Licence 3 d'Informatique — De Scheme à Ruby en passant par Java — TP 5

Début : mercredi 25 octobre 2023

Cette séance vise à vous faire découvrir les constructions fonctionnelles du langage Java, après un ultime rappel de quelques constructions équivalentes en Scheme. Après cette mise en bouche, la suite a pour but de vous familiariser avec les outils du langage Ruby. *Trois* fichiers sources sont à télécharger :

— le fichier Scheme d'amorce for-lc-5.scm, introduisant la fonction `writeln/return`, que vous connaissez bien, à présent ;

— le fichier source CompilationTest.java, utilisé au § 1.2 ;

— le fichier Ruby d'amorce showRE-plus.rb ;

tous trois se trouvant sur le serveur moodle, dans le cours :

Programmation fonctionnelle et scripts — XML

dans le répertoire « Travaux pratiques Ruby 2023 > for-lc-5 ».

1 *Preludio*

1.1 *In* Scheme

$\Longrightarrow$ Définir la fonction Scheme `make-average-f`, telle que l'évaluation de l'expression :

`(make-average-f` x`)`

— où x est un nombre réel — retourne une fonction calculant la moyenne de son argument avec x. Exemples :

```
(define average-with-10 (make-average-f 10.0))
(average-with-10 12.0)          ⟹ 11.0
((make-average-f 12.0) 13.0)    ⟹ 12.5
```

$\Longrightarrow$ Écrire une fonction `twice-f`, telle que l'évaluation de l'expression `(twice-f` f_1`)` — où f_1 est une fonction utilisable avec un argument — retourne la fonction $f_1 \circ f_1$. Exemple :

`((twice-f (lambda (x) (+ x 1))) 2023)` $\Longrightarrow$ `2025`

1.2 *In* Java

Passant au fichier CompilationTest.java, dont le texte vous a été distribué en cours, compilez-le et exécutez-le [1], vérifiez que vous avez bien compris la technique qu'il emploie. D'autres exemples peuvent être trouvés sous moodle, sous le répertoire « Programmation fonctionnelle en Java 8 ».

$\Longrightarrow$ Définir, en Java — il s'agit d'exemples différents, dans des fichiers différents — :

- une classe `Averages` fournissant une méthode statique `makeAverageF` rendant les mêmes services que la fonction `make-average-f` du § 1.1 ;
- une classe `IntTwiceC1`, fournissant une méthode statique `intTwice` dont les deux arguments sont une fonction f_1 de type `int` $\rightarrow$ `int` et un entier x, le résultat de `intTwice` est de type `int` et est égal à $f_1(f_1(x))$;
- une classe `IntTwiceC2`, alternative à la classe précédente `IntTwiceC1`, mais fournissant une méthode statique `intTwiceV2F` dont l'argument est une fonction f_1 de type `int` $\rightarrow$ `int` et le résultat est la fonction réalisant $x \mapsto f_1(f_1(x))$, avec x de type `int` ;
- puis, dans deux nouveaux fichiers différents — `TwiceC1.java` et `TwiceC2.java` —, des méthodes statiques et génériques généralisant les deux fonctionnalités précédentes — dans les deux cas, `T` est un paramètre de classe — :
  - ⋆ les deux arguments de la méthode statique et générique `twiceV2` (dans le fichier `TwiceC1.java`) sont une fonction f_1 de type `T` $\rightarrow$ `T` et une donnée x de type `T`, le résultat de `twiceV2` est de type `T` et est égal, comme précédemment, à $f_1(f_1(x))$;
  - ⋆ l'argument de la méthode statique et générique `twiceV2F` (cette fois dans le fichier `TwiceC2.java`) est une fonction f_1 de type `T` $\rightarrow$ `T` et le résultat de cette méthode `twiceV2F` est la fonction $x \mapsto f_1(f_1(x))$, où x est un objet de classe `T`.

2 *In* Ruby

Pour cette première séance avec le langage Ruby, nous ne définirons pas de nouvelles classes, nous contentant d'utiliser quelques méthodes des classes `Float`, `Integer` [2] et `String`, ainsi que des méthodes fournies par des classes `Regexp` et `MatchData`, liées à la manipulation d'*expressions régulières* [3], qui sont par ailleurs un bon exemple d'utilisation des langages de *script*. Nous verrons ensuite comment les expressions régulières permettent d'aisément programmer des *substitutions* dans une chaîne de caractères.

