Clojure for Beginners

Elango Cheran

June 22, 2013

Get Clojure I

- Clojure (actually) implemented as a Java library
  - Need standard (Sun/Oracle) Java 1.6+ - `http://www.oracle.com/technetwork/java/javase/downloads/index.html`
  - Clojure JAR downloads - `http://clojure.org/downloads`
  - Can run the REPL ("interpreter") with `java -cp clojure-1.5.1.jar clojure.main`
- Try Clojure - online vanilla REPL - `http://tryclj.com/`

Get Clojure II

- Leiningen - de facto build tool - http://leiningen.org/
  - New project - `lein new <project_name>`
  - Open a REPL - `lein repl`
    - The REPL from Leiningen maintains proj. libs (classpath), command history, built-in docs, etc.
  - So easy that you don't notice Maven is underneath
- Light Table - evolving instant-feedback IDE - http://www.lighttable.com/

“Traditional” IDEs for Clojure I

- Emacs (!)
  - Paredit mode - one unique advtange of Lisp syntax
    - Imbalanced parenthases (& unclosed strings) no longer possible
    - Editing code structure as natural as editing code
  - Integrated REPL, lightweight editor, etc.
  - Get Emacs 24b or later, and install `emacs-starter-kit`
- Eclipse + Counterclockwise
  - “Strict Structural Edit Mode” is steadily replicating Paredit mode
- Vi, IntelliJ, etc.

“Traditional” IDEs for Clojure II

Shortcuts to learn (and my configurations)

`paredit-forward` (`C-M-f`), `paredit-backward` (`C-M-b`), `paredit-forward-slurp-sexp` (`C-<right>`), `paredit-forward-barf-sexp` (`C-<left>`), `paredit-backward-slurp-sexp` (`C-M-<left>`), `paredit-backward` (`C-M-<right>`), `paredit-backward` (`C-M-b`), `paredit-backward` (`C-M-b`), `paredit-split-sexp` (`M-S`), and there’s more . . .

What This Presentation Covers

- An introduction to Clojure
- A cursory comparison of Java, Clojure, Ruby, and Scala
- Code snippets as needed
- Explanation of design considerations
- Additional resources

Interesting Things Not Covered

- ClojureScript
- Specific DSLs & frameworks
- Clojure's concurrency constructs & STM

Overview of Presentation

- Brief intro of Clojure dev tools
- Brief comparison of languages w/ snippets
- Explanation of main Clojure concepts
- Hands-on example(s)

Teasers

1. Add sum of squares of all numbers in a list

Teasers

1. Add sum of squares of all numbers in a list
2. Open, use, and close multiple system resources

Teasers

1. Add sum of squares of all numbers in a list
2. Open, use, and close multiple system resources
3. Filter all lines of a file based on a reg. exp.

Teasers

1. Add sum of squares of all numbers in a list
2. Open, use, and close multiple system resources
3. Filter all lines of a file based on a reg. exp.
4. Read in a line, skip first line, take every 3rd

Teaser #1

- Idea: Add sum of squares of all numbers in a list
- Java

```
// int[] nums = {8, 6, 7, 5, 3, 0, 9};
float average(int[] nums) {
    float sum = 0.0;
    for (int x :  nums) {
        sum += x;
    }
    return sum / nums.length;
}
```

- Clojure

```
; (def nums [8 6 7 5 3 0 9])
(defn average [nums]
  (/ (reduce + nums) (count nums)))
```

- All values in input Java array, etc. must be of same type
  - Unless you use an untyped Java collection ...
    - ... and pre-emptively cast to float

Teaser #2 I

- Idea: Open, use, and close multiple system resources
- Java

```
Socket s = new Socket("http://tryclj.com/", 80);
OutputStream fos = new
FileOutputStream("index_copy.html");
PrintWriter out = new PrintWriter(fos);
try {
    // do stuff...
}
finally {
    out.close();
    fos.close();
    s.close();
}
```

Teaser #2 II

- Clojure

```
(with-open [s (Socket.  "http://tryclj.com" 80)
            fos (FileOutputStream.
"index_copy.html")
            out (PrintWriter.  fos)]
  ;; do stuff
  )
```

