CLOJURE ADVANCED WORKSHOP

1 DAY, 5 UNITS AND RELATED LABS

HTTPS://GITHUB.COM/USWITCH/USWITCH-ACADEMY/TREE/MASTER/CLOJURE-ADVANCED

AGENDA

1. Lazy Sequences
2. Polymorphism
3. Macros
4. Specs
5. Performance tuning, java interop

1. LAZY SEQUENCES

LAZINESS

```
 (println "hi")  ;; evaluates immediately
 ;; hi

#(println "hi")  ;; needs invocation
;; #object[user$eval1843$fn__1844 0x9036860]
```

- Deferred code evaluation
- Code evaluates when requested

WHY DO I CARE?

Performances and expressiveness:

- Consume data beyond memory capacity
- Avoid unnecessary evaluations
- Leverage caching
- Work with infinite sequences
- Detach producers/consumers

HOW?

- Everything needs wrapping in `#()`
- Without language support it would look like:

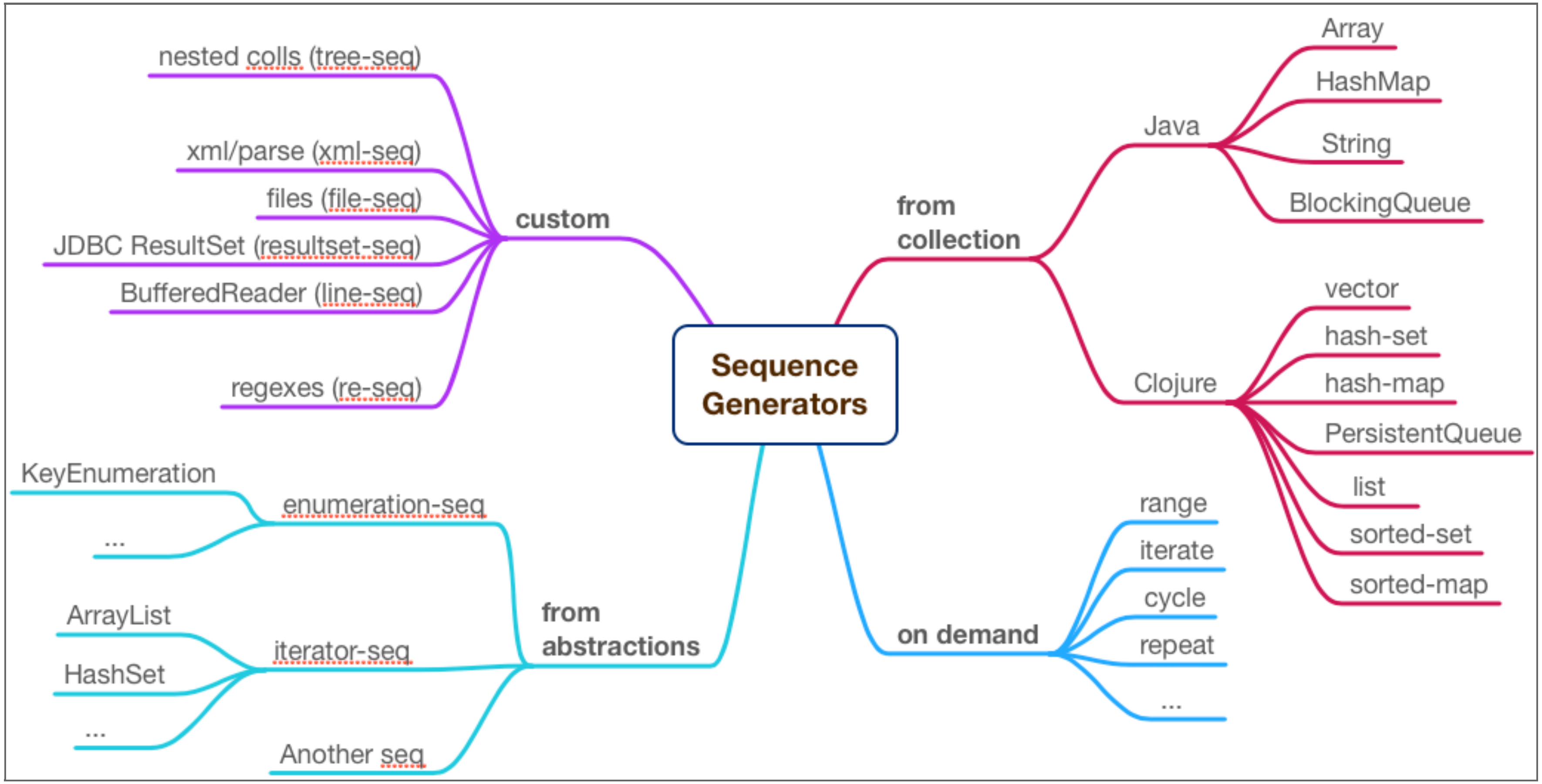
```
(defn lazy-list [coll]
  (map
    (fn [x]
      (fn [] (println "eval" x) x)) ;; need wrapping
    coll))

(def l (lazy-list [1 2 3]))

((first l)) ;; need invocation
;; eval 1
```