2.1 *Getting Started*

Votre gentil animateur va vous démontrer la commande `ruby` — utilisable sous le système d'exploitation `Linux` —, employée pour l'exécution d'un fichier source en Ruby, le suffixe d'un tel fichier étant usuellement .rb. Vous pouvez aussi utiliser la commande `irb` [4], qui ouvre une

1. Si vous travaillez sur votre propre matériel, utilisez une version relativement récente de Java, c'est-à-dire Java 8 ou une version ultérieure.
2. Il se peut que vous ayez déjà utilisé d'anciennes versions de Ruby, dans lesquelles les entiers relatifs par défaut de classe `Fixnum`. Cette classe ne s'emploie plus dans les dernières versions de ce langage, et les entiers relatifs sont à présent de classe `Integer`.
3. Quelques auteurs francophones utilisent l'expression « expressions rationnelles ».
4. Pour « ***I****nteractive* ***R****u****B****y* ».

session interactive de l'interprète. Pour exécuter les définitions et ordres d'un fichier à l'intérieur d'une telle session, taper :

```
load 'file-name'
```

Nous vous rappelons qu'en Ruby, *tout* est objet, y compris les classes, et que la notation pointée peut s'employer avec les constantes. Pour vous en convaincre, essayer les évaluations suivantes :

```
'Ruby'.class   ⟹ ??          s.index('y')     ⟹ ??
2023.class     ⟹ ??          s.index('z')     ⟹ ??
3.14.class     ⟹ ??          String           ⟹ ??
s = 'players'  ⟹ ??          String.class     ⟹ ??
s.class        ⟹ ??          s == 'players'   ⟹ ??
s.size         ⟹ ??          s.==('players')  ⟹ ??
```

Le dernier exemple montre que la notation pointée peut s'employer systématiquement, même pour des opérateurs qui sont d'ordinaire écrits de façon infixée [5]. Par convention, un nom de méthode terminé par un point d'exclamation indique que l'application de la méthode *peut* modifier physiquement l'objet. Pour vous en convaincre, essayez la session ci-après :

```
s.capitalize ⟹ ??          s.capitalize!  ⟹ ??
s            ⟹ ??          s              ⟹ ??
```

Attention ! le fait que la méthode *puisse* modifier l'objet auquel elle s'applique ne signifie *pas* que cet objet est *toujours* modifié, comme cela vous est montré ci-après avec les méthodes `tr` et `tr!` [6] :

```
s.tr('y','Y')  ⟹ ??          s              ⟹ ??
s.tr('z','Z')  ⟹ ??          s.tr!('z','Z') ⟹ ??
s.tr!('y','Y') ⟹ ??          s              ⟹ ??
```

Remarquer que ces méthodes `capitalize!` et `tr!` retournent l'objet auquel elles viennent d'être appliquées lorsqu'une modification physique a eu lieu, et retournent la valeur `nil` dans le cas contraire. En ce qui concerne les chaînes de caractères de Ruby, les notations entre guillemets simples et guillemets doubles ne sont pas synonymes :

```
"little #{s}" ⟹ ??          'little #{s}' ⟹ ??
```

Lorsque des expressions se trouvent dans un fichier source, le chargement de ce fichier provoque l'évaluation de toutes ses expressions, mais les résultats ne sont pas affichés à l'écran, à moins d'utiliser un ordre d'écriture. À ce sujet, remarquez bien la nuance entre les fonctions `p` et `puts`, aussi bien au niveau de l'*effet* que du résultat retourné :

```
p 2023    ⟹ ??          p s    ⟹ ??
puts 2023 ⟹ ??          puts s ⟹ ??
```

et rappelons qu'une variable définie dans un fichier source est par défaut locale à ce fichier, sauf si elle est globale, c'est-à-dire que son nom commence par le caractère « `$` ». Découvrons les constantes `true`, `false`, `nil` — cette dernière ayant déjà été rencontrée dans quelques exemples précédents —, et remarquons qu'il n'y a pas réellement de classe pour les valeurs booléennes :

5. Attention ! pour des raisons dans le détail desquelles nous n'entrerons pas ici, les opérateurs d'affectation (`=`) et d'affectations abrégées (`+=`, `-=`, `*=`, `/=`, `%=`, ...) échappent à cette règle. Ce ne sont d'ailleurs *pas* des méthodes.

