perlre

Table des matières

1 NAME/NOM

perlre - Les expressions rationnelles en Perl

2 DESCRIPTION

Cette page décrit la syntaxe des expressions rationnelles en Perl. (NdT: on emploie couramment le terme "expression régulière" car le terme anglais est "regular expression" qui s'abrège en "regexp". Mais ne nous y trompons pas, en français, ce sont bien des "expressions rationnelles".)

Si vous n'avez jamais utilisé les expressions rationnelles, vous trouverez une rapide introduction dans *perlrequick* et un tutoriel d'introduction un peu plus long dans *perlretut*.

Dans Opérateurs d'expression rationnelle in *perlop*, vous trouverez une présentation des opérateurs `m//`, `s///`, `qr//` et `??` avec une description de l'*usage* des expressions rationnelles dans des opérations de reconnaissances assortie de nombreux exemples.

Les opérations de reconnaissances peuvent utiliser différents modificateurs. Les modificateurs qui concernent l'interprétation des expressions rationnelles elles-mêmes sont présentés ici. Pour les modificateurs qui modifient l'usage des expressions rationnelles fait par Perl, regarder Opérateurs d'expression rationnelle in *perlop* et Les détails sordides de l'interprétation des chaînes in *perlop*.

i

Reconnaissance de motif indépendamment de la casse (majuscules/minuscules).

Si `use locale` est actif, la table des majuscules/minuscules est celle du locale courant. Voir *perllocale*.

m

Permet de traiter les chaînes multi-lignes. Les caractères "^" et "$" reconnaissent alors n'importe quel début ou fin de ligne plutôt qu'au début ou à la fin de la chaîne.

s

Permet de traiter une chaîne comme une seule ligne. Le caractère "." reconnaît alors n'importe quel caractère, même une fin de ligne qui normalement n'est pas reconnue.

Les modificateurs `/s` et `/m` passent outre le réglage de `$*`. C'est à dire que, quel que soit le contenu de `$*`, `/s` sans `/m` obligent "^" à reconnaître uniquement le début de la chaîne et "$" à reconnaître uniquement la fin de la chaîne (ou juste avant le retour à la ligne final). Combinés, par /ms, ils permettent à "." de reconnaître n'importe quel caractère tandis que "^" et "$" reconnaissent alors respectivement juste après ou juste avant un retour à la ligne dans la chaîne.

x

Augmente la lisibilité de vos motifs en autorisant les espaces et les commentaires.

Ils sont couramment nommés « le modificateur `/X` », même si le délimiteur en question n'est pas la barre oblique (`/`). En fait, tous ces modificateurs peuvent être inclus à l'intérieur de l'expression rationnelle elle-même en utilisant la nouvelle construction `(?...)`. Voir plus bas.

Le modificateur `/x` lui-même demande un peu plus d'explication. Il demande à l'interpréteur d'expressions rationnelles d'ignorer les espaces qui ne sont ni précédés d'une barre oblique inverse (\) ni à l'intérieur d'une classe de caractères. Vous pouvez l'utiliser pour découper votre expression rationnelle en parties (un peu) plus lisibles. Le caractère `#` est lui aussi traité comme un méta-caractère introduisant un commentaire exactement comme dans du code Perl ordinaire. Cela signifie que, si vous voulez de vrais espaces ou des `#` dans un motif (en dehors d'une classe de caractères qui n'est pas affectée par `/x`), vous devez les précéder d'un caractère d'échappement ou les coder en octal ou en hexadécimal. Prises ensembles, ces fonctionnalités rendent les expressions rationnelles de Perl plus lisibles. Faites attention à ne pas inclure le délimiteur de motif dans un commentaire (perl n'a aucun moyen de savoir que vous ne vouliez pas terminer le motif si tôt). Voir le code de suppression des commentaires C dans *perlop*.

2.1 Expressions rationnelles

Les motifs utilisés par la mise en correspondance de motifs sont des expressions rationnelles telles que fournies dans les routines de la Version 8 des expressions rationnelles. En fait, les routines proviennent (de manière éloignée) de la réécriture gratuitement redistribuable des routines de la Version 8 par Henry Spencer. Voir Version 8 des expressions rationnelles pour de plus amples informations.

Notamment, les méta-caractères suivants gardent leur sens à la *egrep* :

```
\   Annule le meta-sens du meta-caractère qui suit
^   Reconnaît le debut de la ligne
.   Reconnaît n'importe quel caractère (sauf le caractère nouvelle ligne)
$   Reconnaît la fin de la ligne (ou juste avant le caractère nouvelle ligne final)
|   Alternative
()  Groupement
[]  Classe de caractères
```

Par défaut, le caractère "^" ne peut reconnaître que le début de la ligne et le caractère "$" que la fin (ou juste avant le caractère nouvelle ligne de la fin) et Perl effectue certaines optimisations en supposant que la chaîne ne contient qu'une seule ligne. Les caractères nouvelle ligne inclus ne seront donc pas reconnus par "^" ou "$". Il est malgré tout possible de traiter une chaîne multi-lignes afin que "^" soit reconnu juste après n'importe quel caractère nouvelle ligne et "$" juste avant. Pour ce faire, au prix d'un léger ralentissement, vous devez utiliser le modificateur `/m` dans l'opérateur de reconnaissance de motif. (Les anciens programmes obtenaient ce résultat en positionnant `$*` mais cette pratique est maintenant désapprouvée.)

Pour faciliter les substitutions multi-lignes, le méta-caractère "." ne reconnaît jamais un caractère nouvelle ligne à moins d'utiliser le modificateur `/s` qui demande à Perl de considérer la chaîne comme une seule ligne même si ce n'est pas le cas. Le modificateur `/s` passe outre le réglage de `$*` au cas où vous auriez quelques vieux codes qui le positionnerait dans un autre module.

Les quantificateurs standards suivants sont reconnus :

```
*      Reconnaît 0 fois ou plus
+      Reconnaît 1 fois ou plus
?      Reconnaît 0 ou 1 fois
{n}    Reconnaît n fois exactement
{n,}   Reconnaît au moins n fois
{n,m}  Reconnaît au moins n fois mais pas plus de m fois
```

(Si une accolade apparaît dans n'importe quel autre contexte, elle est traitée comme un caractère normal. En particulier, la borne minimale n'est pas optionnelle) Le quantificateur "*" est équivalent à `{0,}`, le quantificateur "+" à `{1,}` et le quantificateur "?" à `{0,1}`. n et m sont limités à des valeurs entières (!) inférieures à une valeur fixée lors de la compilation de perl. Habituellement cette valeur est 32766 sur la plupart des plates-formes. La limite réelle peut être trouvée dans le message d'erreur engendré par le code suivant :

```
$_ **= $_ , / {$_} / for 2 .. 42;
```

Par défaut, un sous-motif quantifié est « gourmand », c'est à dire qu'il essaie de se reconnaître un maximum de fois (à partir d'un point de départ donné) sans empêcher la reconnaissance du reste du motif. Si vous voulez qu'il tente de se reconnaître un minimum de fois, il faut ajouter le caractère "?" juste après le quantificateur. Sa signification ne change pas. Seule sa « gourmandise » change :

```
*?     Reconnaît 0 fois ou plus
+?     Reconnaît 1 fois ou plus
??     Reconnaît 0 ou 1 fois
{n}?   Reconnaît n fois exactement
{n,}?  Reconnaît au moins n fois
{n,m}? Reconnaît au moins n fois mais pas plus de m fois
```

Puisque les motifs sont traités comme des chaînes entre guillemets, les séquences suivantes fonctionnent :

```
\t          tabulation             (HT, TAB)
\n          nouvelle ligne         (LF, NL)
\r          retour chariot         (CR)
\f          page suivante          (FF)
\a          alarme (bip)           (BEL)
\e          escape (pensez a troff)(ESC)
\033        caractère en octal (pensez au PDP-11)
\x1B        caractère hexadecimal
\x{263a}    caractère hexadecimal etendu (Unicode SMILEY)
\c[         caractère de controle
\N{nom}     caractère nomme
\l          convertit en minuscule le caractère suivant (pensez a vi)
\u          convertit en majuscule le caractère suivant (pensez a vi)
\L          convertit en minuscule jusqu'au prochain \E (pensez a vi)
\U          convertit en majuscule jusqu'au prochain \E (pensez a vi)
\E          fin de modification de casse (pensez a vi)
\Q          desactive les meta-caractères de motif jusqu'au prochain \E
```

Si `use locale` est actif, la table de majuscules/minuscules utilisée par `\l`, `\L`, `\u` et `\U` est celle du locale courant. Voir *perllocale*. Pour de la documentation concernant `\N{nom}`, voir *charnames*.

