Clojure

"The Art of Abstraction"

Alex Miller
Revelytix

Why are We Here?

Why are We Here?

I think abstraction is central to what we do as programmers.

Why are We Here?

I think abstraction is central to what we do as programmers.

I think Clojure is a great language for creating abstractions.

What is abstraction?

"Abstraction is the **elimination** of the irrelevant and the **amplification** of the essential."

- Bob Martin

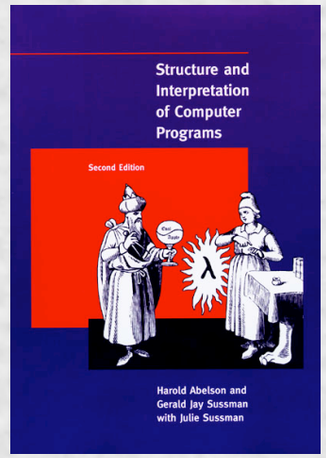
"I've been doing a lot of abstract painting lately... extremely abstract. No brush, no paint, no canvas. I just think about it."

- Steven Wright

Clojure

- A Lisp dialect on the JVM (and CLR)
- Dynamically typed
- Compiled

"When we **describe a language**, we should pay particular attention to the means that the language provides for **combining simple ideas to form more complex ideas**. Every powerful language has three mechanisms for accomplishing this:



1. **primitive expressions**,
   which represent the simplest entities the language is concerned with
2. **means of combination**,
   by which compound elements are built from simpler ones
3. **means of abstraction**,
   by which compound elements can be named and manipulated as units "

Structure and Interpretation of Computer Programs

- Abelson, Sussman, Sussman

Primitive Expressions

nil	`nil`
numbers	`1, 2.3, 22/7`
strings	`"abc"`
characters	`\a, \b, \space`
symbols	`math/fib`
keywords	`:bar`

