Clojure

Developed by Rich Hickey

Started 2007

Variant of Lisp

Functional programming language

Dynamic typing

Interactive development - REPL

Tight Java Integration

Active development community
Variants

Clojure  ClojureScript  ClojureCLR

Java  JavaScript  .NET

Base language the same

Few changes due differences between Java/JavaScript/.NET
Development Environment

Light Table
 Clojure/Web IDE
 http://lighttable.com/

IntelliJ
 Cursive plugin
 https://cursiveclojure.com

Eclipse
 Counterclockwise plugin
 https://code.google.com/p/counterclockwise/

Command Line

Leiningen

Night Code

Emacs
 CIDER

Vim
 Fireplace
Light Table

http://www.lighttable.com

Recommended IDE to start learning Clojure

```clojure
;; Anything you type in here will be executed immediately with the results shown on the right.
```
Lots of Irritating Superfluous Parenthesis-LISP

Actually not more that Java's

But only () and they build up
  (+ 5 (- 2 (/ 4 (* 2 (inc (read-string "123"))))))

Use editor that is parenthesis aware

Useful forms
  let
  ->
Resources

Clojure Home Page

http://clojure.org

Clojure Cookbook

Safari Books On-line
http://proquest.safaribooksonline.com.libproxy.sdsu.edu/
Elements of Clojure Code

symbols  functions
keywords  macros
literals  special forms (functions)
lists
vectors
maps
sets
REPL

Read-Eval-Print Loop
Light Table - front end to Clojure REPL

Executable code (program) in repl

"hi there"

42

[1 2 3]

(+1 2)
Clojure Programs

Chain of functions calling functions

```
(defn factorial [n]
  (if (= n 1)
    (bigint 1)
    (* n (factorial (- n 1)))))
```
Clojure Function Calls

foo(1, "cat")  

C function call

(foo 1 "cat")  

Clojure function call

Function Name  Arguments
## Some Basic Operations

<table>
<thead>
<tr>
<th>Function</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>(+ 1 2)</td>
<td>3</td>
</tr>
<tr>
<td>(+ 1 2 4 6)</td>
<td>13</td>
</tr>
<tr>
<td>(= &quot;cat&quot; &quot;dog&quot;)</td>
<td>false</td>
</tr>
<tr>
<td>(= 1 1)</td>
<td>true</td>
</tr>
<tr>
<td>(= 1 1 2)</td>
<td>false</td>
</tr>
<tr>
<td>(even? 8)</td>
<td>true</td>
</tr>
<tr>
<td>(/ 10 2)</td>
<td>5</td>
</tr>
<tr>
<td>(/ 10 2 3)</td>
<td>5/3</td>
</tr>
<tr>
<td>(bit-shift-left 4 1)</td>
<td>8</td>
</tr>
</tbody>
</table>
Operators

No built-in operators

Just functions

(if (> x y)
   "cat"
   "dog")

Condition
true value
false value
Assignment

No built-in operators

Just functions

(def a 10)

(def b (+ a 12))

(def a 20)

Called a binding which is sort of like assignment
No Precedence

\[ a - b \times c + d \]

\[ (- a (+ (* b c) d)) \]

Clojure expressions read inside out

Will see several ways to change this
Recursion
Higher Order Functions
The Functional Way
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

(first [1 2 3]) 1

(second [1 2 3]) 2

(third [1 2 3]) Error

(last [1 2 3]) 3

(rest [1 2 3]) (2 3)
Compute the Sum

```java
public float sum(ArrayList<float> list) {
    float sum = 0;
    for (int k = 0; k < list.length; k++)
        sum = sum + list.get(k);
    return sum;
}
```

Does not work in Functional World

No "for" statement

No side effects
Recursion replaces Iteration

(defn sum-1
    [list]
    (if (empty? list)
        0
        (+ (first list) (sum-1 (rest list)))))

(sum-1 [1 2 3]) 6

(sum-1 (range 9900)) Stack overflow

(range 9900) [1 2 3 4 5 ... 9898 9899]
(defn sum-2
  [partial-sum list]
  (if (empty? list)
    partial-sum
    (sum-2 (+ partial-sum (first list))
      (rest list))))

(sum-2 0 [1 2 3]) 6

(sum-2 0 (range 9900)) Stack over flow
Recursive verses Iterative Process

Recursive Process

\[
\text{(sum-1 [1 2 3])}
\]

\[
\text{(+ 1 (sum-1 [2 3]))}
\]

\[
\text{(+ 1 (+ 2 (sum-1 [3])))}
\]

\[
\text{(+ 1 (+ 2 (+ 3 (sum-1 [])))}}
\]

\[
\text{(+ 1 (+ 2 (+ 3 0)))}
\]

\[
\text{(+ 1 (+ 2 3))}
\]

\[
\text{(+ 1 5)}
\]