Vous ne pouvez pas inclure littéralement les caractères `$` et `@` à l'intérieur d'une séquence `\Q`. Tels quels, ils se référeraient à la variable correspondante. Précédés d'un `\`, ils correspondraient à la chaîne `\$` ou `\@`. Vous êtes obligés d'écrire quelque chose comme `m/\Quser\E\@\Qhost/`.

Perl définit aussi les séquences suivantes :

```
\w  Reconnaît un caractère de "mot" (alphanumérique plus "_")
\W  Reconnaît un caractère de non "mot"
\s  Reconnaît un caractère d'espace.
\S  Reconnaît un caractère autre qu'un espace
\d  Reconnaît un chiffre
\D  Reconnaît un caractère autre qu'un chiffre
\pP Reconnaît la propriété P (nommée). Utilisez \p{Prop}
    pour les noms longs
\X  Reconnaît une séquence d'un caractère Unicode étendue,
    équivalent à (?:\PM\pM*)
\C  Reconnaît un caractère (d'un seul octet) meme sous utft8.
```

\w reconnaît un seul caractère alphanumérique (un caractère de l'alphabet ou un chiffre) ou _, pas à un mot entier. Pour reconnaître un mot entier au sens des identificateurs Perl, vous devez utiliser \w+ (ce qui n'est pas la même chose que reconnaître les mots anglais ou français). Si use locale est actif, la liste de caractères alphanumériques couverts par \w dépend du locale courant. Voir *perllocale*. Vous pouvez utiliser \w, \W, \s, \S, \d, et \D à l'intérieur d'une classe de caractères mais si vous essayez des les utiliser comme borne d'un intervalle, ce n'est plus un intervalle et le "-" est compris littéralement. Si Unicode est actif, \s reconnaître aussi "\x{85}", "\x{2028} et "\x{2029}". Voir *perlunicode* pour les détails à propos de \pP, \PP et \X, et *perluniintro* pour tout ce qui concerne Unicode en général. Vous pouvez définir de nouvelles propriétés pour \p et \P, voir *perlunicode*.

La syntaxe POSIX des classes de caractères :

```
[:class:]
```

est aussi disponible. Les classes disponibles et leur équivalent via la barre oblique inversée (si il existe) sont les suivants :

```
alpha
alnum
ascii
blanck            [1]
cntrl
digit       \d
graph
lower
print
punct
space       \s    [2]
upper
word        \w    [3]
xdigit
```

[1]

Une extension GNU équivalente à [\t] (tous les espaces horizontaux).

[2]

Pas exactement équivalent à \s puisque [[:space:]] inclut aussi le (très rare) tabulateur vertical : "\ck" ou chr(11).

[3]

Une extension Perl, voir plus loin.

Par exemple, utilisez [:upper:] pour reconnaître tous les caractères minuscules. Remarquez que les crochets ([]) font partie de la construction [::] et non de la classe de caractères. Par exemple :

```
[01[:alpha:]%]
```

reconnaît les chiffres un et deux, le caractère pourcent et tous les caractères alphabétiques.

Voici un tableau d'équivalence des classes avec les constructions Unicode \p{} et avec les classes représentées par un caractère précédé d'un backslash lorsqu'elles existent :

```
[:...:]     \p{...}      backslash

alpha       IsAlpha
alnum       IsAlnum
ascii       IsASCII
blank       IsSpace
cntrl       IsCntrl
digit       IsDigit      \d
graph       IsGraph
lower       IsLower
print       IsPrint
punct       IsPunct
space       IsSpace
            IsSpacePerl  \s
upper       IsUpper
word        IsWord
xdigit      IsXDigit
```

Par exemple `[:lower:]` et `\p{IsLower}` sont équivalents.

Si le pragma `utf8` n'est pas actif mais que le pragma `locale` l'est, les classes sont corrélées via l'interface isalpha(3) (sauf pour "word" et "blank").

Les classes nommées non évidentes sont :

cntrl
: N'importe quel caractère de contrôle. Ce sont habituellement des caractères qui ne produisent aucune sortie mais qui, par contre, modifie le terminal : par exemple newline (passage à la ligne) et backspace (retour arrière) sont des caractères de contrôle. Tous les caractères pour lesquels ord() renvoie une valeur inférieure à 32 sont des caractères de contrôle (avec l'ASCII, les codages de caractères ISO Latin et Unicode) ainsi que le caractère dont la valeur ord() vaut 127 (`DEL`).

graph `graph`
: N'importe quel caractère alphanumérique ou de ponctuation (ou spécial).

print
: N'importe quel caractère alphanumérique, de ponctuation (ou spécial) ou espace.

punct
: N'importe quel caractère de ponctuation (ou spécial).

xdigit
: Un chiffre hexadécimal. Bien que peu utile (`[0-9A-Fa-f]` fonctionne très bien), elle est inclue pour la complétude.

Vous pouvez utilisez la négation d'une classe de caractères [::] en préfixant son nom par un '^'. C'est une extension de Perl. Par exemple :

```
POSIX       traditionnel  Unicode

[:^digit:]      \D        \P{IsDigit}
[:^space:]      \S        \P{IsSpace}
[:^word:]       \W        \P{IsWord}
```

Perl respecte le standard POSIX puisque les classes de caractères POSIX ne sont supportées qu'à l'intérieur d'une classe de caractères. Les deux classes de caractères POSIX [.cc.] et [=cc=] sont reconnues **sans** être supportées : toute tentative d'utilisation de ces classes provoquera une erreur.

Perl définit les assertions de longueur nulle suivantes :

```
\b  Reconnaît la limite d'un mot
\B  Reconnaît autre chose qu'une limite de mot
\A  Reconnaît uniquement le début de la chaîne
\Z  Reconnaît uniquement la fin de la chaîne (ou juste avant le
    caractère de nouvelle ligne final)
\z  Reconnaît uniquement la fin de la chaîne
\G  Reconnaît l'endroit ou s'est arrete le precedent m//g
    (ne fonctionne qu'avec /g)
```

Une limite de mot (`\b`) est définie comme le point entre deux caractères qui sont d'un côté un `\w` et de l'autre un `\W` (dans n'importe quel ordre). Le début et la fin de la chaîne correspondent à des caractères imaginaires de type `\W`. (À l'intérieur d'une classe de caractères, `\b` représente le caractère "backspace" au lieu d'une limite de mot.) `\A` et `\Z` agissent exactement comme "^" et "$" sauf qu'ils ne reconnaissent pas les lignes multiples quand le modificateur `/m` est utilisé alors que "^" et "$" reconnaissent toutes les limites de lignes internes. Pour reconnaître la fin réelle de la chaîne, en tenant compte du caractère nouvelle ligne, vous devez utiliser `\z`.

