

CS 596 Functional Programming and Design
Fall Semester, 2014
Doc 3 Clojure Data & Form
Sep 4, 2014

Copyright ©, All rights reserved. 2014 SDSU & Roger Whitney, 5500 Campanile Drive, San Diego, CA 92182-7700 USA. OpenContent (<http://www.opencontent.org/openpub/>) license defines the copyright on this document.

Data Structures

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 ఆరు 12)
```

```
(def పన్నీరెండు 12)
```

```
(def बारह 12)
```

```
(def ಹನ್ನೆರಡು 12)
```

```
(def बारो 12)
```

```
(def ਬਾਰ੍ਹਾ 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"

\c

"Another
string"

\u00ff

unicode

(class "cat")
java.lang.String

\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		
36xCRAZY		

BigInt	BigDecimal
12N	4.2M

(factorial 100N)

```
93326215443944152681699238856266700490715  
96826438162146859296389521759999322991560  
89414639761565182862536979208272237582511  
852109168640000000000000000000000000000N
```

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

Vector functions

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

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 or "Not found" arg	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 \longleftrightarrow [1 2 3]

b \longleftrightarrow [1 2 3 4]

c \longleftrightarrow [8 2 3 4]

Java

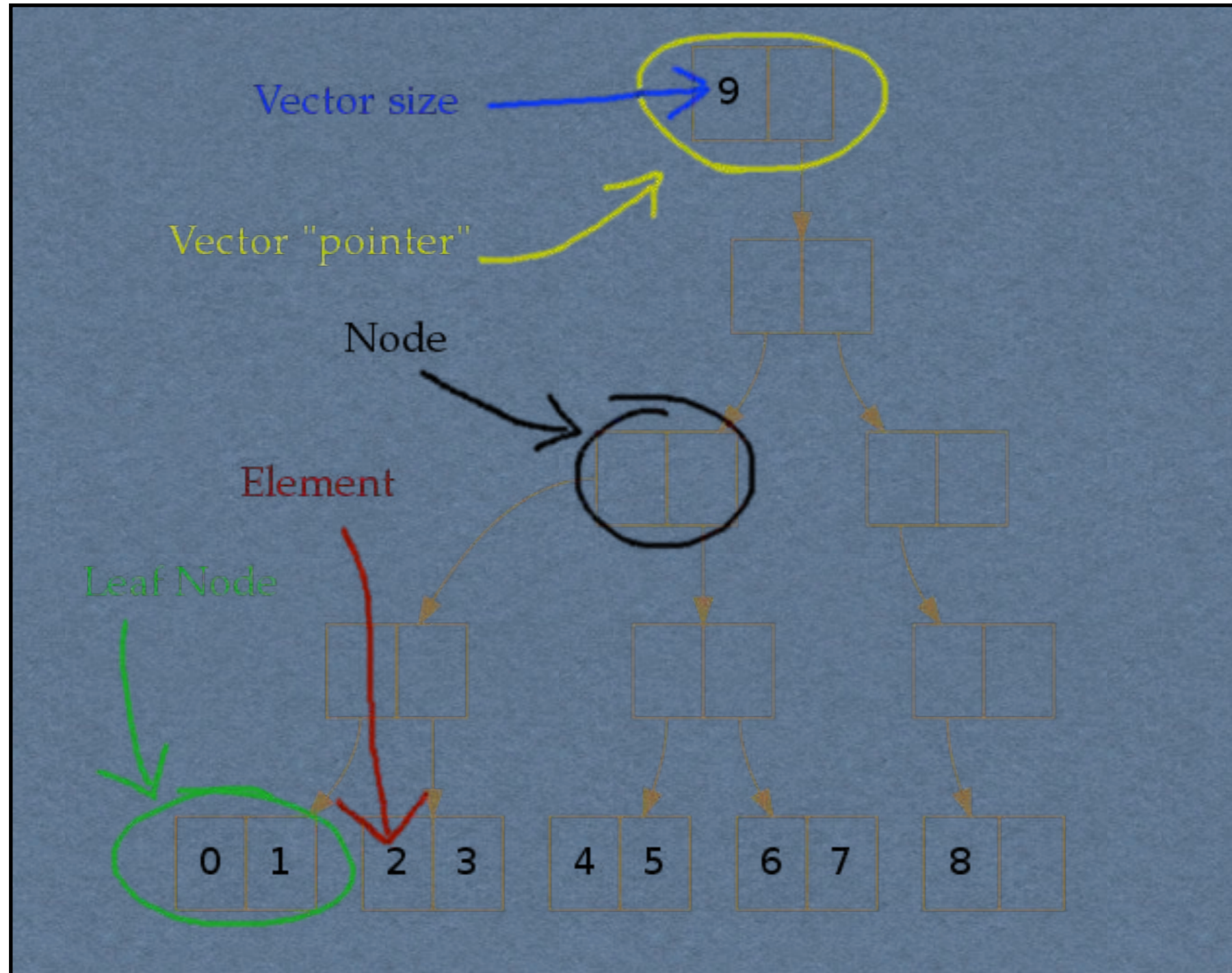
```
int[] d = {1, 2, 3};
```

```
d[0] = 8;
```

