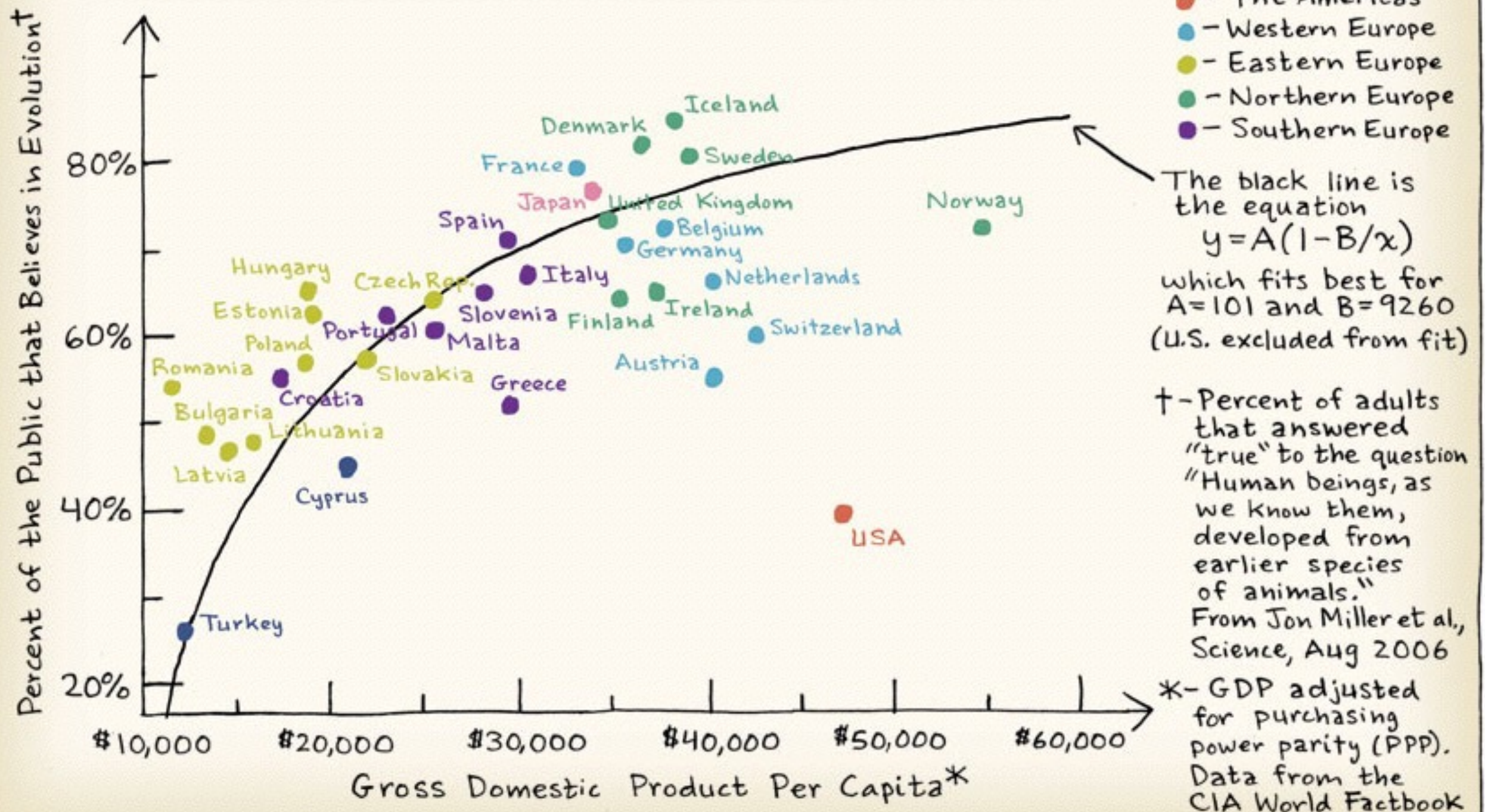


Functional Programming eXchange 2015

Functional Programming Principles

@reborg #fpX

Belief in Evolution Versus National Wealth



Agenda

- Just enough history and context
- Side effects and state implications
- The long list of benefits of pure functions
- Sparse examples in multiple languages

