Everyone set up?

http://bit.ly/flatmap-clojure-ws

Intro to Clojure

flatMap(Oslo) 2014
Alf Kristian Støyle

KODEMAKER

Clojure workshop

- Basic syntax
- Data structures
- Functions
- Reading code
- More functions
- Side effects
- Destructuring
- Java interop
- Macros

Clojure

- General purpose
- Lisp (List Processing)
- Functional
- Compiled
- Dynamically typed

Literals

```
(class "String")   ;=> java.lang.String
(class #"regex")   ;=> java.util.regex.Pattern
(class 123)        ;=> java.lang.Integer
(class 2147483648) ;=> java.lang.Long
(class 123M)       ;=> java.math.BigDecimal
(class 123N)       ;=> clojure.lang.BigInt
(class true)       ;=> java.lang.Boolean
(class false)      ;=> java.lang.Boolean
(class 42/43)      ;=> clojure.lang.Ratio
(class \c)         ;=> java.lang.Character
(class 'foo)       ;=> clojure.lang.Symbol
(class :bar)       ;=> clojure.lang.Keyword
(class nil)        ;=> nil
```

Collection literals

```
; List
'(3 2 1) -> (list 3 2 1)

; Vector
[1 2 3] -> (vector 1 2 3)

; Set
#{1 2 3} -> (hash-set 1 2 3)

; Map
{:one "one", :two "two", :three "three"} ->
(hash-map 1 "one", 2 "two", 3 "three")
```

This is a list

```
'(+ 1 2)
 ;=> (+ 1 2)
```

This is an expression (form)

```
(+ 1 2)
;=> 3
```

Prefix notation

```
(+ 1 2)
;=> 3
```

This is an expression (form)

```
(apply + '(1 2))
     ;=> 3
```

This is a java.lang.String

```
"(+ 1 2)"
;=> "(+ 1 2)"
```

This is a java.lang.String
which can be converted to a form

```
(read-string "(+ 1 2)")
;=>(+ 1 2)
```

This is a java.lang.String
which can be converted to a form
that can be evaluated

```clojure
(eval (read-string "(+ 1 2)"))
;=> 3
```

Read-compile-evaluate

1. Text is converted to forms
2. Forms are converted to byte code by the compiler
3. Byte code is evaluated

Functions

```
(fn [n] (* 2 n))
;=> #<core$eval376$fn__377 user$eval376$fn__377@5c76458f>
```

Functions

```
((fn [n] (* 2 n)) 4)
;=> 8
```

Functions

```
(fn [n] (* 2 n))
#(* 2 %)
#(* 2 %1)
```

Functions & Vars

```
(def times-two
  (fn [n] (* 2 n)))
;=> #'user/times-two

(times-two 4)

;=> 8
```

Functions & Vars

```
(def times-two
  (fn [n] (* 2 n)))

(defn times-two
  [n] (* 2 n))
```

Arity

```
(defn times [a b] (* a b))
```

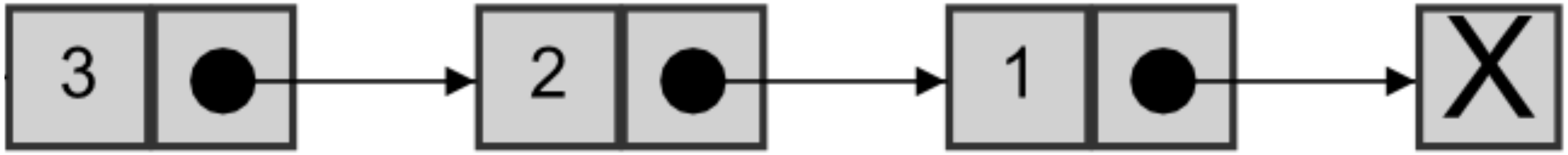
```
(times 1 2)
;=> 2
(times 1 2 3)
;=> ArityException Wrong number of args (3) passed to: user/times ...
```

```
(defn times [& s] (apply * s))
```

```
(times 1)
;=> 1
(times 1 2)
;=> 2
(times 1 2 3)
;=> 6
```

