CS 596 Functional Programming and Design
Fall Semester, 2015
Doc 3 Clojure Data
Aug 27, 2015

Basic Data Elements

symbols
keywords
literals
lists
vectors
maps
sets

Symbols

Can reference another value

```
(def foo 12)

(defn bar [n] (inc n))
```

When evaluated returns the value

```
foo        12
     ---->
bar        fn
```

When quoted & evaluated returns it self

```
'foo       foo
     ---->
'bar       bar
```

Symbols

Can start with any non-numeric character

Can contain alphanumeric characters and ! $ % & * - + = _ | < > ?

```
(def !$%&*-+=_|<>? "wtf")
```

Unicode is supported

```
(def పన్నెండు 12)
(def बारह 12)
(def বারো 12)
(def બાર 12)
(def ಹನ್ನೆರಡು 12)
(def ਬਾਰ੍ਹਾ 12)
(def பன்னிரெண்டு 12)
```

Keywords

Like symbols but evaluates to itself

Literal syntax starts with a colon

```
:foobar
:2
:?
:ThisIsALongKeyWordWhichShowsThatTheCanBeLong
```

Colon is part of literal syntax, but not the name of the keyword

```
(= :cat (keyword "cat"))        true
(= :cat (keyword ":cat"))       false
```

Literals - Strings & Characters

"A String"

"Another
string"

(class "cat")
java.lang.String

\c

\u00ff
unicode

\o64
octal

Whitespace Characters

\space
\newline
\formfeed
\return
\backspace
tab

So what is \n ?

\n verses "\n"

\n	"\n"
Character n	newline in string

(str "a" \n "b" 5)	*"anb5"*
(str "a" \newline "b")	*"a* *b"*
(str "a" "\n" "b")	*"a* *b"*

Numbers

Long	Double	Ratio
12	2.11	3/5
0xfe	1.3e-4	
2r111		
5r123		
36rCRAZY		

BigInt	BigDecimal
12N	4.2M

(factorial 100N)

93326215443944152681699238856266700490715
96826438162146859296389521759999322991560
89414639761565182862536979208272237582511
852109168640000000000000000000000000N

Cast/Convert

byte	(long 12.8)	*12*
short		
int	(rationalize 0.25)	*1/4*
long		
float	(read-string "12.6")	*12.6*
double		
bigdec	*(str 12.3)*	*"12.3"*
bigint		
num		
rationalize		
biginteger		

Collections

Immutable

Heterogeneous

Persistent

Vectors

Sets

Maps

Lists

Queues

Vectors

Expandable, indexed list [4 "cat" \c]

Fast insert at end [4, "cat", \c]

Expensive insert in front [ ]

Fast indexed lookup

Vectors

(vector 8 4 2)	[8 4 2]
(nth [:a :b :c] 2)	:c
(get ["a" "b" "c"] 2)	"c"
(["a" "b" "c"] 2)	"c"
(nth [:a :b :c] 2 "rat")	:c
(nth [:a :b :c] 4 "rat")	"rat"
(.indexOf ["a" "b" "c"] "b")	1
(peek ["a" "b" "c"])	"c"
(pop ["a" "b" "c"])	["a" "b"]
(conj [1 2 3] 4)	[1 2 3 4]
(assoc [1 2 3] 0 9)	[9 2 3]

Accessing Elements - 3 ways

	nth	get	Vector as function
nil vector	Returns nil	Returns nil	Exception
Index out of range	Exception or "Not found" arg	Returns nil	Exception
Not found arg	Yes	Yes	No

Immutability & Persistence

```
(def a  [1 2 3])
(def b  (conj a 4))
(def c  (assoc b 0 8))
```

a ⟷ [1 2 3]

b ⟷ [1 2 3 4]

c ⟷ [8 2 3 4]

Java

```
int[] d = {1, 2, 3};
d[0] = 8;
```

d ⟷ {8, 2, 3}

Vector Implementation

(def brown [0 1 2 3 4 5 6 7 8])