Clojure has that baked directly into Sequences

SEQUENCES

- Abstract Data Type (or ADT)
- Iterated sequentially (can't access Nth before Nth-1)
- Stateless cursor: no shared callers, only forward
- Commonly (not necessarily) lazy
- Persistent and immutable

GENERATORS OPTIONS



SOME EXAMPLES

```clojure
;; No hanging
(def a (map inc (range 100000000000)))

;; Self referential
(def fibs
  (cons 0
    (cons 1
      (lazy-seq (map +' fibs (rest fibs))))))

;; Moving window memory load
(require '[clojure.java.io :refer [reader]])
(with-open [r (reader "/huge/petabytes/file")]
  (count (line-seq r)))
```

TRAPS AND GOTCHAS

- Holding the head

```
(let [res (map inc (range 1e7))] (first res) (last res))
;; 10000000
(let [res (map inc (range 1e7))] (last res) (first res))
;; Out of mem
```

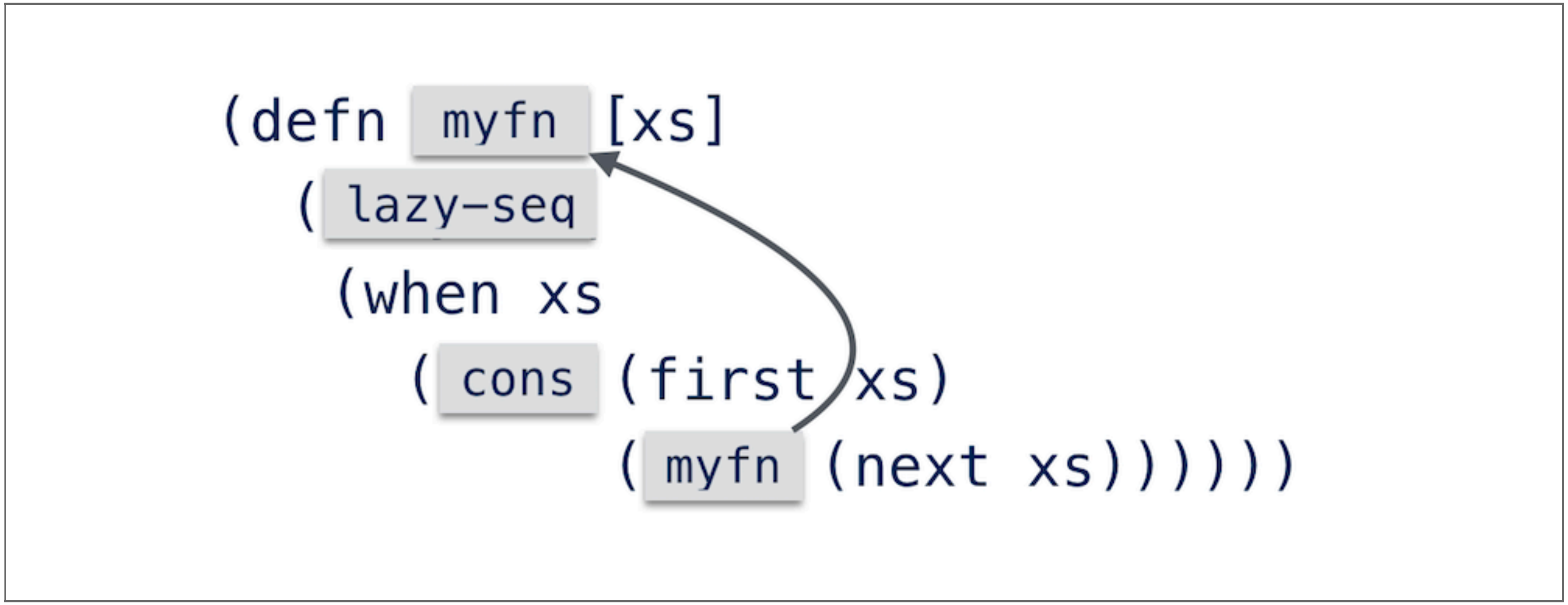
- Chunkiness

```
(first (map #(do (print ".") %) (range 1000)))
;; ................................0
```