6. Pour « ***TR****anslate* ».

```
true       ⟹ ??        false.class ⟹ ??
true.class ⟹ ??        nil         ⟹ ??
false      ⟹ ??        nil.class   ⟹ ??
```

En fait, les expressions considérées comme fausses dans le test d'une instruction conditionnelle ou itérative sont `false` et `nil`, toutes les autres expressions étant assimilées à une valeur logique *vrai*, comme vous pouvez le voir par la session ci-dessous :

```
if 2023                          if false
  2024                             2028
else                             else
  2025                             2029
end ⟹ ??                         end ⟹ ??
if true                          if nil
  2026                             2030
else                             else
  2027                             2031
end ⟹ ??                         end ⟹ ??
```

Rappelons que la fin de ligne est interprétée comme une fin d'instruction, sauf si vous avez tapé une sous-expression qui attend une suite — par exemple, une ligne terminée par un opérateur infixé —, comme cela vous a été montré durant le cours. De même, comme les exemples précédents le montrent, certains séparateurs sont inutiles si une fin de ligne est employée à leur place. Si vous souhaitez écrire une instruction `if` sur une seule ligne, procédez ainsi :

```
if 0 then 2032 else 2033 end ⟹ ??
```

Pour finir cette section introductive, nous rappelons aussi que la syntaxe des expressions conditionnelles peut se simplifier dans des cas tels que :

```
p 'here' unless s != 'players'
```

2.2 Exercices préliminaires

Sous la direction de votre gentil animateur, vous allez :

— ajouter une méthode `bit_length` à la classe des entiers naturels, retournant le nombre de *bits* nécessaires à la représentation de cet entier naturel ; vous pouvez ainsi constater qu'il est possible en Ruby de *ré-ouvrir* une classe — en l'occurrence, la classe `Integer` — pour y ajouter une méthode ; en ce qui concerne la méthode `bit_length`, vous pouvez utiliser :

  ⋆ la fonction `Math.log`[7], telle que l'évaluation de l'expression `Math.log(a,b)` retourne le logarithme de base `b` du nombre `a`,

  ⋆ la méthode `ceil` de la classe `Float` des nombres réels, retournant le plus petit entier supérieur ou égal à un nombre réel ;

```
0.bit_length ⟹ 1          2023.bit_length ⟹ 11
```

7. Dans l'expression « `Math.log` », « `Math` » n'est pas un nom de classe, mais un nom de *module*. La différence entre ces deux notions vous sera expliquée ultérieurement.

notez également que l'objet courant s'obtient par l'identificateur `self` et n'oubliez pas — comme vous le montre l'exemple précédent — que votre méthode `bit_length` doit retourner le nombre `1` lorsqu'on l'applique à zéro [8] ;

— définir une fonction globale `fibonacci`, telle que l'évaluation de l'expression `fibonacci(n)` — où *n* est un entier naturel — retourne le n^{e} nombre de la suite de Fibonacci ; rappelons que cette suite est :

Rang	0	1	2	3	4	5	6	...
Suite de Fibonacci	0	1	1	2	3	5	8	13	...

c'est-à-dire que chaque terme de cette suite est égal à la somme des deux termes qui le précèdent ; le plus simple est de procéder par une instruction itérative : partir de `0` et `1`, puis effectuer *n* décalages vers la droite avant de donner le résultat. Votre gentil animateur vous montrera que dans un tel cas, une instruction d'*affectations* parallèles :

```
L,M = 3,1
L,M = M,L
```

vous permettra une écriture légère de cette fonction.

2.3 Expressions régulières

2.3.1 Découverte

À titre de premier essai, vous pouvez utiliser le fichier showRE-plus.rb, récapitulant beaucoup de points discutés en cours. Nous rappelons que si une chaîne de caractères *s* vient d'être filtrée par une expression régulière *r*, alors Ruby met à jour les variables globales suivantes :

`$‘` la partie de *s* située *avant* le filtrage ;

`$&` la partie la plus à gauche de *s* et filtrée par l'expression régulière *regexp* ;

`$’` la partie de *s* située après la partie filtrée ;

ce qui vous explique comment notre fonction `showRE` du fichier showRE-plus.rb met en évidence la partie d'une chaîne de caractères qui est filtrée par un motif. Nous donnons également des exemples de substitutions, précisant que :

— `sub` et `sub!` ne réalisent qu'une seule substitution alors que `gsub` et `gsub!` itèrent la recherche du motif et réalisent à chaque fois la substitution indiquée ;

