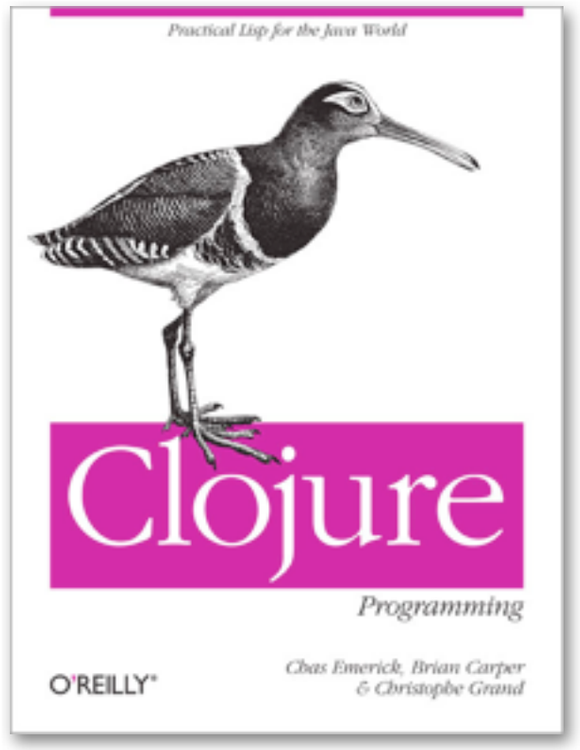
Getting Started with Clojure

Berlin, Jan 23-25 2013
Lambda Next

Principles



- The slideware is an outline
- Ask! Ponder! Discuss!
- Practice!
- The book as reference material

Syntax

Homoiconic

- Datastructures to represent the code
- No real syntax or reserved word

Atomic types

nil	`nil`
Boolean	`true false`
Character	`\h \newline \u12B4`
String	`"hello world"`
Regex	`#"[0-9]*"`
Numbers	`8 0.8e1 24/3 0x8 010 2r1000 8N 8M`
Keyword	`:name :ns/name ::alias/name`
Symbol*	`'name 'ns/name` `alias/name`

*Quotes are not integral part of the symbol

Types composites

List	`(a b c)`
Vector	`[a b c]`
Set	`#{a b c}`
Map	`{a b, c d}`

Metadata

- Composite types and symbols can bear some metadata (as a map)
- `^{:meta "data"} [a b c]`

Not quite

- Strictly speaking that's enough to write some Clojure
- In practice there are comments and syntaxic sugars

Comments

line comments	`; comment` `;; important comment`
commented-out expression	`#_(I'm not there)` `#_#_beware pitfall`
shebang	`#!line comment`

Syntaxic sugars

- Introduced as we go
- Explained later on

Functional Programming

Differences

Prog :	Procedural	Object	Functional
Data & logic	distinct	mixed	distinct
Execution flow	fixed***	dynamic	dynamic
Closures	no	yes*	yes
Side effects	endemic	endemic	controlled**

*More or less tedious depending on the language
**A continuum ranging from frowned-upon to verboten or even impossible
***Unless the language features function pointers

Differences

Prog :	Functional	Impacts
Data & logic	distinct	Polymorphism, coupling, serialisation
Execution flow	dynamic	Genericity (HOF and polymorphism)
Closures	yes	HOF catalyser
Side effects	controlled	«reasonable», readable

*More or less tedious depending on the language
**A continuum ranging from frowned-upon to verboten or even impossible
***Unless the language features function pointers

Functions

Call	`(f arg1 arg2 arg3)`
Global definition	`(defn sq [x] (* x x))`
Local definition	`(fn [x y] (+ (sq x) (sq y)))`
Sugar	`#(+ (sq %1) (sq %2))`

Control

	Example	Evaluates to
if	`(if expr then else)` `(if expr then) ; else = nil`	then or else, depending on expr

do	`(do`
	`  expr-1`
	`  ...`
	`  expr-N)`
		the last expr (expr-N)

let	`(let [x expr-x`
	`  y expr-y]`
	`  expr)`
		expr or the last expr when several (implicit do)

More control!

