Licence 3 d'Informatique — Langages de *script* — TP 8

Début : jeudi 30 novembre 2023

Cette avant-dernière série d'exercices en Ruby a pour but de vous familiariser, d'une part, avec les *échappements lexicaux* du langage Ruby, c'est-à-dire l'instruction `return`. Plus tard, les *échappements dynamiques* seront également à l'honneur avec les formes `catch` et `throw`. Chemin faisant, vous aurez l'occasion d'effectuer quelques révisions sur les blocs et les objets de classe `Proc`, qui sont — rappelons-le — des *fonctions calculées*. Cette série a également pour but de vous introduire aux *threads* de ce langage (la classe `Thread`), après une courte introduction aux *tables de hachage* [1] (la classe `Hash`). Nous terminerons par une démonstration introductive au langage JRuby, qui marie Ruby et Java. Pour ne pas trop ralentir la lecture de l'énoncé, les descriptions de méthodes utiles des classes `Array` et `Hash` ont été regroupées dans le § 6. Comme de coutume, vous pourrez trouver sur le serveur moodle, dans le cours :

Programmation fonctionnelle, scripts et XML

puis dans le répertoire « Travaux pratiques Ruby 2023 > for-lc-8 » :

— le présent énoncé : c'est le fichier lc-8.pdf ;

ainsi que quelques fichiers complémentaires :

— le fichier d'exemple de *thread* prime-numbers.rb ;

— le fichier d'archive .tar [2] getting-started-with-jruby.tar, qui sera utilisé au § 5.

1 Échappements : premier contact

$\Longrightarrow$ Définir une méthode globale `sumeverywhereignore` en Ruby, qui appliquée à un entier — de classe `Integer`[3] — le retourne. Lorsque cette méthode `sumeverywhereignore` est appliquée à un tableau (de classe `Array`), elle retourne la somme de tous les entiers présents dans le tableau, à quelque niveau que ce soit. Exemples :

```
sumeverywhereignore(2023)                  ⟹ 2023
sumeverywhereignore([30,11,2023])          ⟹ 2064
sumeverywhereignore([[30,11],2023])        ⟹ 2064
sumeverywhereignore([[30,[11,[2023],[]]]]) ⟹ 2064
```

1. Déjà entr'aperçues dans la série précédente de travaux pratiques, elles sont parfois appelées *dictionnaires* (notamment dans le langage Python), quoique cette appellation soit à notre avis impropre.
2. ***T****ape* ***AR****chive*.
3. . . . ou `Fixnum`, si vous utilisez une vieille version de Ruby sur votre propre matériel.

Cette méthode retourne `0` si elle est appliquée à un tableau vide. Si une donnée autre qu'un entier ou un tableau est rencontrée, cette donnée est *ignorée* :

```
sumeverywhereignore('L3')                    ⟹    0
sumeverywhereignore([30,11,['today'],2024]) ⟹ 2064
```

La technique, pour réaliser une telle méthode, est de définir une fonction locale — de classe `Proc`, rappelons qu'il n'est pas permis d'imbriquer une définition de méthode (`def`) à l'intérieur d'une autre définition de méthode — au moyen de la construction « `Proc.new do ... end` », qui prend deux arguments, le montant de la somme déjà calculée, puis l'élément rencontré. Schématiquement, voici le *look* d'une telle réalisation :

```
def sumeverywhereignore(x)
  goingeverywhere = Proc.new do |alreadysummed,x0|
    ...
  end
  goingeverywhere[0,x]
end
```

et rappelons que pour une fonction *calculée* p_0 (de classe `Proc`), la notation « p_0`[...]` » est une abréviation pour un appel de la méthode `call`, « p_0`.call(...)` ». S'il s'agit d'une fonction calculée à zéro argument, les deux notations précédentes se simplifient en « p_0`[]` » et « p_0`.call` ».

