

perlretut

Table des matières

1	NAME/NOM	1
2	DESCRIPTION	2
3	Partie 1: les bases	2
3.1	Reconnaissance d'un mot simple	2
3.2	Utilisation des classes de caractères	6
3.3	Reconnaître ceci ou cela	8
3.4	Regroupement et reconnaissance hiérarchique	9
3.5	Extraire ce qui est reconnu	10
3.6	Reconnaissances répétées	12
3.7	Construction d'une expression rationnelle	15
3.8	Utilisation des expressions rationnelles en Perl	17
3.8.1	Recherche et remplacement	20
3.8.2	L'opérateur de découpage (split)	21
4	Partie 2: au-delà	21
4.1	Les plus des caractères, des chaînes et des classes de caractères	22
4.2	Compilation et stockage d'expressions rationnelles	24
4.3	Commentaires et modificateurs intégrés dans une expression rationnelle	25
4.4	Les regroupements sans mémorisation	26
4.5	Regarder en arrière et regarder en avant	26
4.6	Utilisation de sous-expressions indépendantes pour empêcher les retours arrière	27
4.7	Les expressions conditionnelles	28
4.8	Un peu de magie : exécution de code Perl dans une expression rationnelle	28
4.9	Directives (pragma) et déverminage	32
5	BUGS	33
6	VOIR AUSSI	34
7	AUTEURS ET COPYRIGHT	34
7.1	Remerciements	34
8	TRADUCTION	34
8.1	Version	34
8.2	Traducteur	34
8.3	Relecture	34
9	À propos de ce document	34

1 NAME/NOM

perlretut - Tutoriel des expressions rationnelles en Perl

2 DESCRIPTION

Remarque sur la traduction : 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".

Ce document propose un tutoriel dans le but de comprendre, créer et utiliser des expressions rationnelles en Perl. Il sert de complément à la documentation de référence sur les expressions rationnelles, le manuel *perlre*. Les expressions rationnelles font partie intégrante des opérateurs `m//`, `s///`, `qr//` et `split`, donc ce tutoriel a aussi des recoupements avec le titre Opérateurs d'expression rationnelle dans le manuel *perlop* et le titre `split` dans le manuel *perlfunc*.

Perl est largement reconnu pour ses capacités de manipulation de textes et les expressions rationnelles y sont pour beaucoup. Les expressions rationnelles en Perl permettent une flexibilité et une efficacité inconnues dans la plupart des autres langages. La maîtrise des expressions rationnelles même les plus simples vous permettra de manipuler du texte avec une surprenante facilité.

Qu'est-ce qu'une expression rationnelle ? Une expression rationnelle est tout simplement une chaîne de caractères qui décrit un motif. La notion de motif est couramment utilisée de nos jours. Par exemple, les motifs utilisés par un moteur de recherche pour trouver des pages web ou les motifs utilisés pour lister les fichiers dans un répertoire, e.g. `ls *.txt` ou `dir *.*`. En Perl, les motifs d'expressions rationnelles sont utilisés pour chercher dans des chaînes de caractères, pour extraire certaines parties d'une chaîne et pour réaliser des opérations de recherche et de remplacement.

Les expressions rationnelles ont la réputation d'être abstraite et difficile à comprendre. Les expressions rationnelles sont construites par assemblage de concepts simples tels que des conditions et des boucles qui ne sont pas plus compliqués à comprendre que les conditions `if` et les boucles `while` du langage Perl lui-même. En fait, le véritable enjeu dans l'apprentissage des expressions rationnelles réside dans la compréhension de la notation laconique utilisée pour exprimer ces concepts.

Ce tutoriel aplanit la courbe d'apprentissage en présentant les concepts des expressions rationnelles, ainsi que leur notation, un par un et accompagnés d'exemples. La première partie de ce tutoriel commence par la simple recherche de mots pour aboutir aux concepts de base des expressions rationnelles. Si vous maîtriser cette première partie, vous aurez tous les outils nécessaires pour résoudre 98% de vos besoins. La seconde partie de ce tutoriel est destinée à ceux qui sont déjà à l'aise avec les bases et qui recherche des outils plus puissants. Elle explique les opérateurs les plus avancés des expressions rationnelles ainsi que les dernières innovations de la version 5.6.0.

Remarque : pour gagner du temps, 'expression rationnelle' est parfois abrégée par `regexp` ou `regex`. `Regexp` est plus naturel (pour un anglophone) que `regex` mais est aussi plus dur à prononcer (toujours pour un anglophone). Dans la documentation Perl, on oscille entre `regexp` et `regex` ; en Perl, il y a toujours plusieurs façons d'abréger. Dans ce tutoriel en français, nous n'utiliserons que rarement `regex` (N.d.t: même si l'abréviation française donne `exprat` !).

3 Partie 1: les bases

3.1 Reconnaissance d'un mot simple

L'expression rationnelle la plus simple est un simple mot ou, plus généralement, une chaîne de caractères. Une expression rationnelle constituée d'un mot reconnaît toutes les chaînes qui contiennent ce mot :

```
"Hello World" =~ /World/; # est reconnu
```

Que signifie cette instruction perl ? "Hello World" est une simple chaîne de caractères entre guillemets. `World` est l'expression rationnelle et les `//` qui l'entourent (`/World/`) demandent à perl d'en chercher une correspondance dans une chaîne. L'opérateur `=~` associe la chaîne avec l'expression rationnelle de recherche de correspondance et produit une valeur vraie s'il y a correspondance ou faux sinon. Dans notre cas, `World` correspond au second mot dans "Hello World" et donc l'expression est vraie. De telles expressions sont pratique dans des conditions :

```
if ("Hello World" =~ /World/) {
    print "Il y a correspondance\n";
}
else {
    print "Il n'y a pas correspondance\n";
}
```

Il existe de nombreuses variations utiles sur ce thème. Le sens de la correspondance peut-être inversée en utilisant l'opérateur `!~` :

```

if ("Hello World" !~ /World/) {
    print "Il n'y a pas correspondance\n";
}
else {
    print "Il y a correspondance\n";
}

```

La chaîne littérale dans l'expression rationnelle peut être remplacée par une variable :

```

$greeting = "World";
if ("Hello World" =~ /$greeting/) {
    print "Il y a correspondance\n";
}
else {
    print "Il n'y a pas correspondance\n";
}

```

Si vous recherchez dans la variable spéciale par défaut \$_, la partie \$_ =~ peut être omise :

```

$_ = "Hello World";
if (/World/) {
    print "Il y a correspondance\n";
}
else {
    print "Il n'y a pas correspondance\n";
}

```

Et finalement, les délimiteurs par défaut // pour une recherche de correspondance peuvent être remplacés par n'importe quels autres délimiteurs en les préfixant par un 'm' :

```

"Hello World" =~ m!World!; # correspond, délimiteur '!'
"Hello World" =~ m{World}; # correspond, notez le couple '{}
"/usr/bin/perl" =~ m"/perl"; # correspond après '/usr/bin'
# '/' devient un caractère comme un autre

```

/World/, m!World! et m{World} représentent tous la même chose. Lorsque, par exemple, "" est utilisé comme délimiteurs, la barre oblique '/' devient un caractère ordinaire et peut être utilisé dans une expression rationnelle sans problème.

Regardons maintenant comment différentes expressions rationnelles peuvent ou non trouver une correspondance dans "Hello World" :

```

"Hello World" =~ /world/; # ne correspond pas
"Hello World" =~ /o W/; # correspond
"Hello World" =~ /oW/; # ne correspond pas
"Hello World" =~ /World /; # ne correspond pas

```

La première expression rationnelle world ne correspond pas car les expressions rationnelles sont sensibles à la casse. La deuxième expression rationnelle trouve une correspondance car la sous-chaîne 'oW' apparaît dans la chaîne "HelloWorld". Le caractère espace ' ' est traité comme n'importe quel autre caractère dans une expression rationnelle et il est nécessaire ici pour trouver une correspondance. L'absence du caractère espace explique la non-reconnaissance de la troisième expression rationnelle. La quatrième expression rationnelle 'World ' ne trouve pas de correspondance car il y a un espace à la fin de l'expression rationnelle et non à la fin de la chaîne. La leçon à tirer de ces exemples est qu'une expression rationnelle doit correspondre *exactement* à une partie de la chaîne pour être reconnue.

Si une expression rationnelle peut être reconnue à plusieurs endroits dans une chaîne, perl choisira toujours la correspondance au plus tôt :

```

"Hello World" =~ /o/; # correspond au 'o' dans 'Hello'
"That hat is red" =~ /hat/; # correspond au 'hat' dans 'That'

```

Il y a encore quelques points que vous devez savoir à propos de la reconnaissance de caractères. En premier lieu, tous les caractères ne peuvent pas être utilisés tels quels pour une correspondance. Quelques caractères, appelés **meta-caractères**, sont réservés pour des notations d'expressions rationnelles. Les meta-caractères sont :

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

La signification de chacun d'eux sera expliquée plus loin dans ce tutoriel. Pour l'heure, il vous suffira de savoir qu'un meta-caractère sera recherché tel quel si vous le précédez d'un backslash (une barre oblique inversée) :

```
"2+2=4" =~ /2+2/;      # pas de correspondance, + est un meta-caractère
"2+2=4" =~ /2\+2/;    # correspond, \+ est traité comme un + ordinaire
"The interval is [0,1)." =~ /[0,1)./      # c'est une erreur de syntaxe !
"The interval is [0,1)." =~ /\[0,1\)\/    # correspond
"/usr/bin/perl" =~ /\usr\bin\perl/;      # correspond
```

Dans la dernière expression rationnelle, les slash '/' sont aussi précédés d'un backslash parce que le slash est utilisé comme délimiteur de l'expression rationnelle. Par contre, cela peut aboutir au LTS (leaning toothpick syndrome) et il est donc souvent préférable de changer de délimiteur.

```
"/usr/bin/perl" =~ m!/usr/bin/perl!;      # plus facile à lire
```

Le caractère backslash '\ ' est lui-même un meta-caractère et doit donc être backslashé (précédé d'un backslash) :

```
'C:\WIN32' =~ /C:\\WIN/;      # correspond
```

En plus des meta-caractères, il y a quelques caractères ASCII qui n'ont pas d'équivalent affichable et sont donc représentés par des **séquences d'échappement**. Les plus courants sont \t pour une tabulation, \n pour un saut de ligne, \r pour un retour chariot et \a pour un beep. Si votre chaîne se présente plutôt comme une séquence d'octets quelconques, les séquences d'échappement en octal, tel que \033, ou en hexadécimal, tel que \x1B seront peut-être une représentation plus naturelle pour vos octets. Voici quelques exemples d'utilisation des séquences d'échappement :

```
"1000\t2000" =~ m(0\t2)      # correspond
"1000\n2000" =~ /0\n20/      # correspond
"1000\t2000" =~ /\000\t2/    # ne correspond pas, "0" ne "\000"
"cat"           =~ /\143\x61\x74/ # correspond, une façon bizarre d'épeler 'cat'
```

Si vous pratiquez déjà Perl, tout cela doit vous sembler familier. Des séquences similaires sont utilisées dans les chaînes entre guillemets. De fait, les expressions rationnelles en Perl sont traitées comme des chaînes entre guillemets. Cela signifie que l'on peut utiliser des variables dans les expressions rationnelles. Exactement comme pour les chaînes entre guillemets, chaque variable sera remplacée par sa valeur avant que l'expression rationnelle soit utilisée à la recherche de correspondances. Donc :

```
$foo = 'house';
'housecat' =~ /$foo/;      # correspond
'cathouse' =~ /cat$foo/;  # correspond
'housecat' =~ /${foo}cat/; # correspond
```

Jusqu'ici, ça va. Avec les connaissances qui précèdent vous pouvez déjà rechercher toutes les chaînes littérales imaginables. Voici une émulation *très simple* du programme Unix grep :

```
% cat > simple_grep
#!/usr/bin/perl
$regexp = shift;
while (<>) {
    print if /$regexp/;
}
^D

% chmod +x simple_grep
```

```
% simple_grep abba /usr/dict/words
Babbage
cabbage
cabbages
sabbath
Sabbathize
Sabbathizes
sabbatical
scabbard
scabbards
```

Ce programme est très simple à comprendre. `#!/usr/bin/perl` est le moyen standard pour invoquer perl. `$regexp = shift`; mémorise le premier argument de la ligne de commande en tant qu'expression rationnelle à utiliser et laisse le reste des arguments pour qu'ils soient traités comme des fichiers. `while (<>)` parcourt toutes les lignes de tous les fichiers. Pour chaque ligne, `print if /$regexp/`; affiche la ligne si l'expression rationnelle trouve une correspondance dans la ligne. Dans cette ligne, `print` et `/$regexp/` utilisent implicitement tous les deux la variable par défaut `$_`.

Dans toutes les expressions rationnelles précédentes, si l'expression rationnelle trouvait une correspondance n'importe où dans la chaîne, on considérerait qu'elle correspondait. Parfois, par contre, nous aimerions spécifier *l'endroit* dans la chaîne où l'expression rationnelle doit trouver une correspondance. Pour cela, nous devons utiliser les meta-caractères d'ancrage `^` et `$`. L'ancre `^` demande à correspondre au début de la chaîne et l'ancre `$` demande à correspondre à la fin de la chaîne ou juste avant le passage à la ligne avant la fin de la chaîne. Voici comment elles sont utilisées :

```
"housekeeper" =~ /keeper/;      # correspond
"housekeeper" =~ /^keeper/;     # ne correspond pas
"housekeeper" =~ /keeper$/;     # correspond
"housekeeper\n" =~ /keeper$/;   # correspond
```

La deuxième expression rationnelle ne correspond pas parce que `^` contraint `keeper` à ne correspondre qu'au début de la chaîne. Or `"housekeeper"` contient un `"keeper"` qui débute au milieu de la chaîne. La troisième expression rationnelle correspond puisque le `$` contraint `keeper` à n'être reconnue qu'à la fin de la chaîne.