- The predictable parts:
  - `.close()`
  - Close in reverse order
  - A `try-catch-finally` block for clean I/O usage

Teaser #3

- Idea: Filter all lines of a file based on a reg. exp.
- Java

```
BufferedReader br = new BufferedReader(new
FileReader(file));
String line;
while ((line = br.readLine()) != null) {
    if (line.matches("\\d{3}-\\d{3}-\\d{4}")) {
        System.out.println(line);
    }
}
br.close();
```

- Clojure

```
(with-open [br (BufferedReader.
(clojure.java.io/reader file))]
  (doseq [line (line-seq br)]
    (when (re-matches #"\d{3}-\d{3}-\d{4}" line)
      (println line))))
```

Teaser #4

- Idea: Read in a line, skip first line, take every 3rd
- Java

```java
String line;
int counter = 0;
br.readLine(); // assume not EOF
while ((line = br.readLine()) != null) {
    if (counter % 3 == 0) {
        System.out.println(line);
    }
    counter++;
}
```

- Clojure

```clojure
(doseq [line (take-nth 3 (rest (line-seq br)))]
  (println line))
```

REPL

- REPL = Read-Eval-Print Loop
- "Interactive interpreter"
- `user> 1`
  `1`
  `user> 4.5`
  `4.5`
- Also try 22/7, \e, 1000000000000000000000, first, str, +, 2r10101010, "hello", 0.000000000000000000000000314, [2 4 8], {"key" "value"}

Bindings I

- "binding" = assigning a value to a symbol
  - Clojure promotes alternative ways to manage state, and "variable" would be misleading
- In general
  - Bindings are made at diff. times w.r.t. compiling (static / dynamic)
  - Bindings are made within a context (lexical / dynamic scope)
- Clojure is dynamic (uses dynamic bindings)
  - Clojure promotes lexical scoping, allows easy dynamic scoping
  - You can "hot swap" live code
  - Lexical scope + a function = a closure

Bindings II

- Clojure

```
user> (def a 3)
#'user/a
user> a
3
user> (def b 5)
#'user/b
user> b
5
```

- Java

```
int a = 3;
a;
int b = 5;
b;
```

Bindings III

- Ruby

```
irb(main):001:0> a = 3
3
irb(main):002:0> a
3
irb(main):003:0> b = 3
3
irb(main):004:0> b
3
```

- Scala

```
scala> val a = 3
a:  Int = 3

scala> a
res10:  Int = 3

scala> val b = 5
```

Bindings IV

```
b: Int = 5

scala> b
res11: Int = 5
```

Typing

- The types of values and how they are resolved
- Through Clojure, still using Java, just differently
- Strong typing (like Java, Ruby, Scala; unlike Perl)
  - Type hierarchies, interfaces, etc.
  - Types of basic values are actual Java types. Try:
    ```
    (class 1)
    (class 4.5)
    (class "yolo")
    ```
- Dynamic typing (like Perl, Ruby, Scala; unlike Java)
  - Type checking happens at run-time, not compile-time
  - Trust in programmer's ability to write good code
  - Benefit is expressive power (ex: macros)
  - Incremental development via REPL ⇒ less unexpected surprises

Typing Examples I

- Clojure

```
user> (def a "not a Long")
#'user/a
user> (class a)
java.lang.String
user> (def a [1 2 3])  ;; no commas!  commas
treated like whitespace
#'user/a
user> (class a)
clojure.lang.PersistentVector
```

  - Side note: Clojure has other "container types" (beyond just a "variable") to manage state
- Java
  - Variables are declared with a type that cannot change
  - Prevents a lack of clarity on what a symbol represents...
  - ...but also restricts power of functions, collections, etc.

