Thème 2 : Séquences en Clojure

Dans ce thème nous présentons les *séquences* (paresseuses) de Clojure.

Ce projet propose quelques exercices sur ce thème.

Voici quelques exercices supplémentaires sur le même thème :

— *Clojure koans*: exercices 09, 11, 12, 21, 22
— *4Clojure*: exercices 17, 22 à 35, 39, 40, 41, 43, 44, 45, 49, 56, 60, 62, 64

Exercice 1 : Génération de séquences

```
(ns theme02-sequences.ex01-gen-seq
  (:use midje.sweet))
```

Dans cet exercice, nous construisons des séquences paresseuses *lazy sequences* en utilisant des générateurs définis par l'intermédiaire de la macro `lazy-seq`. On construit ici des suites numériques simples.

Question 1

Définir une fonction `const-seq` prenant un élément `n` et retournant la séquence paresseuse `(n n n n ...`

Remarque : cette fonction s'appelle `repeat` dans la bibliothèque standard de Clojure.

```
(declare const-seq)
```

```
(fact "`const-seq` est la bonne séquence."
      (take 4 (const-seq 1)) => '(1 1 1 1)
      (take 3 (drop 1000 (const-seq :bip)))
      => '(:bip :bip :bip)
      (reduce + (take 1000 (const-seq 1)))
      => 1000)
```

Question 2

Définir la fonction `ints` générant la sequence `(n n+1 n+2 ...)` à partir d'un entier `n`.

```
(declare ints)
```

```
(fact "`ints` génère la bonne séquence."
      (take 5 (ints 1)) => '(1 2 3 4 5)
      (first (drop 99 (ints 1))) => 100)
```

```
(fact "la factorielle peut être calculée de façon assez originale."
      (reduce * (take 6 (ints 1))) => 720)
```

Question 3

Définir la fonction `fibo-seq` générant la séquences des nombres de *fibonacci*:

```
(0 1 1 2 3 5 8 ...)
```

Remarque : la fonction pourra prendre deux arguments optionels : les deux derniers nombres de fibonacci calculés.

```
(declare fibo-seq)

(fact "Les premiers termes de `fibo-seq` sont justes,
      (d'après Wikipedia."
      (take 11 (fibo-seq)) => '(0 1 1 2 3 5 8 13 21 34 55))

(fact "le 100-ème fibonacci est..."
      (first (drop 100 (fibo-seq))) => 354224848179261915075N)
```

Question 4

Définir la fonction `primes` permettant de générer la séquence des nombres premiers.

```
(declare primes)

(fact "Les premiers termes de `primes` sont justes."
      (take 10 (primes)) => '(2 3 5 7 11 13 17 19 23 29))

(fact "Le 1000-ème nombre premier est ?"
      (first (drop 1000 (primes))) => 7927)
```

Question 5

Redéfinir les fonctions `const-seq` et `ints` en utilisant les fonctions suivantes de la bibliothèque standard: `repeat` et/ou `iterate`.

```
(declare const-seq')

(fact "`const-seq'` retourne la bonne séquence."
      (take 4 (const-seq' 1)) => '(1 1 1 1)
      (take 3 (drop 1000 (const-seq' :bip)))
      => '(:bip :bip :bip)
      (reduce + (take 1000 (const-seq' 1)))
      => 1000)

(declare ints')

(fact "`ints'` génère la bonne séquence."
      (take 5 (ints' 1)) => '(1 2 3 4 5)
      (first (drop 99 (ints' 1))) => 100)
```

Question 6

Définir, en utilisant `repeatedly`, une fonction `rand-seq` permettant de générer des entiers aléatoires compris entre deux bornes `n` et `m` indiquées en paramètre.

Vous pourrez utiliser la fonction `rand-int` de la bibliothèque standard :

```
=> (doc rand-int)
-------------------------
clojure.core/rand-int
([n])
  Returns a random integer between 0 (inclusive) and n (exclusive).
```

```
(fact "`rand-seq` retourne bien des entiers dans les bornes fixées."
      (every? #(and (<= 3 %)
                    (< % 9)) (take 10000 (rand-seq 3 9))) => true)
```

Exercice 2 : Combinateurs map, filter et reduce

```
(ns theme02-sequences.ex02-map-filt-red
  (:use midje.sweet))
```

Dans cet exercice, nous (re)définissons les fonctions de manipulation de séquences : `map`, `filter` et `reduce`.