Lorsque `^` et `$` sont utilisés ensemble, l'expression rationnelle doit être ancrée à la fois au début et à la fin de la chaîne, i.e., l'expression rationnelle correspond donc à la chaîne entière. Considérons :

```
"keeper" =~ /^keep$/;          # ne correspond pas
"keeper" =~ /^keeper$/;        # correspond
""          =~ /^$/;            # ^$ correspond à la chaîne vide
```

La première expression rationnelle ne peut pas correspondre puisque la chaîne contient plus que `keep`. Puisque la deuxième expression rationnelle est exactement la chaîne, elle correspond. L'utilisation combinée de `^` et de `$` force la chaîne entière à correspondre et vous donne donc un contrôle complet sur les chaînes qui correspondent et celles qui ne correspondent pas. Supposons que vous cherchez un individu nommé `bert` :

```
"dogbert" =~ /bert/;          # correspond, mais ce n'est pas ce qu'on cherche
"dilbert" =~ /^bert/;         # ne correspond pas mais...
"bertram" =~ /^bert/;         # correspond, ce n'est donc pas encore bon

"bertram" =~ /^bert$/;        # ne correspond pas, ok
"dilbert" =~ /^bert$/;        # ne correspond pas, ok
"bert"    =~ /^bert$/;        # correspond, parfait
```

Bien sûr, dans le cas d'une chaîne littérale, n'importe qui utiliserait l'opérateur d'égalité des chaînes `$stringeq'bert'` et cela serait beaucoup plus efficace. L'expression rationnelle `^...$` devient beaucoup plus pratique lorsqu'on lui ajoute la puissance des outils d'expression rationnelle que nous verrons plus bas.

3.2 Utilisation des classes de caractères

Bien que certains puissent se satisfaire des expressions rationnelles précédentes qui ne reconnaissent que des chaînes littérales, nous n'avons qu'effleuré la technologie des expressions rationnelles. Dans cette section et les suivantes nous allons présenter les concepts d'expressions rationnelles (et les meta-caractères associés) qui permettent à une expression rationnelle de représenter non seulement une seule séquence de caractères mais aussi *toute une classe* de séquences.

L'un de ces concepts et celui de **classe de caractères**. Une classe de caractères autorise un ensemble de caractères, plutôt qu'un seul caractère, à correspondre en un point particulier de l'expression rationnelle. Les classes de caractères sont entourées de crochets [...] avec l'ensemble de caractères placé à l'intérieur. Voici quelques exemples :

```
/cat/;      # reconnaît 'cat'
/[bcr]at/;  # reconnaît 'bat', 'cat', ou 'rat'
/item[0123456789]/; # reconnaît 'item0' ou ... ou 'item9'
"abc" =~ /[cab]/;  # reconnaît le 'a'
```

Dans la dernière instruction, bien que 'c' soit le premier caractère dans la classe, c'est le 'a' qui correspond car c'est le caractère le plus au début de la chaîne avec lequel l'expression rationnelle peut correspondre.

```
/[oO][uU][iI]/;      # reconnaît 'oui' sans tenir compte de la casse
                      # 'oui', 'Oui', 'OUI', etc.
```

Cette expression rationnelle réalise une tâche courante : une recherche de correspondance insensible à la casse. Perl fournit un moyen d'éviter tous ces crochets en ajoutant simplement un 'i' après l'expression rationnelle. Donc `/[oO][uU][iI]/;` peut être réécrit en `/oui/i;`. Le 'i' signifie insensible à la casse et est un exemple de **modificateur** de l'opération de reconnaissance. Nous rencontrerons d'autres modificateurs plus tard dans ce tutoriel.

Nous avons vu dans la section précédente qu'il y avait des caractères ordinaires, qui se représentaient eux-mêmes et des caractères spéciaux qui devaient être backslashés pour se représenter. C'est la même chose dans les classes de caractères mais les ensembles de caractères ordinaires et spéciaux ne sont pas les mêmes. Les caractères spéciaux dans une classe de caractères sont `-\^$.]` est spécial car il indique la fin de la classe de caractères. `$` est spécial car il indique une variable scalaire. `\` est spécial car il est utilisé pour les séquences d'échappement comme précédemment. Voici comment les caractères spéciaux `]\$\ sont utilisés :`

```
/[\]c]def/; # reconnaît ']'def' ou 'cdef'
$x = 'bcr';
/[$x]at/;   # reconnaît 'bat', 'cat', ou 'rat'
/[\$x]at/;  # reconnaît '$at' ou 'xat'
/[\\$x]at/; # reconnaît '\at', 'bat', 'cat', ou 'rat'
```

Les deux derniers exemples sont un peu complexes. Dans `[\$x]`, le backslash protège le signe dollar et donc la classe de caractères contient deux membres : `$` et `x`. Dans `[\\$x]`, le backslash est lui-même protégé et donc `$x` est traité comme une variable à laquelle on substitue sa valeur comme dans les chaînes entre guillemets.

Le caractère spécial `'-'` agit comme un opérateur d'intervalle dans une classe de caractères et donc un ensemble de caractères contigus peut être décrit comme un intervalle. Grâce aux intervalles, les classes peu maniables comme `[0123456789]` et `[abc...xyz]` deviennent très simples : `[0-9]` et `[a-z]`. Quelques exemples :

```
/item[0-9]/; # reconnaît 'item0' ou ... ou 'item9'
/[0-9bx-z]aa/; # reconnaît '0aa', ..., '9aa',
               # 'baa', 'xaa', 'yaa', or 'zaa'
/[0-9a-fA-F]/; # reconnaît un chiffre hexadécimal
/[0-9a-zA-Z_]/; # reconnaît un caractère d'un "mot",
                # comme ceux qui composent les noms en Perl
```

Si `'-'` est le premier ou le dernier des caractères dans une classe, il est traité comme un caractère ordinaire ; `[-ab]`, `[ab-]` et `[a\b]` sont équivalents.

Le caractère spécial `^` en première position d'une classe de caractères indique une **classe de caractères inverse** qui reconnaît n'importe quel caractère sauf ceux présents entre les crochets. Dans les deux cas, `[...]` et `[^...]`, il faut qu'un caractère soit reconnu sinon la reconnaissance échoue. Donc :

```

/[^a]at/; # ne reconnaît pas 'aat' ou 'at', mais reconnaît
           # tous les autres 'bat', 'cat', '0at', '%at', etc.
/[^0-9]/; # reconnaît un caractère non numérique
/[a^]at/; # reconnaît 'aat' or '^at'; ici '^' est ordinaire

```

Même `[0-9]` peut être ennuyeux à écrire plusieurs fois. Donc pour minimiser la frappe et rendre les expressions rationnelles plus lisibles, Perl propose plusieurs abréviations pour les classes les plus communes :

- `\d` est un chiffre et représente `[0-9]`
- `\s` est un caractère d'espace et représente `[\ \t\r\n\f]`
- `\w` est un caractère de mot (alphanumérique ou `_`) et représente `[0-9a-zA-Z_]`
- `\D` est la négation de `\d`; il représente n'importe quel caractère sauf un chiffre `[^0-9]`
- `\S` est la négation de `\s`; il représente n'importe quel caractère qui ne soit pas d'espace `[^\s]`
- `\W` est la négation de `\w`; il représente n'importe quel caractère qui ne soit un pas un caractère de mot `[^\w]`
- Le point `'.'` reconnaît n'importe quel caractère sauf `"\n"`

Les abréviations `\d\s\w\D\S\W` peuvent être utilisées à l'intérieur ou à l'extérieur des classes de caractères. Voici quelques exemples :

```

\d\d:\d\d:\d\d/; # reconnaît une heure au format hh:mm:ss
/[\d\s]/;        # reconnaît n'importe quel chiffre ou espace
/\w\W\w/;        # reconnaît un caractère mot, suivi d'un
                  # caractère non-mot, suivi d'un caractère mot
/..rt/;          # reconnaît deux caractères, suivis de 'rt'
/end\./;         # reconnaît 'end.'
/end[.]/;        # la même chose, reconnaît 'end.'

```

Puisque un point est un meta-caractère, il doit être backslashé pour reconnaître un point ordinaire. Puisque, par exemple, `\d` et `\w` sont des ensembles de caractères, il est incorrect de penser que `[^\d\w]` est équivalent à `[\D\W]`; en fait `[^\d\w]` est la même chose que `[^\w]`, qui est la même chose que `[\W]`. C'est l'application des lois ensemblistes de De Morgan.

Une ancre pratique dans les expressions rationnelles de base est l'**ancrage de mot** `\b`. Elle reconnaît la frontière entre un caractère mot et un caractère non-mot `\w\W` ou `\W\w`:

```

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

```

Notez que dans le dernier exemple, la fin de la chaîne est considérée comme une frontière de mot.

Vous devez vous demander pourquoi `'.'` reconnaît tous les caractères sauf `"\n"` - pourquoi pas tous les caractères ? C'est parce que le plus souvent on effectue des mises en correspondance par ligne et que nous voulons ignorer le caractère de passage à la ligne. Par exemple, bien que la chaîne `"\n"` représente une ligne, nous aimons la considérer comme vide. Par conséquent :

```

""    =~ /^$/; # est reconnue
"\n"  =~ /^$/; # est reconnue; "\n" est ignoré

""    =~ /.;/; # n'est pas reconnue; nécessite un caractère
""    =~ /^.$/; # n'est pas reconnue; nécessite un caractère
"\n"  =~ /^.$/; # n'est pas reconnue; nécessite un caractère autre que "\n"
"a"   =~ /^.$/; # est reconnue
"a\n" =~ /^.$/; # est reconnue; "\n" est ignoré

```

Ce comportement est pratique parce qu'habituellement nous voulons ignorer les passages à la ligne lorsque nous mettons en correspondance les caractères d'une ligne. Parfois, en revanche, nous voulons tenir compte des passages à la ligne. Nous pouvons aussi vouloir que les ancres `^` et `$` puissent s'ancrer au début et à la fin des lignes plutôt que simplement au début et à la fin de la chaîne. Perl nous permet de choisir entre ignorer ou tenir compte des passages à la ligne grâce aux modificateurs `//s` et `//m`. `//s` et `//m` signifient ligne simple et multi-ligne et ils déterminent si une chaîne doit être considérée comme une chaîne continue ou comme un ensemble de lignes. Les deux modificateurs agissent sur deux aspects de l'interprétation de l'expression rationnelle : 1) comment la classe de caractères `'.'` est définie et 2) où les ancres `^` et `$` peuvent s'ancrer. Voici les quatre combinaisons :

- pas de modificateurs (*//*) : comportement par défaut. `'.'` reconnaît n'importe quel caractère sauf `"\n"`. `^` correspond uniquement au début de la chaîne et `$` correspond uniquement à la fin de la chaîne ou avant le passage à la ligne avant la fin de la chaîne.
- modificateur *s* (*//s*) : traite la chaîne comme une seule longue ligne. `'.'` reconnaît tous les caractères, même `"\n"`. `^` correspond uniquement au début de la chaîne et `$` correspond uniquement à la fin de la chaîne ou avant le passage à la ligne avant la fin de la chaîne.
- modificateur *m* (*//m*) : traite la chaîne comme un ensemble de lignes. `'.'` reconnaît n'importe quel caractère sauf `"\n"`. `^` et `$` peuvent correspondre au début et à la fin de n'importe quelle ligne dans la chaîne.
- les deux modificateurs *s* et *m* (*//sm*) : traite la chaîne comme une seule longue ligne mais détecte les lignes multiples. `'.'` reconnaît tous les caractères, même `"\n"`. `^` et `$` peuvent correspondre au début et à la fin de n'importe quelle ligne dans la chaîne.

Voici des exemples d'utilisation de *//s* et *//m* :

```
$x = "There once was a girl\nWho programmed in Perl\n";

$x =~ /^Who/; # non reconnue, "Who" n'est pas en début de chaîne
$x =~ /^Who/s; # non reconnue, "Who" n'est pas en début de chaîne
$x =~ /^Who/m; # reconnue, "Who" au début de la seconde ligne
$x =~ /^Who/sm; # reconnue, "Who" au début de la seconde ligne

$x =~ /girl.Who/; # non reconnue, "." ne reconnaît pas "\n"
$x =~ /girl.Who/s; # reconnue, "." reconnaît "\n"
$x =~ /girl.Who/m; # non reconnue, "." ne reconnaît pas "\n"
$x =~ /girl.Who/sm; # non reconnue, "." ne reconnaît pas "\n"
```

La plupart du temps, le comportement par défaut est ce que nous voulons mais *//s* et *//m* sont occasionnellement très utiles. Si *//m* est utilisé, le début de la chaîne peut encore être reconnu par `\A` et la fin de la chaîne peut aussi être reconnue soit par l'ancre `\Z` (qui est reconnue à la fin de la chaîne ou juste avant le passage à la ligne, comme `$`) soit par l'ancre `\z` (qui n'est reconnue qu'à la fin de la chaîne) :

```
$x =~ /^Who/m; # reconnue, "Who" au début de la seconde ligne
$x =~ /\AWho/m; # non reconnue, "Who" n'est pas au début de la chaîne

$x =~ /girl$/m; # reconnue, "girl" à la fin de la première ligne
$x =~ /girl\Z/m; # non reconnue, "girl" n'est pas à la fin de la chaîne

$x =~ /Perl\Z/m; # reconnue, "Perl" est juste avant le passage à la ligne
# de la fin de chaîne
$x =~ /Perl\z/m; # non reconnue, "Perl" n'est pas à la fin de la chaîne
```

Nous savons maintenant comment créer des choix à travers des classes de caractères dans les expressions rationnelles. Qu'en est-il de choix entre des mots ou des chaînes de caractères ? Ce sont ces choix qui sont décrits dans la section suivante.

3.3 Reconnaître ceci ou cela

Parfois nous aimerions que notre expression rationnelle soit capable de reconnaître différents mots ou suites de caractères. Ceci s'effectue en utilisant le meta-caractère d'alternative `|`. Pour reconnaître `dog` ou `cat`, nous formons l'expression rationnelle `dog|cat`. Comme précédemment, perl essaie de reconnaître l'expression rationnelle le plus tôt possible dans la chaîne. À chaque position, perl essaie en premier de reconnaître la première branche de l'alternative, `dog`. Si `dog` n'est pas reconnu, perl essaie ensuite la branche suivante, `cat`. Si `cat` n'est pas reconnu non plus alors la mise en correspondance échoue et perl se déplace à la position suivante dans la chaîne. Quelques exemples :

```
"cats and dogs" =~ /cat|dog|bird/; # reconnaît "cat"
"cats and dogs" =~ /dog|cat|bird/; # reconnaît "cat"
```

Même si `dog` est la première branche de l'alternative dans le second exemple, `cat` est reconnu plus tôt dans la chaîne.

```
"cats" =~ /c|ca|cat|cats/; # reconnaît "c"
"cats" =~ /cats|cat|ca|c/; # reconnaît "cats"
```

Ici, toutes les branches peuvent correspondre à la première position dont la première branche reconnue est celle qui est retenue. Si certaines branches de l'alternative sont des tronçures des autres, placez les plus longues en premier pour leur donner une chance d'être reconnues.

```
"cab" =~ /a|b|c/ # reconnaît "c"
                # /a|b|c/ == /[abc]/
```

Ce dernier exemple montre que les classes de caractères sont comme des alternatives entre caractères. À une position donnée, la première branche qui permet une mise en correspondance de l'expression rationnelle est celle qui sera reconnue.

