perlfaq6

Table des matières

1 NAME/NOM

perlfaq6 - Expressions rationnelles

2 DESCRIPTION

Cette partie de la FAQ est incroyablement courte car les autres parties sont parsemées de réponses concernant les expressions rationnelles. Par exemple, décoder une URL ou vérifier si quelque chose est un nombre ou non relève du domaine des expressions rationnelles, mais ces réponses se trouvent ailleurs que dans ce document (dans *perlfaq9* « Comment décoder ou créer ces %-encodings sur le web ? » et dans *perlfaq4* « Comment déterminer si un scalaire est un nombre/entier/à virgule flottante ? »).

2.1 Comment utiliser les expressions rationnelles sans créer du code illisible et difficile à maintenir ?

Trois méthodes peuvent rendre les expressions rationnelles faciles à maintenir et compréhensibles.

Commentaires en dehors de l'expression rationnelle

Décrivez ce que vous faites et comment vous le faites en utilisant la façon habituelle de commenter en Perl.

```
# transforme la ligne en le premier mot, le signe deux points
# et le nombre de caractères du reste de la ligne
s/^(\w+)(.*)/ lc($1) . ":" . length($2) /meg;
```

Commentaires dans l'expression rationnelle

En utilisant le modificateur `/x`, les espaces seront ignorés dans le motif de l'expression rationnelle (sauf dans une classe de caractères) et vous pourrez ainsi utiliser les commentaires habituels dedans. Comme vous pouvez l'imaginer, espaces et commentaires aident pas mal.

Un `/x` vous permet de transformer ceci :

```
s{<(?:[^>'"]*|".*?"|'.*?')+>}{}gs;
```

en :

```
s{ <                # signe inférieur
    (?:             # parenthèse ouvrante ne créant pas de référence
         [^>'"] *   # 0 ou plus de caract. qui ne sont ni >, ni ' ni "
            |       #    ou
         ".*?"      # une partie entre guillemets (reconnaissance min)
            |       #    ou
         '.*?'      # une partie entre apostrophes (reconnaissance min)
    ) +             #   tout cela se produisant une ou plusieurs fois
   >                # signe supérieur
}{}gsx;             # remplacé par rien, c.-à-d. supprimé
```

Ce n'est pas encore aussi clair que de la prose, mais c'est très utile pour décrire le sens de chaque partie du motif.

Différents délimiteurs

Bien qu'habituellement délimités par le caractère `/`, les motifs peuvent en fait être délimités par quasiment n'importe quel caractère. *perlre* l'explique. Par exemple, `s///` ci-dessus utilise des accolades comme délimiteurs. Sélectionner un autre délimiteur permet d'éviter de le quoter à l'intérieur du motif :

```
s/\/usr\/local/\/usr\/share/g;        # mauvais choix de délimiteur
s#/usr/local#/usr/share#g;            # meilleur
```

2.2 J'ai des problèmes pour faire une reconnaissance sur plusieurs lignes. Qu'est-ce qui ne va pas ?

Ou vous n'avez en fait pas plus d'une ligne dans la chaîne où vous faites la recherche (cas le plus probable), ou vous n'appliquez pas le bon modificateur à votre motif (cas possible).

Il y a plusieurs manières d'avoir plusieurs lignes dans une chaîne. Si vous voulez l'avoir automatiquement en lisant l'entrée, vous initialiserez $/ (probablement à "" pour des paragraphes ou `undef` pour le fichier entier) ce qui vous permettra de lire plus d'une ligne à la fois.

Lire *perlre* vous aidera à décider lequel des modificateurs `/s` ou `/m` (ou les deux) vous utiliserez : `/s` autorise le point à reconnaître le caractère fin de ligne alors que `/m` permet à l'accent circonflexe et au dollar de reconnaître tous les débuts et fins de ligne, pas seulement le début et la fin de la chaîne. Vous pouvez utiliser cela pour être sûr que vous avez bien plusieurs lignes dans votre chaine.

Par exemple, ce programme détecte les mots répétés, même dans des lignes différentes (mais pas dans plusieurs paragraphes). Pour cet exemple, nous n'avons pas besoin de `/s` car nous n'utilisons pas le point dans l'expression rationnelle qui doit enjamber les fins de ligne. Nous n'avons pas besoin non plus de `/m` car nous ne voulons pas que le circonflexe ou le dollar fasse une reconnaissance d'un début ou d'une fin d'une nouvelle ligne. Mais il est impératif que $/ ait une autre valeur que la valeur par défaut, ou alors nous n'aurons pas plusieurs lignes d'un coup à se mettre sous la dent.

```
$/ = '';              # lis au moins un paragraphe entier,
                      # pas qu'une seule ligne
while ( <> ) {
    while ( /\b([\w'-]+)(\s+\1)+\b/gi ) { # les mots commencent par
                                          # des caractères alphanumériques
       print "$1 est répété dans le paragraphe $.\n";
    }
}
```