`\G` est utilisé pour enchaîner plusieurs mises en correspondance (en utilisant `m//g`). Voir Opérateurs d'expression rationnelle in *perlop*. Il est aussi utile lorsque vous écrivez un analyseur lexicographique à la `lex` ou lorsque vous avez plusieurs motifs qui doivent reconnaître des sous-chaînes successives de votre chaîne. Voir la référence précédente. L'endroit réel à partir duquel `\G` va être reconnu peut être modifié en affectant une nouvelle valeur à `pos()`. Voir pos in *perlfunc*. Actuellement, `\G` n'est utilisable que lorsqu'il est placé en tout début de motif; bien qu'utilisable partout en théorie, comme dans `/(?<=\G..)./g`, quelques cas (comme, par exemple, `/.\G/g`) posent problème et il est donc recommandé de ne pas le faire.

La construction `(...)` crée des zones de mémorisation (des sous-motifs). Pour vous référer au *n*-ième sous-motifs, utilisez `\`*n* à l'intérieur du motif. À l'extérieur du motif, utilisez toujours "$" à la place de "\" devant *n*. (Bien que la notation `\`*n*

fonctionne en de rares occasions à l'extérieur du motif courant, vous ne devriez pas compter dessus. Voir l'avertissement au sujet de \1 et de $1.) Une référence à un sous-motif mémorisé est appelée une référence arrière.

Vous pouvez utiliser autant de sous-motifs que nécessaire. Par contre, Perl utilise les séquences \10, \11, etc. comme des synonymes de \010, \011, etc. (Souvenez-vous que 0 signifie octal et donc \011 est le caractère codé par un neuf dans votre jeu de caractère, une tabulation en ASCII.) Perl résoud cette ambiguïté en interprétant \10 comme une référence arrière uniquement si il y a déjà au moins 10 parenthèses ouvrantes avant. De même, \11 sera une référence arrière uniquement si il y a au moins onze parenthèses ouvrantes avant. Et ainsi de suite. Les séquences de \1 à \9 sont toujours interprétées comme des références arrières.

Exemples :

```
s/^([^ ]*) *([^ ]*)/$2 $1/;     # echange les deux premiers mots

 if (/(.)\1/) {                 # trouve le premier caractère répété
     print "'$1' is the first doubled character\n";
 }

if (/Time: (..):(..):(..)/) {
    $hours = $1;
    $minutes = $2;
    $seconds = $3;
}
```

Plusieurs variables spéciales se réfèrent à des portions de la dernière reconnaissance. `$+` renvoie le dernier sous-motif entre parenthèses reconnu. `$&` renvoie le dernier motif reconnu. (Autrefois `$0` était utilisé pour le même usage mais maintenant, il renvoie le nom du programme.) `$`` renvoie tout ce qui est avant le motif reconnu. `$'` renvoie tout ce qui est après le motif reconnu. Et `$^N` renvoie ce qui a été reconnu par le dernier sous-motif refermé. `$^N` peut-être utilisé dans les motifs étendus (voir plus bas), par exemple pour affecter un sous-motif à une variable.

Les variables numérotées ($1, $2, $3, etc.) et les variables spéciales ci-dessus (`$+`, `$&`, `$`` , `$'` et `$^N`) ont toutes une portée dynamique allant jusqu'à la fin du bloc englobant ou jusqu'à la prochaine reconnaissance réussie selon ce qui arrive en premier. (Voir Instructions composées in *perlsyn*.)

Note : en Perl, l'échec d'une tentative de reconnaissance ne réinitialise pas ces variables. Cela facilite l'écriture de codes qui testent une succession de cas de plus en plus spécifiques et qui mémorisent la meilleure reconnaissance.

Attention : à partir du moment où perl voit que vous avez besoin de l'une des variables `$&`, `$`` ou `$'` quelque part dans votre programme, il les calculera à chaque reconnaissance de motif et ce pour tous les motifs. Cela peut ralentir considérablement votre programme. Le même mécanisme est utilisé lors de l'utilisation de $1, $2, etc.. Ce ralentissement a donc lieu aussi pour les motifs mémorisant des sous-motifs. (Pour éviter ce ralentissement tout en conservant la possibilité d'utiliser des regroupements, utilisez l'extension `(?:...)` à la place.) Mais si vous n'utilisez pas $&, etc. dans vos scripts alors vos motifs *sans* mémorisation ne seront pas pénalisés. Donc, évitez `$&`, `$'` et `$`` si vous le pouvez. Dans le cas contraire (et certains algorithmes en ont réellement besoin), si vous les utilisez une fois, utilisez-les partout car vous en supportez déjà le coût. Depuis la version 5.005, `$&` n'est plus aussi coûteux que les deux autres.

Les méta-caractères précédés d'un caractère barre oblique inversée en Perl sont alphanumériques tels \b, \w, \n. À l'inverse d'autres langages d'expressions rationnelles, il n'y a pas de symbole précédé d'un caractère barre oblique inversée qui ne soit pas alphanumérique. Donc tout ce qui ressemble à \\, \(, \), \<, \>, \{, ou \} est toujours interprété littéralement et non comme un méta-caractère. Ceci est utilisé pour désactiver (ou «quoter») les éventuels méta-caractères présents dans une chaîne que vous voulez utiliser comme motif. Tout simplement, il vous suffit de précéder tous les caractères non-"mot" par un caractère barre oblique inversée :

```
$pattern =~ s/(\W)/\\$1/g;
```

(Si `use local` est actif alors le résultat dépend de votre locale courant.) Aujourd'hui, il est encore plus simple d'utiliser soit le fonction quotemeta() soit la séquence \Q pour désactiver les éventuels méta-caractères :

```
/$unquoted\Q$quoted\E$unquoted/
```

Sachez que si vous placez une barre oblique inversée directement (pas celles qui sont dans des variables interpolées) entre \Q et \E, la double interpolation peut vous amener des résultats très étonnants. Si vous avez *besoin* de constructions de ce genre, consultez Les détails sordides de l'interprétation des chaînes in *perlop*

2.2 Motifs étendus

Perl intègre une extension de la syntaxe des expressions rationnelles afin d'ajouter les fonctionnalités inexistantes dans les outils standard tels que **awk** et **lex**. La syntaxe est une paire de parenthèses avec un point d'interrogation comme premier caractère entre les parenthèses. Le caractère qui suit le point d'interrogation précise l'extension choisie.

La stabilité de ces extensions est très variable. Certaine font partie du langage depuis de longues années. D'autres sont encore expérimentales et peuvent changer sans préavis ou même être retirées complètement. Vérifier le documentation de chacune de ces extensions pour connaître leur état.

Le point d'interrogation a été choisi pour les extensions ainsi que pour les modificateurs non gourmands parce que 1) les points d'interrogation sont rares dans les vieilles expressions rationnelles et 2) pour qu'à chaque fois que vous en voyez un, vous vous arrêtiez pour vous "interroger" sur son comportement. C'est psychologique...

(?#texte)
: Un commentaire. Le texte est ignoré. Si le modificateur `/x` est utilisé pour autoriser la mise en forme avec des blancs, un simple `#` devrait suffire. Notez que Perl termine le commentaire dès qu'il rencontre `)`. Il n'y a donc aucun moyen de mettre le caractère `)` dans ce commentaire.

(?imsx-imsx)
: Un ou plusieurs modificateurs de motifs à activer (ou à désactiver s'ils sont précédés par `-`) jusqu'à la fin du motif ou du sout-motif courant. C'est très pratique pour les motifs dynamiques tels ceux lus depuis un fichier de configuration, ceux provenant d'un argument ou ceux qui sont spécifiés dans une table quelque part. Certains devront être sensibles à la casse, d'autres non. Dans le cas où un motif ne doit pas être sensible à la casse, il est possible d'inclure `(?i)` au début du motif. Par exemple :

```
$pattern = "foobar";
if ( /$pattern/i ) { }

# plus flexible :

$pattern = "(?i)foobar";
if ( /$pattern/ ) { }
```

: Ces modificateurs sont locaux au groupe englobant (si il existe). Par exemple :

```
( (?i) blah ) \s+ \1
```

: reconnaîtra un mot `blah` répété (*y compris sa casse* !) (en supposant le modificateur `x` et aucun modificateur `i` à l'extérieur du groupe).