3.4 Regroupement et reconnaissance hiérarchique

Les alternatives permettent à une expression rationnelle de choisir entre différentes branches mais elles restent non satisfaisantes en elles-mêmes. Pour la simple raison que l'alternative est une expression rationnelle complète alors que parfois nous aimerions que ce ne soit qu'une partie de l'expression rationnelle. Par exemple, supposons que nous voulions reconnaître `housecat` ou `housekeeper`. L'expression rationnelle `housecat|housekeeper` fonctionne mais est inefficace puisque nous avons du taper `house` deux fois. Il serait pratique d'avoir une partie de l'expression rationnelle qui soit constante, comme `house`, et une partie qui soit une alternative, comme `cat|keeper`.

Les meta-caractères de **regroupement** `()` résolvent ce problème. Le regroupement permet à une partie d'une expression rationnelle d'être traitée comme une seule entité. Une partie d'une expression rationnelle est regroupée en la plaçant entre des parenthèses. Donc nous pouvons résoudre le problème `housecat|housekeeper` en formant l'expression rationnelle `house(cat|keeper)`. L'expression rationnelle `house(cat|keeper)` signifie cherche `house` suivi soit par `cat` soit par `keeper`. Quelques exemples :

```
/(a|b)b/;      # reconnaît 'ab' ou 'bb'
/(ac|b)b/;    # reconnaît 'acb' ou 'bb'
/^(a|b)c/;    # reconnaît 'ac' au début de la chaîne ou 'bc' n'importe où
/(a|[bc])d/;  # reconnaît 'ad', 'bd', ou 'cd'

/house(cat|)/; # reconnaît soit 'housecat' soit 'house'
/house(cat(s|)|)/; # reconnaît soit 'housecats' soit 'housecat' soit
                  # 'house'. Les groupes peuvent être imbriqués.

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

Les alternatives se comportent de la même manière, qu'elles soient dans ou hors d'un groupe : à une position donnée, c'est la branche la plus à gauche qui est choisie tant qu'elle permet la reconnaissance de l'expression rationnelle. Donc dans notre dernier exemple, à la première position de la chaîne, "20" est reconnu par la deuxième branche de l'alternative mais il ne reste rien pour être reconnu par les deux chiffres suivants `\d\d`. Alors perl essaie la branche suivante qui est la branche vide et ça marche puisque "20" est composé de deux chiffres.

Le processus d'essai d'une branche pour voir si elle convient puis d'une autre sinon est appelé **retour arrière** (**backtracking** en anglais). Les termes 'retour arrière' viennent de l'idée que la reconnaissance d'une expression rationnelle est comme une marche en forêt. La reconnaissance de l'expression rationnelle est comme l'arrivée à destination. Il y a plusieurs points de départ possibles, un pour chaque position dans la chaîne et chacun d'eux est essayé dans l'ordre, de gauche à droite. À partir de chaque point de départ, il y a plusieurs chemins dont certains sont des impasses et d'autres vous amènent à destination. Lorsque vous marchez sur un chemin et qu'il aboutit à une impasse, vous devez retourner en arrière pour essayer un autre chemin. Vous êtes persévérant et vous ne vous déclarez vaincu que lorsque vous avez essayé tous les chemins à partir de tous les points de départ sans aboutir à destination. Pour être plus concret, voici une analyse pas à pas de ce que perl fait lorsqu'il essaie de reconnaître l'expression rationnelle :

```
"abcde" =~ /(abd|abc)(df|d|de)/;
```

1. 1

On démarre avec la première lettre de la chaîne 'a'.

2. 2

On essaie la première branche de la première alternative, le groupe 'abd'.

3. 3

On reconnaît un 'a' suivi d'un 'b'. Jusqu'ici, ça va.

4. 4

'd' dans l'expression rationnelle ne correspond pas au 'c' dans la chaîne - une impasse. On effectue alors un retour arrière de deux caractères et on essaie la seconde branche de la première alternative, le groupe 'abc'.

5. 5

On reconnaît un 'a' suivi d'un 'b' suivi d'un 'c'. Nous avons donc satisfait le premier regroupement. On positionne \$1 à 'abc'.

6. 6

On continue avec la seconde alternative et on choisit la première branche 'df'.

7. 7

On reconnaît le 'd'.

8. 8

'f' dans l'expression rationnelle ne correspond pas au 'e' dans la chaîne; c'est donc une impasse. Retour arrière d'un caractère et choix de la seconde branche de la seconde alternative 'd'.

9. 9

'd' est reconnu. Le second regroupement est satisfait et on positionne \$2 à 'd'.

10. 10

Nous sommes à la fin de l'expression rationnelle. Nous sommes donc arrivés à destination ! Nous avons reconnu 'abcd' dans la chaîne "abcde".

Il y a deux choses à dire sur cette analyse. Tout d'abord, la troisième alternative ('de') dans le second regroupement permet aussi une reconnaissance, mais nous nous sommes arrêtés avant d'y arriver - à une position donnée, l'alternative la plus à gauche l'emporte. Ensuite, nous avons obtenu une reconnaissance au premier caractère ('a') de la chaîne. Si la reconnaissance n'avait pas été possible à cette première position, perl se serait déplacé à la deuxième position ('b') pour essayer à nouveau une reconnaissance complète. C'est uniquement lorsque tous les chemins et toutes les positions ont été essayés sans succès que perl s'arrête et déclare fausse l'assertion `$string =~ / (abd|abc) (df|d|de) / ;`.

Même avec tout ce boulot, les expressions rationnelles restent remarquablement rapides. Pour améliorer cela, durant la phase de compilation, perl compile les expressions rationnelles en une séquence compacte de opcodes qui peut parfois tenir dans le cache du processeur. Lorsque le code est exécuté, ces opcodes peuvent alors tourner à plein régime et chercher très rapidement.

3.5 Extraire ce qui est reconnu

Les meta-caractères de regroupement () servent aussi à une fonction totalement différente : ils permettent l'extraction des parties d'une chaîne qui ont trouvé une correspondance. C'est très pratique pour trouver ce qui a été reconnu et pour le traitement de texte en général. Pour chaque groupe, la partie qui a été reconnue est stockée dans les variables spéciales \$1, \$2, etc. Elles peuvent être utilisées comme des variables ordinaires :

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

Pour l'instant, nous savons que, dans un contexte scalaire, `$time =~ /(\d\d):(\d\d):(\d\d)/` retourne une valeur vraie ou fausse. Dans un contexte de liste, en revanche, il retourne la liste de valeurs reconnues (`$1, $2, $3`). Nous pouvons donc écrire du code plus compact :

```
# extraction des heures, minutes et secondes
($heures, $minutes, $secondes) = ($time =~ /(\d\d):(\d\d):(\d\d)/);
```

Si les regroupements sont imbriqués dans l'expression rationnelle, \$1 recevra le groupe dont la parenthèse ouvrante est la plus à gauche, \$2 recevra le groupe dont la parenthèse ouvrante est la seconde la plus à gauche, etc. Par exemple, voici une expression rationnelle complexe et les variables correspondantes indiquées au-dessous :

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

donc si cette expression rationnelle est reconnue, \$2 devrait contenir soit 'cd' soit 'ef'. De manière pratique, perl assigne à \$+ la chaîne associée au plus haut numéro des \$1, \$2, ... qui a été le plus récemment affectée.

Associées avec les variables \$1, \$2, ..., on trouve les **références arrières**: \1, \2, ... Les références arrières sont simplement des variables de reconnaissance qui peuvent être utilisées à l'intérieur de l'expression rationnelle. C'est une fonctionnalité vraiment sympathique - ce qui est reconnu plus tard dans une expression rationnelle peut dépendre de ce qui a été reconnu plus tôt dans l'expression rationnelle. Supposons que nous voulons chercher les mots doublés dans un texte comme 'the the'. L'expression rationnelle suivante trouve tous les mots de trois lettres doublés avec un espace entre les deux :

```
/(\w\w\w)\s\1/;
```

Le regroupement affecte une valeur à \1 et donc la même séquence de 3 lettres est utilisée pour les deux parties. Voici quelques mots avec des parties répétées :

```
% simple_grep '^(\w\w\w\w|\w\w\w|\w\w|\w)\s\1$' /usr/dict/words
beriberi
booboo
coco
mama
murmur
papa
```

L'expression rationnelle commence par un regroupement qui considère d'abord les combinaisons de 4 lettres, puis de 3 lettres, etc. et utilise ensuite \1 pour chercher une répétition. Bien que \$1 et \1 représente la même chose, faites attention à n'utiliser les variables \$1, \$2, ... qu'à l'extérieur d'une expression rationnelle et à n'utiliser les références arrières \1, \2, ... qu'à l'intérieur d'une expression rationnelle; ne pas y prêter attention peut amener à des résultats surprenants et/ou indéfinis.

En plus de ce qui a été reconnu, Perl 5.6.0 fournit aussi la position de ce qui a été reconnu grâce aux tableaux @- et @+. \$-[0] est la position du début de l'ensemble de la reconnaissance et \$+[0] est la position de la fin. De manière similaire, \$-[n] est la position du début du groupe \$n et \$+[n] est la position de sa fin. Si \$n est indéfini alors \$-[n] et \$+[n] le sont aussi. Donc le code :

```
$x = "Mmm...donut, thought Homer";
$x =~ /^(Mmm|Yech)\.\.\.(donut|peas)/; # reconnu
foreach $expr (1..$#-) {
    print "Match $expr: '${$expr}' at position ($-[$expr],$+[$expr])\n";
}
```

affiche :

```
Match 1: 'Mmm' at position (0,3)
Match 2: 'donut' at position (6,11)
```

Même s'il n'y a aucun regroupement dans l'expression rationnelle, il est encore possible de retrouver exactement ce qui a été reconnu dans la chaîne. Si vous les utilisez, perl donnera pour valeur à \$' la partie de la chaîne qui est avant ce qui a été reconnu, à \$& la partie de la chaîne qui a été reconnue et à \$' la partie de la chaîne qui est après ce qui a été reconnu. Un exemple :

```
$x = "the cat caught the mouse";
$x =~ /cat/; # $' = 'the ', $& = 'cat', $' = ' caught the mouse'
$x =~ /the/; # $' = '', $& = 'the', $' = ' cat caught the mouse'
```

Lors de la seconde mise en correspondance, \$' = "" parce que l'expression rationnelle est reconnue au premier caractère dans la chaîne et elle ne voit donc jamais le second 'the'. Il est important de noter que l'utilisation de \$' et \$' ralentissent un peu le processus de reconnaissance des expressions rationnelles et que l'utilisation de \$& le ralentit aussi, mais un peu moins parce que si ces variables sont utilisées une fois pour une expression rationnelle, elles sont alors calculées pour **toutes** les expressions rationnelles du programme. Donc si les performances font parties des buts dans votre projet, elles doivent être évitées. Préférez alors l'utilisation de @- et de @+ :

```
$' est la même chose que substr( $x, 0, $-[0] )
$& est la même chose que substr( $x, $-[0], $+[0]-$-[0] )
$' est la même chose que substr( $x, $+[0] )
```


ici est que lorsque qu'il y a plusieurs éléments dans une expression rationnelle, c'est le quantificateur le plus à gauche, s'il existe, qui est servi en premier et qui laisse le minimum au reste de l'expression rationnelle. Donc dans notre exemple, c'est le premier quantificateur `.*` qui consomme la plus grande partie de la chaîne alors que le second quantificateur `.*` obtient la chaîne vide. Les quantificateurs qui consomment autant que possible sont qualifiés de **maximaux** ou de **gourmands**.

Lorsque une expression rationnelle peut être mise en correspondance avec une chaîne de différentes façons, nous pouvons utiliser les principes suivants pour prédire la manière dont elle sera reconnue :

- Principe 0: Tout d'abord, une expression rationnelle sera toujours reconnue à la position la plus à gauche possible dans la chaîne.
- Principe 1: Dans un choix `a|b|c...`, c'est la branche la plus à gauche permettant une reconnaissance de l'ensemble de l'expression rationnelle qui sera choisie en premier.
- Principe 2: Les quantificateurs gourmands comme `?`, `*`, `+` et `{n,m}` consomme la plus grande partie possible de la chaîne tant qu'ils permettent encore de reconnaître l'ensemble de l'expression rationnelle.
- Principe 3: s'il existe plusieurs éléments dans une expression rationnelle, le quantificateur gourmand le plus à gauche, s'il existe, sera mis en correspondance avec la partie de la chaîne la plus longue possible tout en gardant possible la reconnaissance de toute l'expression rationnelle. Le quantificateur gourmand suivant, s'il existe, sera mis en correspondance avec la partie la plus longue possible de ce qui reste de la chaîne tout en gardant possible la reconnaissance de toute l'expression rationnelle. Et ainsi de suite, jusqu'à la reconnaissance complète de l'expression rationnelle.

Comme vu précédemment, le Principe 0 est prioritaire sur tous les autres - l'expression rationnelle est reconnue le plus tôt possible en utilisant les autres principes pour déterminer comment l'expression rationnelle est reconnue à cette position la plus à gauche.

Voici des exemples qui montrent l'application de ces principes :

```
$x = "The programming republic of Perl";
$x =~ /^(.+)(e|r)(.*)$/; # est reconnue avec
                        # $1 = 'The programming republic of Pe'
                        # $2 = 'r'
                        # $3 = 'l'
```

L'expression rationnelle est reconnue à la position la plus tôt, 'T'. On pourrait penser que le `e` le plus à gauche de l'alternative devrait être reconnu mais le `r` produit une chaîne plus longue pour le premier quantificateur.

```
$x =~ /(m{1,2})(.*)$/; # est reconnue avec
                      # $1 = 'mm'
                      # $2 = 'ing republic of Perl'
```

Ici, la mise en correspondance au plus tôt se fait au premier 'm' de `programming.m{1,2}` est le premier quantificateur et donc il cherche la correspondance maximale `mm`.

```
$x =~ /.*(m{1,2})(.*)$/; # est reconnue avec
                        # $1 = 'm'
                        # $2 = 'ing republic of Perl'
```

Ici, l'expression rationnelle est mise en correspondance dès le début de la chaîne. Le premier quantificateur `.*` consomme le plus de caractères possibles en ne laissant qu'un seul 'm' pour le second quantificateur `m{1,2}`.