TAKING CONTROL

- Create your own lazy sequence
- Optionally define chunkiness
- Your friends: `lazy-seq`, `cons`, `chunk-cons`

RECURSIVE PATTERN



```
(defn myfn [xs]
  (lazy-seq
   (when xs
     (cons (first xs)
           (myfn (next xs))))))
```

UNCHUNK EXAMPLE

- A lazy sequence generator using the pattern
- Apparently doing nothing, but it removes chunking:

```
(defn unchunk [xs]
  (lazy-seq
    (when-first [x xs]
      (cons x (unchunk (rest xs))))))

(first (map #(do (print ".") %) (unchunk (range 1000))))
;; .0
```

CHUNKED-SEQ EXAMPLE

Read bytes from disk by block size (e.g. 4096):

```
(import '[java.io FileInputStream InputStream])

(defn byte-seq [^InputStream is size]
  (let [ib (byte-array size)]
    ((fn step []
       (lazy-seq
         (let [n (.read is ib)]
           (when (not= -1 n)
             (let [cb (chunk-buffer size)]
               (dotimes [i size] (chunk-append cb (aget ib i)))
               (chunk-cons (chunk cb) (step))))))))))
```

BYTE-SEQ HOW TO USE

```clojure
(with-open [fis (FileInputStream. "/usr/share/dict/words")]
  (let [bs (byte-seq fis 4096)]
    (String. (byte-array (take 20 bs)))))
;; "A\na\naa\naal\naalii\naam"
```

LAB 01: LAZY S3

- Create a lazy-seq out of S3 objects.
- Objects are fetched in batch of 1000 each.
- Goal: hide batching and produce a lazy sequence.
- Suggestions: `concat` them into next recursive request.
- Uncomment the fist test in lab01-test
- Run `clj -Atest` from `./labs` folder and make tests green.

2. POLYMORPHISM

POLYMORPHISM IN CLOJURE

- Emphasis on functional dispatch
- Emphasis on flexibility
- Less interested in inheritance
- Less interested in subtyping
- Less interested in type based dispatch

AVAILABLE OPTIONS

- Map lookup (simple and centralised)
- Namespace lookup (same function names)
- Multimethods (flexible)
- Protocols (fast, multiple fns)

MAP LOOKUP

```clojure
(defn process-bg   [instr] (println "bg"))
(defn process-edf  [instr] (println "edf"))
(defn process-bulb [instr] (println "bulb"))

(def vendors
  {:bg     process-bg
   :edf    process-edf
   :bulb   process-bulb})

(defn handle-request [instr]
  ((vendors (keyword (:vendor instr))) instr))

(handle-request {:date "today" :vendor "edf"})
;; edf
```

NAMESPACE LOOKUP

```clojure
(in-ns 'bg)
(clojure.core/defn process [instr] (clojure.core/println "bg"))
(in-ns 'edf)
(clojure.core/defn process [instr] (clojure.core/println "edf"))
(in-ns 'bulb)
(clojure.core/defn process [instr] (clojure.core/println "bulb"))

(in-ns 'user)

(defn handle-request [instr]
  (when-let [f (find-var (symbol (str (:vendor instr) "/process")))]
    (@f instr)))

(handle-request {:date "today" :vendor "edf"})
;; edf
```

MULTIMETHODS

```
(defmulti process (comp keyword :vendor))
(defmethod process :bg   [instr] (println "bg"))
(defmethod process :edf  [instr] (println "edf"))
(defmethod process :bulb [instr] (println "bulb"))

(process {:date "today" :vendor "edf"})
;; edf
```