⟹ Donner à présent une variante, `sumeverywhererereject`, qui se comporte comme la méthode précédente, mais si une donnée autre qu'un tableau ou un entier est rencontrée, c'est *tout l'appel principal* de la méthode `sumeverywhererereject` qui retourne `nil` :

```
sumeverywhererereject(2023)                    ⟹ 2023
sumeverywhererereject([30,11,2023])            ⟹ 2064
sumeverywhererereject([[30,11],2023])          ⟹ 2064
sumeverywhererereject([[30,[11,[2023],[]]]])   ⟹ 2064
sumeverywhererereject('L3')                    ⟹ nil
sumeverywhererereject([30,11,['today'],2023]) ⟹ nil
```

La technique de réalisation est analogue, la fonction calculée locale pouvant mener le calcul récursif jusqu'à son terme, ou *s'échapper* de manière forcée avec une valeur d'échec en cas de rencontre d'une donnée non conforme.

⟹ Considérons à présent les deux définitions suivantes :

```
def sumeverywhereignore_v2(x)
  sumeverywhereperform(x) do 0 end
end

def sumeverywhererereject_v2(x)
  sumeverywhereperform(x) do return nil end
end
```

où il vous est demandé d'écrire la méthode globale `sumeverywhereperform` — qui admet un argument formel et s'utilise avec un bloc à zéro argument formel — de telle sorte que ces deux méthodes `sumeverywhereignore_v2` et `sumeverywhererereject_v2` se comportent *exactement* comme les deux méthodes précédentes `sumeverywhereignore` et `sumeverywhererereject`.

⟹ Comment compléter les points de suspension et réaliser l'évaluation suivante :

```
...
  2024 + sumeverywhereperform([30,11,['today'],2023]) do ... end
... ⟹ nil
```

dans laquelle nous souhaitons que la rencontre de la chaîne de caractères `today` déclenche de façon *immédiate* le rendu de la valeur `nil` comme résultat de l'évaluation précédente *toute entière* ?

2 Aller plus loin avec les échappements

Nous allons à présent ajouter à la classe `Array` des tableaux de Ruby des méthodes considérant qu'il n'existe dans le tableau qu'un seul élément satisfaisant une propriété. Bien sûr, cette propriété, sera représentée par un bloc à un argument formel. Plus précisément, voici quelles sont les méthodes à réaliser — dans la suite, `a` est le tableau auquel s'appliquent ces méthodes — :

`unique?` retourne `true` si un seul élément de `a` satisfait la propriété indiquée par le bloc, `false` dans tous les autres cas ;

`the_unique_1a` retourne, si un seul élément de `a` satisfait la propriété indiquée par le bloc, un tableau uniquement constitué de cet élément, la valeur `nil` dans tous les autres cas ;

`the_unique_st` retourne, le seul élément de `a` qui satisfait la propriété indiquée par le bloc, si cet élément est bel et bien unique, déclenche l'exception `NoUnique` dans tous les autres cas. Noter que l'exception `NoUnique` doit être une sous-classe de la classe d'exceptions `StandardError`.

Les exemples donnés ci-après vont pouvoir vous permettre des comparaisons entre les comportements de ces diverses méthodes :

```
[30,11,2023].unique? do |x| x.even? end       ⟹ true
[30,11,2023].the_unique_1a do |x| x.even? end ⟹ [30]
[30,11,2023].the_unique_st do |x| x.even? end ⟹ 30
[30,11,2023].unique? do |x| x.odd? end        ⟹ false
[30,11,2023].the_unique_1a do |x| x.odd? end  ⟹ nil
[30,11,2023].the_unique_st do |x| x.odd? end  ⟹ NoUnique raised
```

Ensuite, donner des nouvelles versions de ces trois méthodes, intitulées :

`unique_v2?` `the_unique_1a_v2` `the_unique_st_v2`

qui réalisent exactement les mêmes fonctionnalités, mais qui sont construites à partir d'une nouvelle méthode privée de la classe `Array` :

`search_for_unique(`$proc_0$`,`$proc_1$`)`

à laquelle on passe le même bloc ; $proc_0$ et $proc_1$ sont toutes deux des fonctions calculées — admettant respectivement zéro argument et un seul argument ; elles représentent ce qu'il faut réaliser s'il est impossible (resp. possible) de trouver un unique élément dans le tableau satisfaisant la propriété.

