Clojure and The Robot Apocalypse

Needfuls for Newbies

Introduction

- Today we will learn about Clojure!
- Quick background - Scott and Portico
- twitter.com/sfraser

Why Cloj ure?

From Mark Volkmann's Clojure Page's "long article":

- *Are you looking for a way to make concurrent programming easier?*
- *Are open to branching outside the world of object-oriented programming to try functional programming?*
- *Is it important for the applications you write to run on the JVM in order to take advantage of existing Java libraries, portability and other benefits?*
- *Do you prefer dynamically-typed languages over statically-typed ones?*
- *Do you find the minimal, consistent syntax of Lisp dialects appealing?*

Tell me more about this. NOW.

- Clojure and it's "Four Legs"
  - Functional Programming
  - Lisp
  - Hosted on Java Virtual Machine (JVM)
  - Concurrency

Sidebar - Programming "Paradigms"

- Programming Languages have different ways they abstract, present and organize their constructs to facilitate the problem solving process
- Some programming languages are "purely functional"
- Others, such as Clojure and Scala, are "multi-paradigm"
- Clojure is Imperative, Functional, Concurrent and Reflective
- Other Paradigms you may know:
  - Visual
  - Object-Oriented
  - Actor based
  - Declarative

Second Sidebar - Some Basics

- Clojure has Numbers, Strings, Characters, Keywords, Symbols, and Collections
- Collections:
  - List: '(1 2 3)
  - Vector: [1 2 3]
  - Maps: {:key1 "Value1" :key2 "Value2"}
  - Sets: #{"This" "is" "a" "Set"}
- All collections can be treated as "Sequences"
- Sequences in Clojure are like LISP Lists but at a higher level of abstraction
- "Lazy" Sequences are evaluated on demand

Second Sidebar - More Basics

- Data is immutable with optional metadata attached
- Code is Data - Data is Code!
- Clojure is Dynamically Typed
- There is NO interpreter!
- Uses Prefix notation (aka Polish Notation)
- Programs are composed of expressions:

```
'( 1 2 3 )
( + 2 3 )
( - ( + 2 3 ) 5 )
( + 1 2 3 )
( prn ( + 1 2 3) )
( hello "world" )
```

First Leg - Functional Programming

First Leg - Functional Programming

- An ***Imperative*** Program describes an explicit series of steps that mutate state
- Functional Programming is **NOT** Imperative
- Functional Programming has a Declarative style
- Declarative programming means stating WHAT you want as opposed to HOW
- Functional programs avoid "side-effects" by absence of state and mutable data
- Functional programs lean on composition of mathematical functions as opposed to Imperative logic operating on stateful mutating VARIABLES
- Consider what a "Variable" is... something that VARIES

Declarative versus Imperative

- Declarative:
  - SELECT * FROM users WHERE fname = 'Dude'
- Imperative:
  - tbl = users
  - IF tbl.fname.hasIndex():
    - resultSet = tbl.fname.searchWithIndex('Dude')
  - ELSE
    - resultSet = tbl.fname.tableScan('Dude')

Functional versus Imperative

- Functional Programming avoids "side effects"
    - Imperative Programming relies on "side effects"
    - Functional Programming's absence of side effects provides referential transparency
- Functional Programming focuses on assembling functions
- Imperative Programming focuses on specifying a series of steps that mutate state to solve a problem
- In simple cases Functional programs resemble mathematical expressions describing a relationship
- Imperative Programs read like scripted steps

Functional vs. Imperative Example

- Add digits 1 to 10 and print result:
  - imperative style in Java:
    - int accumulate = 0;
    - for(int i=1; i<=10; i++) accumulate+=i
    - System.out.println(accumulate);
  - functional style in Clojure:
    - (println (reduce + (range 11)))
- "reduce" in clojure is a sequence *transformation*
- Other transformations available include "map" and "for"
- Note in the above example the "print" is a "side-effect"

More Variations on the Theme

- Enough Clojure trickery... what if you didn't know about "reduce"? How else can we accumulate the digits without a sequence transformation?

```
(loop [accum 0 i 10]
  (if (pos? i)
    (recur (+ accum i) (dec i))
    accum))
```

How Functional is Clojure?

