

REGEX

04/2023 - Jos NGUYEN

REGEX (REGular EXpression)

—

Expression régulière

Site de référence :

<https://slideplayer.fr/slide/174292/>

INTRODUCTION

Définition

- C'est une chaîne de caractère qui permet de décrire un modèle (ou pattern) de chaîne de caractères.

Utilité d'une REGEX

- 1) **MATCH** : vérifier qu'une chaîne de caractères respecte un modèle donné : la REGEX
 - Est-ce que ma chaîne commence / finit par ... ?
 - Est-ce que ma chaîne contient / ne contient pas ... ?
 - Est-ce que ma chaîne a la syntaxe d'une adresse e-mail/web ?
- 2) **EXTRACT** : extraire une partie d'une chaîne de caractères (= un groupe)
(Exemple : Je veux les paramètres param1 et param3 de l'URL ...)
- 3) **SUBSTITUTE** : remplacer une chaîne de caractères par une autre
(Exemple : modifier massivement le format d'un fichier CSV, SQL, ...)

Quels outils utilisent les REGEX ?

- La plupart des langages de programmation (Java, PHP, Perl, JavaScript, C#, ...)
- La plupart des éditeurs de texte (Eclipse, Notepad++, VI, ...)

Remarque :

Il existe des générateurs de REGEX !

- Introduction

- Syntaxe

 - Bases

 - Caractères simples
 - Caractères de limite
 - Opérateurs logiques

 - Capturer des groupes

 - Classes de caractères

 - Classes abrégées

 - Quantificateurs

 - Gourmands
 - Réticents
 - Possessifs

 - Drapeaux

- Exemples

- Exercices

 - A reconnaître
 - A construire

- Utilisation

 - Java
 - PHP
 - Perl
 - Javascript
 - Eclipse
 - Notepad++
 - Vi

- Conclusion

- Bibliographie

Syntaxe

Bases

- ▶ Caractères simples
 - ▶ `x` satisfait le caractère `x`
 - ▶ `\t` satisfait le caractère « tabulation »
 - ▶ `\n` satisfait le caractère « nouvelle ligne »
 - ▶ `\r` satisfait le caractère « retour charriot »
 - ▶ `\f` satisfait le caractère « saut de page »
 - ▶ `\e` satisfait le caractère « échappement »
- ▶ Caractères de limite
 - ▶ `^` caractérise le début de la ligne
 - ▶ `$` caractérise la fin de la ligne
 - ▶ `\b` caractérise une coupure de mot (des caractères au début ou à la fin d'un mot)
 - ▶ `\beur` satisfait `eurelis` mais pas `agitateur` (mot qui commence par `eur`)
 - ▶ `eur\b` satisfait `agitateur` mais pas `eurelis` (mot qui finit par `eur`)
 - ▶ `\B` caractérise les caractères à l'intérieur d'un mot (ni au début, ni à la fin)
 - ▶ `\Bi` satisfait `git` et `oi` (mot qui ne commence pas par `i`)
 - ▶ `i\B` satisfait `git` mais pas `oi` (mot qui ne finit pas par `i`)
- ▶ Opérateurs logiques
 - ▶ `XY` satisfait le caractère `X` suivi du caractère `Y`
 - ▶ `X|Y` satisfait le caractère `X` ou le caractère `Y`
 - ▶ `X(?!Y)` satisfait une chaîne contenant `X` non-suivi de `Y` (sans capturer `Y`)

Capter des groupes

▶ Un groupe permet de définir un sous-motif dans une expression régulière

- ▶ capturer le caractère X : (X)
- ▶ récupérer le $n^{\text{ième}}$ groupe capturé
 - ▶ en référence arrière (dans la regex): $\backslash n$
 - ▶ en substitution (hors de la regex): $\$n$

▶ Références arrières

- ▶ $\backslash n$ correspond au $n^{\text{ième}}$ groupe capturé précédemment
 - ▶ Exemple:
`<(b|i)>text</\1> satisfait text et <i>text</i>
 ne satisfait ni text</i> ni <i>text`

▶ Toujours suivre l'ordre des parenthèses

- ▶ Exemple: `(anti)((con)sti(tu)tion)n(elle(ment))`
 - ▶ Le groupe 0 contient `anticonstitutionnellement`
 - ▶ Le groupe 1 contient `anti`
 - ▶ Le groupe 2 contient `constitution`
 - ▶ Le groupe 3 contient `con`
 - ▶ Le groupe 4 contient `tu`
 - ▶ Le groupe 5 contient `ellement`
 - ▶ Le groupe 6 contient `ment`



