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 alphanumerick characters and ! $ % & * - + = _ | < > ?

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

Unicode is supported

(def పన్ని ఎరణాంతు 12)  (def ବାର୍ହ 12)  (def പന്നിക്കുലൂര്‍ഷം 12)
(def ବାରହ 12)  (def കൊല്ലം 12)
(def ବାର୍ଶ 12)  (def ଘনൂർ 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")

\c
\u00ff
\o64

java.lang.String

unicode
octal
Whitespace Characters

\space
\newline
\formfeed
\return
\backspace
\tab

So what is \n?
\n verses "\n"

\n "\n"

Character n newbie in string

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

(str "a\nnewline "b") "a
  b"

(str "a" "\n" "b") "a
  b"
## Numbers

<table>
<thead>
<tr>
<th>Long</th>
<th>Double</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>2.11</td>
<td>3/5</td>
</tr>
<tr>
<td>0xfe</td>
<td>1.3e-4</td>
<td></td>
</tr>
<tr>
<td>2r111</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5r123</td>
<td></td>
<td></td>
</tr>
<tr>
<td>36xCRAZY</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BigInt</th>
<th>BigDecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>12N</td>
<td>4.2M</td>
</tr>
</tbody>
</table>

(factorial 100N)

```
93326215443944152681699238856266700490715
96826438162146859296389521759999322991560
89414639761565182862536979208272237582511
85210916864000000000000000000000000N
```
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
bigint
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 loopup
Vector functions

(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

<table>
<thead>
<tr>
<th></th>
<th>nth</th>
<th>get</th>
<th>Vector as function</th>
</tr>
</thead>
<tbody>
<tr>
<td>nil vector</td>
<td>Returns nil</td>
<td>Returns nil</td>
<td>Exception</td>
</tr>
<tr>
<td>Index out of range</td>
<td>Exception or &quot;Not found&quot; arg</td>
<td>Returns nil or &quot;Not found&quot; arg</td>
<td>Exception</td>
</tr>
<tr>
<td>Not found arg</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>
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;

{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

19

(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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>O(1)</td>
</tr>
<tr>
<td>first</td>
<td>O(&lt;7)</td>
</tr>
<tr>
<td>rest</td>
<td>O(&lt;7)</td>
</tr>
<tr>
<td>nth</td>
<td>O(&lt;7)</td>
</tr>
<tr>
<td>last</td>
<td>O(n)</td>
</tr>
<tr>
<td>get</td>
<td>O(&lt;7)</td>
</tr>
<tr>
<td>assoc</td>
<td>O(&lt;7)</td>
</tr>
<tr>
<td>peek</td>
<td>O(1) ?</td>
</tr>
</tbody>
</table>
For More Details

See

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

{: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 {"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          camelCase
words-separated-by-hyphen
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

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

(name function def)

(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 பன்னிெரண்டு-ேசர்க்க
  [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)))
We will see later how to add metadata to a function
Doc Strings

(doc pop)  Prints doc string in REPL
(clojure.repl/doc pop)

(find-doc "pop")  Finds functions related to "pop"
(clojure.repl/find-doc "pop"
find-doc in Light Table

```
(ns basiclectures.basic)

(apply.clj binding.clj)
```

---

**pop**
clojure.core

```clojure
[[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. 

**pop!**
clojure.core

```clojure
[[coll]]
```

Removes the last item from a transient vector. If the collection is empty, throws an exception. Returns coll.

**pop-thread-bindings**
clojure.core

```clojure
([])
```

Pop one set of bindings pushed with push-binding before. It is an error to pop bindings without pushing before.

**push-thread-bindings**
clojure.core

```clojure
([bindings])
```

WARNING: This is a low-level function. Prefer high-level macros.
(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

```plaintext
user

Settings: User keymap
Settings: User behaviors
Editor: Open current file in browser
Plugins: Show plugin manager
Status bar: Reset working indicator

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 add
;; 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 to
;; 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-q" [:meta.next]}}
```
Some Useful keymaps


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))
(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