À la découverte de Clojure

Introduction

Cet article a pour objectif de présenter un peu le langage de programmation Clojure, que j'ai découvert récemment, et de résumer les aspects du langage qui me semblent intéressants.

L'objectif de cet article n'est donc pas d'être un tutoriel pour ce langage, mais de vous donner éventuellement envie d'en apprendre plus, ou au contraire de réaliser que ce n'est pas du tout pour vous. Je me sens aussi obligée de prévenir qu'il s'agit d'un avis somme toute personnel d'une débutante enthousiaste mais pas d'une experte du langage ; tout ça pour dire qu'il ne s'agit pas d'une revue exhaustive des points forts et des points faibles du langage.

Ceci étant dit, présentons Clojure. Pour faire très bref, il s'agit d'un langage dérivé de Lisp et qui tourne sur la JVM (Java Virtual Machine).

Lisp est un langage réputé pour son nombre important de parenthèses, et qui est un des premiers langages à être orienté programmation fonctionnelle. Je ne m'amuserais pas à donner une définition exacte de ce qu'est la programmation fonctionnelle (je vous renvoie à la page Wikipédia pour ça), mais il s'agit en quelques mots de privilégier l'usage de fonctions « pures », c'est-à-dire sans effets de bord.

Le principe du Lisp est que tout ou presque est liste. D'ailleurs, ça veut un peu dire « LISt Processing ». Quand vous écrivez du code, vous écrivez en fait une suite de listes qui sont exécutées. Concrètement pour additionner 2 et 2 :

```
(+ 2 2)
```

Lisp est un langage préfixé, c'est-à-dire que l'opérateur (ici « + ») est toujours mis en premier. Il n'y a pas d'histoire de priorité des opérateurs : vous devez le faire explicitement. Avec des parenthèses. Par exemple, l'équivalent de « 2+2*4/3 » donne :

```
(+ 2 (* 2 (/ 4 3))
```

Des tas de gens détestent Lisp et ses dérivés à cause de ces parenthèses. Je ne vois vraiment pas pourquoi.

Clojure

Un des dérivés connus du Lisp est Scheme, qui a l'avantage d'être extrêmement simple (au sens où il y a un nombre de concepts réduits ; cela ne veut pas

forcément dire qu'il est nécessairement plus simple de programmer en Scheme que dans d'autres langages pour autant).

En programmation fonctionnelle, j'ai l'impression que la fonction factorielle a tendance à remplacer le classique « Hello World », donc voici comment elle s'implémente en Scheme :

```
(define (factorial n)
    (if (= n 0)
        1
        (* n (factorial (- n 1)))))

(factorial 6)
;; renvoie 720
```

Les trois points qui me semblent caractériser la programmation dans les langages dérivés de Lisp (en dehors des parenthèses), c'est :

- l'utilisation beaucoup plus fréquente de la récursion ;
- l'utilisation extensive et beaucoup plus souple de fonctions : une fonction peut être passée en paramètre, ou à l'inverse être créée à la volée par une autre fonction ;
- l'utilisation de listes, généralement manipulées en utilisant les primitives `cons` (ajoute un élément au début d'une liste), `car` (renvoie le premier élément d'une liste), et `cdr` (renvoie la liste, moins le premier élément).

Il me semble que les deux premiers points s'appliquent aussi à tout le reste de la programmation fonctionnelle, et que le dernier est un peu plus spécifique à la famille des Lisp.

Venons-en maintenant à Clojure. À chaque fois que j'ai essayé de m'initier au langage Scheme, je n'ai pas été spécialement effrayée par les parenthèses (un éditeur correct peut faire beaucoup pour vous), et j'ai trouvé que ce langage était génial. Je trouvais l'approche fonctionnelle attirante, et qu'il y avait une simplicité élégante dans ce langage. Malheureusement, j'avais l'impression que les exemples que je voyais consistaient surtout à des fonctions mathématiques (par exemple la classique suite de Fibonacci) ou à des exemples qui me paraissaient bien choisis pour montrer l'intérêt de la programmation fonctionnelle ; j'avais plus de mal à imaginer comment utiliser ce langage pour les projets que j'aurais pu avoir envie de coder, et j'avais l'impression qu'il était très difficile d'utiliser des bibliothèques d'usage courant (à ce sujet, Chicken Scheme me paraît néanmoins intéressant car il compile du code Scheme en C et permet d'utiliser des fonctions utilisées de bibliothèques C ; mais je ne m'en suis jamais servie personnellement).