3 Tables de hachage

$\Longrightarrow$ Ajouter à la classe `Hash` la méthode `enrich_with`, telle que l'évaluation de l'expression :

$$h\texttt{.enrich\_with(}h_0\texttt{)}$$

— où h et h_0 sont tous deux des objets de classe `Hash` — ajoute à la table h les associations de la table h_0 dont les clés n'appartiennent pas à celles de h. Le résultat retourné est la table h modifiée. Exemple [4] :

```
{1 => 'Januar',2 => 'Februar',4 => 'April',5 => 'Mai',6 => 'Juni',7 => 'Juli'}.
enrich_with({1 => 'Janner',8 => 'August',9 => 'September',10 => 'Oktober',
             11 => 'November',12 => 'Dezember'}) ⟹
    {1 => 'Januar',2 => 'Februar',4 => 'April',5 => 'Mai',6 => 'Juni',
     7 => 'Juli',8 => 'August',9 => 'September',10 => 'Oktober',11 => 'November',
     12 => 'Dezember'}
```

4 *Thread*

4.1 Découverte

Rappelons qu'à l'occasion de votre dernier cours de Ruby, on vous a déjà distribué un *listing* du fichier prime-numbers.rb, réalisant la suite des nombres premiers à l'aide d'un *thread*. Remarquez :

— l'utilisation d'un échappement, au moyen de la forme `return`, pour réaliser des sorties forcées dans la méthode `no_divisor_found?` de la classe `Integer` ;

— le *thread* `$prime_number_thread`, tel que son activation provoque le calcul du nombre premier qui suit `current`, et son ajout à droite du tableau `prime_number_a`, après quoi ce *thread* se met en sommeil ; ce *thread* communique avec son environnement par le truchement des messages `:current_prime_number` et `:prime_number_a` ;

— ce *modus operandi* — c'est-à-dire, l'activation du *thread* `$prime_number_thread` et l'attente de sa prochaine mise en sommeil — peut être commandé par un appel de la fonction `next_prime_number` ; le résultat retourné par cette fonction est le nombre premier qui vient d'être calculé ;

— la fonction `next_prime_number_after` est telle que l'évaluation de l'expression :

$$\texttt{next\_prime\_number\_after(}i\texttt{)}$$

— où i est un entier naturel — retourne le plus petit nombre premier strictement supérieur à i, soit parce que cette fonction peut directement lire cette information parmi les nombres premiers déjà calculés par le *thread* `$prime_number_thread`, soit parce qu'elle lance ce *thread* autant de fois qu'il est nécessaire pour atteindre cette information.

4. Comme vous devez vous en douter, nous vous avons donné les noms des mois de l'année en allemand. Quant à « *Janner* », c'est la variante autrichienne pour « *Januar* » (*janvier*).

4.2 Programmation

Il vous est à présent demandé une réalisation analogue pour les nombres de Hamming. Un **nombre de Hamming** est un entier naturel non nul de la forme $2^a.3^b.5^c$ avec $a, b, c \in \mathbb{N}$, c'est-à-dire qu'il s'agit d'un nombre naturel qui n'admet pas de facteurs premiers autres que 2, 3 et 5. Par exemple, 6 et 8 sont des nombres de Hamming, tandis que 7 et 14 n'en sont pas. Le but de l'opération est d'obtenir un *thread* qui retourne, dans un ordre strictement croissant [5], un nouveau nombre de Hamming à chaque fois que ce *thread* est activé. L'algorithme à implanter repose sur l'idée suivante :

(i) `1` est un nombre de Hamming : c'est le nombre de Hamming courant lorsque le *thread* est défini et qu'il s'endort pour la première fois ;

(ii) si h est un nombre de Hamming, alors $2 * h$, $3 * h$ et $5 * h$ sont eux aussi des nombres de Hamming ;

(iii) le procédé décrit en (i) & (ii) permet d'obtenir tous les nombres de Hamming.