Remarques

- ▶ Les quantificateurs nous obligent souvent à entourer de parenthèses et donc créer des groupes capturant pour signifier qu'il s'applique à un sous-motif: pour ne pas créer un groupe capturant, il faut écrire: `(?:XY)`
- ▶ Utiliser des groupes non-capturant est plus rapide
- ▶ Si à un sous-motif est capturé plusieurs fois, la valeur du groupe sera le dernier sous-motif

Classes de caractères (1/2)

▶ Classes simples

- ▶ `[abc]` satisfait a, b ou c. Equivalent à `a|b|c`

▶ Négation

- ▶ `[^abc]` satisfait tout *sauf* a, b ou c

▶ Intervalle

- ▶ `[a-z]` satisfait tous les caractères de a à z
- ▶ `[A-Z0-9]` satisfait tous les caractères de A à Z et de 0 à 9

▶ Union

- ▶ `[a-d[c-p]]` est l'équivalent de `[a-p]`

▶ Intersection

- ▶ `[b-p&&[a-d]]` est l'équivalent de `[b-d]`

▶ Soustraction

- ▶ `[a-z&&[^bc]]` est l'équivalent de `[ad-z]`
- ▶ `[a-z&&[^m-p]]` est l'équivalent de `[a-lq-z]`



Attention!

- ▶ `^` est le début de la ligne quand il est au début de la regex
- ▶ `^` est une négation dans une classe de caractères
- ▶ `[-a-z]` satisfait tous les caractères de a à z et -

Classes de caractères (2/2)

▶ Classes abrégées

- ▶ `.` satisfait tous les caractères
- ▶ `\d` est un chiffre («digit»).
Equivalent à `[0-9]`
- ▶ `\D` est un non-numérique.
Equivalent à `[^0-9]` ou `[^\d]`
- ▶ `\s` est un espacement («whitespace»).
Equivalent à `[\t\n\f\r]`
- ▶ `\S` est un non-espacement.
Equivalent à `[^\s]`
- ▶ `\w` est un alphanumérique («word»).
Equivalent à `[a-zA-Z_0-9]`
- ▶ `\W` est un non-alphanumérique.
Equivalent à `[^\w]`



Remarques

- ▶ `.` représente généralement tous les caractères mais pas le saut de ligne
- ▶ si la chaîne de caractères contient plusieurs lignes, il faudra spécifier `(.|\n)*` plutôt que `.*`
- ▶ Si on utilise un drapeau spécial (`s` - voir slide sur les drapeaux), le `.` indique qu'il contient le `\n`

Quantificateurs (1/2)

- ▶ Quantificateurs « gourmands » ou « avides » (greedy) :
cherchent le nombre *maximal* de répétitions qui autorisent le succès de la recherche
 - $X?$ satisfait le caractère X , 0 ou 1 fois
 - X^* satisfait le caractère X , 0 ou plusieurs fois
 - X^+ satisfait le caractère X , 1 ou plusieurs fois. Equivalent à XX^*
 - $X\{n\}$ satisfait le caractère X , exactement n fois
 - $X\{n, \}$ satisfait le caractère X , au moins n fois
 - $X\{n, p\}$ satisfait le caractère X , au moins n fois, mais pas plus de p fois

- ▶ Quantificateurs « réticents » ou « paresseux » (reluctant) :
cherchent le nombre *minimal* de répétitions qui autorisent le succès de la recherche

- ▶ C'est l'exacte même syntaxe que les quantificateurs « gourmands » à laquelle on ajoute ?

- ▶ Quantificateurs « possessifs » (possessive):
cherchent le nombre *maximal* de répétition mais sans assurer le succès de la recherche (sans regarder si la suite de l'expression est satisfaite dans la chaîne)

- ▶ C'est l'exacte même syntaxe que les quantificateurs « gourmands » à laquelle on ajoute +



Remarque

- ▶ Les quantificateurs possessifs sont beaucoup plus rapides pour s'exécuter

Quantificateurs (2/2)

Exemples:

Si la chaîne est `Eurelis`

Gourmand: ``

=> la chaîne satisfait la regex

le groupe 1 est `"http://www.eurelis.com">Eurelis</a`

Réticent: ``

=> la chaîne satisfait la regex

le groupe 1 est `"http://www.eurelis.com"`

Possessif: ``

=> la chaîne ne satisfait pas la regex

`(.*)>` commence par capturer `"http://www.eurelis.com">Eurelis` et ne trouve pas de `>` après