```
$x =~ /(.(?)(m{1,2})(.*)$/; # est reconnue avec
                          # $1 = 'a'
                          # $2 = 'mm'
                          # $3 = 'ing republic of Perl'
```

Ici, `.?` consomme le maximum de caractères (un caractère) à la position la plus à gauche possible dans la chaîne, le 'a' dans `programming`, en laissant l'opportunité à `m{1,2}` de correspondre aux deux `m`. Finalement :

```
"aXXXb" =~ /(X*)/; # est reconnue avec $1 = ''
```

parce qu'il peut reconnaître zéro 'X' au tout début de la chaîne. Si vous voulez réellement reconnaître au moins un 'X', utilisez `X+` au lieu de `X*`.

Parfois la gourmandise n'est pas une bonne chose. Dans ce cas, nous aimerions des quantificateurs qui reconnaissent une partie *minimale* de la chaîne plutôt que *maximale*. Pour cela, Larry Wall a créé les quantificateurs **minimaux** ou quantificateurs **sobres** : `??`, `*?`, `+?` et `{ }?`. Ce sont les quantificateurs habituels auxquels on ajoute un `?`. Ils ont la sémantique suivante :

- `a??` = reconnaît un 'a' 0 ou 1 fois. Essaie 0 d'abord puis 1 ensuite.
- `a*?` = reconnaît zéro 'a' ou plus mais un minimum de fois.
- `a+?` = reconnaît un 'a' ou plus mais un minimum de fois.
- `a{n,m}?` = reconnaît entre `n` et `m` 'a' mais un minimum de fois.
- `a{n,}?` = reconnaît au moins `n` 'a' mais un minimum de fois.
- `a{n}?` = reconnaît exactement `n` 'a'. C'est équivalent à `a{n}`. Ce n'est donc que pour rendre la notation cohérente.

Reprenons les exemples précédents mais avec des quantificateurs minimaux :

```
$x = "The programming republic of Perl";
$x =~ /^(.+?) (e|r) (.*)$/; # est reconnue avec
# $1 = 'Th'
# $2 = 'e'
# $3 = ' programming republic of Perl'
```

La chaîne minimale permettant à la fois de reconnaître le début de chaîne `^` et l'alternative est `Th` avec l'alternative `e|r` qui reconnaît `e`. Le second quantificateur est libre de consommer tout le reste de la chaîne.

```
$x =~ /(m{1,2}?) (.*)$/; # est reconnue avec
# $1 = 'm'
# $2 = 'ming republic of Perl'
```

La première position à partir de laquelle cette expression rationnelle peut être reconnue et le premier 'm' de `programming`. À cette position, le sobre `m{1,2}?` reconnaît juste un 'm'. Ensuite, bien que le second quantificateur `.*` préfère reconnaître un minimum de caractères, il est contraint par l'ancre de fin de chaîne `$` à reconnaître tout le reste de la chaîne.

```
$x =~ /(.*) (m{1,2}?) (.*)$/; # est reconnue avec
# $1 = 'The progra'
# $2 = 'm'
# $3 = 'ming republic of Perl'
```

Dans cette expression rationnelle, vous espérez peut-être que le premier quantificateur minimal `.*` soit mis en correspondance avec la chaîne vide puisqu'il n'est pas contraint à s'ancrer en début de chaîne par `^`. Mais ici le principe 0 s'applique. Puisqu'il est possible de reconnaître l'ensemble de l'expression rationnelle en s'ancrant au début de la chaîne, ce sera cet ancrage qui sera choisi. Donc le premier quantificateur doit reconnaître tout jusqu'au premier `m` et le troisième quantificateur reconnaît le reste de la chaîne.

```
$x =~ /(.?) (m{1,2}) (.*)$/; # est reconnue avec
# $1 = 'a'
# $2 = 'mm'
# $3 = 'ing republic of Perl'
```

Comme dans l'expression rationnelle précédente, le premier quantificateur peut correspondre au plus tôt sur le 'a', c'est donc ce qui est retenu. Le second quantificateur est gourmand donc il reconnaît `mm` et le troisième reconnaît le reste de la chaîne.

Nous pouvons modifier le principe 3 précédent pour prendre en compte les quantificateurs sobres :

- Principe 3: s'il existe plusieurs éléments dans une expression rationnelle, le quantificateur gourmand (ou sobre) le plus à gauche, s'il existe, sera mis en correspondance avec la partie de la chaîne la plus longue (ou la plus courte) possible tout en gardant possible la reconnaissance de toute l'expression rationnelle. Le quantificateur gourmand (ou sobre) suivant, s'il existe, sera mis en correspondance avec la partie la plus longue (ou la plus courte) possible de ce qui reste de la chaîne tout en gardant possible la reconnaissance de toute l'expression rationnelle. Et ainsi de suite, jusqu'à la reconnaissance complète de l'expression rationnelle.

Comme avec les alternatives, les quantificateurs sont susceptibles de déclencher des retours arrière. Voici l'analyse pas à pas d'un exemple :

```
$x = "the cat in the hat";
$x =~ /^(.*) (at) (.*)$/; # est reconnue avec
# $1 = 'the cat in the h'
# $2 = 'at'
# $3 = '' (reconnaissance de la chaîne vide)
```

1. 1
On démarre avec la première lettre de la chaîne 't'.
2. 2
Le premier quantificateur '.' commence par reconnaître l'ensemble de la chaîne 'the cat in the hat'.
3. 3
Le 'a' de l'élément 'at' ne peut pas être mis en correspondance avec la fin de la chaîne. Retour arrière d'un caractère.
4. 4
Le 'a' de l'élément 'at' ne peut pas être mis en correspondance avec la dernière lettre de la chaîne 't'. Donc à nouveau un retour arrière d'un caractère.
5. 5
Maintenant nous pouvons reconnaître le 'a' et le 't'.
6. 6
On continue avec le troisième élément '.'. Puisque nous sommes à la fin de la chaîne, '.' ne peut donc reconnaître que 0 caractère. On lui affecte donc la chaîne vide.
7. 7
C'est fini.

La plupart du temps, tous ces aller-retours ont lieu très rapidement et la recherche est rapide. Par contre, il existe quelques expressions rationnelles pathologiques dont le temps d'exécution augmente exponentiellement avec la taille de la chaîne. Une structure typique est de la forme :

```
/(a|b+)*;/
```

Le problème est l'imbrication de deux quantificateurs indéterminés. Il y a de nombreuses manières de partitionner une chaîne de longueur n entre le $+$ et le $*$: une répétition avec b^+ de longueur n , deux répétitions avec le premier b^+ de longueur k et le second de longueur $n-k$, m répétitions dont la somme des longueurs est égale à n , etc. En fait, le nombre de manières différentes de partitionner une chaîne est exponentielle en fonction de sa longueur. Une expression rationnelle peut être chanceuse et être reconnue très tôt mais s'il n'y a pas de reconnaissance possible, perl essaiera *toutes* les possibilités avant de conclure à l'échec. Donc, soyez prudents lorsque vous imbriquez des $*$, des $\{n, m\}$ et/ou des $+$. Le livre *Mastering regular expressions* par Jeffrey Friedl donne de très bonnes explications sur ce problème en particulier et sur tous les autres problèmes de performances.

3.7 Construction d'une expression rationnelle

À ce point, nous avons couvert tous les concepts de base des expressions rationnelles. Nous pouvons donc présenter un exemple plus complet d'utilisation des expressions rationnelles. Nous allons construire une expression rationnelle qui reconnaît les nombres.

La première tâche de la construction d'une expression rationnelle consiste à décider ce que nous voulons reconnaître et ce que nous voulons exclure. Dans notre cas, nous voulons reconnaître à la fois les entiers et les nombres en virgule flottante et nous voulons rejeter les chaînes qui ne sont pas des nombres.

La tâche suivante permet de découper le problème en plusieurs petits problèmes qui seront plus simplement transformés en expressions rationnelles.

Le cas le plus simple concerne les entiers. Ce sont une suite de chiffres précédée d'un éventuel signe. Les chiffres peuvent être représentés par $\backslash d^+$ et le signe peut être reconnu par $[+-]$. Donc l'expression rationnelle pour les entiers est :

```
/[+-]?\d+;/ # reconnaît les entiers
```

Un nombre en virgule flottante contient potentiellement un signe, une partie entière, un séparateur décimal (un point), une partie décimale et un exposant. Plusieurs de ces parties sont optionnelles donc nous devons vérifier les différentes possibilités. Les nombres en virgule flottante bien formés contiennent entre autres 123., 0.345, .34, -1e6 et 25.4E-72. Comme pour les entiers, le signe au départ est complètement optionnel et peut être reconnu par $[+-]?$. Nous pouvons voir que s'il n'a pas d'exposant, un nombre en virgule flottante doit contenir un séparateur décimal ou alors c'est un entier. Nous pourrions être tentés d'utiliser $\backslash d^* \backslash . \backslash d^*$ mais cela pourrait aussi reconnaître un séparateur décimal seul (qui n'est pas un nombre). Donc, les trois cas de nombres sans exposant sont :

```
/[+-]?\d+\./; # 1., 321., etc.
/[+-]?\.\d+;/; # .1, .234, etc.
/[+-]?\d+\.\d+;/; # 1.0, 30.56, etc.
```

On peut combiner cela en une seule expression rationnelle :

```
/[+-]?(\d+\.\d+|\d+\.\|\.\d+)/; # virgule flottante, sans exposant
```

Dans ce choix, il est important de placer '`\d+\.\d+'` avant '`\d+\.`'. Si '`\d+\.`' était en premier, l'expression rationnelle pourrait être reconnue en ignorant la partie décimale du nombre.

Considérons maintenant les nombres en virgule flottante avec exposant. Le point clé ici est que les nombres avec séparateur décimal *ainsi* que les entiers sont autorisés devant un exposant. Ensuite la reconnaissance de l'exposant, comme celle du signe, est indépendante du fait que le nombre possède ou non une partie décimale. Elle peut donc être découplée de la mantisse. La forme de l'expression rationnelle complète devient donc claire maintenant :

```
/^(signe optionnel)(entier | mantisse)(exposant optionnel)$/;
```

L'expression est un `e` ou un `E` suivi d'un entier. Donc l'expression rationnelle de l'exposant est :

```
/[eE][+-]?\d+;/ # exposant
```

En assemblant toutes les parties ensemble nous obtenons l'expression rationnelle qui reconnaît les nombres :

```
/^[+-]?(\d+\.\d+|\d+\.\|\.\d+|\d+)([eE][+-]?\d+)?$/; # Ta da!
```

De longues expressions rationnelles comme celle-ci peuvent peut-être impressionner vos amis mais elles sont difficiles à déchiffrer. Dans des situations complexes comme celle-ci, le modificateur `//x` est très pratique. Il vous permet de placer des espaces et des commentaires n'importe où dans votre expression sans en changer la signification. En l'utilisant, nous pouvons réécrire notre expression sous une forme plus plaisante :

```
/^
  [+]?      # en premier, reconnaissance du signe optionnel
  (
    # ensuite reconnaissance d'un entier ou d'un
    # nombre en virgule flottante
    \d+\.\d+ # mantisse de la forme a.b
    |\d+\.   # mantisse de la forme a.
    |\.\d+   # mantisse de la forme .b
    |\d+     # entier de la forme a
  )
  ([eE][+-]?\d+)? # finalement, reconnaissance optionnelle d'un exposant
$/x;
```

Si les espaces ne sont pas pris en compte, comment inclure un caractère espace dans une telle expression rationnelle ? Il suffit de le «backslasher» '`\`' ou de le placer dans une classe de caractères `[]`. La même chose est valable pour le dièse : utilisez `\#` ou `[#]`. Par exemple, Perl autorise un espace entre le signe la mantisse ou l'entier. Nous pouvons ajouter cela dans notre expression rationnelle comme suit :

```
/^
  [+]? \ *  # en premier, reconnaissance du signe optionnel *et des espaces*
  (
    # ensuite reconnaissance d'un entier ou d'un
    # nombre en virgule flottante
    \d+\.\d+ # mantisse de la forme a.b
    |\d+\.   # mantisse de la forme a.
    |\.\d+   # mantisse de la forme .b
    |\d+     # entier de la forme a
  )
  ([eE][+-]?\d+)? # finalement, reconnaissance optionnelle d'un exposant
$/x;
```

Sous cette forme, il est plus simple de découvrir un moyen de simplifier le choix. Les branches 1, 2 et 4 du choix commencent toutes par `\d+` qu'on doit donc pouvoir factoriser :

```

/^
  [+]? \ *      # en premier, reconnaissance du signe optionnel
  (           # ensuite reconnaissance d'un entier ou d'un
              # nombre en virgule flottante
    \d+       # on commence par a ...
    (
      \.\d*   # mantisse de la forme a. ou a.b
    )?       # ? pour les entiers de la forme a
    |\.\d+   # mantisse de la forme .b
  )
  ([eE][+]? \d+)? # finalement, reconnaissance optionnelle d'un exposant
$/x;

```

ou écrit dans sa forme compacte :

```

/^ [+]? \ * (\d+ (\.\d*)? |\.\d+) ([eE] [+]? \d+)? $/;

```

C'est notre expression rationnelle finale. Pour récapituler, nous avons construit notre expression rationnelle :

- en spécifiant la tâche en détail,
- en découpant le problème en plus petites parties,
- en transformant les petites parties en expressions rationnelles,
- en combinant les expressions rationnelles,
- et en optimisant l'expression rationnelle combinée finale.

On peut faire le parallèle avec les différentes étapes de l'écriture d'un programme. C'est normal puisque les expressions rationnelles sont des programmes écrits dans un petit langage informatique de spécification de motifs.

3.8 Utilisation des expressions rationnelles en Perl

Pour terminer cette première partie, nous allons examiner brièvement comment les expressions rationnelles sont utilisées dans un programme Perl. Comment s'intègrent-elles à la syntaxe Perl ?

Nous avons déjà parlé de l'opérateur de recherche de correspondances dans sa forme par défaut `/regexp/` ou avec des délimiteurs arbitraires `m!regexp!`. Nous avons utilisé l'opérateur de mise en correspondance `=~` ainsi que sa négation `!~` pour rechercher les correspondances. Associé avec l'opérateur de recherche de correspondances, nous avons présenté les modificateurs permettant de traiter de simples lignes `//s`, des multi-lignes `//m`, des expressions insensibles à la casse `//i` et des expressions étendues `//x`.

Il y a encore quelques points que vous devez connaître à propos des opérateurs de mise en correspondance. Nous avons montré que les variables étaient substituées avant l'évaluation de l'expression rationnelle :

```

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

```

Cela affichera toutes les lignes contenant le mot `Seuss`. Par contre, ce n'est pas aussi efficace qu'il y paraît car perl doit réévaluer `$pattern` à chaque passage dans la boucle. Si `$pattern` ne doit pas changer durant la vie du script, nous pouvons ajouter le modificateur `//o` qui demande à perl de n'effectuer la substitution de variables qu'une seule fois :

