Clojure

Daniel McCarney
COSC 3P93

Material Covered

- What is Clojure?
- Why do we care?
- What isn't it?
- Intro to Functional Programming
- Concurrency Support
- Conclusions

What is Clojure?

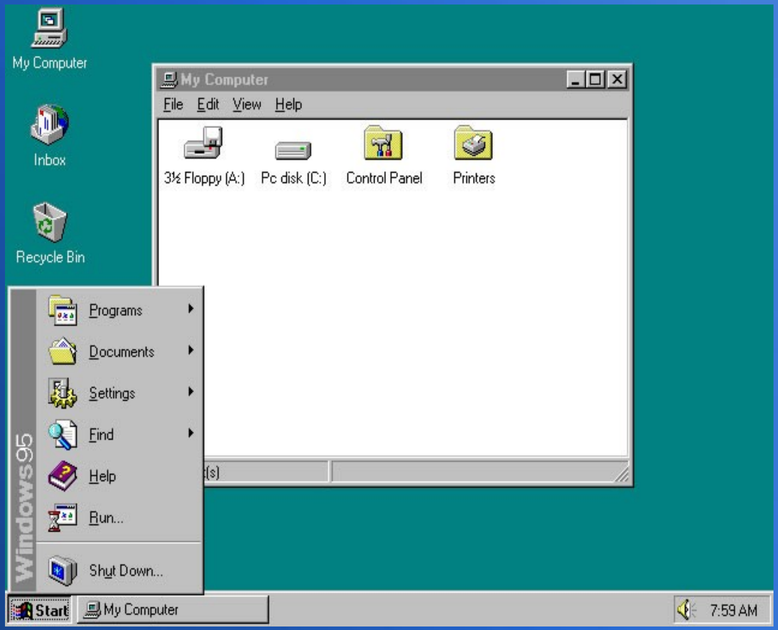
The Clojure Rationale

- Created by Rich Hickey in 2007
- A LISP
  - LISt Processing (or Lots of Irritating Superfluous Parentheses)
- For functional programming
  - Immutable Data
  - First Class Functions
- Exploiting Java
  - Compiles to bytecode (JVM run)
- Concurrency built-in
  - Not an after thought or library

Ok... So what?

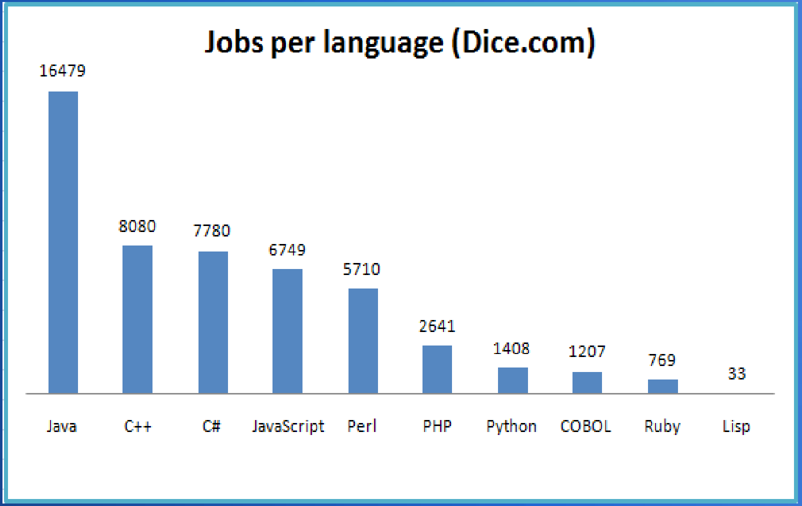
Backwards is Forwards

- Backwards compatibility matters
  - You already have programmers
  - You already have applications & libraries
  - You already have hardware
- Need concurrency? Now what?
  - Retrain, rewrite, replace, make do?
- Need to be realistic
  - A stepping stone to concurrent bliss



Exploit the Market

- Java is well established in the programming industry
  - Lots of trained programmers
  - Lots of applications, libraries and support
  - "Write once, run anywhere"
- Java Virtual Machine (JVM)
  - Incredibly well tested
  - Over 10 years of optimization and tuning
- Already established as a platform for non-Java languages
  - Jython, Jruby, Jscheme, Groovy, Scala, etc
    - Join the party. Bring a "j".