Question 1

Définir la fonction `mymap` qui construit la séquence `((f e1) (f e2) ...` à partir de la séquence `(e1 e2 ...`

Remarque : cette fonction, qui s'appelle `map` dans la bibliothèque standard de Clojure, doit préserver le caractère paresseux de la séquence initiale.

```
(declare mymap)
```

```
(fact "`mymap` fonctionne comme `map`."
      (map inc (range 0 10)) => '(1 2 3 4 5 6 7 8 9 10)
      (mymap inc (range 0 10)) => '(1 2 3 4 5 6 7 8 9 10))
```

```
(fact "`mymap` préserve la caractère paresseux"
      (first (drop 1000000 (mymap inc (range)))) => 1000001)
```

Question 2

Définir la fonction `myfilter` permettant de filter les éléments d'une séquence (en préservant le caractère paresseux de cette dernière).

Remarque : cette fonction se nomme `filter` dans la bibliothque standard de Clojure.

```
(declare myfilter)
```

```
(fact "`myfilter` fonctionne comme `filter`."
      (filter #(= (rem % 2) 0) (range 1 10)) => '(2 4 6 8)
      (myfilter #(= (rem % 2) 0) (range 1 10)) => '(2 4 6 8))
```

```
(fact "`myfilter` préserve la caractère paresseux."
      (first (drop 1000000 (myfilter #(= (rem % 2) 0) (range)))) => 2000000)
```

Question 3

Définir la fonction `myreduce` de réduction d'une séquence par un opérateur binaire associatif `op`, à partir d'un élément neutre `en`.

Remarque : la fonction devra être récursive terminale.

```
(declare myreduce)
```

```
(fact "`myreduce` fonctionne comme `reduce`."
      (reduce + 0 (range 11)) => 55
      (myreduce + 0 (range 11)) => 55
      (reduce (fn [acc e]
                (if (> e acc)
                  e
                  acc)) 0 '(1 3 7 3 2 9 1 2 5)) => 9
      (myreduce (fn [acc e]
                  (if (> e acc)
                    e
                    acc)) 0 '(1 3 7 3 2 9 1 2 5)) => 9)
```

Question 4

Remplacer les "trous" >_____< ci-dessous par des expressions utilisant les fonctions `map`, `filter` et/ou `reduce` .

```
(letfn [(>_____< [& more] 0)]
  (facts "A propos de map, reduce et filter."
         (>_____< (range 1 11)) => '(-2 -4 -6 -8 -10)
         (>_____< '(10 20 30 40 50 60)) => 720
         (>_____< { :a 1 :b 22 :c 31 :d 14 :e 1}) => 31))
```

Exercice 3 : Fonctions "standard" sur les séquences

```
(ns theme02-sequences.ex03-seq-cheatsheet
  (:use midje.sweet))
```

Cet exercice propose un petit jeu de piste pour se répérer sur la carte de référence ("*cheatsheet*") des fonctions de manipulation de séquences. Chaque question est composé d'un petit exercice "à trous" qu'il s'agit de compléter en utilisant la fonction de la carte de référence qui vous semble la plus *spécifiquement* adaptée au problème. Dans un deuxième temps, il est demandé de proposer une nouvelle définition de cette fonction "mystère", avec l'objectif de faire passer le même jeu de tests (on pourra faire un petit copier/coller).

