Finding Clojure

Raju Gandhi

Integrallis

Ideas, Implemented.

Disclaimers

- I tend to speak fast
- I *may* have an accent

```clojure
(def speaker {
  :name "raju",
  :pronunciation "/raa-jew/",
  :description ["java/ruby developer",
                "technophile",
                "language geek"],
  :profiles {:twitter "looselytyped"
             :facebook "raju.gandhi"}})
```

Integrallis

Ideas, Implemented.

About us...

- Small consulting/training/mentoring shop
- Based out of Ohio and Arizona
- Specialize in open-source technologies - Java/Ruby/Rails/Groovy/Grails

Integrallis
Ideas, Implemented.

Clojure?

- Lisp on the JVM
- Dynamic
- Excellent concurrency support
- Strong Java inter-op
- Lazy*

Clojure Syntax

A whirlwind tour

Clojure Syntax

```
;lists - these are special
'(+ 1 2 1/3)
;a comment
["this" "is" "a" "vector"]
;commas are whitespace
{:yes true, :no false, :null nil}
;sets
#{\a \e \i \o \u}
```

Clojure Syntax

```
;lists - these are special
'(+ 1 2 1/3)
;a comment
["this" "is" "a" "vector"]
;commas are whitespace
{:yes true, :no false, :null nil}
;sets
#{\a \e \i \o \u}
```

Clojure Syntax

```
;lists - these are special
'(+ 1 2 1/3)
;a comment
["this" "is" "a" "vector"]
;commas are whitespace
{:yes true, :no false, :null nil}
;sets
#{\a \e \i \o \u}
```

Clojure Syntax

```
;lists - these are special
'(+ 1 2 1/3)
;a comment
["this" "is" "a" "vector"]
;commas are whitespace
{:yes true, :no false, :null nil}
;sets
#{\a \e \i \o \u}
```

Clojure Syntax

```
;lists - these are special
'(+ 1 2 1/3)
;a comment
["this" "is" "a" "vector"]
;commas are whitespace
{:yes true, :no false, :null nil}
;sets
#{\a \e \i \o \u}
```

LISt Processing

```clojure
;can be a regular function
;Yes! + is a function :)
(+ 1 2 3)
;or a macro
(defn say-hello [name]
  (str "Hello, " name))
;or a special form
(if (< x 3) "less than 3" "or not")
```

LISt Processing

```
;can be a regular function
;Yes! + is a function :)
(+ 1 2 3)
;or a macro
(defn say-hello [name]
  (str "Hello, " name))
;or a special form
(if (< x 3) "less than 3" "or not")
```

LISt Processing

```clojure
;can be a regular function
;Yes! + is a function :)
(+ 1 2 3)
;or a macro
(defn say-hello [name]
  (str "Hello, " name))
;or a special form
(if (< x 3) "less than 3" "or not")
```

LISt Processing

```
;can be a regular function
;Yes! + is a function :)
(+ 1 2 3)
;or a macro
(defn say-hello [name]
  (str "Hello, " name))
;or a special form
(if (< x 3) "less than 3" "or not")
```

LISt Processing

```
;can be a regular function
;Yes! + is a function :)
(+ 1 2 3)
;or a macro
(defn say-hello [name]
  (str "Hello, " name))
;or a special form
(if (< x 3) "less than 3" "or not")
```

LISt Processing

```
;can be a regular function
;Yes! + is a function :)
(+ 1 2 3)
;or a macro
(defn say-hello [name]
  (str "Hello, " name))
;or a special form
(if (< x 3) "less than 3" "or not")
```

LISt Processing

```
;can be a regular function
;Yes! + is a function :)
(+ 1 2 3)
;or a macro
(defn say-hello [name]
  (str "Hello, " name))
;or a special form
(if (< x 3) "less than 3" "or not")
```

Homoiconicity

```clojure
;defining a function
(defn say-hello [name]
  (str "Hello, " name))
```

Homoiconicity

```
;defining a function
(defn say-hello [name]
  (str "Hello, " name))
```

Homoiconicity

```
;defining a function
(defn say-hello [name]
  (str "Hello, " name))
```

Homoiconicity

```
;defining a function
(defn say-hello [name]
  (str "Hello, " name))
```

Homoiconicity

code == data

What is FP?