— conformément à la signification du point d'exclamation utilisé à la fin d'un identificateur (*cf.* § 2.1), `sub` et `gsub` créent de nouvelles chaînes de caractères pour le résultat des substitutions alors que `sub!` et `gsub!` altèrent directement la chaîne originale lorsque la substitution est effectivement réalisée et retournent la valeur `nil` lorsqu'aucune modification physique n'a eu lieu ;

8. Pour réaliser une comparaison à zéro, vous pouvez utiliser l'opérateur général `==` de comparaison, ou la méthode `zero?` de la classe `Integer`. Cette dernière méthode ne s'applique qu'à un nombre — entier ou réel — alors que l'opérateur `==` s'applique quels que soient les types des deux opérandes, même s'il s'agit de deux types différents.

```
def showRE(target,regexp)
  # Highlights with chevrons where match occurs.
  p "#{$‘}>>>>#{$&}<<<<#{$’}" if target =~ regexp
end

p showRE(’beginning <a>Try 1</a><a>Try 2</a> ending’,/<a>.*<\/a>/)
p showRE(’beginning <a>Try 1</a><a>Try 2</a> ending’,/<a>.*?<\/a>/)

p showRE(’beginning!nning!nning!nningending’,/(.*)!\1/)
p showRE(’beginning!nning!nning!nningending’,/(.*?)!\1/)

p ’Little players!’.sub(/little/,’master’)
p ’Little players!’.sub(/little/i,’master’)

$s = ’beginning <a>Try 1</a><a>Try 2</a> ending’

p $s.sub(/(<a>.*?<\/a>)(<a>.*?<\/a>)/,’\2\1’)
p $s
p $s.sub!(/(<a>.*?<\/a>)(<a>.*?<\/a>)/,’\2\1’)
p $s
$r = /(<a>(?:.*?)<\/a>)*/
p $r.class
p $s.gsub($r,’<b>\1</b>’)
$r0 = /<a>(.*?)<\/a>/
p $s.gsub($r0,’<b>\1</b>’)

p /<a>.*<\/a>/.match($s)
p /<a>.*?<\/a>/.match($s)

m = /<a>(.*?)<\/a><a>(.*?)<\/a>/.match($s)
p m

## Dans la suite, « 0..2 » est un intervalle, de classe Range. Notez aussi qu’en pratique, la construction
## « for » n’est pas utilisée, nous ne tarderons pas à voir pourquoi.
for i in 0..2
  p m[i]
  p m.begin(i)
  p m.end(i)
end

p m.post_match
m0 = $r.match($s)
p m0
m1 = $r0.match($s)
p m1
```

Figure 1 : Fichier source showRE-plus.rb.

ainsi que quelques exemples de données de classe `MatchData`.

Dans un premier temps, assurez-vous qu'après chargement du fichier showRE-plus.rb, vous pouvez consulter les valeurs des variables `$s`, `$r`, `$r0`, et *pas* celle des variables `m`, `m0` et `m1`. Qu'en concluez-vous ? et comment y remédier ? Ensuite, il est *essentiel* que vous reconstituiez comment les expressions retournées par Ruby sont construites et, en particulier, que vous compreniez la différence entre opérateurs *gloutons* [9] — `?`, `+`, `*` — et *paresseux* [10] — `??`, `+?`, `*?`.

2.3.2 Vos premières expressions régulières

$\Longrightarrow$ Donner un motif permettant de filtrer tous les numéros des départements français, en sachant que :

- ces numéros doivent être considérés comme des chaînes de caractères et non comme des nombres : par exemple, le numéro du premier département (*Ain*) est `01`, pas « `1` » ;
- le département de numéro `20` n'existe plus [11] : il a été subdivisé en deux départements `2A` (*Corse du Sud*) et `2B` (*Haute-Corse*) ;
- après le dernier département métropolitain (`95` pour le *Val d'Oise*) viennent des départements d'Outre-Mer, avec une numérotation parfois interrompue car certains numéros ont été supprimés ou correspondent à des *territoires* d'Outre-Mer qui ne sont pas des départements. Voici la liste des *départements d'Outre-Mer* :

`971`	Guadeloupe	`973`	Guyane	`976`	Mayotte
`972`	Martinique	`974`	La Réunion		

Bien noter qu'il ne s'agit pas de savoir si une sous-chaîne d'une chaîne de caractères est un département français, mais de dire si *toute* une chaîne représente le numéro d'un département français. Par exemple, `225` n'est pas un numéro licite de département français, mais `22` et `25` le sont. Aussi, il est nécessaire d'utiliser les marqueurs `\A` et `\z` pour les début et fin d'une chaîne de caractères, comme pourra vous le démontrer votre gentil animateur. Ensuite, donner une fonction `match_french_department_number`, applicable à une chaîne de caractères, retournant `true` si la chaîne représente le numéro d'un département français, `false` sinon.

