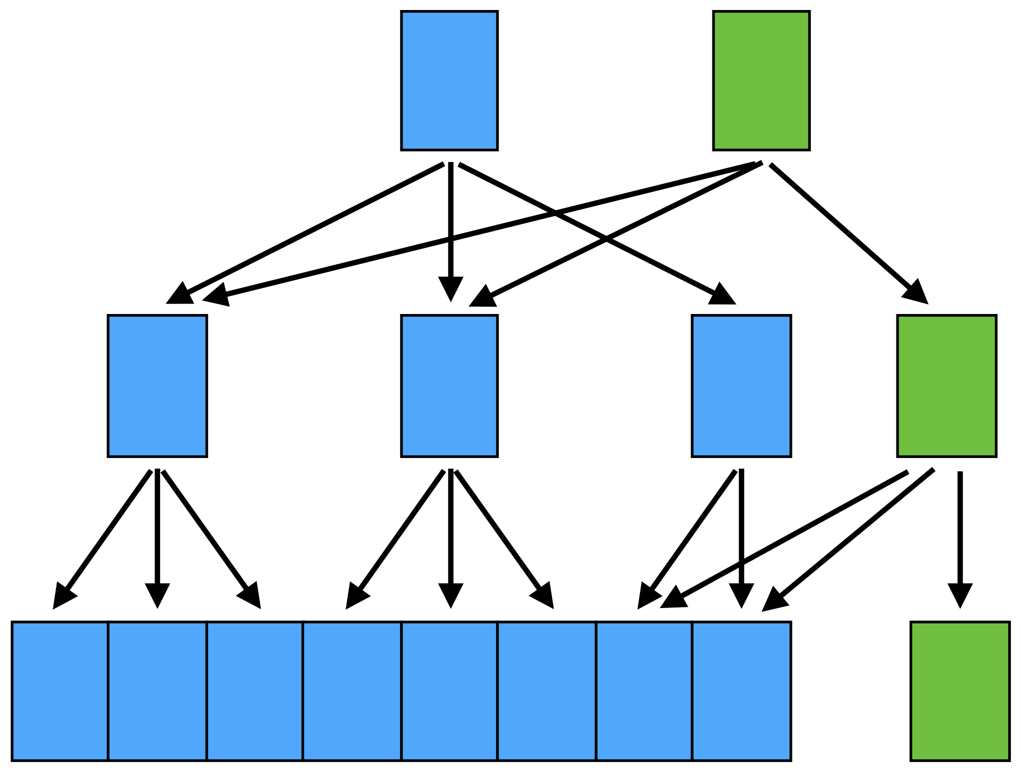
O(1) to read anywhere
O(1) to grow at the end

```
[ 1  2  3  4  5 ]
```

