Clojure

Julien Ponge – julien.ponge@insa-lyon.fr (mailto:julien.ponge@insa-lyon.fr)

Table of Contents

Qu’est-ce ?

Clojure (http://clojure.org/) est un *dialecte moderne* de la famille des LISP (http://en.wikipedia.org/wiki/Lisp_(programming_language)). Clojure s’exécute sur la machine virtuelle Java (JVM), mais il propose aussi une compilation vers JavaScript et le Microsoft CLR (.Net).

Domaines d’application

Si Clojure reste un langage de programmation généraliste, il s’applique avec pertinence dans les domaines suivants :

- big data / traitement de données massives (logs, finance, capteurs, etc),
- résolution et modélisation de problèmes algorithmiques, mathématiques et statistiques,
- applications fortemment concurrentes.

Caractéristiques

Clojure possède les caractéristiques suivantes.

1. Il est fonctionnel.
2. Il est *optionellement* typé.
3. Les références et collections sont immutables. Il existe des moyens limités de disposer de variables que nous n'aborderons pas aujourd'hui.
4. Il est homoiconique : la représentation du langage (des listes) est aussi un type primitif du langage.
5. Il peut utiliser du code Java existant.

LISP est un acronyme pour **"LISt Processor"**. La structure de donnée principale est la liste, qui sert également pour la représentation du langage. Si `'(1 2 3)` est une liste, `(add 1 2 3)` est une *forme* correspondant à un appel de fonction `add` avec les arguments `1, 2, 3`. L'homoiconicité du langage vient du fait qu'un appel de fonction se matérialise par une liste (aussi appellée S-Forme). N'est-ce pas intrigant ?

Il est conseillé d'ouvrir http://clojure.org (http://clojure.org), http://clojure.org/cheatsheet (http://clojure.org/cheatsheet) et http://clojuredocs.org/ (http://clojuredocs.org/) dans votre navigateur favori.

Hello world

Récupérez la dernière version stable de Clojure depuis http://clojure.org (http://clojure.org).

Le `.jar` principal est exécutable directement :

```
$ java -jar clojure-1.8.0.jar
Clojure 1.8.0
user=>
```

En lançant Clojure de la sorte, vous êtes dans le **REPL** (*read-eval print loop*) qui vous permet d'exécuter du code Clojure de façon interactive.

Commençons par un *"hello world"* raffiné :

```
(println "Hello, world!")
```

Vous devriez avoir un affichage console proche de :

```
$ java -jar clojure-1.8.0.jar
Clojure 1.8.0
user=> (println "Hello")
Hello
nil
user=>
```

Vous pouvez quitter le REPL avec `^D`.

Il est bien entendu possible de placer du code Clojure dans des fichiers, de préférence avec l'extension `.clj` :

hello_world.clj

```
(ns hello-world)  ; Commentaire

; Commentaire
(println "Hello, world!")
```

Pour exécuter ce programme, rien de plus simple :

```
$ java -jar clojure-1.8.0.jar hello_world.clj
```

La déclaration `ns` permet de spécifier un espace de nommage. Par défaut celui-ci est `user`, la preuve dans un REPL :

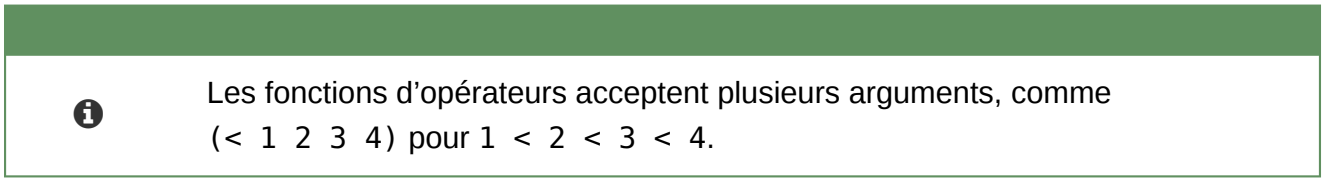
```
user=> (ns foo/bar)
nil
foo/bar=>
```

Opérateurs

Clojure utilise comme tout bon LISP une *"notation polonaise inversée"* : `1 + 2` s'écrit :

```
(+ 1 2)
```

`+` est donc une fonction, comme `-`, `/`, `=`, `and`, `or`, etc.



Les fonctions d'opérateurs acceptent plusieurs arguments, comme `(< 1 2 3 4)` pour `1 < 2 < 3 < 4`.

Exercice

Dans un REPL, faites les calculs suivants :

1. `(1 + 5) * 10`
2. `2 + 1) - (3 * 4 * 5 / 3`
3. Que fait `(inc 1)` ?
4. Que fait `(dec 2)` ?