Means of combination

2 3

Means of combination

+ 2 3

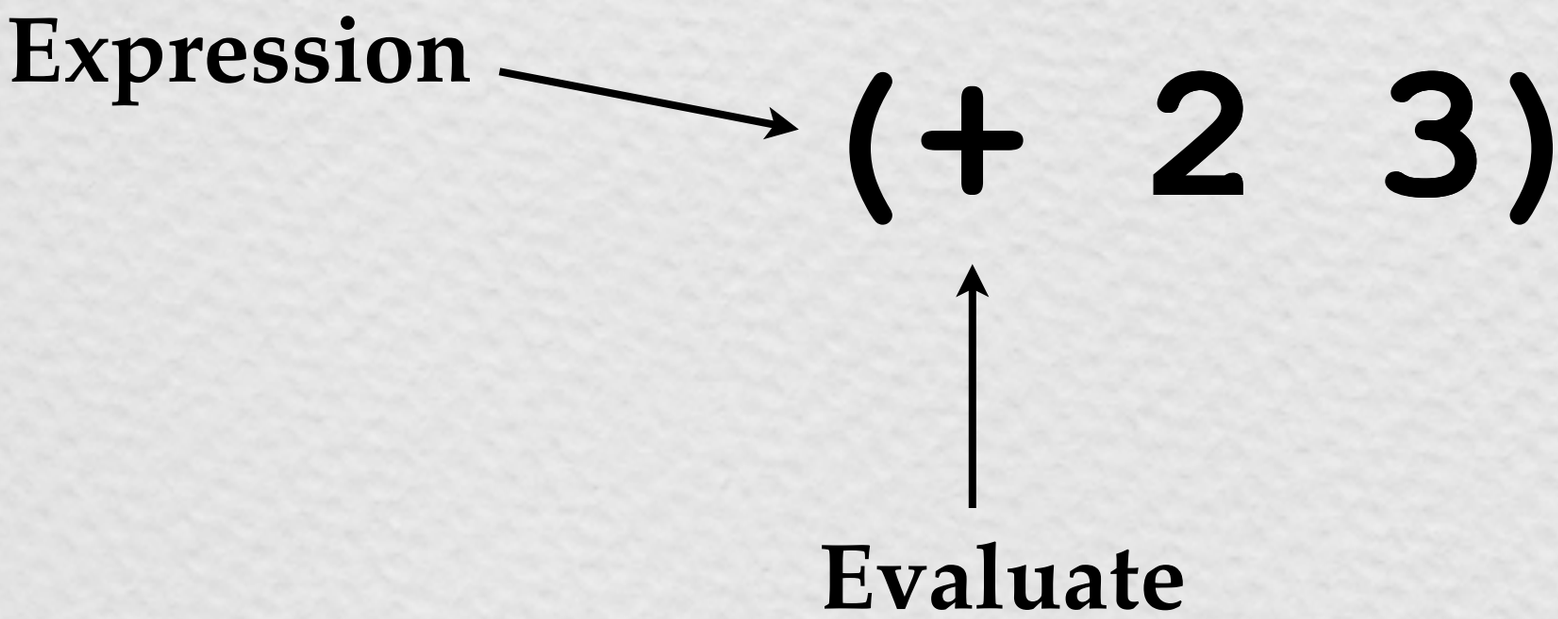
Means of combination

```
(+ 2 3)
```

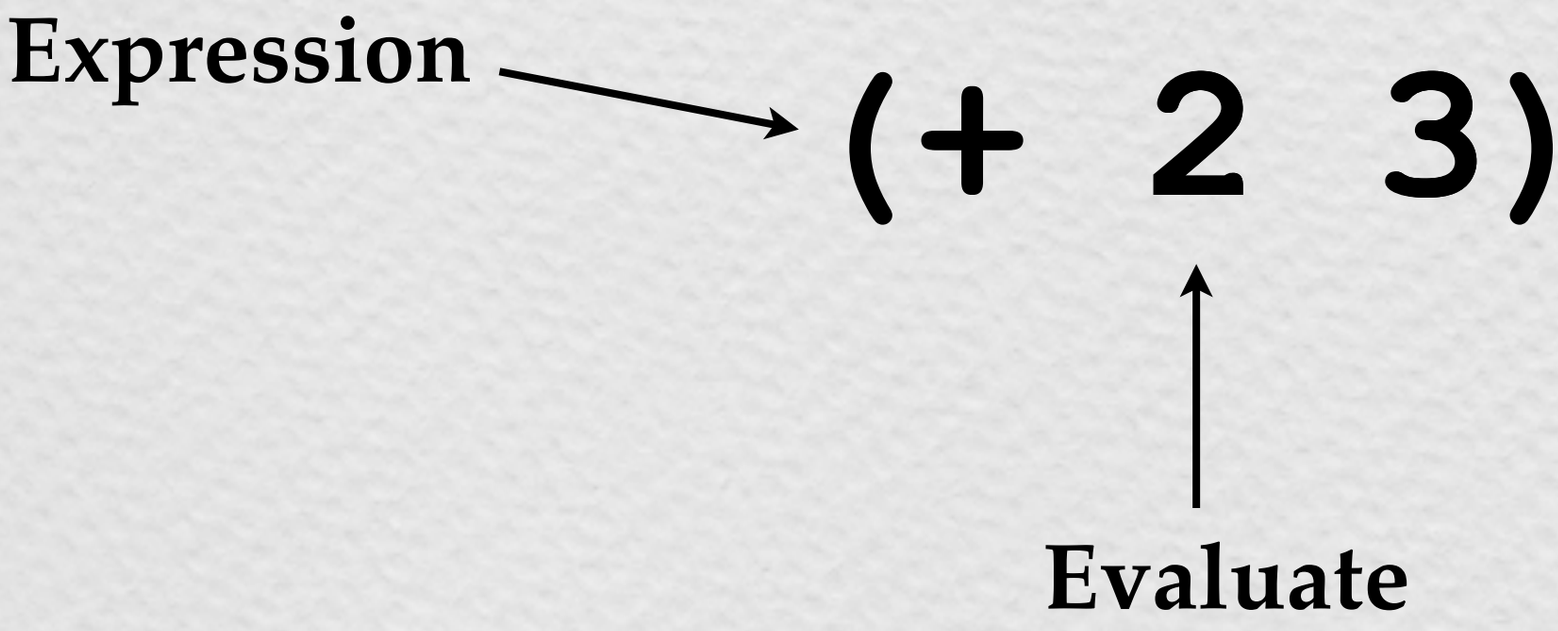
Means of combination

Expression → `(+ 2 3)`

Means of combination



Means of combination

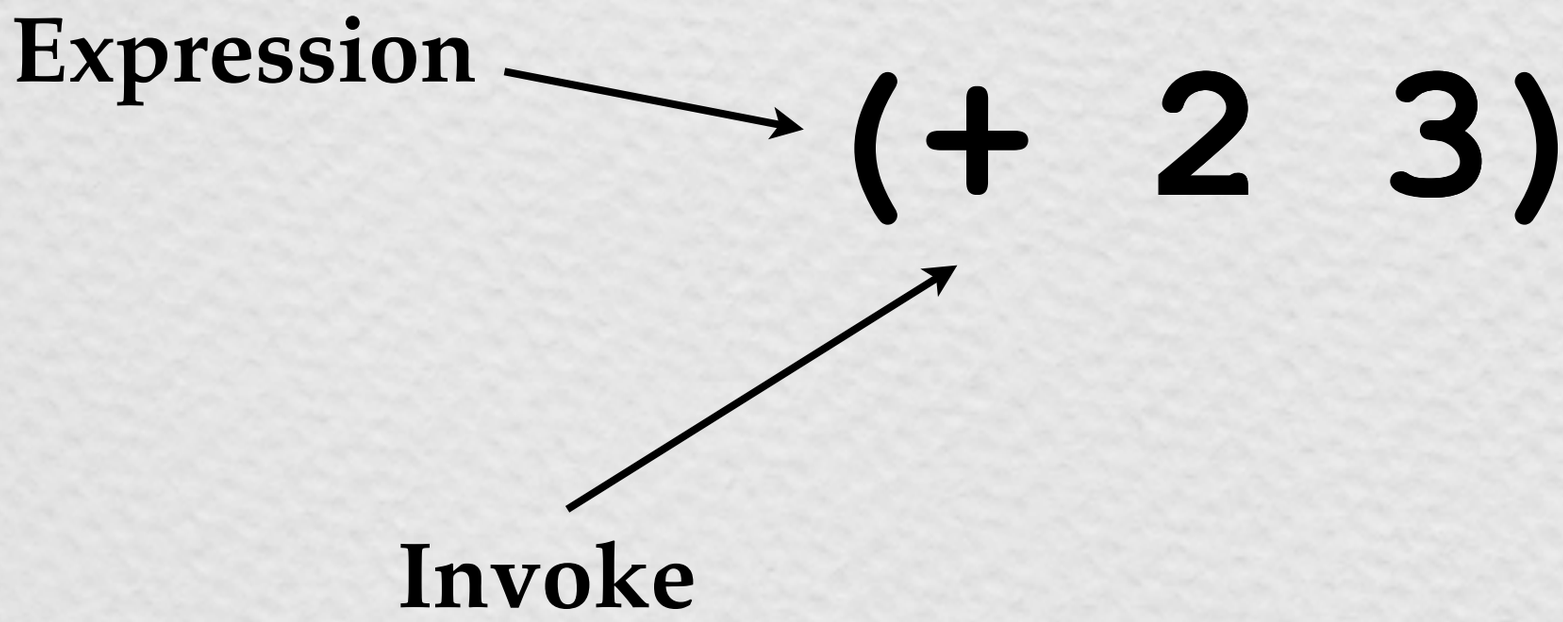


Means of combination

Expression → `(+ 2 3)`

↑ **Evaluate**

Means of combination



Means of abstraction

```
(fn [x] (* x x))
```

Means of abstraction

```
(def square (fn [x] (* x x)))
```

Means of abstraction

```
(def square (fn [x] (* x x)))
```

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

Abstracting with functions

Abstracting with functions

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

Abstracting with functions

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

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

Abstracting with functions

