

perlrequick

Table des matières

1	NAME/NOM	1
2	DESCRIPTION	1
3	Le guide	2
3.1	Reconnaissance de mot simple	2
3.2	Utilisation des classes de caractères	3
3.3	Reconnaître ceci ou cela	4
3.4	Groupement et hiérarchie	5
3.5	Mémorisation de la correspondance	5
3.6	Répétitions et quantificateurs	6
3.7	Plus de correspondances	6
3.8	Recherche et remplacement	7
3.9	L'opérateur de découpage : split	8
4	BUGS	8
5	VOIR AUSSI	8
6	AUTEUR ET COPYRIGHT	8
6.1	Remerciements	8
7	TRADUCTION	8
7.1	Version	8
7.2	Traducteur	9
7.3	Relecture	9
8	À propos de ce document	9

1 NAME/NOM

perlrequick - Les expressions rationnelles Perl pour les impatientes

2 DESCRIPTION

Ce document décrit les bases nécessaires à la compréhension, la création et l'utilisation des expressions rationnelles ou régulières (abrégé en regex) en Perl.

3 Le guide

3.1 Reconnaissance de mot simple

L'expression rationnelle la plus simple est juste un mot ou, plus généralement, une chaîne de caractères. Une regex constituée d'un mot est reconnue dans (ou correspond avec) toutes les chaînes qui contiennent ce mot :

```
"Salut tout le monde" =~ /monde/; # correspondance
```

Dans cette instruction, `monde` est la regex et les `//` qui l'entourent demandent à perl de rechercher une correspondance dans la chaîne. L'opérateur `=~` applique la recherche à la chaîne placée à gauche et produit la valeur vraie si la regex correspond ou la valeur fausse sinon. Dans notre cas, `monde` correspond au quatrième mot de "Salut tout le monde". Donc l'expression est vraie. Ce concept a plusieurs usages.

Des expressions de ce type sont utiles dans des conditions :

```
print "Correspondance\n" if "Salut tout le monde" =~ /monde/;
```

Le sens de l'expression peut être inversé en utilisant l'opérateur `!~` :

```
print "Pas de correspondance\n" if "Salut tout le monde" !~ /monde/;
```

La chaîne littérale dans la regex peut être remplacée par une variable :

```
$mot = "monde";
print "Correspondance\n" if "Salut tout le monde" =~ /$mot/;
```

Si vous voulez chercher dans `$_`, la partie `$_ =~` peut être omise :

```
$_ = "Salut tout le monde";
print "Correspondance\n" if /monde/;
```

Finalement, les délimiteurs par défaut `//` pour une recherche de correspondance peuvent être remplacés par des délimiteurs arbitraires en les préfixant par un `'m'` :

```
"Salut le monde" =~ m!monde!; # correspondance, délimité par '!'
"Salut le monde" =~ m{monde}; # correspondance, remarquez la paire '{}
"/usr/bin/perl"  =~ m"/perl"; # correspondance après '/usr/bin',
                        # '/' devient un caractère ordinaire
```

Les regex doivent correspondre *exactement* à une partie de la chaîne pour être reconnue :

```
"Salut le monde" =~ /Monde/; # pas de correspondance, casse différente
"Salut le monde" =~ /e m/;   # correspondance, ' ' est un caractère ordinaire
"Salut le monde" =~ /monde /; # pas de correspondance, pas de ' ' à la fin
```

La reconnaissance a lieu le plus tôt possible dans la chaîne :

```
"Salut le monde" =~ /l/;     # reconnaît le 'l' dans 'Salut
"La peste est là" =~ /est/;  # reconnaît le 'est' dans 'peste'
```

Certains caractères ne peuvent être utilisés tels quels dans une regex. Ces caractères, appelés **meta-caractères**, sont réservés pour des notations spéciales dans les expressions rationnelles. Les voici :

```
{ } [ ] ( ) ^ $ . | * + ? \
```

Un meta-caractère peut-être utilisé en le préfixant par un backslash (une barre oblique inversée) :

```
"2+2=4" =~ /2+2/;      # pas de correspondance, + est un meta-caractère
"2+2=4" =~ /2\+2/;     # correspondance, \+ est traité comme un + ordinaire
'C:\WIN32' =~ /C:\\WIN/; # correspondance
"/usr/bin/perl" =~ /\usr\bin\perl/; # correspondance
```

Dans ce dernier exemple, le symbole divisé / est aussi préfixé par un backslash car il est utilisé comme délimiteur par l'expression rationnelle.