`def`, `fn` **et** `defn`

`def` permet de nommer une valeur :

```
(def a 10)
; (...)
(+ a 2 a)
```

`fn` permet de définir une fonction *anonyme* :

```
(def twice
  (fn [x] (* 2 x)))

(twice a)
```

Enfin `defn` est une contraction de `def` et `fn` pour définir des fonctions nommées :

```
(defn twice [x]
  (* 2 x))

(twice a)
```

Une fonction peut supporter différentes *"formes"* :

```
(defn maximum
  ([x] x )
  ([x y] (if (> x y) x y) ))

(maximum 1)
(maximum 1 2)
```

Des arguments variables peuvent se capturer dans une liste avec `&` :

```
(defn many-args [x y & args]
  (println (str "x=" x))
  (println (str "y=" y))
  (println (str "args=" args)))

(many-args 1 2 3 4 5)
```

Exercice

Définissez la fonction Fibonacci récursive :

```
fib(n) :=
  si n=0 ou n=1:
    n
  sinon:
    fib(n-1) + fib(n-2)
```

Testez `fib(0)`, `fib(1)`, `fib(2)`, `fib(10)` et `fib(30)`.

N'oubliez pas que `f(x - a) + b` se traduit par une forme `(+ (f (- x a)) b)`.

Il existe une forme plus générale que `if`, nommée `cond` :

```
(defn what? [x]
  (cond
    (= x 1) "one"
    (= x 2) "two"
    (> x 100) "at least hundred"
    :else   "something else" ))

(what? 1)
(what? 200)
(what? 50)
```

Exercice

Réécrire `fib` avec `cond` au lieu de `if`.

Listes et traitements de listes

Une liste se définit comme suit :

```
(def my-list '(1 2 3 4 5 6))
```

Exercice

Avec la définition de `my-list` qui précède, que font les fonctions suivantes ?

1. `(cons 100 my-list)`
2. `(conj my-list 10)`
3. `(conj my-list 20 30 70 77)`
4. `(conj my-list '(20 30) '(70 77))`
5. `(rest my-list)`
6. `(first my-list)`
7. `(empty? my-list)`
8. `(distinct '(1 2 2 2 2 3))`
9. `(take 2 '(1 2 3 4 5))`
0. `(count '(1 2 3))`

Ces appels modifient-ils `my-list` ? Qu'en déduisez-vous ?

`filter`, `map`, `reduce`

Ces 3 opérations sont des *combinateurs* de base usuels en programmation fonctionelle.

`filter` permet *d'exclure* des éléments suivant un prédicat :

```
(filter even? ;  1
  [1 2 3 4 5 6 7 8 9 10]) ;  2
; Donne (2 4 6 8 10)
```

1. `even?` est une fonction de `clojure/core`, comme `println`, `even?`, etc.
2. `[ … ]` permet de définir des vecteurs, dont le comportement (séquence) est pour l'essentiel similaire à une liste. Dans le détail, un vecteur propose un accès aléatoire aux éléments en `O(log_32(n))` par rapport à leur index, contrairement à une liste (`O(n)`).

`map` transforme une collection en appliquant une fonction à chaque élément :

```
(map
  (fn [x] (+ 10 x))
  '(1 2 3 4))
; Donne (11 12 13 14)
```

Pour des fonctions anonymes courtes, on peut aussi utiliser une forme lambda :

```
(map
  #(+ 10 %)           ;  1
  '(1 2 3 4))
; Donne (11 12 13 14)
```

1 `%` est l'unique argument, sinon utilisez `%1, …, %n`.

`reduce` permet de calculer une réduction depuis une collection et une valeur initiale. Chaque pas intermédiaire du calcul nécessite une fonction de réduction, un *accumulateur* et la prochaine valeur de la collection.

```
(reduce + 0 '(1 2 3 4 5 6 7 8 9 10))
; Donne 55
```

Question

Comment `reduce` utilise t-elle `+`, `0` et les divers éléments de la collection ?

Exercice

Avec `reduce`, faites la concaténation des éléments de `'("hello" "world" "!")` en séparant les éléments par un espace.

Composition de fonctions

La fonction `apply` permet d'appliquer des paramètres à une fonction, ce qui permet de faire des invocations *dynamiques* :

```
(apply odd? [1])
(apply + [1 2 3])
(apply + '(1 2 3))
```

Clojure propose la fonction `or` :

```
(or false false true)
; true
```

Soit la fonction `either` suivante qui compose 2 prédicats `pred1` et `pred2` :

```
(defn either [pred1 pred2]
  (fn [x] (or
    (apply pred1 [x])
    (apply pred2 [x]) )))
```

`pred1` et `pred2` sont des fonctions qui prennent 1 paramètre et retournent un booléen. `either` renvoie une fonction qui compose ces 2 fonctions avec `or`.