Voilà le code qui trouve les phrases commençant avec "From " (qui devrait être transformés par la plupart des logiciels de courrier électronique) :

```
$/ = '';     # lis au moins un paragraphe entier, pas qu'une seule ligne
while ( <> ) {
    while ( /^From /gm ) { # /m fait que ^ reconnaisse
                           # tous les débuts de ligne
        print "from de départ dans le paragraphe $.\n";
    }
}
```

Voilà le code qui trouve tout ce qu'il y a entre START et END dans un paragraphe :

```
undef $/; # lis le fichier entier, pas seulement qu'une ligne
          # ou un paragraphe
while ( <> ) {
    while ( /START(.*?)END/sgm ) { # /s fait que . enjambe les lignes
        print "$1\n";
    }
}
```

2.3 Comment extraire des lignes entre deux motifs qui sont chacun sur des lignes différentes ?

Vous pouvez utiliser l'opérateur quelque peu exotique `..` de Perl (documenté dans *perlop*):

```
perl -ne 'print if /START/ .. /END/' fichier1 fichier2 ...
```

Si vous voulez du texte et non des lignes, vous pouvez utiliser

```
perl -0777 -ne 'print "$1\n" while /START(.*?)END/gs' fichier1 fichier2 ...
```

Mais si vous voulez des occurrences imbriquées de `START` à l'intérieur de `END`, vous tombez sur le problème décrit, dans cette section, sur la reconnaissance de texte bien équilibré.

Ici un autre exemple d'utilisation de `..` :

```
while (<>) {
    $in_header =   1  .. /^$/;
    $in_body   = /^$/ .. eof();
    # maintenant choisissez entre eux
} continue {
    reset if eof();               # fixe $.
}
```

2.4 J'ai mis une expression rationnelle dans $/ mais cela ne marche pas. Qu'est-ce qui est faux ?

Jusqu'à Perl 5.8 inclus, $/ doit être une chaîne. Cela pourrait changer en 5.10 mais n'y compter pas trop tout de même. En attendant, vous pouvez vous baser sur les exemples ci-dessous si vous en avez réellement besoin.

Si vous disposez de File::Stream, c'est très facile.

```
use File::Stream;
my $stream = File::Stream->new(
  $filehandle,
  separator => qr/\s*,\s*/,
 );

print "$_\n" while <$stream>;
```

Si vous ne disposez pas de File::Stream, il vous faudra en faire un peu plus.

Vous pouvez utiliser la forme à quatre argument de sysread pour remplir peu à peu un tampon. Après chaque ajout au tampon, vous vérifiez si vous avez une ligne entière (en utilisant votre expression rationnelle).

```
local $_ = "";
while( sysread FH, $_, 8192, length ) {
  while( s/^((?s).*?)your_pattern/ ) {
    my $record = $1;
    # votre traitement ici.
  }
}
```

Vous pouvez faire la même chose avec un foreach testant une expression rationnelle utilisant le modificateur c et l'ancre \G si le risque d'avoir tout votre fichier en mémoire à la fin ne vous dérange pas.

```
local $_ = "";
while( sysread FH, $_, 8192, length ) {
  foreach my $record ( m/\G((?s).*?)your_pattern/gc ) {
    # votre traitement ici.
  }
  substr( $_, 0, pos ) = "" if pos;
}
```

2.5 Comment faire une substitution indépendante de la casse de la partie gauche mais préservant cette casse dans la partie droite ?

Voici une adorable solution perlienne par Larry Rosler. Elle exploite les propriétés du xor bit à bit sur les chaînes ASCII.

```
 $_= "this is a TEsT case";

 $old = 'test';
 $new = 'success';

 s{(\Q$old\E)}
  { uc $new | (uc $1 ^ $1) .
     (uc(substr $1, -1) ^ substr $1, -1) x
         (length($new) - length $1)
  }egi;

 print;
```

Et la voici sous la forme d'un sous-programme, selon le modèle ci-dessus :

```
sub preserve_case($$) {
  my ($old, $new) = @_;
  my $mask = uc $old ^ $old;

  uc $new | $mask .
    substr($mask, -1) x (length($new) - length($old))
}

$a = "this is a TEsT case";
$a =~ s/(test)/preserve_case($1, "success")/egi;
print "$a\n";
```

Ceci affiche :

```
this is a SUcCESS case
```

Jeff Pinyan propose une autre manière de faire qui préserve la casse du mot de remplacement s'il s'avère plus long que l'original :

```
sub preserve_case {
  my ($from, $to) = @_;
  my ($lf, $lt) = map length, @_;

  if ($lt < $lf) { $from = substr $from, 0, $lt }
  else { $from .= substr $to, $lf }

  return uc $to | ($from ^ uc $from);
}
```

