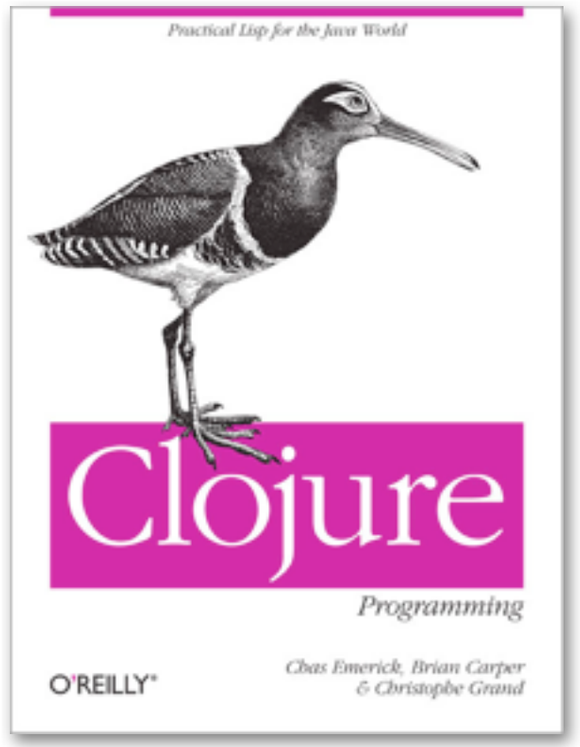
Débuter en Clojure

Genève, 14-15 mai 2012

Christophe Grand

Principes



- La présentation est une trame
- Poser des question
- Pratiquer !
- Se référer au livre pour plus de détails

Installation de CCW

- Pas à pas : bit.ly/mixitclj

Plan

- Installation
- Syntaxe
- Programmation fonctionnelle
- Séquences et collections
- TP : aggrégation, réductions
- Programmation concurrente
- TP : TRON bikes
- TP : Justification de texte
- Interop Java
- TP : TRON la suite
- OO, les meilleurs morceaux
- TP : Zippers
- Macros
- TP : cond-let

Syntaxe

Homoiconique

- Utilisation des structures de données pour représenter le code
- Pas de syntaxe ou de mots réservés à proprement parler

Types atomiques

nil	`nil`
Booléen	`true false`
Caractère	`\h \newline \u12B4`
Chaîne	`"hello world"`
Regex	`#"[0-9]*"`
Nombre	`8 0.8e1 24/3 0x8 010 2r1000 8N 8M`
Mot-clé	`:name :ns/name ::alias/name`
Symbole*	`'name 'ns/name` `alias/name`

*Les quotes ne font pas partie du symbole

Types composites

Liste	`(a b c)`
Vecteur	`[a b c]`
Ensemble	`#{a b c}`
Map	`{a b, c d}`

Métadonnées

- Les types composites et les symboles peuvent avoir des métadonnées (une map)
- `^{:meta "data"} [a b c]`

Presque

- Tout code Clojure peut-être écrit strictement avec cette «syntaxe»
- En pratique il existe des commentaires et des sucres syntaxiques

Commentaires

commentaire ligne	`; commentaire` `;; remarque importante`
expression commentée	`#_(je ne suis pas là)` `#_#_attention piege`
shebang	`#!commentaire ligne`

Sucres syntaxiques

- Introduits au fur et à mesure
- Expliqués plus tard

Programmation fonctionnelle

Différences

Prog :	Procédurale	Objet	Fonctionnelle
Données et traitements	distincts	mêlés	distincts
Flux d'exécution	fixe***	dynamique	dynamique
Fermetures	non	oui*	oui
Effets de bord	endémiques	endémiques	contrôlés**

*Plus ou moins laborieux (voire manuel) selon les langages
**Continuum de «deconseillés» à «interdits»
***Sauf si pointeur de fonctions

Différences

Prog :	Fonctionnelle	Impacts
Données et traitements	distincts	Polymorphisme, couplage, sérialisation
Flux d'exécution	dynamique	Généricité (HOF et polymorphisme)
Fermetures	oui	Catalyseur des HOF
Effets de bord	contrôlés**	«Raisonnabilité», Lisibilité

