Les expressions régulières

1. Construction

Une expression régulière est un objet de classe `Regexp` servant à définir une syntaxe (ou motif) de chaîne de caractères. Ruby crée automatiquement cet objet par `/expression rØguliŁre/` ou par `%r{ expression rØguliŁre }`. Ces deux formes syntaxiques peuvent être suivies d'options :

- `i` (case **I**nsensitive) : ignorer les majuscules/minuscules.
- `o` (Substitute **O**nce) : effectue une seule fois la substitution lorsque l'on doit intégrer une partie de résultat d'expression régulière dans une chaîne.
- `m` (Multiline **M**ode) : dans ce mot, l'instruction '`.`'

prend en compte tous les caractères, y compris les retours à la ligne.

- `x` (E**x**tended Mode) : autorise des espaces, retours à la ligne et commentaires dans l'expression régulière à des fins de facilité de lecture.

Une expression régulière est un langage en soit, il faut donc en apprendre les bases. Cependant avant de commencer, nous allons l'utiliser à minima en cherchant une suite de caractères par la fonction `match`. L'exemple ci-dessous exprime sous trois formes la recherche de la séquence de caractères `bon` :

```
puts /bon/.match( "bonjour" )
puts %r{bon}.match( "bonjour" )
puts Regexp.new( "bon" ).match( "bonjour" )
```

Ces trois instructions donnent le même résultat : bon. Si maintenant nous souhaitons ignorer les majuscules/minuscules avec ces trois formes, nous

utiliserons l'option i :

```
puts %r{bon}i.match( "Bonjour" )
puts /bon/i.match( "Bonjour" )
puts Regexp.new( "bon", "i" ).match( "Bonjour" )
```

Avec pour résultat : `Bon`.

2. Langage

Certains caractères vont servir d'opérateur, si ces caractères sont aussi à rechercher, il suffit d'utiliser un backslash (`\`). Voici les listes des principaux opérateurs :

- `^` : début de ligne.
- `$` : fin de ligne.
- `(..)` : un groupe de caractères.
- `*` : zéro ou plus le caractère ou le groupe précédent.

- `+` : au moins une fois le caractère ou le groupe précédent.
- `{m,n}` : entre `m` et `n` fois le caractère ou le groupe précédent. Il est également possible de spécifier d'avoir seulement `m` fois le caractère.
- `?` : zéro ou une fois le caractère ou le groupe précédent.
- `[ &]` : un caractère dans cet ensemble.
- `[^&]` : tout sauf un caractère de cet ensemble.
- `[n-m&]` : un intervalle de caractère de `n` à `m`.
- `n | m` : le caractère ou groupe n ou bien le caractère ou groupe `m`.

Il existe aussi des raccourcis pour désigner certains caractères :

- `.` : représente n'importe quel caractère (en utilisant l'option `m` également le retour à la ligne).
- `\d` : un chiffre (digit). `\D` pour le contraire.

- `\s` : un espace. `\S` pour le contraire.
- `\w` : un caractère de mot. `\W` pour le contraire.

Exemples :

```
date="19/07/08"
puts /\d{2}\/\d{2}\/\d{2}/.match( date )
# Sortie : 19/07/08
nom = "M.Fogg"
puts /M\.(\w*)/.match( nom )
# Sortie : M.Fogg
email = "ok@hotmail.com"
puts /[^\s]+@[^\s]+\.[^\s]{2,3}/.match( email )
# Sortie : ok@hotmail.com
```

Jusqu'ici nous n'avons fait que vérifier que notre chaîne correspondait à notre syntaxe. Les groupes ont l'intérêt de pouvoir extraire certaines parties de la chaîne. Ils sont numérotés dans l'ordre d'apparition en partant de 1. L'instruction `match` retourne en réalité un objet de type `MatchData`. Cet objet peut se comporter comme un tableau et retourner la valeur

associée à un groupe. Reprenons les exemples précédents et essayons d'extraire grâce aux groupes certaines parties :

```
date="19/07/08"
res = /(\d{2})\/(\d{2})\/(\d{2})/.match( date )
puts "Jour=#{res[1]}\nMois=#{res[2]}\nAnnee=20#{res[3]}"

nom = "M.Fogg"
puts "Bonjour Monsieur #{/M\.(\w*)/.match( nom )[1]}"

email = "ok@hotmail.com"
domaine = /[^\s]+@([^\s]+\.[^\s]{2,3})/.match( email )[1]
puts "Domaine = #{domaine}"
```

Nous obtenons en sortie :

```
Jour=19
Mois=07
Annee=2008
Bonjour Monsieur Fogg
Domaine = hotmail.com
```

3. Opérateurs

L'opérateur `=~` est un raccourci vers la fonction `match`. Il a l'avantage d'être plus court à écrire. La valeur d'un groupe peut être retrouvée par la méthode statique `last_match` de la classe `Regexp` ou bien par les variables `$1`, `$2`...

Exemple :

```
date="19/07/08"
if date =~ /(\d{2})\/(\d{2})\/(\d{2})/ then
  puts "La date comprend :"
  puts "Jour=" + Regexp.last_match(1)
  puts "Mois=" + Regexp.last_match(2)
  puts "Annee=" + Regexp.last_match(3)
end
```

Nous obtenons bien en sortie :

```
La date comprend :
Jour=19
Mois=07
```

```
Annee=08
```

Autre usage de l'expression régulière dans le bloc `case when`.

Exemple :

```
date="19/07/08"

case date
when /(\d{2})/
  puts Regexp.last_match( 1 )
end
```

Nous obtenons bien en sortie : `19`

L'opérateur `=~` est en réalité une méthode de la classe `Object` qui est commune à tous les objets Ruby. Nous expliquerons tous les détails de la programmation par classe dans le prochain chapitre. Pour l'instant, retenez qu'il est possible de surcharger cet opérateur, c'est-à-dire pour faire en sorte qu'un objet agisse de manière particulière avec cet

opérateur.

Exemple :

```
class Personne
  def initialize(nom,prenom)
    @nom = nom
    @prenom = prenom
  end
  def =~( re )
    re.match( @nom + "," + @prenom )
  end
end

p = Personne.new( "fogg", "phileas" )
if p =~ /(\w+),(\w+)/ then
  puts "ok"
end
```

Nous obtenons bien en sortie : `ok`