Les caractères ASCII non affichables sont représentés par des **séquences d'échappement**. Les exemples courants sont \t pour une tabulation, \n pour un passage à la ligne et \r pour un retour chariot. Les octets quelconques sont représentés par une séquence d'échappement en octal (comme \033) ou en hexadécimal (comme \x1B):

```
"1000\t2000" =~ m(0\t2)      # correspondance
"chat"        =~ /\143\150\x61\x74/ # correspondance,
                                     # mais 'chat' est écrit bizarrement
```

Les expressions rationnelles sont traitées quasiment comme des chaînes entre guillemets. Donc l'interpolation des variables fonctionne :

```
$foo = 'son';
'caisson' =~ /cais$foo/; # correspondance
'sonnet'  =~ /${foo}net/; # correspondance
```

Dans toutes les expressions rationnelles qui précèdent, si la regex est reconnue quelque part dans la chaîne, on considère qu'il y a correspondance. Pour spécifier où doit avoir lieu la reconnaissance, vous pouvez utiliser les meta-caractères d'**ancrage** ^ et \$. L'ancrage ^ est reconnu au début de la chaîne alors que l'ancrage \$ est reconnu à la fin de la chaîne ou juste avant une fin de ligne à la fin de la chaîne. Quelques exemples :

```
"housekeeper" =~ /keeper/;      # correspondance
"housekeeper" =~ /^keeper/;     # pas de correspondance
"housekeeper" =~ /keeper$/;     # correspondance
"housekeeper\n" =~ /keeper$/;   # correspondance
"housekeeper" =~ /^housekeeper$/; # correspondance
```

3.2 Utilisation des classes de caractères

Une **classe de caractères** définit un ensemble de caractères acceptables en un point particulier de l'expression rationnelle. Une classe de caractères s'exprime par une paire de crochets [...] contenant l'ensemble des caractères acceptables. Voici quelques exemples :

```
/rame/;          # reconnaît 'rame'
/[clr]ame/;     # reconnaît 'came', 'lame' ou 'rame'
"abc" =~ /[cab]/; # reconnaît 'a'
```

Dans la dernière instruction, bien que 'c' soit le premier caractère de la classe, le premier endroit où cette expression rationnelle peut être reconnue est le 'a'.

```
/[oO][uU][iI]/; # reconnaît 'oui' indépendamment de la casse
                 # 'oui', 'Oui', 'OUI', etc.
/oui/i;         # reconnaît aussi 'oui' indépendamment de la casse
```

Le dernier exemple démontre l'usage du modificateur 'i' qui permet une mise en correspondance indépendante de la casse (majuscule/minuscule).

Les classes de caractères ont elles aussi leurs caractères normaux et spéciaux mais ce ne sont pas les mêmes qu'à l'extérieur d'une classe. Les caractères spéciaux sont -] \ ^ \$. Pour les rendre normaux, il faut les préfixer par \ :

```
/[\]c]def/; # correspond à 'def' ou 'cdef'
$x = 'clr';
/[$x]ame/;  # correspond à 'came', 'lame' ou 'rame'
/[\$x]ame/; # correspond à '$ame' or 'xame'
/[\\$x]ame/; # correspond à '\ame', 'came', 'lame' ou 'rame'
```

Le caractère spécial '-' agit à l'intérieur d'une classe comme un opérateur d'intervalle. Donc les classes peu maniables [0123456789] et [abcde...xyz] deviennent [0-9] et [a-z] :

```
/item[0-9]/; # reconnaît 'item0' ou 'item1' ... ou 'item9'
/[0-9a-fA-F]/; # reconnaît un chiffre hexadécimal
```

Si '-' est le premier ou le dernier caractère d'une classe de caractères, il est traité comme un caractère ordinaire.