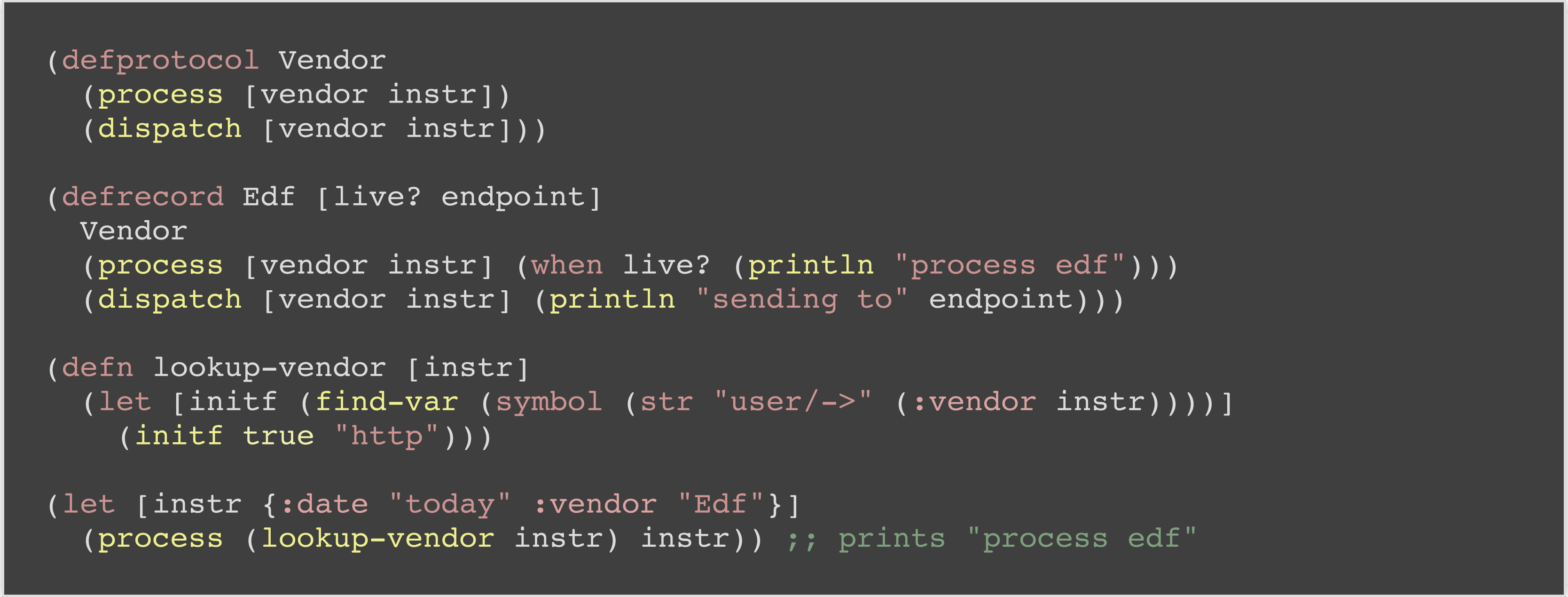
HIERARCHICAL MULTIMETHODS

```
(defmulti process (comp keyword #(str "user/" %) :vendor))
(defmethod process ::junifer [instr] (println "junifer"))
(defmethod process ::bulb    [instr] (println "bulb"))

(derive ::edf ::junifer)
(derive ::bg ::junifer)

(process {:date "today" :vendor "edf"})
;; junifer
```

ISA? RELATIONSHIPS

- `(isa? ::bg ::junifer)` => `true`
- `(parents ::bg)` => `#{:user/junifer}`
- `(descendants ::junifer)` => `#{:user/edf :user/bg}`
- Store in global `@#'clojure.core/global-hierarchy`
- You can pass your own as a param.

PROTOCOLS



```clojure
(defprotocol Vendor
  (process [vendor instr])
  (dispatch [vendor instr]))

(defrecord Edf [live? endpoint]
  Vendor
  (process [vendor instr] (when live? (println "process edf")))
  (dispatch [vendor instr] (println "sending to" endpoint)))

(defn lookup-vendor [instr]
  (let [initf (find-var (symbol (str "user/->" (:vendor instr))))]
    (initf true "http")))

(let [instr {:date "today" :vendor "Edf"}]
  (process (lookup-vendor instr) instr)) ;; prints "process edf"
```

ADDING BEHAVIOUR

```clojure
(defrecord Bg [live? endpoint]
  Vendor
  (process [vendor instr] (when live? (println "process bg")))
  (dispatch [vendor instr] (println "sending to" endpoint)))

(let [instr {:date "today" :vendor "Bg"}]
  (dispatch (lookup-vendor instr) instr))
;; "sending to http"
```