Si la chaîne est `abcdaeefag`

Gourmand: `^(.*)a([a]+).*S`

=> la chaîne satisfait la regex

le groupe 1 est `abcdaeef` et le groupe 2 est `g`

Réticent: `^(.*?)a([a]+?) .*S`

=> la chaîne satisfait la regex

le groupe 1 est vide et le groupe 2 est `b`

Possessif: `^(.*?)a([a]++) .*S`

=> la chaîne satisfait la regex

le groupe 1 est vide et le groupe 2 est `bcd`

Drapeaux (flags)

➤ Les drapeaux sont les options de l'expression régulière

- **c** : au cas où une erreur survient, ne pas réinitialiser la position de la recherche
- **g** : rechercher **g**lobalement, c'est-à-dire trouver toutes les occurrences
- **i** : **i**gnorer la casse
- **m** : **m**ultiple lines, la chaîne de caractères peut comporter plusieurs lignes
- **o** : **o**nce, n'appliquer le modèle de recherche qu'une seule fois
- **s** : **s**ingle line, considérer la chaîne de caractères comme une ligne distincte
- **x** : **x**extended, utiliser la syntaxe étendue



Attention!

- Tous ces drapeaux ne sont pas disponibles dans tous les langages
- Les plus utilisés sont:
 - **g**
 - **i**
 - **m**



Exemples

A réutiliser dans vos développements

Quelques cas courants (1/2)

▶ Code Postal en France

▶ Règles:

- ▶ ce numéro comporte 5 chiffres

▶ Expression régulière:

- ▶ `\b[0-9]{5}\b`

▶ Numéro de téléphone en France

▶ Règles:

- ▶ ce numéro comporte 10 chiffres
- ▶ le premier chiffre est toujours un 0
- ▶ le second chiffre va de 1 à 6 (1 pour la région parisienne... 6 pour les téléphones portables), mais il y a aussi le 8 (ce sont des numéros spéciaux) et le 9 (pour les lignes Free)
- ▶ ensuite viennent les 8 chiffres restants (ils peuvent aller de 0 à 9 sans problème)

▶ Expression régulière:

- ▶ `\b0[1-689]\d{8}\b`

▶ Si on ajoute la règle:

- ▶ Les 10 chiffres peuvent ou non aller par groupes de 2 séparés d'un espace, d'un point ou d'un tiret

▶ On complète notre regex existante:

- ▶ `\b0[1-689]([-.\]?\d{2}){4}\b`

Mise en oeuvre en C# => MATCH

```
// string rg = @"\b\d{5}\b";  
// string rg = "^\\d{5}$";  
string rg = "[0-9]{5}$";  
  
string ch;  
do {  
    Console.WriteLine("\nEntrez un code postal : ");  
    ch = Console.ReadLine();  
}  
while (!Regex.IsMatch(ch, rg));  
  
Console.WriteLine("\nSaisie OK : " + ch);
```

```
Entrez un code postal : a&é"  
Entrez un code postal : A1234  
Entrez un code postal : 1234F  
Entrez un code postal : 12E34  
Entrez un code postal : 123456  
Entrez un code postal : 12345  
Saisie OK : 12345
```

Mise en oeuvre en C# => MATCH

```
string ch;  
string rg = "^0[1-689]\\d{8}$";  
  
do  
{  
    Console.Write("\nEntrez un numéro de téléphone : ");  
    ch = Console.ReadLine();  
}  
while (!Regex.IsMatch(ch, rg));  
  
Console.WriteLine("\nSaisie OK : " + ch);
```

Entrez un numéro de téléphone : 0712345678

Entrez un numéro de téléphone : 1612345678

Entrez un numéro de téléphone : 0612345678

Saisie OK : 0612345678

Mise en oeuvre en C# => MATCH

```
string ch;  
string rg = "^0[1-689]([-. ]?\\d{2}){4}$";  
  
do  
{  
    Console.WriteLine("\nEntrez un numéro de téléphone : ");  
    ch = Console.ReadLine();  
}  
while (!Regex.IsMatch(ch, rg));  
  
Console.WriteLine("\nSaisie OK : " + ch);
```