- Clojure supports FP in many ways:
  - First-class functions
  - Immutable Data Structures
  - Recursive Looping
  - Mutual recursion (trampoline)
  - Lazy Sequences
  - Memoization
- See http://clojure.org/functional_programming
- Rich Hickey is a pragmatist - Clojure encourages functional programming, but does not mandate it

Second Leg - LISP

Second Leg - LISP

- **LIS**t **P**rocessing language
- Known for its heavily parenthesized syntax
- The code is made up of S-expressions
- The DATA is made up of S-expressions!
- So in a LISP, code is data... and data is code.
- This gives LISP (and Clojure) *homoiconicity*
- This facilitates the famous "macro" system of LISP that can extend the very language itself

Second Leg - LISP

- What does LISP look like?
- Factorial function:

```
(defun factorial (n)
  (if (<= n 1)
      1
      (* n (factorial (- n 1)))))
-u:**  *scratch*       All L1        (Lis
Quit
```

- What would this look like in Clojure?

```
(defn factorial [n]
  (if (<= n 1)
      1
      (* n (factorial (- n 1)))))
-u:**  ClojureFactorial.clj   All L4     (Clojure Slime)
Line-Number mode enabled
```

- Actually, you would probably do it like this:

```
(defn factorial [n]
  (apply * (range 2 (inc n))))
-u:**  ClojureFactorial.clj   All L2     (Clojure Slime
```

Second Leg - LISP - Homoiconicity

- The power of Code = Data cannot be overemphasized
- This gives Clojure a leg up over many other popular kids on the JVM, such as JRuby
- Macros allows you to *extend the compiler*
- This is pervasive throughout - many core language features are macros, not language primitives! Examples:
  - Branching: and, or, when
  - Looping: for, while
  - Syntactic Sugar: doto
  - Java interop: proxy
  - Transactions: dosync