Cela transformera la phrase en "this is a SUcCess case."
Rien que pour montrer que les programmeurs C peuvent écrire du C dans tous les langages de programmation, si vous préférez une solution plus proche du style de C, le script suivant fait en sorte que la substitution a la même casse, lettre par lettre, que l'original (il s'avère aussi tourner environ 240 % plus lentement que la version perlienne). Si la substitution a plus de caractères que la chaîne substituée, la casse du dernier caractère est utilisée pour le reste de la substitution.

```
# L'original est de Nathan Torkington, mis en forme par Jeffrey Friedl
#
sub preserve_case($$)
{
    my ($old, $new) = @_;
    my ($state) = 0; # 0 = no change; 1 = lc; 2 = uc
    my ($i, $oldlen, $newlen, $c) = (0, length($old), length($new));
    my ($len) = $oldlen < $newlen ? $oldlen : $newlen;

    for ($i = 0; $i < $len; $i++) {
        if ($c = substr($old, $i, 1), $c =~ /[\W\d_]/) {
            $state = 0;
        } elsif (lc $c eq $c) {
            substr($new, $i, 1) = lc(substr($new, $i, 1));
            $state = 1;
        } else {
            substr($new, $i, 1) = uc(substr($new, $i, 1));
            $state = 2;
        }
    }
    # on se retrouve avec ce qui reste de new
    # (quand new est plus grand que old)
    if ($newlen > $oldlen) {
        if ($state == 1) {
            substr($new, $oldlen) = lc(substr($new, $oldlen));
        } elsif ($state == 2) {
            substr($new, $oldlen) = uc(substr($new, $oldlen));
        }
    }
    return $new;
}
```

2.6 Comment faire pour que \w reconnaisse les caractères nationaux ?

Placez `use locale;` dans votre script. La classe de caractères \w est définie à partir du locale courant.

Voir *perllocale* pour les détails.

2.7 Comment reconnaître une version locale-équivalente de /[a-zA-Z]/ ?

Vous pouvez utiliser la classe de caractères POSIX `/[[:alpha:]]/` qui est documentée dans *perlre*.

Quel que soit le locale courant, les caractères alphabétiques sont tous les caractères reconnus par \w sauf les chiffres et le caractère de soulignement. Exprimé sous la forme d'une expression rationnelle ça ressemble à `/[^\W\d_]/`. Son complémentaire, les caractères non-alphabétiques, est tout \W plus les chiffres et le caractère de soulignement, donc `/[\W\d_]/`.

2.8 Comment protéger une variable pour l'utiliser dans une expression rationnelle ?

L'analyseur syntaxique de Perl remplacera par leur valeur les occurences de $variable et @variable dans les expressions rationnelles à moins que le délimiteur ne soit une apostrophe. Rappelez-vous aussi que la partie droite d'une `s///` substitution est considérée comme une chaîne entre guillemets (voir *perlop* pour plus de détails). Rappelez-vous encore que n'importe quel caractère spécial d'expression rationnelle agira à moins que vous ne précédiez la substitution avec \Q. Voici un exemple :

```
$chaine = "Placido P. Octopus";
$regex  = "P.";

$chaine =~ s/$regex/Polyp/;
# $chaine est maintenant "Polypacido P. Octopus"
```

Dans les expressions rationnelles, le `.` est un caractère spécial qui reconnait n'importe quel caractère, et donc l'expression rationnelle `P.` reconnait le `Pl` du début la chaîne originale.

Pour échapper à la signification spéciale du `.`, nous utilisons `\Q` :

```
$chaine = "Placido P. Octopus";
$regex  = "P.";

$chaine =~ s/\Q$regex/Polyp/;
# $chaine est maintenant "Placido Polyp Octopus"
```

L'utilisation de `\Q` fait que le `.` de l'expression rationnelle est traité comme un caractère normal, et donc `P.` reconnait maintenant un `P` suivi d'un point.

2.9 À quoi sert vraiment /o ?

L'utilisation d'une variable dans une opération de reconnaissance d'expression rationnelle force une réévaluation (et peut-être une recompilation) de l'expression rationnelle à chaque fois qu'elle est appliquée. Le modificateur `/o` verrouille l'expression rationnelle la première fois qu'elle est utilisée. C'est toujours ce qui se passe avec une expression rationnelle constante, et en fait, le motif est compilé dans le format interne en même temps que le programme entier l'est.

Utiliser `/o` est peu pertinent à moins que le remplacement de variable ne soit utilisé dans le motif, et si cela est, le moteur d'expression rationnelle ne prendra pas en compte les modifications de la variable ultérieures à la *toute première* évaluation.