Entrez un numéro de téléphone : 02,12,34,56,78

Entrez un numéro de téléphone : 02 123 456 78

Entrez un numéro de téléphone : 0612345678

Saisie OK : 0612345678

Entrez un numéro de téléphone : 01.12-34.56 78

Saisie OK : 01.12-34.56 78

Mise en oeuvre en C# => MATCH

```
string ch;  
string rg = "^0[1-689]\\d{8}$";  
  
do  
{  
    Console.WriteLine("\nEntrez un numéro de téléphone : ");  
    ch = Console.ReadLine();  
}  
while (!Regex.IsMatch(ch, rg));  
  
Console.WriteLine("\nSaisie OK : " + ch);
```

Entrez un numéro de téléphone : 02,12,34,56,78

Entrez un numéro de téléphone : 02 123 456 78

Entrez un numéro de téléphone : 0612345678

Saisie OK : 0612345678

Entrez un numéro de téléphone : 01.12-34.56 78

Saisie OK : 01.12-34.56 78

Mise en oeuvre en C#

=> EXTRACT

```
string pattern = @"\ba\w*\b";
string input = "An extraordinary day dawns with each new day.";
Match m = Regex.Match(input, pattern, RegexOptions.IgnoreCase);
if (m.Success)
{
    Console.WriteLine("Found '{0}' at position {1}.", m.Value, m.Index);
    Console.WriteLine($"> Found '{m.Value}' at position {m.Index}.");
}
```

Found 'An' at position 0.

=> Found 'An' at position 0.

Quelques cas courants (3/3)

► URL

► Règles: `protocol://(username:password@)?domain(:port)?/.*`

► Elle est composée des parties suivantes:

- Un **protocole** alphanumérique sans `_` suivi de `://`
- Eventuellement un **login** et **mot de passe** séparés par deux points (`:`) et le tout suivi de `@`
- Un domaine composé de 2 parties séparées par un `.`
 - La **1^{ère} partie** est un mot avec éventuellement un `.` ou un `-` mais sans `_`
 - La **2^{ème} partie** est composé de 2 à 4 lettres
- Eventuellement un **numéro de port** précédé de deux points (`:`)
- Eventuellement un `/` suivi de n'importe quel caractère

► Expression régulière:

► `\b[a-zA-Z\d]+://(\w+:\w+@)?([a-zA-Z\d.-]+\.[A-Za-z]{2,4})(:\d+)?(/.*)?\b`

► Adresse IP v4

► Règles:

- Elle est composé de 4 groupes de 1 à 3 chiffres, séparés par un `.`
- Chaque chiffre est compris entre 0 et 255.


► Expression régulière:


► `\b((25[0-5]|2[0-4]\d|[01]\d?\d{1,2})\.(25[0-5]|2[0-4]\d|[01]\d?\d{1,2}))\b`

Mise en oeuvre en C# => REPLACE

L'exemple suivant définit une expression régulière, `\s+` qui correspond à un ou plusieurs caractères d'espace blanc. La chaîne de remplacement, «
», les remplace par un seul caractère d'espace.

C#

 Copier

 Exécuter

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string input = "This is  text with  far too  much  " +
                       "white space.";
        string pattern = @"\s+";
        string replacement = " ";
        string result = Regex.Replace(input, pattern, replacement);

        Console.WriteLine("Original String: {0}", input);
        Console.WriteLine("Replacement String: {0}", result);
    }
}

// The example displays the following output:
//      Original String: This is  text with  far too  much  white space.
//      Replacement String: This is text with far too much white space.
```

Mise en oeuvre en C# => REPLACE

```
string ch = "06.12-34.56 78";  
string pattern = "[.-]";  
string replacement = " ";  
string result = Regex.Replace(ch, pattern, replacement);  
  
Console.WriteLine("Original String: {0}", ch);  
  
Console.WriteLine("\nReplacement String: {0}", result);
```

Original String: 06.12-34.56 78

Replacement String: 06 12 34 56 78

Utilisation des REGEX

Mise en œuvre : JAVASCRIPT

```
function checkRegex(maChaine) {  
    // Match  
    if(/^([a-zA-Z0-9]*)$/gi.test(maChaine)) {  
        alert("OK");  
        // ...  
    }  
  
    // Extract  
    var groups = new RegExp("^(.*<div id='result'>([0-9]+)>(.*)</div>.*$","gi").exec(maChaine);  
    if (groups) {  
        var resultNumber = groups[1];  
        var result = groups[2];  
        alert("Result " + resultNumber + " = " + result);  
        // ...  
    }  
  