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

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

```
(defn exp      [x n]
  (apply * (repeat n x)))
```

Abstracting with functions

```
(defn exp [x n]
  (case n
        0 1
        1 x
        (* x (exp x (dec n)))))
```

```
(exp 2 3)
(* x (exp 2 2))
(* x (* x (exp 2 1)))
(* x (* x x))
```

Abstracting with functions

```
(defn exp [x n]
  (loop [total 1
         counter n]
    (if (= counter 0)
      total
      (recur (* x total) (dec counter)))))
```

Abstracting with functions

```
(defn exp [x n])
```

"It is better to have 100 functions operate on one data structure than to have 10 functions operate on 10 data structures."

– Alan J. Perlis

"Epigrams in Programming"

http://www.cs.yale.edu/quotes.html

Collections

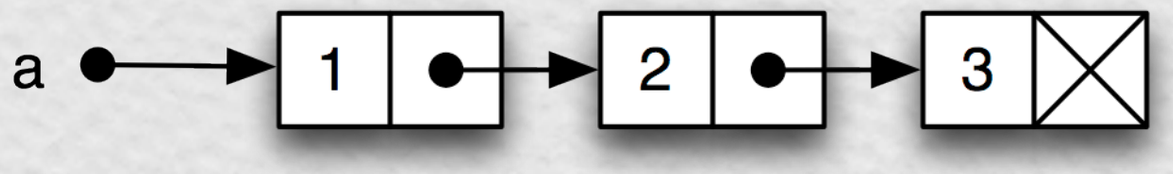
List	`(1 2 3)`
Vector	`[1 2 3]`
Set	`#{1 2 3}`
Map	`{:a 1 :b 2 :c 3}`