`/o` est souvent utilisé pour gagner en efficacité en ne faisant pas les évaluations nécessaires quand vous savez que cela ne pose pas de problème (car vous savez que les variables ne changeront pas), ou plus rarement, quand vous ne voulez pas que l'expression rationnelle remarque qu'elles changent.

Par exemple, voici un programme "paragrep" :

```
$/ = '';  # mode paragraphe
$pat = shift;
while (<>) {
    print if /$pat/o;
}
```

2.10 Comment utiliser une expression rationnelle pour enlever les commentaires de type C d'un fichier ?

Bien que cela puisse se faire, c'est plus compliqué que vous ne pensez. Par exemple, cette simple ligne

```
perl -0777 -pe 's{/\*.*?\*/}{}gs' foo.c
```

marchera dans beaucoup de cas mais pas tous. Vous voyez, c'est un peu simplet pour certains types de programmes C, en particulier, ceux où des chaines protégées sont des commentaires. Pour cela, vous avez besoin de quelque chose de ce genre, créé par Jeffrey Friedl, modifié ultérieurement par Fred Curtis :

```
$/ = undef;
$_ = <>;
s#/\*[^*]*\*+([^/*][^*]*\*+)*/|("(\\.|[^"\\])*"|'(\\.|[^'\\])*'|.[^/"'\\]*)#defined $2 ? $2 : ""#gse;
print;
```

Cela pourrait, évidemment, être écrit plus lisiblement avec le modificateur `/x` en ajoutant des espaces et des commentaires. Le voici étendu, une gentillesse de Fred Curtis.

```
 s{
    /\*         ##  Début d'un commentaire /* ... */
    [^*]*\*+    ##  Non-* suivie par 1 ou plusieurs *
    (
      [^/*][^*]*\*+
    )*          ##  0 ou plusieurs choses ne commençant pas par /
                ##  mais finissent par '*'
    /           ##  Fin d'un commentaire /* ... */

  |         ##     OU  diverses choses qui ne sont pas des commentaires

    (
      "           ##  Début d'une chaîne " ... "
      (
        \\.           ##  Caractère échappé
      |               ##    OU
        [^"\\]        ##  Non "\
      )*
      "           ##  Fin d'une chaîne " ... "

    |         ##     OU

      '           ##  Début d'une chaîne ' ... '
      (
        \\.           ##  Caractère échappé
      |               ##    OU
        [^'\\]        ##  Non '\
      )*
      '           ##  Fin d'une chaîne ' ... '

    |         ##     OU

      .           ##  Tout autre caractère
      [^/"'\\]*   ##  Caractères ne débutant pas un commentaire,
                  ##  une chaîne ou un échappement
    )
  }{defined $2 ? $2 : ""}gxse;
```

Une légère modification retire aussi les commentaires C++ :

```
s#/\*[^*]*\*+([^/*][^*]*\*+)*/|//[^\n]*|("(\\.|[^"\\])*"|'(\\.|[^'\\])*'|.[^/"'\\]*)#defined $2 ? $2 :
```

2.11 Est-ce possible d'utiliser les expressions rationnelles de Perl pour reconnaître du texte bien équilibré ?

Auparavant, les expressions rationnelles de Perl ne pouvaient pas reconnaître du texte bien équilibré. Dans les versions récentes de perl depuis la 5.6.1, des fonctionnalités expérimentales ont été ajoutées afin de rendre possible ces reconnaissances. La documentation de la construction (??{ }) dans *perlre* montre des exemples de reconnaissances de parenthèses bien équilibrées. Assurez-vous de prendre en compte les avertissements présents dans cette documentation avant d'utiliser ces fonctionnalités.

Le CPAN propose de nombreux modules utiles pour reconnaitre des textes dépendant de leur contexte. Damian Conway fournit quelques expressions pratiques dans Regexp::Common. Le module Text::Balanced fournit une solution générale à ce problème.

XML et HTML font parties des usages courants de la reconnaissance de textes équilibrés. Il existe plusieurs modules qui répondent à ce besoin. Parmi eux HTML::Parser et XML::Parser mais il y en a bien d'autres.

Une sous-routine élaborée (pour du 7-bit ASCII seulement) pour extraire de simples caractères se correspondant et peut-être imbriqués, comme `` ` `` et `'`, `{` et `}`, ou `(` et `)` peut être trouvée sur http://www.cpan.org/authors/id/TOMC/scripts/pull_quotes.gz.

Le module C::Scan du CPAN contient de tels sous-programmes pour son usage interne, mais elles ne sont pas documentées.