Functional, or Dysfunctional?

- Immutable data perfect for concurrency
  - If you can't change it, you never need to worry
  - Synchronization, deadlock, etc
- “First class” functions
  - Make functions on the fly
  - Pass them around as data.
  - No “side effects”
- “Homoiconic”
  - Programs represented in the language's own datastructure
  - Code is data, data is code.

- “If you don't think carefully, you might believe that programming is just typing statements in a programming language.” - W. Cunningham

Threads and Locking? No thanks

- Threads and Locking
  - Complex!
  - Error prone
  - A debug nightmare
  - Potentially slow
- Concurrent from the start
  - Protect all memory
  - Higher level abstractions (like Ada's Tasking)

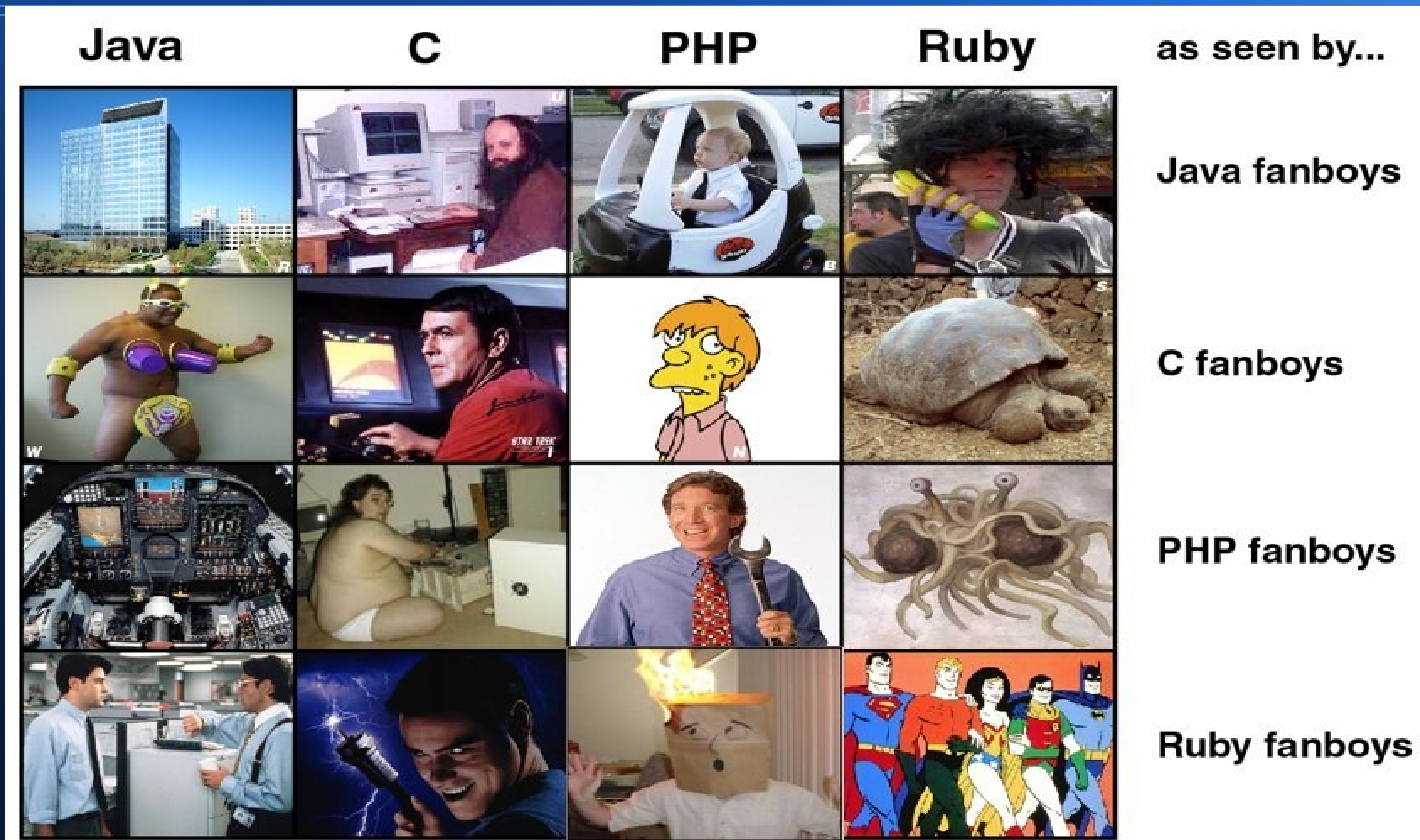
Too good to be true?