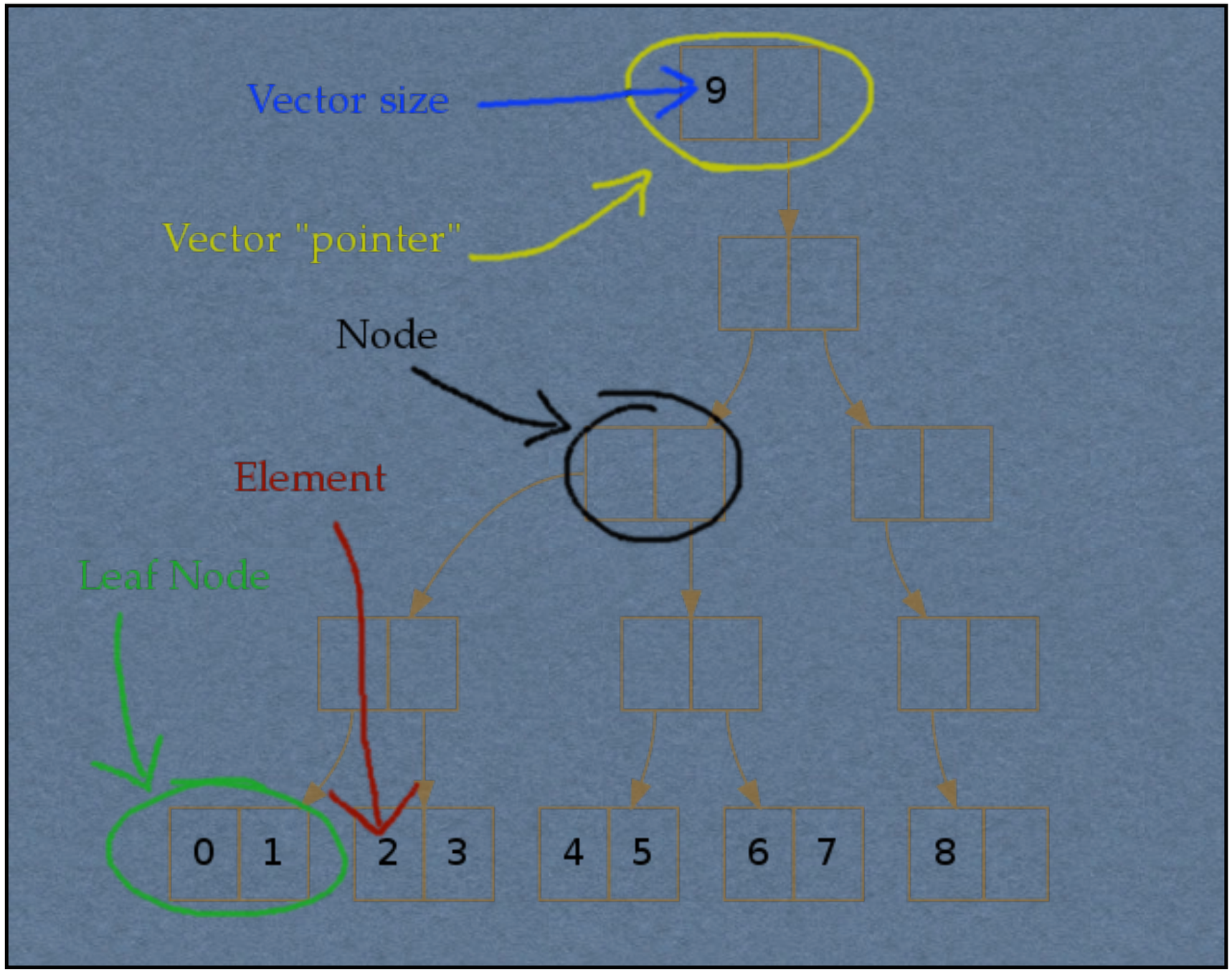