Typing Examples II

- Ruby

```
> a = "not a long"
=> "not a long"
> a.class
=> String
> a = [1, 2, 3] # commas required
=> [1, 2, 3]
> a.class
=> Array
```

- Scala

```
scala> var c = 4.5
c:  Double = 4.5

scala> c.getClass
res0:  java.lang.Class[Double] = double

scala> c = 3.5
c:  Double = 3.5
```

Typing Examples III

```
scala> var c = "not a Long" // re-defining c
required to store object of diff type
c:  java.lang.String = not a Long

scala> val d = Vector(1, 2, 3)
d:  scala.collection.immutable.Vector[Int] =
Vector(1, 2, 3)
```

- A 'val' ("value") in Scala is immutable
- A 'var' ("variable") is mutable but type is fixed, like Java

Follow Along I

1. Install Leiningen and Light Table
2. At the command line, run `lein new oakww`
3. Run a REPL at the command line via Leiningen
   - `cd oakww`
   - `lein repl`
4. Now open Light Table
   - In the "Workspace" tab on the left, choose "Folder" Link at top
   - Select the folder of the Leiningen project we created (`lein repl`)
   - Expand to and click the source file (`oakww > src > oakww > core.clj`)

Follow Along II

5. Enter the following code in both command-line REPL and core.clj open in Light Table

```
(class 4.5)
(class 22/7)
(def a [1 2 3])
(class a)
(first a)
(rest a)
(def b "hella")
(first b)
(rest b)
(class (first b))
(class (rest b))
```

Follow Along III

6. In Light Table, in the "Command" tab on the left, select "Instarepl: Make current editor an Instarepl"

Follow Along IV

7. Some notes on Light Table (curr. ver.: 0.4.11)
   - Constant evaluation
     - Instant feedback
     - Works well in some cases (pure / stateless functions, web, testing)
     - Not what you want in other cases (stateful fns / I/O, GUI)
   - Standard command-line REPL is the "canonical" REPL
     - Especially if you have confusion on return vals vs. stdout, etc.
   - Many people still stick with emacs + nREPL for optimal productivity

Functions

- Prefix notation - functions go in first position

```
(def a 3)
(def b 5)
(+ a b)
(+ a b 7 1 6)
```

Notes on Syntax I

- Clojure
  - Myth: Lisp's parentheses drown out code

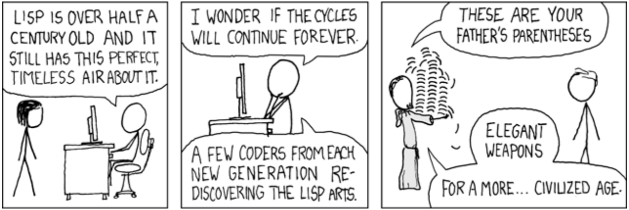

Figure : from XKCD

- Well, Common Lisp does have a lot. . .
- . . . but Clojure reduces them, uses vector square brackets, too

Notes on Syntax II

- Overall, Clojure has same or less parens+brackets+braces than many other languages (less code!)

  ```
  objA.method(b, c, d);
  ```
  ⇓
  ```
  (function a b c d)
  ```
- Using Paredit mode (or equivalent) makes editing easy and having imbalanced parens difficult

Figure : from XKCD

- Commas are whitespace
  - Useful for macros
- Java
  - There is a lot of code

Notes on Syntax III

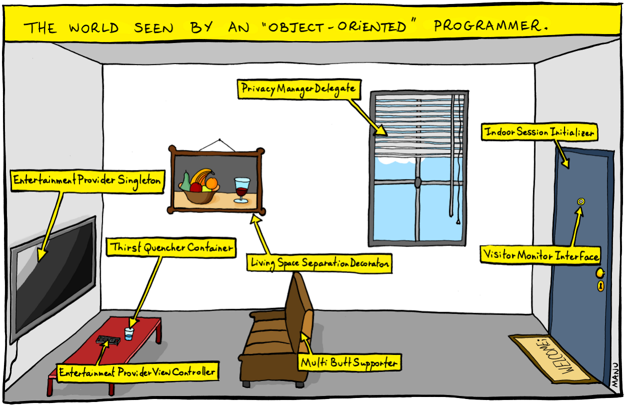

Figure : from Bonkers World

- Ruby
  - fn call parens can be omitted when the result is not ambiguous

Notes on Syntax IV

- semicolon optional at end of the line

```
> def add_two(x)
>   x + 2
> end
=> nil
> add_two 6
=> 8
```