(defn fits-all? [x] (if (= x 'one-size) nil))

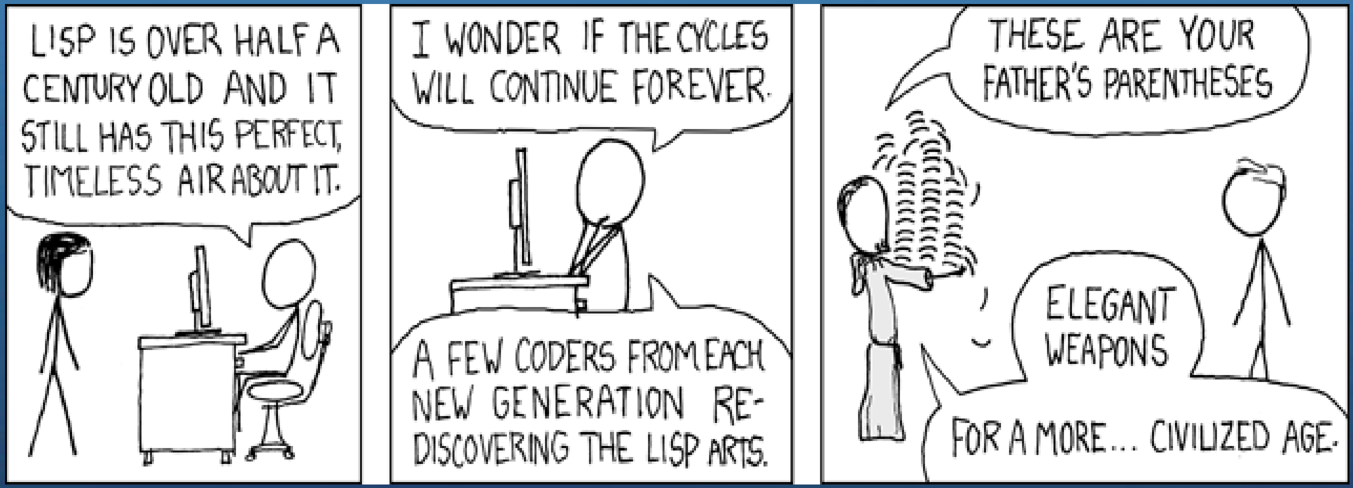
- No panaceas
  - one language isn't going to work for any and all applications
- The benefits of the JVM come with drawbacks
  - Real hardware hidden.
  - Little control of operating system
  - No clusters or “bare metal” execution
- Still LISP-y in syntax
- New = Scary

- “There are only two kinds of programming languages: those people always bitch about and those nobody uses.” - B. Stroustrup

A Matter of Perspective



LISP?

