Seven Languages in Seven Weeks

Correl Roush

August 12, 2015

Created 2007

Author Rich Hickey

A general-purpose, functional Lisp on the Java Virtual Machine.

http://clojure.org

```
(- 1)
;; -1
(* 10 10)
;; 100
(/ 2 4)
;; 1/2
(/ 2.0 4)
;; 0.5
(/ (/ 12 2) (/ 6 2))
;; 2
(/ 8 2 2)
;; 2
(< 1 2 3)
;; true
```

Strings & Characters

```
(println
 "master yoda\nluke skywalker")
;; master yoda
;; luke skywalker
(str "yoda, " "luke,  " "darth")
;; "yoda, luke, darth"
(str \f \o \r \c \e)
;; "force"
```

Booleans & Expressions

```
(= 1 1.0)
;; true
(= 1 2)
;; false
(if nil (println "true") (println "false"))
;; false
;; nil
```

Lists & Vectors

```
(list 1 2 3)
;; (1 2 3)
(cons :battle-droid '(:r2d2 :c3po))
;; (:battle-droid :r2d2 :c3po)

([:hutt :wookie :ewok] 2)
;; :ewok
(concat [:darth-vader] [:darth-maul])
;; (:darth-vader :darth-maul)
```

Sets & Maps

```
(def spacecraft
  #{:x-wing :y-wing :tie-fighter})
(spacecraft :x-wing)
;; :x-wing
(spacecraft :a-wing)
;; nil

(def mentors {:darth-vader "obi wan", :luke "yo
(mentors :luke)
;; "yoda"
```

```clojure
(defn force-it
  "The first function a young Jedi needs"
  [jedi]
  (str "Use the force, " jedi))

(force-it "Luke")
;; "Use the force, Luke"

(doc force-it)
;; -------------------------
;; user/force-it
;; ([jedi])
;;   The first function a young Jedi needs
```

```clojure
(def board [[:x :o :x] [:o :x :o] [:o :x :o]])
(defn center [[_ [_ c _] _]] c)
(center board)
;; :x

(defn center [board]
  (let [[_ [_ c]] board] c))

(def villains [{:name "Godzilla" :size "big"} {:name "Ebola" :size "small"}])
(let [[_ {name :name}] villains] (str "Name of the second villain: " name))
;; "Name of the second villain: Ebola"
```

Anonymous functions can be defined with the `(fn [args] ...)` form, or the shorthand `#(... % ...)`, where each `%` is replaced with one the provided arguments.

```
(def people ["Lea", "Han Solo"])

(map (fn [w] (* 2 (count w))) people)
;; (6 16)

(map #(* 2 (count %)) people)
;; (6 16)
```

Apply and Filter

```
(def v [3 1 2])

(apply + v)
;; 6
(apply max v)
;; 3

(filter odd? v)
;; (3 1)
(filter #(< % 3) v)
;; (1 2)
```

Clojure is designed to be a practical tool for general-purpose production programming by developers in industry and as such adds these additional objectives to the Lisps of old. We work better in teams, we play well with other languages, and we solve some traditional Lisp problems.

Simple Recursion

```
(defn size [v]
  (if (empty? v)
    0
    (inc (size (rest v)))))
(size [1 2 3])
;; 3
```

Using Loop & Recur

```
(loop [x x-initial-value,
       y y-initial-value]
  (do-something-with x y))

(loop [x 1] x)
;; 1

(defn size [v]
  (loop [l v, c 0]
    (if (empty? l)
      c
      (recur (rest l) (inc c)))))
```

A sequence is an implementation-independent abstraction around all the various containers in the Clojure ecosystem. Sequences wrap all Clojure collections (sets, maps, vectors, and the like), strings, and even file system structures (streams, directories). They also provide a common abstraction for Java containers, including Java collections, arrays, and strings. In general, if it supports the functions `first`, `rest`, and `cons`, you can wrap it in a sequence.

```
(def colors ["red" "blue"])
(def toys ["block" "car"])

(for [x colors] (str "I like " x))
;; ("I like red" "I like blue")

(for [x colors, y toys] (str "I like " x " " y "s"))
;; ("I like red blocks" "I like red cars"
;;  "I like blue blocks" "I like blue cars")

(defn small-word? [w] (< (count w) 4))
(for [x colors, y toys, :when (small-word? y)]
  (str "I like " x " " y "s"))
;; ("I like red cars" "I like blue cars")
```

Clojure's sequence library computes values only when they are actually consumed.

```
(range 1 10)
;; (1 2 3 4 5 6 7 8 9)
(range 1 10 3)
;; (1 4 7)
(range 10)
;; (0 1 2 3 4 5 6 7 8 9)
```

```clojure
(take 3 (repeat "Use the Force, Luke"))
;; ("Use the Force, Luke" "Use the Force, Luke" "Use the Force, Luke")
(take 5 (cycle [:lather :rinse :repeat]))
;; (:lather :rinse :repeat :lather :rinse)

(->> [:lather :rinse :repeat] (cycle) (drop 2) (take 5))
;; (:repeat :lather :rinse :repeat :lather)

(defn factorial [n] (apply * (take n (iterate inc 1))))
(factorial 5)
;; 120
```

Protocols define a contract, similar to a Java interface. Types of a protocol will support a specific set of functions, fields, and arguments.