Shared structure
Immutable
Persistent

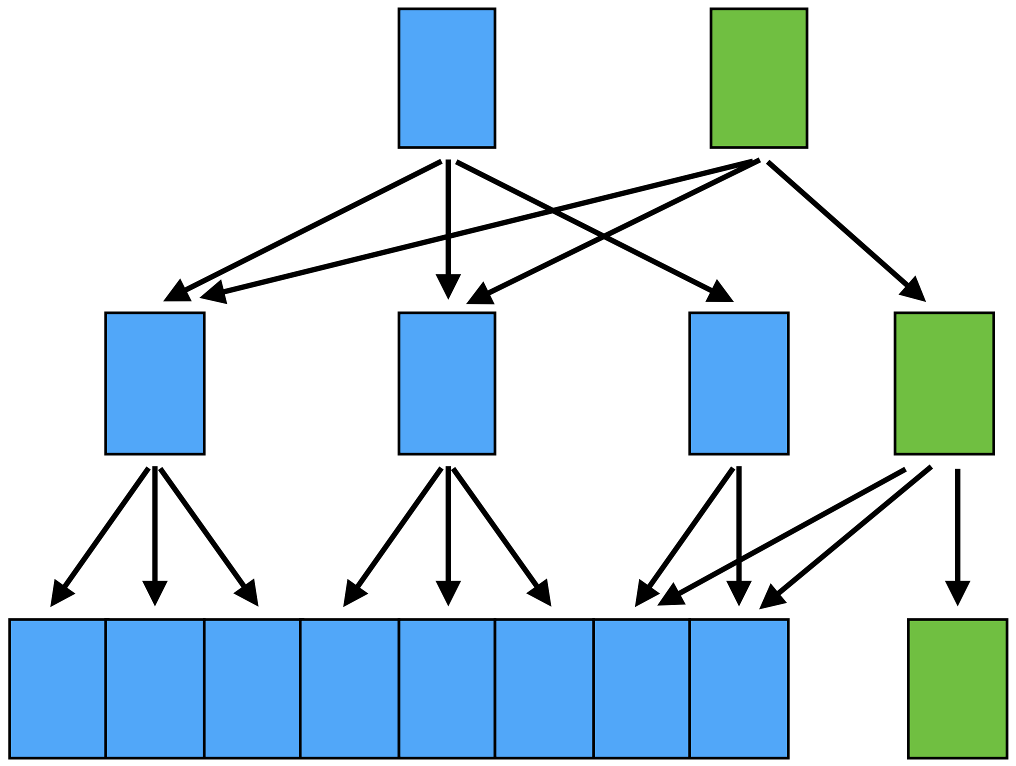


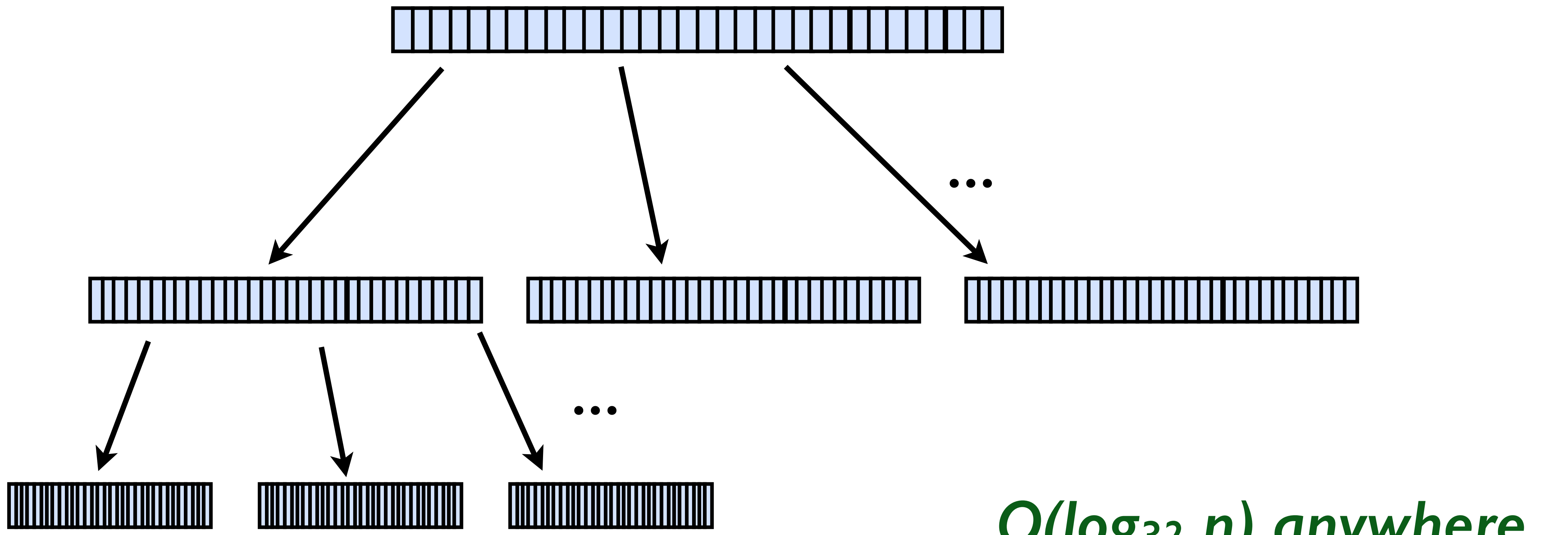


Shared structure
Immutable
Persistent

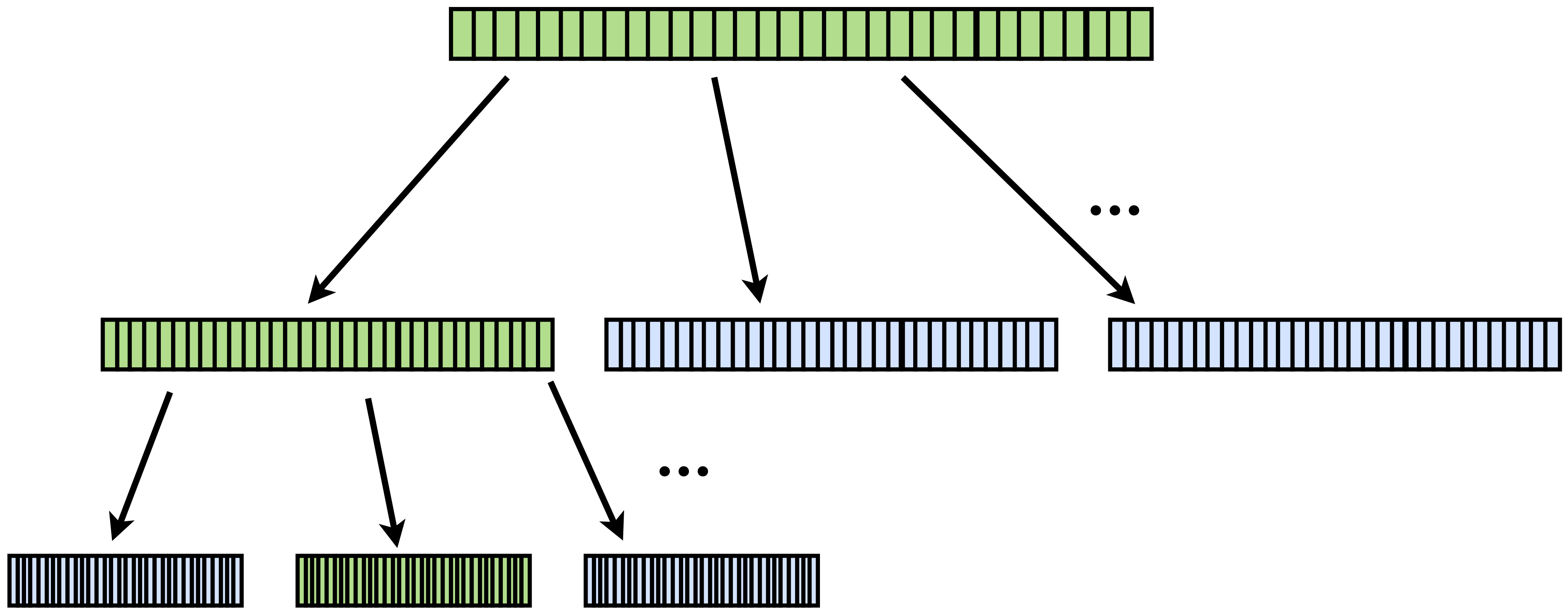
O(log n) anywhere



Shared structure
Immutable
Persistent



O(log$_{32}$ n) anywhere



$O(\log_{32} n)$

$\log_{32} 1000 < 2$
$\log_{32} 10{,}000 < 3$

$\log_{32} 1{,}000{,}000 < 4$
$\log_{32} 10{,}000{,}000 < 5$

$\log_{32} 1{,}000{,}000{,}000 < 6$

$O(\log_{32} n)$

$\log_{32} 1000 < 2$
$\log_{32} 10{,}000 < 3$

$\log_{32} 1{,}000{,}000 < 4$
$\log_{32} 10{,}000{,}000 < 5$

$\log_{32} 1{,}000{,}000{,}000 < 6$

$O(1)$ for $n <$ one billion

Vector

$O(\log_{32} n)$

$O(1)$ for $n <$ one billion

```
[ 1  2  3  4  5 ]
```

Shared structure

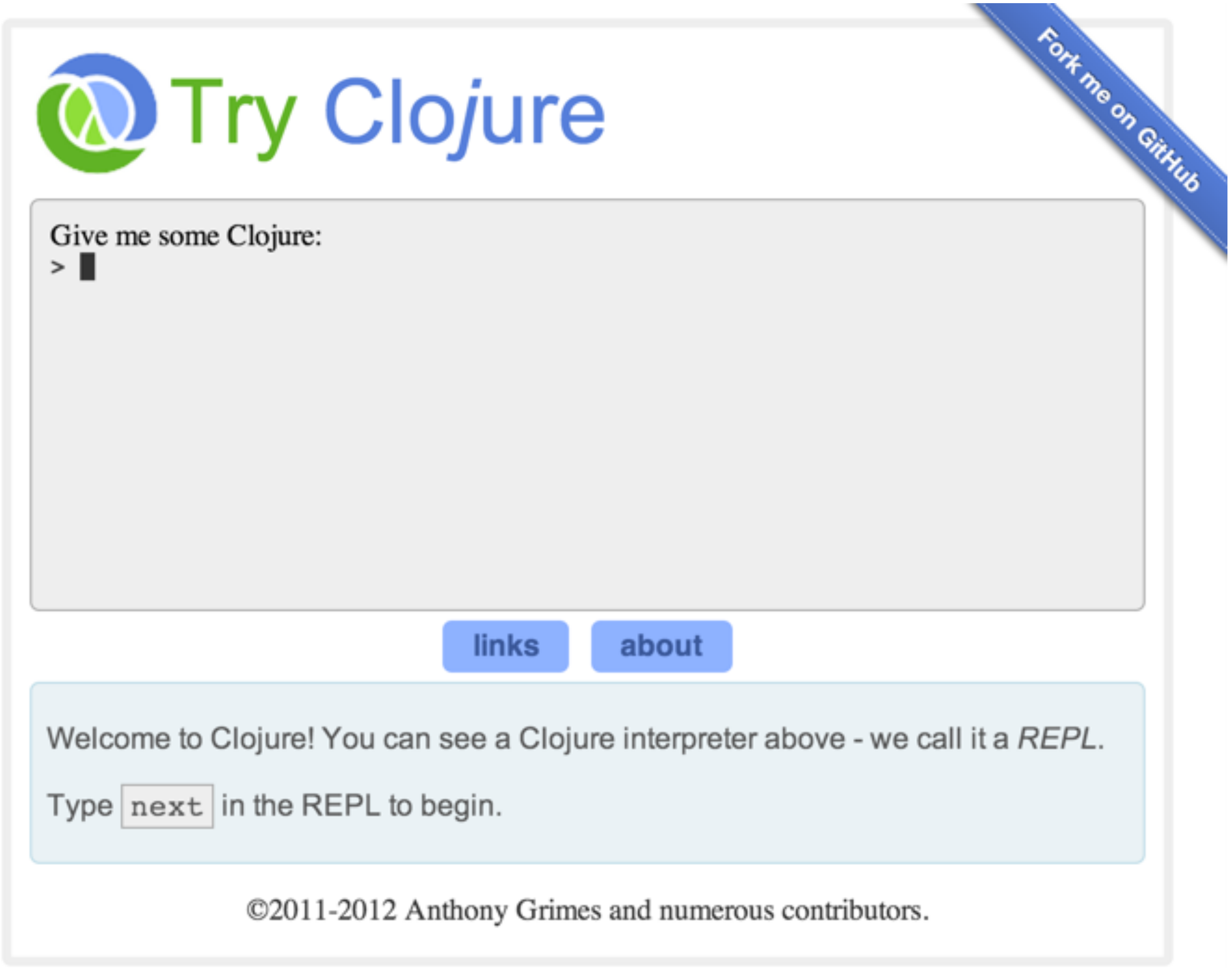
Immutable

Persistent

Hello Clojure

```
(println  "Hello, world!")
```

tryclj.com



```
Give me some Clojure:
> 
```

Welcome to Clojure! You can see a Clojure interpreter above - we call it a *REPL*.

Type `next` in the REPL to begin.

©2011-2012 Anthony Grimes and numerous contributors.

```
( println  "Hello, world!" )
```

Syntax

Symbol *String*

List `( println "Hello, world!" )`

Semantics

Function *Arguments*

Invocation `( println "Hello, world!" )`

Function Definition

```
( defn  greet  [ name ]
  ( println "Hello," name ) )
```

Syntax

```
           Symbols          Vector
List ( defn  greet  [ name ]
    List ( println "Hello," name ) )
```

Semantics

Function definition *Name* *Parameters*

```
( defn greet [ name ]
```

Body

```
( println "Hello," name ) )
```

REPL

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

REPL

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

Read

Evaluate

Print

Loop

REPL

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

Side effect

Return value

Read
Evaluate
Print
Loop

Statements & Expressions

```
for (int i = 0; i < 100; i++) {
  String s;
  if (i % 3 == 0) {
    s = "Fizz";
  } else if (i % 5 == 0) {
    s = "Buzz";
  } else {
    s = new Integer(i).toString();
  }
  System.out.println(s); }
```

Statements & Expressions

```
for (int i = 0; i < 100; i++) {
  String s;
  if (i % 3 == 0) {
    s = "Fizz";
  } else if (i % 5 == 0) {
    s = "Buzz";
  } else {
    s = new Integer(i).toString();
  }
  System.out.println(s); }
```

Statements & Expressions

```
for (int i = 0; i < 100; i++) {
  String s;
  if (i % 3 == 0) {
    s = "Fizz";
  } else if (i % 5 == 0) {
    s = "Buzz";
  } else {
    s = new Integer(i).toString();
  }
  System.out.println(s); }
```

Only Expressions

```
(dotimes [i 100]
  (println
    (cond (zero? (rem i 3)) "Fizz"
          (zero? (rem i 5)) "Buzz"
          :else i)))
```

Lots of Parens?

	Java	Clojure
( )	12	14
{ }	8	0
[ ]	0	2
Total	20	16

Functions

Defining Functions

```
(defn average [& numbers]
  (/ (reduce + numbers)
     (count numbers)))
```

```
(average 3 7 24)  ➡  34/3

(average 7.0 5.0 10 12)  ➡  8.5
```

Defining Functions

```clojure
(def average
  (fn [& numbers]
    (/ (reduce + numbers)
       (count numbers))))
```

Pure Functions

```
(def v [1 2 3])


(conj v 4 5)      ➡ [1 2 3 4 5]
```

“Conjoin”

```
v                 ➡ [1 2 3]
```

Maps

Keyword

```
(def m {:piece "Queen" :score 9})
```

```
(get m :score)  ➡ 9
```

“Associate”

```
(assoc m :letter \Q)
➡ {:piece "Queen"
   :score 9
   :letter \Q}
```

Maps as Functions

```clojure
(def m {:piece "Queen" :score 9})


(get m :score)      ➡ 9

(m :score)          ➡ 9

(:score m)          ➡ 9
```

Higher-Order Functions

```
(def v [1 2 3])

(map inc v)          ➡  (2 3 4)

(filter odd? v)      ➡  (1 3)

(reduce + v)         ➡  6
```

Sequences

```
(reduce +
  (take 500
    (filter odd?
      (range))))
```

```
250000
```

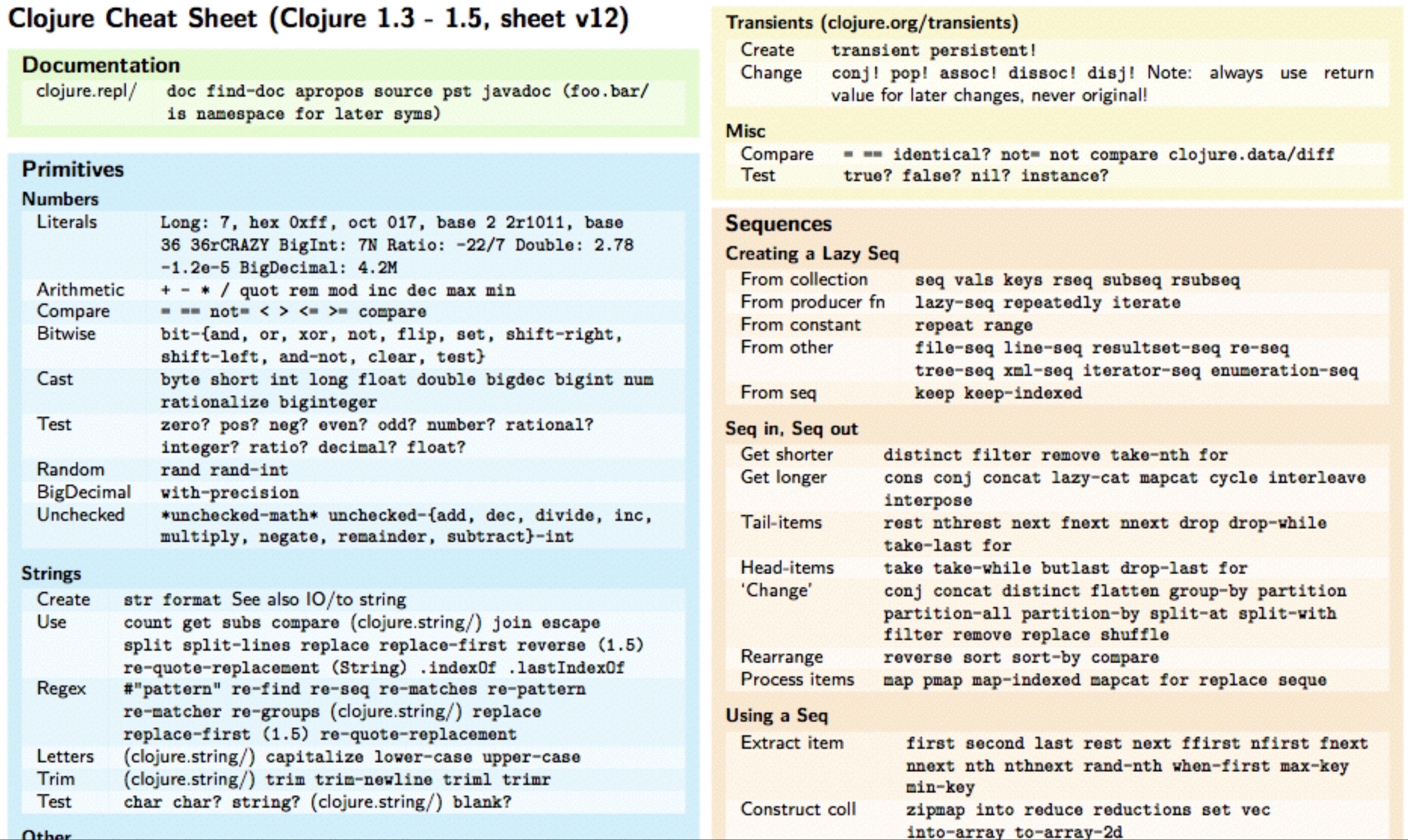
Sequence Generators

- Data structures
- Functions
- Files in a directory
- Lines in a file
- Nodes in an XML document
- Rows in a database query result

Sequence API

- Map, filter, reduce
- Subsequences, splits, and joins
- Sorting and grouping
- Cycles
- Interleaving

clojure.org/cheatsheet



Clojure Cheat Sheet (Clojure 1.3 - 1.5, sheet v12)

Documentation

clojure.repl/	doc find-doc apropos source pst javadoc (foo.bar/ is namespace for later syms)

Primitives

Numbers

Literals	Long: 7, hex 0xff, oct 017, base 2 2r1011, base 36 36rCRAZY BigInt: 7N Ratio: -22/7 Double: 2.78 -1.2e-5 BigDecimal: 4.2M
Arithmetic	+ - * / quot rem mod inc dec max min
Compare	= == not= < > <= >= compare
Bitwise	bit-{and, or, xor, not, flip, set, shift-right, shift-left, and-not, clear, test}
Cast	byte short int long float double bigdec bigint num rationalize biginteger
Test	zero? pos? neg? even? odd? number? rational? integer? ratio? decimal? float?
Random	rand rand-int
BigDecimal	with-precision
Unchecked	*unchecked-math* unchecked-{add, dec, divide, inc, multiply, negate, remainder, subtract}-int

Strings

Create	str format See also IO/to string
Use	count get subs compare (clojure.string/) join escape split split-lines replace replace-first reverse (1.5) re-quote-replacement (String) .indexOf .lastIndexOf
Regex	#"pattern" re-find re-seq re-matches re-pattern re-matcher re-groups (clojure.string/) replace replace-first (1.5) re-quote-replacement
Letters	(clojure.string/) capitalize lower-case upper-case
Trim	(clojure.string/) trim trim-newline triml trimr
Test	char char? string? (clojure.string/) blank?