6

Iterative Process

\[
\text{(sum-2 0 [1 2 3])}
\]

\[
\text{(sum-2 1 [2 3])}
\]

\[
\text{(sum-2 3 [3])}
\]

\[
\text{(sum-2 6 (sum-2 []))}
\]

6
Tail Recursion Optimization

In a recursive function implementing an iterative process

The compiler can optimize the recursion into iteration

But JVM does not support tail recursion optimization
(defn sum-3
  [accumulator list]
  (if (empty? list)
      accumulator
      (recur (+ accumulator (first list))
             (rest list))))

(recur will call the function

But Clojure will convert to iteration

(sum-3 0 [1 2 3]) 6
(sum-3 0 (range 9900)) 49000050
(sum-3 0 (range 100000)) 4999950000
One Name, Multiple Implementations

(defn sum-4
  ([list]
   (sum-4 0 list))
  ([accumulator list]
   (if (empty? list)
     accumulator
     (recur (+ accumulator (first list))
       (rest list)))))

(sum-4 [1 2 3]) 6
(sum-4 0 [1 2 3]) 6
(sum-4 (range 100000)) 4999950000
(sum-4 0 (range 100000)) 4999950000
Major Points

Recursion replaces “for” loops

Accumulators can be used to convert recursive process into iterative process

Tail recursion optimization (recur) can convert iterative process to iterative code

But this is not the way to implement sum
reduce

(reduce + [1 2 3 4 5])
What versus How

<table>
<thead>
<tr>
<th>What</th>
<th>How</th>
</tr>
</thead>
<tbody>
<tr>
<td>(reduce + [1 2 3 4 5])</td>
<td>public float sum(ArrayList&lt;float&gt; list) {</td>
</tr>
<tr>
<td></td>
<td>float sum = 0;</td>
</tr>
<tr>
<td></td>
<td>for (int k = 0; k &lt; list.length; k++)</td>
</tr>
<tr>
<td></td>
<td>sum = sum + list.get(k);</td>
</tr>
<tr>
<td></td>
<td>return sum;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>

Less typing

Fewer details

Less cognitive load

More general solution

Code can be optimized
Higher Order Functions

Function that acts on functions

(reduce + [1 2 3 4 5])
Timing tests

<table>
<thead>
<tr>
<th>Code</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>(sum-3 0 (range 100000))</td>
<td>54450.6 msecs</td>
</tr>
<tr>
<td>(sum-4 0 (range 100000))</td>
<td>26.1 msecs</td>
</tr>
<tr>
<td>(reduce + (range 100000))</td>
<td>6.5 msecs</td>
</tr>
</tbody>
</table>

(def data (range 1000000))

<table>
<thead>
<tr>
<th>Code</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>(sum-4 data)</td>
<td>~55 msecs</td>
</tr>
<tr>
<td>(reduce + data)</td>
<td>~22.5 msecs</td>
</tr>
</tbody>
</table>
The Functional Way

Raw data

- vectors
- maps (hash table)
- sequences

Rich set of powerful functions on data

- map
- map-indexed
- filter
- reduce
- remove
- keep
- zipper
- drop-while
- take-while
- partition
- interpose
- split-at
- etc.
Immediate Goals

Recursion

Master use of built-in functions

Get comfortable with higher-order functions.
# Clojure API

http://clojure.org/cheatsheet

---

Clojure 1.3-1.6 Cheat Sheet (v13)

Download PDF version, Download other versions with tooltips

## Documentation

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

## Primitives

### Numbers

<table>
<thead>
<tr>
<th>Literals</th>
<th>Long: 7, hex 0xff, oct 017, base 2 2r1011, base 3 36rCRAZY BigInt: 7N Ratio: -22/7 Double: 2.78 -1.2e-5 BigDecimal: 4.2M</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>+ - * / quot rem mod inc dec max min</td>
</tr>
<tr>
<td>Compare</td>
<td>= == not= &lt; &gt; &lt;= &gt;= compare</td>
</tr>
<tr>
<td>Bitwise</td>
<td>bit-{and, or, xor, not, flip, set, shift-right, shift-left, and-not, clear, test} (1.6) unsigned-bit- shift-right</td>
</tr>
<tr>
<td>Cast</td>
<td>byte short int long float double bigdec bigint num rationalize</td>
</tr>
</tbody>
</table>

## 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? instance? nil? (1.6) some?                                                                                      |

## Sequences

Creating a Lazy Seq

| From collection   | seq vals keys rseq subseq rsubseq                                                                                           |
| From producer fn  | lazy-seq repeatedly iterate                                                                                                 |
Intro to Strings

Difficulty: Elementary
Topics:

Clojure strings are Java strings. This means that you can use any of the Java string methods on Clojure strings.

```clojure
(= __ (.toUpperCase "hello world"))
```

Code which fills in the blank: