JBCNConf 2015

Functional Programming with Clojure

and Java 8

…and a tiny bit of Haskell

@reborg

https://mitpress.mit.edu/sicp/

Agenda

- Just enough history and context
- Side effects and state implications
- The long list of benefits of pure functions
- Some detailed code
- Sparse examples

About

- Renzo Borgatti - Aka @reborg
- Clojure, formerly Ruby/Rails, formerly Java
- www.dailymail.co.uk one of the biggest clj-webapp
- Follow the Clojure Weekly at http://reborg.net
- SICP-Mailonline study group organizer (link)

λ-calculus

- Alonzo Church ~1930
- Where most of FP concepts come from
- A notational tool for mathematical functions
- Can express all computable functions $\mathbb{N} \Rightarrow \mathbb{N}$
- 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 =**def**= $\lambda x. * 3x$
- **Curried-everything**, multiple params fns: $\lambda y. \lambda x. * \ y \ x$
- **Higher order** fns: T=**def**= λf.(λx.f (f (f x)))
- **Recursive**: G =**def**= λn f x. zero? n x (G (pred n) f (f x))
- **Beta reduction**: referential transparency practical effect
- **It's Turing complete!** (Church-Turing thesis)

λ

- Anonymous
- Named fns
- Curried-everything
- Higher order
- Recursive
- Beta reduction
- It's Turing complete!