Other

Transients (clojure.org/transients)

Create	transient persistent!
Change	conj! pop! assoc! dissoc! disj! Note: always use return value for later changes, never original!

Misc

Compare	= == identical? not= not compare clojure.data/diff
Test	true? false? nil? instance?

Sequences

Creating a Lazy Seq

From collection	seq vals keys rseq subseq rsubseq
From producer fn	lazy-seq repeatedly iterate
From constant	repeat range
From other	file-seq line-seq resultset-seq re-seq tree-seq xml-seq iterator-seq enumeration-seq
From seq	keep keep-indexed

Seq in, Seq out

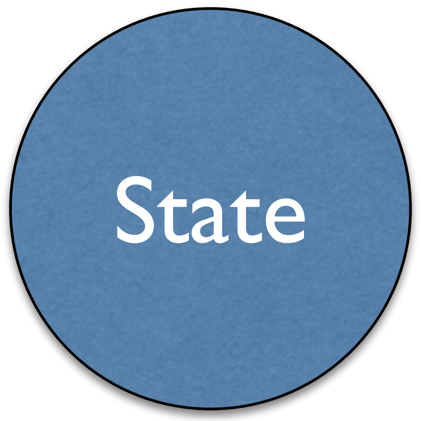
Get shorter	distinct filter remove take-nth for
Get longer	cons conj concat lazy-cat mapcat cycle interleave interpose
Tail-items	rest nthrest next fnext nnext drop drop-while take-last for
Head-items	take take-while butlast drop-last for
'Change'	conj concat distinct flatten group-by partition partition-all partition-by split-at split-with filter remove replace shuffle
Rearrange	reverse sort sort-by compare
Process items	map pmap map-indexed mapcat for replace seque

Using a Seq

Extract item	first second last rest next ffirst nfirst fnext nnext nth nthnext rand-nth when-first max-key min-key
Construct coll	zipmap into reduce reductions set vec into-array to-array-2d

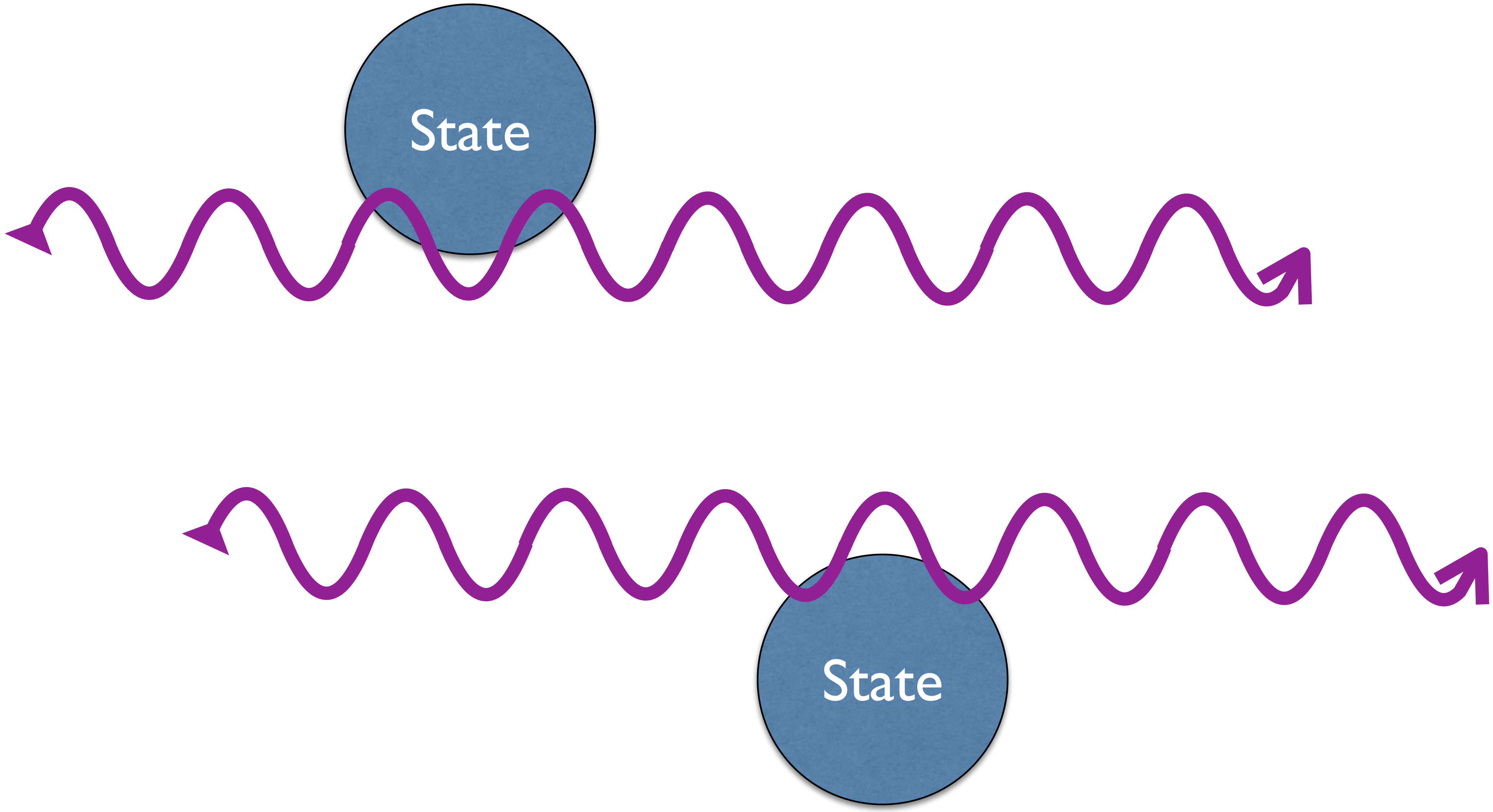
doc

```
user=> (doc filter)
-------------------------
clojure.core/filter
([pred coll])
  Returns a lazy sequence of the items in
coll for which (pred item) returns true.
pred must be free of side-effects.
```

