perlop

Table des matières

1 NAME/NOM

perlop - Opérateurs Perl et priorité

2 DESCRIPTION

2.1 Priorité des opérateurs et associativité

Les règles de priorité et d'associativité des opérateurs en Perl fonctionnent à peu près comme en mathématique.

Les *règles de priorité* entre les opérateurs déterminent l'ordre d'évaluation de ces opérateurs. Par exemple, dans `2 + 4 * 5`, la multiplication a une plus grande priorité et donc `4 * 5` est évalué en premier pour finalement calculer `2 + 20 == 22` et non `6 * 5 == 30`.

Les *règles d'associativité* des opérateurs définissent l'ordre dans lequel est évaluée une séquence composée de plusieurs occurrences d'un même opérateur : ces occurrences, selon l'opérateur, peuvent être évaluées en commençant par celle de gauche ou par celle de droite. Par exemple, dans `8 - 4 -2`, la soustraction étant associative à gauche, Perl évalue l'expression de gauche à droite. `8 - 4` est évalué en premier pour finalement calculer `4 - 2 == 2` et non `8 - 2 == 6`.

Le tableau suivant présente les opérateurs Perl et leur priorité, du plus prioritaire au moins prioritaire. Remarquez que tous les opérateurs empruntés au langage C gardent le même ordre de priorité entre eux même lorsque ces priorités sont légèrement tordues. (Cela rend plus simple l'apprentissage de Perl par les programmeurs C.) À quelques exceptions près, tous ces opérateurs agissent sur des valeurs scalaires et pas sur des tableaux de valeurs.

```
gauche      termes et opérateurs de listes (leftward)
gauche      ->
nonassoc    ++ --
droite      **
droite      ! ~ \ et + et - unaires
gauche      =~ !~
gauche      * / % x
gauche      + - .
gauche      << >>
nonassoc    opérateurs nommés unaires
nonassoc    < > <= >= lt gt le ge
nonassoc    == != <=> eq ne cmp
gauche      &
gauche      | ^
gauche      &&
gauche      || //
nonassoc    ..  ...
droite      ?:
droite      = += -= *= etc.
gauche      , =>
nonassoc    opérateurs de listes (rightward)
droite      not
gauche      and
gauche      or xor err
```

Dans les sections qui suivent, ces opérateurs sont présentés par ordre de priorité.

De nombreux opérateurs peuvent être redéfinis pour des objets. Voir *overload*.

2.2 Termes et opérateurs de listes (leftward)

Un TERME a la plus haute priorité en Perl. Cela inclut les variables, les apostrophes et autres opérateurs style apostrophe, les expressions entre parenthèses et n'importe quelle fonction dont les arguments sont donnés entre parenthèses. En fait, ce ne sont pas vraiment des fonctions, juste des opérateurs de listes et des opérateurs unaires qui se comportent comme des fonctions parce que vous avez mis des parenthèses autour des arguments. Tout cela est documenté dans *perlfunc*.

Si un opérateur de liste (print(), etc.) ou un opérateur unaire (chdir(), etc.) est directement suivi par une parenthèse gauche, l'opérateur et les arguments entre parenthèses se voient attribués la priorité la plus haute exactement comme un appel de fonction.

En l'absence de parenthèses, la priorité des opérateurs de liste comme `print`, `sort` ou `chmod` n'est ni très haute ni très basse et dépend de ce qu'il y a à gauche et/ou à droite de l'opérateur. Par exemple, dans :

```
@ary = (1, 3, sort 4, 2);
print @ary;         # affiche 1324
```

la virgule à droite du tri (sort) est évaluée avant le tri (sort) alors que la virgule de gauche est évaluée après. En d'autres termes, un opérateur de listes a tendance à manger tous les arguments qui le suit et à se comporter comme un simple TERME par rapport à l'expression qui le précède. Faites attention aux parenthèses :

```
# L'évaluation de exit a lieu avant le print !
print($foo, exit);  # Pas vraiment ce que vous vouliez.
print $foo, exit;   # Ni dans ce cas.

# L'évaluation de print a lieu avant le exit.
(print $foo), exit; # C'est ce que vous voulez.
print($foo), exit;  # Ici aussi.
print ($foo), exit; # Et encore dans ce cas.
```

Remarquez aussi que :

```
print ($foo & 255) + 1, "\n";
```

ne donnera probablement pas ce que vous attendiez à priori. Les parenthèses entourent la liste d'arguments du `print` qui est évalué (en affichant le résultat de `$foo & 255`). Ensuite la valeur un est additionnée au résultat retourné par `print` (habituellement 1). Le résultat est donc quelque chose comme :

```
1 + 1, "\n"; # Évidemment pas ce que vous souhaitiez
```

Pour faire cela proprement, il faut écrire :

```
print(($foo & 255) + 1, "\n");
```

Voir les Opérateurs unaires nommés pour plus de détails.