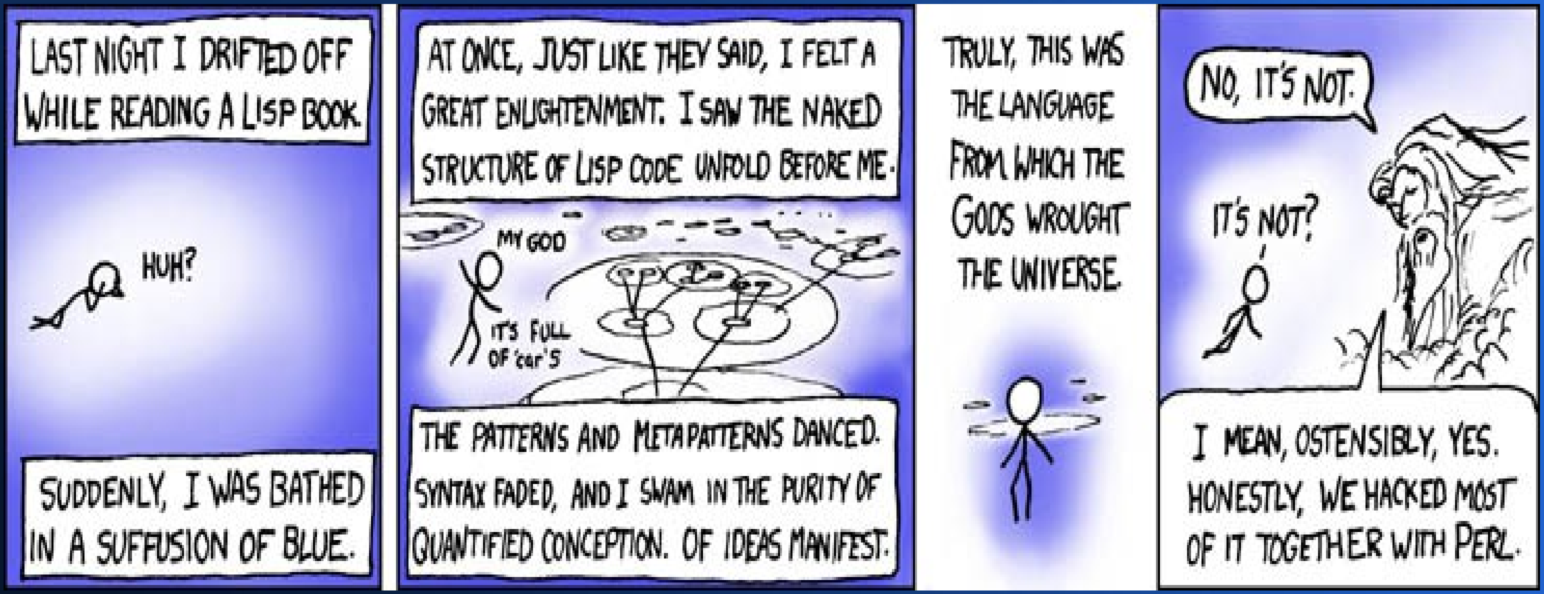

LISP IS OVER HALF A CENTURY OLD AND IT STILL HAS THIS PERFECT, TIMELESS AIR ABOUT IT.
I WONDER IF THE CYCLES WILL CONTINUE FOREVER.
A FEW CODERS FROM EACH NEW GENERATION RE-DISCOVERING THE LISP ARTS.
THESE ARE YOUR FATHER'S PARENTHESES
ELEGANT WEAPONS
FOR A MORE... CIVILIZED AGE.

LISP!?! Why LISP?

- LISP will not die
  - John McCarthy, ~1958. Second oldest high level language.
  - Almost no core syntax to learn
  - Able to adapt to every new programming paradigm.
  - Code is data is code -> homoiconic
    - A “programmable programming language”
- Ahead of the curve
  - Read-Execute-Print model
  - “Often emulated, never duplicated”
- Math doesn't get stale!
  - “Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I”
  - Turing-complete algorithm language

Borrowing Credibility

- “A language that doesn't affect the way you think about programming is not worth knowing.” - A. Perlis
- “The tools we use have a profound (and devious!) influence on our thinking habits, and, therefore, on our thinking abilities.” - E. Dijkstra