*Plus ou moins laborieux (voire manuel) selon les langages
**Continuum de «déconseillés» à «interdits»
***Sauf si pointeur de fonctions

Fonctions

Appel	`(f arg1 arg2 arg3)`
Définition globale	`(defn sq [x] (* x x))`
Définition locale	`(fn [x y] (+ (sq x) (sq y)))`
Sucre	`#(+ (sq %1) (sq %2))`

Contrôle

	Exemple	Valeur
if	`(if expr then else)` `(if expr then) ; else = nil`	then ou else, selon expr

let	`(let [x expr-x`
	`  y expr-y]`
	`  expr)`
		expr ou la dernière expr si plusieurs

do	`(do`
	`  expr-1`
	`  ...`
	`  expr-N)`
		la dernière expr (expr-N)

Plus de contrôle

- Tout le reste est construit sur les «formes spéciales» par des macros
- Peut donc être inspecter par `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)))
```

A propos de if

- Base de tout test booléen (cf impl. and, or etc.)
- Dans un contexte booléen :
  - nil et false sont les seules valeurs fausses

loop/recur

- recur optimise un appel récursif terminal (self-recursive tail call)
- loop permet de limiter la portée de recur

FP en Clojure

- Accent sur les valeurs
  - immutabilité, pas de wrappers
- Favoriser sets et maps aux indices et aux parcours linéaires
  - «orienté relationnel»

Séquences et collections

Larges abstractions

- conj et seq sont les deux fonctions les plus importantes
  - conj ajoute un élément à une collection
  - seq produit une vue séquentielle de la collection

Séquences

- Une séquence a une interface de liste chaînée : first & rest
- seq sur une séquence vide renvoie nil
  - manière idiomatique de tester si qqch est vide : (if (seq coll) ...)
  - très bon avec un if-let
- next = (comp seq rest)

Seqables

- Collections Clojure et Java (Map et tous les Iterables)
- Implems de clojure.lang.Seqable
- Séquences elles-mêmes
- Tableaux
- Chaînes (CharSequence généralement)

API séquence

- Toutes ces fonctions appellent implicitement seq sur leur arguments
  - applicable à tout ce qui est seqable
  - cons
  - map
  - reduce
  - concat, take, drop, take-while, take-nth, drop-last etc.

Construction de séquence

- cons, lazy-seq, lazy-cat
- bas niveau : mieux vaut utiliser les HOF ou for

API Collection

- Plus fragmentée que l'API séquence
- Fragmentation par «aspect»
  - Collection : conj, count
  - Associative : assoc, get, find
  - Indexed : nth
  - Reversible : rseq
  - Stack : pop, peek
  - Set : disj
  - Map : dissoc
  - Sorted : subseq, rsubseq

Support

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

for le couteau suisse

- «seq comprehension»
- combine map/mapcat/filter/take-while
- produit cartésien

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

Programmation concurrente

Types références

- Point de mutabilité, d'articulation
  - Limite les parties mobiles
- Porte la sémantique de synchronisation

Types références

	indépendant	coordonné
synchrone	atom	ref
asynchrone	agent	?

Similarités

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

*Si agent en erreur

Uniform update model

- Le pattern (alter r f arg2... argN) est central
- update-in l'utilise aussi
- Evite la création de closures
  - Plus esthétique, «fluide»
  - Un peu plus performant

STM

- (dosync ...) délimite une transaction
- Une transaction n’échoue jamais*
- Pas de notification de retry

*Sauf si limite de rejeu atteinte, ou erreur de valdiation, dans ce cas exception.

Les 3 temps

Une transaction est bornée par deux instants :

- son départ (start point)
- sa fin (commit point)
- entre les 2 est le temps de la transaction

Entre les deux

- deref d'une ref :
  - si modifiée, valeur courante «en transaction»
  - sinon valeur telle qu'au départ
  - pas de contrainte sur la valeur au moment du commit

Entre les deux

- ensure d'une ref :
  - comme deref sauf que garantie que la ref ne sera pas modifiée par une autre transaction

Entre les deux

- ref-set ou alter d'une ref :
  - exécute un deref
  - met à jour la valeur «en transaction»
  - garantie de non-modification tierce

Entre les deux

- commute d'une ref :
  - exécute un deref
  - met à jour la valeur «en transaction»
  - au moment du commit, recalcul de la valeur à partir de la dernière valeur hors transaction !

Code smell

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

- Rien ne garantit qu'au moment du commit (test @x) soit encore vrai
- «Ensure» si cette cohérence est importante

commute ou alter ?

- Quand l'ordre des opérations n'a pas d'importance
- ni la cohérence entre les valeurs en fin de transaction
- alors commute est préférable

Agents et transactions

- Les send et send-off sont reportés à après le commit
  - De même à l'intérieur d'un agent
    - mais release-pending-sends au cas où

Les autres derefables

- delay, promise, futures
- plus dataflow et parallélisme que concurrence
- delays intéressants combinés aux atoms/refs etc. pour réduire la concurrence
- realized? et deref + timeout

OO : les meilleurs morceaux

OO ?

- Définir des abstractions
- Participer dans ces abstractions
- Réutiliser des implémentations

Problèmes

- Un POJO est-il une abstraction ?
- 1 seul héritage = 1 seule réutilisation
  Délégation fastidieuse (et perte du this)
- Adapter une nouvelle abstraction à un type existant (-> mort par wrappers)
- Pourquoi n'y en a-t-il que pour le 1er argument et son type ? Pourquoi ???

Multiméthodes

- Corrigent tous les problèmes (sauf POJO)
- Leur flexibilité se paye en perfs
- Système optionnel de hiérarchie compliqué
- Pas de groupement logique

Protocoles

- Performants à très performants
- Basés sur le type du 1er argument (= this)
  Nécessitent donc des types
- Groupement de méthodes

Création de types

- Type anonyme (et fermeture) : reify
- Type bas niveau : deftype
  - champs privés mutables + sémantique de synchronisation
- Type «métier» : defrecord
  - pas de champs mutables
  - comme une map -> POJO

defrecord

- Accès aux champs :
  - (.field obj)
  - (:field obj) <- préférentiellement
- Les maps sont toujours une abstraction

Interop

dot dot dot

- (NomClasse. arg1 ... argN)
- (.champ obj)
- (set! (.champ obj) 42)
- (.méthode obj arg1 ... argN)
- NomClasse/champStatique
- (NomClasse/methodeStatique arg...)
- (set! NomClasse/champStatique 42)

-> et 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))
```