- Scala
  - Type declarations go after a variable / function name, not in front
    - Omissible when type can be inferred
  - fn call parens can be omitted when the result is not ambiguous
  - Semicolon optional at end of line

Data Structures I

- 4 basic data structures with literal support in Clojure: lists, vectors, maps, sets
  - List: `(1 1 2 3)`
  - Vector: `[1 1 2 3]`
  - Set: `#{1 2 3}`
  - Map: `{"eins" 1, "zwei" 2, "drei" 3 }`
- A lot of data can be represented through composites of these
- Functions are executed through lists (fn is in first position)

Data Structures II

- Clojure

```clojure
(def l (list 1 1 2 3))
l
(def v [1 1 2 3])
v
(def s #{1 2 3})
s
(def m {"eins" 1, "zwei" 2, "drei" 3})
m
```

- Java

```java
// omitting plain arrays
import java.util.List;
import java.util.ArrayList;
List l = new ArrayList();
l.add(1); // only with auto-boxing starting in
Java 1.5 aka 5
l.add(1);
l.add(2);
l.add(3);
```

Data Structures III

```
System.out.println(l);
// [1, 1, 2, 3]
ArrayList v = new ArrayList(); // ArrayList
replaced Vector in Java 1.2

import java.util.Set;
import java.util.HashSet;
Set s = new HashSet();
set.add(1);
set.add(2);
set.add(3);
System.out.println(s);
// [1, 2, 3]

import java.util.Map;
import java.util.HashMap;
Map m = new HashMap();
m.put("eins", 1);
m.put("zwei", 2);
```

Data Structures IV

```
m.put("drei", 3);
System.out.println(m);
// {zwei=2, drei=3, eins=1}
```

- Ruby

```
v = [1, 2, 3]
v
s = Set.new([1, 2])
s
m = {"eins" => 1, "zwei" => 2, "drei" => 3}
m
```

- Scala

```
val l = List(1, 2, 3)
val l2 = 1 :: 2 :: 3 :: List()
l
val v = Vector(1, 2, 3)
v
val s = Set(1, 2, 3)
s
```

Data Structures V

```
val m = Map("eins" -> 1, "zwei" -> 2, "drei" ->
3)
m
```

Immutability I

- Values don't change after declared
- Clojure
  - Data structures (and any other value) are immutable
  - Try:

```
(def v1 [5 6])
(def v2 [7 8])
(concat v1 v2)
v1
v2
(def m {9 "nine", 8 "eight"})
(assoc m 7 "seven")
m
```

- Java
  - People with experience say no such thing as "somewhat immutable" code

Immutability II

- No immutable data structures originally, except for Strings, actually

```
String str1 = "hobnob with Bob Loblaw";
String str2 = " on his Law Blog";
str1.concat(str2);
System.out.println("str1 = [" + str1 + "]");
System.out.println("str2 = [" + str2 + "]");
// str1 = [hobnob with Bob Loblaw]
// str2 = [ on his Law Blog]

String str3 = str1.concat(str2);
System.out.println("str1 = [" + str1 + "]");
System.out.println("str2 = [" + str2 + "]");
System.out.println("str3 = [" + str3 + "]");
// str1 = [hobnob with Bob Loblaw]
// str2 = [ on his Law Blog]
// str3 = [hobnob with Bob Loblaw on his Law
Blog]
```

Immutability III

- Ruby
  - Like Java, does not have immutable types
- Scala

```
scala> val v1 = Vector(5, 6)
v1:  scala.collection.immutable.Vector[Int] =
Vector(5, 6)

scala> val v2 = Vector(7, 8)
v2:  scala.collection.immutable.Vector[Int] =
Vector(7, 8)

scala> v1 ++ v2
res1:  scala.collection.immutable.Vector[Int] =
Vector(5, 6, 7, 8)

scala> v1
res2:  scala.collection.immutable.Vector[Int] =
Vector(5, 6)
```

