Clojure in a

Tom Van Cutsem

Vrije
Universiteit
Brussel

Software
Languages.Lab

Clojure in a nutshell

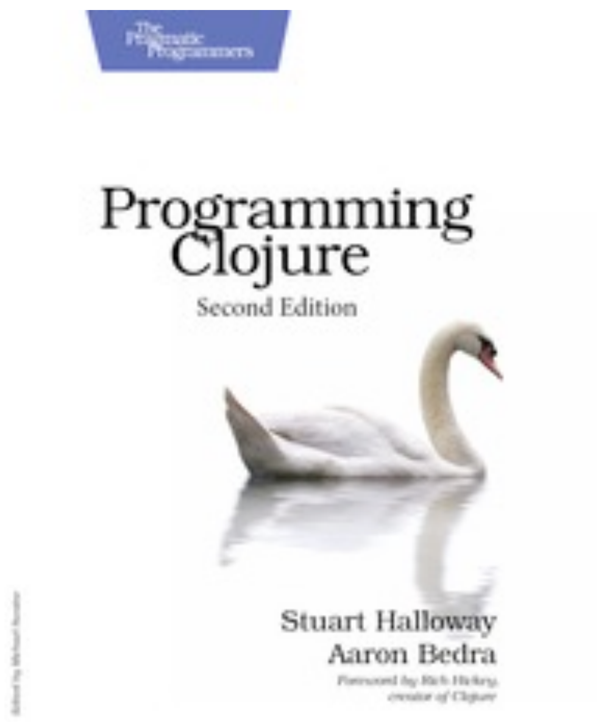
- A modern Lisp dialect (2007), designed by Rich Hickey
- Uses the Java Virtual Machine as runtime platform
- Promotes a Functional Programming style
- Designed for Concurrency

Functional Style

- Clojure is **not** a **pure** functional language (like Haskell), but...
- Emphasis on **immutable data structures**: list, vector, set, map, ...
- Emphasis on recursion rather than looping
- Lisp’s lists generalized to abstract **sequences**

Useful reading material

- clojure.org, in particular clojure.org/rationale and clojure.org/state
- http://www.4clojure.com/
- http://clojuredocs.org/
- Stuart Halloway: Programming Clojure
- Clojure wikibook: en.wikibooks.org/wiki/Clojure_Programming



Exploring Clojure

Syntax

- Clojure *reader* transforms source code into *forms*, then translates forms into Clojure data structures. Examples of Clojure forms:

Boolean	`true, false`
Character	`\a`
Keyword	`:doc`
List	`‘(1 2 3)`
Map	`{ :name “Bill”, :age 42 }`
Nil	`nil`
Number	`1`
Set	`#{:foo :bar :baz}`
String	`“hello world”`
Symbol	`‘foo`
Vector	`[1 2 3]`

Read-eval-print Loop

```
42
=> 42

[1 2 3]
=> [1 2 3]

(+ 1 2)
=> 3

(> 5 2)
=> true
```

```
(/ 22 7)
=> 22/7

(class (* 1000 1000 1000))
=> java.lang.Integer

(class (* 1000 1000 1000 1000 1000
1000 1000 1000))
=> java.lang.BigInteger
```

Lists and vectors

- Immutable!

```
(def x (list 1 2 3)) ; or ‘(1 2 3)
=> #‘user/x
(first x)
=> 1
(rest x)
=> (2 3)
(cons 0 x)
=> (0 1 2 3)
x
=> (1 2 3)
```

```
(def y (vector 1 2 3)) ; or [1 2 3]
=> #‘user/y
(nth y 0)
=> 1
(nth y 5)
=> java.lang.IndexOutOfBoundsException
(assoc y 0 5)
=> [5 2 3]
y
=> [1 2 3]
```

Keywords

- Keywords are immutable, cached, “constant strings”
- Keywords evaluate to themselves

```
:foo
=> :foo

(keyword? :foo)
=> true

(string? :foo)
=> false
```

Maps

- Maps are collections of (key, value) pairs
- Maps are functions *f(key) -> value*
- Any Clojure value can be a key in a map (most common keys are keywords)

```
(def inventors {:Lisp "McCarthy", :Clojure "Hickey"})
=> #‘user/inventors

(inventors :Lisp)
=> “McCarthy”

(inventors :foo)
=> nil

(inventors :foo “unknown”)
=> “unknown”
```