```

#!/usr/bin/perl
# grep simple amélioré
$regexp = shift;
while (<>) {
    print if /$regexp/o; # cela va plus vite...
}

```

Si vous changez `$pattern` après la première substitution, perl l'ignorera. Si vous voulez que perl n'effectue aucune substitution, utilisez le délimiteur spécial `m"` :

```
@pattern = ('Seuss');
while (<>) {
    print if m'@pattern'; # correspond littéralement avec '$pattern',
                        # pas avec 'Seuss'
}
```

`m''` agit exactement comme les apostrophes sur les expressions rationnelles ; tout autre délimiteur agit comme des guillemets. Si l'expression rationnelle s'évalue à la chaîne vide, la dernière expression rationnelle *utilisée avec succès* sera utilisée à la place. Donc :

```
"dog" =~ /d/; # 'd' correspond
"dogbert" =~ //; # mise en correspondance par
                # l'expression rationnelle 'd' précédente
```

Le deux modificateurs restant (`//g` et `//c`) concernent les recherches multiples. Le modificateur `//g` signifie recherche globale et autorise l'opérateur de mise en correspondance à correspondre autant de fois que possible. Dans un contexte scalaire, des invocations successives d'une expression rationnelle modifiée par `//g` appliquée à une même chaîne passeront d'une correspondance à une autre, en se souvenant de la position atteinte dans la chaîne explorée. Vous pouvez consulter ou modifier cette position grâce à la fonction `pos()`.

L'utilisation de `//g` est montrée dans l'exemple suivant. Supposons une chaîne constituée de mots séparés par des espaces. Si nous connaissons à l'avance le nombre de mots, nous pouvons extraire les mots en utilisant les regroupements :

```
$x = "cat dog house"; # 3 mots
$x =~ /^s*(\w+)\s+(\w+)\s+(\w+)\s*$/; # correspond avec
                                        # $1 = 'cat'
                                        # $2 = 'dog'
                                        # $3 = 'house'
```

Mais comment faire si le nombre de mots est indéterminé ? C'est pour ce genre de tâche qu'on a créé `//g`. Pour extraire tous les mots, on utilise l'expression rationnelle simple `(\w+)` et on boucle sur toutes les correspondances possibles grâce à `/(\w+)/g`:

```
while ($x =~ /(\w+)/g) {
    print "Le mot $1 se termine à la position ", pos $x, "\n";
}
```

qui affiche

```
Le mot cat se termine à la position 3
Le mot dog se termine à la position 7
Le mot house se termine à la position 13
```

Un échec de la mise en correspondance ou un changement de chaîne cible réinitialise la position. Si vous ne voulez pas que la position soit réinitialisée après un échec, ajoutez le modificateur `//c` comme dans `/regexp/gc`. La position courante dans la chaîne est associée à la chaîne elle-même et non à l'expression rationnelle. Cela signifie que des chaînes différentes ont des positions différentes et que ces positions peuvent être modifiées indépendamment.

Dans un contexte de liste, `//g` retourne la liste de tous les groupes mis en correspondance ou, s'il n'y a pas de groupes, la liste de toutes les reconnaissances de l'expression rationnelle entière. Donc si nous ne voulons que les mots, nous pouvons faire :

```
@words = ($x =~ /(\w+)/g); # correspond avec
                        # $word[0] = 'cat'
                        # $word[1] = 'dog'
                        # $word[2] = 'house'
```

Étroitement associé avec le modificateur `//g`, il existe l'ancre `\G`. L'ancre `\G` est mis en correspondance avec le point atteint lors d'une reconnaissance précédente par `//g`. `\G` permet de faire de la reconnaissance dépendante du contexte :

```

$metric = 1; # utilisation des unités métriques
...
$x = <FILE>; # lecture des mesures
$x =~ /^(+)?\d+\s*/g; # obtention de la valeur
$weight = $1;
if ($metric) { # vérification d'erreur
    print "Units error!" unless $x =~ /\Gkg\./g;
}
else {
    print "Units error!" unless $x =~ /\Glbs\./g;
}
$x =~ /\G\s+(widget|sprocket)/g; # suite du traitement

```

La combinaison de `//g` et de `\G` permet le traitement pas à pas d'une chaîne et l'utilisation de Perl pour déterminer la suite du traitement. Actuellement, l'ancre `\G` n'est réellement utilisable que si elle est utilisée en début de motif.

`\G` est aussi pratique lors du traitement par des expressions rationnelles d'enregistrement à taille fixe. Supposons une séquence d'ADN, encodée comme une suite de lettres ATCGTTGAAT... et que vous voulez trouver tous les codons du type TGA. Dans une séquence, les codons sont des séquences de 3 lettres. Nous pouvons donc considérer une chaîne d'ADN comme une séquence d'enregistrements de 3 lettres. L'expression rationnelle naïve :

```

# C'est "ATC GTT GAA TGC AAA TGA CAT GAC"
$dna = "ATCGTTGAATGCAAATGACATGAC";
$dna =~ /TGA/;

```

ne marche pas ; elle retrouvera un TGA mais il n'y aura aucune garantie que cette correspondance soit alignée avec une limite de codon, e.g., la sous-chaîne GTTGA donnera une correspondance. Une meilleure solution est :

```

while ($dna =~ /(\w\w\w)*?TGA/g) { # remarquez le minimal *?
    print "Got a TGA stop codon at position ", pos $dna, "\n";
}

```

qui affichera :

```

Got a TGA stop codon at position 18
Got a TGA stop codon at position 23

```

La position 18 est correcte mais pas la position 23. Que s'est-il passé ?

Notre expression rationnelle fonctionne bien tant qu'il reste de bonnes correspondances. Puis l'expression rationnelle ne trouve plus de TGA bien synchronisé et réessaie après s'être décalée d'un caractère à chaque fois. Ce n'est pas ce que nous voulons. La solution est l'utilisation de `\G` pour ancrer notre expression rationnelle sur une limite de codon :

```

while ($dna =~ /\G(\w\w\w)*?TGA/g) {
    print "Got a TGA stop codon at position ", pos $dna, "\n";
}

```

qui affiche :

```

Got a TGA stop codon at position 18

```

qui est la bonne réponse. Cet exemple illustre qu'il ne suffit pas de reconnaître ce que l'on cherche, il faut aussi rejeter ce qu'on ne veut pas.

3.8.1 Recherche et remplacement

Les expressions rationnelles jouent aussi un grand rôle dans les opérations de recherche et remplacement en Perl. Une recherche/remplacement est accomplie via l'opérateur `s///`. La forme générale est `s/regexp/remplacement/modificateurs` avec l'application de tout ce que nous connaissons déjà sur les expressions rationnelles et les modificateurs. La partie `remplacement` est comme une chaîne entre guillemets de Perl qui remplacera la partie de la chaîne reconnue par l'expression rationnelle `regexp`. L'opérateur `=~` est aussi utilisé pour associer une chaîne avec `s///`. Si on veut l'appliquer à `$_`, on peut omettre `$_=~`. S'il y a correspondance, `s///` renvoie le nombre de substitutions effectuées sinon il retourne faux. Voici quelques exemples :

```
$x = "Time to feed the cat!";
$x =~ s/cat/hacker/; # $x contient "Time to feed the hacker!"
if ($x =~ s/^(Time.*hacker)!$/! now!/) {
    $more_insistent = 1;
}
$y = "'quoted words'";
$y =~ s/^(.*)'$/!$/; # suppression des apostrophes,
                      # $y contient "quoted words"
```

Dans le dernier exemple, la chaîne entière est reconnue mais seule la partie à l'intérieur des apostrophes est dans un groupe. Avec l'opérateur `s///`, les variables `$1`, `$2`, etc. sont immédiatement disponibles pour une utilisation dans l'expression de remplacement. Donc, nous utilisons `$1` pour remplacer la chaîne entre apostrophes par ce qu'elle contient. Avec le modificateur global, `s///g` cherchera et remplacera toutes les occurrences de l'expression rationnelle dans la chaîne :

```
$x = "I batted 4 for 4";
$x =~ s/4/four/; # ne fait pas tout :
                # $x contient "I batted four for 4"
$x = "I batted 4 for 4";
$x =~ s/4/four/g; # fait tout :
                 # $x contient "I batted four for four"
```

Si vous préférez `'regex'` à la place de `'regexp'` dans ce tutoriel, vous pourriez utiliser le programme suivant pour les remplacer :

```
% cat > simple_replace
#!/usr/bin/perl
$regexp = shift;
$replacement = shift;
while (<>) {
    s/$regexp/$replacement/go;
    print;
}
^D

% simple_replace regexp regex perlretut.pod
```

Dans `simple_replace` nous utilisons le modificateur `s///g` pour remplacer toutes les occurrences de l'expression rationnelle à chaque ligne et le modificateur `s///o` pour compiler l'expression rationnelle une seule fois pour toutes. Comme avec `simple_grep`, les deux instructions `print` et `s/$regexp/$replacement/go` utilisent implicitement la variable `$_`.

Un modificateur spécifique à l'opération de recherche/remplacement est le modificateur d'évaluation `s///e`. `s///e` ajoute un `eval{...}` autour de la chaîne de remplacement et le résultat de cette évaluation est substitué à la sous-chaîne mise en correspondance. `s///e` est utile si vous avez besoin de faire un traitement sur le texte à remplacer. Cet exemple compte la fréquence des lettres dans une ligne :

```
$x = "Bill the cat";
$x =~ s/(.)/$chars{$1}++;$1/eg; # Le $1 final remplace le caractère par lui-même
print "frequency of '$_' is $chars{$_}\n"
      foreach (sort {$chars{$b} <=> $chars{$a}} keys %chars);
```

qui affiche :

```

frequency of ' ' is 2
frequency of 't' is 2
frequency of 'l' is 2
frequency of 'B' is 1
frequency of 'c' is 1
frequency of 'e' is 1
frequency of 'h' is 1
frequency of 'i' is 1
frequency of 'a' is 1

```

Comme pour l'opérateur `m//`, `s///` peut utiliser d'autres délimiteurs comme `s!!!` ou `s{}{}` et même `s{}//`. Si les délimiteurs sont des apostrophes `s''` alors l'expression rationnelle et le remplacement sont traités comme des chaînes entre apostrophes et aucune interpolation de variables n'est effectuée. `s///` dans un contexte de liste retourne la même chose qu'en contexte scalaire, c'est à dire le nombre de substitutions effectuées.

3.8.2 L'opérateur de découpage (`split`)

La fonction `split` peut, elle aussi, utiliser un opérateur de mise en correspondance `m//` pour découper une chaîne. `split /regexp/, chaine, limite` découpe la chaîne en une liste de sous-chaînes et retourne cette liste. L'expression rationnelle `regexp` est utilisée pour reconnaître la séquence de caractères qui servira de séparateur lors du découpage de chaîne. La `limite`, si elle est présente, indique le nombre maximal de morceaux lors du découpage. Par exemple, pour découper une chaîne en mots, utilisez :

```

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

```

Si l'expression rationnelle vide `//` est utilisée alors l'expression rationnelle correspond partout et la chaîne est découpée en caractères individuels. Si l'expression rationnelle contient des groupes alors la liste produite contient aussi les sous-chaînes reconnues par les groupes. Par exemple :

```

$x = "/usr/bin/perl";
@dirs = split m!/!, $x; # $dirs[0] = ''
                       # $dirs[1] = 'usr'
                       # $dirs[2] = 'bin'
                       # $dirs[3] = 'perl'
@parts = split m!(/!)!, $x; # $parts[0] = ''
                            # $parts[1] = '/'
                            # $parts[2] = 'usr'
                            # $parts[3] = '/'
                            # $parts[4] = 'bin'
                            # $parts[5] = '/'
                            # $parts[6] = 'perl'

```

Puisque le premier caractère de `$X` est reconnu par l'expression rationnelle, `split` ajoute un élément initial vide à la liste. Si vous avez tout lu jusqu'ici, félicitations ! Vous possédez maintenant tous les outils de base pour utiliser les expressions rationnelles afin de résoudre de nombreux problèmes de traitement de textes. Si c'est la première fois que vous lisez ce tutoriel, vous devriez vous arrêter ici et jouer quelques temps avec les expressions rationnelles... La Partie 2 concerne des aspects plus ésotériques des expressions rationnelles et ces concepts ne sont certainement pas nécessaires au début.

4 Partie 2: au-delà

Bon, vous connaissez les bases des expressions rationnelles et vous voulez en savoir plus. Si la mise en correspondance d'une expression rationnelle est analogue à une marche en forêt alors les outils dont nous avons parlé dans la partie 1 sont la carte et le compas, des outils basiques que nous utilisons tout le temps. La plupart des outils de la partie 2 sont alors analogues à un lance fusées éclairantes ou à un téléphone satellite. On ne les utilise pas très souvent mais, en cas de besoin, ils sont irremplaçables.

Ce qui suit présente les fonctionnalités les plus avancées, les moins utilisées ou les plus ésotériques des expressions rationnelles de perl. Dans cette seconde partie, nous supposons que vous êtes à l'aise avec les outils de base pour nous concentrer sur ces nouvelles fonctionnalités.

4.1 Les plus des caractères, des chaînes et des classes de caractères

Il y a de nombreuses séquences d'échappement et classes de caractères dont nous n'avons pas encore parlé.

Il existe plusieurs séquences d'échappement qui convertissent les caractères ou les chaînes entre majuscules et minuscules. `\l` et `\u` convertissent le caractère suivant respectivement en minuscule ou en majuscule :

```
$x = "perl";
$string =~ /\u$x/; # reconnaît 'Perl' dans $string
$x = "M(rs?|s)\."; # notez le double backslash
$string =~ /\l$x/; # reconnaît 'mr.', 'mrs.' et 'ms.',
```

`\L` et `\U` convertissent une sous-chaîne entière délimitée par `\L`, `\U` ou `\E` en minuscule ou majuscule :

```
$x = "This word is in lower case:\L SHOUT\E";
$x =~ /shout/; # correspond
$x = "I STILL KEYPUNCH CARDS FOR MY 360"
$x =~ /\Ukeypunch/; # correspond
```

S'il n'y a pas de `\E`, la changement de casse a lieu jusqu'à la fin de la chaîne. Les expressions rationnelles `\L\u$word` ou `\u\L$word` convertissent le premier caractère de `$word` en majuscule et les autres caractères en minuscules.

Les caractères de contrôle peuvent être codés via `\c`. Le caractère control-z sera mis en correspondance avec `\cZ`. La séquence d'échappement `\Q...\E` protège la plupart des caractères non-alphabétiques. Par exemple :