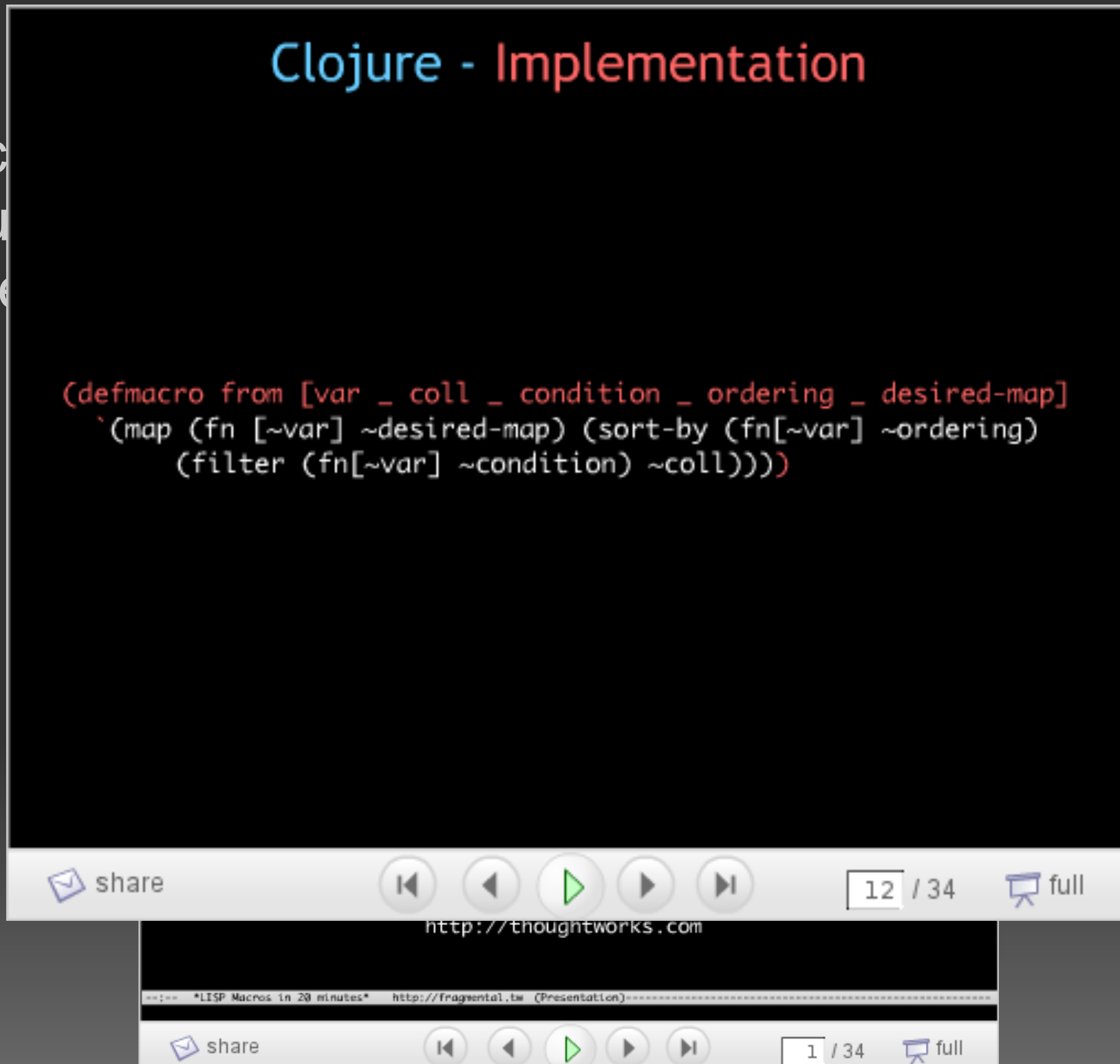
Second Leg - LISP - Macro Example

- Java has the "assert" statement
- To change it would require a new release and implementations of the Java Language Specification
- Clojure has "assert", implemented as a macro:

```
(defmacro assert
  "Evaluates expr and throws an exception if it does not
evaluate to logical true."
  [x]
  `(when-not ~x (throw
       (new Exception (str "Assert failed: " (pr-str '~x))))))
```

Second Leg - LISP - More Macros

- An exc
- The au   enting a
  LINQ fe



Third Leg - JVM

Third Leg - JVM

- Clojure is always compiled to JVM bytecode, either at runtime or Ahead of Time (AOT)
- Full access to the Java platform API's
- Provides special syntactical sugar for Java
- Some of it's libraries operate with Java objects
- Implementation of Java classes is possible using clojure proxies *with some limitations*
    - Interfaces may be implemented
    - Classes may be "extended", but there is no access to protected members of the proxied class
- Clojure uses reflection when needed
- Java "Type Hints" can be used for performance

Third Leg - JVM - Other Notes

- Clojure supports Java arrays natively, primarily for interoperation with existing Java API's
- Java Primitives types are supported, also for performance
- This support shows up in numerous ways including:
  - Coercion operations (think Java unboxing)
  - Clojure libraries that are overloaded when they process arrays of Java primitives
  - Support for primitives in let/loop bound locals
  - Type Hints for primitive arrays

Third Leg - JVM - Gimmee Sugar!

- Clojure has lots of sugary goodness to lighten things up
- Example in Java:

```
final int procs =
    Runtime.getRuntime().availableProcessors();
```

- Now in Clojure:

```
(def procs
    (.. Runtime getRuntime availableProcessors))
```

Third Leg - JVM - More Sugar!

- In Java:

```java
import java.util.HashMap;
import java.util.HashSet;
final HashMap myMapOfSets = new HashMap();
{
    final HashSet mySet = new HashSet();
    mySet.add( "Item" );
    myMapOfSets.put( "Set1", mySet );
}
```

- In Clojure:

```clojure
(import '(java.util HashMap HashSet))
(def myMapOfSets (doto (HashMap.)
      (.put "Set1" (doto (HashSet.) (.add "Item")))))
```

Fourth Leg - Concurrency

Fourth Leg - Concurrency

- Rick Hickey feels strongly about "state" and the challenges of managing concurrency in typical imperative programming languages
- He will reference these feelings when answering "why did you make Clojure"
- His opinions are compelling, and resonate with me, as someone who works with highly multi-threaded code
- Code example!