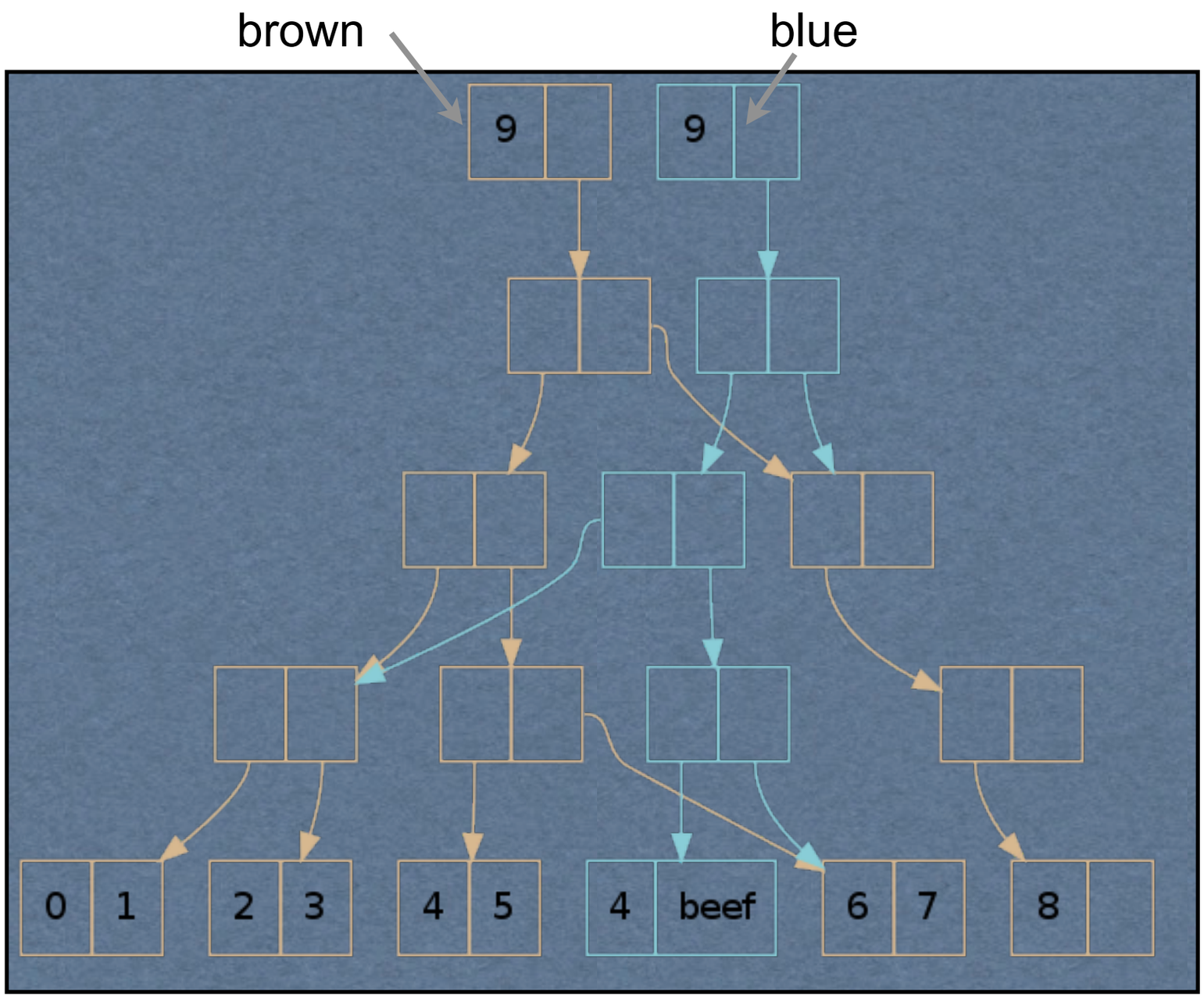


