Thème 3 : Macros en Clojure

Dans ce thème nous présentons les *macros* Clojure pour la méta-programmation.

Exercice 1 : Macros simples

```
(ns theme03-macros.ex01-macros-simples
  (:use midje.sweet))
```

Dans cet exercice, nous abordons la définition de *macros* simples, pour certaines déjà définies dans la bibliothèque standard de Clojure.

Question 1

Nous souhaitons définir la macro `unless` dont le principe d'évaluation est le suivant :

```
(unless <cnd>
  <body>)
```

- évalue la condition `<cnd>`
  - si cette dernière est fausse (au sens de Clojure) alors le corps (une séquence d'expressions) est évaluée
  - et sinon la valeur de l'expression est `nil`.

```
;; à modifier
(defmacro unless [& _] nil)

(fact "`unless` s'évalue selon le principe annoncé."

      (unless (zero? (* 2 21))
          :ok) => :ok

      (unless (zero? 0)
          :ok) => nil

      (unless (zero? 42)
              (* 2 3) ;; expression "perdue"
              (+ 4 2)) => 6)
```

Question 2

Nous voulons définir une macro `show` qui permet de tracer les évaluations d'expressions, une technique utile pour le déboguage par exemple.

Le principe d'évaluation est le suivant :

`(show <expr>)`

- affiche l'expression `<expr>` suivi de `=> <val>` où `<val>` est le résultat de son évaluation
- la valeur retournée finalement est `<val>`.

Par exemple :

```
(show (* 21 2))
```

Affiche :

```
(* 21 2) => 42
```

et retourne `42`

Un autre exemple :

```
(defn factorial [n]
  (loop [k n, res 1]
    (if (zero? k)
      res
      (recur (dec k) (show (* k res))))))


(factorial 5)
```

affiche :

```
(* k res) => 5
(* k res) => 20
(* k res) => 60
(* k res) => 120
(* k res) => 120
```

et retourne `120`

```
;; à modifier
(defmacro show [_] nil)

(fact "`show` évalue l'expression."
      (show 42) => 42
      (show (* 21 2) => 42)
      (show (reduce + 0 (range 5))) => 10)

(fact "`show` affiche les bonnes informations."

      (with-out-str
        (show 42)) => "42 => 42\n"

      (with-out-str
        (show (* 21 2))) => "(* 21 2) => 42\n"

      (with-out-str
        (show (reduce + 0 (range 5))))

      => "(reduce + 0 (range 5)) => 10\n")
```

Question 3

La macro `when-let` est utile lorsque l'on recherche une information, par exemple dans une *map* et que l'on souhaite sinon propager le cas `nil`.

Par exemple :

```
(when-let [n (get {:a 1 :b 2 :c 3} :a)]
  (println "J'ai trouvé: n=" n)
  (* n 2))
```

```
J&#x27;ai trouvé: n= 1


2
```

```
(when-let [n (get {:a 1 :b 2 :c 3} :d)]
  (println "J'ai trouvé: n=" n)
  (* n 2))
```

```
nil
```

Redéfinir cette macro sous le nom `my-when-let`.

```
;; à compléter
(defmacro my-when-let [& _] nil)
```

```
(fact "`my-when-let` fonctionne comme `when-let`."

      (my-when-let [n (get {:a 1 :b 2 :c 3} :a)]
                   (println "J'ai trouvé: n=" n)
                   (* n 2))
      => (when-let [n (get {:a 1 :b 2 :c 3} :a)]
           (println "J'ai trouvé: n=" n)
           (* n 2))

      (my-when-let [n (get {:a 1 :b 2 :c 3} :d)]
                   (println "J'ai trouvé: n=" n)
                   (* n 2))
      => (when-let [n (get {:a 1 :b 2 :c 3} :d)]
           (println "J'ai trouvé: n=" n)
           (* n 2)))
```

Exercice 2 : dotimes

```
(ns theme03-macros.ex02-dotimes
  (:use midje.sweet))
```

Dans cet exercice, nous élaborons une variante de la macro `dotimes` permettant de répéter des calculs un certain nombre de fois. La macro `dotimes` étant présente dans la bibliothèque standard de Clojure, nous appellerons notre variante `my-dotimes`.

Question 1

Essayer de traduire en une expression *loop* le programme suivant :

— répeter k fois pour k de 0 à 4
  — afficher “k =” puis la valeur de k
  — afficher “fin de la boucle” à la fin du dernier tour
— retourne `nil` en fin de boucle

L’affichage attendu est le suivant :

```
k = 0
k = 1
k = 2
k = 3
k = 4
fin de la boucle
```

Et la valeur retournée attendue est `nil`.