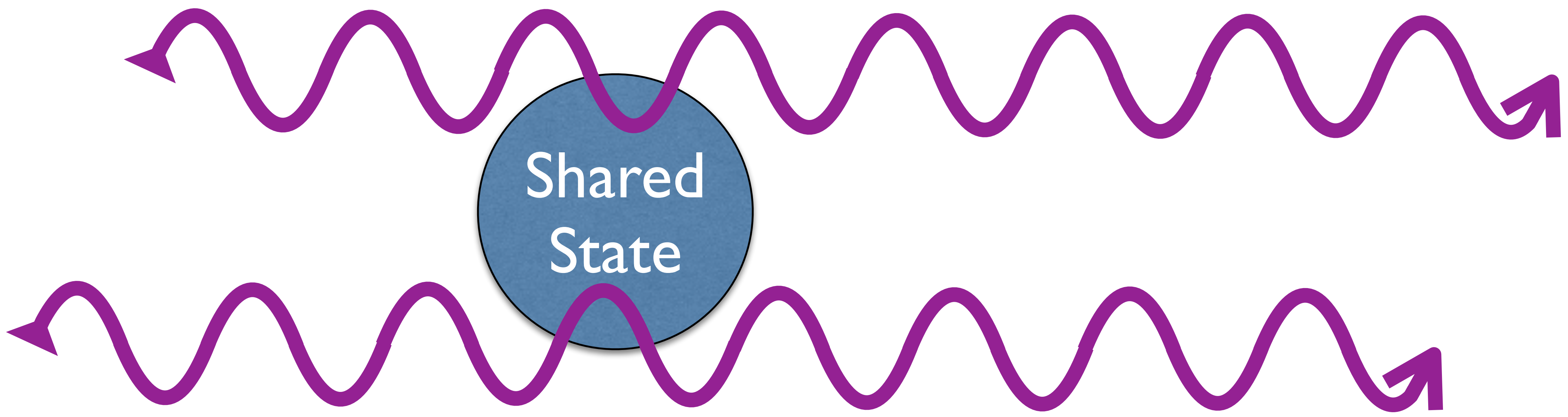

State

Concurrency

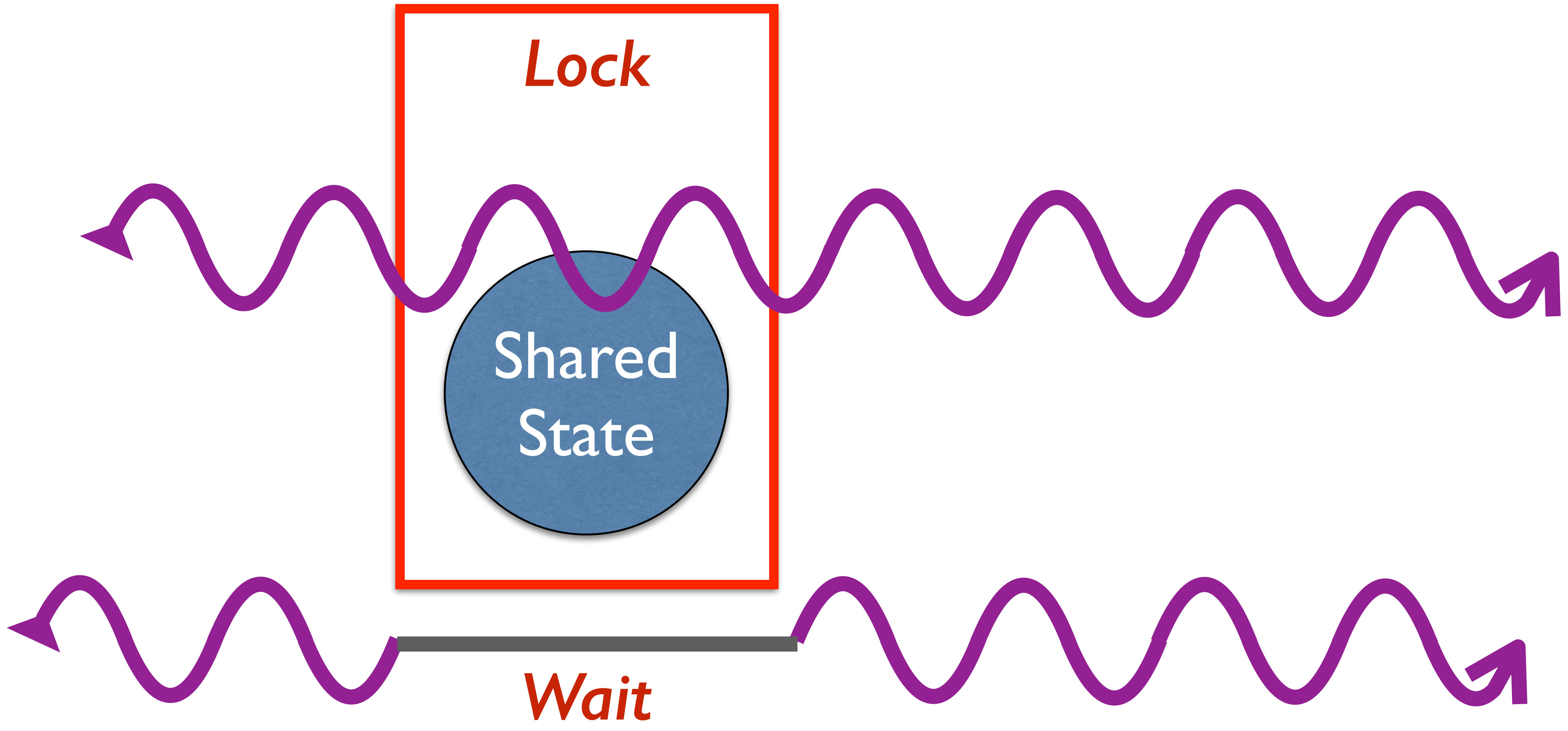
Parallelism



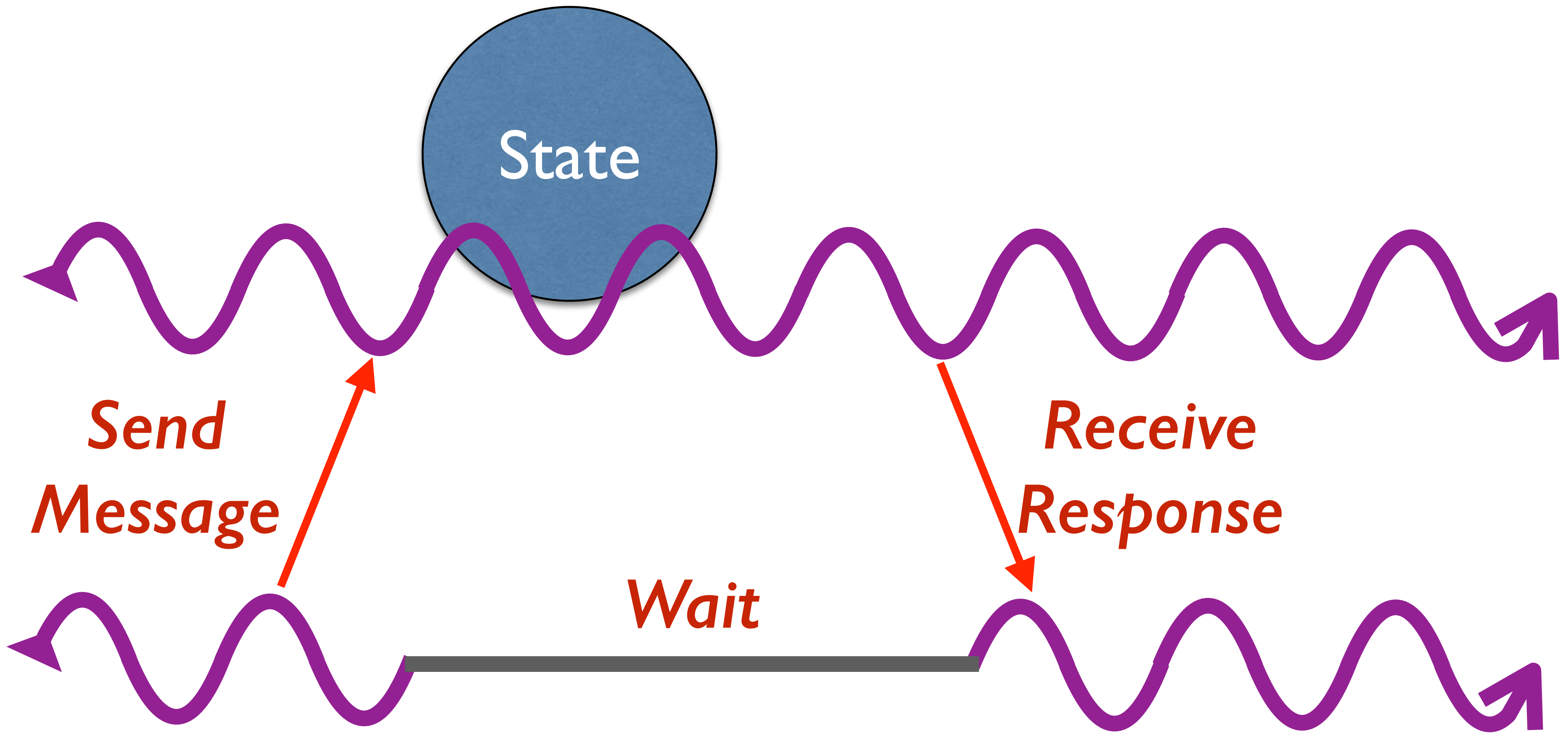
Concurrency



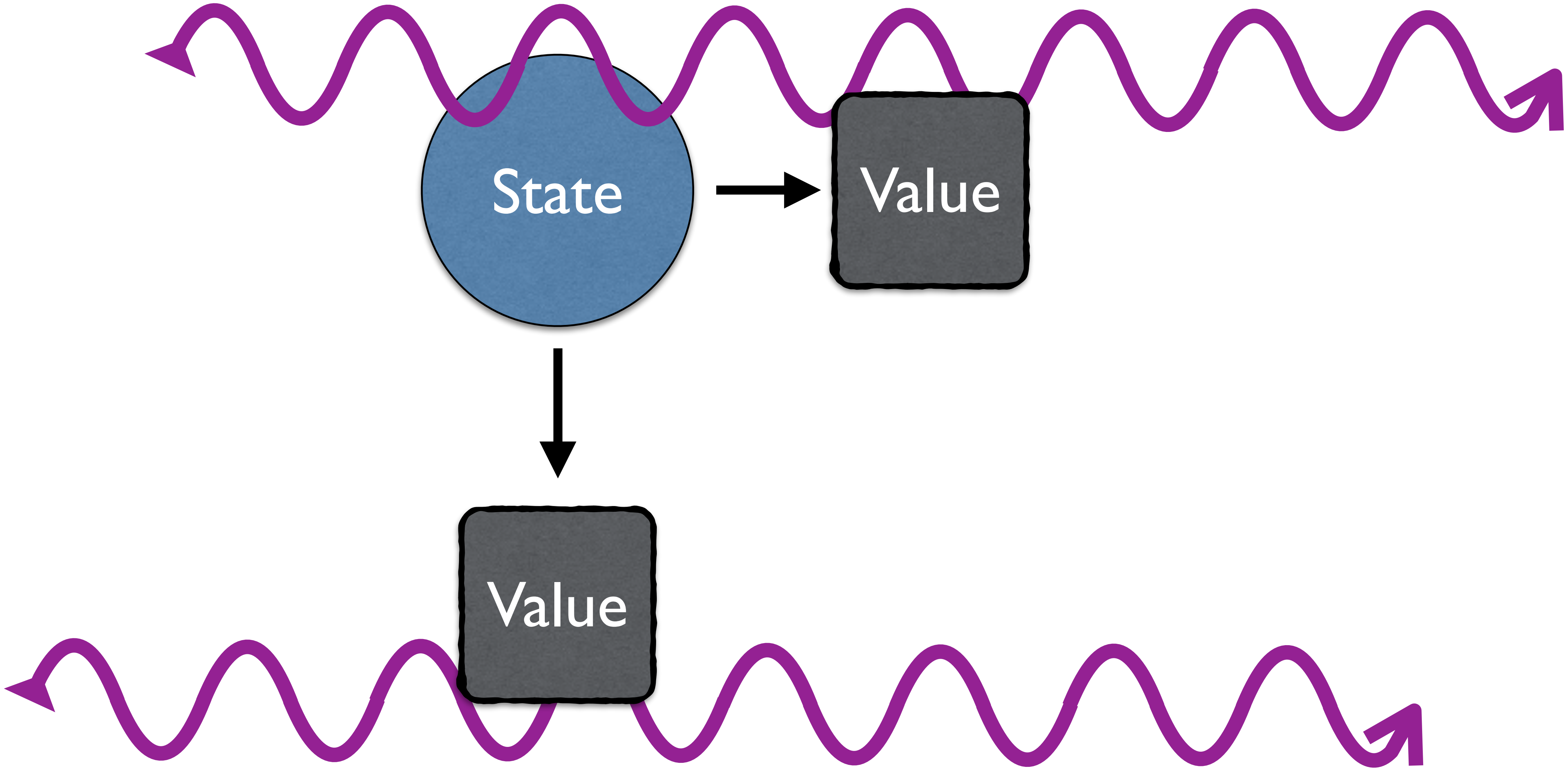
Locks



Message Passing

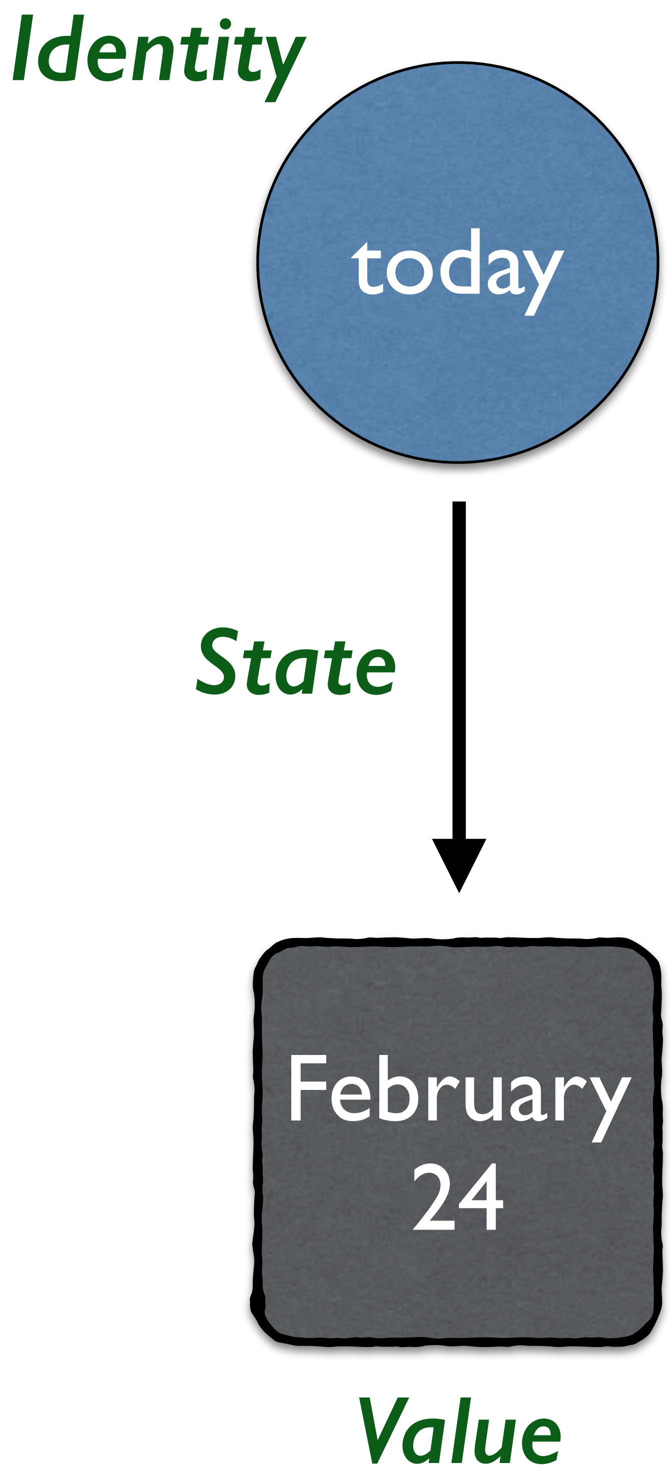


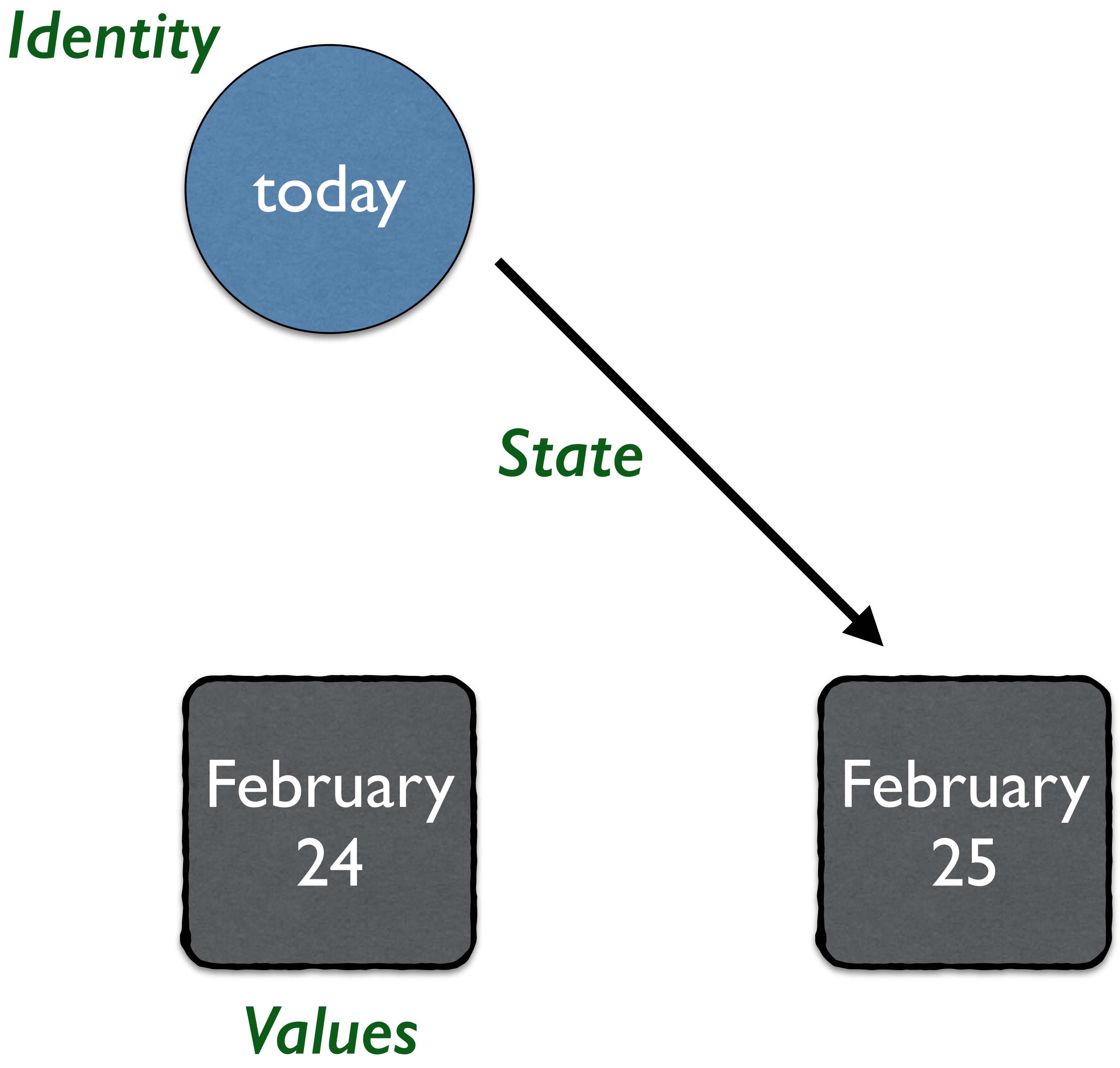
Shared Immutable Values

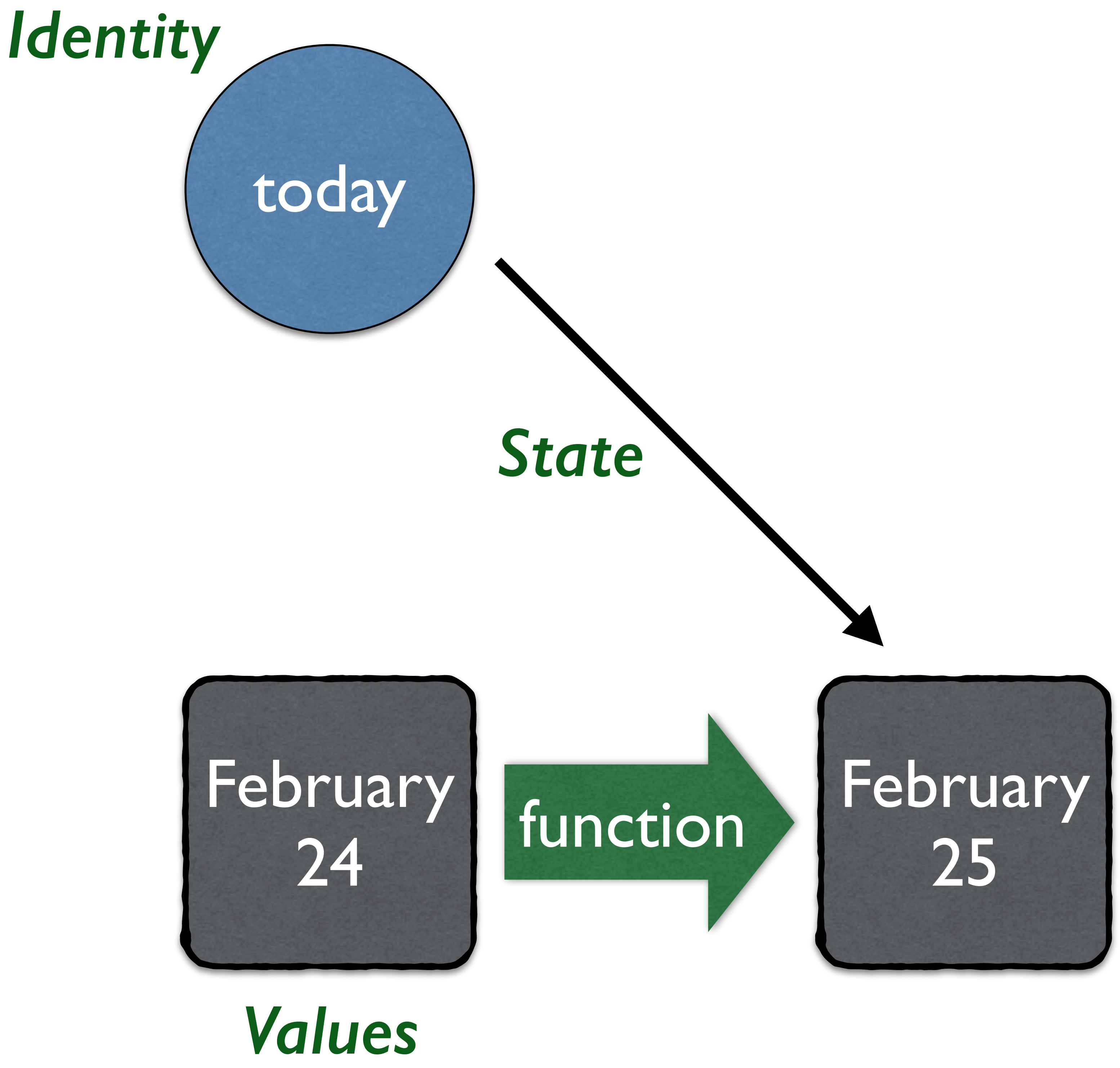


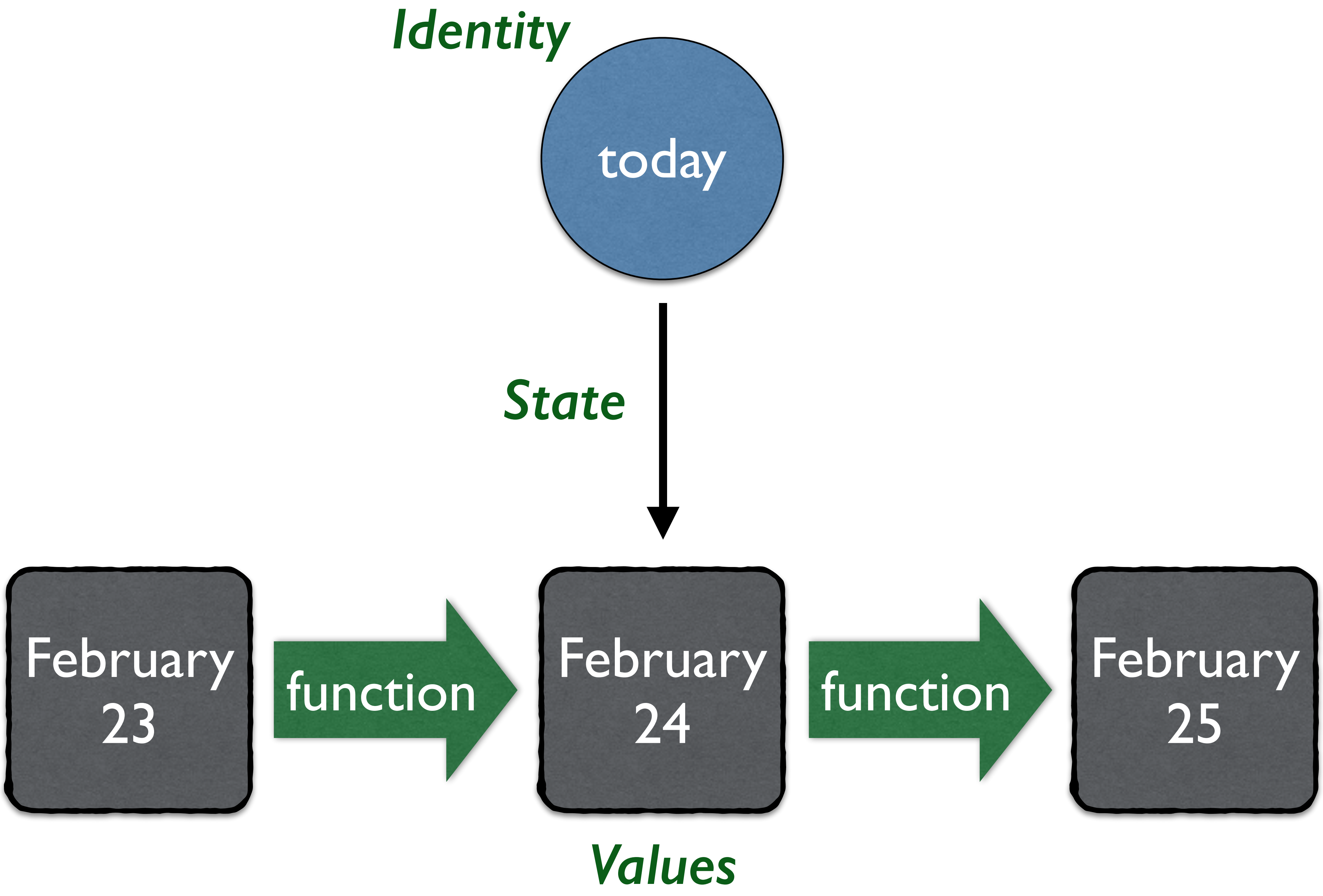
today

Identity


Identity
today
State
February 24
Value


Identity
today
State
February 24
February 25