Le caractère ^ est spécial en première position de la classe. Il indique alors une **classe de caractères complémentaire** qui reconnaît tous les caractères sauf ceux présents entre les crochets. Qu'elle soit de la forme [...] ou [^...], une classe de caractères doit correspondre à un caractère sinon la reconnaissance échoue. Donc :

```
/[^a]at/; # ne reconnaît ni 'aat' ni 'at', mais reconnaît
           # 'bat', 'cat', '0at', '%at', etc.
/[^0-9]/; # reconnaît un caractère non numérique
/[a^]at/; # reconnaît 'aat' ou '^at'; dans ce cas '^' est ordinaire
```

Perl propose plusieurs abréviations pour des classes de caractères courantes :

- \d est un chiffre et est équivalent à
[0-9]
- \s est un blanc et est équivalent à
[\ \t\r\n\f]
- \w est caractère *mot* (alphanumérique ou _) et est équivalent à
[0-9a-zA-Z_]
- \D est la négation de \d; il représente tout autre caractère qu'un chiffre
- [^0-9]
- \S est la négation de \s
- [^\s]
- \W est la négation de \w
- [^\w]
- Le point '.' reconnaît n'importe quel caractère sauf "\n".

Les abréviations \d\s\w\D\S\W peuvent être utilisées à l'extérieur ou à l'intérieur d'une classe de caractères. Quelques exemples :

```
/\d\d:\d\d:\d\d/; # reconnaît une heure au format hh:mm:ss
/[\d\s]/; # reconnaît un chiffre ou un blanc
/\w\W\w/; # reconnaît un caractère mot suivi d'un caractère
           # non mot, suivi d'un caractère mot
/..rt/; # reconnaît deux caractères quelconques suivis de 'rt'
/fin\./; # reconnaît 'fin.'
/fin[.]/; # idem, reconnaît 'fin.'
```

L'ancre \b est reconnue à la limite de mot : entre un caractère mot et un caractère non mot (entre \w\W ou entre \W\w).

```
$x = "Housecat catenates house and cat";
$x =~ /\bcat/; # reconnaît cat dans 'catenates'
$x =~ /cat\b/; # reconnaît cat dans 'housecat'
$x =~ /\bcat\b/; # reconnaît 'cat' en fin de chaîne
```

Dans le dernier exemple, la fin de la chaîne est considérée comme une limite de mot.

3.3 Reconnaître ceci ou cela

Nous pouvons reconnaître différentes chaînes grâce au meta-caractère d'alternative |. Pour reconnaître chien ou chat, nous pouvons utiliser la regex chien|chat. Comme précédemment, perl essayera de reconnaître la regex le plus tôt possible dans la chaîne. À chaque position, perl essayera la première possibilité : chien. Si chien ne correspond pas, perl essayera la possibilité suivante : chat. Si chat ne convient pas non plus alors il n'y a pas correspondance et perl se déplace à la position suivante dans la chaîne. Quelques exemples :

```
"chiens et chats" =~ /chien|chat|rat/; # reconnaît "chien"
"chiens et chats" =~ /chat|chien|rat/; # reconnaît "chien"
```

Bien que `chat` soit la première possibilité dans la seconde regex, `chien` est reconnue plus tôt dans la chaîne.

```
"chat"      =~ /c|ch|cha|chat/; # reconnaît "c"
"chat"      =~ /chat|cha|ch|c/; # reconnaît "chat"
```

En une position donnée, la possibilité qui est retenue est la première qui permet la reconnaissance de l'expression. Ici, toute les possibilités correspondent dès le premier caractère donc c'est la première qui est retenue.