```
$x = "\QThat !^*&%~& cat!";
$x =~ /\Q!^*&%~&\E/; # correspond
```

Les caractères `$` et `@` ne sont pas protégés donc les variables peuvent encore être interpolées.

Avec Perl 5.6.0, les expressions rationnelles peuvent gérer plus que le jeu de caractères ASCII. Perl supporte maintenant l'**Unicode**, un standard pour encoder les caractères de la plupart des langues. Unicode permet cela en autorisant un encodage des caractères sur plusieurs octets. Perl utilise l'encodage UTF-8 dans lequel les caractères ASCII sont encore encodés sur un octet mais où les caractères plus grand que `chr(127)` peuvent être codés sur deux ou plusieurs octets.

Quelles conséquences sur les expressions rationnelles ? Les utilisateurs d'expressions rationnelles n'ont pas besoin de connaître la représentation interne des chaînes de perl. Par contre, il faut qu'ils sachent 1) comment représenter les caractères Unicode dans une expression rationnelle et 2) quand une opération de mise en correspondance traitera une chaîne comme une séquence d'octets (l'ancien mode) ou comme une séquence de caractères Unicode (le nouveau mode). La réponse à la question 1) est que les caractères Unicode plus grand que `chr(127)` peuvent être représentés en utilisant la notation `\x{hex}` où `hex` est un entier hexadécimal :

```
/\x{263a}/; # correspond à l'émoticon souriant d'Unicode :)
```

Les caractères Unicode dans l'intervalle 128-255 utilisent deux chiffres hexadécimaux avec des accolades : `\x{ab}`. Remarquez la différence avec `\xab` qui est juste un octet en hexadécimal sans aucune signification Unicode.

NOTE: en Perl 5.6.0 il fallait ajouter la directive `use utf8` pour activer les fonctionnalités Unicode. Ce n'est plus nécessaire : la quasi totalité des fonctionnalités Unicode ne nécessitent plus la directive ou `pragma utf8`. (Le seul cas où cela reste nécessaire est celui où votre script Perl lui-même est écrit en Unicode encodé en UTF-8.)

Se souvenir de la séquence hexadécimale d'un caractère Unicode ou décoder les séquences hexadécimales d'un autre dans une expression rationnelle est presque aussi fun que de coder en langage machine. Un autre moyen de spécifier des caractères Unicode est d'utiliser des **caractères nommés** via la séquence d'échappement `\N{nom}`. `nom` est le nom d'un caractère Unicode comme spécifié dans le standard Unicode. Par exemple, si vous voulez représenter ou reconnaître le signe astrologique de la planète Mercure, vous pourriez utiliser :

```
use charnames ":full"; # utilise les caractères nommés
                        # avec les noms Unicode complet
$x = "abc\N{MERCURY}def";
$x =~ /\N{MERCURY}/; # correspond
```

Il est aussi possible d'utiliser les noms courts ou restreints à certains alphabets :

```

use charnames ':full';
print "\N{GREEK SMALL LETTER SIGMA} is called sigma.\n";

use charnames ":short";
print "\N{greek:Sigma} is an upper-case sigma.\n";

use charnames qw(greek);
print "\N{sigma} is Greek sigma\n";

```

Une liste de tous les noms complets est disponible dans le fichier Names.txt du répertoire lib/perl5/5.x.x/unicode.

La réponse au point 2), depuis la version 5.6.0, est que si une expression rationnelle contient des caractères Unicode alors la chaîne est considérée comme une séquence de caractères Unicode. Sinon, la chaîne est explorée comme une suite d'octets. Si la chaîne doit être vue comme une séquence de caractères Unicode et que vous voulez reconnaître un seul octet, vous pouvez utiliser la séquence d'échappement `\C`. `\C` est une classe de caractères comme `.` sauf qu'elle reconnaît *n'importe* quel octet (0-255). Donc :

```

use charnames ":full"; # utilisation des noms Unicode longs
$x = "a";
$x =~ /\C/; # correspond à 'a', mange un octet
$x = "";
$x =~ /\C/; # pas de correspondance, aucun octet consommé
$x = "\N{MERCURY}"; # caractère Unicode sur deux octets
$x =~ /\C/; # correspond, mais dangereux

```

La dernière expression rationnelle correspond mais c'est dangereux parce que la position *caractère* dans la chaîne n'est plus synchronisée avec la position *octet*. Cela engendre le message d'avertissement 'Malformed UTF-8 character' ('Caractère UTF-8 mal formé'). Un `\C` est plutôt indiqué pour reconnaître des données binaires dans une chaîne mixant du binaire et des caractères Unicode.

Revoyons maintenant les classes de caractères. Comme pour les caractères Unicode, il existe des noms pour les classes de caractères Unicode représentées par la séquence d'échappement `\p{nom}`. Leur négation `\P{nom}` existe aussi. Par exemple, pour reconnaître les caractères majuscules et minuscules :

```

use charnames ":full"; # utilisation des noms Unicode longs
$x = "BOB";
$x =~ /\p{IsUpper}/; # correspond, les caractères majuscules
$x =~ /\P{IsUpper}/; # pas de correspondance, les caractères sans majuscules
$x =~ /\p{IsLower}/; # pas de correspondance, les caractères minuscules
$x =~ /\P{IsLower}/; # correspond, les caractères sans les minuscules

```

Voici les associations entre quelques noms de classes Perl et les classes traditionnelles Unicode :

Nom de classe Perl	Nom de classe Unicode ou expression rationnelle
IsAlpha	<code>/^[LM]/</code>
IsAlnum	<code>/^[LMN]/</code>
IsASCII	<code>\$code <= 127</code>
IsCntrl	<code>/^C/</code>
IsBlank	<code>\$code =~ /^(0020 0009)\$/ /^Z[^\p]/</code>
IsDigit	<code>Nd</code>
IsGraph	<code>/^([LMNPS] Co)/</code>
IsLower	<code>Ll</code>
IsPrint	<code>/^([LMNPS] Co Zs)/</code>
IsPunct	<code>/^P/</code>
IsSpace	<code>/^Z/ (\$code =~ /^(0009 000A 000B 000C 000D)\$/</code>
IsSpacePerl	<code>/^Z/ (\$code =~ /^(0009 000A 000C 000D 0085 2028 2029)\$/</code>
IsUpper	<code>/^L[ut]/</code>
IsWord	<code>/^[LMN]/ \$code eq "005F"</code>
IsXDigit	<code>\$code =~ /^00(3[0-9] [46][1-6])\$/</code>

Vous pouvez aussi utiliser les noms officiels des classes Unicode grâce à `\p` et `\P` comme dans `\p{L}` pour les 'lettres' Unicode, `\p{Lu}` pour les lettres minuscules ou `\P{Nd}` pour les non-chiffres. Si `nom` est composé d'une seule lettre, les accolades peuvent être omises. Par exemple `\pM` est la classe de caractères Unicode 'marks' pour les accents. Pour la liste complète, voir le manuel *perlunicode*.

Unicode est découpé en plusieurs sous-ensembles de caractères que vous pouvez tester par `\p{In...}` (dans) et `\P{In...}` (hors de). Par exemple `\p{Latin}`, `\p{Greek}` ou `\P{Katakana}`. Pour une liste complète, voir le manuel *perlunicode*.

`\X` est une abréviation pour la séquence de classes de caractères qui contient les 'séquences Unicode de caractères combinés'. Une 'séquence de caractères combinés' est un caractère de base suivi d'un certain nombre de caractères de combinaison. Un exemple de caractères de combinaison est un accent. En utilisant les noms Unicode longs, `A+COMBININGRING` est une séquence de caractères combinés avec `A` comme caractère de base et `COMBININGRING` comme caractère de combinaison et cette séquence représente, en Danois, un `A` avec un cercle au-dessus comme dans le mot `Angstroem`. `\X` est équivalent à `\PM\pM*` qui signifie un caractère non-marque éventuellement suivi d'une série de caractères marques.

Pour obtenir les informations les plus récentes et complètes concernant Unicode, consultez la norme Unicode ou le site web du consortium Unicode <http://www.unicode.org/>.

Et comme si toutes ces classes ne suffisaient pas, Perl définit aussi des classes de caractères de style POSIX. Elles sont de la forme `[:nom:]` où `nom` est le nom d'une classe POSIX. Les classes POSIX sont `alpha`, `alnum`, `ascii`, `cntrl`, `digit`, `graph`, `lower`, `print`, `punct`, `space`, `upper` et `xdigit` plus les deux extensions `word` (une extension Perl pour reconnaître `\w`) et `blank` (une extension GNU). Si `utf8` est actif alors ces classes sont définies de la même manière que les classes Unicode correspondantes: `[:upper:]` est la même chose que `\p{IsUpper}`, etc. Les classes de caractères POSIX, par contre, ne nécessitent pas l'utilisation de `utf8`. Les classes `[:digit:]`, `[:word:]` et `[:space:]` correspondent aux classes familières `\d`, `\w` et `\s`. Pour obtenir la négation d'une classe POSIX, placez un `^` devant son nom comme dans `[:^digit:]` qui correspond à `\D` ou, avec `utf8`, à `\P{IsDigit}`. Les classes Unicode et POSIX peuvent être utilisées comme `\d`, en sachant que les classes POSIX ne sont accessibles qu'à l'intérieur d'une classe de caractères :

```
\s+[abc[:digit:]]xyz\s*/; # reconnaît a,b,c,x,y,z ou un chiffre
/^=item\s[[:digit:]]/;   # reconnaît '=item',
                          # suivi d'un espace et d'un chiffre

use charnames ":full";
\s+[abc\p{IsDigit}xyz]\s+/; # reconnaît a,b,c,x,y,z ou un chiffre
/^=item\s\p{IsDigit}/;    # reconnaît '=item',
                          # suivi d'un espace et d'un chiffre
```

C'est tout pour les caractères et les classes de caractères.

4.2 Compilation et stockage d'expressions rationnelles

Dans la partie 1, nous avons parlé du modificateur `//o` qui compile une expression rationnelle une seule fois. Cela suggère qu'une expression rationnelle compilée est une sorte de structure de données qui peut être stockée une fois et utilisée plusieurs fois. L'opérateur d'expression rationnelle `qr//` fait exactement cela: `qr/chaîne/` compile la chaîne `chaîne` en tant qu'expression rationnelle et transforme le résultat en quelque chose qui peut être affectée à une variable.

```
$reg = qr/foo+bar?/; # reg contient une expression rationnelle compilée
```

Puis `$reg` peut être utilisée en tant qu'expression rationnelle :

```
$x = "foooba";
$x =~ $reg; # est reconnu, exactement comme /foo+bar?/
$x =~ /$reg/; # idem, sous une forme différente
```

`$reg` peut aussi être utilisée au sein d'une expression rationnelle plus grande :

```
$x =~ /(abc)?$reg/; # est encore reconnue
```

Comme avec l'opérateur de recherche de correspondances, l'opérateur d'expression rationnelle peut utiliser différents délimiteurs tels `qr!|`, `qr{}` ou `qr~~`. Le délimiteur apostrophe (`qr"`) supprime l'interpolation des variables.

La pré-compilation des expressions rationnelles est utile pour des mises en correspondances dynamiques qui ne nécessitent pas une recompilation à chaque fois. En utilisant des expressions rationnelles pré-compilées, le programme `simple_grep` peut être transformé en un programme qui cherche plusieurs expressions rationnelles :

```

% cat > multi_grep
#!/usr/bin/perl
# multi_grep - match any of <number> regexps
# usage: multi_grep <number> regexp1 regexp2 ... file1 file2 ...

$number = shift;
$regexp[$_] = shift foreach (0..$number-1);
@compiled = map qr/$_/ , @regexp;
while ($line = <>) {
    foreach $pattern (@compiled) {
        if ($line =~ /$pattern/) {
            print $line;
            last; # correspondance trouvée, on passe à la ligne suivante
        }
    }
}
^D

% multi_grep 2 last for multi_grep
$regexp[$_] = shift foreach (0..$number-1);
foreach $pattern (@compiled) {
    last;
}

```

Le stockage des expressions rationnelles pré-compilées dans le tableau `@compiled` nous permet de faire une boucle sur les expressions rationnelles sans les recompiler. On y gagne en flexibilité sans sacrifier la vitesse.

4.3 Commentaires et modificateurs intégrés dans une expression rationnelle

À partir d'ici, nous allons parler des motifs étendus de Perl. Ce sont des extensions de la syntaxe traditionnelle des expressions rationnelles qui fournissent de nouveaux outils utiles pour la recherche de motifs. Nous avons déjà vu des extensions telles que les quantificateurs minimaux `??`, `*?`, `+?`, `{n,m}?` et `{n,}?`. Les autres extensions sont toutes de la forme `(?car...)` où `car` est un caractère qui détermine le type de l'extension.

La première extension est un commentaire `(?#texte)`. Cela permet d'inclure un commentaire dans l'expression rationnelle sans modifier sa signification. Le commentaire ne doit pas contenir de parenthèse fermante dans son texte. Un exemple :

```
/(?# reconnaît un entier:)[+-]?\d+;/
```

Ce style de commentaires a été largement amélioré grâce au modificateur `//x` qui permet des commentaires beaucoup plus libres et simples.

Les modificateurs `//i`, `//m`, `//s` et `//x` peuvent eux aussi être intégrés dans une expression rationnelle en utilisant `(?i)`, `(?m)`, `(?s)` et `(?x)`. Par exemple :

```

/(?i)yes/; # reconnaît 'yes' sans tenir compte de la casse
/yes/i;    # la même chose
/(?x)(
    [+-]? # un signe optionnel
    \d+   # les chiffres
)
/x;

```

Les modificateurs inclus ont deux avantages importants sur les modificateurs habituels. Tout d'abord, ils permettent de spécifier une combinaison de modificateurs différente pour *chaque* partie de l'expression rationnelle. C'est pratique pour mettre en correspondance un tableau d'expressions rationnelles qui ont des modificateurs différents :

```

$pattern[0] = '(?i)doctor';
$pattern[1] = 'Johnson';
...
while (<>) {
    foreach $patt (@pattern) {
        print if /$patt/;
    }
}

```

Ensuite parce que les modificateurs inclus n'agissent que sur le groupe dans lequel ils apparaissent. Donc, le regroupement peut être utilisé pour «localiser» l'effet des modificateurs :