Expected take away

- FP is not academic stuff
- FP is for everyday programming tasks
- FP is easy (in the “simple made easy sense”)
- A bunch of concepts to further investigate



About



- Renzo Borgatti - Aka @reborg - Clojure Dev
- www.dailymail.co.uk one of the biggest clj-webapp
- Spent a few month in recent Haskell project
- **We're always hiring:** for serious FP and beyond!
- Follow my Clojure Weekly at <http://reborg.net>
- SICP-Mailonline study group organizer [Join Now!](#)

λ -calculus

- Alonzo Church ~1930
- Where most of FP concepts come from
- A notational tool for mathematical functions
- Can express all computable functions $\mathbf{IN} \Rightarrow \mathbf{IN}$
- Can be extended with a type system

λ -calculus had them all

- **Anonymous** (lambdas) and prefixed fns: $(\lambda x. * 3 x) 4$
- **Named fns**: $F =_{\text{def}} \lambda x. * 3 x$
- **Curried-everything**, multiple params fns: $\lambda y. \lambda x. * y x$
- **Higher order** fns: $T =_{\text{def}} \lambda f. (\lambda x. f (f (f x)))$
- **Recursive**: $G =_{\text{def}} \lambda n f x. \text{zero? } n x (G (\text{pred } n) f (f x))$
- **Beta reduction**: swap fns for their values
- **It's Turing complete!** (Church-Turing thesis)

λ

- Anonymous

- Named fns

- Curried-everything

- Higher order

- Recursive

- Beta reduction

**A BOAT-LOAD OF
TRICKY MATH**

λ -inspired programming

- First fact: it can't be pure!
- But still: emphasis on pure functions
- Isolate state changes
- Syntax is usually terse and minimal
- Make use of higher order functions
- Emphasis on symbols (instead numbers)

Side effecting function

A function with other observable “interactions” apart from its returned value

- Interactions: external state changes
- IO: display buffer, disk sectors, TCP packets...
- Also: vars, globals, properties, objects

Side effecting programming

Programming that makes use of state to model business logic

- State changes as an integral part of the paradigm
 - Function evaluation is dependent on time
 - Read/write of values is dependent on time
- > Exponential increase of incidental complexity

Incidental complexity

Any complexity not related to the business problem being solved

- Unnecessary data abstractions
- Duplication (textual or logical)
- Code volume (the amount of lines to write)
- Shared state
- Control (as in Imperative VS Declarative)

It's all about “Incidental”

- Informal reasoning becomes difficult
- Problematic to test
- Often pushing to lock-based concurrency (for state)
- Object identity problem (for OOP)
- Boilerplate which brings repetitive tasks