Maps

- Maps are immutable too

```
(def inventors {:Lisp "McCarthy", :Clojure "Hickey"})
=> #‘user/inventors

(assoc inventors :Python "van Rossum")
=> {:Python "van Rossum", :Lisp "McCarthy", :Clojure "Hickey"}

(dissoc inventors :Lisp)
=> {:Clojure "Hickey"}

inventors
=>{:Lisp "McCarthy", :Clojure "Hickey"}
```

Keywords and Maps

- Keywords are also functions that take a map as argument and look themselves up in the map:

```
(def inventors {:Lisp "McCarthy", :Clojure "Hickey"})
=> #‘user/inventors

(inventors :Clojure)
=> “Hickey”

(:Clojure inventors)
=> “Hickey”
```

Functions

- Defining Functions:

```
(defn name doc-string? [params*] body)
```

- Example:

```
(defn greeting
  "Returns a greeting of the form 'Hello, username.'"
  [username]
  (str "Hello, " username))

(greeting “Tom”)
=> “Hello, Tom”
```

Anonymous Functions

- `defn` defines a named function, `fn` defines an anonymous function (cf. `lambda` in Scheme):

```
(fn [x] (* x x))
```

Anonymous Functions: example

- Create a function that filters out short words from a sequence of words:

```
(defn indexable-word? [word]
  (> (count word) 2))
(filter indexable-word? (split "A fine day it is" #"\W+"))
=> ("fine" "day")
```

```
(filter (fn [word] (> (count word) 2))
        (split "A fine day it is" #"\W+"))
=> ("fine" "day")
```

Anonymous Functions: example

- Use `let` to define local bindings:

```
(defn indexable-words [text]
  (let [indexable-word? (fn [word] (> (count word) 2))]
    (filter indexable-word? (split text #"\W+"))))
```

```
(indexable-words "A fine day it is")
=> ("fine" "day")
```

Closures

- Functions close over their lexical scope:

```
(defn make-greeter [prefix]
  (fn [name]
    (str prefix ", " name)))

(def hello-greeting (make-greeter “Hello”))
(def aloha-greeting (make-greeter “Aloha”))

(hello-greeting “world”)
=> “Hello, world”

(aloha-greeting “world”)
=> “Aloha, world”
```

Destructuring

- Anywhere names are bound, you can nest a vector or map to destructure a collection and bind only specific elements of the collection

```
(def dist [p]
  (let [x (first p)
        y (second p)]
    (Math/sqrt (+ (* x x) (* y y)))))
```

```
(def dist [[x y]]
  (Math/sqrt (+ (* x x) (* y y))))
```

Control flow: loop/recur

- `loop` is like `let`, but sets a *recursion point* that can be jumped to by means of `recur`

```
(loop [result []
       x 5]
  (if (zero? x)
      result
      (recur (conj result x) (dec x))))
=> [5 4 3 2 1]
```

- Like Scheme’s “named let”:

```
(let loop ((result ‘())
           (x 5))
 (if (zero? x)
     result
     (loop (append result (list x)) (- x 1))))
=> (5 4 3 2 1)
```

Accessing Java

```
(new java.util.Random) ; Java: new java.util.Random()
=> java.util.Random@18a4f2

(. aRandom nextInt 10) ; Java: aRandom.nextInt(10)
=> 8

(.nextInt aRandom 10)  ; Java: aRandom.nextInt(10)
=> 8
```

Exception Handling

- Clojure uses essentially the same exception handling model as Java

```
(throw (new Exception “something failed”))


(try
  (do-something)
  (catch IOException e
    (println “caught exception”))
  (finally
    (println “clean up”)))
```

Sequences

Sequences

- An abstract data type: the sequence (seq, pronounce “seek”)
    - A logical list
    - Not necessarily implemented as a linked-list!
- Used pervasively: all Clojure collections, all Java collections, Java arrays and Strings, regular expression matches, files, directories, I/O streams, XML trees, ...