Values


Identity
today
State
February 24
function
February 25
Values

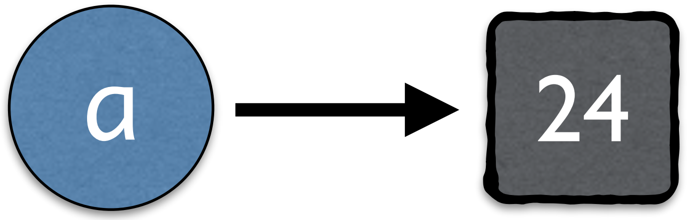

Identity
today
State
February 23
function
February 24
function
February 25
Values

Mutable References

	Synchronous	Asynchronous
Atomic and isolated	Atom	Agent
Coordinated by transactions	Ref	

Atoms

```
(def a (atom 24))
```



```
(deref a)
```

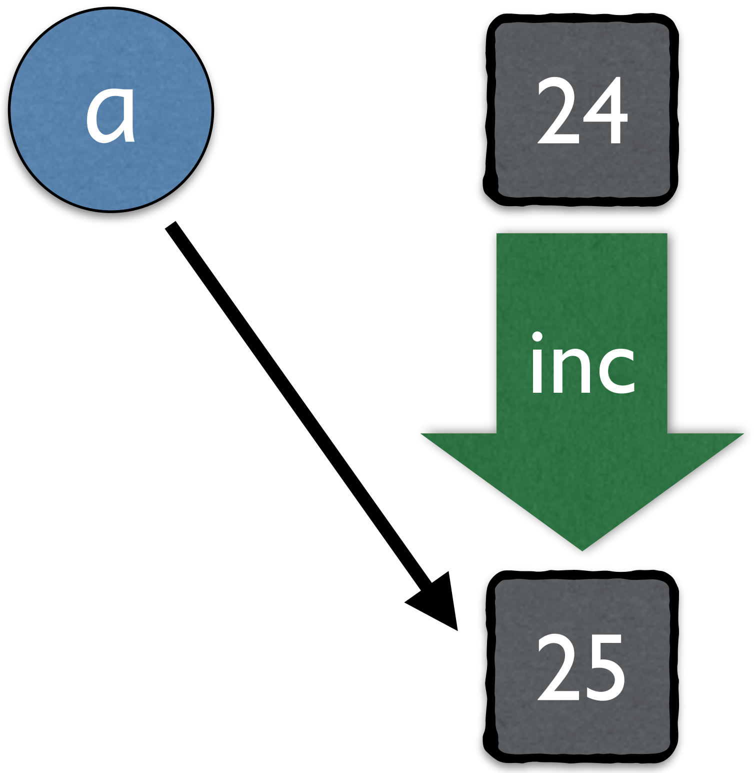
$\Rightarrow$ 24

```
@a
```

$\Rightarrow$ 24

Atoms

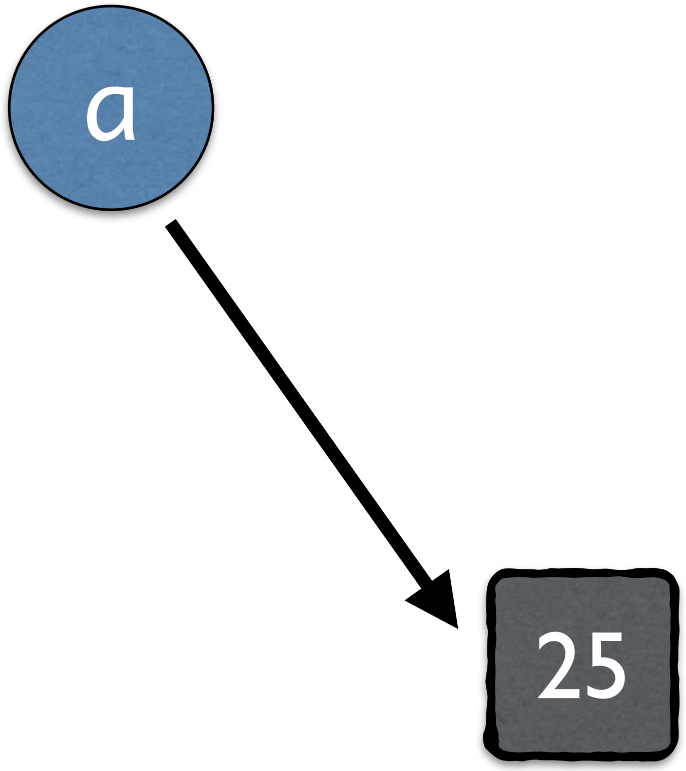
```
(def a (atom 24))


(swap! a inc)
```