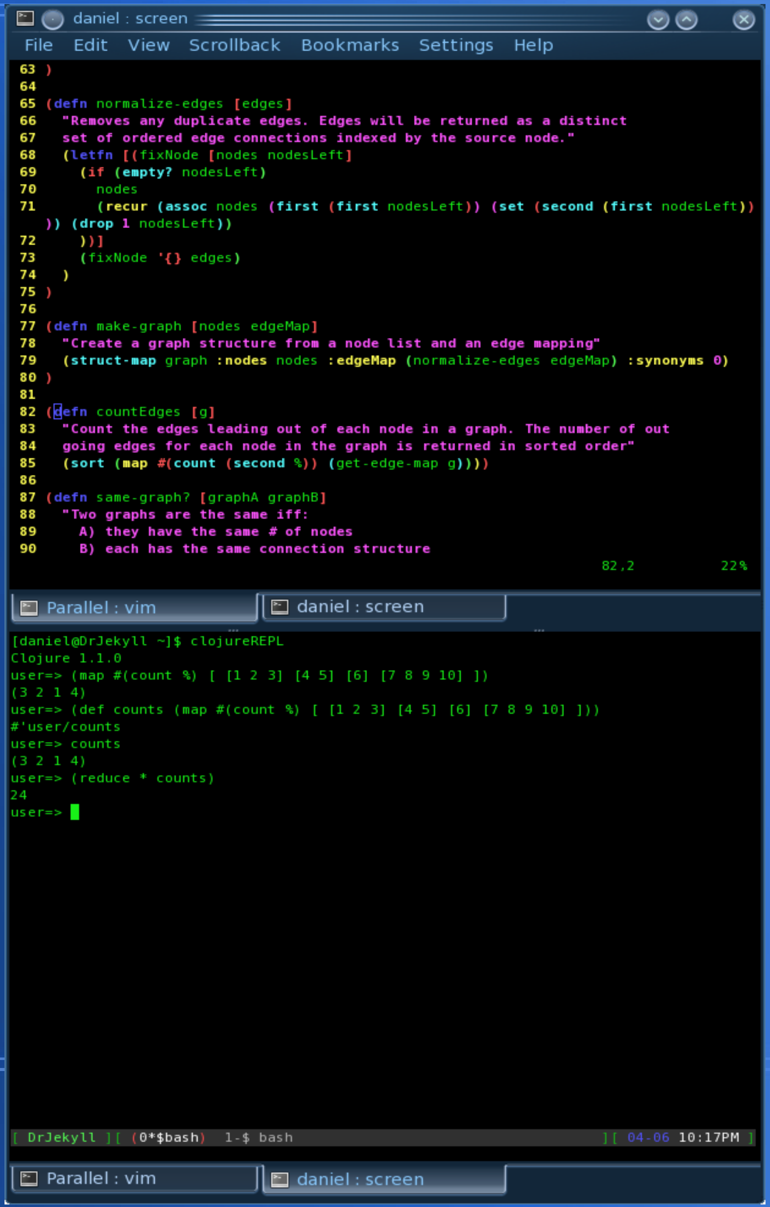


Functional Programming

- A light-speed introduction to FP using Clojure
- If you've taken COSC2P90 – pretend you haven't...
- Resources at the end for the curious

The REPL

- LISP – the origin of iterative development
  - A shell for experimental coding
  - Read -> Evaluate -> Print -> Loop
  - A programmer's best friend
- Code, Documentation, Testing, and Debugging in one.



Core Datatypes

- Numbers: 4, 1.0, 22/7, -99999999999999
  - Convert to BigDecimal as required
  - Built in ratio type
- Strings & Chars: “Foo”, \f
  - Unicode, pretty much the same as Java.
- List: (+ 5 9 9.0)
  - Used for function calls.
  - If you think of an “add” method in Java how do you call it? add (5, 9);
  - Put the function name inside the brackets too, more orthogonal (add 5 9)
  - Now understand that clojure's built in operators are functions too (+ 5 9)
- Vector: [5, 9, 9.0]

Core Datatypes (Cont'd)