Immutable and persistent data structures

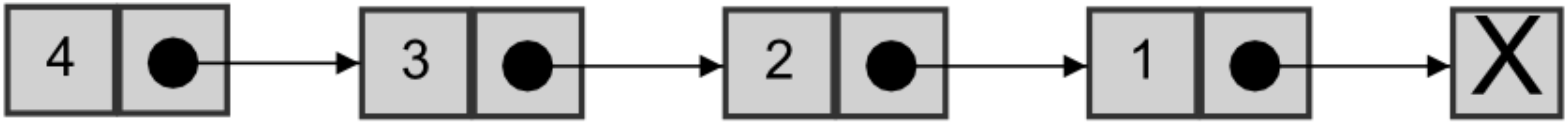
```
(def my-list '(3 2 1))
; => (3 2 1)
```



Immutable and persistent data structures

```
(def my-list '(3 2 1))
; => (3 2 1)

(def my-other-list (cons 4 my-list))
; => (4 3 2 1)
```



Immutable collections

```clojure
; List
'(3 2 1)

; Vector
[1 2 3]

; Set
#{1 2 3}

; Map
{1 "one", 2 "two", 3 "three"}
```

One dash: _ => 0

Two dashes: __ => A function

Three dashes: ___ => A value (list)

```
(deftest very-basic-types
  ; Which number is the same as 1? :)
  (is (= 1 1)))
```

```
(deftest basic-types-with-are
  (are [x y] (= x y)
       (+ 0 1) 1
       (= 1 1) true))
```

Light table

- Ctrl-space - Commands
- Tab - autocomplete
- Ctrl-d - Shows doc-string of current var
- Ctrl-enter - Evaluate current form
- Ctrl-shift-space - Evaluate entire file
- Instarepl
- lein midje :autotest

Exercise 1

- test/clojure_workshop_flatmap/ex_1.clj
- Remove #_ from each test
- Fix tests
- Lots of hints in the comments

http://clojure.github.io/clojure/
http://clojure.org/cheatsheet

http://bit.ly/clj-cheatsheet
http://clojuredocs.org/clojure_core
(Solutions in the solution branch)

Reading code

Reads inside out

```
(- 10 (+ 1 2 (+ 3 4)))
```

```
(if true
  (println "true")
  (println "false"))
```

let

```
(- 10 (+ 1 2 (+ 3 4)))
```

```
(let [inner  (+ 3 4)
      middle (+ 1 2 inner)
      ten    10
      result (- ten middle)]
  result)
```

More functions

```
(count
  (filter ifn?
          (map #(var-get (second %))
               (ns-publics ‘clojure.core))))
     ;=> 567
```

first

```
(first '(1 2 3))
;=> 1

(first [1 2 3])
;=> 1

(first #{1 2 3})
;=> 1

(first {:one "one" :two  "two"})
;=> [:one "one"]
```

rest

```
(rest '(1 2 3))
;=> (2 3)

(rest [1 2 3])
;=> (2 3)

(rest #{1 2 3})
;=> (2 3)

(rest {:one "one" :two  "two"})
;=> ([:two "two"])
```

rest

```
(rest '())
;=> ()

(rest [])
;=> ()

(rest #{})
;=> ()

(rest {})
;=> ()
```

next

```
(next '())
;=> nil

(next [])
;=> nil

(next #{})
;=> nil

(next {})
;=> nil
```

some collections are functions

```
('(1 2 3) 0)
;=> java.lang.ClassCastException: clojure.lang.PersistentList cannot be cast to clojure.lang.IFn


([1 2 3] 0)
;=> 1

(#{3 2 1} 1)
;=> 1

({:one 1 :two 2 :three 3} :one)
;=> 1
```

conj

```
(conj '(2 1) 3)
;=> (3 2 1)

(conj [1 2] 3)
;=> [1 2 3]

(conj #{1 2} 3)
;=> #{1 2 3}

(conj {:k1 1} [:k2 2])
;=> {:k2 2, :k1 1}
```

Filter

```
(defn filter
  "Returns a lazy sequence of the items in coll for which
  (pred item) returns true. pred must be free of side-effects."
  ([pred coll]
    ...))
```

Filter

```
(filter #(= 0 (mod % 2)) [0 1 2 3 4 5])
;=>  (0 2 4)
```

Filter

```
(filter even? [0 1 2 3 4 5])
 ;=>  (0 2 4)
```

Map

```
(defn map
  "Returns a lazy sequence consisting of the result of applying f to the
  set of first items of each coll, followed by applying f to the set
  of second items in each coll, until any one of the colls is
  exhausted.  Any remaining items in other colls are ignored. Function
  f should accept number-of-colls arguments."
  {:added "1.0"
   :static true}
  ([f coll]
    ...))
```