Clojure, de son côté, a l'avantage de tourner sur la Java Virtual Machine. En dehors du bien ou du mal que l'on peut penser de Java, cela a l'intérêt important

de pouvoir utiliser toutes les bibliothèques (et *bindings*) écrites pour Java, de manière tout à fait naturelle.

Clojure se différencie des autres langages dérivés de Lisp en étant moins centré sur les listes, et en proposant d'autres structures de données utilisables directement. Cela se fait avec des ajouts syntaxiques (voir ci-dessous) qui s'éloignent un peu, par exemple, de la simplicité structurelle de Scheme mais qui permettent, à mon goût, une plus grande facilité d'accès.

Structures de données

La syntaxe de Clojure est donc un peu différente de celle des autres Lisp : en particulier, il y a autre chose que des parenthèses. Reprenons l'exemple de la fonction factorielle :

```
(defn factorial [n]
    (if (= n 0)
        1
        (* n (factorial (- n 1)))))

(factorial 6)
;; renvoie 720
```

Comme vous pouvez le constater, il y a quelques différences par rapport au Scheme dans la déclaration de la fonction et de ses paramètres (le code de la fonction lui-même étant ici identique).

D'abord, `defn` est le mot-clé permettant de définir une fonction et de lui assigner un nom. Il est également possible de créer une fonction anonyme avec `fn`, et d'utiliser `def` pour assigner une valeur à un nom.

Ensuite, les paramètres sont passés entre crochets. Pour comprendre les raisons de cette différence, il est utile de présenter très sommairement les principales structures de données de Clojure.

Liste

La liste est la structure de base des langages Lisp, même si c'est un peu moins le cas dans Clojure. Créer une liste se fait grâce à des parenthèses, et c'est en fait ce que vous faites lorsque vous programmez : l'instruction `(+ 2 (* 2 2))`, soit `2 + (2 * 2)` revient en effet à créer une liste contenant '+', '2', et une liste interne contenant '*', '2' et '2'.

Si vous tapez cela, cette liste est interprétée et la fonction '+' est appliquée avec comme paramètres '2' et le résultat de la fonction '*' appliquée avec comme paramètres '2' et '2'.

Pour créer une liste de données, vous devez donc faire en sorte que cette liste ne soit pas interprétée : `(1 2 3)` essaierait en effet d'appeler la fonction '1' — qui n'est pas un nom de symbole valide — avec comme paramètres '2' et '3'. Pour qu'une liste ne soit pas interprétée, il faut utiliser la fonction `quote` ou le raccourci : '.

```
(quote (1 2 3))
;; => (1 2 3)
'(1 2 3)
;; => (1 2 3)
```

Comme dans Scheme et Lisp, les trois fonctions qui permettent de manipuler les listes sont `cons`, qui ajoute un élément à une liste, `first`, qui renvoie le premier élément d'une liste (équivalent de `car`) et `rest` qui renvoie le reste de la liste (équivalent de `cdr`).

Il existe beaucoup d'autres fonctions utiles, mais l'objet de cet article étant de rester succinct, je ne rentrerai pas dans les détails. Ce qu'il est plus important de comprendre, c'est que les données sont représentées sous forme de liste chaînée : la liste `(1 2 3)` contient en fait le premier élément, `1`, et un pointeur vers la liste `(2 3)`, qui est elle-même constituée de l'élément `2` et d'un pointeur vers la liste `(3)`, elle-même constituée de l'élément `3` et d'un pointeur vers la liste vide.

Il est ainsi possible de créer la liste `(1 2 3)` comme suit :

```
(cons 1 (cons 2 (cons 3 nil)))
```

Cela implique que les fonctions `cons`, `first` et `rest` sont très rapides (O(1)) ; en revanche, il va être plus long d'obtenir le n-ième élément d'une liste (O(n)).