$\Longrightarrow$ Une variante, à présent : définir la fonction `match_1811_french_department_number`, telle que l'évaluation de l'expression `match_1811_french_department_number(s)` — où `s` est une chaîne de caractères — retourne `true` si `s` représente un numéro de département français utilisé en 1811 ; à ce moment-là — qui correspond presque à l'apogée du Premier Empire Français —, on comptait *130* départements français [12], qui allaient de `01`, `02`, ... à `131`, le numéro `90` n'étant pas attribué ; dans les autres cas, la fonction `match_1811_french_department_number` retourne `false` ; de même que pour l'exercice précédent, c'est *toute* la chaîne `s` qui doit être un numéro de département de 1811. Exemples :

```
match_1811_french_department_number('131') ⟹ true
match_1811_french_department_number('141') ⟹ false
```

9. « *Greedy* », en anglais.
10. « *Lazy* », en anglais.
11. Anciennement, c'était la *Corse*.
12. ... parmi lesquels les départements du *Léman* (`99`) et du *Simplon* (`127`), dont les préfectures respectives étaient Genève et Sion, aujourd'hui en Suisse.

2.3.3 Premières substitutions

$\Longrightarrow$ Donner un motif permettant de filtrer des noms d'unités utilisés en Physique. Nous n'allons faire aucun test sur les noms des unités « simples » — le mètre, le millimètre, *etc.* — que nous considérerons tout simplement comme des *mots*. Par contre, nous allons convenir qu'une unité est un produit d'unités simples, figuré par un point (« `.` »). Les exposants différents de 1 seront donnés entre parenthèses. De plus, nous n'utiliserons pas la barre de division et indiquerons les unités au dénominateur au moyen d'exposants négatifs. À titre d'exemples, voici comment nous exprimons des unités de moment de force, vitesse, viscosité cinématique, accélération, masse volumique :

Newton-mètre	`N.m`
mètre par seconde	`m.s(-1)`
mètre carré par seconde	`m(2).s(-1)`
mètre par seconde par seconde	`m.s(-2)`
milligramme par litre	`mg.l(-1)`

Là encore, c'est *toute la chaîne* qui doit représenter une unité utilisée en Physique.

Ensuite, indiquer par une substitution comment imprimer joliment de tels noms sous le traitement de texte LaTeX. La mise d'une expression en exposant s'indique par « `^{...}` », et les noms d'unités doivent apparaître en caractères romains — et non en caractères italiques mathématiques —, ce qui se réalise par « `\mathrm{...}` » ; enfin, toute l'expression doit être encadrée par des caractères « `$` » pour signaler un extrait à traiter en mode mathématique ; par exemple, « `$\mathrm{m}^{2}.\mathrm{s}^{-1}$` » produit « $\mathrm{m}^{2}.\mathrm{s}^{-1}$ ». Reprenant les exemples précédents, vous devez pouvoir réaliser les substitutions suivantes :

```
N.m         ⟶ $\mathrm{N}.\mathrm{m}$
m.s(-1)     ⟶ $\mathrm{m}.\mathrm{s}^{-1}$
m(2).s(-1)  ⟶ $\mathrm{m}^{2}.\mathrm{s}^{-1}$
m.s(-2)     ⟶ $\mathrm{m}.\mathrm{s}^{-2}$
mg.l(-1)    ⟶ $\mathrm{mg}.\mathrm{l}^{-1}$
```

Le tout sera réalisé par une fonction `process_unit` qui, appliquée à une chaîne de caractères, vérifiera qu'il s'agit d'une unité et appliquera les substitutions dans ce cas, sans altérer la chaîne de caractères originale. Sinon, le résultat sera la valeur `nil`.