All data and collections are IMMUTABLE

All data and collections are IMMUTABLE

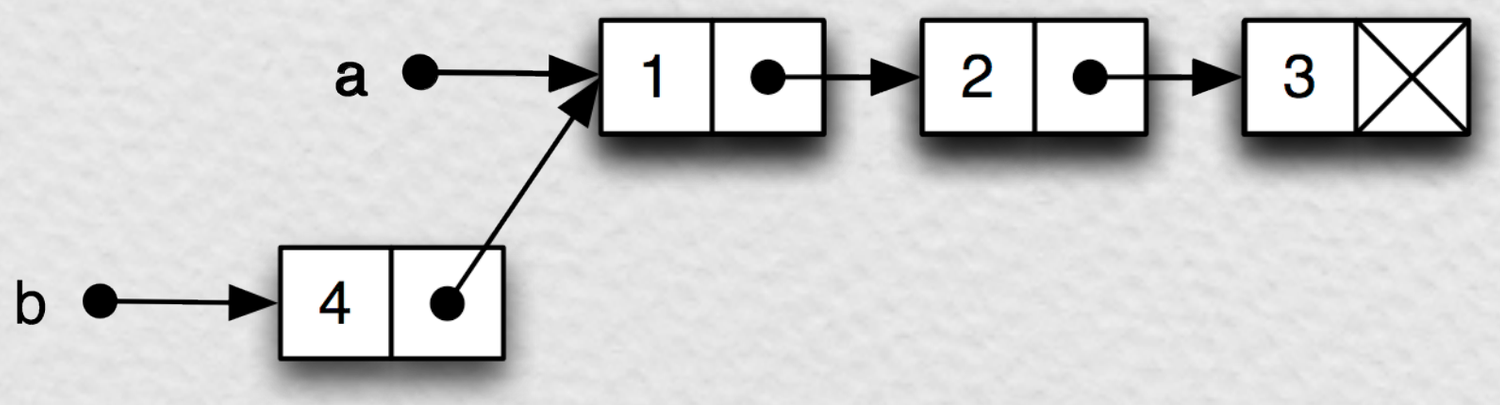
Structural sharing

```
(def a '(1 2 3))
```



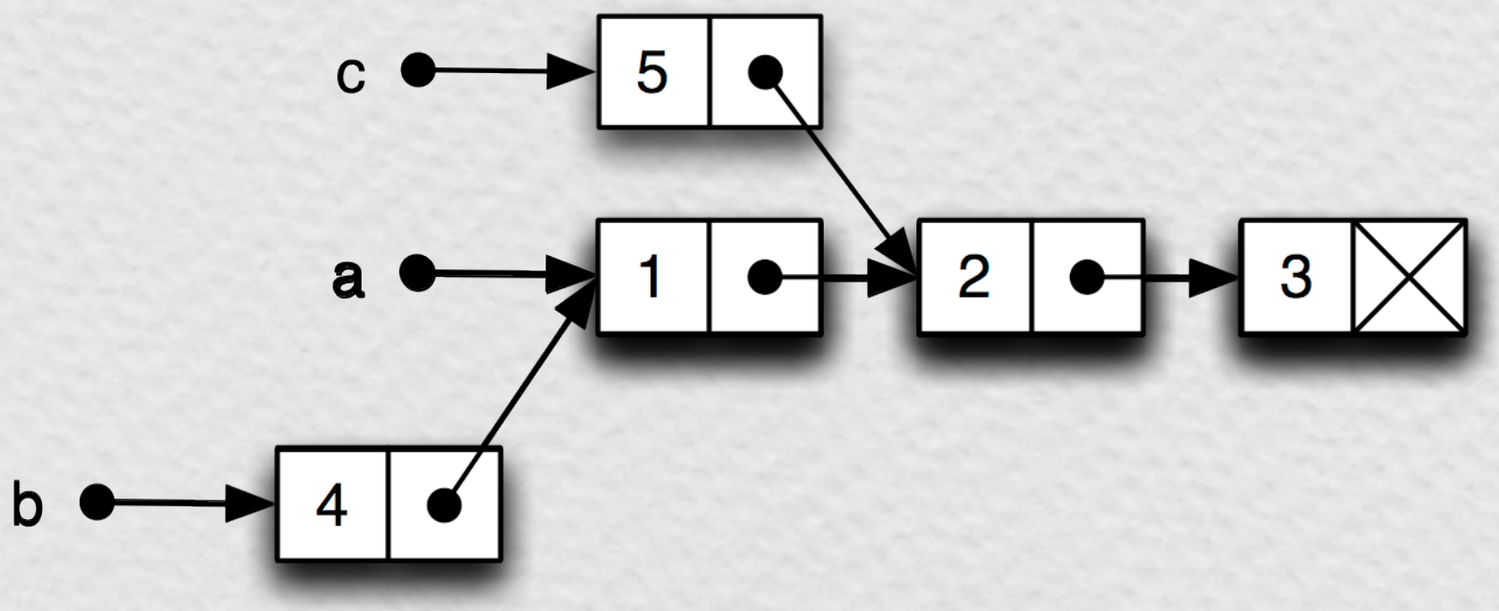
Structural sharing

```
(def a '(1 2 3))
(def b (cons 4 a))
```



Structural sharing

```
(def a '(1 2 3))
(def b (cons 4 a))
(def c (cons 5 (rest a)))
```



What do all collections share in common?

What do all collections share in common?

sequential traversal over values

What do all collections share in common?

sequential traversal over values

"seq"

Iterator Models

	Java Iterator	C# IEnumerator	Clojure seq
more?	hasNext	MoveNext	not null
get	next	Current	first
next	next	MoveNext	rest

*table stolen from Rich Hickey's talk on sequences

seq

vector	seq	first	rest
nil	nil	nil	()
	nil	nil	()
1 2 3 4 5	(1 2 3 4 5)	1	(2 3 4 5)

Seq and ye shall find...