d \longleftrightarrow {8, 2, 3}

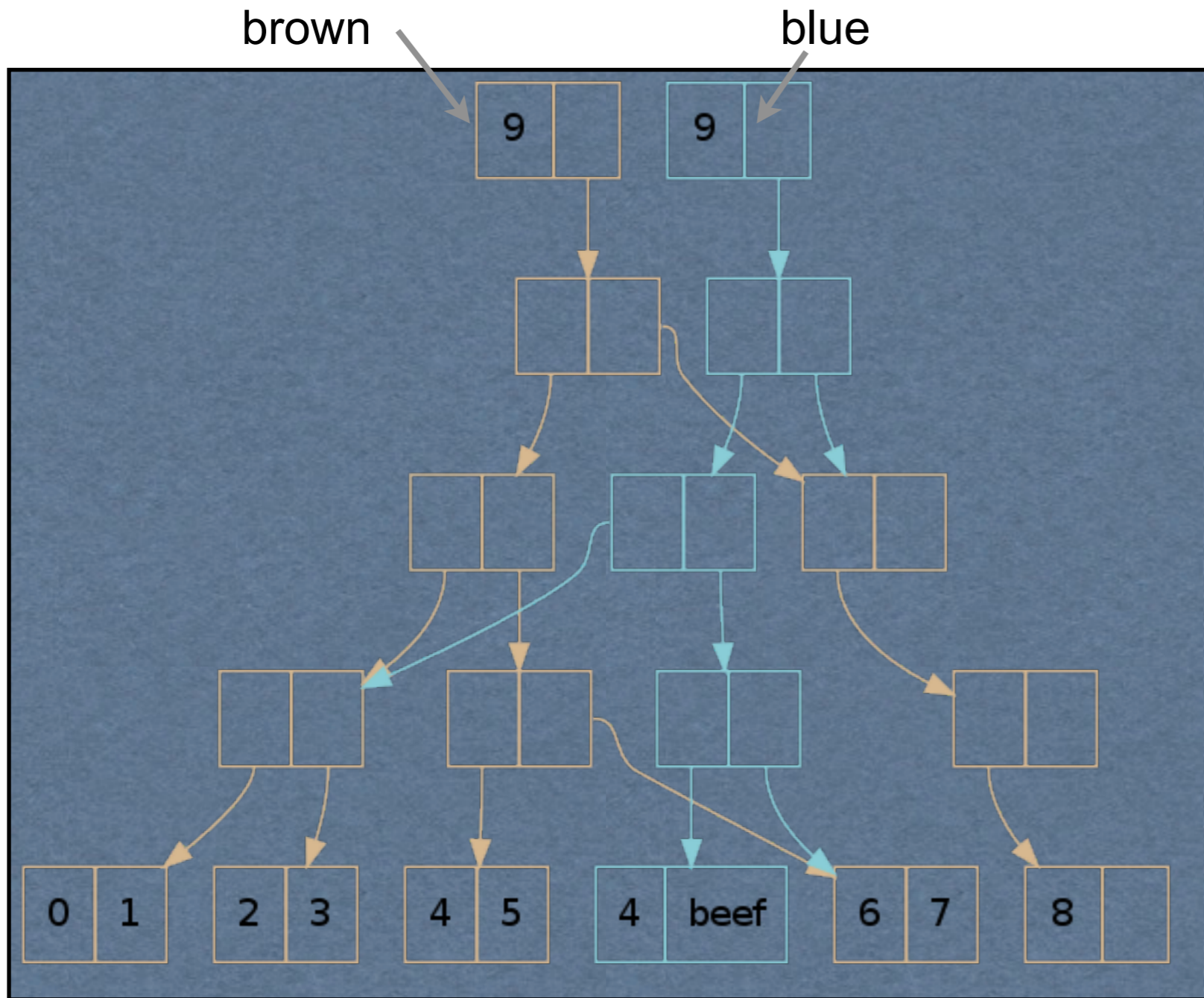
Vector Implementation

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



Update

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

