Basic Data Elements

symbols
general
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 పల్పేయ్యి రశ్మి 12)  (def బార్ 12)  (def మలాయితిగి రశ్మి శిల్పం 12)
(def బారహ 12)  (def కలుగచి రశ్మి 12)
(def బార్రో రశ్మి 12)  (def వారు రశ్మి 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

(class "cat")                        unicode

\o64

*java.lang.String*                    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

<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>36rCRAZY</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
8521091686400000000000000000000000N
```
Cast/Convert

<table>
<thead>
<tr>
<th>Type</th>
<th>Example</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>(long 12.8)</td>
<td>12</td>
</tr>
<tr>
<td>short</td>
<td>(rationalize 0.25)</td>
<td>1/4</td>
</tr>
<tr>
<td>int</td>
<td>(read-string &quot;12.6&quot;)</td>
<td>12.6</td>
</tr>
<tr>
<td>long</td>
<td>(str 12.3)</td>
<td>&quot;12.3&quot;</td>
</tr>
<tr>
<td>float</td>
<td></td>
<td></td>
</tr>
<tr>
<td>double</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bigdec</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bigint</td>
<td></td>
<td></td>
</tr>
<tr>
<td>num</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rationalize</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bigint</td>
<td></td>
<td></td>
</tr>
<tr>
<td>integer</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Collections

Immutable
Heterogeneous
Persistent

Vectors
Sets
Maps
Lists
Queues
Vectors

Expandable, indexed list

Fast insert at end

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

<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</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))

Java

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

a ← [1 2 3]  d ← {8, 2, 3}
b ← [1 2 3 4]  
c ← [8 2 3 4]
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))
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

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

Keys - any value

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

Values - any value

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

Fast insert & find

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

Very common

{ "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

<table>
<thead>
<tr>
<th>Clojure</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>all-lower-case</td>
<td>camelCase</td>
</tr>
<tr>
<td>words-separated-by-hyphen</td>
<td></td>
</tr>
</tbody>
</table>
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))

(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 அணுக்குறையான குழுக்குறையான [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 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) 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

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.

pop!
clojure.core
([coll])
Removes the last item from a
transient vector. If
the collection is empty, throws an
exception. Returns coll

pop-thread-bindings
clojure.core
([])
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
([[bindings]])
WARNING: This is a low-level
function. Prefer high-level macros
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 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-0" [:tabset.neighborhood]}}
```
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)))
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))
(defn foo
  [n]
  (if (> 5 n)
    "What happens now?"
    (println "in if")
    (println "else")))
"This is not a doc comment"
(+ 10 n))
(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))  \(\text{\%n} \rightarrow \text{n'th argument}\)

#+( 2 %)  \(\text{if only one argument can use } \%\)
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