Question 2

En déduire une définition de la macro `my-dotimes` telle que :

```
(my-dotimes [k 5]
  (println "k =" k)
  (when (= k 4)
    (println "fin de la boucle")))
```

effectue le même traitement que précédemment.

```
;; à modifier
(defmacro my-dotimes [& _] nil)
```

Remarque : après la définition, vous essaierez l’expression suivante :

```
(macroexpand-1
  '(my-dotimes [k 5]
     (println "k =" k)
     (when (= k 4)
       (println "fin de la boucle"))))
```

Les tests suivants devraient passer finalement.

```
(fact "`my-dotimes` reproduit le comportement attendu."
      (with-out-str
        (my-dotimes [k 5]
          (println "k =" k)
          (when (= k 4)
            (println "fin de la boucle"))))

      => (str "k = 0\nk = 1\nk = 2\nk = 3\n"
              "k = 4\nfin de la boucle\n"))


(fact "`my-dotimes` fonctionne comme la fonction standard `dotimes`."
      (with-out-str
        (my-dotimes [k 5]
          (println "k =" k)
          (when (= k 4)
            (println "fin de la boucle")))))
```

```
      => (with-out-str
           (dotimes [k 5]
             (println "k =" k)
               (when (= k 4)
                 (println "fin de la boucle")))))
```

Exercice 3 : macros de filetage

```
(ns theme03-macros.ex03-threading-macros
  (:use midje.sweet))
```

Cet exercice permet de comprendre le fonction des macros de filetage (*threading macros*) de Clojure:

- *thread-first* `->` qui permet de chaîner des appels de fonctions en connectant implicitement leur premier argument
- *thread-last* `->>` qui chaîne les appels selon leur dernier argument.
- *thread-as* `as->` qui permet de chaîner les appels de fonction par une connection explicite

Clojure est un langage qui encourage la définition de petites fonctions effectuant des traitements simples, que l'on compose ensuite pour résoudre des problèmes de plus en plus complexe. Si cette approche possède de nombreux avantages, elle s'accompagne aussi des quelques désagréments. En particulier, les enchaînements d'appels de fonction peuvent devenir assez difficile à lire.

Question 1

Prenons un premier exemple :

```
(get (conj (conj [1 2 3 4] 5) 6) 3)
```

```
4
```

Le fil de contrôle *"de l'intérieur vers l'extérieur"* rend la lecture de ce type de code assez difficile.

La macro `->` permet de rendre l'expression ci-dessous beaucoup plus lisible :

```
(-> [1 2 3 4]
    (conj 5)
    (conj 6)
    (get 3))
```

```
4
```

Le premier argument des appels de fonction `conj` et `get` a été "enlevé" et remplacé par le résultat de l'expression qui la précède dans la chaîne. Ce "filetage" systématique du premier argument donne son nom à la macro : *thread-first*.

Compléter le test suivant en utilisant la macro *thread-first*.

```
(fact "Exemple de filetage thread-first"

      (get (assoc (assoc {:a 1 :b 2} :c 3) :d 4) :c)

      ;; à compléter
      => (-> nil))
```

Question 2

La macro `as->` (dite *thread-as*) est un peu moins fréquemment utilisé et facilite cependant la lecture du code :

```
(as-> [1 2 3 4] v
   (conj v 5)
   (conj v 6)
   (get v 3))
```

```
4
```

Ici la variable `v` correspond à chaque fois au résultat de l'évaluation de l'expression qui précède. On a donc en fait quelque chose de la forme suivante :

```
(let [v [1 2 3 4]
      v (conj v 5)
      v (conj v 6)
      v (get v 3)]
  v)
```

```
4
```

Compléter le test suivant en utilisant la macro `as->`.

```
(fact "Exemple de filetage explicite."

      (get (assoc (assoc {:a 1 :b 2} :c 3) :d 4) :c)

      ;; à compléter
      (as-> nil m))
```

Compléter le test suivant en utilisant un `let` "fileté" :

```
(fact "Exemple de let fileté."

      (get (assoc (assoc {:a 1 :b 2} :c 3) :d 4) :c)

      ;; à compléter
      => (let [m nil
               m nil]
           m))
```

Question 3

La macro `->` est très souvent utilisé dans le monde des collections (vecteurs, tables et ensembles) car les fonctions qui les manipulent prennent le plus souvent la collection en premier argument.

En revanche dans le monde des séquences, c'est en général le dernier argument qui précise la séquence sur laquelle on travaille.

```
(reduce + 0 (filter even? (range 10)))
```

```
20
```

En général les compositions par le dernier argument sont un peu plus lisibles, mais Clojure propose tout de même la macro `->>` permettant de rendre un peu plus explicite l'enchaînement des calculs :

```
(->> (range 10)
     (filter even?)
     (reduce + 0))
```

```
20
```

Ici, c'est bien le dernier argument qui a été "enlevé" et remplacé par la valeur de l'expression précédente dans la chaîne, d'où le nom de la macro: *thread-last*.