3.4 Groupement et hiérarchie

Les meta-caractères de regroupement `()` permettent de traiter une partie d'une regex comme une seule entité. Une partie d'une regex est groupée en l'entourant de parenthèses. L'expression `para(pluie|chute)` peut reconnaître `para` suivi soit de `pluie` soit de `chute`. D'autres exemples :

```
/(a|b)b/;      # reconnaît 'ab' ou 'bb'
/^(a|b)c/;     # reconnaît 'ac' au début de la chaîne ou 'bc' n'importe où

/chat(on|)/;   # reconnaît 'chaton' ou 'chat'.
/chat(on(s)|)/; # reconnaît 'chatons' ou 'chaton' ou 'chat'.
               # Notez que les groupes peuvent être imbriqués

"20" =~ /(19|20|)\d\d/; # reconnaît la possibilité vide '()\d\d',
                       # puisque '20\d\d' ne peut pas correspondre
```

3.5 Mémorisation de la correspondance

Les meta-caractères de regroupement `()` permettent aussi la mémorisation de la partie reconnue de la chaîne. Pour chaque groupe, la partie reconnue de la chaîne va dans une variable spéciale `$1` ou `$2`, etc. Elles peuvent être utilisées comme des variables ordinaires :

```
# extraction des heures, minutes, secondes
$temps =~ /(\d\d):(\d\d):(\d\d)/; # reconnaît le format hh:mm:ss
$heures = $1;
$minutes = $2;
$secondes = $3;
```

Dans un contexte de liste, une mise en correspondance `/regex/` avec regroupement retourne la liste des valeurs (`$1`, `$2`, ...). Nous pouvons donc écrire :

```
($heures, $minutes, $secondes) = ($temps =~ /(\d\d):(\d\d):(\d\d)/);
```

Si les regroupements sont imbriqués, `$1` sera le groupe ayant la parenthèse ouvrante la plus à gauche, `$2` celui ayant la parenthèse ouvrante suivante, etc. Voici une expression rationnelle complexe avec les numéros des variables de groupes indiqués au-dessous :

```
/(ab(cd|ef)((gi)|j))/;
 1 2      34
```

Les références arrières `\1`, `\2`... sont associées aux variables `$1`, `$2`... Les références arrières sont utilisées *dans* l'expression rationnelle elle-même :

```
/(\w\w\w)\s\1/; # trouve les séquences telles que 'les les' dans la chaîne
```

`$1`, `$2`... ne devraient être utilisés qu'à l'extérieur de l'expression tandis que `\1`, `\2` ne devraient l'être qu'à l'intérieur.

3.6 Répétitions et quantificateurs

Les meta-caractères ou quantificateurs `?`, `*` et `{}` nous permettent de fixer le nombre de répétitions d'une portion de regex. Un quantificateur est placé juste après le caractère, la classe de caractères ou le regroupement à répéter. Ils ont le sens suivant :

- `a?` : reconnaît 'a' zéro ou une fois.
- `a*` : reconnaît 'a' zéro fois ou plus.
- `a+` : reconnaît 'a' au moins une fois.
- `a{n,m}` : reconnaît 'a' au moins `n` fois, mais pas plus de `m` fois.
- `a{n,}` : reconnaît 'a' au moins `n` fois.
- `a{n}` : reconnaît 'a' exactement `n` fois.

Voici quelques exemples :

```
[a-z]+\s+\d*/; # reconnaît un mot en minuscules suivi d'au moins un blanc
                # et éventuellement d'un certain nombre de chiffres
/(\w+)\s+\d{1}/; # reconnaît la répétition d'un mot de longueur quelconque
$annee =~ /\d{2,4}/; # s'assure que l'année contient au moins 2 chiffres et
                    # pas plus de 4 chiffres
$annee =~ /\d{4}|\d{2}/; # meilleure reconnaissance; exclut le cas de 3 chiffres
```

Ces quantificateurs essayent d'entrer en correspondance avec la chaîne la plus longue possible tout en permettant à la regex d'être reconnue (ils sont gourmands). Donc :

```
$x = 'Ce chien est le mien';
$x =~ /^(.*) (ien) (.*)$/; # correspondance,
                           # $1 = 'Ce chien est le m'
                           # $2 = 'ien'
                           # $3 = '' (aucun caractère)
```

Le premier quantificateur `.*` consomme la chaîne la plus longue possible tout en laissant une possibilité de correspondance pour la regex globale. Le second quantificateur `.*` n'a plus de caractère disponible donc il est reconnu zéro fois.