(?:motif)

(?imsx-imsx:motif)
: C'est pour regrouper et non pas mémoriser. Cela permet de regrouper des sous-expressions comme "()" mais sans permettre les références arrière. Donc :

```
@fields = split(/\b(?:a|b|c)\b/)
```

: est similaire à

```
@fields = split(/\b(a|b|c)\b/)
```

: mais ne produit pas de mémorisations supplémentaires. C'est moins couteux de ne pas mémoriser des caractères si vous n'en avez pas besoin.

: Les lettres entre `?` et `:` agissent comme des modificateurs. Voir `(?imsx-imsx)`. Par exemple :

```
/(?s-i:more.*than).*million/i
```

: est équivalent à l'expression plus verbeuse

```
/(?:(?s-i)more.*than).*million/i
```

(?=motif)
: Une assertion de longueur nulle pour tester la présence de quelque chose en avant. Par exemple, `/\w+(?=\t)/` reconnaît un mot suivit d'une tabulation sans inclure cette tabulation dans `$&`.

(?!motif)

Une assertion de longueur nulle pour tester l'absence de quelque chose en avant. Par exemple, `/foo(?!bar)/` reconnaît toutes les occurrences de "foo" qui ne sont pas suivies de "bar". Notez bien que regarder en avant n'est pas la même chose que regarder en arrière (!). Vous ne pouvez pas utiliser cette assertion pour regarder en arrière.

Si vous cherchez un "bar" qui ne soit pas précédé par "foo", `/(?!foo)bar/` ne vous donnera pas ce que vous voulez. C'est parce que `(?!foo)` exige seulement que ce qui suit ne soit pas "foo" – et ça ne l'est pas puisque c'est "bar", donc "foobar" sera accepté. Vous devez alors utilisez quelque chose comme `/(?!foo)...bar/`. Nous disons "comme" car il se peut que "bar" ne soit pas précédé par trois caractères. Vous pouvez alors utiliser `/(?:(?!foo)...|^.{0,2})bar/`. Parfois, il est quand même plus simple de dire :

```
if (/bar/ && $` !~ /foo$/)
```

Pour regarder en arrière, voir plus loin.

(?<=motif)

Une assertion de longueur nulle pour tester la présence de quelque chose en arrière. Par exemple, `/(?<=\t)\w+/` reconnaît un mot qui suit une tabulation sans inclure cette tabulation dans `$&`. Cela ne fonctionne qu'avec un motif de longueur fixe.

(?<!motif)

Une assertion de longueur nulle pour tester l'absence de quelque chose en arrière. Par exemple, `/(?<!bar)foo/` reconnaît toutes les occurrences de "foo" qui ne suivent pas "bar". Cela ne fonctionne qu'avec un motif de longueur fixe.

(?{ code })

ATTENTION : cette extension est considérée comme expérimentale. Elle peut évoluer voir même disparaître sans préavis.

Une assertion de longueur nulle permettant l'évaluation de code Perl. Elle est reconnue dans tous les cas et le `code` n'est pas soumis à interpolation. Actuellement les règles pour déterminer où le `code` se termine sont quelque peu compliquées.

Cette fonctionnalité, utilisée conjointement avec la variable spéciale `$^N`, permet de mémoriser dans des variables les résultats des sous-groupes reconnues sans se préoccuper du nombre de parenthèses imbriquées. Par exemple :

```
$_ = "The brown fox jumps over the lazy dog";
/the (\S+)(?{ $color = $^N }) (\S+)(?{ $animal = $^N })/i;
print "color = $color, animal = $animal\n";
```

À l'intérieur du bloc `(?{...})`, `$_` fait référence à la chaîne sur laquelle s'applique l'expression rationnelle. Vous pouvez aussi utiliser `pos()` pour connaître la position actuelle dans cette chaîne.

Le `code` a une portée correcte dans le sens où, si il y a un retour arrière sur l'assertion (voir Retour arrière (§2.3)), tous les changements introduits après la `local`isation sont annulés. Donc :

```
$_ = 'a' x 8;
m<
   (?{ $cnt = 0 })              # Initialisation de $cnt.
   (
     a
     (?{
         local $cnt = $cnt + 1;  # Mise a jour de $cnt,
                                 # (en tenant compte du retour arriere)
     })
   )*
   aaaa
   (?{ $res = $cnt })           # En cas de succes, recopie vers une
                                # variable non local-isee.
 >x;
```

produit `$res = 4`. Remarquez qu'après la reconnaissance, $cnt revient à la valeur 0 affectée globalement puisque nous ne sommes plus dans la portée du bloc où l'appel à `local` est effectué.

Cette assertion peut être utilisée comme sélecteur: `(?(condition)motif-oui|motif-non)`. Si elle n'est *pas* utilisée comme cela, le résultat de l'évaluation du `code` est affecté à la variable spéciale `$^R`. L'affectation est immédiate donc `$^R` peut être utilisée dans une autre assertion de type `(?{ code })` à l'intérieur de la même expression rationnelle.

L'affectation à $^R est correctement localisée, par conséquent l'ancienne valeur de $^R est restaurée en cas de retour arrière (voir Retour arrière (§2.3)).

Pour des raisons de sécurité, cette construction n'est pas autorisée si l'expression rationnelle nécessite une interpolation de variables lors de l'exécution sauf si vous utilisez le directive `use re 'eval'` (voir *re*) ou si la variable contient le résultat de l'opérateur qr() (voir qr/STRING/imosx in *perlop*).

Cette restriction est due à l'usage très courant et remarquablement pratique de motifs déterminés lors de l'exécution. Par exemple :

```
$re = <>;
chomp $re;
$string =~ /$re/;
```

Avant que Perl sache comment exécuter du code interpolé à l'intérieur d'un motif, cette opération était complètement sûre d'un point de vue sécurité bien qu'elle puisse générer une exception si le motif n'est pas légal. Par contre, si vous activez la directive `use re 'eval'`, cette construction n'est plus du tout sûre. Vous ne devriez donc le faire que lorsque vous utilisez la vérification des données (via taint et l'option `-T`). Mieux encore, vous pouvez utiliser une évaluation contrainte dans un compartiement Safe. Voir *perlsec* pour les détails à propos de ces mécanismes.

`(??{ code })`

ATTENTION : cette extension est considérée comme expérimentale. Elle peut évoluer voir même disparaître sans préavis. Une version simplifiée de la syntaxe pourrait être introduite pour les cas les plus courants.

C'est un sous-motif d'expression rationnelle à "évaluation retardée". Le `code` est évalué lors de l'exécution au moment où le sous-motif est reconnu. Le résultat de l'évaluation est considéré comme une expression rationnelle dont on doit tenter la reconnaissance comme si elle remplaçait la construction.

Le `code` n'est pas interpolé. Comme précédemment, les règles pour déterminer l'endroit où se termine le `code` sont un peu compliquées.

La motif suivant reconnaît des groupes parenthésés :

```
$re = qr{
           \(
           (?:
              (?> [^()]+ )     # Non-parens without backtracking
            |
              (??{ $re })      # Group with matching parens
           )*
           \)
         }x;
```

`(?>motif)`

ATTENTION : cette extension est considérée comme expérimentale. Elle peut évoluer voir même disparaître sans préavis.

Une sous-expression "indépendante". Reconnaît **uniquement** la sous-chaîne qui aurait été reconnue si la sous-expression avait été seule et ancrée au même endroit. C'est pratique, par exemple pour optimiser certaines constructions qui risqueraient sinon d'être "éternelles", car cette construction n'effectue aucun retour arrière (voir Retour arrière (§2.3)). C'est aussi pratique lorsqu'on souhaite le comportement "prend tout ce que tu peux sans jamais redonner quoique ce soit".