SHARING BEHAVIOUR

```
(defrecord Bg  [live? endpoint])

(def vendor-common
  {:dispatch (fn [vendor instr]
    (println "sending via" (:endpoint vendor)))})

(extend Bg
  Vendor
  (assoc vendor-common
    :process (fn [vendor instr]
      (when (:live? vendor) (println "process bg")))))

(let [instr {:date "today" :vendor "Bg"}]
  (dispatch (lookup-vendor instr) instr))
;; "sending via http"
```

GOTCHAS

- Multimethods and Protocols need explicit "require"
- Using them from other namespaces can be tricky
- Protocol functions can clash with local functions

LAB 02

- Work with vendors multimethods
- Extend to new vendor, add new multimethod
- Refactor into protocols
- Uncomment tests in lab02-test
- Run `clj -Atest` from `./labs` folder and make tests green.

RESOURCES

- **Multimethods announcement** on Clojure ML (28/7/2008)
- **Protocol alpha release** and **feedback release** on the Clojure ML (12/11/2009)

3. MACROS

WHAT ARE MACROS?

- Special functions that evaluate before other code evaluates
- They see normal code as data structures
- Their output is then subject to normal evaluation
- Macro effectively "expands" in place of their call site

WHY MACROS?

- In general, they expand language possibilities
- Loads in the stdlib: `->>`, `for`, `with-redefs`, `with-open`
- Generate lots of similar functions (for example to call AWS services)
- Setup/teardown behaviour (e.g. `with-open`, `with-redefs`)
- DSLs and small compilers (e.g. `for` has a "dsl" and related compiler)

HOME-MADE MACRO

```clojure
(defn like-a-macro [[op & args :as form]]
(println (type op))
  (if (fn? op) (apply op args) form))

(like-a-macro (list + 1 1)) ;; 2
(like-a-macro (list :a :b :c)) ;; (:a :b :c)
```

- Can't really write programs as lists... but what if
- Special fn taking unevaluated forms as arguments
- Compiler uses fn output instead of actual form
- This is what macros are for.
- Access to full power manipulation of sources!

A REAL MACRO

```
(defmacro is-a-macro [[op & args :as form]]
  (if-let [f (find-var (symbol (str "clojure.core/" op)))]
    (cons f args)
    (cons 'list form)))

(is-a-macro (+ 1 1)) ;; 2
(is-a-macro (:a :b :c)) ;; (:a :b :c)
```

- We had to "qualify" and lookup the symbol
- The input is now a list of unevaluated symbols
- It needs to return a new form (list)
- The compiler take that "in place" of original call.
- `defmacro` is itself a macro built on top of `defn`

HELP PLEASE

Syntax quote (the back tick) allows for:

- Auto-qualification of symbols
- Unquote ~ evaluated context
- Unquote-splicing unquote all items in a collection
- Auto-gensym prevents accidental override of symbols
- `&form` contains the surrounding form
- `&env` contains the local bindings

SYNTAX QUOTE

- Like `quote` prevents evaluation
- Plus all mentioned goodies

```
`(1 2 3)
;; (1 2 3)

(= `(1 2 3) '(1 2 3))
;; true
```

AUTO-QUALIFICATION

```clojure
(require '[clojure.string :as s :refer [lower-case]])

`s/upper-case
;; clojure.string/upper-case

`lower-case
;; clojure.string/lower-case

`foo
;; user/foo
```

UNQUOTE AND SPLICING

```
`[1 2 (+ 1 2)  ~(+ 1 2)]
;; [1 2 (clojure.core/+ 1 2) 3]

`[1 2 ~[3 4] ~@[3 (+ 1 2)]]
;; [1 2 [3 4] 3 3]
```

- Used to mix between the macro expansion and evaluation context
- Splicing works similarly to `apply` to spread arguments

ACCIDENTAL CAPTURING

```
(defmacro ** [a b]
  `(let [~'y (* ~a ~a)]
    (* ~'y ~b)))

(macroexpand '(** a b))
;; (let [y (* a a)] (* y b))

(let [x 2 y 5] (** x y))
;; 16
```

- `~'` (tilde single-quote) expands into the actual symbol.
- Equivalent to `~(quote y)`
- Which is the "evaluation of quote y"
- Or y itself.