2.12 Que veut dire "les expressions rationnelles sont gourmandes" ? Comment puis-je le contourner ?

La plupart des gens pensent que les expressions rationnelles gourmandes reconnaissent autant qu'elles peuvent. Techniquement parlant, c'est en réalité les quantificateurs (`?`, `*`, `+`, `{}`) qui sont gourmands plutôt que le motif entier ; Perl préfère les gourmandises locales et les gratifications immédiates à une gloutonnerie globale. Pour avoir les versions non gourmandes des mêmes quantificateurs, utilisez (`??`, `*?`, `+?`, `{}?`).

Un exemple:

```
$s1 = $s2 = "J'ai très très froid";
$s1 =~ s/tr.*s //;      # J'ai froid
$s2 =~ s/tr.*?s //;     # J'ai très froid
```

Notez que la seconde substitution arrête la reconnaissance dès qu'un "s " est rencontré. Le quantificateur `*?` dit effectivement au moteur des expressions rationnelles de trouver une reconnaissance aussi vite que possible et de passer le contrôle à la suite, comme de se refiler une patate chaude.

2.13 Comment examiner chaque mot dans chaque ligne ?

Utilisez la fonction split :

```
while (<>) {
    foreach $word ( split ) {
        # faire quelque chose avec $word ici
    }
}
```

Notez que ce ne sont pas vraiment des mots dans le sens mots français ; ce sont juste des suites de caractères différents des caractères d'espacement.

Pour travailler seulement avec des séquences alphanumériques (caractère de soulignement inclu), vous pourriez envisager

```
while (<>) {
    foreach $word (m/(\w+)/g) {
        # faire quelque chose avec $word ici
    }
}
```

2.14 Comment afficher un rapport sur les fréquences de mots ou de lignes ?

Pour faire cela, vous avez à sortir chaque mot du flux d'entrée. Nous appelerons mot une suite de caractères alphabétiques, de tirets et d'apostrophes plutôt qu'une suite de tout caractère sauf espace comme vue dans la question précédente :

```
while (<>) {
    while ( /(\b[^\W_\d][\w'-]+\b)/g ) {   # on rate "`mouton'"
        $seen{$1}++;
    }
}
while ( ($word, $count) = each %seen ) {
    print "$count $word\n";
}
```

Si vous voulez faire la même chose avec les lignes, vous n'avez pas besoin d'une expression rationnelle :

```
while (<>) {
    $seen{$_}++;
}
while ( ($line, $count) = each %seen ) {
    print "$count $line";
}
```

Si vous voulez que le résultat soit trié, regardez dans *perlfaq4* : « Comment trier une table de hachage (par valeur ou par clef) ? ».

2.15 Comment faire une reconnaissance approximative ?

Regardez le module String::Approx disponible sur CPAN.

2.16 Comment reconnaître efficacement plusieurs expressions rationnelles en même temps ?

(contribution de brian d foy)
Ne demandez pas à Perl de recompiler une expression rationnelle à chaque utilisation. Dans l'exemple suivant, perl doit recompiler l'expression rationnelle à chaque passage dans la boucle foreach() puisqu'il n'a aucun moyen de savoir ce que contient $motif.

```
@motifs = qw( foo bar baz );

LINE: while( <> )
  {
    foreach $motif ( @motifs )
      {
        print if /\b$motif\b/i;
        next LINE;
      }
  }
```

L'opérateur qr// est apparu dans la version 5.005 de perl. Il compile une expression rationnelle sans l'appliquer. Lorsque vous utilisez la version précompilée de l'expression rationnelle, perl a moins de travail à faire. Dans l'exemple suivant, on introduit un map() pour transformer chaque motif en sa version précompilé. Le reste du script est identique mais plus rapide.

```
@motifs = map { qr/\b$_\b/i } qw( foo bar baz );

LINE: while( <> )
  {
    foreach $motif ( @motifs )
      {
        print if /$motif/;
        next LINE;
      }
  }
```

Dans certains cas, vous pouvez même reconnaître plusieurs motifs à l'intérieur d'une même expression rationnelle. Faites tout de même attention aux situations amenant à des retours arrière.

```
$regex = join '|', qw( foo bar baz );

LINE: while( <> )
  {
     print if /\b(?:$regex)\b/i;
  }
```

Pour plus de détails concernant l'efficacité des expressions rationnelles, lisez « Mastering Regular Expressions » par Jeffrey Freidl. Il explique le fonctionnement du moteur d'expressions rationnelles et pourquoi certains motifs sont étonnament inefficaces. Une fois que vous aurez bien compris comment perl utilise les expressions rationnelles, vous pourrez les optimiser finement dans toutes les situations.

2.17 Pourquoi les recherches de limite de mot avec \b ne marchent pas pour moi ?

(contribution de brian d foy)

Assurez-vous de bien comprendre ce que \b représente : c'est la frontière entre un caractère de mot, \w, et quelque chose qui n'est pas un caractère de mot. Ce quelque chose qui n'est pas un caractère de mot peut être un \W mais aussi le début ou la fin de la chaîne.