Par exemple : `^(?>a*)ab` ne pourra jamais être reconnu puisque `(?>a*)` (ancré au début de la chaîne) reconnaîtra *tous* les caractères `a` du début de la chaîne en ne laissant aucun `a` pour reconnaître `ab`. A contrario, `a*ab` reconnaîtra la même chose que `a+b` puisque la reconnaissance du sous-groupe `a*` est influencé par le groupe suivant `ab` (voir Retour arrière (§2.3)). En particulier, `a*` dans `a*ab` reconnaît moins de caractères que `a*` seul puisque cela permet à la suite d'être reconnue.

Un effet similaire à `(?>motif)` peut être obtenu en écrivant `(?=(motif))\1`. La première partie de l'expression reconnaît la même sous-chaîne qu'une expression `a+` seule et ensuite le `\1` mange la chaîne reconnue et transforme donc l'assertion de longueur nulle en une expression analogue à `(?>...)`. (La seule différence entre ces deux expressions est que la dernière utilise un groupe mémorisé et décale donc le numéro des références arrières dans le reste de l'expression rationnelle.)

Supposons le motif suivant :

```
m{ \(
      (
        [^()]+       # x+
      |
        \( [^()]* \)
      )+
   \)
 }x
```

L'exemple précédent reconnaît de manière efficace un groupe non-vide qui contient au plus deux niveaux de parenthèses. Par contre, si un tel groupe n'existe pas, cela prendra un temps potentiellement infini sur une longue chaîne car il existe énormément de manières différentes de découper une chaîne en sous-chaînes. C'est ce que fait `(.+)+` qui est similaire à l'un des sous-motifs du motif précédent. Rendez-vous compte que l'expression précédente détecte la non reconnaissance sur `(()aaaaaaaaaaaaaaaaa` en quelques secondes mais que chaque lettre supplémentaire multiplie ce temps par deux. Cette augmentation exponentielle du temps d'exécution peut faire croire que votre programme est planté. Par contre, le même motif très légèrement modifié :

```
m{ \(
      (
        (?> [^()]+ )    # remplacement de x+ par (?> x+ )
      |
        \( [^()]* \)
      )+
   \)
 }x
```

en utilisant `(?>...)`, reconnaît exactement la même chose (un bon exercice serait de le vérifier vous-même) mais se termine en 4 fois moins de temps sur une chaîne similaire contenant 1000000 `a`. Attention, ce motif produit actuellement un message d'avertissement avec **-w** disant `"matches null string many times in regexp"` (`"reconnaît la chaîne de longueur nulle très souvent dans une regexp"`).

Dans des motifs simples comme `(?> [^()]+ )`, un effet comparable peut être observé en utilisant le test d'absence en avant comme dans `[^()]+ (?! [^()] )`. Ce n'est que 4 fois plus lent sur une chaîne contenant 1000000 `a`.

La comportement "prend tout ce que tu peux sans jamais redonner quoique ce soit" est utile dans de nombreuses situations où une première analyse laisse à penser qu'un simple motif `()*` suffit. Supposez que vous voulez analyser un texte avec des commentaires délimités par `#` suivi d'éventuels espaces (horizontaux). Contrairement aux apparences, `#[ \t]*` n'est *pas* le sous-motif correct pour reconnaître le délimiteur de commentaires parce qu'il peut laisser échapper quelques espaces si la suite du motif en a besoin pour être reconnue. Le réponse correcte est l'une de celles qui suivent :

```
(?>#[ \t]*)
#[ \t]*(?![ \t])
```

Par exemple, pour obtenir tous les commentaires non vides dans $1, il vous faut utiliser l'une de ces constructions :

```
/ (?> \# [ \t]* ) (         .+ ) /x;
/     \# [ \t]*   ( [^ \t] .* ) /x;
```

La meilleure des deux est celle qui correspondrait le mieux à la description des commentaires faite par les spécifications.

`(?(condition)motif-oui|motif-non)`

`(?(condition)motif-oui)`

ATTENTION : cette extension est considérée comme expérimentale. Elle peut évoluer voir même disparaître sans préavis.

Expression conditionnelle. `(condition)` est soit un entier entre parenthèses (qui est vrai si le sous-motif mémorisé correspondant reconnaît quelque chose), soit une assertion de longueur nulle (test en arrière, en avant ou évaluation).

Par exemple :

```
m{ ( \( )?
   [^()]+
   (?(1) \) )
 }x
```

reconnaît une suite de caractères tous différents des parenthèses éventuellement entourée d'une paire de parenthèses.

2.3 Retour arrière

NOTE : cette section présente une abstraction approximative du comportement des expressions rationnelles. Pour une vue plus rigoureuse (et compliquée) des règles utilisées pour choisir entre plusieurs reconnaissances possibles, voir Combinaison d'expressions rationnelles.

Une particularité fondamentale de la reconnaissance d'expressions rationnelles est liée à la notion de *retour arrière* qui est actuellement utilisée (si nécessaire) par tous les quantificateurs d'expressions rationnelles. À savoir `*`, `*?`, `+`, `+?`, `{n,m}` et `{n,m}?`. Les retours arrières sont parfois optimisés en interne mais les principes généraux exposés ici restent valides.

Pour qu'une expression rationnelle soit reconnue, *toute* l'expression doit être reconnue, pas seulement une partie. Donc si le début de l'expression rationnelle est reconnue mais de telle sorte qu'elle empêche la reconnaissance de la suite du motif, le moteur de reconnaissance revient en arrière pour calculer autrement le début – d'où le nom retour arrière.

Voici un exemple de retour arrière. Supposons que vous voulez trouver le mot qui suit "foo" dans la chaîne "Food is on the foo table.":

```
$_ = "Food is on the foo table.";
if ( /\b(foo)\s+(\w+)/i ) {
    print "$2 suit $1.\n";
}
```

Lors de la reconnaissance, la première partie de l'expression rationnelle (`\b(foo)`) trouve un point d'ancrage dès le début de la chaîne et stocke "Foo" dans $1. Puis le moteur de reconnaissance s'aperçoit qu'il n'y pas de caractère d'espacement après le "Foo" qu'il a stocké dans $1 et, réalisant son erreur, il recommence alors un caractère plus loin que cette première tentative. Cette fois, il va jusqu'à la prochaine occurrence de "foo", l'ensemble de l'expression rationnelle est reconnue et vous obtenez comme prévu "table suit foo."

Parfois la reconnaissance minimale peut aider. Imaginons que vous souhaitez reconnaître tout ce qu'il y a entre "foo" et "bar". Tout d'abord, vous écrivez quelque chose comme :

```
$_ =  "The food is under the bar in the barn.";
if ( /foo(.*)bar/ ) {
    print "got <$1>\n";
}
```

qui, de manière inattendue, produit :

```
got <d is under the bar in the >
```

C'est parce que `.*` est gourmand. Vous obtenez donc tout ce qu'il y a entre le *premier* "foo" et le *dernier* "bar". Dans ce cas, la reconnaissance minimale est efficace pour vous garantir de reconnaître tout ce qu'il y a entre un "foo" et le premier "bar" qui suit.

```
  if ( /foo(.*?)bar/ ) { print "got <$1>\n" }
got <d is under the >
```

Voici un autre exemple. Supposons que vous voulez reconnaître un nombre à la fin d'une chaîne tout en mémorisant ce qui précède. Vous écrivez donc :

```
$_ = "I have 2 numbers: 53147";
if ( /(.*)(\d*)/ ) {                                # Rate!
    print "Beginning is <$1>, number is <$2>.\n";
}
```