- String
- Java Collections
- Java Iterators (iterator-seq, enumeration-seq)
- ResultSet (resultset-seq)
- Trees (tree-seq)
- XML (xml-seq)
- Lines of a file (line-seq)
- Files in a directory (file-seq)

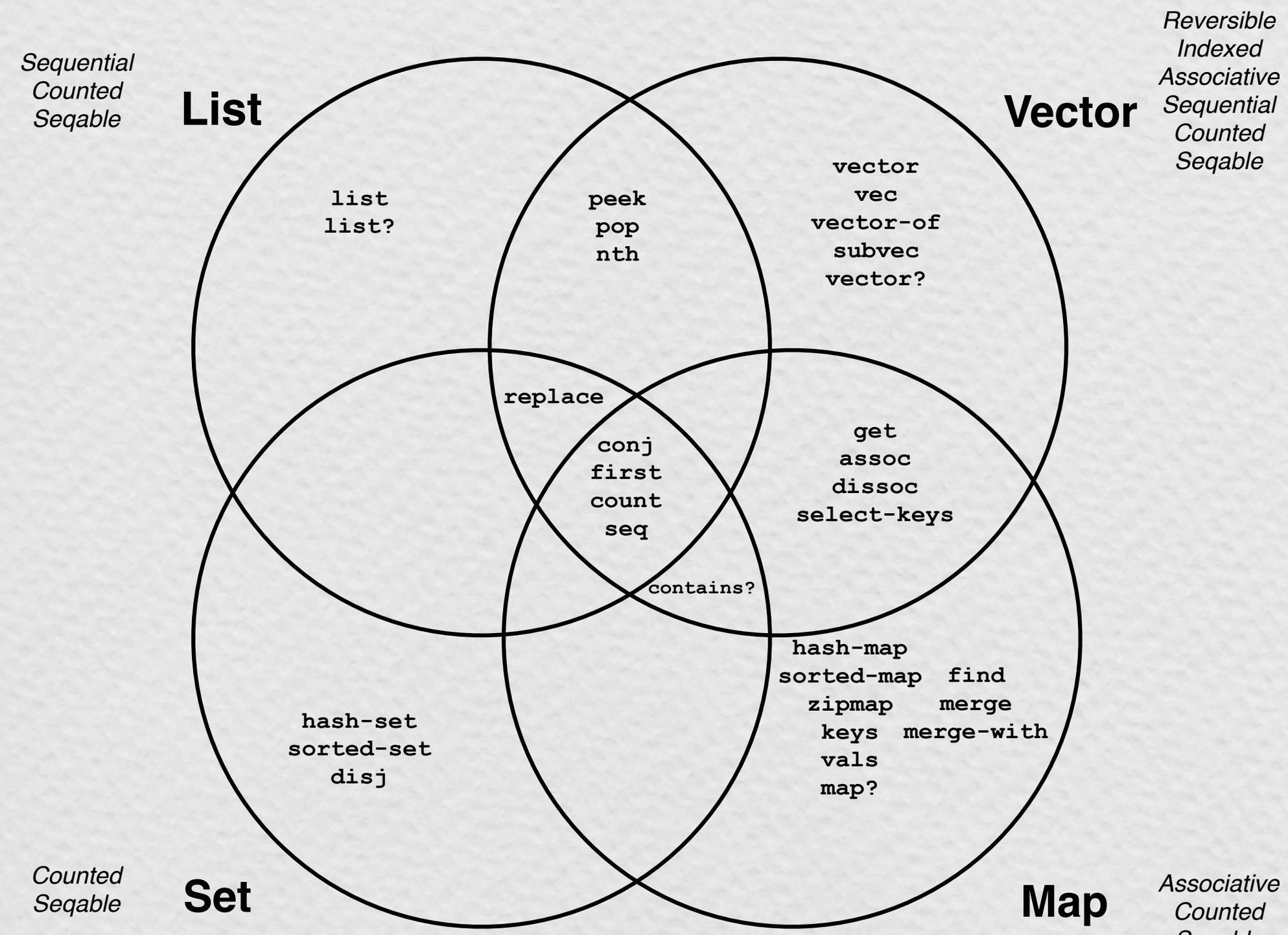
Lazy seqs

```
(take 10 (iterate inc 1))
```

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

Functions on Collections and Sequences

Collection Traits



Lists

```
(def a '(1 2 3))          #'user/a
(def b (cons 0 a))        #'user/b
(first b)                  0
(rest b)                   (1 2 3)
(count b)                  4
(nth b 1)                  1
```

Vectors

```
(def v [1 2 3])          #'user/v
(def w (conj v 4))       #'user/w
(nth w 3)                4
(get w 3)                4
```

Vectors are associative - indices are keys

Maps

```
(def m {:a 1 :b 2})          #'user/m
(def n (assoc m :c 3))       #'user/n
(keys n)                      (:c :a :b)
(vals n)                      (3 1 2)
(get m :a)                   1
(m :a)                       1
(:a m)                       1
```