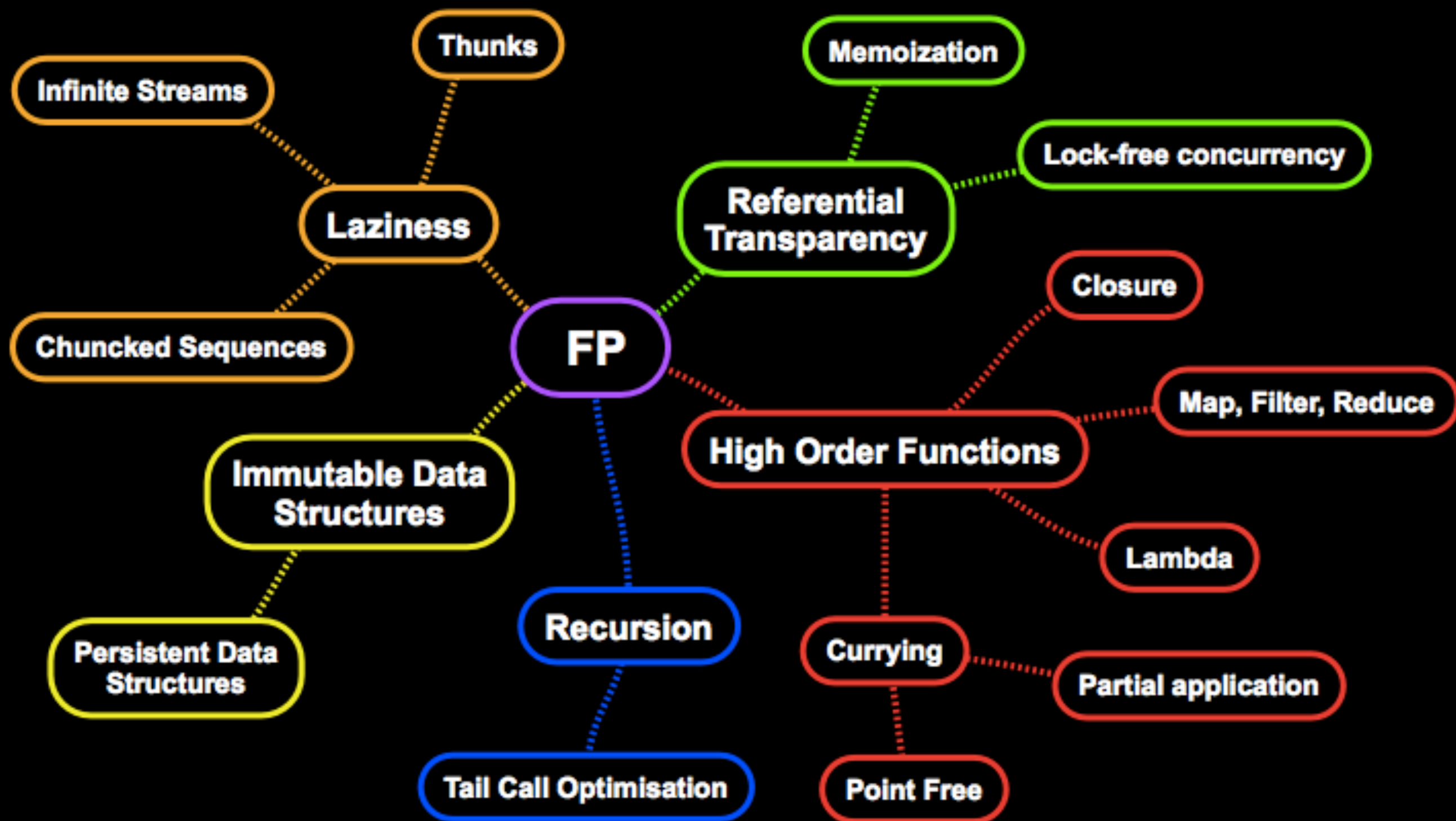
Pure functions facts

- Output can only be influenced by parameters
- Results don't depend on order of evaluation
- Compilers can evaluate the function at any time
- Same output is returned for the same parameters
- Changes to params (if any) are invisible outside

More formally

- Laziness (from removed temporal dependency)
- Immutable data-structures (for parameters)
- Memoization (referential transparency)
- Parallelism (removed context dependency)

λ -map of functional concepts



Lottery Example

- Simple app to show some FP principles
- Given a list of tickets and prizes, draw winners
- Drawing strategy is selected at runtime
- Random ticket number generator

High Order Functions (Java 1/3)

```
public static  
List<String> results (List<String> tickets, int totalPrize) {  
    HashMap<String, Integer> winners = draw(lotteryStrategy(totalPrize), tickets);  
    return display(winners);  
}
```



Passing down a function

High Order Functions (Java 2/3)