À l'intérieur du *thread* des nombres de Hamming, nous tenons à jour trois séquences, rangées par ordre croissant :

$$\begin{array}{cr} 2, 4, 6, \ldots & \text{nombres de Hamming multiples de } 2, \\ 3, 6, 9, \ldots & \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ 3, \\ 5, 10, 15, \ldots & \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ 5. \end{array}$$

Le nombre `1` ayant déjà été produit comme nombre de Hamming, le nombre suivant est obtenu en considérant la plus petite valeur parmi les éléments de tête de ces trois séquences. Cette plus petite valeur est *enlevée* desdites séquences, même si elle apparaît en début de *plusieurs* séquences. Notons h_0 le nombre de Hamming qui vient d'être obtenu, il nous reste — avant que le *thread* s'endorme — à ajouter les trois valeurs $2{*}h_0, 3{*}h_0, 5{*}h_0$ aux trois séquences mentionnées précédemment. Votre gentil animateur vous démontrera ce processus plus graphiquement et plus interactivement. Ces trois séquences seront bien évidemment implantées par des *tableaux* de Ruby. Vous pourrez aussi utiliser une *table de hachage* (de classe `Hash`) pour représenter l'aiguillage vers ces trois séquences :

```
{2 => ...,3 => ...,5 => ...}
```

Sur le modèle de ce que nous vous avons fourni dans le fichier prime-numbers.rb, nous vous demandons aussi de définir les méthodes suivantes :

`next_hamming_number` qui lance le calcul du nombre de Hamming suivant et le retourne ;

`next_hamming_number_after(`*`i`*`)` — où *`i`* est un entier naturel — retourne le plus petit nombre de Hamming strictement supérieur à *`i`*, soit parce que cette fonction peut directement lire cette information parmi les nombres de Hamming déjà calculés par le *thread* qui les gère, soit parce qu'elle lance ce *thread* autant de fois qu'il est nécessaire pour atteindre cette information.

5. C'est-à-dire, sans répétition.

5 Démonstration de JRuby

Cette partie sera entièrement à la charge de votre gentil animateur. Le fichier d'archive getting-started-with-jruby.tar consiste en un répertoire JRuby, dont les éléments sont :

— des fichiers sources en Ruby : trying-jruby.rb, second-jruby.rb et jruby-plus-2021.rb ;

— un sous-répertoire jrubytrying, comportant des textes sources en Java :

AbstractPersons.java Animal.java Imported_By_JRuby.java UsingAnimals.java

une fois que ce répertoire est installé, son adresse absolue doit être ajoutée à la variable d'environnement `CLASSPATH`, pour que Java puisse localiser les classes correspondantes.

Revenant au fichier « . . . /JRuby/jrubytrying/UsingAnimals.java », vous pouvez y voir que le langage Ruby est très souple quant à la définition de *délimiteurs* : `%r` pour les expressions régulières, `%q` (resp. `%Q`) pour les chaînes de caractères constantes (resp. calculées), introduites par défaut par des guillemets simples (resp. doubles) :

```
%q(U r afraid, ain't U?) ⟹ "U r afraid, ain't U?"
y = 2022
%Q[We r in #{y}]         ⟹ "We r in 2022"
%r!abc!                  ⟹ /abc/
```

Cette caractéristique permet à des instructions en Ruby d'être aisément incluses à l'intérieur de chaînes de caractères de Java, comme vous pouvez le remarquer.