Records define an type, similar to a Java class, that can implement a protocol.

```
(defprotocol Compass
  (direction [c])
  (left [c])
  (right [c]))

(def directions [:north :east :south :west])

(defn turn
  [base amount]
  (rem (+ base amount) (count directions)))

(defrecord SimpleCompass [bearing]
  Compass
  (direction [_] (directions bearing))
  (left [_] (SimpleCompass. (turn bearing 3)))
  (right [_] (SimpleCompass. (turn bearing 1)))
  Object
  (toString [this] (str "[" (direction this) "]")))
```

Defining unless as a function

```
(defn unless [test body]
  (if (not test) body))

(unless true (println "Danger, danger Will Robinson"))
;; Danger, danger Will Robinson
;; nil
```

Defining unless as a macro

```
(defmacro unless [test body]
  (list 'if (list 'not test) 'body))

(macroexpand '(unless condition body))
;; (if (not condition) body)

(unless true (println "No more danger, Will."))
;; nil
```

Modern databases use at least two types of concurrency control:

Locks *prevent two competing transactions from accessing the same row at the same time.*

Versioning *uses multiple versions to allow each transaction to have a private copy of its data. If any transaction interferes with another, the database engine simply reruns that transaction.*

- Languages like Java use locking to protect the resources of one thread from competing threads that might corrupt them. Locking basically puts the burden of concurrency control on the programmer. We are rapidly learning that this burden is too much to bear.
- Languages like Clojure use software transactional memory (STM). This strategy uses multiple versions to maintain consistency and integrity.

- A ref (reference) is a wrapped piece of data.
- All access must conform to specified rules to support STM.
- You cannot change a reference outside of a transaction.

```
(def movie (ref "Star Wars"))

(deref movie)
;; "Star Wars"
@movie
;; "Star Wars"

(alter movie str ": The Empire Strikes Back")
;; java.lang.IllegalStateException: No transaction running (NO_SOURCE_FILE:0)

(dosync (alter movie str ": The Empire Strikes Back"))
;; "Star Wars: The Empire Strikes Back"

(dosync (ref-set movie "Star Wars: The Revenge of the Sith"))
;; "Star Wars: The Revenge of the Sith"

@movie
;; "Star Wars: The Revenge of the Sith"
```

- An atom is a wrapped piece of data.
- Allows change outside the context of a transaction.
- Useful to provide thread-safety for a single reference.

```
(def danger (atom "Split at your own risk."))

@danger
;; "Split at your own risk."

(reset! danger "Split with impunity")
;; "Split with impunity"

@danger
;; "Split with impunity"

(def top-sellers (atom []))
(swap! top-sellers conj {:title "Seven Languages", :author "Tate"})
;; [{:title "Seven Languages", :author "Tate"}]
(swap! top-sellers conj {:title "Programming Clojure", :author "Halloway"})
;; [{:title "Seven Languages", :author "Tate"}
;;  {:title "Programming Clojure", :author "Halloway"}]
```

```clojure
(ns solutions.atom-cache
  (:refer-clojure :exclude [get]))

(defn create
  []
  (atom {}))

(defn get
  [cache key]
  (@cache key))

(defn put
  ([cache value-map]
     (swap! cache merge value-map))
  ([cache key value]
     (swap! cache assoc key value)))
(def ac (create))
(put ac :quote "I'm your father, Luke.")
(println (str "Cached item: " (get ac :quote)))
```

- Like an atom, an agent is a wrapped piece of data.
- Like a future, the state of a dereferenced agent will block until a value is available.
- Users can mutate the data asynchronously using functions, and the updates will occur in another thread.
- Only one function can mutate the state of an agent at a time.

```
(def tribbles (agent 1))

(defn twice [x] (* 2 x))
(send tribbles twice)
@tribbles
;; 2

(defn slow-twice [x]
  (do
    (Thread/sleep 5000)
    (* 2 x)))
(send tribbles slow-twice)
@tribbles
;; 2

;; -*- 5 seconds later -*-
@tribbles
;; 4
```

- You will get a value of `tribbles`. You may not get the latest changes from your own thread.
- If you want to be sure to get the latest value with respect to your own thread, you can call `(await tribbles)` or `(await-for timeout tribbles)`.

Clojure's tools involve working with a snapshot whose value is instantaneous and potentially out-of-date immediately. That's exactly how versioning databases work for fast concurrency control.

A future is a concurrency construct that allows an asynchronous return before computation is complete.
Creating a future returns a reference immediately, starting the computation in another thread. Dereferencing the reference blocks until the computation completes.

```
(def finer-things (future (Thread/sleep 5000) "take time"))

@finer-things
;; -*- After 5 seconds -*-
;; "take time"
```

- Metadata
- Java Integration
- Multimethods
- Thread State

Clojure combines the power of a Lisp dialect with the convenience of the JVM. From the JVM, Clojure benefits from the existing community, deployment platform, and code libraries. As a Lisp dialect, Clojure comes with the corresponding strengths and limitations.

WRAPPING UP: STRENGTHS

- A Good Lisp
- Concurrency
- Java Integration
- Lazy Evaluation
- Data as Code

- Prefix Notation
- Readability
- Learning Curve
- Limited Lisp
- Accessibility

Most of Clojure's strengths and weaknesses are related to the power and flexibility.
If you need an extreme programming model and are willing to pay the price of learning the language, Clojure is a great fit. I think this is a great language for disciplined, educated teams looking for leverage. You can build better software faster with Clojure.