Clojure Basic Training - Module 2

Functional Programming Concepts

λ-calculus

- Where much of the inspiration comes
- A notation for arbitrary mathematical functions
- Can express all computable functions I**N** => I**N**
- Can be extended with a type system
- The rest of the inspiration is from AI

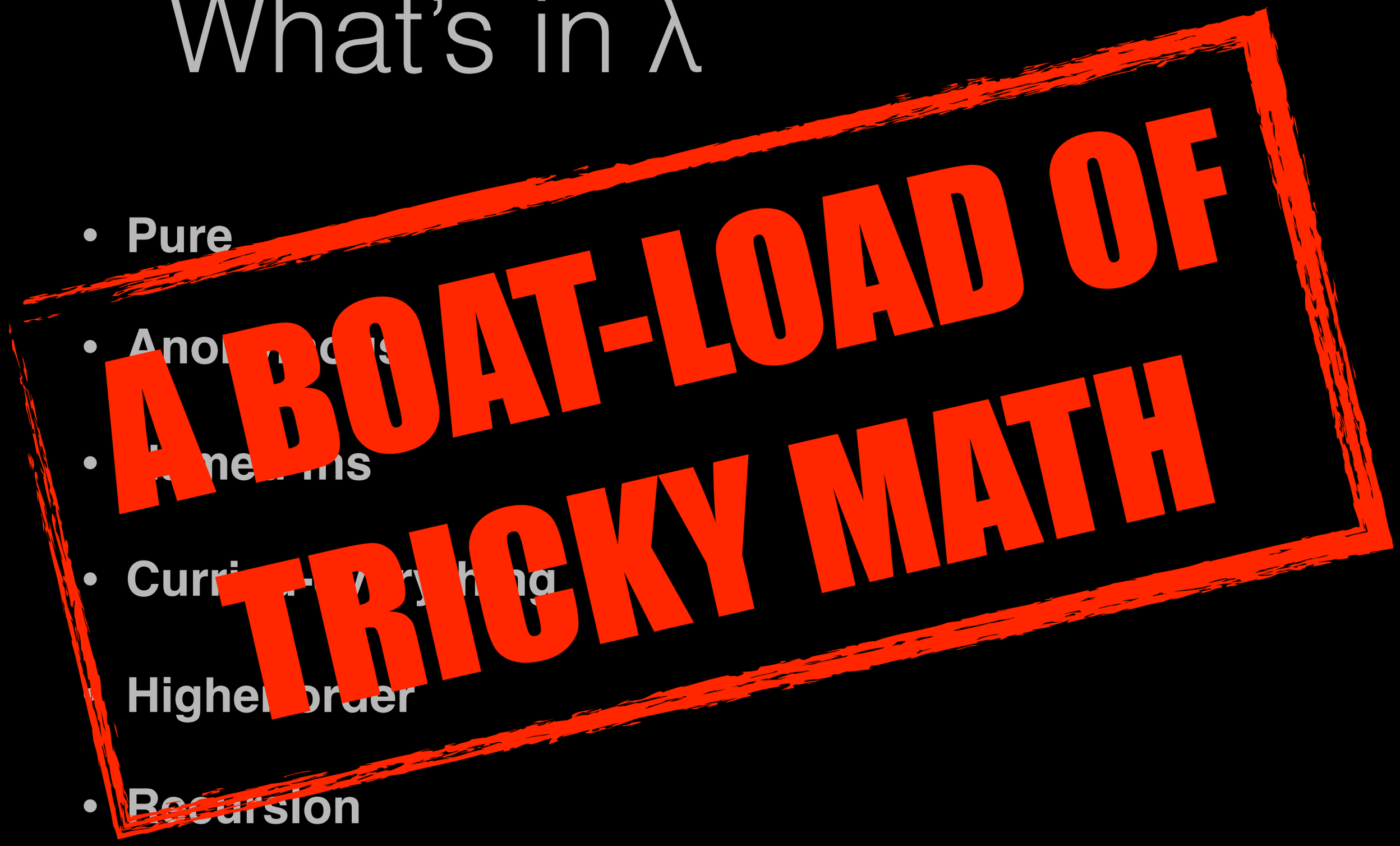
What's in λ-calculus?

- **Pure** mathematical functions (referential transparency)
- **Anonymous** (lambdas) and prefixed fns: (λx.∗ 3 x) 4
- **Named fns**: F =**def**= λx.∗3x
- **Curried-everything**, multiple params fns: λy.λx.∗ y x
- **Higher order** fns: T=**def**= λf.(λx.f (f (f x)))
- **Recursion**: G =**def**= λn f x. zero? n x (G (pred n) f (f x))

=> Clojure has all that (plus inevitable side-effects)

What's in λ

- **Pure**
- **Anonymous**
- **Theorems**
- **Currying**
- **Higher order**
- **Recursion**



It’s Turing complete!

- A fn from I**N** => I**N** is computable if a Turing machine can compute it
- Hence λ-calculus is Turing-complete!
- Let’s make it into a language!
- Lisp and many others were born!
- FP languages use λ-calculus inspired concepts

Referential Transparency

```
(defn whohoo [f win money]
  (f win money))
```

- A function without side effects
- Output is only influenced by parameters
- Results don’t depend on time of evaluation

-> Could “win” or “money” parameters be mutable?

Nice consequences

- Laziness
- Immutable data-structures
- Memoization
- Identity by reference equality
- Almost free parallelization

Clojure is lazy

```
user=> (take 3 (range))
(0 1 2)
```

- No problem invoking **infinitely recursive fns**
- Evaluation can happen at any time with pure fns

Well, not always lazy

- Clojure is lazy but **not strictly** lazy
- Clojure uses **applicative** order evaluation
- Parameters to function are evaluated first!
- Inner to outer, left to right

Immutable data structures

```
user=> (def m1 {:a "a" :b "b"})
user=> (assoc m1 :c "c")
{:c "c", :b "b", :a "a"}
user=> m1
{:b "b", :a "a"}
```

- Standard Clojure sequences are all immutable
- No implicit/hidden way to mutate them
- Smart implementation with structural sharing

Memoization

```
user=> (defn ff [] (println "blah"))
user=> (def m (memoize ff))
user=> (m)
blah
user=> (m)
nil
```

- Caching is just a “memoize” away (very simple form, no TTL or other caching strategy)
- Caching only depends on parameters
- No messing around with identity concepts

Reference Equality

```
user=> (def dom {:a "a" :b {:u "u" :y {:h "red"} :x "x"} :c 0})
user=> (def dom* (update-in dom [:b :y :h] (constantly “green")))
user=> (identical? dom* dom)
false
```

- Comparing nested sequences is blazing fast
- The only thing to compare is their references
- No “equals()” or “hashCode()” custom definition

Parallel execution

```
user=> (defn heavy [ms] (Thread/sleep (* 10000 ms)))
user=> (time (doall (map heavy (repeat 3 (Math/random)))))
"Elapsed time: 27994.376 msecs"
user=> (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

More FP idioms

- Partial application: (def add1 (partial + 1))
- Functional composition: ((comp dec last) [1 2 0])
- List comprehensions: (for [i (range 5) j (range 5)])
- Recursion

References

- “**A short introduction to Lambda Calculus**” www.cs.bham.ac.uk/~axj/pub/papers/lambda-calculus.pdf

- “**Out of the tar pit**” http://shaffner.us/cs/papers/tarpit.pdf/

- “**Structure and interpretation of computer programs**” https://mitpress.mit.edu/sicp/full-text/book/book.html