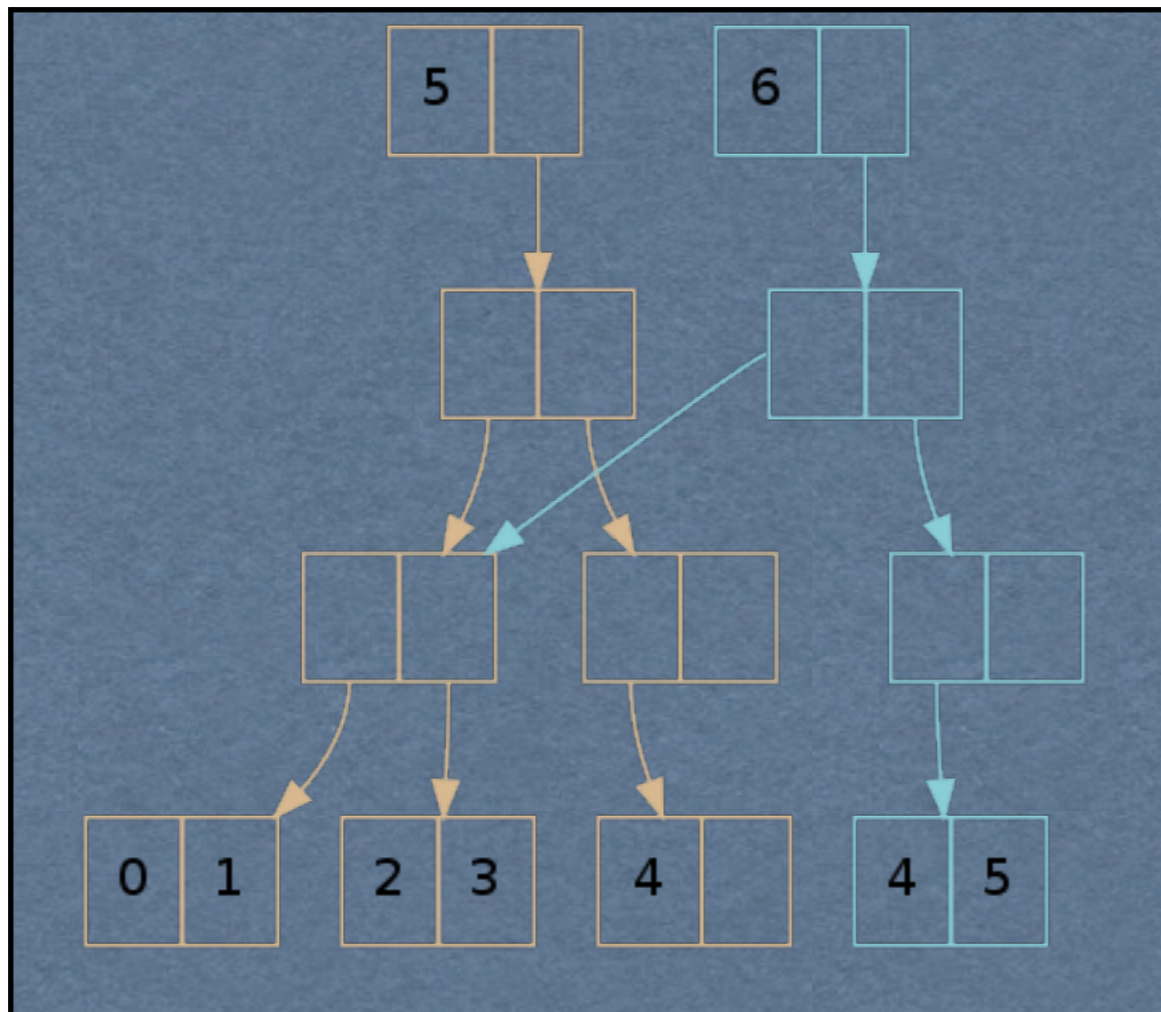


Adding

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

brown

blue



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

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

Maps (Hash Table)