3.7 Plus de correspondances

Il y a encore quelques détails que vous devez connaître à propos des opérateurs de correspondance. Dans le code

```
$motif = 'Seuss';
while (<>) {
    print if /$motif/;
}
```

perl doit réévaluer `$motif` à chaque passage dans la boucle. Si `$motif` ne change jamais, utilisez le modificateur `//o` pour n'effectuer qu'une seule fois l'interpolation. Si vous ne voulez aucune interpolation, utilisez les délimiteurs spéciaux `m''` :

```
@motif = ('Seuss');
m/@motif/; # reconnaît 'Seuss'
m'@motif'; # reconnaît la chaîne littérale '@motif'
```

Le modificateur `//g` demande à l'opérateur de mise en correspondance de s'appliquer à une chaîne autant de fois que possible. Dans un contexte scalaire, une recherche de correspondance assortie du modificateur `//g` sautera de reconnaissance en reconnaissance en se souvenant à chaque fois de l'endroit où elle s'est arrêtée la fois précédente. Vous pouvez récupérer la position atteinte via la fonction `pos()`. Par exemple :

```
$x = "chien chat maison"; # 3 mots
while ($x =~ /(\w+)/g) {
    print "le mot $1 se termine en ", pos $x, "\n";
}
```

affiche

```
le mot chien se termine en 5
le mot chat se termine en 10
le mot maison se termine en 17
```

L'échec de la reconnaissance ou la modification de la chaîne réinitialise la position. Si vous ne voulez pas de réinitialisation en cas d'échec, ajoutez le modificateur `//c` comme dans `/regex/gc`.

Dans un contexte de liste, `//g` retournera la liste complète des groupes reconnus ou, si il n'y a pas de regroupement, la liste des sous-chaînes reconnues par la regex complète. Donc :

```
@mots = ($x =~ /(\w+)/g); # correspondance,
                        # $mots[0] = 'chien'
                        # $mots[1] = 'chat'
                        # $mots[2] = 'maison'
```

3.8 Recherche et remplacement

On effectue une recherche et remplacement en utilisant `s/regex/remplacement/modificateurs`. `remplacement` est comme une chaîne Perl entre guillemets qui remplacera dans la chaîne la partie reconnue par la `regex`. Là aussi, l'opérateur `~` permet de choisir à quelle chaîne sera appliquée `s///`. Si `s///` doit s'appliquer à `$_`, il est possible d'omettre `$_=`. S'il y a une correspondance, `s///` retourne le nombre de remplacements effectués sinon il retourne faux. Voici quelques exemples :

```
$x = "Time to feed the cat!";
$x =~ s/cat/hacker/; # $x contient "Time to feed the hacker!"
$y = "'quoted words'";
$y =~ s/^(.*)'$/\1/; # supprime les apostrophes,
                    # $y contient "quoted words"
```

Lorsqu'on utilise l'opérateur `s///`, les variables `$1`, `$2`, etc. sont directement utilisables dans l'expression de remplacement. Avec le modificateur `s///g`, la recherche et remplacement auront lieu sur toutes les occurrences de l'expression rationnelle :

```
$x = "I batted 4 for 4";
$x =~ s/4/four/; # $x contient "I batted four for 4"
$x = "I batted 4 for 4";
$x =~ s/4/four/g; # $x contient "I batted four for four"
```

Le modificateur d'évaluation `s///e` ajoute un `eval{...}` autour de la chaîne de remplacement et c'est le résultat de cette évaluation qui sera substitué à la sous-chaîne reconnue. Quelques exemples :

```
# inverser tous les mots d'une chaîne
$x = "the cat in the hat";
$x =~ s/(\w+)/reverse $1/ge; # $x contient "eht tac ni eht tah"

# convertir un pourcentage en fraction
$x = "A 39% hit rate";
$x =~ s!(\d+)%!$1/100!e; # $x contient "A 0.39 hit rate"
```

Le dernier exemple montre que `s///` peut utiliser d'autres délimiteurs tels que `s!!!` ou `s{ }{ }...` ou même `s{ }//`. Si les délimiteurs sont des apostrophes `s''` alors l'expression rationnelle et la chaîne de remplacement sont considérées comme des chaînes entre apostrophes (pas d'interpolation des variables).