Clojure Sequence Library

- Most Clojure sequences are lazy: they generate elements “on demand”
    - Sequences can be infinite
- Sequences are immutable and thus safe for concurrent access

Operations on sequences

```
(first aseq)

(rest aseq)

(cons elem aseq)
```

Example: lists and vectors

- Lists and Vectors are sequences

```
(first ‘(1 2 3))
=> 1

(rest ‘(1 2 3))
=> (2 3)

(cons 0 ‘(1 2 3))
=> (0 1 2 3)
```

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

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

(cons 0 [1 2 3])
=> (0 1 2 3)
```

Example: maps

- Maps are sequences of (key, value) pairs:

```
(first { :fname “Rich” :lname “Hickey” })
=> [:fname “Rich”]

(rest  { :fname “Rich” :lname “Hickey” })
=> ([:lname “Hickey”])
```

- Element order is undefined!

Creating sequences

```
(range 5)
=> (0 1 2 3 4)

(range 5 10)
=> (5 6 7 8 9)

(range 1 10 2)
=> (1 3 5 7 9)
```

Creating and filtering sequences

- `(iterate f x)` lazily constructs the infinite sequence
  `x, f(x), f(f(x)), f(f(f(x))), ...`

- `(take n seq)` returns a lazy sequence of the first n items in seq

```
(defn natural-numbers []
  (iterate inc 0))

(take 5 (natural-numbers))
=> (0 1 2 3 4)
```

- `(filter pred seq)` returns a (lazy) filtered sequence

```
(take 5 (filter even? (natural-numbers)))
=> (0 2 4 6 8)
```

Transforming sequences

- (map f seq) maps function f lazily over each element of the sequence

```
(map inc [0 1 2 3])
=> (1 2 3 4)
```

- (reduce f val seq) applies f to val and the first argument, then applies f to the result and the second element, and so on. Returns the accumulated result.

```
(reduce + 0 (range 1 11))
=> 55
```

Imperative vs. Functional style: case study

- indexOfAny walks a string and reports the index of the first char that matches any char in searchChars, or -1 if no match is found:

```
public static int indexOfAny(String str, char[] searchChars);
```

```
indexOfAny(null, _) => -1
indexOfAny("", _)   => -1
indexOfAny(_, null) => -1
indexOfAny(_, [])   => -1
indexOfAny("zzabyycdxx",['z','a']) => 0
indexOfAny("zzabyycdxx",['b','y']) => 3
indexOfAny("aba", ['z'])           => -1
```

Imperative vs. Functional style: case study

- Consider the following typical Java implementation:

```java
// From Apache Commons Lang, http://commons.apache.org/lang/
public static int indexOfAny(String str, char[] searchChars) {
  if (isEmpty(str) || ArrayUtils.isEmpty(searchChars)) {
    return -1;
  }

  for (int i = 0; i < str.length(); i++) {
    char ch = str.charAt(i);
    for (int j = 0; j < searchChars.length; j++) {
      if (searchChars[j] == ch) {
        return i;
      }
    }
  }
  return -1;
}
```

Strings in Clojure

- Clojure strings are Java strings

```
(.toUpperCase “hello”)
=> “HELLO”
```

- Clojure can manipulate strings as sequences of Characters

```
(count ‘(1 2 3))
=> 3

(count “hello”)
=> 5
```

Imperative vs. Functional style: case study

- Clojure version: first, define a helper function `indexed` that takes a collection and returns an indexed collection:

```
(defn indexed [coll]
  (map vector
       (iterate inc 0) coll))



(indexed ‘(a b c))
=> ([0 a] [1 b] [2 c])

(indexed “abc”)
=> ([0 \a] [1 \b] [2 \c])
```

Imperative vs. Functional style: case study

- Next, find the indices of all characters in the string that match the search set:

```
(defn index-filter [pred coll]
  (loop [icoll (indexed coll)
         acc   []]
    (if (empty? icoll)
      acc
      (let [[idx elt] (first icoll)]
        (if (pred elt)
          (recur (rest icoll) (conj acc idx))
          (recur (rest icoll) acc))))))
```

Imperative vs. Functional style: case study

- In Clojure, sets are functions (predicates) that test membership of their argument in the set:

```
(#{\a \b} \a)
=> \a
(#{\a \b} \c)
=> nil
```

- So we can pass a set of characters to index-filter:

```
(index-filter #{\a \b} "abcdbbb")
=> (0 1 4 5 6)

(index-filter #{\a \b} "xyz")
=> nil
```

Imperative vs. Functional style: case study

- To define `index-of-any`, simply take the first result from index-filter:

```
(defn index-of-any [pred coll]
  (first (index-filter pred coll)))
```

```
(index-of-any #{\z \a} "zzabyycdxx")
=> 0
(index-of-any #{\b \y} "zzabyycdxx")
=> 3
```

Concurrency in Clojure

Threads

- Clojure reuses JVM threads as the unit of concurrency

```
(.start (new Thread
          (fn [] (println "Hello from new thread"))))
```

Clojure Philosophy

- Immutable state is the default
- Where mutable state is required, programmer must explicitly select one of the following APIs:

state change is	**Asynchronous**	**Synchronous**
Coordinated	-	Refs
Independent	Agents	Atoms

Clojure Refs

- Ref: a mutable *ref*erence to an immutable value

```
(def today (ref "Monday"))
```

- The ref wraps and protects its internal state. To read its contents, must explicitly dereference it:

```
(deref today)
=> "Monday"

@today
=> "Monday"
```

Refs and Software Transactional Memory (STM)

- To update a reference:

```
(ref-set today "Tuesday")
```

- Updates can only occur in the context of a transaction:

```
(ref-set today "Tuesday")
=> java.lang.IllegalStateException: No transaction running
```

Refs and Software Transactional Memory (STM)

- To start a transaction:

```
(dosync body)
```

- Example:

```
(dosync (ref-set today "Tuesday"))
=> "Tuesday"
```

Coordinated updates

- “Coordinated”: isolated and atomic

```
(dosync
  (ref-set yesterday "Monday")
  (ref-set today "Tuesday"))
```

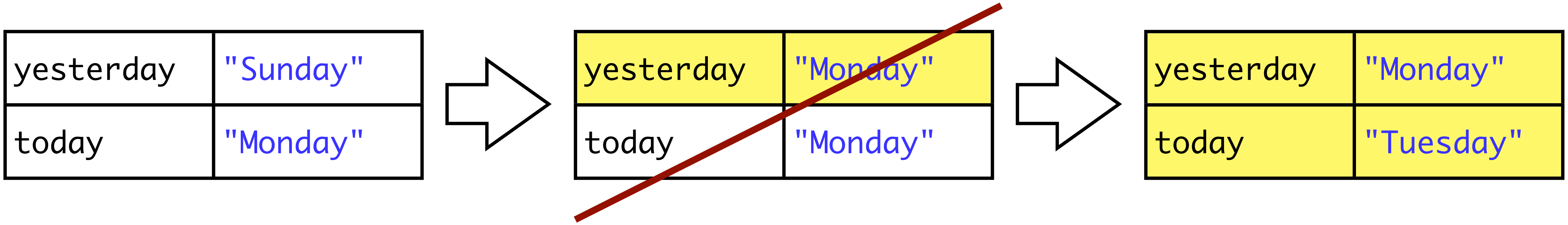
- No thread will be able to observe a state in which `yesterday` is already updated to `"Monday"`, while `today` is still set to `"Monday"`.

Coordinated updates

- “Coordinated”: isolated and atomic

```
(dosync
  (ref-set yesterday "Monday")
  (ref-set today "Tuesday"))
```

- No thread will be able to observe a state in which `yesterday` is already updated to `"Monday"`, while `today` is still set to `"Monday"`.



yesterday	"Sunday"
today	"Monday"

yesterday	"Monday"
today	"Monday"

yesterday	"Monday"
today	"Tuesday"

Example: money transfer

- Transferring money atomically from one bank account to another

```
(defn make-account [sum]
  (ref sum))

(defn transfer [amount from to]
  (dosync
    (ref-set from (- @from amount))
    (ref-set to   (+ @to   amount))))

(def accountA (make-account 1000))
(def accountB (make-account 0))

(transfer 100 accountA accountB)
(println @accountA) ; 900
(println @accountB) ; 100
```

Side-effects & retries

- Transactions may be aborted and retried.
- The transaction `body` may be executed multiple times.
- Should avoid side-effects other than assigning to refs (no I/O)

```
(dosync
  (println "launch missiles")
  (perform-update))
```

Wrap-up

Clojure: Summary

- Functional style: a Lisp on the JVM
- Immutable data structures: lists, vectors, sets, maps
- Direct access to Java objects
- All collections are sequences: abstract lists
- Most operations support lazy/infinite sequences
- Designed for concurrency

Important features not covered

- Atoms
- Agents
- Macros
- Multimethods
- Protocols
- Transients
- List comprehensions
- Unit testing
- Metadata
- Namespaces
- ...