Accepting and invoking the function

```
private static
HashMap<String, Integer> draw
    (Function<List<String>, HashMap<String, Integer>> algorithm,
List<String> tickets) {
    List<String> take3 = tickets.stream().limit(3).collect(Collectors.toList());
    return algorithm.apply(take3);
}

private static
List<String> display
    (HashMap<String, Integer> winners) {
    List<String> results = new ArrayList();
    final int[] i = {0}; // trick to have an increasing index in the lambda.
    Stream<Map.Entry<String, Integer>> st = winners.entrySet().stream();
    Comparator<Map.Entry<String, Integer>> cmp =
Collections.reverseOrder(Comparator.comparing(e -> e.getValue()));
    st.sorted(cmp).forEach((e) -> results.add(
        String.format("winner %s: [%s %s]\n", ++i[0], e.getKey(),
e.getValue())));
    return results;
}
```

High Order Functions (Java 3/3)

```
private static
HashMap<String, Integer> zipmap
    (List<String> tickets, int[] prizes) {
    HashMap<String, Integer> results = new HashMap();
    for (int i = 0; i < prizes.length; i++) {
        results.put(tickets.get(i), prizes[i]);
    }
    return results;
}
```

```
private static
Function<List<String>, HashMap<String, Integer>> lotteryStrategy
    (int totalPrize) {
    Function<List<String>, HashMap<String, Integer>> result;
    if (totalPrize > 100) {
        result = (tickets) -> zipmap(tickets, new int[]{20, 30, 50});
    } else {
        result = (tickets) -> zipmap(tickets, new int[]{10, 10, 10});
    }
    return result;
}
```



Returning a function

High Order Functions (Hs)

```
type Winner = (String, Int)
```

```
lotteryStrategy :: Int -> [String] -> [Winner]
```

```
lotteryStrategy totalPrize
```

```
  | totalPrize > 100 = \tickets -> zip tickets [50, 30, 20]
```

```
  | otherwise = \tickets -> zip tickets [10, 10, 10]
```

Returning a function

```
display :: [Winner] -> [String]
```

```
display winners = map fmt indexedWinners
```

```
  where indexedWinners = zip ([1..] :: [Int]) winners
```

```
        fmt w = "winner " ++ show (fst w) ++ ": " ++ show (snd w) ++ "\n"
```

```
draw :: ([String] -> [Winner]) -> [String] -> [Winner]
```

```
draw lottery tickets = lottery $ take 3 tickets
```

Accepting and invoking the function

```
results :: [String] -> Int -> String
```

```
results tickets totalPrize =
```

```
  show $ display (draw (lotteryStrategy totalPrize) tickets)
```

passing down the function

High Order Functions (clj 1/1)

```
(def tickets ["QA123A3" "ZR2345Z" "GT4535A"])
```

```
(defn lottery-strategy [total-prize]
  (if (> total-prize 100)
    #(zipmap % [20 30 50])
    #(zipmap % [10 10 10])))
```

Returning a function

```
(defn display [winners]
  (map-indexed #(str "winner " (inc %1) ": " %2 "\n") winners))
```

```
(defn draw [lottery tickets]
  (lottery (take 3 tickets)))
```

Accepting and invoking the function

```
(defn results [tickets total-prize]
  (display (draw (lottery-strategy total-prize) tickets)))
```

Passing down a function

```
; (results tickets 200)
; winner 1: ["GT4535A" 50] winner 2: ["ZR2345Z" 30] winner 3: ["QA123A3" 20]
```


Laziness

```
(0..Float::INFINITY).lazy.select { |n| n.even? }.first(100)
```

```
(take 100 (filter even? (range)))
```

```
take 100 $ [0,2..]
```

```
IntStream.iterate(0, i -> i + 2).limit(100);
```

- No problem invoking **infinitely recursive fns**
- Evaluation can happen at any time with pure fns
- Pushes toward pipeline data processing