Image from http://hypirion.com/musings/understanding-persistent-vector-pt-1

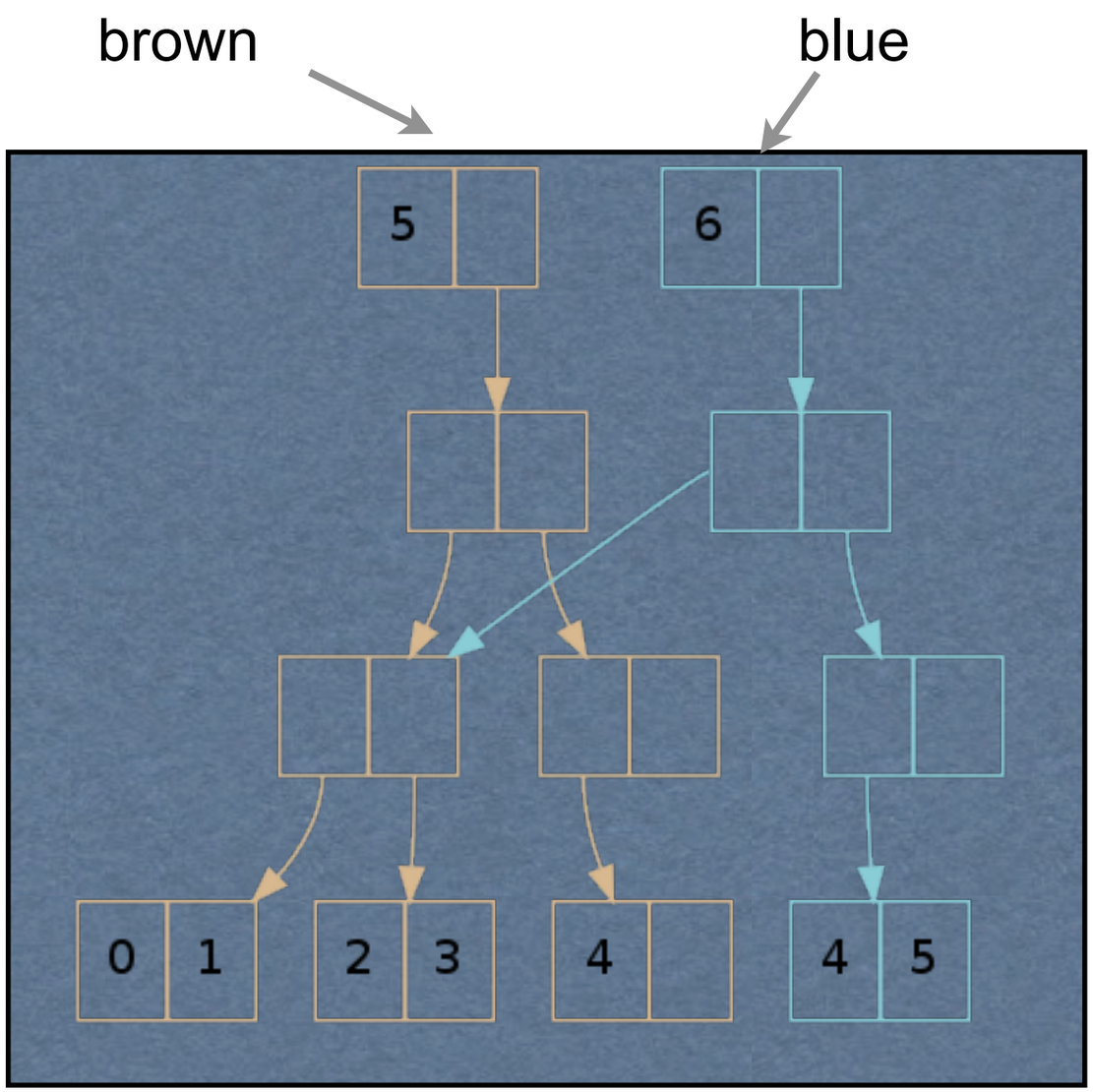
Update

```
(def brown [0 1 2 3 4 5 6 7 8])
(def blue (assoc brown 5 'beef))
```



Adding

```
(def brown [0 1 2 3 4])
(def blue (conj brown 5)
```



More Details

Bit-partitioned trie with branching factor of 32

Nodes

- Contain 32 elements

Tree

- Trie on index of elements

1 billion elements

- Tree of depth 6

Some Operation Costs

Operation	Cost
count	O(1)
first	O(<7)
rest	O(<7)
nth	O(<7)
last	O(n)
get	O(<7)
assoc	O(<7)
peek	O(1) ?

For More Details

See

http://hypirion.com/musings/understanding-persistent-vector-pt-1

http://hypirion.com/musings/understanding-persistent-vector-pt-2

Sets

No duplicates

Fast insert & contains

Sets

(contains? #{1 2} 1)	true
(#{2 4} 2)	2
(#{2 4} 3)	nil
(get #{1 2} 1)	1
(get #{1 2} 3)	nil
(get #{1 2} 3 :not-found)	:not-found
(nth #{4 2} 2)	2
(conj #{ 1 2 } 3 4 5)	#{1 2 3 4 5}
(disj #{1 2 3} 2)	#{1 3}
(clojure.set/intersection #{1 2 3} #{2 4 8})	#{2}

Maps (Hash Table)

Key-value map

Keys - any value

Values - any value

Fast insert & find

Very common

```
{:first-name "Roger"
 :last-name "Whitney" }

{:first-name "Roger",
 :last-name "Whitney" }

{:name {:first "Roger" :last "Whitney" }
 :phone-numbers
    ["111-2222" "222-3333"]}

{ "a" 1, 2 "b", [4 3] :me}

{  }
```

Maps (Hash Table)

(get {:a 1} :a)	1
({:a 1} :a)	1
(:a {:a 1})	1
({2 "b"} 2)	"b"
(2 {2 "b"})	Error
(conj {:a 1 :b 2} {:a 3} {:c 4})	{:c 4, :a 3, :b 2}
(merge {:a 1 :b 2} {:a 3 :c 4})	{:c 4, :a 3, :b 2}
(assoc {:a 1 :b 2} :a 3 :c 4)	{:c 4, :a 3, :b 2}

Naming Conventions

Clojure	Java
all-lower-case words-separated-by-hyphen	camelCase

Lists

Linked List

Fast insert & remove at front

```
'( 1 2 3)

'( "cat" {:a 1})

'(+ 1 2)
```

Lists

(list 8 4 2)	(8 4 2)
(nth '("a" "b" "c") 2)	"c"
('("a" "b" "c") 2)	Error
(.indexOf '("a" "b" "c") "b")	1
(peek '("a" "b" "c"))	"a"
(pop '("a" "b" "c"))	("b" "c")
(conj '(1 2 3) 4)	(4 1 2 3)
(class '(1))	clojure.lang.PersistentList