- Everything else is built on top of those «special forms» using macros
- Can be inspected by `source`

```
=> (source when)
(defmacro when
  "Evaluates test. If logical true, evaluates body in
an implicit do."
  {:added "1.0"}
  [test & body]
  (list 'if test (cons 'do body)))
```

About if

- Behind all boolean tests (see implementations of and, or etc.)
- In a boolean context:
  - nil and false are the only falses

loop/recur

- recur optimises self-recursive tail call
- loop allows to limit recur's scope (enclosing fn by default)

FP in Clojure

- Focus on values (data)
  - immutability, no wrappers
- Prefer sets and maps to indices and linear scans
  - «relational-oriented»

Sequences and collections

Large abstractions

- conj and seq are the two most important functions
- conj adds an item to a collection
- seq returns a sequential (list-like) view of a collection

Sequences

- A sequence has linked-list interface: first & rest
- seq on an empty sequence yields nil
  - idiomatic way to test for emptiness: (if (seq coll) ...)
  - super tasty with an if-let
- next = (comp seq rest)

Seqables

- Clojure collections and Java ones (incl. Map and all Iterables)
- Implementations of clojure.lang.Seqable
- Sequences themselves
- Java arrays
- Strings (CharSequences in truth)

Sequence API

- All those functions implicitly call seq on their arguments
  - applicable to anything seqable
  - cons
  - map
  - reduce
  - concat, take, drop, take-while, take-nth, drop-last etc.

Sequence building

- cons, lazy-seq, lazy-cat
- Too low-level! Better use HOFs or for

Collection API

- More fragemenetd than the sequence API
- Fragmented by «aspect»
  - Collection : conj, count
  - Associative : assoc, get, find
  - Indexed : nth
  - Reversible : rseq
  - Stack : pop, peek
  - Set : disj
  - Map : dissoc
  - Sorted : subseq, rsubseq

Support

	Sequence	List	Vector	Map	Set
Collection	✔ (count O(n))	✔	✔	✔	✔
Associative	✘	✘	✔	✔	✘ ✔
Indexed	✔O(n)	✔O(n)	✔	✘	✘
Reversible	✘	✘	✔	✘ ✔	✘ ✔
Stack	✘	✔	✔	✘	✘
Set	✘	✘	✘	✘	✔
Map	✘	✘	✘	✔	✘
Sorted	✘	✘	✘	✘ ✔	✘ ✔

for: the Swiss Army knife

- «seq comprehension»
- combines map/mapcat/filter/take-while
- cartesian product

```
(for [a (range 20)
      :let [ha (/ a 2)]
      b (range 2 (inc ha))
      :when (zero? (rem a b))]
  [a b])
```

Concurrent programming

Reference types

- Mutability/articulation points
  - Keep the number of moving parts low
- Bear synchronisation semantics

Reference types

	independent	coordinated
synchronous	atom	ref
asynchronous	agent	?

Similarities

Type :	Atom	Ref	Agent
creation	`(atom x)`	`(ref x)`	`(agent x)`
update	`(swap! a * 2)`	`(alter r * 2)` `(commute r * 2)`	`(send a * 2)` `(send-off a * 2)`
reset	`(reset! a y)`	`(ref-set r y)`	`(restart-agent a y)*` `(send a (constantly y))`
read	`@x (deref x)`		
validators	`set-validator! get-validator`		
watchers	`add-watch remove-watch`		

*When agent in error

Uniform update model

- (alter r f arg2... argN) is a central pattern
- update-in uses it too
- It avoids closures creation
  - More esthetic, «fluid»
  - A tad more performant

STM

- (dosync ...) delimits a transaction's scope
- A transaction never fails*
- No retry notification

*Unless when the retry limit is reached, or when a validation error occurs. In both cases, an exception is thrown.

3 times

A transaction is bounded by two instants:

- its start point
- its end (commit point)
- in between, is the transaction time

In between

- deref of a ref:
  - if already modified, in-transaction value
  - else its value as if at the start point
  - no check on the actual (out-of-txn) value at commit

In between

- ensure of a ref :
    - like deref except the ref is guaranteed to not be modified by another transaction

In between

- ref-set or alter of a ref:
  - performs a deref
  - updates its in-transaction value
  - guarantees no concurrent update

In between

- commute of a ref:
  - performs a deref
  - updates its in-transaction value
  - at commit point, a to-be-commited value is recomputed based on the last out-of-transaction value!