Infinite Streams (java 1/2)


```
public class InfiniteStream {  
  
    // [...] other functions  
  
    private static String nextTicket() {  
        String t = "%03d%s%06d";  
        return String.format(t, randInt(999), randString(2),  
randInt(999999));  
    }  
  
    public static Stream<String> ticketGen() {  
        return Stream.generate(() -> nextTicket());  
    }  
}
```



function invoking itself hidden
under the stream abstraction

Infinite Streams (java 2/2)

```
public class InfiniteStream {  
  
    private static final String alpha = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";  
  
    private static List<String> shuffle(String s) {  
        List<String> letters = asList(s.split(""));  
        Collections.shuffle(letters);  
        return letters;  
    }  
  
    private static String randString(int l) {  
        return shuffle(alpha).stream().limit(l).collect(joining());  
    }  
  
    public static int randInt(int max) {  
        return new Random().nextInt(max + 1);  
    }  
}
```



in-place mutation to shuffle them up

Infinite Streams (clj 1/1)

```
(def letters (map char (range 65 91)))
```

```
(defn rand-string [n]  
  (apply str (take n (shuffle letters))))
```

```
(defn next-ticket []  
  (format "%03d%s%06d"  
    (rand-int 999)  
    (rand-string 2)  
    (rand-int 999999)))
```

```
(defn ticket-gen []  
  (cons (next-ticket) (lazy-seq (ticket-gen))))
```

```
; example usage  
(take n (ticket-gen))
```

function invoking itself
made explicit

Infinite Streams (Hs 1/1)

```
randInt :: Int -> Int
randInt len = head $ take 1 $ randomRs (0, len) (mkStdGen len)

randString :: Int -> String
randString len = take len $ randomRs ('A', 'Z') (mkStdGen len)

nextTicket :: String
nextTicket =
    printf "%03d%s%06d" (randInt 999) (randString 2) (randInt 999999)

ticketGen :: [String]
ticketGen = nextTicket:ticketGen
```

infinite recursion lazy
by default

```
-- Not truly random (it needs different seeds):
-- take 3 ticketGen
```

How lazy can you get?

- Clojure is not strictly lazy
- Haskell is very lazy by default
- Java can also be lazy through Streams
- Call by value, call by need

Immutable data structures

```
{:a "a" :b "b"}
```

```
Collections.unmodifiableList(Arrays.asList(stuff))
```

```
[("a", 1), ("b", 2)]
```

- Can only be mutated by copy
- Plain copy tend to be inefficient
- Normally used with “persistence”