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

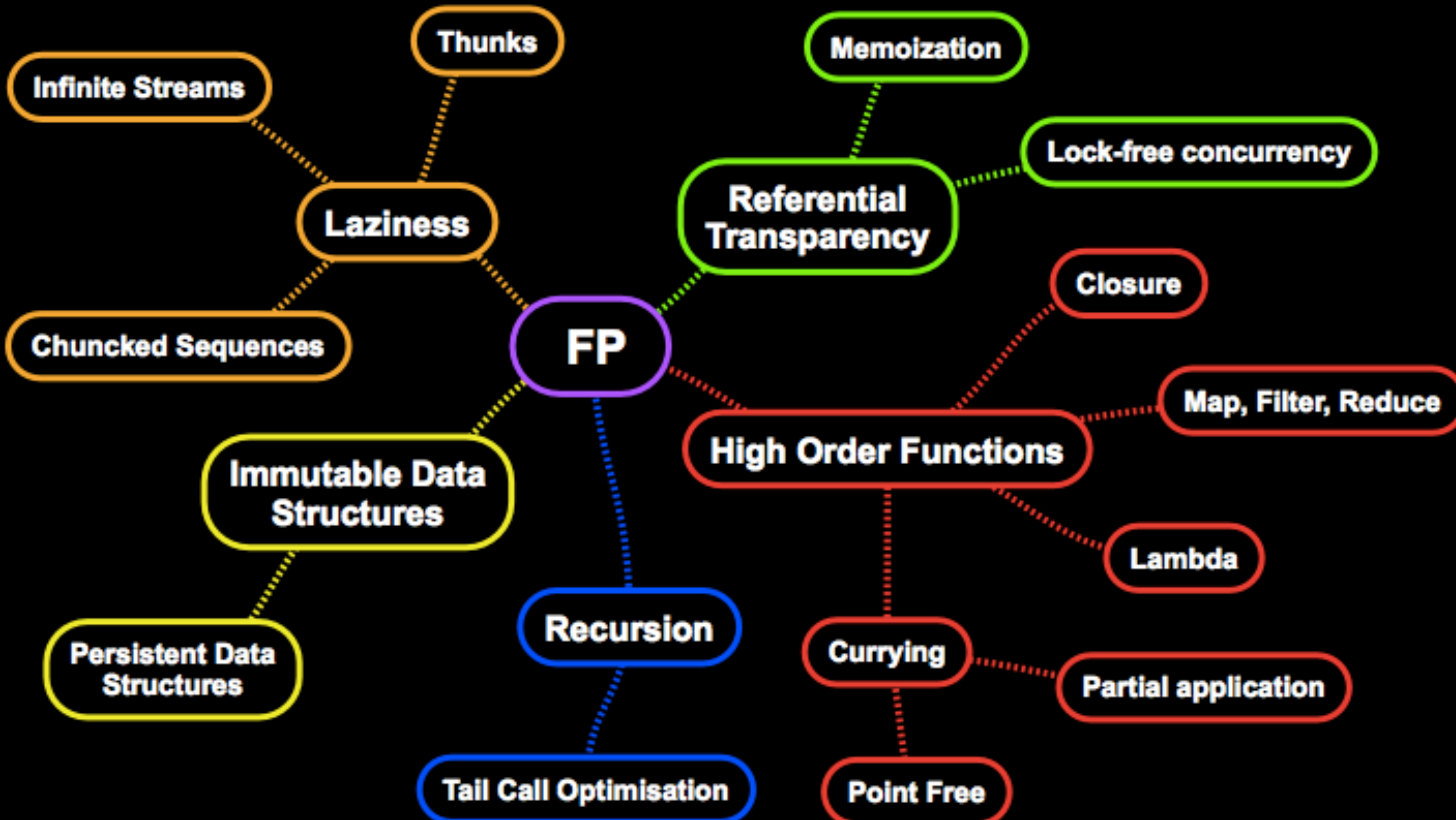
Pure functions facts

- 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



A word about the examples

- Lottery: 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
- .clj first, .java next, .hs for comparison

High Order Functions (clj 1/1)

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

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

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

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

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

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

Returning a function

Accepting and invoking the function

Passing down a function

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)

```java
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 (Java 3/3)

Accepting 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);
}
```

Invoking the function

```
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 (Hs)

```haskell
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]

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

results :: [String] -> Int -> String
results tickets totalPrize =
  show $ display $ draw (lotteryStrategy totalPrize) tickets
```

Returning a function

Accepting and invoking the function

passing down the function

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 (clj 1/1)

```clojure
(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
with no exit condition

lazy-seq makes
this possible

Infinite Streams (java 1/2)

```java
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());
  }
}
```

do {} while
hidden in the API

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);
    }
}
```

stream to pick l-length random string

Infinite Streams (Hs 1/1)