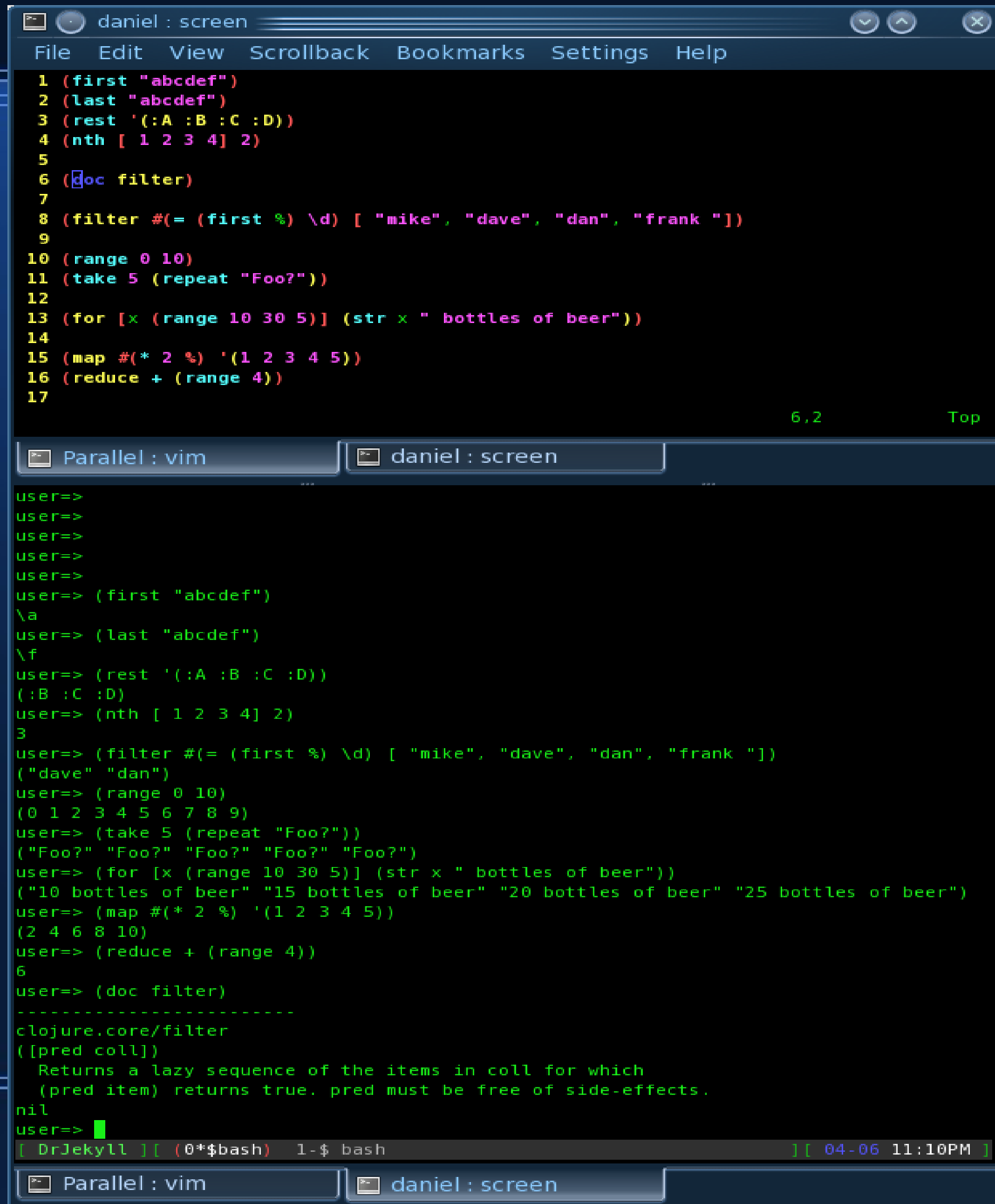
- Dictionary/Mapping: { :key “value1” :key2 “value2 }
  - Key to Value lookup table.
  - Allows for key missing default values, access to just values, just keys.
- Keyword: :key
  - Evaluate to themselves, used for fast equality checks
  - (:key2 { :key “value1” :key2 “value2” }) results in “value2”
- Symbol: '(someFunc 2 4)
  - Code is data, we need a way to express code so it won't be run
  - Without the ' shorthand for **quote**() Clojure will run **someFunc(2, 4)**

“One sequence to access them all, and in the bytecode bind them”

- Clojure ***unifies*** the majority of the mentioned data types under the Sequence interface.
- Accessing vectors, lists, maps, sets, and strings uniformly
  - first, rest, cons, next, conj, into
- Create sequences on the fly:
  - Range, repeat, cycle, interleave
  - “for” (see: python list comprehension)
- “Filter” sequences to find elements with specific properties
  - filter, take-while, drop-while
- Transform sequences
  - map, reduce, sort

- “Any sufficiently advanced technology is indistinguishable from magic.” - A. Clarke

~~Pics~~ Code or it didn't happen.