Ce n'est pas la frontière entre une caractère d'espacement et un caractère autre et ce n'est pas ce qui sépare les mots d'une phrase.

En terme d'expression rationnelle, une frontière de mot (\b) est une « assertion de longueur nulle », ce qui signifie qu'elle ne représente pas un caractère de la chaîne mais une condition à une position donnée.

Dans l'expression rationnelle /\bPerl\b/, il doit y avoir une frontière de mot avant le "P" et après le "l". À partir du moment où autre chose qu'un caractère de mot précède le "P" et suit le "l", le motif est reconnu. Les chaînes suivantes sont reconnues par /\bPerl\b/.

```
"Perl"    # pas de caractère de mot avant "P" ou après "l"
"Perl "   # idem (l'espace n'est pas un caractère de mot)
"'Perl'"  # l'apostrophe n'est pas un caractère de mot
"Perl's"  # pas de caractère de mot avant "P",
          # un caractère non-mot après "l"
```

Les chaînes suivantes ne sons pas reconnues par /\bPerl\b/.

```
"Perl_"   # _ est un caractère de mot !
"Perler"  # pas de caractère de mot avant "P", mais bien après "l"
```

L'usage de \b ne se limite pas aux mots. Vous pouvez chercher des caractères non-mot entourés de caractères de mot. Les chaînes suivantes sont reconnues par le motif /\b'\b/.

```
"don't"   # l'apostrophe est entourée par "n" et "t"
"qep'a'"  # l'apostrophe esr entourée par "p" et "a"
```

La chaîne suivante n'est pas reconnue par /\b'\b/.

```
"foo'"    # aucun caractère de mot après l'apostrophe
```

Vous pouvez aussi utiliser le complémentaire de \b, \B, pour spécifier qu'il ne doit pas y avoir de frontière de mot.

Dans le motif /\Bam\B/, il doit y avoir un caractère de mot avant le "a" et après le "m". Ces chaînes sont reconnues par /\Bam\B/.

```
"llama"   # "am" est entouré par des caractères de mot
"Samuel"  # idem
```

Les chaînes suivantes ne sont pas reconnues par /\Bam\B/.

```
"Sam"      # pas de frontière de mot avant "a", mais après "m"
"I am Sam" # "am" entourés par des caractères non-mot
```

2.18 Pourquoi l'utilisation de $&, $‘, or $’ ralentit tant mon programme ?

(contribution de Anno Siegel)

Une fois que Perl sait que vous avez besoin d'une de ces variables quelque part, il les calcule pour chaque reconnaissance de motif. Cela signifie qu'à chaque reconnaissance de motif, la chaîne entière est recopiée, une partie dans $‘, une autre dans $& et le reste dans $’. Le ralentissement est d'autant plus perceptible que les chaînes sont longues et que les motifs sont reconnus. Donc, évitez $&, $’ et $‘ si vous le pouvez. Et si vous ne le pouvez pas, une fois que vous les avez utilisées, utilisez-les tant que vous voulez, car vous en avez déjà payé le prix. Souvenez-vous que certains algorithmes les apprécient vraiment. À partir de la version 5.005, la variable $& n'a plus le même coût que les deux autres.

Depuis Perl 5.6.1, les variables spéciales @- et @+ peuvent remplacer fonctionnellement $‘, $& et $’. Ces tableaux contiennent des pointeurs vers le début et la fin de chaque partie reconnue (voir *perlvar* pour tous les détails). Ils vous donnent donc bien accès aux mêmes informations mais sans l'inconvénient des recopies de chaînes inutiles.

2.19 À quoi peut bien servir \G dans une expression rationnelle ?

On peut utiliser l'ancre \G pour débuter une reconnaissance dans une chaîne là où la recherche précédente s'est arrêtée. Le moteur d'expression rationnelle ne peut pas sauter de caractère pour trouver la prochaine reconnaissance avec cette ancre. Donc <\G> est similaire à l'ancre de début de chaîne, ^. L'ancre \G est habiutellement utilisée en conjonction avec le modificateur /g. Elle utilise la valeur de pos() comme position de départ de la prochaine reconnaissance. Au fur et à mesure que l'opérateur de reconnaissance fait des reconnaissances, il met à jour pos() avec la position du caractère suivant la dernière reconnaissance (ou le premier caractère de la prochaine reconnaissance selon le point de vue qu'on adopte). Chaque chaîne a sa propre valeur pos().