Compléter le test suivant en utilisant la macro *thread-last*.

```
(fact "Exemple de filetage thread-last"

      (reduce * 1 (filter #(< 10 % 30) (map #(* % %) (range 10))))

      ;; à compléter
      => (->> nil))
```

Question 4

Encore une fois, la macro `as->` permet de mieux se rendre compte de ce qui s'est produit dans le *thread-last* :

```
(as-> (range 10) s
   (filter even? s)
   (reduce + 0 s))
```

```
20
```

Et il y a encore une correspondance directe avec le `let` "fileté":

```
(let [s (range 10)
      s (filter even? s)
      s (reduce + 0 s)]
  s)
```

```
20
```

Compléter le test suivant avec un *thread-as* :

```
(fact "Exemple de filetage thread-as"

      (reduce * 1 (filter #(< 10 % 30) (map #(* % %) (range 10))))

      ;; à compléter
      => (as-> nil s))
```

Compléter le test suivant avec un `let` "fileté" :

```
(fact "Exemple de filetage thread-as"

      (reduce * 1 (filter #(< 10 % 30) (map #(* % %) (range 10))))

      ;; à compléter
      => (let [s nil
               s nil]
           s))
```

Question 5

Proposer une définition de la macro `my-as->` effectuant le même travail que `as->` en passant par une expansion sous forme de `let` "fileté".

```clojure
;; à compléter
(defmacro my-as-> [expr v & body] nil)

(fact "`my-as->` fonctionne comme `as->`."

      (my-as-> [1 2 3 4] v
          (conj v 5)
          (conj v 6)
          (get v 3))

      => (as-> [1 2 3 4] v
             (conj v 5)
             (conj v 6)
             (get v 3))

      (my-as-> (range 10) s
          (filter even? s)
          (reduce + 0 s))

      => (as-> (range 10) s
             (filter even? s)
             (reduce + 0 s)))
```

Question 6 (plus difficile)

En utilisant `my-as->` donner une définition de la macro `my->` effectuant le même travail *thread-first* que `->`.

Remarque : il est fortement conseillé, dans cette question et la suivante, de définir une ou plusieurs fonctions auxiliaires pour décrire les transformations de code source à effectuer.

```clojure
;; à modifier
(defmacro my-> [expr & body] nil)

(fact "`my->` fonctionne comme `->`."

      (my-> [1 2 3 4]
         (conj 5)
         (conj 6)
         (get 3))

      => (-> [1 2 3 4]
             (conj 5)
             (conj 6)
             (get 3)))
```

Question 7 (plus difficile)

En utilisant `my-as->` donner une définition de la macro `my->>` effectuant le même travail *thread-last* que `->>`.

```clojure
;; à modifier
(defmacro my->> [expr & body] nil)

(fact "`my->>` fonctionne comme `->>`."
```

```
(my->> (range 10)
       (filter even?)
       (reduce + 0))

=> (->> (range 10)
        (filter even?)
        (reduce + 0)))
```

Exercice 4 : macros de comprehension

```
(ns theme03-macros.ex04-comprehensions
  (:use midje.sweet))
```

Dans cet exercice nous discutons des *expressions de compréhension* qui existent dans certains langages de programmation, en particulier Python mais également Haskell ou Scala.

En Python par exemple, nous pouvons générer les carrés des premiers nombres pairs avec l'expression suivante :

```
(x * x for x in range(13) if x % 2 == 0)
```

Cette expression se nomme un *générateur par compréhension* en Python.

Si Clojure ne supporte pas directement les compréhensions, il est possible de les introduire grâce à une définition de macro. De fait, la macro `for` de la bibliothèque standard de Clojure permet d'exprimer le même générateur dans une syntaxe proche :

```
(for [x (range 13)
      :when (even? x)]
  (* x x))
```

```
(0 4 16 36 64 100 144)
```

Dans cet exercice, nous souhaitons définir une variante simplifiée dont le nom est `myfor`.

Les compréhensions peuvent introduire plusieurs *variables de compréhensions*.

Par exemple, l'expression suivante :

```
(for [i (range 1 5)
      j (range 1 5)
      :when (< i j)]
  [i, j])
```

```
([1 2] [1 3] [1 4] [2 3] [2 4] [3 4])
```

génère les couples `[i, j]` sur l'intervalle $[1; 4]$ et tels que $i < j$.

Question 1

Pour construire les compréhensions, nous définissons les fonctions suivantes.

```
(defn seq-bind [s f]
 (if (seq s)
    (lazy-cat (f (first s)) (seq-bind (rest s) f))
    s))
```

```
(defn seq-ret [x]
  (list x))
```

Ces fonctions, dites *de monade séquence*, permettent malgré leur simplificité de construire des compréhensions non-conditionnelles.