Question 0

Cette question préliminaire, proposée avec son corrigée, permet de fixer le cadre de résponse des questions suivantes.

```
;;(letfn [(<_____________> [& more] 0)]

(fact "Je valide les tests suivants et donc je suis ... map"

        ;; (<_____________> second { :a 1, :b 2, :c 3}) => '(1 2 3)

        (map second { :a 1, :b 2, :c 3}) => '(1 2 3)


        ;; (<_____________> * #(* % 4) [1 2 3]) => [4 8 12])

        (map #(* % 4) [1 2 3]) => [4 8 12]) ;)
(defn mymap
  "Retourne la séquence `((f e1) (f e2) ... à partir de
  la séquence `s=(e1 e2 ...`."
  [f s]
  (if (seq s)
    (lazy-seq (cons (f (first s)) (mymap f (rest s))))
    s))
(fact "Je valide les tests suivants et donc je suis ... mymap"
      (mymap second { :a 1, :b 2, :c 3}) => '(1 2 3)
      (mymap #(* % 4) [1 2 3]) => [4 8 12])
```

Question 1

```
(letfn [(<_____________> [& more] 0)]

  (fact "Je valide les tests suivants et donc je suis ... ????????"


        (take 11 (<_____________> '(1 2 3))) => '(1 2 3 1 2 3 1 2 3 1 2)



        (take 6 (<_____________> [:a :b])) => '(:a :b :a :b :a :b)))
```

Question 2

```
(letfn [(<_____________> [& more] 0)]

  (fact "Je valide les tests suivants et donc je suis ... ????????"


        (<_____________> [:a :b :c] '(1 2 3 4)) => '(:a 1 :b 2 :c 3)
```

```
        (<_____________> (range) ["zéro" "un" "deux" "trois" "quatre"])

        => '(0 "zéro" 1 "un" 2 "deux" 3 "trois" 4 "quatre")))
```

Question 3

```
(letfn [(<_____________> [& more] 0)]
  (fact "Je valide les tests suivants et donc je suis ... ????????"

        (<_____________> even? [2 4 6 8 9 10 11 12]) => '(9 10 11 12)

        (<_____________> (fn [[a b]] (>= a b))
                         '([4 2] [5 1] [3 3] [1 2] [2 3] [3 4]))

        => '([1 2] [2 3] [3 4])))
```

Question 4

```
(letfn [(<_____________> [& more] 0)]

  (fact "Je valide les tests suivants et donc je suis ... ????????"

        (<_____________> [2 3 3 4 5 5 5 6 5 5 3 2 3 1 1])

        => '(2 3 4 5 6 5 3 2 3 1)

        (<_____________> '(:a :b :c :d :e :e :e :e)) => '(:a :b :c :d :e)))
```

Question 5

```
(letfn [(<_____________> [& more] 0)]

  (fact "Je valide les tests suivants et donc je suis ... ????????"

        (<_____________> :et [1 2 3 4]) => '(1 :et 2 :et 3 :et 4)

        (<_____________> 0 '(:a :b :c :d :e)) => '(:a 0 :b 0 :c 0 :d 0 :e)))
```

Question 6

```
(letfn [(<_____________> [& more] 0)]
```

```
(fact "Je valide les tests suivants et donc je suis ... ????????"

      (<_______________> odd? [1 1 1 2 2 3 3]) => '((1 1 1) (2 2) (3 3))

      (<_______________> odd? [1 3 5 2 4 5 3]) => '((1 3 5) (2 4) (5 3))

      (<_______________> #(= % :c) [:a :b :c :d :e :c :c :f :g])

      => '((:a :b) (:c) (:d :e) (:c :c) (:f :g))))
```

Exercice 4 : Filmania (miniprojet)

```
(ns theme02-sequences.ex04-filmania
  (:use midje.sweet))
```

```
#'user/ex04-filmania
```

Dans cet exercice, nous mettons en application les séquences sur une base de données de films de cinéma `ml-latest-small` diffusée gratuitement par le laboratoire de recherche *Grouplens* (http://grouplens.org/datasets/).

Nous travaillerons dans le cadre d'un projet *filmania* dont le squelette est disponible sur le site de l'UE.

L'objectif est de décortiquer la base de données et d'effectuer des statistiques sur les films contenus.

Première partie : base de films (movies.csv)

Question 0 : préparation de la base

La première étape consiste à charger la base de données en mémoire, et à son nettoyage éventuel.