Atoms

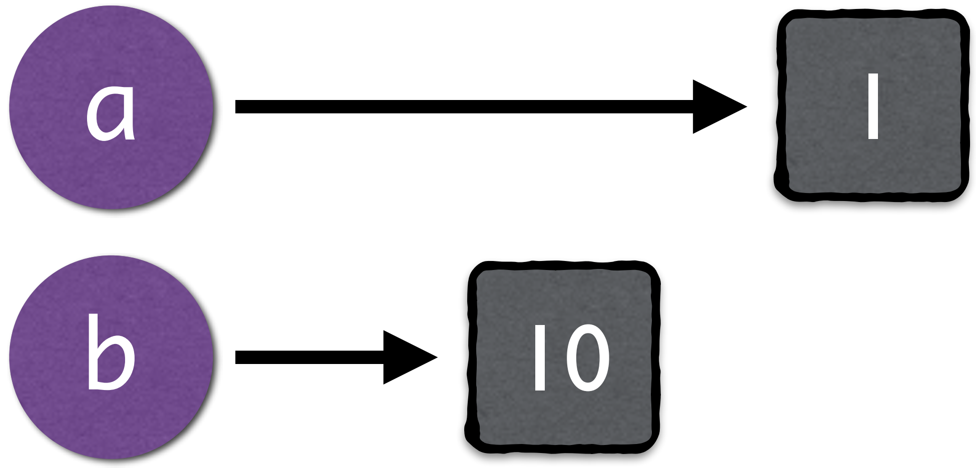
```
(def a (atom 24))

(swap! a inc)
```

@a ➡ 25



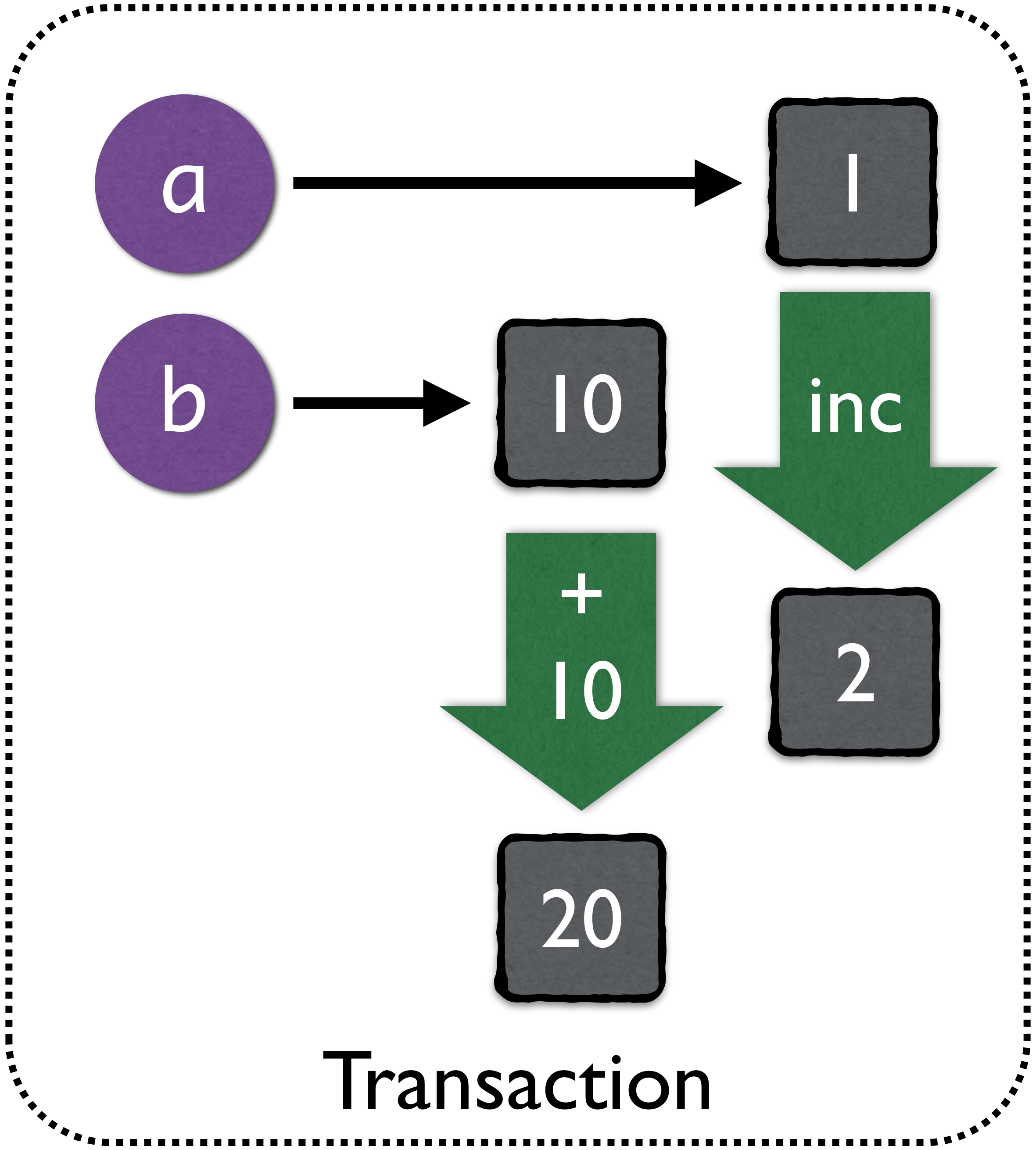
Refs

```
(def a (ref 1))
(def b (ref 10))
```



Refs

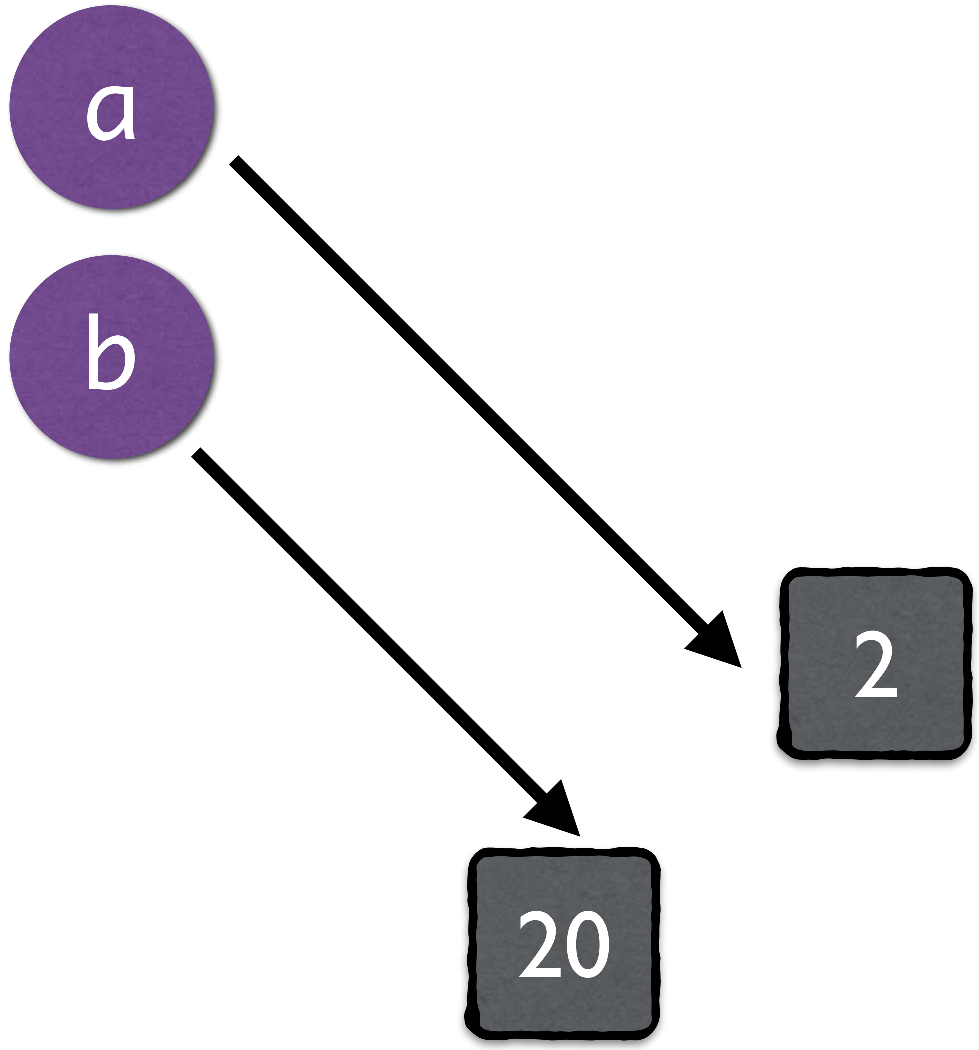
```
(def a (ref 1))
(def b (ref 10))

(dosync
  (alter a inc)
  (alter b + 10))
```



Refs

```
(def a (ref 1))
(def b (ref 10))

(dosync
  (alter a inc)
  (alter b + 10))
```

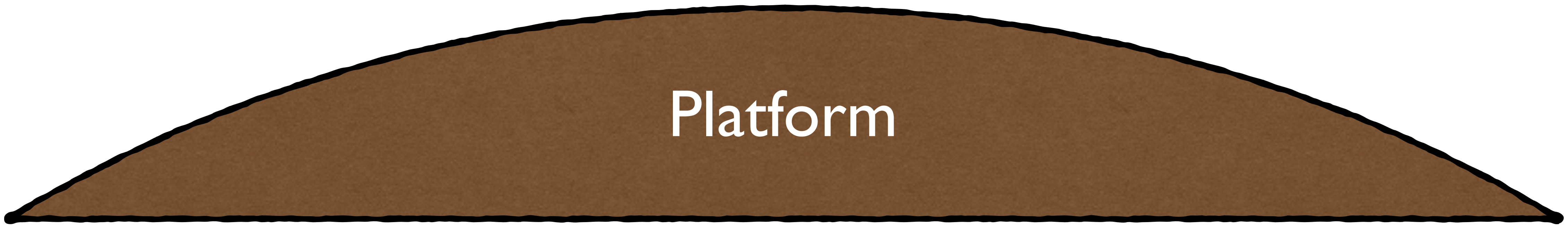
```
@a  ➡ 2

@b  ➡ 20
```



Communication

core.async

```
(defn search [query & servers]
  (let [c (chan)
        t (timeout 150)]
    (doseq [server servers]
      (thread
        (>!! c (do-search server query))))
    (go (alt! c ([result] result)
              t :timed-out))))
```


Platform

Java Virtual Machine

- Garbage collector
- Just-in-time optimizing compiler
- Memory model
- Libraries!

Clojure and Java

Clojure	Java
`(.method object arg)`	`object.method(arg)`
`(SomeClass. "foo")`	`new SomeClass("foo")`
`(Math/sin 3.0)`	`Math.sin(3.0)`
`Integer/MAX_VALUE`	`Integer.MAX_VALUE`
`(.-field object)`	`object.field`

Embracing the Host Platform

Clojure Type	Java Type
String, Long, Double	java.lang.String, java.lang.Long, java.lang.Double
List, Vector	java.util.List*
Map, Set	java.util.Map,* java.util.Set*
Function	java.lang.Runnable, java.util.concurrent.Callable

*Read-only immutable portion only

JavaScript

- It's everywhere
- Desktop, mobile, set-top, embedded, servers
- Fast, lightweight runtimes
- Libraries!