Map

```
(map #(+ 1 %) [1 2 3 4 5])
;=>  (2 3 4 5 6)
```

Map

```
(map inc [1 2 3 4 5])
;=> (2 3 4 5 6)
```

Reduce (Fold)

```
(defn reduce
  "f should be a function of 2 arguments. If val is not supplied,
  returns the result of applying f to the first 2 items in coll, then
  applying f to that result and the 3rd item, etc. If coll contains no
  items, f must accept no arguments as well, and reduce returns the
  result of calling f with no arguments.  If coll has only 1 item, it
  is returned and f is not called.  If val is supplied, returns the
  result of applying f to val and the first item in coll, then
  applying f to that result and the 2nd item, etc. If coll contains no
  items, returns val and f is not called."
  {:added "1.0"}
  ([f coll]
     ...)
  ([f val coll]
     ...))
```

Reduce (Fold)

```
(reduce #(+ %1 %2) [1 2 3 4 5])
;=> 15
```

Reduce (Fold)

```
(reduce + [1 2 3 4 5])
;=> 15
```

Reduce (Fold)

```
(reduce conj () [1 2 3 4 5])
;=>  (5 4 3 2 1)
```

Side-effects

IO

Mutability

IO (side-effects)

```
(defn foo []
  (println "Doing IO, and returning 1")
  1)
```

```
 (do
  (println "Doing IO, and returning 1")
   1)
```

```
(dotimes [x 10]
  (println x))
```

```
(doseq [x (range 10)]
  (println x))
```

Mutability (side-effects)

```
(atom 0)
;=> #<Atom@4b8baa34: 0>

(def an-atom (atom 0))
;=> #'user/an-atom

(swap! an-atom inc)
;=> 1

(deref an-atom)
;=> 1
@an-atom
;=> 1
```

Midje syntax

```
(fact "A couple of things about midje"
      1 => 1
      (throw (Exception.)) => (throws Exception))
```

Exercise 2

- test/clojure_workshop_flatmap/ex_2.clj
- Remove #_ from each test
- Fix tests
- Lots of hints in the comments

Destructuring

Destructuring

```
(defn foo [s]
  (let [one (first s)
        two (fnext s)]
    (println one two)))

(defn foo [s]
  (let [[one two] s]
    (println one two)))
```

Destructuring

```
(defn bar [m]
  (let [one (:one m)
        two (:two m)]
    (println one two)))

(defn bar [m]
  (let [{one :one two :two} m]
    (println one two)))
```

Destructuring

```
(defn bar [m]
  (let [{one :one two :two} m]
    (println one two)))
```

```
(defn bar [{one :one two :two}]
  (println one two))
```

```
(defn foo [s]
  (let [[one two] s]
    (println one two)))
```

```
(defn foo [[one two]]
  (println one two))
```

Java interop

```
(new java.util.ArrayList)
;=> #<ArrayList []>

(java.util.ArrayList.)
;=> #<ArrayList []>
```

Java interop

```
(System/currentTimeMillis)
;=> 1318164613423

(.size (new java.util.ArrayList))
;=> 0

(. (new java.util.ArrayList) size)
;=> 0
```

Java interop

```
(map .toString [1 2 3])
;=> CompilerException java.lang.RuntimeException: Unable to
resolve symbol: .toString in this context
```

```
(map (memfn toString) [1 2 3])
;=> ("1" "2" "3")
```

Macros

```
(infix (1 + 2))
;=> 3
```

```
(defmacro infix [form]
  (list (second form) (first form) (first (nnext form))))
;=> #'user/infix
```

Read-compile-evaluate

1. Text is converted to forms
2. Forms are converted to byte code by the compiler
3. If the compiler finds a macro, expand the macro and start at 1.
4. Byte code is evaluated

Macros

```
(infix (1 + 2))
;=> 3
```

```
(defmacro infix [form]
  (list (second form) (first form) (first (nnext form))))
;=> #'user/infix
```

```
(macroexpand '(infix (1 + 2)))
;=> (+ 1 2)
```

Macros

```
(infix (1 + 2))
;=> 3
```

```
(defmacro infix [form]
  (list (second form) (first form) (first (nnext form))))
;=> #'user/infix
```

```
(defmacro infix [form]
  `(~(second form) ~(first form) ~@(nnext form)))
;=> #'user/infix
```

Macros

- The two rules of the macro club

1. Don’t write macros.
2. Only write macros if that is the only way to encapsulate a pattern.
3. You can write any macro that makes life easier for your callers when compared with an equivalent function.