Vecteurs

Les vecteurs permettent, de leur côté, d'accéder rapidement au n-ième élément d'une liste et peuvent, pour cette raison, être plus pratiques pour un certain nombre d'utilisations.

La création d'un vecteur peut se faire soit en utilisant la fonction `vector`, soit avec les crochets `[ ]`. Il est ensuite possible d'obtenir le n-ième élément (en comptant à partir de zéro) avec la fonction `nth` :

```
(def x [1 2 3])
(nth x 1)
;; => 2
```

Clojure ajoute encore un peu de sucre syntaxique en permettant d'utiliser directement le vecteur comme fonction, prenant alors comme paramètre l'index de la valeur à renvoyer :

```
(x 2)
;; => 3
```

Ainsi, si l'on utilise les crochets lorsque l'on définit les paramètres d'une fonction, c'est parce que `defn` attend la liste des paramètres sous la forme d'un vecteur.

(En pratique, c'est peut-être un peu plus compliqué que cela, car il n'est pas possible de construire le vecteur de paramètres avec la fonction `vector`.)

Dictionnaires

Clojure ajoute une autre syntaxe (les accolades `{` et `}`) pour créer facilement des dictionnaires (*maps*) (il est également possible d'utiliser les fonctions `hash-map` ou `array-map`), qui permettent d'associer un élément à un autre. Je passerai très vite sur leur utilisation, et vais me contenter de donner un exemple d'utilisation :

```
(def m {:chien "waf", :chat "miaou"})
(m :chien)
;; => "waf"
(m :chat)
;; => "miaou"
```

On crée ici un dictionnaire `m` associant chien et chat à leurs « cris » respectifs. `:chien` et `:chat` sont les deux clés du tableau ; le `:` permet de créer des mots-clés, ce qui permet d'effectuer des comparaison plus rapidement que sur les chaînes de caractère (les clés d'un dictionnaire peuvent cependant être de n'importe quel type : nombre, chaîne, mot-clé…).

La virgule n'a, en Clojure, aucune valeur syntaxique et est considérée comme un espace blanc ; elle est simplement utilisée pour que le code soit plus lisible.

Enfin, comme pour les vecteurs, un dictionnaire peut également être utilisé comme fonction prenant en paramètre une clé. `(m :chat)` renvoie donc la valeur associée à la clé `:chat` (en l'occurrence, `"miaou"`).

Déstructuration

Pour terminer un peu sur le sucre syntaxique qu'apporte Clojure, imaginons le code suivant, qui permet de définir et afficher un point en deux dimensions :

```
(def point [4 2]])
(defn affiche-point [point]
        (println "x : " (point 0))
        (println "y : " (point 1)))
;; affiche :
;; x : 4
;; y : 2
```

Ce code marche, mais n'est pas très élégant : on doit manuellement accéder au premier élément de `point`, puis au second. Clojure permet cependant la déstructuration de variables, et `affiche-point` peut être écrit plus simplement de la façon suivante :

```
(defn affiche-point [[x y]]
        (println "x : " x)
        (println "y : " y))
```

Le mécanisme de déstructuration de Clojure est assez puissant, et fonctionne également pour les structures emboîtées, pour les dictionnaires, etc.

Tout cela fait que Clojure est moins « épuré » que peut l'être Scheme, et rend peut-être ce langage un peu plus compliqué à maîtriser ; cependant je trouve que cette « perte_» est nettement compensée par un gain de temps, de facilité et de lisibilité à l'utilisation.

Utilisation de bibliothèques Java

Comme Clojure est construit au-dessus de la JVM, il est possible d'utiliser n'importe quelle classe Java ; ce qui permet non seulement d'utiliser des bibliothèques Java dans un code écrit en Clojure, mais également de mélanger, au sein du même projet, Java et Clojure.