Data as code

Sequence Functions

Seq in, Seq out

Shorter seq from a longer seq: distinct filter remove for keep keep-indexed

Longer seq from a shorter seq: cons concat lazy-cat mapcat cycle interleave interpose

Seq with head-items missing: rest next fnext nnext drop drop-while nthnext for

Seq with tail-items missing: take take-nth take-while butlast drop-last for

Rearrangment of a seq: flatten reverse sort sort-by shuffle

Create nested seqs: split-at split-with partition partition-all partition-by

Process each item of a seq to create a new seq: map pmap mapcat for replace reductions map-indexed seque

Using a seq

Extract a specific-numbered item from a seq: first ffirst nfirst second nth when-first last rand-nth

Construct a collection from a seq: zipmap into reduce set vec into-array to-array-2d frequencies group-by

Pass items of a seq as arguments to a function: apply

Compute a boolean from a seq: not-empty some reduce seq? every? not-every? not-any? empty?

Search a seq using a predicate: some filter

Force evaluation of lazy seqs: doseq dorun doall

Creating a seq

Lazy seq from collection: seq vals keys rseq subseq rsubseq

Lazy seq from producer function: lazy-seq repeatedly iterate

Lazy seq from constant: repeat replicate range

Lazy seq from other objects: line-seq resultset-seq re-seq tree-seq file-seq xml-seq iterator-seq enumeration-seq

Sequence Functions

```
(range 6)                          (0 1 2 3 4 5)
(filter odd? (range 6))            (1 3 5)
(reverse (range 4))                (3 2 1 0)
(partition 2 (range 4))            ((0 1) (2 3))
(map inc (range 5))                (1 2 3 4 5)
(reduce + (range 10))              45
```

Higher order functions

```
(defn mult [x] (fn [y] (* x y)))
#'user/mult

(def x10 (mult 10))
#'user/x10

(map x10 (range 5))
(0 10 20 30 40)
```

Functional Kingdom

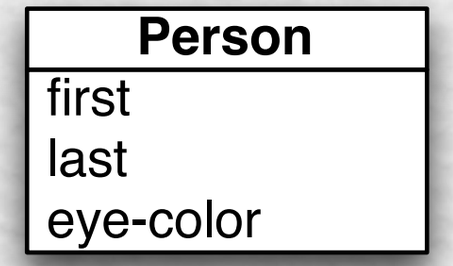
"In Javaland, by King Java's royal decree, **Verbs are owned by Nouns**."

"**In the Functional Kingdoms, Nouns and Verbs are generally considered equal-caste citizens.** However, the Nouns, being, well, nouns, mostly sit around doing nothing at all. They don't see much point in running or executing anything, because the Verbs are quite active and see to all that for them."

http://steve-yegge.blogspot.com/2006/03/execution-in-kingdom-of-nouns.html

Data types

```
(def alex
  {  :first "Alex"
     :last "Miller"
     :eye-color :blue })
(:last alex)
```




```
(defrecord Person [first last eye-color])
(def alex (Person. "Alex" "Miller" :blue))
(:last alex)
```

Polymorphism

Multimethods

```
(defrecord Programmer [language])
(defrecord NormalPerson [activity])

(defmulti relax class)
(defmethod relax Programmer [programmer]
  (println "I'm writing" (:language programmer)))
(defmethod relax NormalPerson [person]
  (println "I'm" (:activity person)))

(relax (Programmer. "Clojure"))
I'm writing Clojure
(relax (NormalPerson. "taking a walk"))
I'm taking a walk
```

Multimethods

```
(defrecord Programmer [language])

(defmulti quip :language)
(defmethod quip "Clojure" [programmer]
  (println "Running out of parens"))
(defmethod quip "Java" [programmer]
  (println "OOP rulez!"))

(relax (Programmer. "Clojure"))
Running out of parens
(relax (Programmer. "Java"))
OOP rulez!
```