Supposons que vous vouliez reconnaîne toutes les paires consécutives de chiffres dans une chaîne telle que "1122a44" et vous arrêtes dès que vous rencontrez autre chose qu'un chiffre. Vous voulez donc reconnaître 11 et 22 mais la lettre a apparaissant entre 22 et 44 doit vous stopper là. La simple reconnaissance de paires de chiffres passerait au-delà du a et reconnaîtrait 44.

```
$_ =  "1122a44";
my @paires = m/(\d\d)/g;   # qw( 11 22 44 )
```

Si vous utilisez l'ancre \G, vous forcez la reconnaissance suivant 22 à débuter avec a. L'expression rationnelle ne peut donc être reconnue puisqu'elle ne trouve pas un chiffre et donc la prochaine reconnaissance échoue et l'opérateur de reconnaissance retourne la liste des paires déjà trouvées.

```
$_ =  "1122a44";
my @paires = m/\G(\d\d)/g;   # qw( 11 22 )
```

Vous pouvez aussi utiliser l'ancre \G dans un contexte scalaire. Il faut alors utiliser le modificateur /g.

```
$_ = "1122a44";
while( m/\G(\d\d)/g )
  {
    print "Vu $1\n";
  }
```

Après un échec de la reconnaissance sur la lettre a, perl réinitialise pos() et la prochaine recherche dans la même chaîne débutera à nouveau au début.

```
$_ = "1122a44";
while( m/\G(\d\d)/g )
  {
    print "Vu $1\n";
  }

print "Vu $1 après while" if m/(\d\d)/g; # trouve "11"
```

Vous pouvez désactiver la réinitialisation de pos() lors de l'échec d'une reconnaissance en utilisant le modificateur /c. La prochaines reconnaissances débuteront toutes là où s'est terminée la dernière reconnaissance (la valeur de pos()) même si une reconnaissance sur la même chaîne à échouer entre temps. Dans notre cas, la reconnaissance après la boucle while() commencera au a (là où s'est terminée la dernière reconnaissance) et puisqu'elle n'utiliser aucune ancre, elle pourra sauter le a et trouver "44".

```
$_ = "1122a44";
while( m/\G(\d\d)/gc )
  {
    print "Vu $1\n";
  }

print "Vu $1 après while" if m/(\d\d)/g; # trouve "44"
```

Typiquement, on utiliser l'ancre `\G` avec le modificateur `/c` lorsqu'on souhaite pouvoir essayer un autre motif si celui-ci échoue, comme dans analyseur syntaxique. Jeffrey Friedl nous offre cet exemple qui fonctionne à partir de la version 5.004.

```
 while (<>) {
   chomp;
   PARSER: {
        m/ \G( \d+\b    )/gcx    && do { print "nombre: $1\n";  redo; };
        m/ \G( \w+      )/gcx    && do { print "mot:   $1\n";  redo; };
        m/ \G( \s+      )/gcx    && do { print "espace:  $1\n";  redo; };
        m/ \G( [^\w\d]+ )/gcx    && do { print "autre:  $1\n";  redo; };
   }
 }
```

Pour chaque ligne, la boucle PARSER essaie de reconnaître une série de chiffres suivies d'une limite de mot. Cette reconnaissance doit débuter là où s'est arrêtée la précédente reconnaissance (où au début de la chaîne la première fois). Puisque `m/ \G( \d+\b )/gcx` utilise le modificateur </c>, même si la chaîne n'est pas reconnue par l'expression rationnelle, perl ne réinitialise pas pos() et essaie le prochain motif en démarrant à la même position.

2.20 Les expressions rationnelles de Perl sont-elles AFD ou AFN ? Sont-elles conformes à POSIX ?

Bien qu'il soit vrai que les expressions rationnelles de Perl ressemblent aux AFD (automate fini déterminé) du programme egrep(1), elles sont dans les faits implémentées comme AFN (automate fini indéterminé) pour permettre le retour en arrière et la mémorisation de sous-motif. Et elles ne sont pas non plus conformes à POSIX, car celui-ci garantit le comportement du pire dans tous les cas. (Il semble que certains préfèrent une garantie de conhérence, même quand ce qui est garanti est la lenteur). Lisez le livre "Mastering Regular Expressions" (from O'Reilly) par Jeffrey Friedl pour tous les détails que vous pouvez espérer connaître sur ces matières (la référence complète est dans *perlfaq2*).

2.21 Qu'est ce qui ne va pas avec l'utilisation de grep dans un contexte vide ?

Le problème c'est que grep construit une liste comme résultat, quel que soit le contexte. Cela signifie que vous amenez Perl à construire une liste que vous allez jeter juste après. Si cette liste est grosse, vous gâchez du temps et de l'espace mémoire. Si votre objectif est de parcourir la liste alors utiliser une boucle foreach pour cela.

Dans les versions de Perl antérieures à la version 5.8.1, map souffrait du même problème. Mais depuis la version 5.8.1, cela a été corrigé et map tient compte maintenant de son contexte - dans un contexte vide, aucune liste n'est construite.