AUTO-GENSYM

```
(defmacro ** [a b]
  `(let [y# (* ~a ~a)]
    (* y# ~b)))

(macroexpand '(** a b))
;; (let [y__2270 (* a a)] (* y__2270 b))

(let [x 2 y 5] (** x y))
;; 20
```

&FORM

```
(defmacro just-print-me [& args] (println &form))

(just-print-me foo :bar 123)
;; (just-print-me foo :bar 123)
```

- `&form` captures the actual macro call as data
- Useful to inspect metadata (like type hints)

&ENV

```clojure
(defmacro with-negatives [& body]
  (let [pos (vec (keys &env))
        neg (mapv #(symbol (str % "'")) pos)
        values (mapv #(.eval (.init %)) (vals &env))]
    `(let [~pos ~values
           ~neg (mapv - ~values)]
       ~@body)))

(let [x 1 y 2]
  (with-negatives [[x y] [x' y']]))
;; [[1 2] [-1 -2]]
```

- Useful for pre-processing of local bindings
- For example generating additional ones
- x' and y' are generated from the originals

LAB 03

- We want to create a parallel let macro.
- The general idea is to transform this:

```
(let [a (+ 1 1) b (* 2 2)]
  (+ a b))
```

Into:

```
(let [a (future (+ 1 1)) b (future (* 2 2))]
  (let [a (deref a) b (deref b)]
    (+ a b)))
```

USEFUL TIPS FOR THE LAB

- Use the example as a starting point for the macro.
- Work you way out removing specific keys/expressions.
- The macro should output the code ready for evaluation.
- Treat the bindings as an actual vector you can manipulate
- `(take-nth 2 bindings)` gives you the names of the locals
- `(take-nth 2 (rest bindings))` gives you the expressions
- `(list 'future '(+ 1 1))` creates `(future (+ 1 1))`

4. SPECS

INTRO

- Declarative description of code properties
- `[clojure.spec.alpha :as s]` main ns
- `[clojure.spec.test.alpha :as stest])` testing tools
- `org.clojure/test.check` for generative
- `[clojure.spec.gen.alpha :as gen]`
- Usages: validation, documentation, testing, parsing

VOCABULARY

- rich APIs, it takes some time to get fluent
- but it's relatively easy to learn
- Building blocks: `s/def`, `s/fdef`, `s/cat`, `s/keys`...
- Interaction: `s/valid?`, `s/conform`, `s/explain`...
- Expressiveness: `s/and`, `s/or`, `s/*` ...

PROCESS

Mostly personal taste but:

- Sketch out concepts from the bottom
- Compose them up into abstractions
- Spec-out public facing functions (or endpoints)
- Have some generative testing (optional)

EXAMPLE

- A "powerset" is the set of all subsets of a set.
- There are 2^n subsets of [0 1 2] (2^3 = 8)
- ((0 1 2) (2) (0) (1) (1 2) (0 1) (2 0) ())
- We could write a lazy version for that

```
(defn powerset [xs]
  (letfn [(rotate [xs]
            (map rest (iterate #(concat (rest %) (list (first %))) xs)))
          (step [n xs]
            (when-let [rs (seq (take n (rotate xs)))]
              (when (ffirst rs)
                (into (mapcat step (range (dec n) 0 -1) rs) rs))))]
    (keep seq (conj (step (count xs) xs) xs))))

(powerset [0 1 2]) ;; okay
;; ((0 1 2) (0 1) (2 0) (1 2) (1) (2) (0))

(powerset [2 2 2]) ;; we can delegate distinct to the caller
;; ((2 2 2) (2 2) (2 2) (2 2) (2) (2) (2))
```

PROPERTIES

- We could write many examples of calls to `powerset`
- Or capture the essence of a powerset into a spec
- Including the need for the input to be `distinct`

```
(require '[clojure.spec.alpha :as s]
         '[clojure.spec.test.alpha :as stest])

(s/fdef powerset
  :args (s/cat :xs (s/coll-of int? :distinct true))
  :ret seqable?
  :fn (fn [{:keys [args ret]}]
        (and (= (Math/pow 2 (count (:xs args))) (count ret))
             (= (count ret) (count (distinct ret))))))
```

GENERATIVE TESTING

- We have the properties for the function
- We can generate as much examples as we want

```
(s/exercise-fn `powerset)
;; wall of numbers scrolling indefinitely
```

- We didn't say "xs" can't be millions of items...