Code smell

```
(dosync
  (if (test @x)
    (alter y action)))
```

- Nothing guarantees that (test @x) still holds at commit time
- «Ensure» when such a coherency is meaningful

commute or alter ?

- When operations order don't matter
- Nor the coherency between several refs at the end of the transaction
- Then commute is preferable
  - Alter is the safe choice

Agents and transactions

- Sends and send-off are postponed to after the commit point
- Likewise inside an agent
  - but release-pending-sends when in urge

The other derefables

- delay, promise, futures
- more about dataflow and parallelism than concurrencey
- delays are interesting to alleviate concurrency on hot atoms/refs atoms/refs etc.
- realized? and deref + timeout

OO : the finest bits

OO?

- Define abstractions
- Participate in abstractions
- Reuse implementations

Problems

- Is a POJO an abstraction ?
- 1 inheritance = only 1 reuse
  Tedious delegation (and «this» is lost)
- Adapt a new abstraction to an existing type
  (-> choked to death by wrappers)
- Why does the 1st argument get all the love? It's so unfair!!!

Multimethods

- Fix all those problems (except POJO)
- Their expressivity is expensive (perfs)
- Optional system of hierarchies is complex
- No logical grouping

Protocols

- Performant to very performant
- Dispatch on the type of the 1st argument
  (= this)
  Types are this required
- Methods are grouped

Type creation

- Don’t do it!

Type creation

- Anonymous type (and closure) reify
- Low-level techical type: deftype
  - may have private mutable fields with explicit synchronisation semantics
- «Business» types: defrecord
  - no mutable fields
  - access and update like maps -> POJO

defrecord

- Field access:
  - (.field obj)
  - (:field obj) <- preferable
- Maps are an abstraction

Interop

dot dot dot

- (ClassName. arg1 ... argN)
- (.field obj)
- (set! (.field obj) 42)
- (.method obj arg1 ... argN)
- ClassName/StaticField
- (ClassName/StaticMethod arg...)
- (set! ClassName/StaticField 42)

-> and doto

```
(doto (javax.swing.JFrame. "sjacket")
  (com.apple.eawt.FullScreenUtilities/setWindowCanFullScreen
true)
  (.setContentPane
    (doto (javax.swing.JEditorPane.)
      (.setPreferredSize (java.awt.Dimension. 550 700))
      (.setBackground (java.awt.Color/decode "0xfdf6e3"))
      (.setForeground (java.awt.Color/decode "0x657b83"))
      (.setFont (java.awt.Font/decode "Monospaced"))))
  .pack
  (.setVisible true))
```

Implement

- In decreasing preference order:
  - reify/deftype/defrecord (interfaces/protos)
  - proxy (extends abstract (or not) classes without protected fields)
  - Java (seriously)
  - gen-class (:impl-ns is very important)
- See nice diagram flow in the book

Java's calling

```
// inside static init
RT.var("clojure.core", "require").invoke(
    Symbol.create("your.namespace"));
...
// inside method
RT.var("your.namespace", "your-fn").invoke(
    arg1, ... , argN);
```

Type hints

- ^NomClass (expression)
- (set! *warn-on-reflection* true)
- Type rarely, type upstream!
- Beware of type-hints-eating macros!

Macros

The macro club

- Don’t write macros
- Use them wisely
- Control
- Syntaxic sugar
- Optimisation

Homoiconic

- code is represented by regular types
- code is thus manipulated using the regular API
- a macro, at core, is just a function from code to code which is executed at compile-time

Macroexpansion

- Macro invocation = macroexpansion
- macroexpand and macroexpand-1
- From the outside to the inside

Syntax-quote

- Like quote except :
  - ns-qualify symbols, ::keywords and ::key/words but not symbols#, nor %N or :keywords or even :key/words.
  - replace all ~expr and ~@expr by what they evaluate to

Fill-in-the-blanks

```
`(let [r# ~expr]
   (if r#
     (let [~binding r#]
       ~then
       ~else)))
```

Hygiene

- syntax-quote and autogensyms#
- compiler checks locals have no ns
  - catch missing #s
- prevents unwanted interactions with user code

Counter-example

```
(defmacro aif [expr then else]
`(let [~'it ~expr]
   (if ~'it ~then ~else)))
```

```
=> (let [it 3]
     (aif (first (range 10))
          it
          -1))
0
```