Implémenter

- Par ordre de préférence :
  - reify/deftype/defrecord (interfaces/protos)
  - proxy (classes abstraites ou non sans champs protected)
  - gen-class (:impl-ns est très important)
  - Java (sérieusement)
- cf. schéma dans le livre

Appel depuis Java

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

Type hints

- ^NomClass (expression)
- (set! *warn-on-reflection* true)
- Typage en amont
- Attention aux macros mangeuses de type hints !

Macros

Le macro club

- N’écrivez pas les macros
- Utilisez les sagement
  - Contrôle
  - Sucre syntaxique
  - Optimisation

Homoiconique

- le code est représenté par des types «normaux»
- le code est donc manipulé par des fonctions normales
- une macro est juste une fonction code-> code exécutée à la compilation

Macroexpansion

- Evaluation d'une macro = macroexpansion
- macroexpand et macroexpand-1
- De l'extérieur vers l'intérieur

Syntax-quote

- Comme quote sauf :
  - ns-qualifie les symboles, les ::mots-clés et les ::mots/clés mais pas les symboles#, ni les %N ni les :mots-clés ou les :mots/clés.
  - remplace les ~expr et les ~@expr par le résultat de leur évaluation

Texte à trou

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

Hygiène

- syntax-quote et les autogensyms#
- check compilateur : locales sans ns
- éviter les interactions avec le code utilisateur

Contre-exemple

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

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