LIMITED RANGE

- We can create our own "type" as a new spec.
- A list of distinct integers up to 20 in size

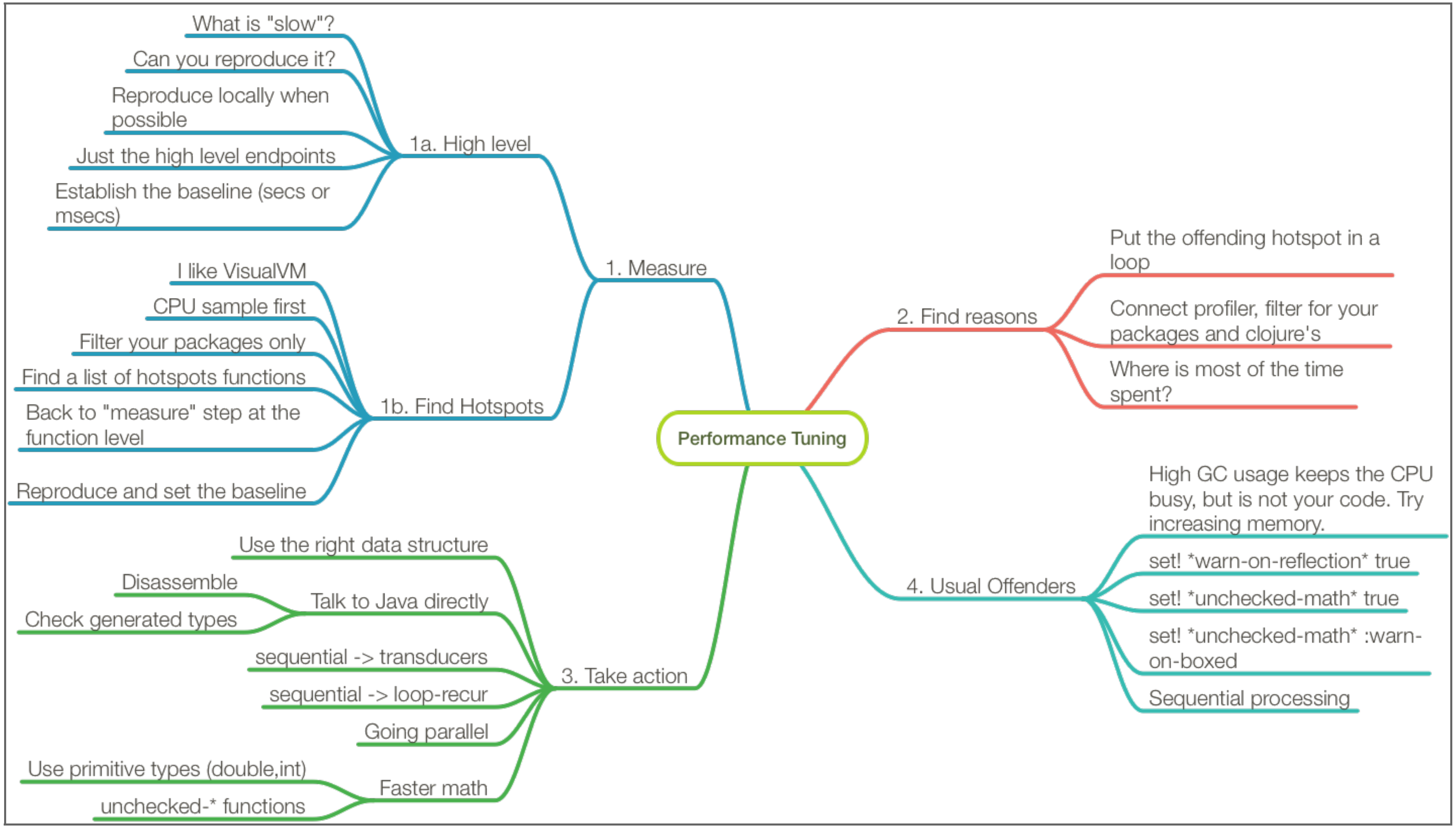
```
(s/def ::limited-range
  (s/coll-of int? :max-count 20 :distinct true))
```

- Then use it to specify the argument.

LAB 04

- I wrote a "fizz-buzz" and I tested it!
- I'm 100% sure it works, even with negatives!
- Can you write a spec for it and prove me wrong?
- Be sure to not over specify it
- Your spec should not contain the implementation
- Search for general facts.
- For example: for any given "n" how many distinct strings?