3.9 L'opérateur de découpage : split

`split /regex/, chaine` découpe `chaine` en une liste de sous-chaînes et retourne cette liste. La `regex` détermine la séquence de caractères à utiliser comme séparateur lors du découpage de `string`. Par exemple, pour découper une chaîne en mots, utilisez :

```
$x = "Calvin and Hobbes";
@mots = split /\s+/, $x; # $mots[0] = 'Calvin'
                        # $mots[1] = 'and'
                        # $mots[2] = 'Hobbes'
```

Pour extraire une liste de nombres séparés par des points-virgules :

```
$x = "1,618;2,718; 3,142";
@const = split /;\s*/, $x; # $const[0] = '1,618'
                        # $const[1] = '2,718'
                        # $const[2] = '3,142'
```

Si vous utilisez l'expression rationnelle vide `//`, la chaîne est découpée en ses caractères. Si l'expression rationnelle contient des regroupements alors la liste produite contiendra aussi les groupes reconnus :

```
$x = "/usr/bin";
@parts = split m!(/)!/, $x; # $parts[0] = ''
                          # $parts[1] = '/'
                          # $parts[2] = 'usr'
                          # $parts[3] = '/'
                          # $parts[4] = 'bin'
```

Puisque le premier caractère de `$x` est reconnu comme délimiteur, `split` ajoute un élément vide au début de la liste.

4 BUGS

Aucun.

5 VOIR AUSSI

C'est un simple guide d'introduction. Pour un tutoriel plus complet voir le manuel *perlretut* et pour une référence complète voir le manuel *perle*.

6 AUTEUR ET COPYRIGHT

Copyright (c) 2000 Mark Kvale All rights reserved.

This document may be distributed under the same terms as Perl itself.

Tous droits réservés.

Ce document peut être distribuer sous les mêmes termes que Perl.

6.1 Remerciements

L'auteur tient à remercier Mark-Jason Dominus, Tom Christiansen, Ilya Zakharevich, Brad Hughes et Mike Giroux pour leurs commentaires précieux.

7 TRADUCTION

7.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.8.8. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

7.2 Traducteur

Paul Gaborit (Paul.Gaborit at enstimac.fr).

7.3 Relecture

Aucune pour l'instant.

8 À propos de ce document

Ce document est la traduction française du document original distribué avec perl. Vous pouvez retrouver l'ensemble de la documentation française Perl (éventuellement mise à jour) en consultant l'URL <<http://perl.enstimac.fr/>>.

Ce document PDF a été produit Paul Gaborit. Si vous utilisez la version PDF de cette documentation (ou une version papier issue de la version PDF) pour tout autre usage qu'un usage personnel, je vous serai reconnaissant de m'en informer par un petit message <<mailto:Paul.Gaborit@enstimac.fr>>.

Si vous avez des remarques concernant ce document, en premier lieu, contactez la traducteur (vous devriez trouver son adresse électronique dans la rubrique TRADUCTION) et expliquez-lui gentiment vos remarques ou critiques. Il devrait normalement vous répondre et prendre en compte votre avis. En l'absence de réponse, vous pouvez éventuellement me contacter.

Vous pouvez aussi participer à l'effort de traduction de la documentation Perl. Toutes les bonnes volontés sont les bienvenues. Vous devriez trouver tous les renseignements nécessaires en consultant l'URL ci-dessus.

Ce document PDF est distribué selon les termes de la license Artistique de Perl. Toute autre distribution de ce fichier ou de ses dérivés impose qu'un arrangement soit fait avec le(s) propriétaire(s) des droits. Ces droits appartiennent aux auteurs du document original (lorsqu'ils sont identifiés dans la rubrique AUTEUR), aux traducteurs et relecteurs pour la version française et à moi-même pour la version PDF.