Why the Single Quote

'(+ 1 2) verses (+ 1 2)

All Clojure programs are just lists

Reader/interpreter/compiler evaluates all lists

Single quote turns off evaluation of the list

Homoiconicity - Code-as-Data

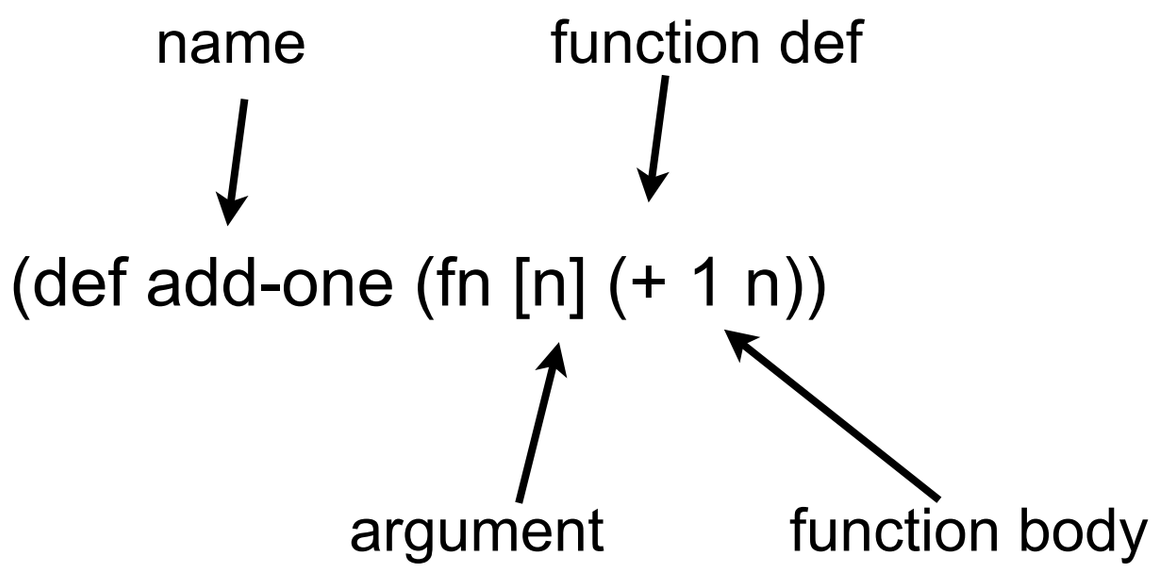
Clojure programs are represented by Clojure data structures

List structure is the Clojure syntax

Makes it easy for Clojure programs to modify Clojure programs

Macros

Defining a function



(add-one 5)

Defining a function - Compact version

```
(def add-one (fn [n] (+ 1 n)))
```

```
(defn add-one
    [n]
    (+ 1 n))
```

```
(add-one 5)
```

Valid function names

Function definitions are just Clojure data structures

Function names are just symbols

So any valid symbol can be used as a function name

```
(defn பன்னிரெண்டு-சேர்க்க
    [n]
    (+ 12 n))
```

Multiple Arguments

```
(defn sum
      [a b c d]
   (+ a b c d))
```

```
(defn foo-bar
      [a b]
   (if (< a b)
            "smaller"
            (+ a b)))
```

defn Format

```
(defn function-name
    "Doc string"
    [arg1 arg2 … argN]
    (form1)
    (form2)
    …
    (formN))
```


We will see later how to add metadata to a function

Doc Strings

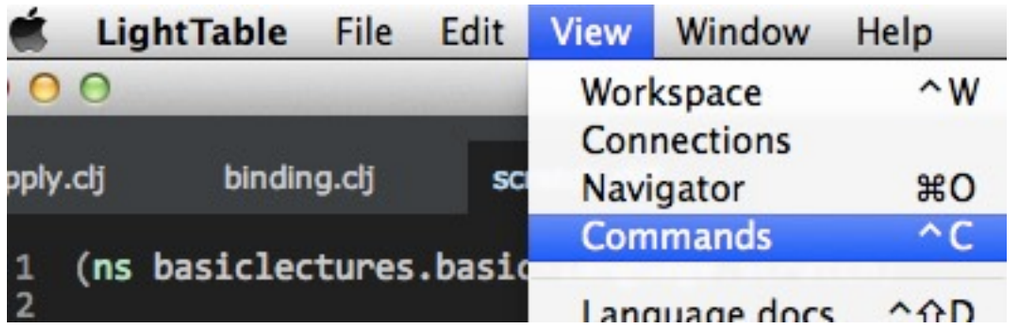
(doc pop)
(clojure.repl/doc pop)