Pour cette étape, nous utilisons des fonctions prédéfinies dans le squelette du projet :

- la fonction `csv-seq` qui retourne une séquence à partir d'un fichier contenant un table encodée en `CSV` (*Comma-Separated Values*).
- la fonction `parse-movie` retourne sous la forme d'une *map* un enregistrement de film de la base (en nettoyant les informations si nécessaire).
- la fonction `movie-map` qui construit la table des films à partir d'un fichier CSV.

On travaillera dans le fichier `core.clj` (point d'entrée du projet) avec une variable `movies` définie ainsi :

```
(def movie-filename "resources/ml-latest-small/movies.csv")

(def movies (movie-map (rest (csv-seq movie-filename))))
```

(le fichier `movies.csv` contient la base des films).

On pourra lister par exemple les 10 premiers films de la base avec l'expression suivante :

```
(take 10 movies)
```

(ou `(clojure.pprint/pprint (take 10 movies0))` pour plus de lisibilité).

Question 1

Dans cette question on souhaite effectuer quelques statistiques générales sur la base de films.

On souhaite répondre (par programme) aux question suivantes :

- combien y a-t-il de films dans la base ?
- combien y a-t-il de films de science-fiction ?
- combien y a-t-il de films de romance ?

Question 2

Définir (et tester) les fonctions suivantes :

- `all-genres` retournant l'ensemble des genres de film de la base
- `films-by-genre` permettant d'obtenir la base composée uniquement des films dont le `genre` est spécifé en paramètre.

Répondre à nouveau aux questions précédentes.

Question 3

Définir (et tester) la fonction `card-genres` qui construit une table de "cardinalité par genre" où chaque entrée est de la forme `genre card` avec

- `genre` est un nom de genre de la base (i.e. dans `all-genres`)
- `card` est un entier représentant le nombre de films associés à ce genre.

Répondre aux questions suivantes :

- quel est le genre le plus représenté dans la base ?
- quel est le genre le moins représenté dans la base ?

Partie 2 : évaluations (ratings.csv)

Le fichier `ratings.csv` contient un ensemble d'évaluations sur les films de la base par des utilisateurs du site *Movielens* (https://movielens.org/).

Chaque entrée est une ligne du fichier composée de quatre chaînes de caractères :

```
userId,movieId,stars,clock
```

avec :

- `userId` un numéro d'utilisateur
- `movieId` le numéro du film évalué (tel qu'enregistré dans `movies.csv`)
- `stars` une note entre `0.0` et `5.0`
- `clock` qui est un *timestamp* identifiant de façon unique l'évaluation dans le temps

Question 1

Définir une variable `ratings` contenant la base des évaluations sous la forme d'une *map* clojure où chaque entrée :

- a pour clé l'*entier* identifiant l'utilisateur
- a pour valeur une *map* avec comme clé l'identifiant (entier) du film et comme valeur la note correspondant (un *double*)

Montrer les 10 premières évaluations.

Question 2

Définir une fonction `movie-avg-rating` retournant une *map* associant à chaque film de la base *movies* (identifié par son *id*) sa note moyenne dans la base *ratings*.

Répondre aux questions suivantes :

- quels sont les films les mieux notés, en moyenne, dans la base ?
- quels sont les films les moins bien notés ?
- quelle est la note moyenne de la base de film ?

Question 3

Répondre aux questions suivantes :

- quels sont les utilisateurs les plus "sympatiques" ?
- quels sont les utilisateurs les plus "critiques" ?
- quel est le film de science-fiction le mieux noté ... le moins bien noté ?

Remarque : il ne faudra pas hésiter à introduire des fonctions et structures de données utilitaires permettant d'effectuer les requêtes ci-dessus.

Question 4

Ajouter une fonction `-main` permettant de lancer votre analyse de la base avec `lein run` en affichant sur la sortie standard les réponses aux questions précédentes.

Vous ajouterez également d'autres statistiques qui vous semblent intéressantes.

Votre programme doit être interactif (en mode texte) pour permettre à l'utilisateur de choisir les statistiques qui l'intéressent.

Vous pourrez tester votre programme avec les bases plus complètes du site *movielens*. (attention, les fichiers sont assez volumineux).