Immutability IV

```
scala> v2
res3:  scala.collection.immutable.Vector[Int] =
Vector(7, 8)

scala> val m = Map( 9 -> "nine", 8 -> "eight")
m:
scala.collection.immutable.Map[Int,java.lang.String]
= Map(9 -> nine, 8 -> eight)

scala> m + (7 -> "seven")
res4:
scala.collection.immutable.Map[Int,java.lang.String]
= Map(9 -> nine, 8 -> eight, 7 -> seven)

scala> m
res5:
scala.collection.immutable.Map[Int,java.lang.String]
= Map(9 -> nine, 8 -> eight)
```

Immutability V

- Referential transparency
  - Don't rebind symbols/names (bind fn results to new symbols)
  - Any code that references a symbol (ex: `v1`) always sees same value
    - "Either it works (all the time) or it doesn't work at all" happens more often
- Structural sharing through persistent data structures
  - Any code creating a new value using `v1` reuses memory
    - EX: copying, appending, subsets, etc.

Immutability VI

- Value semantics
  - Clojure

```
(def v3 v1)
v1
v3
(= v1 v3)
(= v3 [5 6])
(def v4 [1 [2 [3]]])
(def v5 [2 [3]])
(second v4)
(= v5 (second v4))
```

  - Scala

```
val v3 = v1
v1
v3
v1 == v3
v3 == Vector(5,6)
val v4 = Vector(1, Vector(2, Vector(3)))
val v5 =  Vector(2, Vector(3))
```

Immutability VII

```
v5 == v4(1)
```

- Immutable values can be safely used in sets and in map keys
  - Whereas Java allows mutable objects in sets or map keys (unadvisable)
  - Python disallows mutable objectslists in sets or map keys
- In general, Clojure uniquely teases out
  - State as value + time, and...
  - Identity transcends time

Java, Ruby, Scala, & Clojure

aspect	Java	Ruby	Scala	Clojure
strong typing	Y	Y	Y	Y
dynamic typing	N	Y	N	Y
interpreter/REPL	N	Y	Y	Y
functional style	N	Y	Y	Y
"fun web prog."	N	Y	Y	Y
good for CLI script	N	Y	N	N
efficient with memory	Y	N	Y	Y
true multi-threaded	Y	N	Y	Y

Clojure ↔ Scala I

aspect	Clojure	Scala	why? (Clojure)
STM	yes	yes	does for concurrency what GC did for memory
OOP	not really	yes	“It is better to have 100 functions operate on one data structure than 10 functions on 10 data structures.”
design patterns	no	??	equivalent outcomes done in other ways
FP	yes	sort of	fns compose and can be used as arguments to other fns

Clojure ↔ Scala II

aspect	Clojure	Scala	why? (Clojure)
concurrency	yes	yes* (?)	Clojure designed for this from the beginning
persistent data structures	yes	yes	only reasonable way to support immutable data structures
sequence abstraction	yes	yes	fns on seqs : objects :: UNIX : DOS
syntax regularity	yes	sort of	nice for macros, readability (& pasting into REPL)

Clojure ↔ Scala III

aspect	Clojure	Scala	why? (Clojure)
language extensibility (macros)	yes	yes*	abstract repetitive code not possible via fns and patterns
backwards compatibility	yes	yes*	Clojure is relatively very good at working with old version code

Defining a Function

- Basic structure of a new fn

```
(defn fn-name
  "documentation string"
  [arg1 arg2]
  ;; return value is last form
  )
```

Defining a Function

- Basic structure of a new fn

```
(defn fn-name
  "documentation string"
  [arg1 arg2]
  ;; return value is last form
  )
```

- Enter the following (in Light Table, if possible):

```
(defn square
  [x]
  (* x x))
```

Defining a Function

- Basic structure of a new fn

```
(defn fn-name
  "documentation string"
  [arg1 arg2]
  ;; return value is last form
  )
```

- Enter the following (in Light Table, if possible):

```
(defn square
  [x]
  (* x x))
```

- Now enter:

```
(square 2)
```

Lexical scope - `let` I

- Can think of `let` form as giving "local variables"
  - Except they must all be declared at the beginning
- The `let` bindings also used to break up a nested form into something more readable
- Example: Let's find the solutions of a quadratic equation
  - For $ax^2 + bx + c = 0$, the solution is