Pour créer un nouvel objet à partir d'une classe, deux syntaxes sont possibles :

- `(new classe paramètres)`
- `(classe.  paramètres)`

De la même manière, il existe deux syntaxes pour appeler la méthode d'un objet :

- `(.  objet méthode paramètres)`
- `(.méthode objet paramètres)`

En pratique, ce sont le plus souvent les deux dernières versions qui sont le plus utilisées, mais il s'agit essentiellement d'une question de goût.

L'exemple suivant utilise la bibliothèque Swing pour afficher une fenêtre intitulée « Hello ! » contenant « Hello, world ! » :

```
(import '(javax.swing JFrame JLabel)) ;; importe JFrame et JLabel

(def frame (JFrame. "Hello !"))
(def label (JLabel. "Hello, world !"))
(.add (.getContentPane frame) label)
;; équivaut à frame.getContentPane().add(label)
(.pack frame)
(.setVisible frame true)
```

Quelques autres fonctionnalités

J'ai présenté ici de façon très sommaire quelques unes des fonctionnalités de Clojure qui rendent, à mon avis, ce langage intéressant. Il y en a un certain nombre d'autres. Je me contenterais d'en évoquer rapidement certaines :

- les *protocoles*, qui correspondent *grosso modo* à des interfaces en Java, et qui permettent par exemple que l'appel `(foo bar)` invoque une implémentation différente en fonction du type de `bar` ;
- les *multiméthodes*, qui ont quelques similarités avec les protocoles, mais qui permettent un *dynamic dispatch* (c'est-à-dire, de choisir une implémentation différente lors de l'exécution — je ne sais pas trop la traduction recommandée de ce terme en français ?) qui n'est pas limité au type du premier argument.

Ces deux fonctionnalités (avec *defrecord*, qui permet de créer un nouveau type de données) sont ce qui permet d'avoir des fonctionnalités se rapprochant de la programmation orientée objet.

Pour terminer sur les fonctionnalités qu'apporte Clojure, il est intéressant de noter que ce langage prend le parti de décourager l'utilisation de variables muables (ou *mutable* en anglais), ce qui permet notamment d'éviter un certain nombre de problème liés à la parallélisation et à la concurrence (autrement dit, à l'utilisation d'une même zone de mémoire par deux *threads* s'exécutant en parallèle). Bien entendu, il est parfois indispensable d'utiliser des variables muables, auquel cas Clojure fournit un certain nombre de fonctionnalités pour faciliter l'accès partagé de ces variables ou la parallélisation du code, sans avoir à manipuler directement des *threads* ou à poser explicitement des verrous sur des variables.

Conclusion

Clojure est un langage que je trouve vraiment intéressant, non seulement pour les fonctionnalités qu'il apporte mais, aussi parce qu'il (ou, plus exactement, ses concepteurs) assume une certaine vision de la programmation ce qui induit une certaine cohérence plutôt qu'un agglomérat de fonctionnalités.

Le fait qu'il tourne sur la JVM (ou, tout du moins, l'implémentation principale ; il y a également des implémentations de Clojure pour CLR et pour Javascript) permet d'utiliser les bibliothèques Java existantes et n'est sans doute pas étranger au relatif succès de ce langage, même si cela n'est pas sans présenter aussi quelques inconvénients (notamment un temps de lancement un peu long et quelques problèmes techniques d'optimisation).

Pour terminer sur une conclusion plus personnelle : j'aime vraiment beaucoup Clojure, mais au-delà, je pense qu'en apprendre plus sur la «programmation fonctionnelle» (sans prétendre être une gourou) m'a vraiment apporté beaucoup. C'est une façon d'aborder les choses assez différente et qui demande un certain investissement (plus, je trouve, que l'approche orientée objet), mais je trouve que le jeu en vaut la chandelle. Bref, tout ça pour dire que la programmation fonctionnelle n'est pas juste un truc abscons de chercheurs ou de spécialistes qui utilisent des mots compliqués, et je pense qu'un langage comme Clojure (ou, par exemple, Scala, pour citer un autre langage tournant au-dessus de la JVM qui facilite la programmation fonctionnelle, même si l'approche est assez différente) peut permettre d'utiliser cette approche dans des applications «de la vie réelle».