6 Quelques rappels et précisions au sujet des tableaux et des tables de hachage

6.1 Classe `Array`

Voici quelques méthodes qui peuvent vous être utiles pour les objets de classe `Array` du langage Ruby — « `a` » désigne un objet de classe `Array` dans la suite — :

`all?` s'utilise avec un bloc à un argument formel et retourne `true` si tous les éléments du tableau satisfont la propriété donnée par le bloc, `false` sinon :

```
a.all? do |x| ... end
```

`at(n)` retourne l'élément n° *n* (ou *n* est un entier naturel) du tableau [6], ou `nil` si cet élément n'existe pas ; on utilise aussi la notation entre crochets droits [7] pour cette méthode ;

`each` s'utilise avec un bloc à un argument formel et réalise une itération sur les éléments du tableau :

```
a.each do |x| ... end
```

la valeur retournée est le tableau `a` ;

`first` retourne l'élément le plus à gauche du tableau, ou `nil` si un tel élément n'existe pas ;

`inject` s'emploie avec un argument *x* et un bloc à deux arguments formels, opère une composition du bloc par la gauche :

6. Rappelons qu'en Ruby, les positions sont comptées à partir de zéro.

7. La notation entre crochets droits est utilisable en partie gauche d'une affectation ou d'opérateurs tels que `+=` ou `-=`, ce qui n'est pas le cas de la méthode `at`.

`a.inject(x) do |y,y0| f2(y,y0) end` $\Longrightarrow$ `f2(... f2(f2(x,a[0]),a[1]),...)`

`last` retourne l'élément le plus à droite du tableau, ou `nil` si un tel élément n'existe pas ;

`length` retourne la dimension d'un tableau, c'est-à-dire le nombre de ses éléments ;

`max` retourne le grand élément du tableau [8], la valeur `nil` si le tableau est vide ;

`min` retourne le plus petit élément du tableau [8], la valeur `nil` si le tableau est vide ;

`pop` enlève l'élément le plus à droite du tableau et le retourne, n'effectue aucune opération et retourne `nil` si un tel élément n'existe pas ;

`push(`*x*`)` ajoute l'objet *x* à la fin du tableau et retourne le tableau ainsi obtenu ;

`shift` enlève l'élément le plus à gauche du tableau et le retourne, n'effectue aucune opération et retourne `nil` si un tel élément n'existe pas ;

`unshift(`*x*`)` ajoute l'objet *x* au début du tableau et retourne le tableau ainsi obtenu.

6.2 Classe `Hash`

Vous pouvez créer une table de hachage, de classe `Hash`, comme suit :

```
h = {1 => 'January',2 => 'February'}
```

tester l'appartenance d'une clé à une table h_0 par la méthode `has_key?` :

`h.has_key?(2)` $\Longrightarrow$ `true` `h.has_key?(3)` $\Longrightarrow$ `false`

et accéder à la valeur associée à une clé au moyen de la notation entre crochets droits :

`h[2]` $\Longrightarrow$ `"February"` `h[14]` $\Longrightarrow$ `nil`

cette notation pouvant aussi servir pour une mise à jour d'une valeur associée ou l'ajout d'une association clé/valeur dans une table h_0 :

h_0`[...] = ...`

La méthode `fetch` permet de retourner la valeur associée à son premier argument et de préciser la donnée à retourner si ce premier argument n'est pas une clé de la table :

`h.fetch(2,'KO')` $\Longrightarrow$ `"February"` `h.fetch(14,'KO')` $\Longrightarrow$ `"KO"`

La valeur par défaut du second argument de la méthode `fetch` est `nil`, ce qui équivaut à la convention suivie lorsqu'on emploie des crochets droits (voir les exemples précédents).

Les méthodes suivantes sont disponibles pour les objets de classe `Hash` et permettent de parcourir toutes les associations clé/valeur d'une table h_0 — dans tous les cas, c'est la table h_0 qui est retournée par ces méthodes — :

`each` s'emploie avec un bloc à deux arguments formels représentant les clé et valeur de chaque association :

h_0`.each |key,value| ... end`

`each_key` s'emploie avec un bloc à un argument formel représentant la clé de chaque association :

h_0`.each_key |key| ... end`

`each_value` s'emploie avec un bloc à un argument formel représentant la valeur associée à une clé de la table :

h_0`.each_value |value| ... end`

8. ... à condition que ces éléments soient tous comparables entre eux.