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

  - Test case:

$$a = 1, b = -5, c = 6$$

$$\Rightarrow x^2 - 5x + 6 = 0$$

$$x = \{2, 3\}$$

Lexical scope - let II

- First pass:
```
(defn quadsolve
  "solve a quad eqn"
  [a b c]
  [(/ (+ (- b) (- (square b) (* 4 a c))) (*
2 a)) (/ (- (- b) (- (square b) (* 4 a c)))
(* 2 a))])
```
  - Check:
```
(quadsolve 1 -5 6)
```

Lexical scope - let III

- Define:

```
(defn discriminant
  "for a quadratic eqn's coefficients,
return the discriminant"
  [a b c]
  (- (square b) (* 4 a c)))
```

  - Check:

    ```
    (discriminant 1 -5 6)
    ```

- Rewrite:

```
(defn quadsolve
  [a b c]
  (let [disc (discriminant a b c)
        disc-sqrt (Math/sqrt disc)]
    [(/ (+ (- b) disc-sqrt) (* 2 a)) (/ (-
(- b) disc-sqrt) (* 2 a))]))
```

Lexical scope - `let` IV

- `Math/sqrt` refers to the `sqrt` static method of Java's `java.lang.Math`
- Check:
  `(quadsolve 1 -5 6)`

Control Flow - `if`, etc. I

- `if`
  - Takes a 3 expressions: a test, the "then", and the "else"
  - Note: test passes for all values except `false` and `nil`
    - This "truthiness" holds for everything built off of `if` - `when`, `and`, `or`, `if-not`, `when-not`, etc.
  - ```
    (if (< disc 0)
      (println "I don't like imaginary numbers!")
      [(/ (+ (- b) disc-sqrt) (* 2 a)) (/ (- (- b) disc-sqrt) (* 2 a))])
    ```
- `do`
  - Creates a form that evaluates/executes multiple forms inside it
    ```

Control Flow - `if`, etc. II

- Returns the value of the last form

```
(if (< disc 0)
  (println "I don't like imaginary numbers")
  (do
    (println "I like real numbers!")
    [(/ (+ (- b) disc-sqrt) (* 2 a)) (/ (-
(- b) disc-sqrt) (* 2 a))]))
```

- `when` is the same as `if`, but with `nil` as "else" and a `do` built in for "then"
- Both `and` and `or` do short-circuit evaluation

`map` & `reduce` I

- Where's my for loop??
  - Instead of dealing with index-based looping, you can apply higher-order functions
- `map` applies a fn on every element of a sequence
- `reduce` uses a fn to accumulate an answer
  - Apply fn on first 2 elements (or an initial value and first element)
  - Continue applying fn on accumulated value and next element

map & reduce II

```
user> (def data [3 5 9 1 5 4 2])
#'user/data
user> (map square data)
(9 25 81 1 25 16 4)
user> (reduce + data)
29
user> (defn sum-sq
        [nums]
        (reduce + (map square nums)))
#'user/sum-sq
user> (sum-sq data)
161
```

map & reduce III

- Since Clojure fns are first-class citizens
  - You can have a vector of fns: `[+ -]`
  - You can have an anonymous fn (doesn't have a name): `(fn [x] (if (pos?  x) x (- x)))`
- Our next rewrite of quadsolve:

```
(defn quadsolve
  [a b c]
  (let [disc (discriminant a b c)
        disc-sqrt (Math/sqrt disc)
        soln-fn (fn [op] (/ (op (- b)
disc-sqrt) (* 2 a)))
        ops [+ -]]
    (map soln-fn ops)))
```

Closures

- `soln-fn` is a closure – the values of `a` and `b` are pulled from surrounding scope
- Even if `soln-fn` is passed elsewhere, the values of `a` and `b` in `soln-fn` don't change after fn creation & binding
  - fns ⇒ values ⇒ immutable
- Ex: you have to decrypt a lot of strings encrypted with the same public key
  - Instead of repeated `(decrypt priv-key s ...)` calls

```
(defn decrypt-with-priv
  [priv-key]
  (fn [s]
    (decrypt priv-key s)))