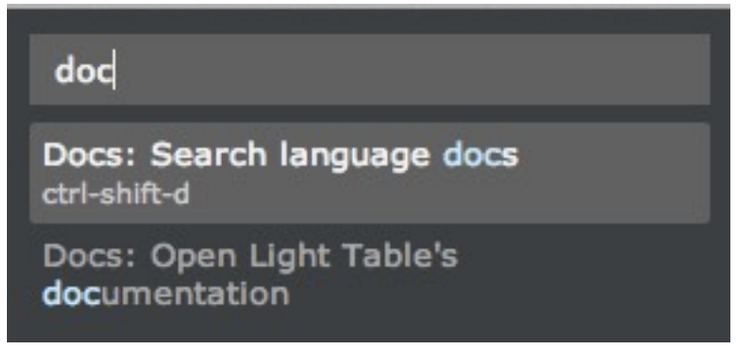
Prints doc string in REPL

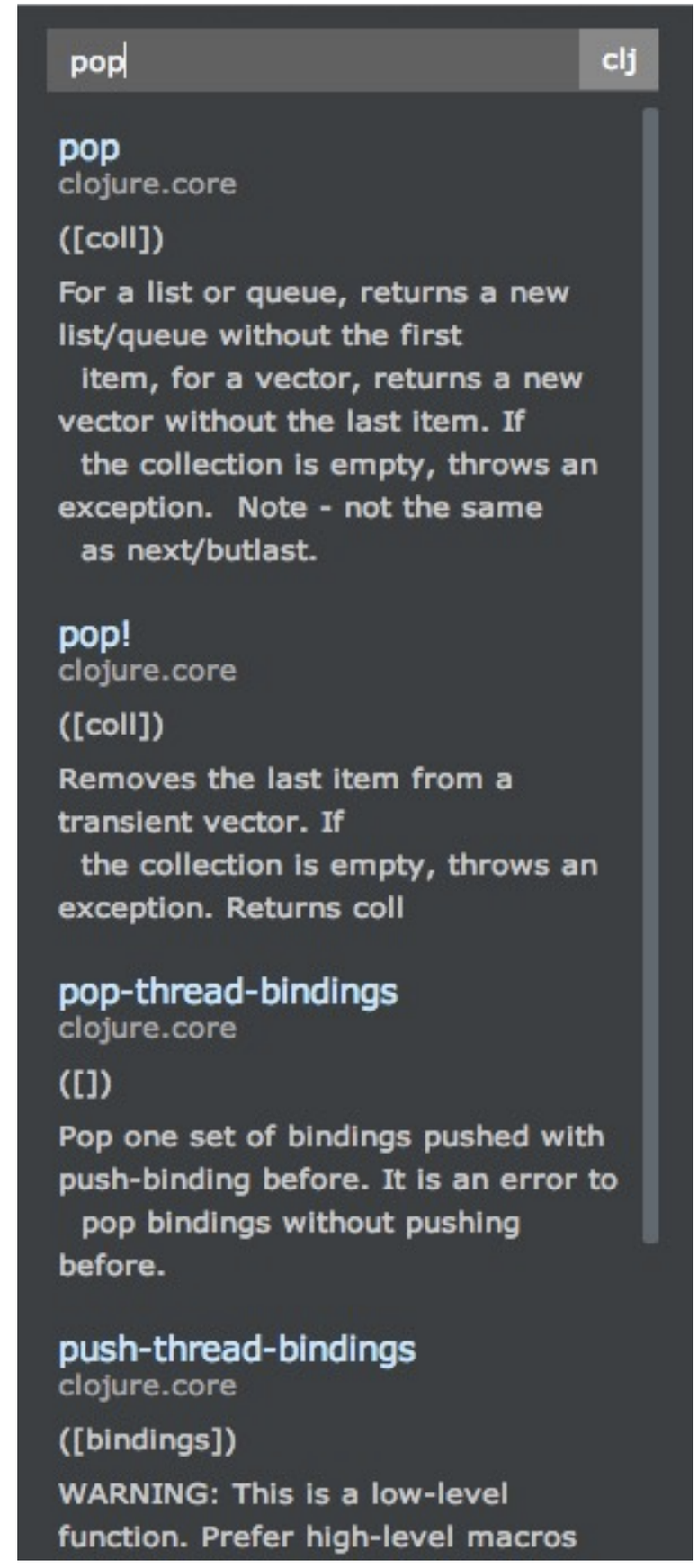
(find-doc "pop")
(clojure.repl/find-doc "pop"