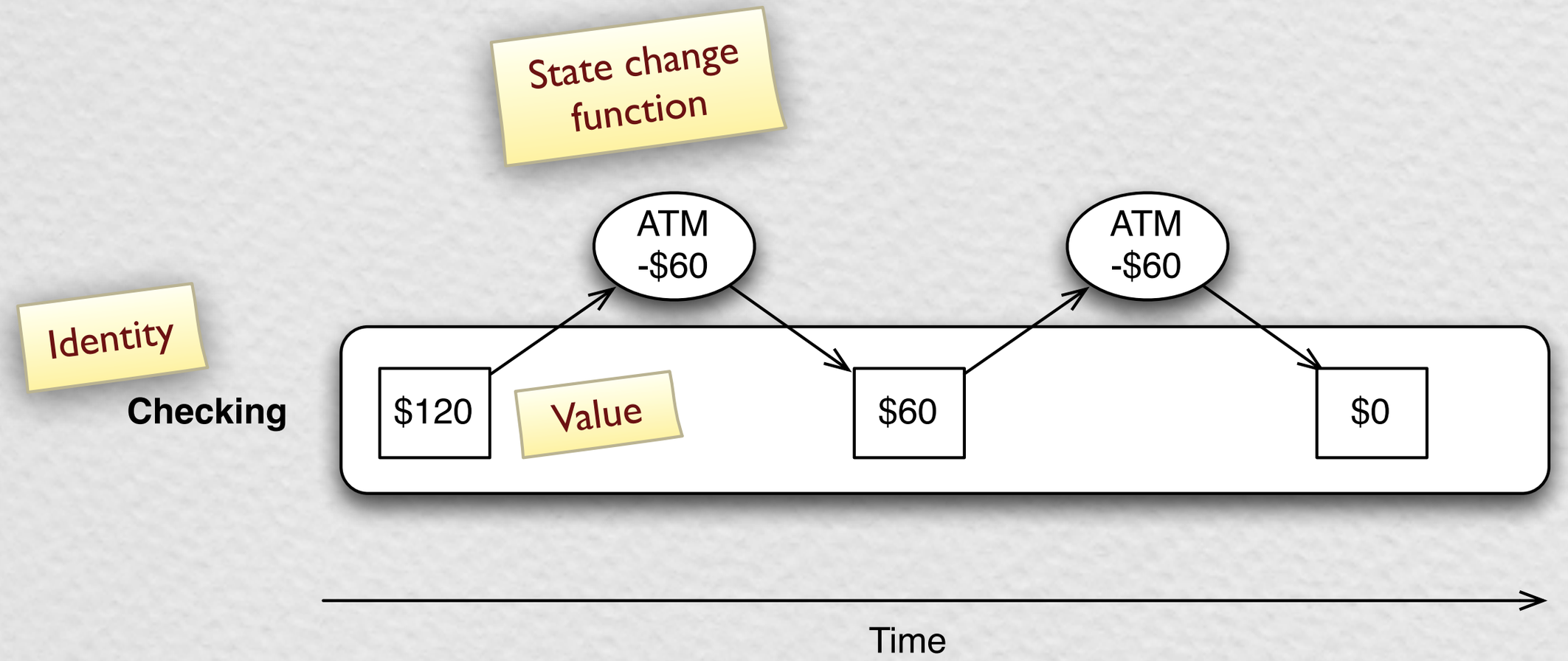
Protocols

```
(defrecord Programmer [language])

(defprotocol Teacher
  (teach [p])
  (read [p]))

(extend-type Programmer
  Teacher
    (teach [p] (println "Teaching" (:language p)))
    (read [p] (println "Reading Hacker News")))

(teach (Programmer. "Clojure"))
Teaching Clojure
```

State

- Immutable data is great!
- But how do I maintain state and coordinate changes?

Epochal Model of Time



State Constructs

- **Atoms** - uncoordinated synchronous change
  - Like Atomic classes
- **Refs** - coordinated synchronous change
  - STM to coordinate changes across refs
- **Agents** - coordinated asynchronous change
  - Like actors but not "active" and state always visible to anyone

Macros

```
(defmacro and
  ([] true)
  ([x] x)
  ([x & next]
   `(let [and# ~x]
      (if and#
        (and ~@next)
        and#))))
```

Macros

```
(defmacro and
  ([] true)
  ([x] x)
  ([x & next]
   `(let [and# ~x]
      (if and#
        (and ~@next)
        and#))))
```