(let [my-decrypt (decrypt-with-priv
priv-key)]
  (my-decrypt s1)
  (my-decrypt s2)
  ...)
```

  - In many cases, as above, `partial` does the same

Java Interop

- Java classes in JVM and classpath accessible
  - Use full name unless imported, ex: `(import 'java.net.URL)`
  - All of `java.lang.*` always imported, just like Java
- New objects through `new`: `(new URL "http://clojure.org")`
  - Syntax shorcut: `(URL. "http://clojure.org")`
- Static methods called through `Class/method` (ex: `Math/sqrt`)
- Idiomatic member method call ex: `(.toLowerCase "sUpEr UgLy CaSiNg")`
- More (& interesting) Java interop available (ex: `proxy`, `memfn`, etc.)
- Clojure way for Java patterns very neat (multimethods, protocols, records, types)

Sequence/List Processing Functions I

- Many useful fns exist to transform sequences, work on specific collection types, or convert from one to another
- Examples:

```
user> (filter even?  data)
(4 2)
user> (remove even?  data)
(3 5 9 1 5)
user> (take 3 data)
(3 5 9)
user> (drop 3 data)
(1 5 4 2)
user> (first data)
3
user> (rest data)
(5 9 1 5 4 2)
user> (last data)
```

Sequence/List Processing Functions II

```
2
user> (butlast data)
(3 5 9 1 5 4)
user> (take-while (fn [x] (< 1 x)) data)
(3 5 9)
user> (drop-while (fn [x] (< 1 x)) data)
(1 5 4 2)
user> (take-nth 2 data)
(3 9 5 2)
```

Sequence/List Processing Functions III

```
user> (def nums [1 1 1 2 1 1 2 1 1 1 1 1 2 2
1 3 1 2 2 1 1])
#'user/nums
user> (frequencies nums)
{1 13, 2 6, 3 1}
user> (group-by odd? nums)
{true [1 1 1 1 1 1 1 1 1 1 3 1 1 1], false
[2 2 2 2 2 2]}
user> (partition-by even? nums)
((1 1) (2) (1 1) (2) (1 1 1 1 1) (2 2) (1 3
1) (2 2) (1 1))
```

Adding/Removing/Getting single elements

- `cons` puts an element at the front and returns a sequence
- `conj` adds an element in the most efficient manner and preserves the collection/sequence type

```
user> (cons 12 data)
(12 3 5 9 1 5 4 2)
user> (conj data 12)
[3 5 9 1 5 4 2 12]
user> (cons 12 s)
(12 1 2 3)
user> (conj s 12)
#{1 2 3 12}
```

- `assoc` (for maps) adds a key and its value, `dissoc` removes a key and value given a key
- `disj` is the opposite of `conj` for a set

apply - unpacking sequences in fn calls

- Some fns are meant for scalar args, not sequences:

```
user> (max 3 8 9 5 -1 4 1 6)
9
user> (max [3 8 9 5 -1 4 1 6])
[3 8 9 5 -1 4 1 6]
```

- When what you want comes as a sequence:

```
user> (max (filter odd?  [3 8 9 5 -1 4 1
6]))
(3 9 5 -1 1)
```

- Use apply to “unpack” the sequence and apply the fn:

```
user> (apply max (filter odd?  [3 8 9 5 -1 4
1 6]))
9
```

Macros I

- Powerful pre-evaluation step
- A fn that transforms code (input and output is code)
- Only possible when language's code written in language's data structures
  - Changing a language to accept code in its own data structures $\Rightarrow$ Lisp

Macros II

- Threading macros (-> and ->>)
  - Write nested forms "inside out" (more readable)
  - -> puts result of previous form in 2nd position of next
  - ->> puts result of previous form in last position of next
- Our previous sum of squares example
  - Before

    ```
    (reduce + (map square nums))
    ```
  - After

    ```
    (->> nums
         (map square)
         (reduce +))
    ```
- Our previous teaser # 4 example
  - Before

    ```
    (take-nth 3 (rest (line-seq br)))
    ```
  - After

    ```
    (->> br
         line-seq
         rest
         (take-nth 3))
    ```

Macros III

- Example with ->
  - Setup
    ```
    (require '[clojure.string :as string])
    (def line "col1\tcol2\tcol3\tcol4"))
    ```
  - Before
    ```
    (Integer/parseInt (.substring (second
    (string/split line #"\t")) 3))
    ```
  - After
    ```
    (-> line
        (string/split #"\t")
        second
        (.substring 3)
        (Integer/parseInt))
    ```
- Nested `nil` checks

Macros IV

- Before