```haskell
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

-- But not truly random without Monads (it needs different seeds):
-- take 3 ticketGen
```

infinite recursion lazy by default

Immutable data structures

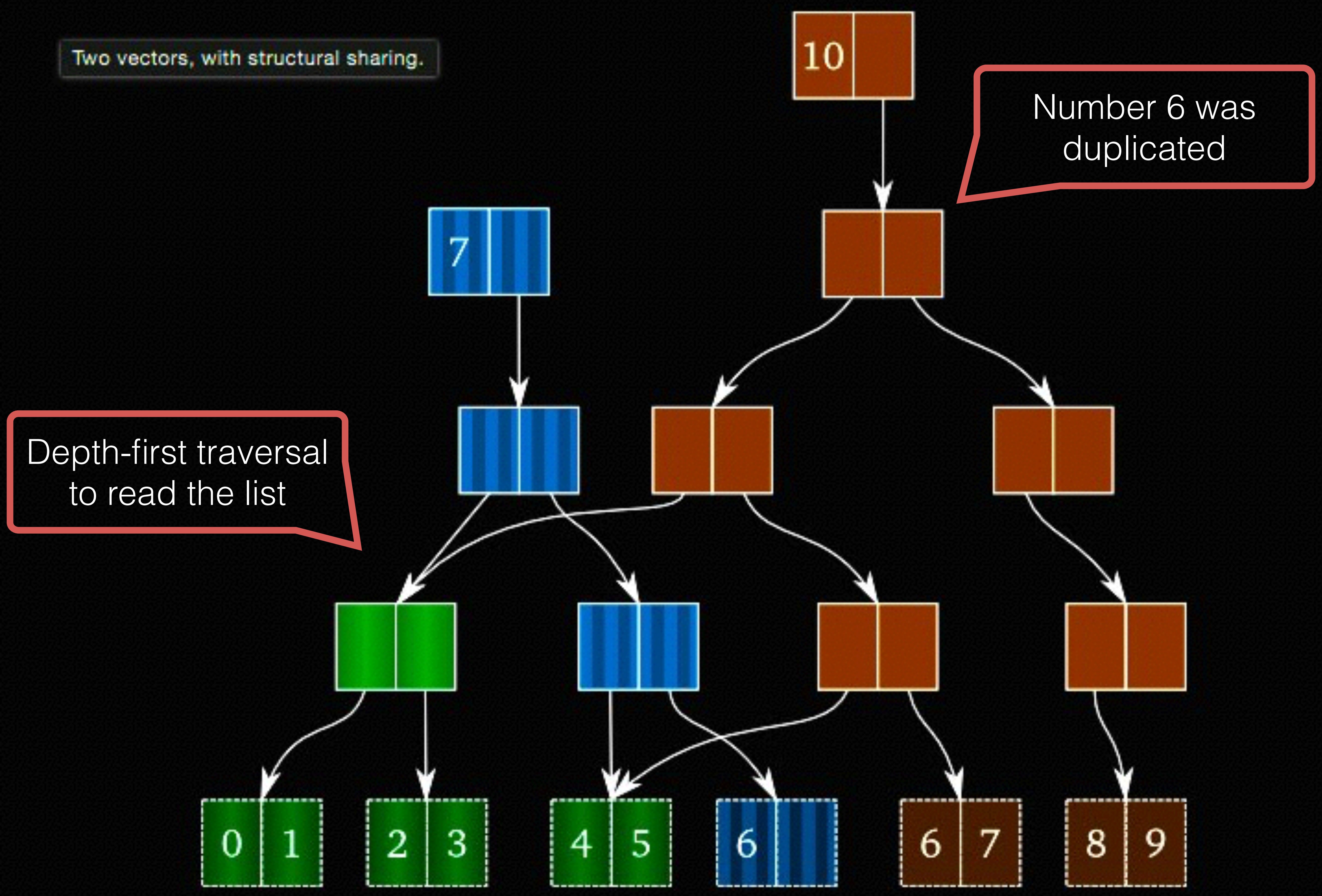
```
{:a "a" :b "b"}
Collections.unmodifiableList(Arrays.asList(stuff))
[("a", 1), ("b", 2)]
```

- Can only be mutated by copy
- (But) Plain copy tend to be inefficient
- Normally used with “persistency”

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 (Log32N in clj)


Two vectors, with structural sharing.
10
Number 6 was duplicated
7
Depth-first traversal to read the list
0
1
2
3
4
5
6
6
7
8
9

Memoization

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

Memoization (Clj 1/1)

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

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


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

Wrapped memoized version
(memoize is in standard library)

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)

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

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

bigSumMemo :: Int -> Int
bigSumMemo = memoize bigSum

– pretty slow
bigSumMemo 10000000 `shouldBe` 4

– pretty fast 2nd time
bigSumMemo 10000000 `shouldBe` 4
```

Wrapped memoized version (memoized in standard lib)

Currying

```
(map (partial str "Winner: ") [:a :b :c])
addOneBang.apply("Lambdas are sweet”)
f = (5+) . (8/)
```

- Currying (in computer science) 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)

```clojure
(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))

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

Need for partial

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);
```

Curying (Hs)

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

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

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

No need for partial

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
- Just: 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
- Similar in Java with parallelStream()

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

- List comprehensions
- Y-combinator
- If strictly typed: functors, applicative, monads

Summary

- FP is about a lot of powerful λ-concepts
- FP is not just about map-filter-collect
- Go check what your favourite language can do

Final Wisdom

- FP covers all needs of everyday programming
- There are 50+ years of Lisp, 20+ Haskell (and others) of wisdom to leverage
- Clojure is a powerful Lisp that runs on the JVM
- Go check it out! (I can give you a demo :)

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