Persistent data structures

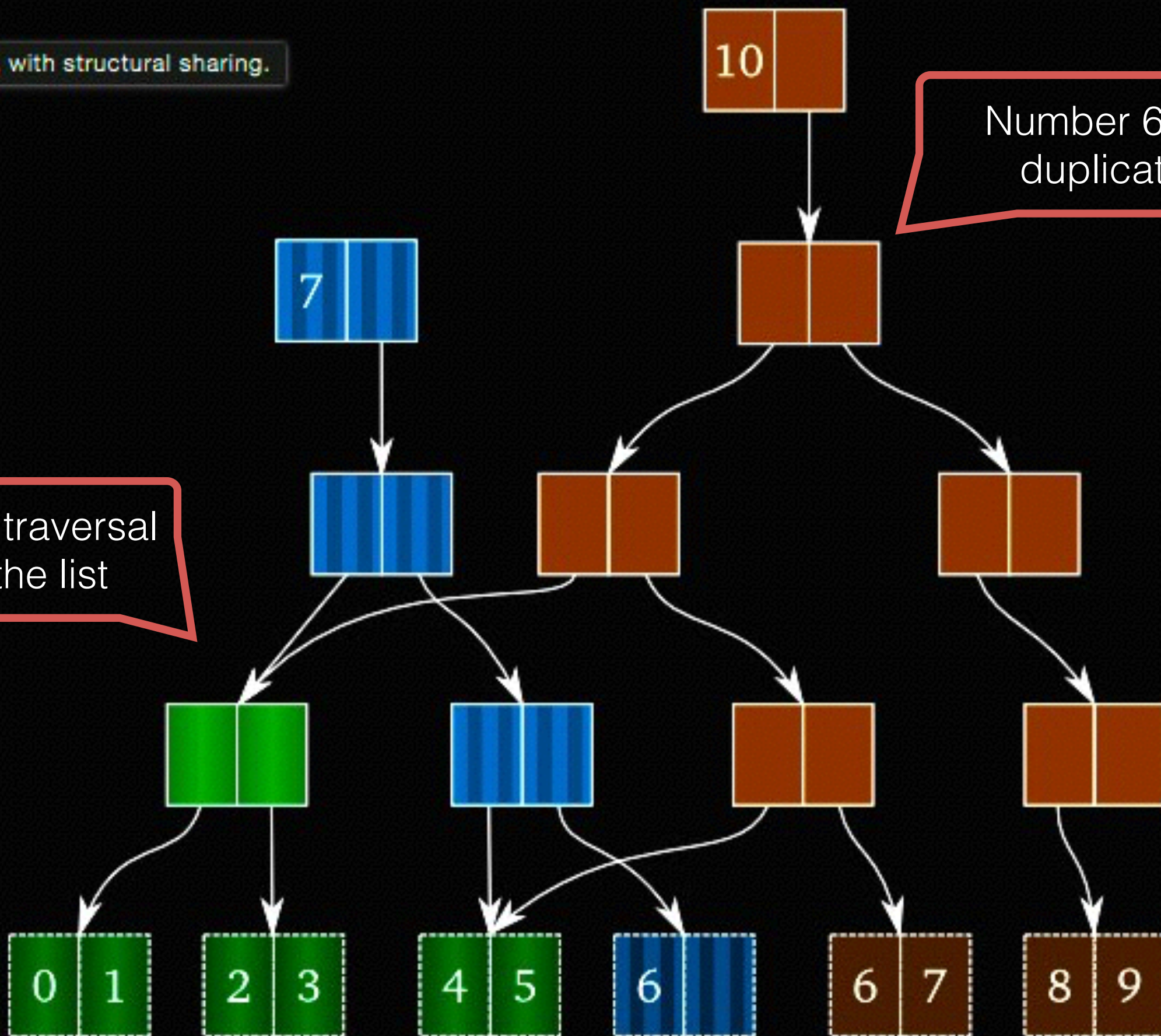
```
(assoc {1 "a" 2 "b"} 3 "c")  
PersistentHashMap.EMPTY.assoc(1, "a")  
Map.insert 1 "a" . Map.insert 2 "b"
```

- A mutable-like interface to immutable data
- Structural sharing is used instead of copy
- Almost linear search time ($\log_{32} N$ in clj)

Two vectors, with structural sharing.

Number 6 was duplicated

Depth-first traversal
to read the list



Memoization

- Memoization is the caching of a function call
- Parameters used as key
- Pure functions never expire!
- Immutability: no hashCode() or equals() required

Memoization (Java 1/1)

```
private static <T, U> Function<T, U> doMemoize(final Function<T, U> function) {  
    Map<T, U> cache = new ConcurrentHashMap<>();  
    return input -> cache.computeIfAbsent(input, function::apply);  
}
```

```
public static int bigSum(int n) {  
    return IntStream.rangeClosed(0, n).reduce(0, (x, y) -> x + y);  
}
```

Wrapped memoized
version

```
public static void main(String[] args) {  
    Function<Integer, Integer> memoBigSum = doMemoize(Memoize::bigSum);  
    System.out.println("memoBigSum 1 " + memoBigSum.apply(100));  
    System.out.println("memoBigSum 2 " + memoBigSum.apply(100));  
}
```

Memoization (Hs 1/1)

```
import Data.Function.Memoize (memoize)
```

```
bigSum :: Int -> Int  
bigSum n = sum [0..n]
```

```
bigSumMemo :: Int -> Int  
bigSumMemo = memoize bigSum
```