Par exemple, la compréhension `(for [n (range 13)] (* n n))` peut s'écrire :

```
(seq-bind (range 13) (fn [n] (seq-ret (* n n))))
```

```
(0 1 4 9 16 25 36 49 64 81 100 121 144)
```

Définir la macro `myfor1` telle que :

```
(myfor1 [x <expr1>] <expr2>)
```

se macro-expanse en :

```
(seq-bind <expr1> (fn [x] (seq-ret <expr2>)))
```

```
;; à compléter
(defmacro myfor1 [[x expr1] expr2] nil)
```

```
(fact "La macro `myfor1` permet de faire des compréhensions simples."

      (myfor1 [n (range 6)] n) => '(0 1 2 3 4 5)

      (myfor1 [n (range 6)] (* n n)) => '(0 1 4 9 16 25))
```

Question 2

Pour construire des compréhensions conditionnelles, nous pouvons ajouter une troisième fonction pour la *monade sequence* :

```
(defn seq-fail []
  ())
```

On peut maintenant reproduire la séquence initiale

```
(for [x (range 13)
      :when (even? x)]
  (* x x))
```

```
(seq-bind (range 13) (fn [x] (if (even? x)
                                 (seq-ret (* x x))
                                 (seq-fail))))
```

```
(0 4 16 36 64 100 144)
```

Définir la macro `myfor2` telle que

```
(myfor2 [x <expr1>
         :when <cnd>] <expr2>)
```

se macro-expanse en :

```
(seq-bind <expr1> (fn [x] (if <cnd>
                              (seq-ret <expr2>)
                              (seq-fail))))
```

```
;; à compléter
(defmacro myfor2 [[x expr1 wh cnd] expr2] nil)
```

```
(fact "La macro `myfor2` permet de faire des compréhensions conditionnelles simples."

      (myfor2 [n (range 6)
               :when (odd? n)]
         n) => '(1 3 5)

      (myfor2 [n (range 7)
               :when (even? n)]
         (* n n)) => '(0 4 16 36))
```

Question 3

Pour créer des compréhensions multiples, le principe est d'emboîter les appels à `seq-bind`.

Par exemple, la compréhension :

```
(for [i (range 1 4)
      j [:a :b]]
  [i, j])
```

peut s'écrire :

```
(seq-bind
  (range 1 4)
  (fn [i] (seq-bind
            [:a :b]
            (fn [j] (seq-ret [i, j])))))
```

```
([1 :a] [1 :b] [2 :a] [2 :b] [3 :a] [3 :b])
```

On souhaite définir une macro `myfor3` permettant d'exprimer les compréhensions multiples.

On commence par définir une **fonction** `expand-myfor3` prenant en entrée un vecteur de liaisons `[[x1 expr1] [x2 expr2] ...]` et une expression `expr` et telle que :

```
(expand-myfor3 [x1 <expr1>  ... etc ...] <expr>)
```

produit une expression de la forme:

```
(seq-bind <expr1> (fn [x1] (expand-myfor3 [... etc ...] <expr>)))
```

et

```
(expand-myfor3 [] <expr>)
```

se macro-expanse en :

```
(seq-ret <expr>)
```

```
;; à compléter
(defn expand-myfor3 [& _] nil)
```

```
(defmacro myfor3 [xs expr]
  (expand-myfor3 xs expr))
```

```
(fact "La macro `myfor3` permet les compréhensions multiples."

      (myfor3 [i (range 1 4)
               j [:a :b]]
          [i, j])
```

```
=> '([1 :a] [1 :b] [2 :a] [2 :b] [3 :a] [3 :b]))
```

Question 4 (plus difficile)

En vous inspirant des questions précédentes, proposer une fonction d'expansion `expand-myfor` telle que la macro `myfor` fonctionne correctement, c'est-à-dire retourne la même compréhension que `for`.

```
;; à compléter
(defn expand-myfor [& _] nil)

(defmacro myfor [xs expr] (expand-myfor xs expr))

(fact "La macro `myfor` fonctionne comme `for`."

      (myfor [i (range 1 5)
              j (range 1 5)
              :when (< i j)]
             [i, j])

      => (for [i (range 1 5)
               j (range 1 5)
               :when (< i j)]
           [i, j])


      (myfor [i (range 0 10)
              :when (even? i)
              j (range 0 10)
              :when (odd? j)
              :when (< i j)]
             [i, j])

      => (for [i (range 0 10)
               :when (even? i)
               j (range 0 10)
               :when (odd? j)
               :when (< i j)]
           [i, j]))
```