Fourth Leg - Clojure's Approach

- Clojure's explicitly isolates changeable state using four specific constructs:
  - Vars
  - Refs
  - Atoms
  - Agent
- Ideally a program will have:
  - a safe, immutable, functional model
  - a stateful mutable model accessed only via constructs listed above
- Clojure uniquely combines Software Transactional Memory (STM) and immutable data to ease concurrent pain and eliminate locks

Fourth Leg - Concurrency Links

- Clojure's approach to Identity and State.
- Clojure Concurrency - Rich Hickey presents to Western Mass. Developers Group
    - *22 Min* : "mutable objects are the new spaghetti code" and then he posits that the drive for TDD is related to the difficulties related to mutability
    - *53 Min* : great explanation of problems with multithreaded code and locking strategies
- Clojure could be to Concurrency-Oriented Programming what Java was to OOP - Compares Erlang Actors to Clojure Agents, and lot's more thoughts and links

Fourth Leg - Concurrency - Vars

- Vars refer to a mutable storage location that can be rebound at the thread level
- Rebindings on a thread can occur over and over, and the bindings will "pop" out of scope like a stack
- Think of "thread local" variables with an optional "root level" binding
- Usage of a Var is safe because it is isolated at the thread level
- If a thread binds a new value to a Var, only that thread sees it
- Vars are Dynamically Scoped

Fourth Leg - Concurrency - Vars

- Example:

```
(def myVar "Outer Value")
(do (prn myVar)
    (binding [myVar "Inner Value"] (prn myVar))
    (prn myVar))
```

- Output:

```
Outer Value
Inner Value
Outer Value
```

Fourth Leg - Concurrency - Vars

- Clojure functions defined with defn are actually in Vars too, and thus can be rebound too!

```
(defn holla [name] (prn (format "%s Rulz!" name)))

(defn rebind-example []
  (binding
      [holla (fn [name] (prn (format "%s Droolz!" name)))]
    (holla "SQL Server"))
  (holla "Oracle"))
```

- Output:
  SQL Server Droolz!
  Oracle Rulz!

Fourth Leg - Concurrency - Refs

- Vars can't share state between threads, Refs can
- Refs use STM (discussed later) to provide shared state that is safe
- Mutation of what a Ref points at must be in a "transaction"
- For more detail see Refs and Transactions
- For strict and synchronized mutation of Refs, Clojure provides "alter", or for "last one in wins" use "commute"
- Refs are dereferenced with "deref" or "@"
- Validators may be attached to Refs and used in a similar fashion to a database constraint - they literally roll the transaction back when violated!

Fourth Leg - Refs Example

```
user> (def autoindex (ref 0))
user> @autoindex
0
user> (alter autoindex inc)
java.lang.IllegalStateException: No transaction running
user> (dosync (alter autoindex inc))
1
user> @autoindex
1
user> (defn get-index [] (dosync (alter autoindex inc)))
user> (get-index)
2
user> @autoindex
2
```

Fourth Leg - Refs and STM

- Refs rely on Clojure's implementation of Software Transactional Memory (STM)
- Clojure's STM is based on MultiVersion Concurrency Control (MVCC) and provides Atomicity, Consistency, and Isolation - but not Durability
- For a short and sweet description, refer to the Concurrency chapter in Programming Clojure
- There are some STM detractors
- These detractors come at this from the Operating System/Database/VM Kernal perspective, so I am not sure how to apply their thoughts to this world

Fourth Leg - STM Debate

- Clojure: STMs vs Locks - debate between Hickey and Cliff Click Jr of Azul Systems (former architect and lead developer at Sun of the HotSpot Server Compiler)
- Concurrency's Shysters - Written by Bryan Cantrill of Sun (and author of DTrace) on why the sky is not falling
- Software transactional memory: why is it only a research toy? Authored by numerous IBM researchers and Colin Blundell of U. Penn.
- Real-world Concurrency - Great history on concurrency in hardware and software, followed by best practices. Written by Bryan Cantrill and Jeff Bonwick (author of ZFS).