```
/Answer: ((?i)yes)/; # reconnaît 'Answer: yes', 'Answer: YES', etc.
```

Les modificateurs inclus peuvent aussi annuler des modificateurs déjà présents en utilisant par exemple `(?-i)`. Les modificateurs peuvent être combinés en une seule expression comme `(?s-i)` qui active le mode simple ligne et annule l'insensibilité à la casse.

4.4 Les regroupements sans mémorisation

Nous avons fait remarquer dans la partie 1 qu'un regroupement `()` avait deux fonctions : 1) regrouper plusieurs éléments d'une expression rationnelle en une seule entité et 2) extraire ou mémoriser la sous-chaîne reconnue par le regroupement. Les regroupements sans mémorisation, notés `(?:regexp)`, effectuent un regroupement qui peut être traité comme une seule entité mais n'extraient pas la chaîne correspondante et ne la mémorise pas dans les variables `$1`, etc. Les deux types de regroupements (avec ou sans mémorisation) peuvent coexister dans une même expression rationnelle. Puisqu'il n'y a pas d'extraction, les regroupements sans mémorisation sont plus rapides que les regroupements avec mémorisation. Les regroupements sans mémorisation sont aussi pratiques pour choisir exactement les parties de l'expression rationnelle qui seront affectées aux variables :

```
# reconnaît un nombre, $1-$4 sont positionnées, mais nous ne voulons que $1
/([+-]?\ *(\d+(\.\d*)?)|\.\d+) ([eE][+-]?\d+)?/;

# reconnaît un nombre plus rapidement, seule $1 est positionnée
/([+-]?\ *(?:\d+(?:\.\d*)?)|\.\d+) (?:[eE][+-]?\d+)?/;

# reconnaît un nombre, on obtient $1 = nombre complet, $2 = exposant
/([+-]?\ *(?:\d+(?:\.\d*)?)|\.\d+) (?:[eE]([+-]?\d+))?/;
```

Les regroupements sans mémorisation sont aussi pratiques pour supprimer les effets indirects nuisibles lors d'une opération de découpage (via `'split'`) :

```
$x = '12a34b5';
@num = split /(a|b)/, $x; # @num = ('12', 'a', '34', 'b', '5')
@num = split /(?:a|b)/, $x; # @num = ('12', '34', '5')
```

Les regroupements sans mémorisation peuvent eux aussi inclure des modificateurs : `(?i-m:regexp)` est un regroupement sans mémorisation qui reconnaît `regexp` sans tenir compte de la casse et en désactivant le mode multi-lignes.

4.5 Regarder en arrière et regarder en avant

Cette section présente les assertions permettant de regarder en arrière ou en avant. Tout d'abord quelques petits éléments d'introduction.

Dans les expressions rationnelles en Perl, la plupart des éléments 'consomment' une certaine quantité de la chaîne avec lesquels ils sont mis en correspondance. Par exemple, l'élément `[abc]` consomme un caractère de la chaîne lorsqu'il est reconnu, dans le sens où Perl avance au caractère suivant dans la chaîne après la mise en correspondance. Il existe certains éléments, par contre, qui ne consomment aucun caractère (il ne change pas la position) lorsqu'ils sont reconnus. Les exemples que nous avons déjà vus sont des ancres. L'ancre `^` reconnaît le début de ligne mais ne consomme aucun caractère. De même, l'ancre de bordure de mot `\b` est reconnue entre un caractère mot et un caractère non-mot mais elle ne consomme aucun caractère. Ces ancres sont des exemples d'assertions de longueur nulle. « De longueur nulle » parce qu'elles ne consomment aucun caractère et « assertion » parce qu'elles testent une propriété de la chaîne. Dans le contexte de notre analogie avec la traversée d'une forêt, la plupart des éléments des expressions rationnelles nous font avancer le long du chemin alors que les ancres nous arrêtent pour regarder autour de nous. Si l'environnement local convient, nous pouvons continuer. Sinon, il faut effectuer un retour arrière.

Vérifier l'environnement peut impliquer un regard en avant, en arrière ou les deux. `^` regarde en arrière pour vérifier qu'il n'y a aucun caractère avant. `$` regarde en avant pour vérifier qu'il n'y a pas de caractère après. `\b` regarde en avant et en arrière pour vérifier que les deux caractères sont de natures différentes (mot et non-mot).

Les assertions en avant et en arrière sont des généralisations du concept d'ancres. Ce sont des assertions de longueur nulle qui nous permettent de spécifier les caractères que nous voulons. L'assertion de regard en avant est notée `(?=regexp)` alors que l'assertion de regard en arrière est notée `(?<=fixed-regexp)`. Voici quelques exemples :

```

$x = "I catch the housecat 'Tom-cat' with catnip";
$x =~ /cat(?:\s+)/; # reconnaît 'cat' dans 'housecat'
@catwords = ($x =~ /(?:\s)cat\w+/g); # correspond avec
                                     # $catwords[0] = 'catch'
                                     # $catwords[1] = 'catnip'
$x =~ /\bcat\b/; # reconnaît 'cat' dans 'Tom-cat'
$x =~ /(?:\s)cat(?:\s)/; # pas de correspondance; pas de 'cat' isolé
                          # au milieu de $x

```

Notez que les parenthèses dans `(?=regex)` et `(?<=regex)` sont sans mémorisation puisque ce sont des assertions de longueur nulle. Donc, dans le deuxième exemple, les sous-chaînes capturées sont celles reconnues par l'expression rationnelle complète. Les assertions en avant `(?=regex)` peuvent reconnaître une expression rationnelle quelconque. Par contre, les assertions en arrière `(?<=fixed-regex)` ne fonctionnent que pour des expressions rationnelles de longueur fixe (dont le nombre de caractères est fixé). Donc `(?<=(ab|bc))` est correct mais pas `(?<=(ab)*)`. Les négations de ces assertions se notent respectivement `(?!regex)` et `(?<!\s)`. Elles sont évaluées à vrai si l'expression rationnelle ne correspond pas :

```

$x = "foobar";
$x =~ /foo(?:!bar)/; # pas de correspondance, 'bar' suit 'foo'
$x =~ /foo(?:!baz)/; # reconnue, 'baz' ne suit pas 'foo'
$x =~ /(?:!\s)foo/; # reconnue, il n'y a pas de \s avant 'foo'

```

La classe `\C` n'est pas utilisable dans une assertion en arrière parce que cette classe utilise une tricherie sur la définition d'un caractère qui le deviendrait encore plus en regardant vers l'arrière.

4.6 Utilisation de sous-expressions indépendantes pour empêcher les retours arrière

Les dernières extensions d'expression rationnelle de ce tutoriel sont encore expérimentales en 5.6.0. Jouez avec, utilisez les dans certains codes, mais ne dépendez pas d'elles pour du code véritable.

Les **sous-expressions indépendantes** sont des expressions rationnelles qui, dans le contexte d'une expression rationnelle plus grande, fonctionnent indépendamment de cette expression rationnelle globale. C'est à dire qu'elles consomment tout ce qu'elles veulent de la chaîne sans prendre en compte la possibilité de reconnaissance de l'expression rationnelle globale. Les sous-expressions indépendantes sont représentées par `(?>regex)`. Pour illustrer leur comportement, considérons tout d'abord une expression rationnelle ordinaire :

```

$x = "ab";
$x =~ /a*ab/; # correspondance

```

Il y a évidemment correspondance mais lors de la reconnaissance, la sous-expression `a*` consomme d'abord le `a`. Ce faisant, elle empêche la reconnaissance de l'expression rationnelle globale. Donc, après retour arrière, `a*` laisse le `a` et reconnaît la chaîne vide. Ici, ce que `a*` a reconnu est *dépendant* de ce qui est reconnu par le reste de l'expression rationnelle.

En revanche, considérons l'expression rationnelle avec une sous-expression indépendante :

```

$x =~ /(?:>a*)ab/; # pas de correspondance

```

La sous-expression indépendante `(?>a*)` ne tiens pas compte du reste de l'expression rationnelle et donc consomme le `a`. Le reste de l'expression rationnelle `ab` ne peut plus trouver de correspondances. Puisque `(?>a*)` est indépendante, il n'y a pas de retour arrière et la sous-expression indépendante ne relâche pas son `a`. Donc l'expression rationnelle globale n'est pas reconnue. On observe un comportement similaire avec deux expressions rationnelles complètement indépendantes :

```

$x = "ab";
$x =~ /a*/g; # est reconnue, consomme le 'a'
$x =~ /\Gab/g; # pas de correspondance, plus de 'a' disponible

```

Ici `//g` et `\G` créent un point de non-retour entre les deux expressions rationnelles. Les expressions rationnelles avec des sous-expressions indépendantes font un peu la même chose en plaçant un point de non-retour après la reconnaissance d'une sous-expression indépendante.

Cette possibilité de blocage du retour arrière grâce aux sous-expressions indépendantes est très pratique. Supposons que nous voulons reconnaître une chaîne non-vide entre parenthèses (pouvant contenir elle-même un niveau de parenthèses). L'expression rationnelle suivante fonctionnera :

```
$x = "abc(de(fg)h); # parenthèse non refermée
$x =~ /\( ( [^()]+ | \([^()]*\ )+ )\)/x;
```

L'expression rationnelle reconnaît une parenthèse ouvrante, une ou plusieurs occurrences d'une alternative et une parenthèse fermante. La première branche de l'alternative `[^()]+` reconnaît un sous-chaîne sans parenthèse et la seconde branche `\([^()]*\)` reconnaît une sous-chaîne entourée de parenthèses. Cette expression rationnelle est malheureusement pathologique : elle contient des quantificateurs imbriqués de la forme `(a+|b)+`. Nous avons expliqué dans la partie 1 pourquoi une telle imbrication impliquait un temps d'exécution exponentiel en cas de non-reconnaissance. Pour prévenir cette explosion combinatoire, nous devons empêcher les retours arrière inutiles quelque part. On peut y arriver en incluant le quantificateur interne dans une sous-expression indépendante :

```
$x =~ /\( ( (?>[^()]+) | \([^()]*\ )+ )\)/x;
```

Ici, `(?>[^()]+)` interrompt le partitionnement combinatoire de la chaîne en consommant la partie la plus grande possible de la chaîne sans permettre le retour arrière. Donc, la non-reconnaissance sera détectée beaucoup plus rapidement.

4.7 Les expressions conditionnelles

Une **expression conditionnelle** est une forme d'instruction si-alors-sinon qui permet de choisir entre deux motifs à reconnaître selon une condition. Il existe deux types d'expressions conditionnelles : `(?(condition)motif-oui)` et `(?(condition)motif-oui|motif-non)`. `(?(condition)motif-oui)` agit comme une instruction `'if() {}'` en Perl. Si la condition est vraie, le motif-oui doit être reconnu. Si la condition est fausse, le motif-oui n'est pas pris en compte et perl passe à l'élément suivant dans l'expression rationnelle. La seconde forme est comme une instruction `'if() {}else {}'` en Perl. Si la condition est vraie le motif-oui doit être reconnu sinon c'est le motif-non qui doit être reconnu.

La condition a deux formes possibles. La première forme est simplement un entier entre parenthèses (`entier`). La condition est vraie si le regroupement mémorisé correspondant `\entier` a été reconnue plus tôt dans l'expression rationnelle. La seconde forme est une assertion de longueur nulle (`?...`), soit en avant, soit en arrière, soit une assertion de code (dont on parlera dans la section suivante).

La première forme de la condition nous permet de choisir, avec plus de flexibilité, ce que l'on veut reconnaître en fonction de ce qui a déjà été reconnu. L'exemple suivant cherche les mots de la forme `"xx"` ou `"xyyx"` :

```
% simple_grep '^(\\w+) (\\w+)?(? (2) \\2 \\1 | \\1) $' /usr/dict/words
beriberi
coco
couscous
deed
...
toot
toto
tutu
```

La condition sous la forme d'une assertion de longueur nulle en arrière, combinée avec des références arrières, permet à une partie de la reconnaissance d'influencer une autre partie de la reconnaissance qui arrive plus tard. Par exemple :

```
/[ATGC]+(? (?<=AA) G|C) $/;
```

reconnaît une séquence d'ADN qui ne se termine ni par AAG ni par C. Remarquez la notation `(? (?<=AA) G|C)` et non pas `(? ((?<=AA)) G|C)` ; pour les assertions de longueur nulle, les parenthèses ne sont pas nécessaires.

4.8 Un peu de magie : exécution de code Perl dans une expression rationnelle

Normalement les expressions rationnelles sont une partie d'une expression Perl. Les expressions d'**évaluation de code** renversent cela en permettant de placer n'importe quel code Perl à l'intérieur d'une expression rationnelle. Une expression d'évaluation de code est notée `{code}` où `code` est une chaîne contenant des instructions Perl.

Une expression d'évaluation de code est une assertion de longueur nulle et la valeur qu'elle retourne dépend de son environnement. Il y a deux possibilités : soit l'expression est utilisée comme condition dans une expression conditionnelle `(?(condition) ...)` soit ce n'est pas le cas. Si l'expression d'évaluation de code est une condition, le code est évalué et son résultat (c'est à dire le résultat de la dernière instruction) est utilisé pour déterminer la véracité de la condition. Si l'expression d'évaluation de code n'est pas une condition, l'assertion est toujours vraie et le résultat du code est stocké dans la variable spéciale `$^R`. Cette variable peut être utilisée dans des expressions d'évaluation de code apparaissant plus tard dans l'expression rationnelle. Voici quelques exemples :

```
$x = "abcdef";
$x =~ /abc(?:{print "Hi Mom!";})def/; # reconnue,
                                     # affiche 'Hi Mom!'
$x =~ /aaa(?:{print "Hi Mom!";})def/; # non reconnue,
                                     # pas de 'Hi Mom!'
```

Attention à l'exemple suivant :

```
$x =~ /abc(?:{print "Hi Mom!";})ddd/; # non reconnue,
                                     # pas de 'Hi Mom!'
                                     # mais pourquoi ?
```

En première approximation, vous pourriez croire qu'il n'y a pas d'affichage parce que ddd ne peut être reconnue dans la chaîne explorée. Mais regardez l'exemple qui suit :

```
$x =~ /abc(?:{print "Hi Mom!";})[d]dd/; # non reconnue,
                                     # mais _affiche_ 'Hi Mom!'
```

Que s'est-il passé dans ce cas ? Si vous avez tout suivi jusqu'ici, vous savez que les deux expressions rationnelles précédentes sont équivalentes – enfermer le d dans une classe de caractères ne change en rien ce qui peut être reconnu. Alors pourquoi l'une n'affiche rien alors que l'autre le fait ?

La réponse est liée aux optimisations faites par le moteur d'expressions rationnelles. Dans le premier cas, tout ce que le moteur voit c'est une série de caractères simples (mis à part la construction `{}()`). Il est assez habile pour s'apercevoir que la chaîne 'ddd' n'apparaît pas dans la chaîne explorée avant même de commencer son exploration. Alors que dans le second cas, nous avons utilisé une astuce pour lui faire croire que notre motif était plus compliqué qu'il ne l'est. Il voit donc notre classe de caractères et décide qu'il doit commencer l'exploration pour déterminer si notre expression rationnelle peut ou non être reconnue. Et lors de cette exploration, il atteint l'instruction d'affichage avant de s'apercevoir qu'il n'y a pas de correspondance possible.

Pour en savoir un peu plus sur les optimisations faites par le moteur, reportez-vous à la section §4.9 plus bas.

D'autres exemples avec `{}()` :