5. PERFORMANCES



- Performance Tuning
  - 1. Measure
    - 1a. High level
      - What is "slow"?
      - Can you reproduce it?
      - Reproduce locally when possible
      - Just the high level endpoints
      - Establish the baseline (secs or msecs)
    - 1b. Find Hotspots
      - I like VisualVM
      - CPU sample first
      - Filter your packages only
      - Find a list of hotspots functions
      - Back to "measure" step at the function level
      - Reproduce and set the baseline
  - 2. Find reasons
    - Put the offending hotspot in a loop
    - Connect profiler, filter for your packages and clojure's
    - Where is most of the time spent?
  - 3. Take action
    - Use the right data structure
    - Talk to Java directly
      - Disassemble
      - Check generated types
    - sequential -> transducers
    - sequential -> loop-recur
    - Going parallel
    - Faster math
      - Use primitive types (double,int)
      - unchecked-* functions
  - 4. Usual Offenders
    - High GC usage keeps the CPU busy, but is not your code. Try increasing memory.
    - set! *warn-on-reflection* true
    - set! *unchecked-math* true
    - set! *unchecked-math* :warn-on-boxed
    - Sequential processing

HIGH LEVEL BASELINE

- Isolate and replicate (possibly locally)
- Stub out network call (when possible)
- Measure deterministically: that's the baseline
- Not necessarily exact, but deterministic
- Usually "seconds" at high level

HOTSPOTS

- Pick a profiler, for example VisualVM
- CPU sample, filter your app packages
- Replicate hotspots in code
- Loop the hotspots if necessary
- Measure deterministically
- Usually ms, us, or even ns
- You need a profiler (e.g. Criterium)

TYPE HINTING

- Only useful if you have Java Interop
- Especially useful in tight loops
- Less useful at high level (e.g. `(.close conn)`)

```clojure
(import 'java.nio.charset.StandardCharsets)

(defn get-bytes [s] (.getBytes s (StandardCharsets/UTF_8)))

(get-bytes "clojure")
;; #object["[B" 0x5f254608 "[B@5f254608"]

(set! *warn-on-reflection* true)
(defn get-bytes [s] (.getBytes s (StandardCharsets/UTF_8)))

;; Reflection warning call to method getBytes can't be resolved
```

TYPE HINTS IMPACT

Usually 1 order of magnitude

```
(require '[criterium.core :refer [quick-bench]])
(quick-bench (get-bytes "clojure"))
;; Execution time mean : 2.503821 μs
(defn get-bytes [^String s] (.getBytes s (StandardCharsets/UTF_8)))
(quick-bench (get-bytes "clojure"))
;; Execution time mean : 62.361678 ns
```

WHAT'S GOING ON

- We are going to use the `no.disassemble` library

```
(require '[no.disassemble :refer [disassemble]])
(println (disassemble get-bytes))
```

- Search for `invokeStatic`
- `invoke` is used `when` get-bytes is high order

BEFORE TYPE HINTS

```
public static java.lang.Object invokeStatic(java.lang.Object s);
    3  ldc <String "getBytes"> [13]
    5  iconst_1
    6  anewarray java.lang.Object [15]
    9  dup
   10  iconst_0
   11  getstatic java.nio.charset.StandardCharsets.UTF_8
   14  aastore
   15  invokestatic clojure.lang.Reflector.invokeInstanceMethod
         (java.lang.Object, java.lang.String, java.lang.Object[])
```

AFTER TYPE HINTING

```
public static java.lang.Object invokeStatic(java.lang.Object s);
    6  getstatic java.nio.charset.StandardCharsets.UTF_8
    9  checkcast java.nio.charset.Charset [21]
   12  invokevirtual java.lang.String.getBytes
         (java.nio.charset.Charset)
```

TRANSDUCERS

- Easy win for long chain of threaded macros
- Less win for shorter chain or trivial transforms
- Sometimes porting to transducers is not trivial

PARALLELISM

- `pmap`
- `r/fold`
- `core.async` pipelines
- custom with `future` and `deref`

LAB 05

- Pick a project you work on
- Better if not too big and easily runnable
- Connect VisualVM
- Find hotspots
- Easy fixes?

- **Before type hints**
- **After type hinting**
- **Transducers**
- **Parallelism**
- **Lab 05**