Fourth Leg - Concurrency - Agents

- Think of Erlang's Actors (message-passing actors), but non-autonomous and not meant to be "distributed"
- From Agents and Asynchronous Actions:
    - *"Clojure's Agents are reactive, not autonomous - there is no imperative message loop and no blocking receive."*
- Clojure Agents provide asynchronous concurrency
- You "send" an action to an Agent, and Clojure uses a worker thread to dispatch those actions interleaved with others *one at a time* to the Agent
- Agents, like Refs, rely on Clojure's STM
- Agents can have validators and watchers

Fourth Leg - Concurrency - Atoms

- Atoms are like Refs without the STM overhead
- You can update multiple Refs in a single transaction because of it's reliance on Clojure's STM
- Atoms are standalone entities whose updating cannot be coordinated - they do NOT participate in transactions
- Clojure uses Atoms in its memoization implementation

Fourth Leg - Atoms

Clojure uses Atoms in it's implementation of memoize:

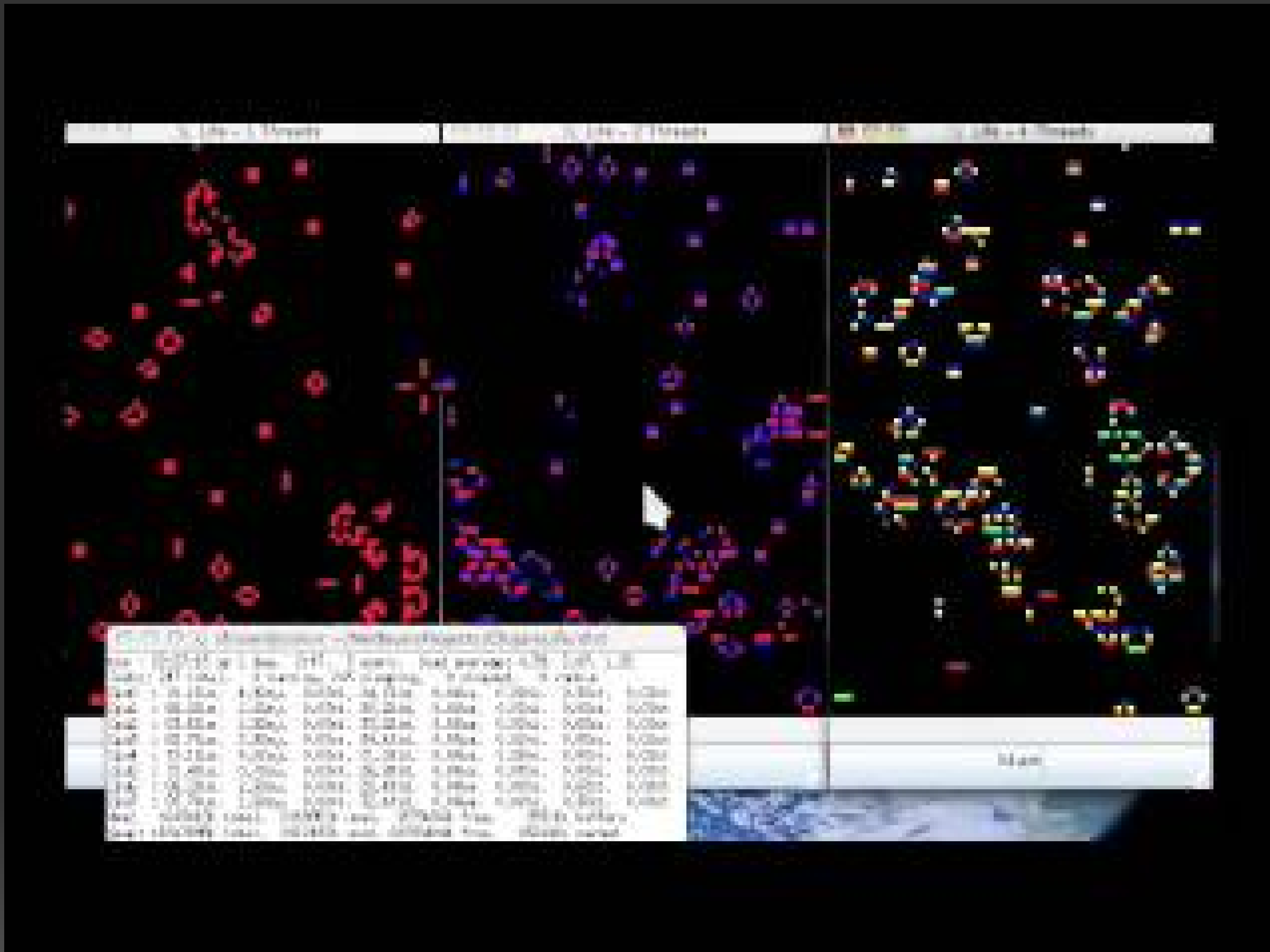
```
(defn memoize
  [f]
  (let [mem (atom {})]
    (fn [& args]
      (if-let [e (find @mem args)]
        (val e)
        (let [ret (apply f args)]
          (swap! mem assoc args ret)
          ret)))))
```