```
$x =~ /(?:{print "Hi Mom!";})/;      # reconnue,
                                     # affiche 'Hi Mom!'
$x =~ /(?:{$c = 1;})({print "$c";})/; # reconnue,
                                     # affiche '1'
$x =~ /(?:{$c = 1;})({print "$^R";})/; # reconnue,
                                     # affiche '1'
```

La magie évoquée dans le titre de cette section apparaît lorsque le processus de recherche de correspondance effectue un retour arrière. Si le retour arrière implique une expression d'évaluation de code et si les variables modifiées par ce code ont été localisées (via `local`) alors les modifications faites sur ces variables sont annulées ! Donc, si vous voulez compter le nombre de fois ou un caractère à été reconnu lors d'une mise en correspondance, vous pouvez utiliser le code suivant :

```
$x = "aaaa";
$count = 0; # initialisation du compteur de 'a'
$c = "bob"; # pour montrer que $c n'est pas modifié
$x =~ /(?:{local $c = 0;})          # initialisation du compteur
      ( a                            # 'a' est reconnu
        (?:{local $c = $c + 1;})     # incrémentation du compteur
      )*                             # on répète cela autant que possible
      aa                             # mais on reconnaît 'aa' à la fin
      (?:{$count = $c;})            # copie de $c local dans $count
/x;
print "'a' count is $count, \ $c variable is '$c'\n";
```

Ce code affiche :

```
'a' count is 2, $c variable is 'bob'
```

Si nous remplaçons `(?{local $c=$c+1;})` par `(?{$c=$c+1;})`, les modifications faites à la variable ne sont *pas* annulées lors du retour arrière et nous obtenons :

```
'a' count is 4, $c variable is 'bob'
```

Notez bien que seules les modifications aux variables localisées sont annulées. Les autres effets secondaires de l'exécution du code sont permanents. Donc :

```
$x = "aaaa";
$x =~ /(a(?{print "Yow\n";})*aa/;
```

produit :

```
Yow
Yow
Yow
Yow
```

Le résultat `$^R` est automatiquement localisés et donc il se comporte bien lors de retours arrière.

Cet exemple utilise une expression d'évaluation de code dans une condition pour reconnaître l'article 'the' soit en anglais soit en allemand :

```
$lang = 'DE'; # en allemand
...
$text = "das";
print "matched\n"
    if $text =~ /(?{
        $lang eq 'EN'; # est-on en anglais ?
    })
    the |           # si oui, alors il faut reconnaître 'the'
    (die|das|der)  # sinon, reconnaître 'die|das|der'
    )
/xi;
```

Remarquez ici que la syntaxe est `(?{...}motif-oui|motif-non)` et non pas `(?(?{...})motif-oui|motif-non)`. En d'autres termes, dans le cas d'une expression d'évaluation de code, les parenthèses autour de la condition sont optionnelles.

Si vous tentez d'utiliser des expressions d'évaluation de code combinées avec des variables interpolées, perl risque de vous surprendre :

```
$bar = 5;
$pat = '(?{ 1 })';
/foo(?{ $bar })bar/; # se compile bien, $bar n'est pas interpolée
/foo(?{ 1 })$bar/;  # erreur de compilation !
/foo${pat}bar/;    # erreur de compilation !

$pat = qr/(?{ $foo = 1 })/; # précompilation d'une expression
/foo${pat}bar/;          # se compile bien
```

Si une expression rationnelle contient soit des expressions d'évaluation de code et des variables interpolées, soit une variable dont l'interpolation donne une expression d'évaluation de code, perl traite cela comme une erreur dans l'expression rationnelle. En revanche, si l'expression d'évaluation de code est précompilée dans une variable, l'interpolation fonctionne. Pourquoi cette erreur ?

C'est parce que la combinaison de l'interpolation de variables et des expressions d'évaluation de code est risquée. Elle est risquée parce que beaucoup de programmeurs qui écrivent des moteurs de recherche utilisent directement les valeurs fournies par l'utilisateur dans leurs expressions rationnelles :

```
$regexp = <>; # lecture de l'expression rationnelle de l'utilisateur
$chomp $regexp; # suppression d'un éventuel passage à la ligne
$text =~ /$regexp/; # recherche de $regexp dans $text
```

Si la variable `$regex` pouvait contenir des expressions d'évaluation de code, l'utilisateur pourrait exécuter n'importe quel code Perl. Par exemple, un petit rigolo pourrait chercher `system('rm -rf *')` ; pour effacer tous vos fichiers. En ce sens, la combinaison de l'interpolation de variables et des expressions d'évaluation de code **souillerait** votre expression rationnelle. Donc, par défaut, cette combinaison n'est pas autorisée. SI vous n'êtes pas concerné par les utilisateurs malicieux, il est possible de désactiver cette interdiction en invoquant `use re 'eval'` :

```
use re 'eval';      # pas de sécurité
$bar = 5;
$pat = '(?{ 1 })';
/foo(?{ 1 })$bar/; # se compile bien
/foo${pat}bar/;   # se compile bien
```

Une autre forme d'expressions impliquant une évaluation de code est l'expression d'évaluation de code produisant un motif. Cette expression est comme la précédente sauf que le résultat de l'évaluation du code est traité comme un élément d'expression rationnelle à mettre en correspondance immédiatement. Voici un exemple simple :

```
$length = 5;
$char = 'a';
$x = 'aaaaabb';
$x =~ /(??{$char x $length})/x; # reconnue, il y a bien 5 'a'
```

Le dernier exemple contient à la fois des expressions ordinaires et des expressions impliquant de l'évaluation de code. Il détecte si les espacements entre les 1 d'une chaîne binaire 1101010010001... respectent une suite de Fibonacci 0, 1, 1, 2, 3, 5...

```
$s0 = 0; $s1 = 1; # conditions initiales
$x = "1101010010001000001";
print "It is a Fibonacci sequence\n"
    if $x =~ /^1          # on reconnaît le '1' initial
        (
            (??{'0' x $s0}) # reconnaît $s0 '0'
            1                # puis un '1'
            (??{
                $largest = $s0; # la plus large séquence
                $s2 = $s1 + $s0; # calcule du terme suivant
                $s0 = $s1;      # dans la suite de Fibonacci
                $s1 = $s2;
            })
        )+ # répété autant que possible
    $      # c'est la fin
/x;
print "Largest sequence matched was $largest\n";
```

Cela affiche :

```
It is a Fibonacci sequence
Largest sequence matched was 5
```

Remarquez que les variables `$s0` et `$s1` ne sont pas interpolées lorsque l'expression rationnelle est compilée comme cela se passerait pour des variables en dehors d'une expression d'évaluation de code. En fait, le code est évalué à chaque fois que perl le rencontre lors de la mise en correspondance.

Cette expression rationnelle sans le modificateur `//x` est :

```
/^1((??{'0' x $s0})1(??{$largest=$s0;$s2=$s1+$s0;$s0=$s1;$s1=$s2;}))+$/;
```

et c'est un bon départ pour le concours d'Obfuscated Perl ;-). Lorsqu'on utilise du code et des expressions conditionnelles, la forme étendue des expressions rationnelles est toujours nécessaire pour écrire et déboguer ces expressions.

4.9 Directives (pragma) et déverminage

Il existe plusieurs directives (pragma) pour contrôler et déboguer les expressions rationnelles en Perl. Nous avons déjà rencontré une directive dans la section précédente, `use re 'eval'`, qui autorise la coexistence de variables interpolées et d'expressions d'évaluation de code dans la même expression rationnelle. Les autres directives sont :

```
use re 'taint';
$tainted = <>;
@parts = ($tainted =~ /(\w+)\s+(\w+)/); # @parts est maintenant souillé
```

La directive `taint` rend souillées les sous-chaînes extraites lors d'une mise en correspondance appliquée à une chaîne souillée. Ce n'est pas le cas par défaut puisque les expressions rationnelles sont souvent utilisées pour extraire une information sûre d'une chaîne souillée. Utilisez `taint` lorsque l'extraction ne garantit pas la sûreté de l'information extraite. Les deux directives `taint` et `eval` ont une portée lexicale limitée au bloc qui les englobe.

```
use re 'debug';
/^(.*)$/s; # affichage d'information de debug

use re 'debugcolor';
/^(.*)$/s; # affichage d'information de debug en couleur
```

Les directives globales `debug` et `debugcolor` vous permettent d'obtenir des informations détaillées lors de la compilation et de l'exécution des expressions rationnelles. `debugcolor` est exactement comme `debug` sauf que les informations sont affichées en couleur sur les terminaux qui sont capables d'afficher les séquences `termcap` de colorisation. Voici quelques exemples de sorties :

```
% perl -e 'use re "debug"; "abc" =~ /a*b+c/;'
Compiling REx 'a*b+c'
size 9 first at 1
 1: STAR(4)
 2:  EXACT <a>(0)
 4: PLUS(7)
 5:  EXACT <b>(0)
 7: EXACT <c>(9)
 9: END(0)
floating 'bc' at 0..2147483647 (checking floating) minlen 2
Guessing start of match, REx 'a*b+c' against 'abc'...
Found floating substr 'bc' at offset 1...
Guessed: match at offset 0
Matching REx 'a*b+c' against 'abc'
Setting an EVAL scope, savestack=3
 0 <> <abc>      | 1:  STAR
                  EXACT <a> can match 1 times out of 32767...
Setting an EVAL scope, savestack=3
 1 <a> <bc>      | 4:  PLUS
                  EXACT <b> can match 1 times out of 32767...
Setting an EVAL scope, savestack=3
 2 <ab> <c>      | 7:  EXACT <c>
 3 <abc> <>      | 9:  END
Match successful!
Freeing REx: 'a*b+c'
```

Si vous êtes arrivés aussi loin dans ce tutoriel, vous pouvez probablement comprendre à quoi correspondent les différentes parties de cette sortie. La première partie :

```
Compiling REx 'a*b+c'
size 9 first at 1
 1: STAR(4)
 2:  EXACT <a>(0)
 4: PLUS(7)
 5:  EXACT <b>(0)
 7: EXACT <c>(9)
 9: END(0)
```

décrit la phase de compilation. STAR(4) signifie qu'il y a un objet étoilé, dans ce cas 'a', et qu'une fois reconnu, il faut aller en 4, c'est à dire PLUS(7). Les lignes du milieu décrivent des heuristiques et des optimisations appliquées avant la mise en correspondance :

```
floating 'bc' at 0..2147483647 (checking floating) minlen 2
Guessing start of match, REx 'a*b+c' against 'abc'...
Found floating substr 'bc' at offset 1...
Gussed: match at offset 0
```

Puis la mise en correspondance est effectuée et les lignes qui restent décrivent ce processus :

```
Matching REx 'a*b+c' against 'abc'
Setting an EVAL scope, savestack=3
 0 <> <abc>          | 1: STAR
                       EXACT <a> can match 1 times out of 32767...
Setting an EVAL scope, savestack=3
 1 <a> <bc>          | 4:  PLUS
                       EXACT <b> can match 1 times out of 32767...
Setting an EVAL scope, savestack=3
 2 <ab> <c>          | 7:  EXACT <c>
 3 <abc> <>          | 9:  END
Match successful!
Freeing REx: 'a*b+c'
```

Chaque pas est de la forme `n<x><y>` où `<x>` est la partie de la chaîne reconnue et `<y>` est la partie de la chaîne non encore reconnue. | 1: STAR indique que perl est rendu à la ligne 1 de la phase de compilation vue précédemment. Voir le titre *Débogage des expressions rationnelles* dans le manuel *perldebguts* pour de plus amples informations.

Une autre méthode pour déboguer les expressions rationnelles consiste à inclure des instructions `print` dans l'expression rationnelle.

```
"that this" =~ m@(?{print "Start at position ", pos, "\n";})
                t(?{print "t1\n";})
                h(?{print "h1\n";})
                i(?{print "i1\n";})
                s(?{print "s1\n";})
                |
                t(?{print "t2\n";})
                h(?{print "h2\n";})
                a(?{print "a2\n";})
                t(?{print "t2\n";})
                (?{print "Done at position ", pos, "\n";})
@x;
```

qui affiche :

```
Start at position 0
t1
h1
t2
h2
a2
t2
Done at position 4
```

5 BUGS

Les expressions d'évaluation de code, les expressions conditionnelles et les expressions indépendantes sont **expérimentales**. Ne les utilisez pas dans du code de production. Pas encore.

6 VOIR AUSSI

C'est juste un tutoriel. Pour une documentation complète sur les expressions rationnelles en Perl, voir le manuel *perlre*. Pour plus d'informations sur l'opérateur de mise en correspondance `m//` et l'opérateur de substitution `s///`, voir le titre Opérateurs d'expression rationnelle dans le manuel *perlop*. Pour plus d'informations sur les opérations de découpage via `split`, voir le titre `split` dans le manuel *perlfunc*.

Un très bon livre sur l'utilisation des expressions rationnelles, voir le livre *Mastering Regular Expressions* par Jeffrey Friedl (édité par O'Reilly, ISBN 1556592-257-3) (N.d.t: une traduction française existe.)

7 AUTEURS ET COPYRIGHT

Copyright (c) 2000 Mark Kvale All rights reserved.

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

Copyright (c) 2000 Mark Kvale Tous droits réservés.

Ce document peut être distribué sous les mêmes termes que Perl lui-même.

7.1 Remerciements

L'exemple des codons dans une chaîne d'ADN est librement inspiré de l'exemple de codes ZIP du chapitre 7 de *Mastering Regular Expressions*.

L'auteur tient à remercier Jeff Pinyan, Andrew Johnson, Peter Haworth, Ronald J Kimball et Joe Smith pour leur aide et leurs commentaires.

8 TRADUCTION

8.1 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/>.

8.2 Traducteur

Traduction initiale et mise à jour: Paul Gaborit [Paul.Gaborit at enstimac.fr](mailto:Paul.Gaborit@enstimac.fr).

8.3 Relecture

Jean-Louis Morel [<jl_morel@bribes.org>](mailto:jl_morel@bribes.org)

9 À propos de ce document

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