Cela ne marche pas parce que `.*` est gourmand et engloutit toute la chaîne. Puisque `\d*` peut reconnaître la chaîne vide, toute l'expression rationnelle est reconnue.

```
Beginning is <I have 2 numbers: 53147>, number is <>.
```

Voici quelques variantes dont la plupart ne marche pas :

```
$_ = "I have 2 numbers: 53147";
@pats = qw{
    (.*)(\d*)
    (.*)(\d+)
    (.*?)(\d*)
    (.*?)(\d+)
    (.*)(\d+)$
    (.*?)(\d+)$
    (.*)\b(\d+)$
    (.*\D)(\d+)$
};
```

```
for $pat (@pats) {
    printf "%-12s ", $pat;
    if ( /$pat/ ) {
        print "<$1> <$2>\n";
    } else {
        print "FAIL\n";
    }
}
```

qui affichera :

```
(.*)(\d*)    <I have 2 numbers: 53147> <>
(.*)(\d+)    <I have 2 numbers: 5314> <7>
(.*?)(\d*)   <> <>
(.*?)(\d+)   <I have > <2>
(.*)(\d+)$   <I have 2 numbers: 5314> <7>
(.*?)(\d+)$  <I have 2 numbers: > <53147>
(.*)\b(\d+)$ <I have 2 numbers: > <53147>
(.*\D)(\d+)$ <I have 2 numbers: > <53147>
```

Comme vous pouvez le constater, c'est un peu délicat. Il faut comprendre qu'une expression rationnelle n'est qu'un ensemble d'assertions donnant une définition du succès. Il peut y avoir aucun, un ou de nombreux moyens de répondre à cette définition lorsqu'on l'applique à une chaîne particulière. Et lorsqu'il y a plusieurs moyens, vous devez comprendre le retour arrière pour savoir quelle variété de succès vous obtiendrez.

Avec l'utilisation des assertions de tests d'absence ou de présence, cela peut devenir carrément épineux. Supposons que vous souhaitez retrouver une suite de caractères non numériques suivie par "123". Vous pouvez essayer d'écrire quelque chose comme :

```
$_ = "ABC123";
if ( /^\D*(?!123)/ ) {              # Rate!
    print "Heu, pas de 123 dans $_\n";
}
```

Mais ça ne fonctionne pas... tout du moins pas comme vous l'espériez. Ça affiche qu'il n'y a pas de 123 à la fin de la chaîne. Voici une présentation claire de ce qui est reconnu par ces motifs contrairement à ce qu'on pourrait attendre :

```
$x = 'ABC123' ;
$y = 'ABC445' ;

print "1: got $1\n" if $x =~ /^(ABC)(?!123)/ ;
print "2: got $1\n" if $y =~ /^(ABC)(?!123)/ ;

print "3: got $1\n" if $x =~ /^(\D*)(?!123)/ ;
print "4: got $1\n" if $y =~ /^(\D*)(?!123)/ ;
```

Affiche :

```
2: got ABC
3: got AB
4: got ABC
```

On aurait pu croire que le test 3 échouerait puisqu'il semble être une généralisation du test 1. La différence importante entre les deux est que le test 3 contient un quantificateur (`\D*`) et peut donc effectuer des retours arrière alors que le test 1 ne le peut pas. En fait, ce que demande le test 3 c'est "en partant du début de $x, peut-on trouver quelque chose qui n'est pas 123 précédé par 0 ou plusieurs caractères autres que des chiffres ?". Si on laisse `\D*` reconnaître "ABC", cela entraîne l'échec du motif complet.

Le moteur de reconnaissance met tout d'abord en correspondance `\D*` avec "ABC". Puis il essaie de mettre en correspondance `(?!123)` avec "123", ce qui évidemment échoue. Mais puisque un quantificateur (`\D*`) est utilisé dans l'expression

rationnelle, le moteur de reconnaissance peut faire des retours arrière pour tenter une reconnaissance différente afin de reconnaître l'ensemble de l'expression rationnelle.

Le motif veut *vraiment* être reconnu, alors il utilise le mécanisme de retour arrière et limite cette fois l'expansion de `\D*` juste à "AB". Maintenant, il y a réellement quelque chose qui suit "AB" et qui n'est pas "123 : c'est "C123". C'est suffisant pour réussir.

On peut faire la même chose en mixant l'utilisation des assertions de présence et d'absence. Nous dirons alors que la première partie doit être suivie par un chiffre mais doit aussi être suivie par quelque chose qui n'est pas "123". Souvenez-vous que les tests en avant sont des assertions de longueur nulle – ils ne font que regarder mais ne consomment pas de caractères de la chaîne lors de leur reconnaissance. Donc, réécrit comme suit, cela produira le résultat attendu. C'est à dire que le test 5 échouera et le 6 marchera :

```
print "5: got $1\n" if $x =~ /^(\D*)(?=\d)(?!123)/ ;
print "6: got $1\n" if $y =~ /^(\D*)(?=\d)(?!123)/ ;

6: got ABC
```

En d'autres termes, cela signifie que les deux assertions de longueur nulle consécutives doivent être vraies toutes les deux ensembles, exactement comme quand vous utilisez des assertions prédéfinies : `/^$/` est reconnue uniquement si vous êtes simultanément au début ET à la fin de la ligne. La réalité sous-jacente est que la juxtaposition de deux expressions rationnelles signifie toujours un ET sauf si vous écrivez explicitement un OU en utilisant la barre verticale. `/ab/` signifie reconnaître "a" ET (puis) reconnaître "b", même si les tentatives de reconnaissance n'ont pas lieu à la même position puisque "a" n'est pas une assertion de longueur nulle (mais de longueur un).

Avertissement : certaines expressions rationnelles compliquées peuvent prendre un temps exponentiel pour leur résolution à cause du grand nombre de retours arrières effectués pour essayer d'être reconnue. Par exemple, sans les optimisations internes du moteur d'expressions rationnelles, l'expression suivante calculerait très longtemps :

```
'aaaaaaaaaaaa' =~ /((a{0,5}){0,5})*[c]/
```

et si vous utilisiez des `*` au lieu de limiter entre 0 et 5 reconnaissances, elle pourrait alors prendre littéralement un temps infini – ou plutôt jusqu'au dépassement de la capacité de la pile. De plus, ces optimisations ne sont pas toujours applicables. Par exemple, en remplaçant `{0,5}` par `*` dans le groupe externe, plus aucune optimisation n'a lieu et le calcul devient extrêmement long.

Un outil puissant pour optimiser ce genre de choses est ce qu'on appelle les "groupes indépendants" qui n'effectuent pas de retour arrière (voir `(?`motif`)`>). Remarquez aussi que les assertions de longueur nulle n'effectuent pas de retour arrière puisqu'elles sont considérées dans un contexte "logique" : seul importe qu'elles soient reconnaissables ou non. Pour un exemple de l'influence éventuelle sur la reconnaissance voir `(?`motif`)`>.

2.4 Version 8 des expressions rationnelles

Si vous n'êtes pas familier avec la version 8 des expressions rationnelles, vous trouverez ici les règles de reconnaissance de motifs qui n'ont pas été décrites plus haut.

Un caractère se reconnaît lui-même sauf si c'est un *méta-caractère* avec un sens spécial décrit ici ou précédemment. Pour interpréter un méta-caractère littéralement, il faut le préfixer par un "\" (c.-à-d., "\." reconnaît un "." au lieu de n'importe quel caractère, "\\" reconnaît "\"). Une suite de caractères reconnaît cette suite de caractères dans la chaîne à analyser. Le motif `blurfl` reconnaîtra donc "blurfl" dans la chaîne à analyser.