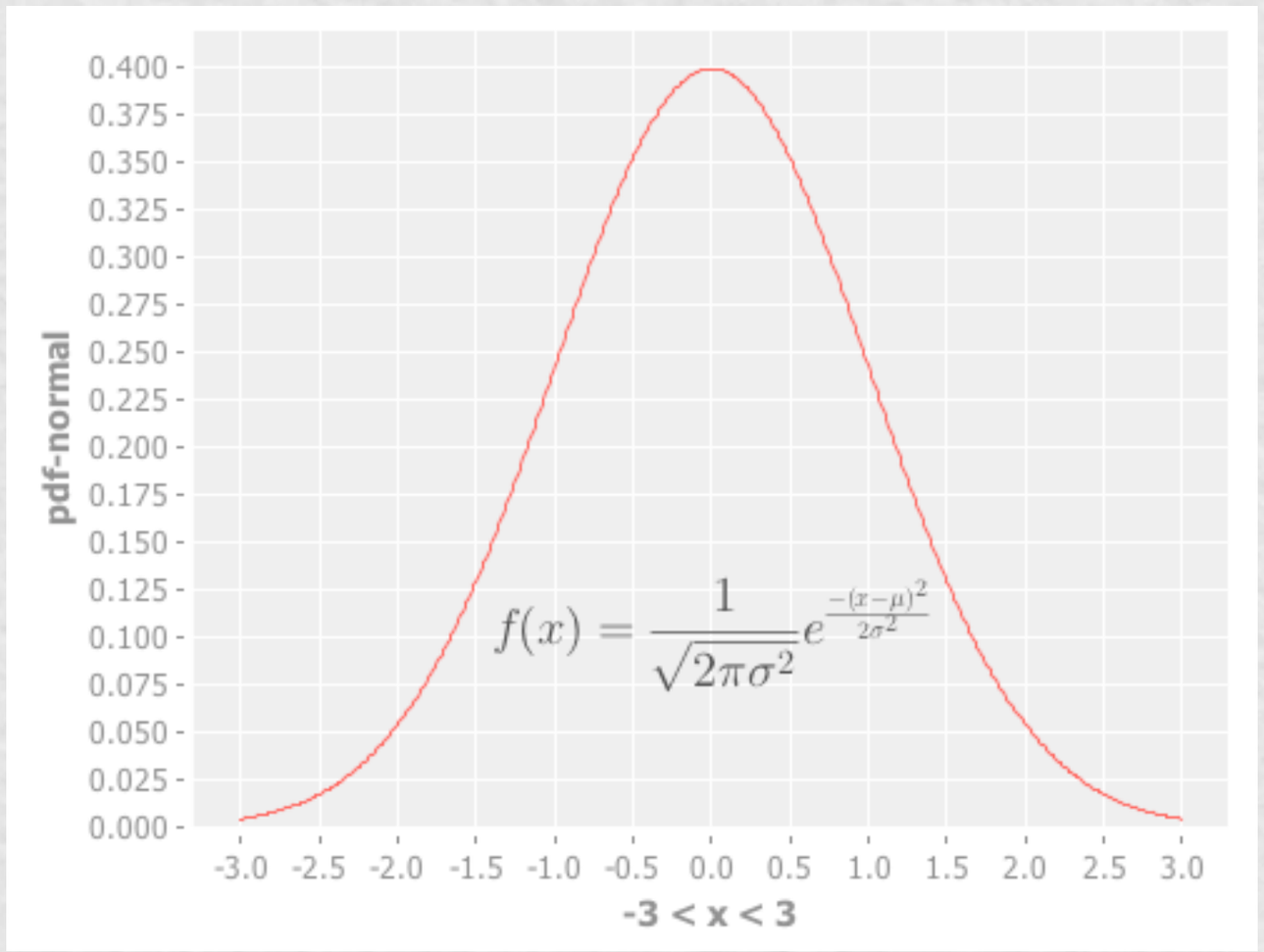
```
(and 1 2 3)
(let* [i 1]
  (if i
    (let* [j 2]
      (if j
        3
        j))
    i))
```

Abstractions

- functions (name and manipulate code)
- collections (trait-based, immutable collections of data)
- seq (logical lists of values)
- records (data types)
- multimethods, protocols (polymorphism)
- atoms, refs, agents (state)
- macros (syntax, order of evaluation)
- *namespaces* (modularity)
- *metadata* (out-of-band information passing)

Incanter

```
(doto (function-plot pdf-normal
        (add-latex 0 0.1 eq)
```



Ring

Http request

```
{:protocol       :http
 :request-method :get
 :uri            "/home"
 :server-name    "http://example.org"}
```

Http response

```
 {:status  200
  :headers {"Content-Type" "text/plain"
            "Content Length" 11}
  :body    "Rainbows and unicorns"}
```

Hiccup

```
(html
  [:head
    [:title "My home page"]]
  [:body
    [:h1 "Links"]
    [:p [:a {:href "http://tech.puredanger.com"}
                   "Alex's blog"]]])
```

Cascalog

- People in the Hadoop data set who are 25 years old

```
(?<- (stdout) [?person]
     (age ?person 25))
```

- Split sentences to words then count words

```
(?<- (stdout) [?word ?count]
     (sentence ?s)
     (split ?s :> ?word)
     (c/count ?count))
```

Greenspun's 10th Rule of Programming

10) Any sufficiently complicated C or Fortran program contains an ad hoc, informally-specified, bug-ridden, slow implementation of half of Common Lisp.

Corollaries

- Robert Morris' corollary: "…including Common Lisp."
- Norvig's corollary: "Any sufficiently complicated LISP program is going to contain a slow implementation of half of Prolog"

Snowclones

- Orange is the new black
- GOTO considered harmful
- Got milk ?
- I'm a doctor, not a bricklayer

Snowclones

- Bacon is the new black
- Inheritance considered harmful
- Got nachos ?
- I’m a doctor, not a programmer

Words for snow

"If Eskimos have ___ words for snow,
___ surely have ___ words for ___."

Words for snow

"If Eskimos have ___ words for snow, ___ surely have ___ words for ___."

"If Eskimos have 100 words for snow, programmers surely have 100 words for abstraction."

Sapir-Whorf hypothesis

Do the words we have available determine what we are able to think?

Do the abstractions in our language determine what we can program?

Thanks!

Blog: http://tech.puredanger.com

Twitter: @puredanger

Slides: http://slideshare.com/alexmiller/presentations

Strange Loop: http://thestrangeloop.com