- What is OOP???
- Functional programming
  - Functions are first class citizens

Why FP?

- Compartmentalize
- Better re-use
- Referential Transparency
    - Easier to test
- Easier to parallelize

Clojure's Approach

- Side effects are explicit
- State manipulation via
  - Persistent data-structures
  - Multiple reference types with appropriate semantics

Declaring Functions

```
;explicit definition
(defn times-2
  "Multiplies its arg by 2"
  [n]
  (* 2 n))
```

Declaring Functions

```
;alternate approach
;no docs though
(def times-2
     (fn [n] (* 2 n)))
```

Declaring Functions

```
;anonymous function
(map #(* 2 %) [1 2 3])
```

Declaring Functions

```
;anonymous function
(map #(* 2 %) [1 2 3])
```

Consuming Functions

```
;map takes ([f coll] ...)
(map times-2 [1 2 3])
;> (2 4 6)
```

Consuming Functions

```
;map takes ([f coll] ...)
(map #(* 2 %) [1 2 3])
;> (2 4 6)
```

Consuming Functions

```
;reduce takes ([f coll] ...)
(reduce + [1 2 3])
;> 6
```

Functions Everywhere

```
([4 5 6] 0)
;> 4
(#{\a \e \i \o \u} \a)
;> \a
({:yes true, :no false, :null nil} :yes)
;> true
(:yes {:yes true, :no false, :null nil})
;> true
```

Manipulating Data

```
(def lst '(1 2 3 4))
(first lst)
; 1
(second lst)
; 2
(nth lst 2)
; 3
(last lst)
; 4
```

Manipulating Data

```
; (def lst ‘(1 2 3 4))
(list
  (first lst)
  (second lst)
  (nth lst 2)
  5
  (last lst))

; (1 2 3 5 4)
```

Manipulating Code As Data

```
(+ 1 2)
; 3
‘(+ 1 2)
; (+ 1 2)
(eval ‘(+ 1 2))
; 3
```

Manipulating Code As Data

```
(defn say-hello
  [name]
  (str “Hello ” name))
; #'user/say-hello
(say-hello “Raju”)
; “Hello Raju”
```

Manipulating Code As Data

```
‘(defn say-hello
   [name]
   (str “Hello ” name))
; (defn say-hello [name] (str "Hello " name))
(eval *1)
; #'user/say-hello
(*1 “Raju”)
; “Hello Raju”
```

Manipulating Code As Data

```
(def lst ‘(defn say-hello
  [name]
  (str “Hello ” name)))
; #'user/lst
(first lst)
; defn
(second lst)
; say-hello
(nth lst 2)
; [name]
(last lst)
; (str "Hello " name)
```

Manipulating Code As Data

```
;(def lst ‘(defn say-hello
  [name]
  (str “Hello ” name)))
(def logged-say-hello
  (apply list
         (first lst)
         (second lst)
         (nth f lst)
         (list
           '(println "Args: " name)
           (last lst))))
; #'user/logged-say-hello

logged-say-hello
(defn say-hello [name] (println "Arg: " name) (str "Hello "
name))
```

Manipulating Code As Data

```
; logged-say-hello
; (defn say-hello [name] (println "Arg: " name) (str "Hello
" name))

(eval logged-say-hello)
; #'user/say-hello
(say-hello "Catherine")
; Args:  Catherine
; "Hello Catherine"
```

Macros

```
(defmacro log-it [fn-name args & body]
  `(defn ~fn-name ~args
     (println "Args are: " ~args)
     ~@body))

(log-it say-hello [name]
        (str "Hello " name))
; #'user/say-hello

(say-hello "Raju")
; Args are:  [Raju]
; "Hello Raju"
```

Lots More

- Immutable/Persistent data structures
- Reference types for concurrency
  - Refs
  - Agents
  - Atoms
- ...

References

- Clojure Home
- ClojureDocs
- Books
  - Clojure Programming
  - The Joy Of Clojure
  - Functional Programming for the Object-Oriented Programmer *

Thanks!