Vous pouvez spécifier une classe de caractères en entourant une liste de caractères entre `[]`. Cette classe reconnaîtra l'un des caractères de la liste. Si le premier caractère après le "[" est "^", la classe reconnaîtra un caractère qui n'est pas dans la liste. À l'intérieur d'une liste, la caractère "-" sert à décrire un intervalle. Par exemple `a-z` représente tous les caractères entre "a" et "z" inclus. Si vous voulez inclure dans la classe le caractère "-" ou "]" lui-même, mettez le au début de la classe (après un éventuel "^") ou préfixez le par un "\". "-" est aussi interprété littéralement si il est placé en fin de classe, juste avant le "]". (Les exemples suivants décrivent tous la même classe de trois caractères: `[-az]`, `[az-]`, `[a\-z]`. Ils sont tous différents de `[a-z]` qui décrit une classe contenant 26 caractères.) Notez aussi que si vous essayez d'utiliser `\w`, `\W`, `\s`, `\S`, `\d` ou `\D` comme borne d'un intervalle, ce ne sera pas un intervalle et le "-" sera interprété littéralement.

Remarquez aussi que le concept d'intervalle de caractères n'est pas vraiment portable entre différents codages – et même dans un même codage, cela peut produire un résultat que vous n'attendez pas. Un bon principe de base est de n'utiliser que des intervalles qui commencent et se terminent dans le même alphabet ([a-e], [A-E]) ou dans les chiffres ([0-9]). Tout le reste n'est pas sûr. Dans le doute, énumérez l'ensemble des caractères explicitement.

Certains caractères peuvent être spécifiés avec la syntaxe des méta-caractères comme en C : "\n" reconnaît le caractère nouvelle ligne, "\t" reconnaît une tabulation, "\r" reconnaît un retour chariot, "\f" reconnaît un changement de page, etc. Plus généralement, *nnn*, où *nnn* est une suite de chiffres octaux, reconnaît le caractère dont le code ASCII est *nnn*. De même, \x*nn*, où *nn* est une suite de chiffres hexadécimaux, reconnaît le caractère dont le code ASCII est *nn*. L'expression \c*x* reconnaît le caractère ASCII control-*x*. Pour terminer, le méta-caractère "." reconnaît n'importe quel caractère sauf "\n" (à moins d'utiliser `/s`).

Vous pouvez décrire une série de choix dans un motif en les séparant par des "|". Donc `fee|fie|foe` reconnaîtra n'importe quel "fee", "fie" ou "foe" dans la chaîne à analyser (comme le ferait `f(e|i|o)e`). Le premier choix inclut tout ce qui suit le précédent délimiteur de motif ("(", "[" ou le début du motif) jusqu'au premier "|" et le dernier choix inclut tout ce qui suit le dernier "|" jusqu'au prochain délimiteur de motif. C'est pour cette raison qu'il est courant d'entourer les choix entre parenthèses pour minimiser les risques de mauvaises interprétation de début et de fin.

Les différents choix sont essayés de la gauche vers la droite et le premier choix qui autorise la reconnaissance de l'ensemble du motif est retenu. Ce qui signifie que les choix ne sont pas obligatoirement "gourmand". Par exemple: en appliquant `foo|foot` à "barefoot", seule la partie "foo" est reconnue puisque c'est le premier choix essayé et qu'il permet la reconnaissance du motif complet. (Cela peu sembler anodin mais ne l'est pas lorsque vous mémorisez du texte grâce aux parenthèses.)

Souvenez-vous aussi que "|" est interprété littéralement lorsqu'il est présent dans une classe de caractères. Donc si vous écrivez `[fee|fie|foe]`, vous recherchez en fait `[feio|]`.

À l'intérieur d'un motif, vous pouvez mémoriser ce que reconnaît un sous-motif en l'entourant de parenthèses. Plus loin dans le motif, vous pouvez faire référence au *n*-ième sous-motif mémorisé en utilisant le méta-caractère *n*. Les sous-motifs sont numérotés de droite à gauche dans l'ordre de leurs parenthèses ouvrantes. Une référence reconnaît exactement la chaîne actuellement reconnue par le sous-motif correspondant et non pas n'importe quelle chaîne qu'il aurait pu reconnaître. Par conséquent, `(0|0x)\d*\s\1\d*` reconnaîtra "0x1234 0x4321" mais pas "0x1234 01234" car, même si `(0|0x)` peut reconnaître le 0 dans le second nombre, ce sous-motif reconnaît dans ce cas "0x".

2.5 AVERTISSEMENT concernant \1 et $1

Quelques personnes ont l'habitude d'écrire des choses comme :

```
$pattern =~ s/(\W)/\\\1/g;
```

Dans la partie droite d'une substitution, ce sont des vieilleries pour ne pas choquer les fanas de `sed` mais ce sont de mauvaises habitudes. Parce que du point du vue Perl, la partie droite d'un `s///` est considérée comme une chaîne entre guillemets. `\1` dans une chaîne entre guillemets signifie normalement control-A. Le sens Unix habituel de `\1` n'est repris que dans `s///`. Par contre, si vous prenez cette mauvaise habitude, vous serez très perturbé si vous ajoutez le modificateur `/e` :

```
s/(\d+)/ \1 + 1 /eg;        # provoque un "warning" avec -w
```

ou si vous essayez :

```
s/(\d+)/\1000/;
```

Vous ne pouvez lever l'ambiguïté en écrivant `\{1}000` alors que c'est possible grâce à `${1}000`. En fait, il ne faut pas confondre l'opération d'interpolation et l'opération de reconnaissance d'une référence. Ce sont deux choses bien différents dans la partie *gauche* de `s///`.

2.6 Répétition de motifs de reconnaissance de longueur nulle

AVERTISSEMENT: ce qui suit est difficile. Cette section nécessiterait une réécriture.

Les expressions rationnelles fournissent un langage de programmation concis et puissant. Comme avec la plupart des autres outils puissants, ce pouvoir peut faire des ravages.

Un "abus de pouvoir" fréquent découle de la possibilité de créer des boucles sans fin avec des expressions rationnelles aussi innocentes que :

```
'foo' =~ m{ ( o? )* }x;
```

`o?` peut être reconnu au début de `'foo'` et, puisque la position dans la chaîne n'est pas modifiée par la reconnaissance, `o?` sera reconnu indéfiniment à cause du modificateur `*`. L'utilisation du modificateur `//g` est un autre moyen courant d'obtenir de telle cycle :

```
@matches = ( 'foo' =~ m{ o? }xg );
```

ou

```
print "match: <$&>\n" while 'foo' =~ m{ o? }xg;
```

ou encore dans la boucle implicte de split().

En revanche, on sait depuis longtemps que de nombreuses tâches de programmation peuvent vraiment se simplifier grâce à la reconnaissance répétée de sous-expressions éventuellement de longueur nulle. En voici un exemple simple:

```
@chars = split //, $string;           # // n'est pas magique pour split
($whitewashed = $string) =~ s/()/ /g; # les parentheses supprime la magie de s// /
```

Donc Perl autorise la construction `/()/` qui *rompt de force la boucle infinie*. Les règles pour les boucles de bas niveau obtenues par les modificateurs gourmands `*+{}` sont différentes de celles de haut niveau induites par exemples par `/g` ou par l'opérateur split().

Les boucles de bas niveau sont interrompues lorsqu'on détecte la répétition d'une expression reconnaissant une sous-chaîne de longueur nulle. Donc

```
m{ (?: LONGUEUR_NON_NULLE| LONGUEUR_NULLE| )* }x;
```

est en fait équivalent à

```
m{    (?: LONGUEUR_NON_NULLE| )*
    |
      (?: LONGUEUR_NULLE| )?
 }x;
```

Les boucles de haut niveau se souviennent entre les itérations que la dernière chaîne reconnue était de longueur nulle. Pour briser la boucle infinie, une chaîne reconnue ne peut pas être de longueur nulle si elle l'était la fois précédente. Cette interdiction interfère avec le retour arrière (voir Retour arrière (§2.3)) et dans ce cas, la *seconde meilleure* chaîne est choisie si la *meilleure* est de longueur nulle.