```
(fn [n]
  (when-let [nth-elem (get ["http://g.co"
"http://t.co"] n)]
    (when-let [fl (get nth-elem 7)]
      (get #{\g \t \f} fl))))
```

- After

```
(fn [n]
  (some-> ["http://g.co" "http://t.co"]
          (get n)
          (get 7)
          (#{\ \t \f})))
```

Macros V

- Don't create your own macros unless you have to
  - Can't compose like fns ($\Leftrightarrow$ can't take value of macro)
  - Macros harder to debug
- Macros can (and/or should) be used in a few cases, including:
  - Abstracting repetitive code where fns can't (ex: patterns)
    - Or even for simplifying control flow, if common enough
  - Creating a DSL on top of domain-relevant fns
  - Controlling when a form is evaluted
- Macros allow individuals to add on to their language
  - `with-open`
    - ...is a macro in Clojure
    - "Imported" into Python, but can only be done as language syntax (only by language maintainers)
  - The `some->` macro
    - (officially added in Clojure 1.5)
    - already functionally existed in contrib library as `-?>`

Macros VI

- Most of Clojure is implemented as fns and macros
  - A few "special form" fns exist as elemental building blocks
  - Rest of language is composed of previously-defined fns and macros
  - Syntax is simple and doesn't change
  - New lang. versions mostly just add fns, macros, etc. $\Rightarrow$ backwards-compatibility

High-level Design Decision Cascade

- Simplicity → isolate state
- Simplicity → immutability
- Concurrency → immutability
- Concurrency → STM
- Functional programming → immutability
- Immutability → persistant data structures
- Functional programming → referential transparency

Effects of Decisions

- Lisp
  - Flexible syntax
  - Less parentheses + brackets + etc. (!)
  - Macros
- Functional programming
  - Simpler code
  - Easier to reason about
  - Places of mutation minimized, isolated
  - Refential transparency elsewhere
  - Design patterns handled in simpler, more powerful ways

My Parting Message to You

- The basics are simple, but tremendous depth
- May take time at first (initial investment), but simpler code is perpetual payoff
- Clojure/Lisp compared to other languages
  - Lisp helps you get better at programming (even if you don't use it)
  - Not a better vs. worse
  - But maybe a powerful vs. more powerful
    - If we agree that two languages can differ in power (ex: Perl vs. Basic)
  - Tradeoffs exist – always choose right tool for the job
    - Ex: a language's power may cost performance
  - Language discussions → emotional arguments b/c of proximity to mind & identity
    - Or so wrote Paul Graham - "Keep Your Identity Small" (& Paul Buchheit - "I am Nothing")
- Keep exploring
  - There are more cool aspects to Clojure I couldn't fit here
  - And it's still a young language

Abridged Set of Useful Resources

- Videos of Easy-to-follow Lectures by Rich Hickey
  - Data structures (Pt. 1 & Pt. 2); Sequences; Concurrency
  - Clojure for Java Programmers (Pt. 1 & Pt. 2)
  - Clojure for Lisp Programmers (Pt. 1 & Pt. 2)
- Books (my recommendations)
  - The Joy of Clojure - good intro that explains the 'why' of Clojure
  - Clojure Programming - deeper, more comprehensive guide to Clojure for all levels
- ClojureDocs
- Clojure Cheatsheet
- 4Clojure
  - Getting through the first 100 is worth the challenge to get better
  - I learned a lot by following these users' solutions: 0x89, _pcl, austintaylor, jbear, maximental, nikelandjelo, jfacorro, jsmith145, chouser, cgrand

The End

- Thanks!