2.22 Comment reconnaître des chaînes avec des caractères multi-octets ?

Depuis la version 5.6, Perl gére les caractères multi-octets avec une qualité qui s'accroît de version en version. Perl 5.8 ou plus est recommandé. Le support des caractères multi-octets inclut Unicode et les anciens encodages à travers le module Encode. Voir *perluniintro*, *perlunicode* et *Encode*.

Si vous utilisez ude vieilles version de Perl, vous pouvez gérer l'Unicode via le module `Unicode::String` et les conversion de caractères en utilisant les modules `Unicode::Map8` andet `Unicode::Map`. Si vous utilisez les encodages japonais, vous pouvez essayer jperl 5.005_03.

Pour finir, Jeffrey Friedl qui a consacré à ce sujet un article dans le numéro 5 de The Perl Journal, vous offre les approches suivantes.

Supposons que vous ayez un codage de ces mystérieux Martiens où les paires de lettre majuscules ASCII codent une lettre simple martienne (i.e. les 2 octets "CV" donne une simple lettre martienne, ainsi que "SG", "VS", "XX", etc.). D'autres octets représentent de simples caractères comme l'ASCII.

Ainsi, la chaîne martienne "Je suis CVSGXX!" utilise 15 octets pour coder les 12 caractères 'J', 'e', ' ', 's', 'u', 'i', 's', ' ', 'CV', 'SG', 'XX', '!'.

Maintenant, vous voulez chercher le simple caractère /GX/. Perl ne connaît rien au martien, donc il trouvera les 2 octets "GX" dans la chaîne "Je suis CVSGXX!", alors que ce caractère n'y est pas: il semble y être car "SG" est à côté de "XX", mais il n'y a pas de réel "GX". C'est un grand problème.

Voici quelques manières, toutes pénibles, de traiter cela :

```
$martian =~ s/([A-Z][A-Z])/ $1 /g; # garantit que les octets "martiens"
                                   # ne sont plus contigüs
print "GX trouvé!\n" if $martian =~ /GX/;
```

Ou ainsi :

```
@chars = $martian =~ m/([A-Z][A-Z]|[^A-Z])/g;
# c'est conceptuellemnt similaire à:     @chars = $text =~ m/(.)/g;
#
foreach $char (@chars) {
    print "GX trouvé!\n", last if $char eq 'GX';
}
```

Ou encore ainsi :

```
while ($martian =~ m/\G([A-Z][A-Z]|.)/gs) {  # \G probablement inutile
    print "GX trouvé!\n", last if $1 eq 'GX';
}
```

Voici une autre solution proposée par Benjamin Goldberg et qui utilise une assertion négative de longueur nulle pour regarder en arrière.

```
print "found GX!\n" if  $martian =~ m/
            (?<![A-Z])
            (?:[A-Z][A-Z])*?
            GX
         /x;
```

L'expression rationnelle est reconnue si le caractère "martien" GX est présent dans la chaîne et échoue sinon.

Si vous ne souhaitez pas utiliser (?<!), une assertion négative arrière, vous pouvez remplacer (?<![A-Z]) par (?:^|[^A-Z]). Cela a pour défaut de placer de mauvaises valeurs dans $-[0] et $+[0] mais, habituellement, on peut faire avec.

2.23 Comment rechercher un motif fourni par l'utilisateur ?

Eh bien, si c'est vraiment un motif, alors utilisez juste

```
chomp($pattern = <STDIN>);
if ($line =~ /$pattern/) { }
```

Ou, puisque vous n'avez aucune garantie que votre utilisateur a entré une expression rationnelle valide, piégez une éventuelle exception de cette façon :

```
if (eval { $line =~ /$pattern/ }) { }
```

Si vous voulez seulement chercher une chaîne et non pas un motif, alors vous devriez soit utiliser la fonction index(), qui est faite pour cela, soit, s'il est impossible de vous convaincre de ne pas utiliser une expression rationnelle pour autre chose qu'un motif, assurez-vous au moins d'utiliser \Q...\E, documenté dans *perlre*.

```
$pattern = <STDIN>;

open (FILE, $input) or die "Couldn't open input $input: $!; aborting";
while (<FILE>) {
    print if /\Q$pattern\E/;
}
close FILE;
```

3 AUTEUR ET COPYRIGHT

4 TRADUCTION

La traduction française est distribuée avec les même droits que sa version originale (voir ci-dessus).

4.1 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/.

4.2 Traducteur

Marianne Roger <maroger@maretmanu.org>. Mise à jour : Roland Trique <roland.trique@uhb.fr>, Paul Gaborit (paul.gaborit at enstimac.fr).

4.3 Relecture

Régis Julié (Regis.Julie@cetelem.fr), Gérard Delafond.

5 À propos de ce document

Ce document est la traduction française du document orginal 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.