Par exemple :

```
$_ = 'bar';
s/\w??/<$&>/g;
```

produit `<><b><><a><><r><>`. À chaque position dans la chaîne, la meilleur chaîne reconnue par le sobre `??` est la chaîne de longueur nulle et la seconde meilleure chaîne est celle reconnue par `\w`. Donc les reconnaissances de longueur nulle alternent avec celles d'un caractère de long.

De manière similaire, pour des `m/()/g` répétés, la seconde meilleure reconnaissance est un cran plus loin dans la chaîne.

La mémorisation de l'état *précédente reconnaissance de longueur nulle* est liée à chaque chaîne explorée. De plus, elle est réinitialisée à chaque affectation de pos(). Les reconnaissances de longueur nulle à la fin de la précédente reconnaissance sont ignorées durant un `split`.

2.7 Combinaison d'expressions rationnelles

Dans une expression rationnelle, chaque composant élémentaire tel que décrit précédemment (comme `ab` ou `\Z`) peut reconnaître au plus une sous-chaîne à une position donnée de la chaîne examinée. Par contre, dans une expression rationnelle typique, ces composants élémentaires sont combinés entre eux pour faire des expressions rationnelles plus complexes en utilisant les opérateurs de combinaison `ST`, `S|T`, `S*`, etc. (dans ces exemples `S` et `T` sont des expressions rationnelles).

De telles combinaisons peuvent inclure des alternatives, amenant ainsi à un problème de choix : lorsqu'on applique l'expression rationnelle `a|ab` à la chaîne `"abc"`, est-ce `"a"` ou `"ab"` qui est reconnu ? Un moyen de décrire le choix effectué passe par le concept de retour arrière (voir Retour arrière (§2.3)). Par contre, cette description est de trop bas niveau et vous amène à penser en termes d'une implémentation particulière.

Une autre manière de décrire les choses commence par les notions de "meilleur"/"pire". Toutes les sous-chaînes qui peuvent être reconnues par une expression rationnelle sont triées de la "meilleure" à la "pire" et c'est la meilleure qui est choisie. Cela remplace la question "Qu'est-ce qui est choisi ?" par la question "Quelle la meilleure reconnaissance et quelle est la pire ?".

Pour la plupart des expressions élémentaires, la question ne se pose pas puisqu'au plus une sous-chaîne peut être reconnue à un emplacement donné. Cette section décrit la notion de meilleure/pire pour les opérateurs de combinaison. Dans les descriptions ci-dessous, `S` et `T` désignent des sous-expressions rationnelles.

`ST`

Soit deux correspondances possibles : `AB` et `A'B'`. `A` et `A'` sont deux sous-chaînes qui peuvent être reconnues par `S`. `B` et `B'` sont deux sous-chaînes qui peuvent être reconnues par `T`.

Si `A` est une meilleure correspondance pour `S` que `A'` alors `AB` est une meilleure correspondance pour `ST` que `A'B'`.

Si `A` et `A'` sont similaires alors `AB` est une meilleure correspondance pour `ST` que `A'B'` si `B` est une meilleure correspondance pour `T` que `B'`.

`S|T`

Lorsque `S` peut être reconnu, c'est une meilleure correspondance qui si seul `T` peut correspondre.

L'ordre entre deux correspondances pour `S` est le même que pour `S` seul. Et c'est la même chose pour deux correspondances pour `T`.

`S{COMPTEUR}`

Correspond à `SSS...S` (répété autant que nécessaire).

`S{min,max}`

Correspond à `S{max}|S{max-1}|...|S{min+1}|S{min}`.

`S?`, `S*`, `S+`

Pareil que `S{0,1}`, `S{0,INFINI}` et `S{1,INFINI}` respectivement.

`S??`, `S*?`, `S+?`

Pareil que `S{0,1}?`, `S{0,INFINI}?` et `S{1,INFINI}?` respectivement.

`(?>S)`

Reconnaît la meilleure correspondance pour `S` et uniquement celle-là.

`(?=S)`, `(?<=S)`

Seul la meilleure correspondance pour `S` ets utilisé. (Cela n'a d'importance que si `S` est mémorisé et utilisé ensuite via une référence arrière)

`(?!S)`, `(?<!S)`

Pour ces constructions, il n'y a pas d'ordre à décrire puisque seul compte le fait que `S` est ou non reconnu.

`(??{ EXPR })`

L'ordre est le même que celui de l'expression rationnelle qui est le résultat de EXPR.

`(?(condition)motif-oui|motif-non)`

Souvenez-vous que le choix du motif à reconnaître (entre `motif-oui` et `motif-non`) est déjà fait. L'ordre des reconnaissances est le même que celui du motif retenu.

Les principes précédents décrivent l'ordre des reconnaissances *à une position donnée*. Une règle supplémentaire est nécessaire pour comprendre comment est déterminée la reconnaissance pour une expression rationnelle complète : une reconnaissance à une position donnée est toujours meilleure qu'une reconnaissance à une position plus lointaine.

2.8 Création de moteurs RE spécifiques (customisés)

La surcharge de constantes (voir *overload*) est un moyen simple d'augmenter les fonctionnalités du moteur RE (le moteur de reconnaissance d'expressions rationnelles).

Imaginons que vous voulez définir une nouvelle séquence `\Y|` qui peut être reconnue à la limite entre un espace et un autre caractère. Remarquez que `(?=\S)(?<!\S)|(?!\S)(?<=\S)` reconnaît exactement ce qu'on veut mais nous voulons remplacer cette expression compliquée par `\Y|`. Pour ce faire, nous pouvons créer un module `customre` :

```
package customre;
use overload;

sub import {
  shift;
  die "No argument to customre::import allowed" if @_;
  overload::constant 'qr' => \&convert;
}

sub invalid { die "/$_[0]/: invalid escape '\\$_[1]'"}

my %rules = ( '\\' => '\\',
              'Y|' => qr/(?=\S)(?<!\S)|(?!\S)(?<=\S)/ );
sub convert {
  my $re = shift;
  $re =~ s{
            \\ ( \\ | Y . )
          }
          { $rules{$1} or invalid($re,$1) }sgex;
  return $re;
}
```

Il vous suffit alors d'utiliser `use customre` pour utiliser cette nouvelle séquence dans les expressions rationnelles constantes, c.-à-d. celles qui ne nécessitent pas d'interpolation de variables lors de l'exécution. Telle que documentée dans *overload*, cette conversion ne fonctionne que sur les parties littérales des expressions rationnelles. Dans `\Y|$re\Y|`, la partie variable de cette expression rationnelle doit être convertie explicitement (mais uniquement si la signification spécifique de `\Y|` est nécessaire dans $re) :

```
use customre;
$re = <>;
chomp $re;
$re = customre::convert $re;
/\Y|$re\Y|/;
```

3 BUGS

Le niveau de difficuté de ce document varie de "difficile à comprendre" jusqu'à "totalement opaque". La prose divaguante et criblée de jargon est difficile à interpréter en divers endroits.

4 VOIR AUSSI

Opérateurs d'expression rationnelle in *perlop*.

Les détails sordides de l'interprétation des chaînes in *perlop*.

perlfaq6.

pos in *perlfunc*.

perllocale.

perlebcdic.

Mastering Regular Expressions par Jeffrey Friedl, publié chez O'Reilly and Associates.

5 TRADUCTION

5.1 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/.

5.2 Traducteur

Traduction initiale et mise à jour 5.6.0, 5.6.1 et 5.8.8 par Paul Gaborit <Paul.Gaborit @ enstimac.fr>.

5.3 Relecture

Régis Julié <Regis.Julie@cetelem.fr>.

6 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.