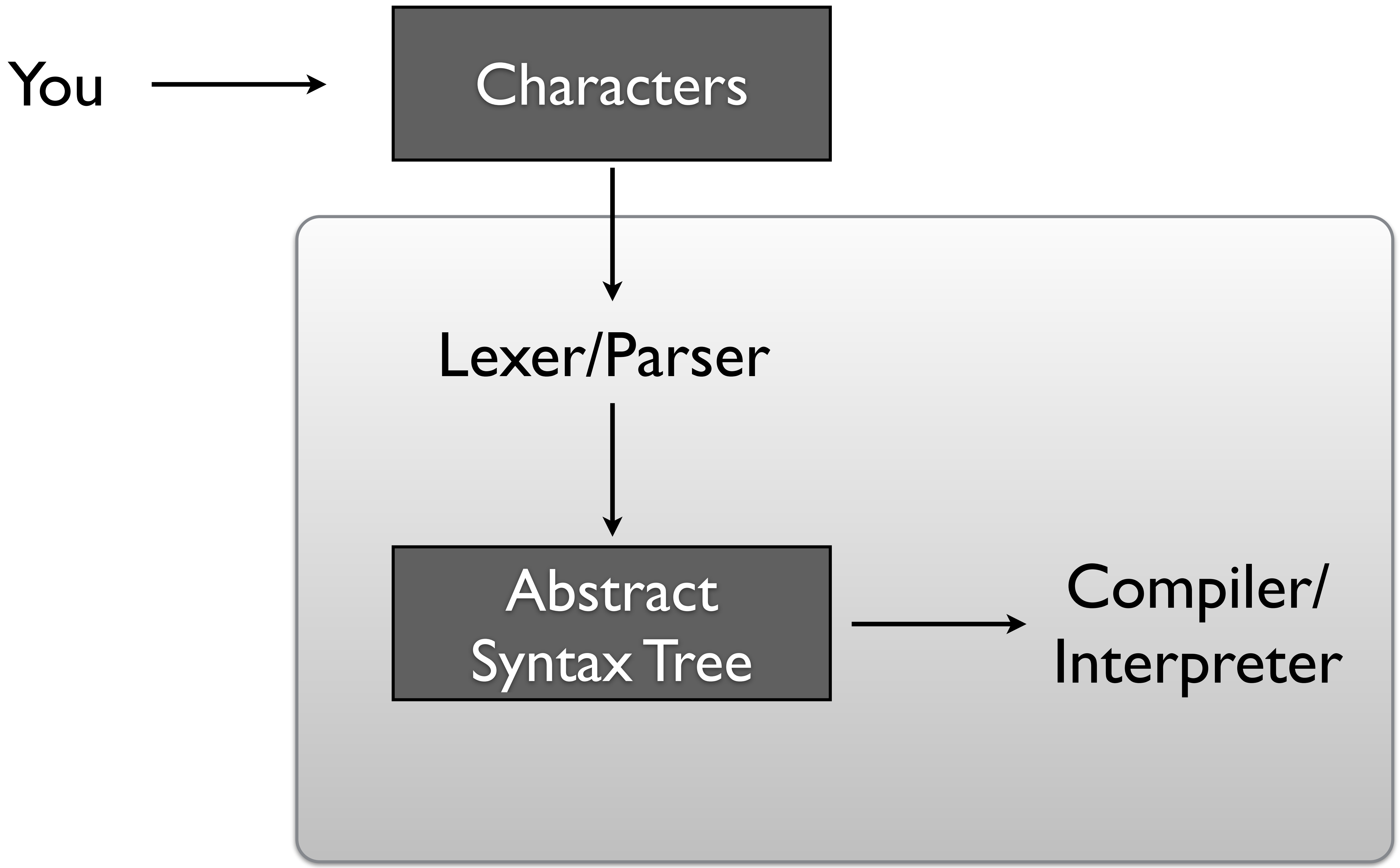
ClojureScript

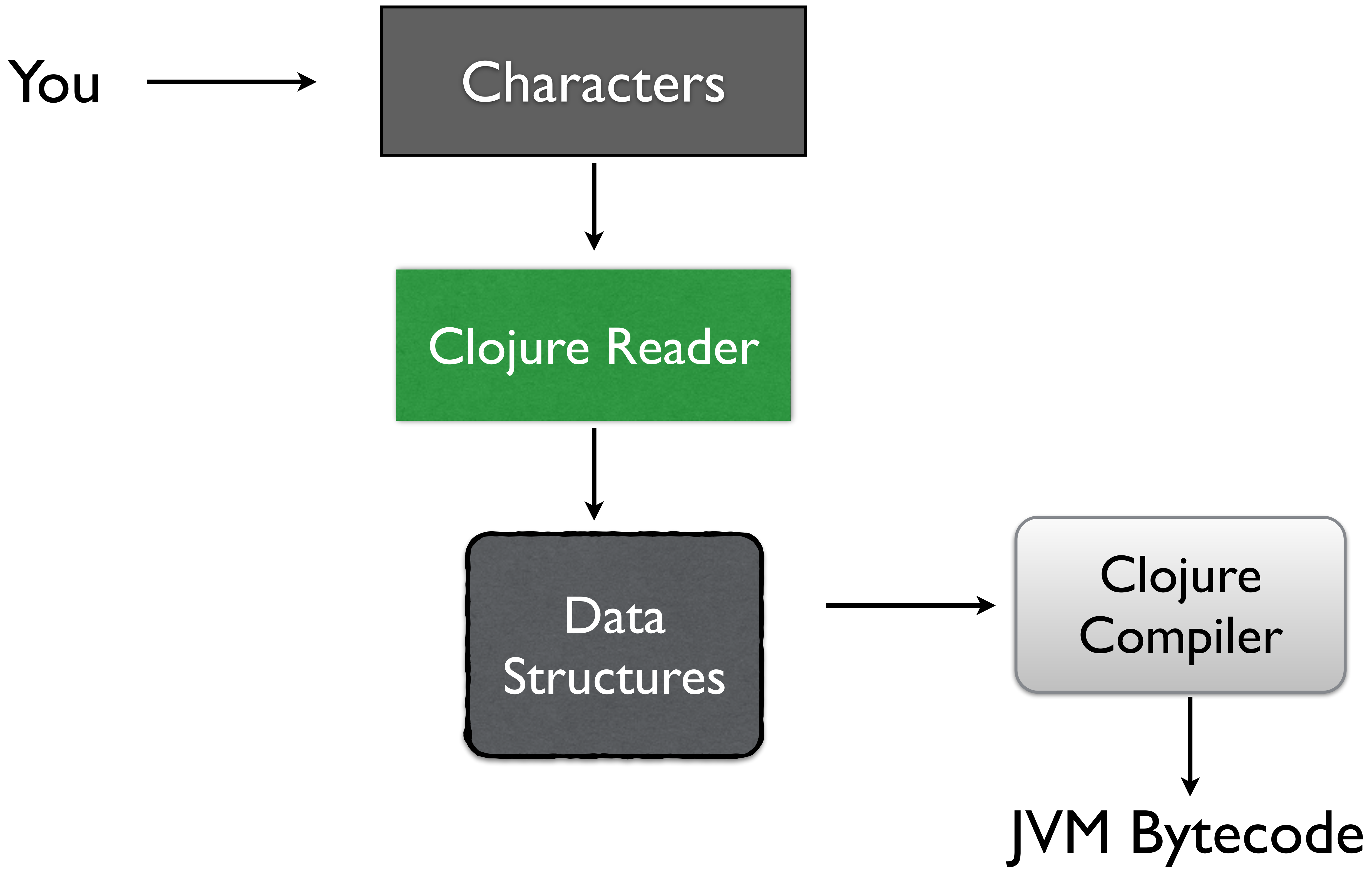
- Clojure compiled to JavaScript
- “Pure” source identical to JVM Clojure
- Host interop and basic types differ
- core.async vs. “callback hell”

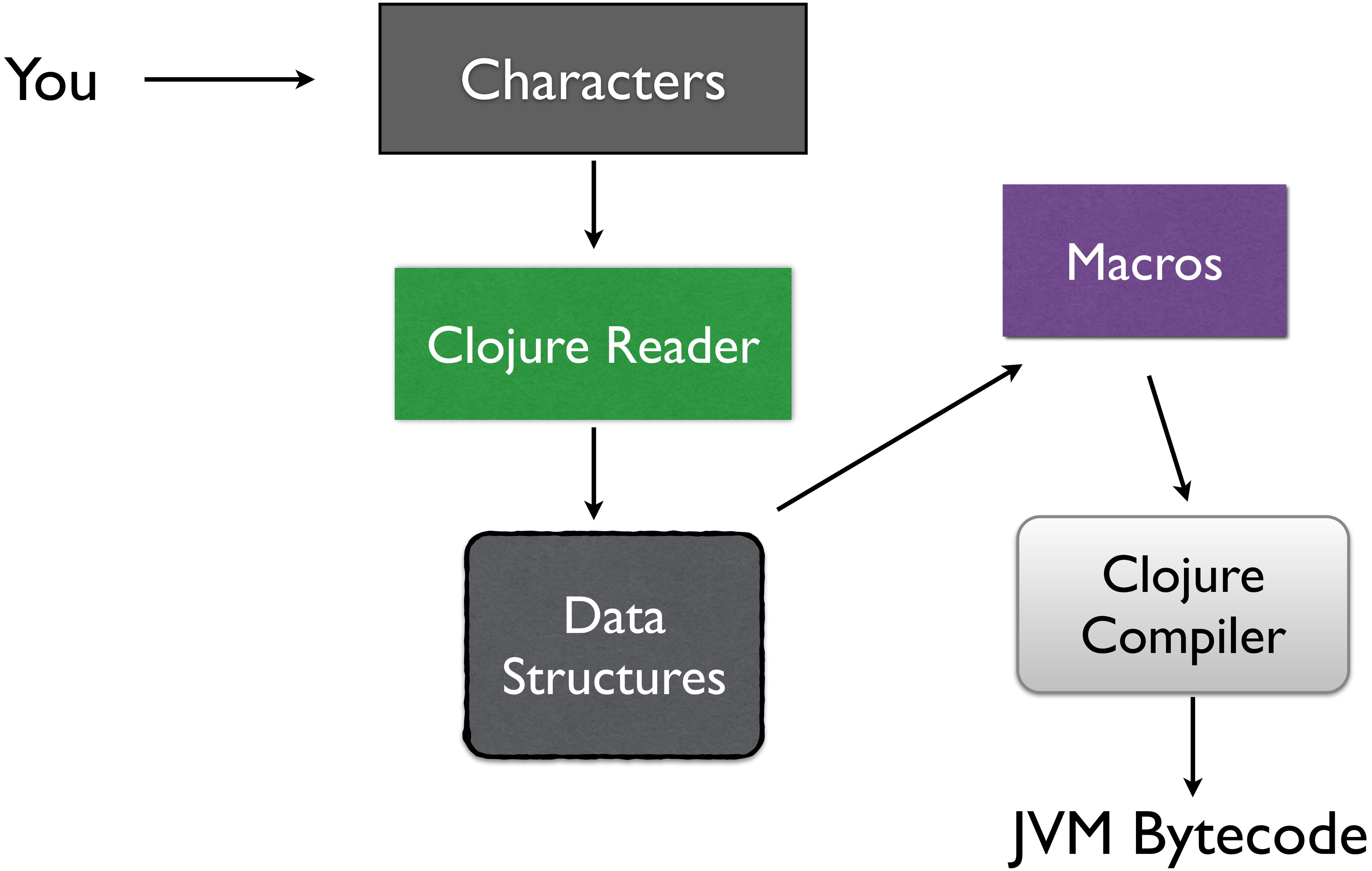
Extensible Data Notation (EDN)

```
{[1 1] :king
 [3 4] :pawn}

#inst "2014-02-23T22:03:15"

#chess/move [:rook 3 8 "takes pawn"]
```