The Art of Lazy

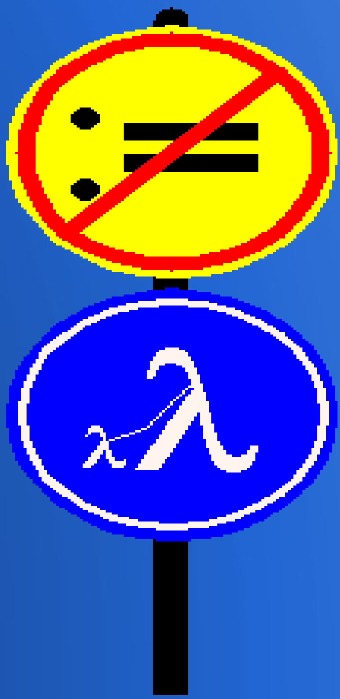
- **A lazy sequence:**
  - Elements not calculated until needed
  - Postpones expensive computations, delays I/O
  - Work with data sets bigger than your memory capacity
- Create “lazy sequences” on the fly out of function results
  - See “*yield*” in some other languages. Concept of a “**generator**”
- Other sequences already provided “lazy”
- Lazy sequences make it possible to have “infinite sequences”
  - If the next value is computable into infinity...
  - Compute the ones you need on a lazy basis

Of Wizards and Lambdas

- First class functions mean that we can pass them around as data
  - See (reduce + (range 5))
  - Passes the '+' function to the higher order function "reduce"
- Further, functions can be nested within other functions
- Functions can be created on the fly.
  - A "lambda" is an unnamed function. Similar to an anon. inner class in Java
  - Can use the "fn" function to specify a function with formal parameter names
  - Can use the %() reader macro to create a function that uses "%1" style tokens to access arguments
- Returning a function created on the fly to wrap a piece of data is "closing over" the data. Think: abstraction in OO terms (*private member scope*).

A while() before loops...

- Looping requires mutable state
  - Counter variables (i,j,k), boolean status flags (is_done, has_data)
- Functional programming uses recursion
- Clojure has no loops, only advanced recursion options.
- But what of performance?
  - Language support for Memoization
  - Partial tail-recursion
  - Libraries for easily “trampolining”



Java Inter-op

- Create new Java objects
  - (new Random)
- Call methods on the object
  - (. (new Random) nextInt)
- Masquerade as a subclass or an interface implementer
  - Runtime **proxy** function
  - Lets you take a binding of functions and get an object
  - Can save runtime computed bytecode to a .class file
- Clojure functions **all** implement the Runnable & Callable interfaces
  - Can immediately be run on their own thread. No changes required.

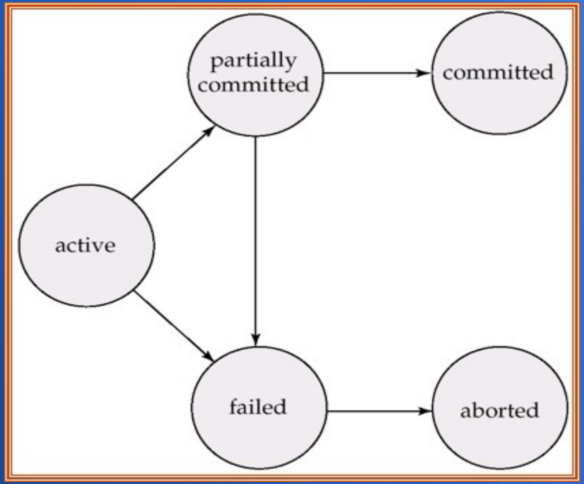
Concurrency

A problem

- Traditional concurrent programming (in particular Java) requires the programmer manage data access **very** carefully.
- Locking based schemes used to synchronize access to key resources.
  - Allows one one thread access at a time *(see: bottleneck)*
  - Not being careful leads to hard to reproduce deadlock and concurrent memory access issues.
- Clojure has a natural advantage due to it's pure functions and immutable data.
- For everything else Clojure provides a layer above memory that acts as a controller for concurrent modification without explicit programmer interaction.
- This layer is based on the concept of ACID transactions from enterprise database servers.

ACID (hallucination free)

- ACID is defined:
  - Atomicity
  - Consistency
  - Isolation
  - Durability



- Atomicity defines “all or nothing” behavior
- Consistency ensures that the system is always in a known state.
- Isolation requires that no other operations can access/view data from an in-progress transaction
- Durability ensures that once a user has been notified of a successful transaction result, the transaction will not be lost in the result of a crash.