Get You Some Clojure Dawg

- Three Easy Steps:
  - svn co http://clojure.googlecode.com/svn/trunk/ clojure
  - cd clojure
  - ant
- You will probably want clojure-contrib also:
  - svn co http://clojure-contrib.googlecode.com/svn/trunk/ clojure-contrib
  - cd clojure-contrib
  - ant
- Got REPL?
  - java -cp clojure-contrib.jar:clojure.jar clojure.main

Tooling?

- Enclojure - http://enclojure.net/Index.html
- Emacs Clojure mode - http://www.lysator.liu.se/~lenst/darcs/clojure-mode/
- Another Emacs mode, and Swank server (slime) for clojure - http://clojure.codestuffs.com/
- Syntax file for VIM: http://repo.or.cz/w/clojure-patches.git?a=blob_plain;f=vim/clojure.vim;hb=HEAD
- IntelliJ Plugin - http://plugins.intellij.net/plugin/?id=4050

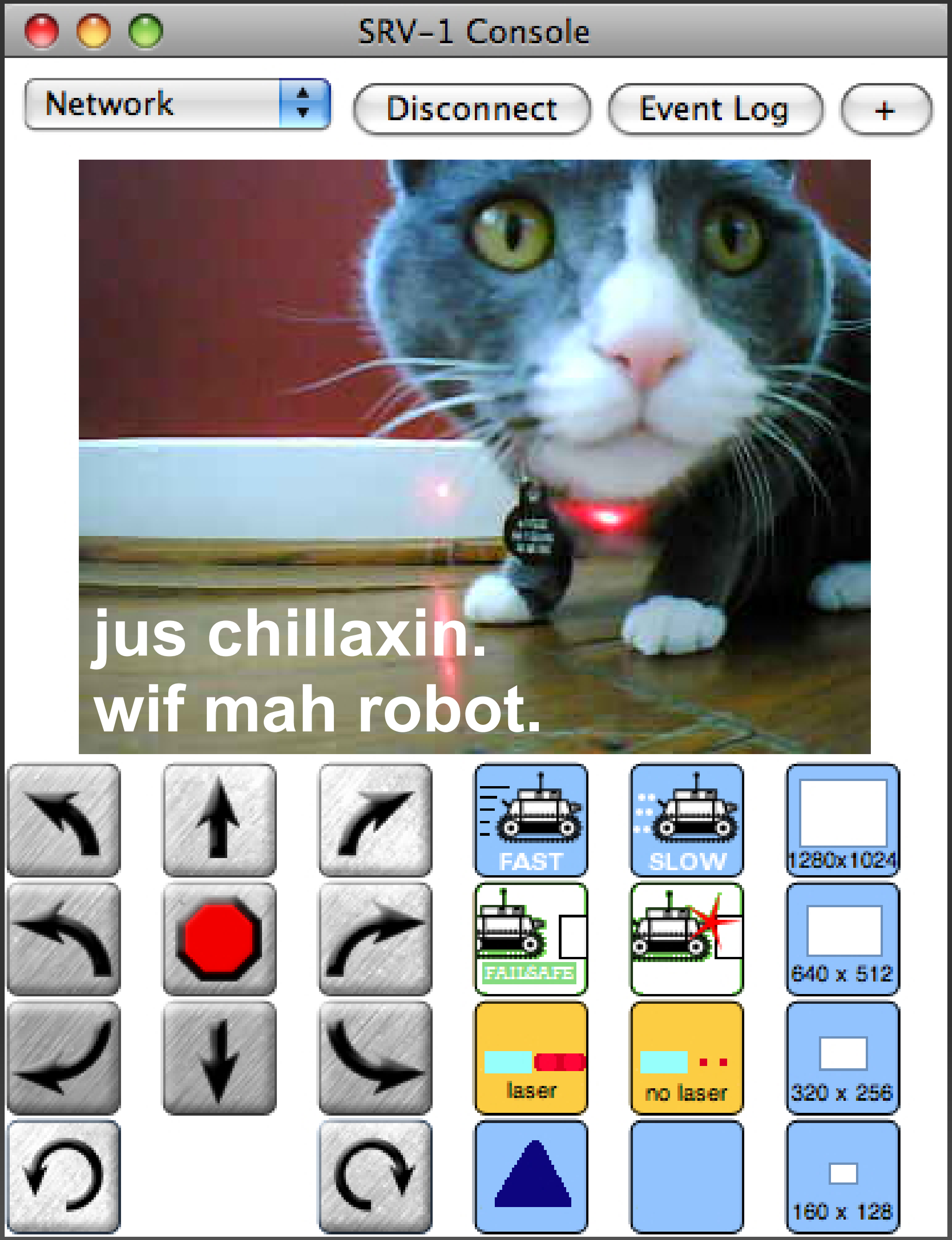


Tim Conway's Game of Life

Parallelized Game of Life Code

- Code Review
- Where does the magic happen?

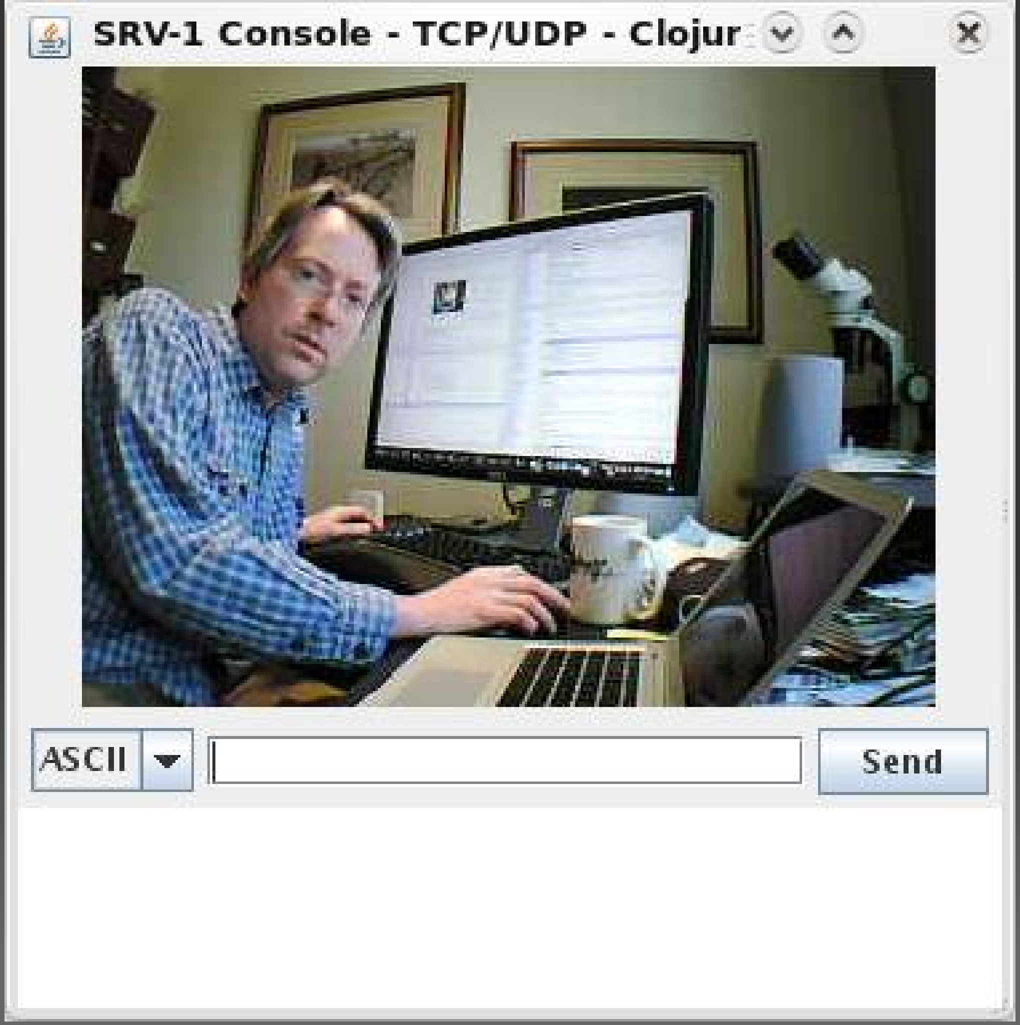
```
(defn calc-state [cell-state mycells batch-set next-color]
  (let [new-cells (ref {})]
    (dorun
      (pmap #(update-batch-of-new-cells new-cells %)
        (pmap #(calc-batch-of-new-cell-states
                         cell-state % mycells next-color)
          batch-set)))
    (dosync (ref-set mycells @new-cells))))
```