Key-value map

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

Keys - any value

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

Values - any value

Fast insert & find

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

Very common

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

```
{ }
```

Maps (Hash Table)

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

Naming Conventions

Clojure

all-lower-case

words-separated-by-hyphen

Java

camelCase

Lists

Linked List

'(1 2 3)

Fast insert & remove at front

'("cat" {:a 1})

'(+ 1 2)

Lists

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

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

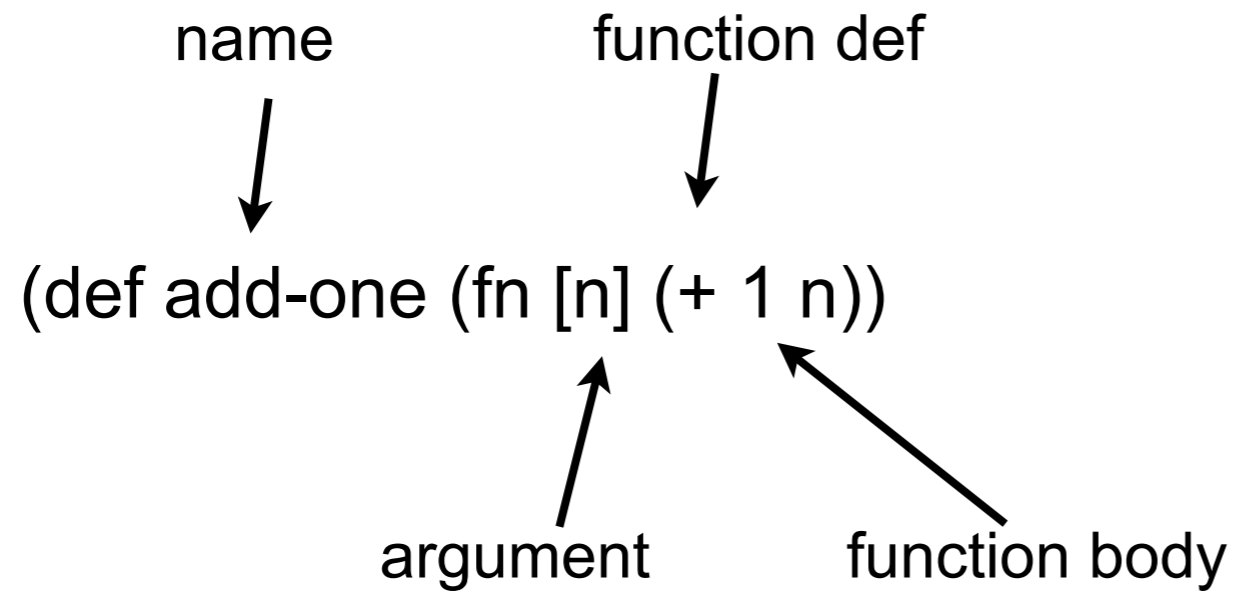
Clojure programs are represented by Clojure data structures