Software Transactional Memory

- Clojure adapts the concept of ACID transactions to main memory
- References to mutable state created
- The reference can only be changed inside of a transaction
- Transactions are managed in an ACID fashion. If a transaction is queued, it will retry until successful. STM loses the “Durability aspect”. RAM Only.
- Very optimistic locking
  - Read-only access will never block writers or other readers.
  - “Speculative evaluation”, things may be undone, or re-tried inside of a transaction. Changes are isolated and can be undone if required.
- More noticeable overhead on small # of processors

Clojure Concurrency Library

- Clojure provides several options for concurrent code

- At the most fine grain level is the STM implementation and refs.
- Atoms manage ***uncoordinated*, *synchronous*** changes to shared state.
- Agents manage ***asynchronous*** changes to shared state.
- Vars manage ***thread-local*** state. (Dynamic rebinding)

Clojure Concurrency Library Contd.

- **Atoms**
- Protect a single ref from uncoordinated synchronous changes.
- Because they protect a singular reference an atom does not need to be updated inside of a transaction.
- As a side-effect, you can not update two atoms at once in a coordinated fashion.
- Lighter weight than directly using refs and transactions. Less for Clojure to protect you from.
- **Agents**
- Specialized for tasks that can proceed independently - minimal coordination.
- Comparable to Ada's tasking approach.
- Wrap an initial state and accept functions to update this state. Update funcs are queued, eventually run on their own thread.

Some Code

If time has permitted...

Counting node edges

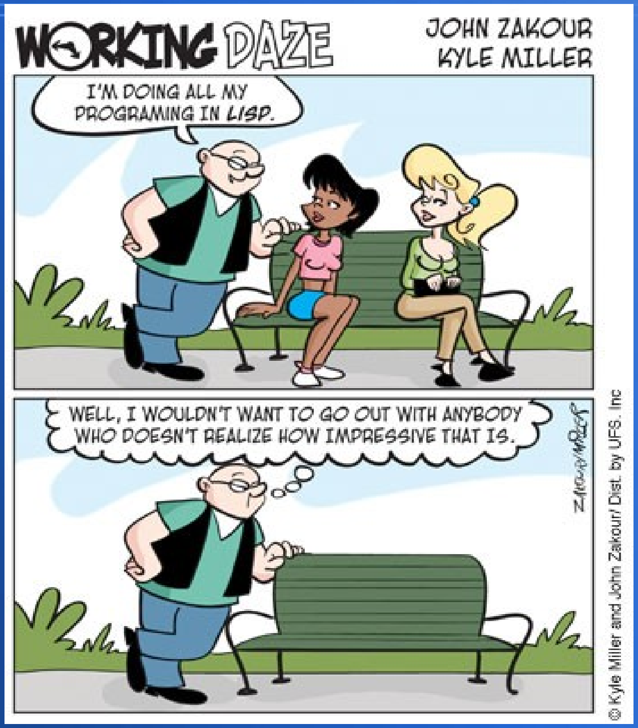
```
(defn countEdges [g]
  "Count the edges leading out of each node in a graph. The number of out
  going edges for each node in the graph is returned in sorted order"
  (sort (map #(count (second %)) (get-edge-map g))))
```

- Extracts a map of node to edge connections from graph g
  { :0 [:1 :2], :1 [:0], :2 [:0] }
- Counts the second element of each keypair (i.e. the value)
  (count [:1 :2]) (count [:0]) and (count [:0])
- Creates a list from those count results
- Sorts that list
  (1, 1, 2)

Questions?

Resources

http://clojure.org specifically:
- http://clojure.org/rationale
- http://clojure.org/concurrent_programming
- http://clojure.org/getting_started



http://norvig.com/
http://www.paulgraham.com/icad.html
http://mitpress.mit.edu/sicp/

http://groups.csail.mit.edu/mac/classes/6.001/abelson-sussman-lectures/
"Programming Clojure" - http://bit.ly/dAaKrW
http://kotka.de/projects/clojure/vimclojure.html