Wrapped memoized
version (memoized in
standard lib)

– pretty slow

```
bigSumMemo 10000000 `shouldBe` 4
```

– pretty fast 2nd time

```
bigSumMemo 10000000 `shouldBe` 4
```

Memoization (Clj 1/1)

```
(defn the-reduce [n]  
  (reduce + (range n)))
```

Wrapped memoized version
(memoize is in standard library)

```
(def big-reduce (memoize the-reduce))
```

```
; slow (big-reduce 10000000) => 49999995000000  
; fast (big-reduce 10000000) => 49999995000000
```

Currying

```
map (add 5) [1 2 3 4]  
(map (partial + 5) [1 2 3 4])
```

- Currying is a language feature
- Multiple args functions treated as single arg
- Explicit currying is called partial application
- Main effect is on expressiveness

Partial application (Clj)

```
(defn my-great-sum [a b]  
  (+ a b))
```

```
(defn unroll-1 [a]  
  (fn [b] (my-great-sum a b)))
```

```
(defn unroll-2 []  
  (fn [a] (fn [b] (my-great-sum a b)))))
```

```
(defn partial-unroll-1 [a]  
  (partial my-great-sum a))
```



Need for partial

```
(defn partial-unroll-2 []  
  (partial my-great-sum))
```

Partial application (java)

```
IntBinaryOperator myGreatSum = (a, b) -> a + b;  
IntFunction<IntUnaryOperator> unroll1 = (a) -> (b -> a + b);  
// () -> a -> b -> a + b; please help!  
  
myGreatSum.applyAsInt(4, 5);  
unroll1.apply(4).applyAsInt(5);
```

Currying (Hs)

```
myGreatSum :: Int -> Int -> Int  
myGreatSum a b = a + b
```

```
unroll1 :: Int -> (Int -> Int)  
unroll1 a = myGreatSum a
```

No need for partial

```
unroll2 :: (Int -> (Int -> Int))  
unroll2 = myGreatSum
```

With the extreme effect of
not needing unroll at all

Composition

```
((comp str inc) 1)  
show . succ $ 1
```

- A new function that composes other functions
- Practically: $y=g(x)$; $z=f(y)$; $z=f(g(x))$;
- Compositions can be named and re-used
- Not to be confused with $f(g(x))$ only

Parallel execution

```
(defn heavy [ms] (Thread/sleep (* 10000 ms)))
```

```
(time (doall (map heavy (repeat 3 (Math/random))))))  
"Elapsed time: 27994.376 msecs"
```

```
(time (doall (pmap heavy (repeat 3 (Math/random))))))  
"Elapsed time: 9687.184 msecs"
```

- Note the added “p” in “pmap”
- Taking care of thread pools and number of CPUs
- Alternative fork/join model with reducers

Other FP idioms

Exercise for the reader: pick your favourite language (with lambda support) and see how the following are implemented:

- List comprehensions
- Recursion
- Fixed point of a function
- Y-combinator
- Typed stuff: functors, applicative, monads



Did I mention we're hiring?

Come and speak to me :)

Also don't forget our [SICP Study Group](#)

Resources

- “**Out of the tar pit**” <http://shaffner.us/cs/papers/tarpit.pdf/>
- “**Simple Made Easy**” https://github.com/matthiasn/talk-transcripts/blob/master/Hickey_Rich/SimpleMadeEasy.md
- “**Structure and interpretation of computer programs**” <https://mitpress.mit.edu/sicp/full-text/book/book.html>
- “**SICP-Mailonline Study Group**” <https://groups.google.com/forum/?hl=en#!forum/sicp-mailonline>
- “**Understanding persistent vectors**” <http://hypirion.com/musings/understanding-persistent-vector-pt-1>
- “**A short introduction to Lambda Calculus**” www.cs.bham.ac.uk/~axj/pub/papers/lambda-calculus.pdf