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 பன்னிரண்டு-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))
```

Doc Strings

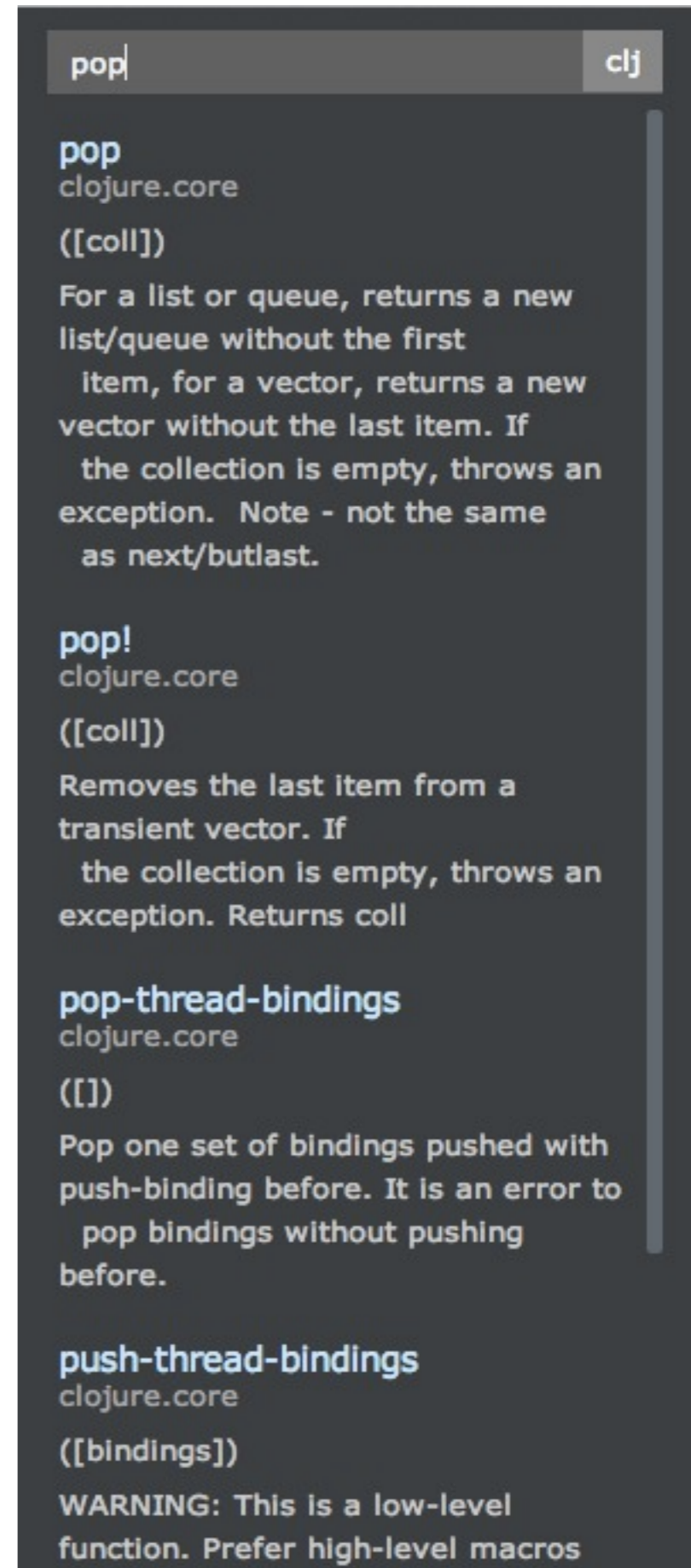
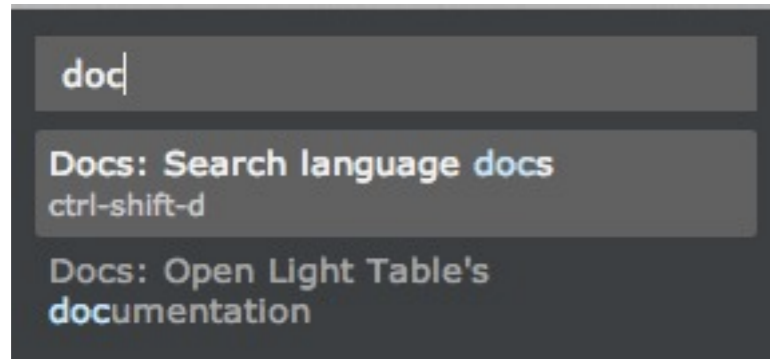
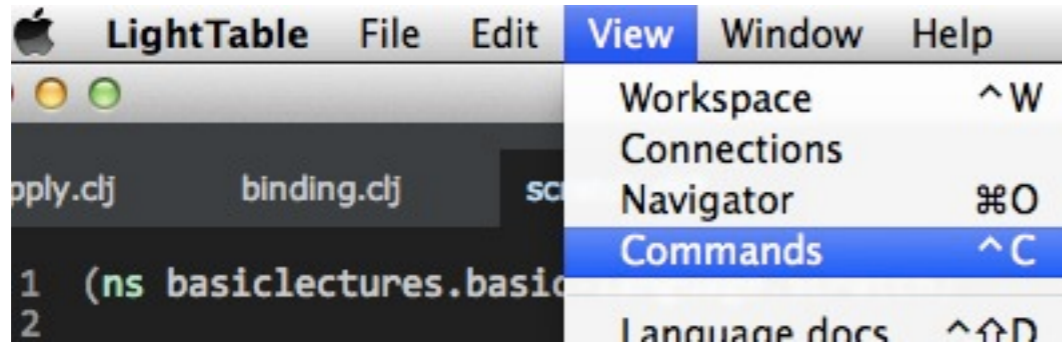
```
(doc pop)  