Finds functions related to "pop"

find-doc in Light Table





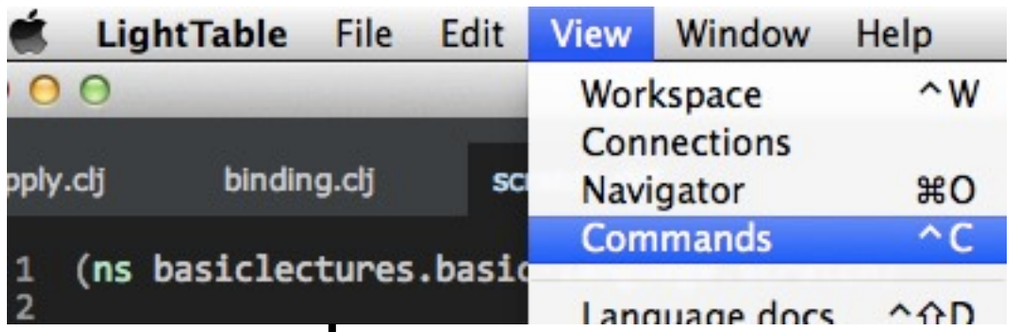


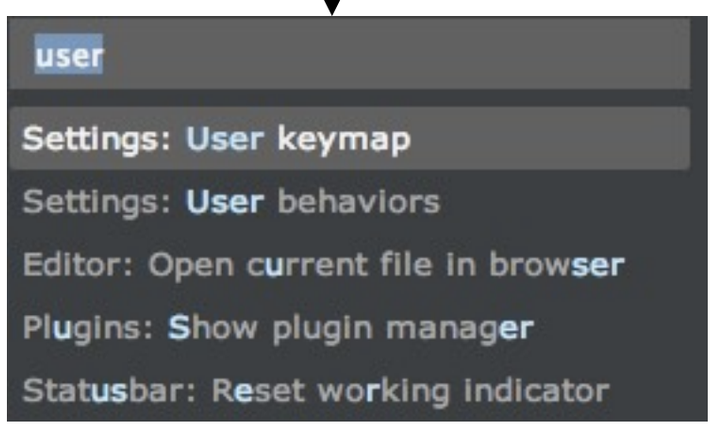
doc in Light Table

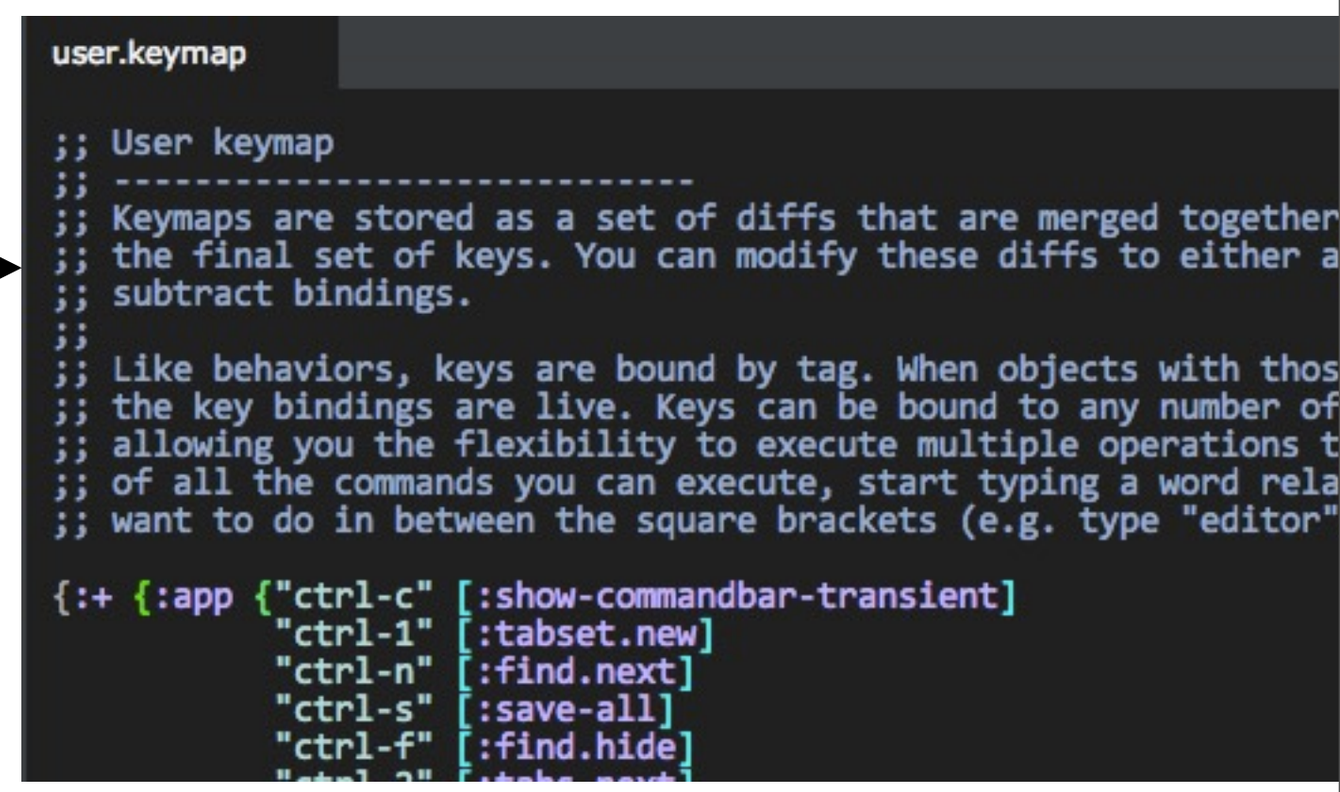
```
3
4 (pop [1234])
```

```
pop
clojure.core

([coll])

For a list or queue, returns a new list/queue without the first
  item, for a vector, returns a new vector without the last item. If
  the collection is empty, throws an exception.  Note - not the same
  as next/butlast.
```