$\Longrightarrow$ Utiliser des expressions régulières pour transformer une date donnée suivant le format anglo-saxon :

$$\langle \mathit{year} \rangle\text{-}\langle \mathit{month} \rangle\text{-}\langle \mathit{day} \rangle \qquad \text{ou} \qquad \langle \mathit{month} \rangle\text{-}\langle \mathit{day} \rangle\text{-}\langle \mathit{year} \rangle$$

en une date donnée suivant le format français :

$$\langle \mathit{day} \rangle/\langle \mathit{month} \rangle/\langle \mathit{year} \rangle$$

Exemples :

```
2022-09-04 ⟶ 04/09/2022
09-04-2022 ⟶ 04/09/2022
```

Plus précisément, on donnera deux fonctions réalisant ce *modus operandi* :

— `american_date_2_french_date`, qui retourne une nouvelle chaîne en cas de substitution, la chaîne originale lorsqu'aucune substitution n'a pu être réalisée ;

— `american_date_2_french_date!`, qui applique la substitution en modifiant physiquement la chaîne donnée en argument, auquel cas cette chaîne modifiée est retournée ; dans le cas contraire — aucune substitution n'a été réalisée — c'est la valeur `nil` qui est retournée ;

bien remarquer que si — à la différence des exercices précédents — les marqueurs de début et de fin de chaîne ne sont pas utilisés dans les motifs à filtrer, les substitutions peuvent porter sur une partie seulement de la chaîne.

$\Longrightarrow$ Donner une fonction `update_your_department`, telle que l'évaluation de l'expression :

`update_your_department(`*`s`*`)`

— en supposant que les assemblages de chiffres de la chaîne de caractères *`s`* sont à interpréter comme des numéros de départements français de 1811 — retourne une chaîne de caractères composée des mêmes caractères que *`s`* et apparaissant dans le même ordre, sauf pour le numéro de département `24`, qui est remplacé par le numéro `25` [13] ; *attention !* prendre garde à ne pas modifier un numéro tel que `124`, comme nous le montrons dans les exemples suivants :

```
update_your_department('24th Department')   ⟹ "25th Department"
update_your_department('124th Department')  ⟹ "124th Department"
update_your_department('The 24th place in France')
                                            ⟹ "The 25th place in France"
```

Indication Considérer que la sous-chaîne à substituer — `24` — peut apparaître en début de chaîne ou après un caractère qui n'est pas un chiffre.

2.4 Utilisation de quantificateurs possessifs

$\Longrightarrow$ Définir une fonction `target_p`, applicable à une chaîne de caractères, retournant le fragment entre la première parenthèse ouvrante et la dernière parenthèse fermante, à condition que cette dernière parenthèse fermante soit suivie d'un point. Dans les autres cas de figure, la fonction `target_p` retourne la valeur `nil`. Ne pas perdre de vue qu'une expression entre parenthèses peut inclure des fragments entre parenthèses. Voici quelques exemples, suivis des résultats à retourner [14] :

```
target_p\
('(He Could Stop the World). (The Man of Bronze) (Meteor Menace). Doc Savage.')
        ⟹ "(He Could Stop the World). (The Man of Bronze) (Meteor Menace)."
target_p('(The Polar Treasure) (The Lost Oasis). (Fear Cay)') ⟹ nil
target_p('(Yeah! (I (really) enjoy (programming in) Ruby). Don\'t you?')
        ⟹ "(I (really) enjoy (programming in) Ruby)."
```

Votre gentil animateur vous montrera que le moyen le plus simple de réaliser cette fonctionnalité repose sur l'emploi de quantificateurs *possessifs* — `?+`, `++`, `*+` —, quelques cas où le filtrage échoue étant traités de façon vraiment inefficace par les quantificateurs gloutons.

13. En 1811, le Département du Doubs était numéroté `24`, ce n'est que plus tard qu'il a été numéroté `25`, ce numéro étant encore en usage de nos jours. Quant au numéro `25`, il s'agissait en 1811 de la *Drôme*, aujourd'hui numéroté `26`. Ce décalage s'explique par le fait qu'à l'époque, le département des *Alpes Maritimes* — aujourd'hui numéroté `06` — avait un numéro bien ultérieur (`85`), c'était une conquête récente des guerres de la Révolution Française.

14. En outre, le premier exemple vous montre comment indiquer à `Ruby` de ne pas considérer la fin de ligne comme une fin d'expression. Quant au troisième exemple, il vous montre comment insérer un guillemet simple dans une chaîne de caractères délimitée par des guillemets simples. Pour insérer un caractère de contre-oblique (« \ »), le doubler.