(clojure.repl/doc pop)
```

Prints doc string in REPL

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

Finds functions related to "pop"

find-doc in Light Table



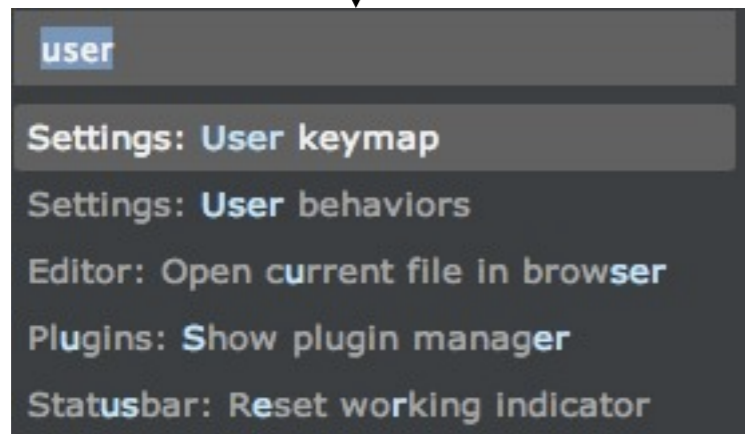
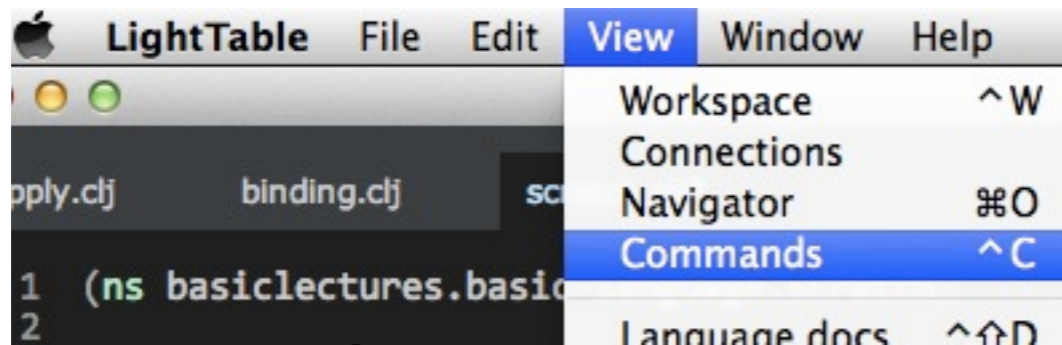
doc in Light Table