Testez :

```
(defn check []
  (let [p1 odd? p2 even?]                ;  1
    (let [always-true (either p1 p2)]
      (println (always-true 1))
      (println (always-true 2)) )))

(check)
```

1 Nous en profitons pour introduire `let` qui assigne localement un nom à une valeur sous la forme `[nom1 valeur1 nom2 valeur2 …]`

Séquences "fainéantes" et infinies

La séquence `'(1 2 3)` est une séquence dite *"déjà évaluée"* dans le sens où son existence mémoire passe par une liste de 3 éléments dont les valeurs sont données.

Pour de grands flux de données, il n’est pas forcémment pratique d’évaluer des collections à leur création. Certaines peuvent ne pas tenir en mémoire, comme la suite des nombres entiers.

Pour cela, il existe des collections fainéantes depuis des valeurs ou des fonctions. Ces collections n’évaluent leurs éléments que quand nécessaire.

Dans un REPL, que se passe t-il pour le code suivant ?

```
(def h (repeat "Hello"))

(take 10 h)
(take 10 (range 0 10000))
(take 10 (range 0 10000 10))

(reduce (fn [acc next] (str acc ">" next "< ")) "" (take 5 h))
```

Quid de ceci ?

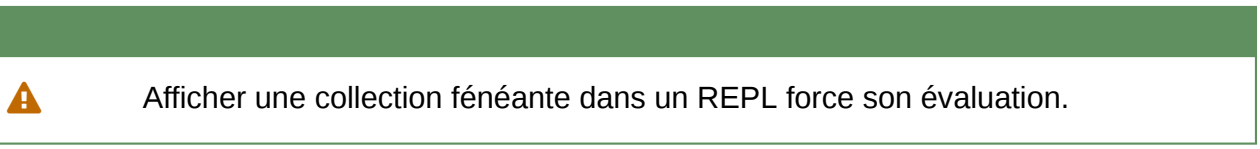
```
(str h)
```

Exercice

La fonction `repeatedly` prend en paramètre une fonction sans arguments qui renvoie une nouvelle valeur à chaque appel. Il existe 2 formes : `(repeatedly n fn)` (séquence de `n` éléments) et `(repeatedly fn)` (séquence infinie).

La fonction `(rand-int n)` donne un nombre aléatoire entre `0` et `n`.

1. Faites une fonction qui retourne une séquence fénéante de 10 000 entiers aléatoires entre 0 et un paramètre `n` de cette fonction.
2. Faites une fonction qui calcule la moyenne arithmétique des 1000 premiers éléments d'une telle séquence.
3. Faites une fonction qui calcule la moyenne arithmétique des 1000 premiers éléments pairs d'une telle séquence infinie.
4. Avec une liste fénéante des 30 premiers entiers naturels construite depuis la fonction `range`, utilisez `map` pour construire une liste des 30 premiers éléments de la suite de Fibonacci. Quel est l'impact de l'évaluation sur les temps d'exécution ?



Exercices finaux

Les 2 exercices suivants vous feront travailler votre façon d'aborder les problèmes avec un langage fonctionnel. La principale difficulté est d'oublier vos réflexes issus de langages avec des variables (C, Java, etc) et d'utiliser à la place des définitions récursives de fonctions.

Vous serez sans doute surpris par la concision des solutions !

Exercice 1 : parenthèses équilibrées

Écrire une fonction récursive `balanced?` qui renvoie `true` si les parenthèses d'une chaîne de caractères sont *équilibrées*, et `false` sinon:

```
(balanced? "(())")
; true

(balanced? "")
; true

(balanced? ")(")
; false

(balanced? "(()")
; false
```

Pour faire cela, notez qu'une chaîne de caractères est une séquence, et que donc :

```
(first "abc")  ; 'a'
(rest "abc")   ; "bc"
(empty? "")    ; true
(empty? "a")   ; false
```

> ⚠ Les caractères `(` et `)` se désignent avec `\(` et `\)`, et non pas `'('` ou `')'` car ce sont des cas spéciaux dans la grammaire de Clojure.

Exercice 2 : un problème "de sac à dos"

Écrivez une fonction récursive qui **compte** les différentes façon de donner le change pour un montant, étant donné une liste de dénominations de pièces.

Exemple : il y a 3 façons de donner le change pour 4 euros si vous avez des pièces de 1 et 2 euros : `{1, 1, 1, 1}`, `{1, 1, 2}` et `{2, 2}`, donc `(compte 4 '(1 2))` donne 3.