Programming Clojure - Stuart Halloway 2009

Exercise 3

- You can choose, either
- Open src/clojure_workshop_flatmap/cat.clj
  - Implement the cat command. Tests are in test/clojure_workshop_flatmap/cat.clj
  - Shell of program is there, you should provide logic
  - ‘lein uberjar’ build executable.
  - ‘java -jar target/cat.jar -h’ for usage.
- Or implement infix macros
  - Open clojure_workshop_flatmap/ex_3.clj
  - Remove #_ from each test
  - Fix tests
  - Lots of hints in the comments

Clojure workshop

- Basic syntax
- Data structures
- Functions
- Reading code
- More functions
- Side effects
- Destructuring
- Java interop
- Macros

Where to go next?

- Recursion
- Laziness
- Reference types (atoms, agents, refs, vars)
- Multi methods
- Protocols
- Macros
- Libraries

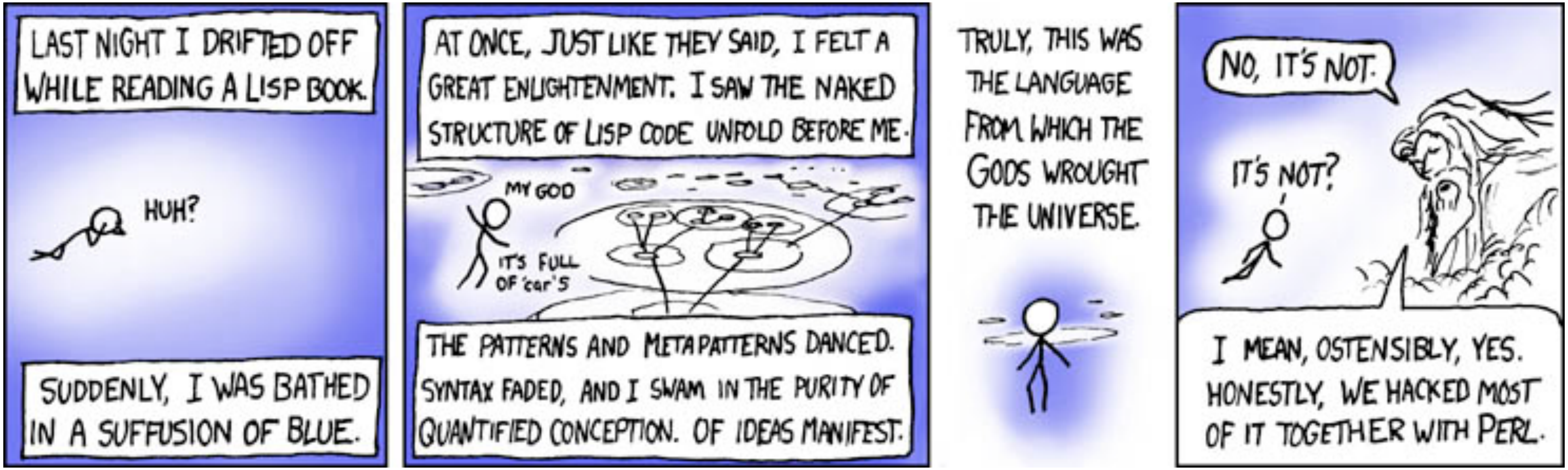
Where to go next?

http://clojure.org/books

https://groups.google.com/forum/#!forum/clojure

http://www.4clojure.com/

Thank you!
clojure.org
alf.kristian@kodemaker.no


LAST NIGHT I DRIFTED OFF WHILE READING A LISP BOOK.
HUH?
SUDDENLY, I WAS BATHED IN A SUFFUSION OF BLUE.
AT ONCE, JUST LIKE THEY SAID, I FELT A GREAT ENLIGHTENMENT. I SAW THE NAKED STRUCTURE OF LISP CODE UNFOLD BEFORE ME.
MY GOD
IT'S FULL OF 'car'S
THE PATTERNS AND METAPATTERNS DANCED. SYNTAX FADED, AND I SWAM IN THE PURITY OF QUANTIFIED CONCEPTION. OF IDEAS MANIFEST.
TRULY, THIS WAS THE LANGUAGE FROM WHICH THE GODS WROUGHT THE UNIVERSE.
NO, IT'S NOT.
IT'S NOT?
I MEAN, OSTENSIBLY, YES. HONESTLY, WE HACKED MOST OF IT TOGETHER WITH PERL.

http://xkcd.com/224/