Configuring Light Table







Some Useful keymaps

```
{:+ {:app {"ctrl-c" [:show-commandbar-transient]
           "ctrl-1" [:tabset.new]
           "ctrl-n" [:find.next]
           "ctrl-s" [:save-all]
           "ctrl-f" [:find.hide]
           "ctrl-2" [:tabs.next]
           "ctrl-i" [:instarepl]
           "ctrl-w" [:workspace.show]
           "ctrl-z" [:window.zoom-in]
           "ctrl-shift-z" [:window.zoom-out]
           "ctrl-m"  [:window.maximize]
           "ctrl-t" [:toggle-console]}

       :editor {"ctrl-r" [:clear-inline-results]
                "ctrl-d" [:editor.doc.toggle]
                "ctrl-a" [:paredit.select.parent]
                "ctrl-l" [:paredit.grow.left]
                "ctrl-;" [:paredit.shrink.left]}}}
```

Comments

; a semi-colon starts a comment that goes to end of the line

#_ when prepended to a form makes the entire form a comment

```
(defn foo
  [n]
  #_(if (> 5 n)
    (println "in if")
    (println "else"))
  (+ 10 n))
```

Comment starts

Comment ends

Explain This

```
(defn foo
  [n]
  "How does this work? Not a compile error."
  (if (> 5 n)
    (println "in if")
    (println "else"))
  "This is not a doc comment"
  (+ 10 n))
```

And This?

```
(defn foo
  [n]
  (if (> 5 n)
    "What happens now?"
    (println "in if")
    (println "else"))
  "This is not a doc comment"
  (+ 10 n))
```

Recall

```
(defn function-name
    "Doc string"
    [arg1 arg2 … argN]
    (form1)
    (form2)
    …
    (formN))
```

Anonymous Function - Lambda

Function not bound to symbol

```
(fn [args] (form1) (form2)…(formn))
```

```
(fn [a b] (< (first a) (first b)))
```

```
((fn [a b] (< (first a) (first b))) [2 3] [5])
```

```
((fn [a b]
    (println a  b)
    (< (first a) (first b))) [2 3] [5])
```

Short Syntax for Lambda

```
(fn [a b] (< (first a) (first b)))
```

```
#(< (first %1) (first %2))
```

%n -> n'th argument

```
#(+ 2 %)
```

if only one argument can use %

Passing Functions as Arguments

```
(sort < [3 1 2])

(sort > [3 1 2])

(sort (fn [a b] (< a b))  [3 1 2])

(sort #(< %1 %2)  [3 1 2])

(sort (fn [a b] (compare (str a) (str b))) [ 4 3 16])

(sort  #(compare (str %1)  (str %2))  [4 3 16])
```

Closure

function + reference to its environment

```
(defn adder
  [n]
  #(+ n %))

(def add-5 (adder 5))

(add-5 10)
```

Returns 15