```
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



```
user.keymap

;; User keymap
;; -----
;; Keymaps are stored as a set of diffs that are merged together
;; the final set of keys. You can modify these diffs to either a
;; subtract bindings.
;;
;; Like behaviors, keys are bound by tag. When objects with those
;; the key bindings are live. Keys can be bound to any number of
;; allowing you the flexibility to execute multiple operations
;; of all the commands you can execute, start typing a word related
;; want to do in between the square brackets (e.g. type "editor")

{:+ [:app {"ctrl-c" [:show-commandbar-transient]
           "ctrl-1" [:tabset.new]
           "ctrl-n" [:find.next]
           "ctrl-s" [:save-all]
           "ctrl-f" [:find.hide]
           "ctrl-?" [:tabset.next]}
```


Some Useful keymaps

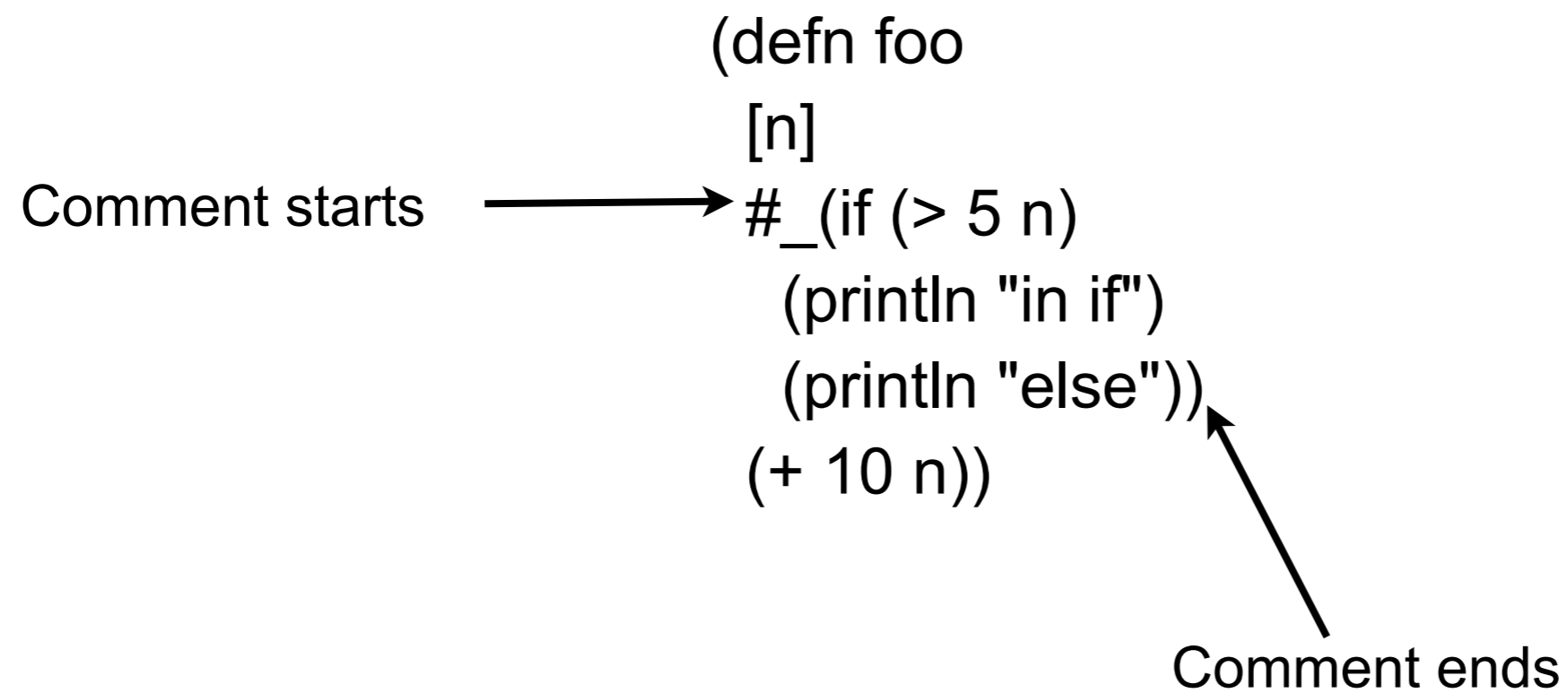
```
{:+ {:app {"ctrl-c" [:show-commandbar-transient]
  "ctrl-1" [:tabset.new]
  "ctrl-n" [:find.next]
  "ctrl-s" [:save-all]
  "ctrl-f" [:find.hide]
  "ctrl-2" [:tabs.next]
  "ctrl-i" [:instarepl]
  "ctrl-w" [:workspace.show]
  "ctrl-z" [:window.zoom-in]
  "ctrl-shift-z" [:window.zoom-out]
  "ctrl-m"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



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))
```

Clojure Form

Clojure expression

symbols

keywords

literals

lists

vectors

maps

sets

```
(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))
```

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