Macros


You
Characters
Lexer/Parser
Abstract Syntax Tree
Compiler/ Interpreter


You
Characters
Clojure Reader
Data Structures
Clojure Compiler
JVM Bytecode


You
Characters
Clojure Reader
Data Structures
Macros
Clojure Compiler
JVM Bytecode

```
(let [file ...initializer...]
  (try ... do something ...
    (finally
      (.close file))))
```

```
(defmacro with-open [bindings & body]
  `(let ~bindings
     (try ~@body
       (finally
         (.close ~(first bindings))))))
```

```
(let [file ...initializer...]
  (try ... do something ...
    (finally
      (.close file))))
```

Macros

```
(with-open [file ...initializer...]
   ... do something ...)
```

Polymorphism

Generic Functions

```
(count [:a :b :c :d])          ➡ 4

(count "Hello, world!")        ➡ 13

(count (.split "1,2,3,4" ","))   ➡ 4
```

Maps

```
(def db {:host "localhost"
         :port 7400
         :dbname "customers"})

(type db)
➡ clojure.lang.PersistentHashMap

(:dbname db)
➡ "customers"
```

Records

```
(defrecord Database [host port dbname])

(def db (->Database "localhost"
                    7400
                    "customers"))

(type db)     ➡ user.Database

(:dbname db) ➡   "customers"
```

Protocols

```
(defprotocol Connection
  (connect [conn]
    "Initiate connection")
  (shutdown [conn]
    "Close database connection"))
```

Records Implement Protocols

```
(defrecord Database [host port conn]
  Connection
  (connect [this]
    (let [conn (DBClient. host port)]
      (assoc this :conn conn)))
  (shutdown [this]
    (.close conn)
    this))
```

Records Implement Protocols

```
(defrecord TestDB [host port conn]
  Connection
  (connect [this]
    (let [conn (DBClient. host port)]
      (create-schema conn)
      (load-seed-data conn)
      (assoc this :conn conn)))
  (shutdown [this]
    (drop-database conn)
    (.close conn)
    this))
```

Tools

Leiningen

- Dependency management (Maven repositories)
- Build automation
- Command-line invocation

Editors & IDEs

- Emacs: clojure-mode, paredit, CIDER, clj-refactor
- Vim: Fireplace
- Eclipse: Counterclockwise
- IntelliJ: Cursive Clojure

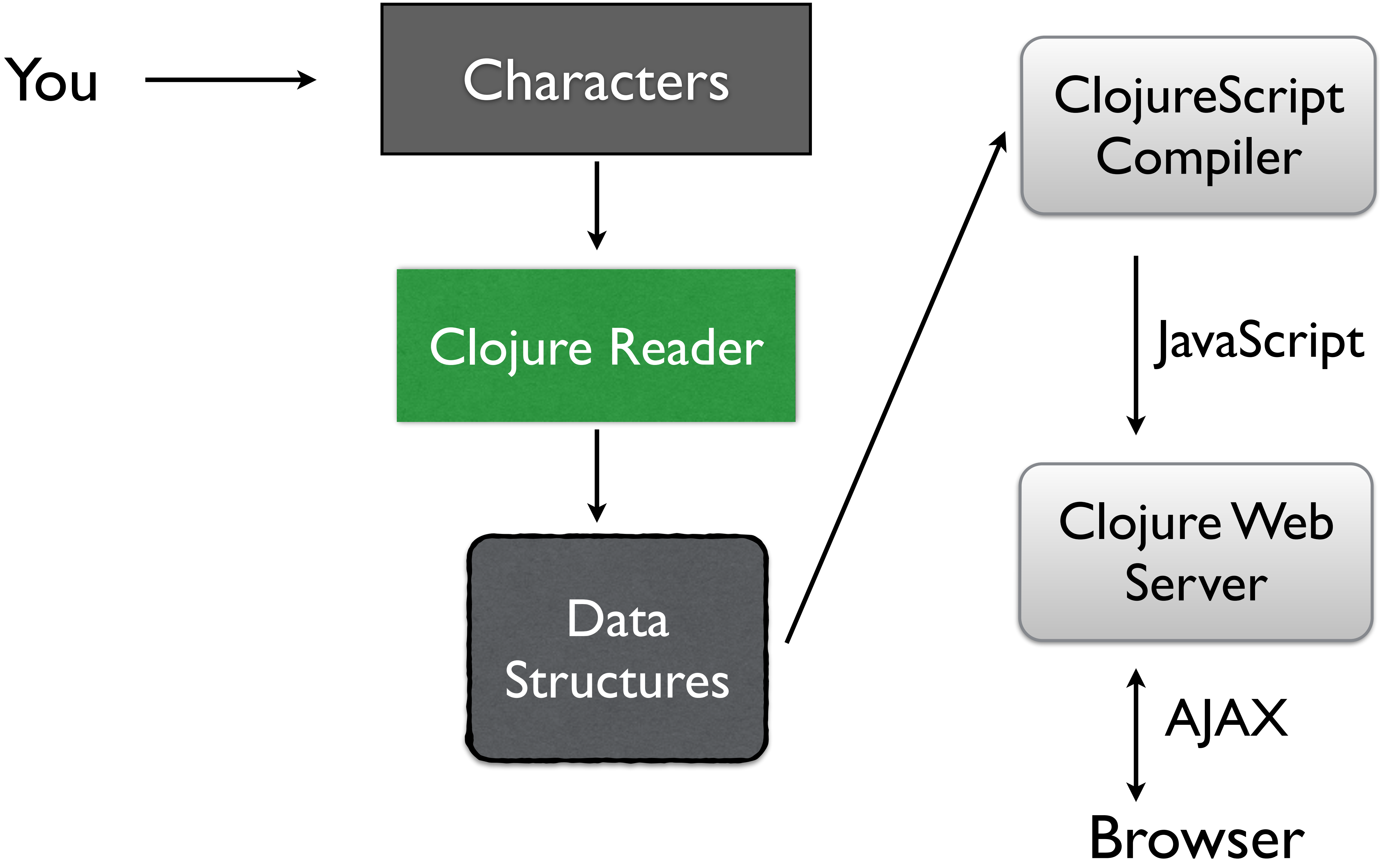
Light Table

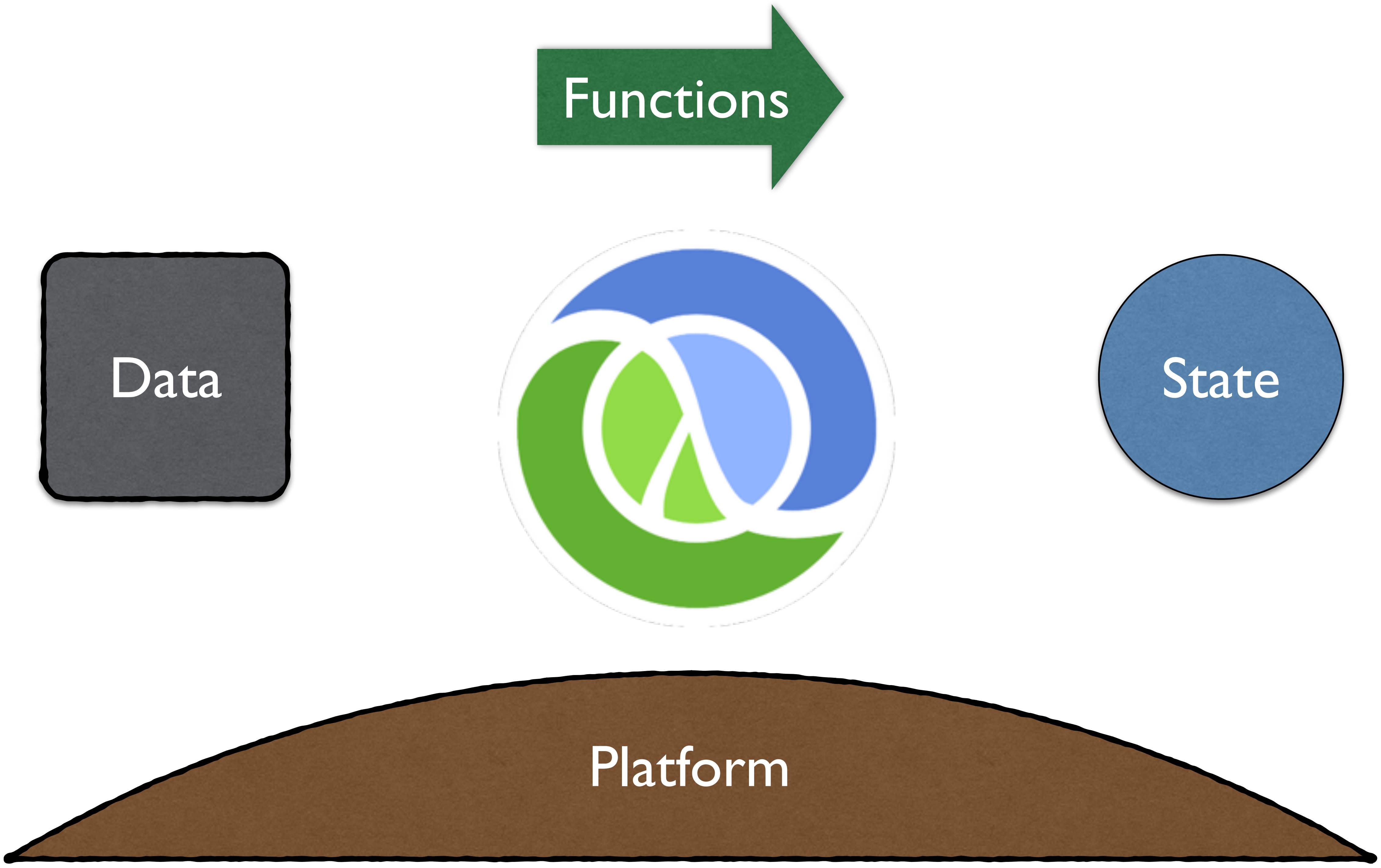
- Clojure / JavaScript / Python IDE
- Written in ClojureScript
- “InstaREPL”
- Embedded browser

nREPL

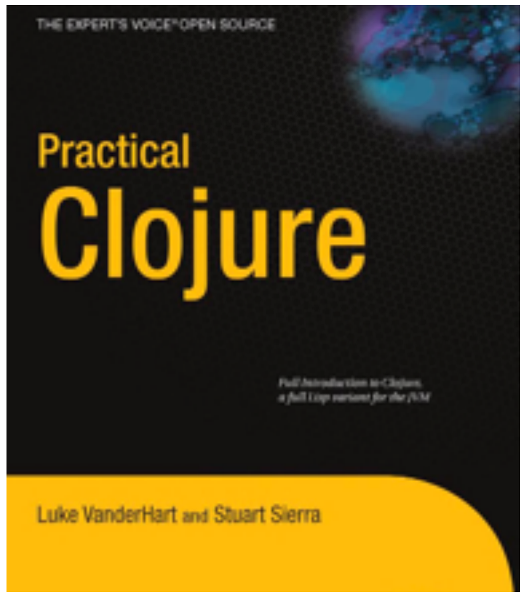
- Network-enabled REPL
- Common backend for tools & IDEs

Browser-connected REPL




Functions
Data
State
Platform

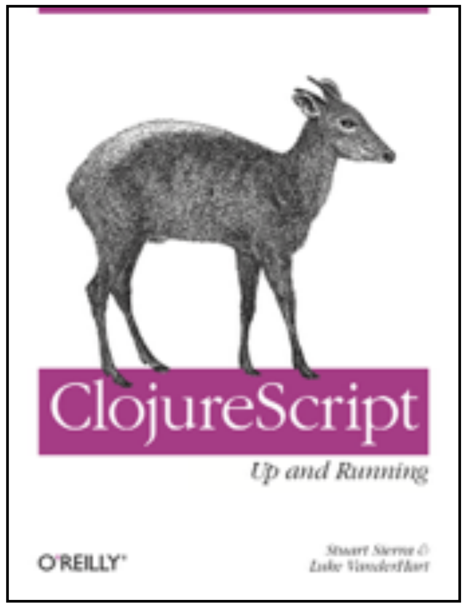
clojure.org


THE EXPERT'S VOICE® OPEN SOURCE

Practical Clojure

Full Introduction to Clojure, a full Lisp variant for the JVM

Luke VanderHart and Stuart Sierra


ClojureScript

Up and Running

O'REILLY®

Stuart Sierra & Luke VanderHart

stuartsierra.com
@stuartsierra

Datomic

cognitect

cognitect.com
@cognitect
clojure.com
datomic.com