    // Substitute  
    maChaine = maChaine.replace(new RegExp("\\\\((([0-9]+), ('[^']*')+), ([0-9]+)\\\\)"), "($3,$2,$1)");  
    alert(maChaine);  
}
```


CONCLUSION

- ▶ Une bonne expression régulière est une expression régulière:
 - ▶ Concise
 - ▶ Ecrire le strict minimum pour que cette expression régulière soit nécessaire et suffisante
 - ▶ Claire
 - ▶ Être capable de comprendre facilement cette expression régulière, même en la regardant longtemps après
 - ▶ Qui correspond précisément au besoin
 - ▶ Bien définir le besoin (les règles)
 - ▶ Pas d'erreur possible en fonction de différentes chaînes de caractères en entrée



A reconnaître (1/2)

► A quoi correspondent ces expressions régulières ?

► `[a-zA-Z_0-9]+ [0-9]* ?:`

- Elle permet de trouver un mot suivi d'un espace lui-même suivi éventuellement d'un nombre, puis éventuellement d'un espace, et enfin du signe deux points comme dans « Exercice 1: ». On aurait pu aussi l'écrire: `\w+ \d* ?:`

► `(?:\d{2}/){2}\d{4}`

- Elle permet de trouver une date au format `jj/mm/aaaa` sans capturer de groupe

► `\d{2}(-|/)\d{2}\1\d{4}`

- Elle permet de trouver une date au format `jj/mm/aaaa` ou `jj-mm-aaaa`

► `(\w+)\W+\1`

- Elle permet de trouver un mot répété comme dans « Agitateur de de technologies »

► `<img\s+src="([^\"]*?)"`

- Elle permet de capturer dans \$1 la source d'une image

► `(?:https?|ftp)://[^\/]+(/.*)`

- Elle permet de capturer dans \$1 l'URL absolue sans le nom de domaine

➤ Trouver les expressions régulières qui permettent de:

- Vérifier que le login saisi contient bien entre 5 et 12 caractères alphanumériques (y compris -)
 - `^[\w-]{5,12}$`
- Définir un décimal: un nombre, négatif ou positif, qui contient ou non un point ou une virgule pour séparateur des décimales, dont le 0 avant le point est facultatif si le décimal est entre 0 et 1 et le caractère + est facultatif (exemples: +1.5 / -,34 / 0,5 / -2 / 1.
contre-exemples: ++1 / ,1,2 / 4+ / -.3)
 - `^[+-]?[d*[,]?d*$` qui accepte une chaîne vide
 - `^[+-]?(?:[d+[.,]?d*|d*[.,]?d+)$` qui n'accepte pas les chaînes vides
- Récupérer la valeur du paramètre param1 de l'URL dans un lien type href="..."
 - `\?.*?param1=([^\&'"]*)`
- Valider que la chaîne comporte 2 fois de suite un même nombre séparé par un signe d'opération mathématique (exemple: 5-6+6*2/3)
 - `(\d+)[-+*/]\1`
- Valider qu'une balise XML encadrant un texte est bien fermée et a au moins 2 attributs
 - `<(\w+)(?:\s+[^\s]+\s*){2,}>[<]*</\1>`
- Vérifier que le chemin est de la forme /default/main/.../.../WORKAREA/work mais qu'aucun des répertoires entre main et WORKAREA n'est config ou import
 - `^/default/main/(?!config|import).*?/WORKAREA/work$`

Bibliographie

- ❏ Wiki Eurelis:
<http://wiki.intranet.eurelis.net/index.php/Regex>
- ❏ Test en ligne d'expressions régulières:
<http://www.fileformat.info/tool/regex.htm>
<http://tools.ap2cu.com/regex>
- ❏ Cours et exemples d'expressions régulières:
<http://www.regular-expressions.info>
- ❏ Expressions régulières avancées:
http://benhur.telug.uqam.ca/SPIP/inf6104/article.php?id_article=95&id_rubrique=6&sem=Semaine%204
- ❏ Expression régulières en Java:
<http://java.sun.com/javase/7/docs/api/java/util/regex/Pattern.html>
- ❏ Expressions régulières en PHP:
<http://www.php.net/manual/fr/reference.pcre.pattern.syntax.php>
- ❏ Expressions régulières en Perl:
http://sylvain.lhullier.org/publications/intro_perl/chapitre10.html
http://www.univ-orleans.fr/webada/html/selfhtml_fr/cgi/perl/langage/expresreg.htm
- ❏ Expressions régulières en Javascript:
<http://www.javascriptkit.com/javatutors/redev.shtml>

MERCI !