SRV-1 Console
Network
Disconnect
Event Log
+
jus chillaxin.
wif mah robot.
FAST
SLOW
1280x1024
FAILSAFE
640 x 512
laser
no laser
320 x 256
160 x 128

Robot Time

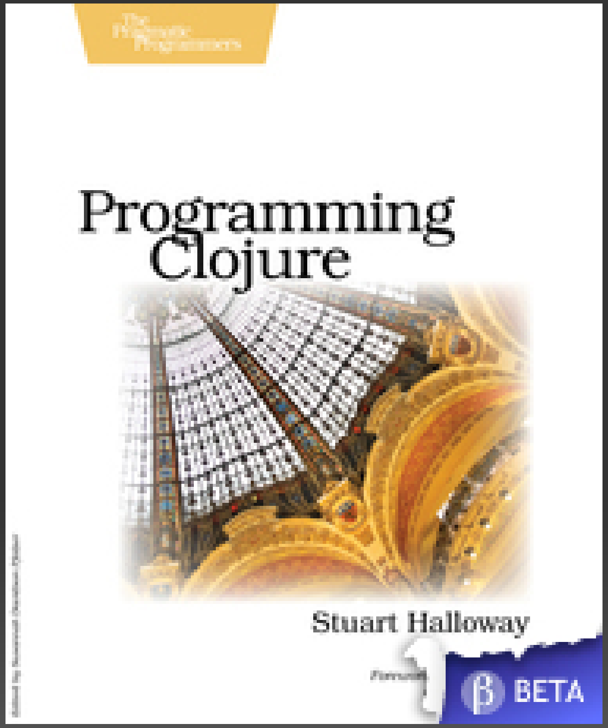
Robots!!!

- SRV-1 Console Code Review
- http://www.inertialabs.com/srv.htm



Recommended

http://www.pragprog.com/titles/shcloj/programming-clojure



Thank you!

- The Philly Lambda Group and the Functional Programming evangelists who introduced me to Clojure (Kyle Burton, Mike DeLaurentis, etc...)
  - http://groups.google.com/group/philly-lambda
- Rich Hickey and the folks behind Clojure:
  - http://clojure.org/