Sont aussi reconnus comme des termes les constructions `do {}` et `eval {}`, les appels à des subroutines ou à des méthodes ainsi que les constructeurs anonymes `[]` et `{}`.

Voir aussi Opérateurs apostrophe et type apostrophe à la fin de cette section mais aussi Opérateurs d'E/S (§**??**).

2.3 L'opérateur flèche

Comme en C et en C++, "`->`" est un opérateur de déréférencement infixe. Si du côté droit on trouve soit `[...]`, soit `{...}`, soit `(...)` alors le côté gauche doit être une référence vraie ou symbolique vers respectivement un tableau, une table de hachage ou une subroutine (ou techniquement parlant, un emplacement capable de stocker une référence en dur si c'est une référence vers un tableau ou une table de hachage utilisée pour une affectation). Voir *perlreftut* et *perlref*.

Par contre, si le côté droit est un nom de méthode ou une simple variable scalaire contenant un nom de méthode ou une référence vers une subroutine alors le côté gauche doit être soit un objet (une référence bénie – par bless()) soit un nom de classe (c'est à dire un nom de package). Voir *perlobj*.

2.4 Auto-incrémentation et auto-décrémentation

Les opérateurs "++" et "–" fonctionnent comme en C. Placés avant la variable, ils incrémentent ou décrémentent la variable avant de retourner sa valeur. Placés après, ils incrémentent ou décrémentent la variable après avoir retourner sa valeur.

```
$i = 0;  $j = 0;
print $i++;  # affiche 0
print ++$j;  # affiche 1
```

Remarquez que, comme en C, Perl ne définit pas **quand** la variable est incrémentée ou décrémentée. Vous savez juste que cela sera fait à un moment avant ou après le moment où la valeur est retournée. Cela signifie aussi que plusieurs modifications de la même variable dans une instruction n'ont pas un comportement garanti. Évitez donc des instructions comme :

```
$i = $i ++;
print ++ $i + $i ++;
```

Perl ne garantie pas le résultat des instructions ci-dessus.

De plus, l'opérateur d'auto-incrémentation inclut un comportement magique. Si vous incrémentez une variable numérique ou qui a déjà été utilisée dans un contexte numérique, vous obtenez l'incrément normal. Si, par contre, la variable n'a été utilisée que dans un contexte de chaîne et correspond au motif `/^[a-zA-Z]*[0-9]*\z/`, l'incrémentation a lieu sur la chaîne elle-même en préservant les caractères dans leur intervalle et en gérant les éventuelles retenues :

```
print ++($foo = '99');      # affiche '100'
print ++($foo = 'a0');      # affiche 'a1'
print ++($foo = 'Az');      # affiche 'Ba'
print ++($foo = 'zz');      # affiche 'aaa'
```

La valeur `undef` est toujours considérée comme numérique et en particulier est changée en `0` avant son incrémentation (et donc l'incrémentation postfixée d'une valeur indéfinie retournera `0` plutôt que `undef`).

L'opérateur d'auto-décrémentation n'est pas magique.

2.5 Puissance

L'opérateur binaire "**" est l'opérateur puissance. Remarquez qu'il est plus prioritaire que le moins unaire donc -2**4 signifie -(2**4) et non pas (-2)**4. (Il est implémenté par la fonction C pow(3) qui travaille réellement sur des doubles en interne.)

2.6 Opérateurs symboliques unaires

L'opérateur unaire "!" est la négation logique, c.-à-d. "non". Voir aussi `not` pour une version moins prioritaire de cette opération.

L'opérateur unaire "-" est la négation arithmétique si l'opérande est numérique. Si l'opérande est un identificateur, il retourne une chaîne constituée du signe moins suivi de l'identificateur. Si la chaîne qui suit commence par un plus ou un moins, la valeur retournée est la chaîne commençant par le signe opposé. L'un des effets de ces règles est que `-bareword` est équivalent à `"-bareword"`. En revanche, si la chaîne qui suit commence par un caractère non-alphabétique (sauf "+" ou "-"), Perl tentera de convertir cette chaîne en une valeur numérique à laquelle la négation arithmétique sera appliquée. Si cette chaîne ne peut pas être convertie proprement en une valeur numérique, Perl émettra l'avertissement **Argument "the string" isn't numeric in negation (-) at ...**.

L'opérateur unaire "~" effectue la négation bit à bit, c.-à-d. le complément à 1. Par exemple `0666 & ~027` vaut 0640. (Voir aussi Arithmétique entière et Opérateurs bit à bit sur les chaînes.) Notez que la taille du résultat dépend de la plate-forme : `~0` a une taille de 32 bits sur une plate-forme 32-bit et une taille de 64 bits sur une plate-forme 64-bit. Donc si vous attendez une certain nombre de bits, souvenez-vous d'utiliser l'opérateur & pour masquer les bits en excès.

L'opérateur unaire "+" n'a aucun effet même sur les chaînes. Il est pratique pour séparer syntaxiquement un nom de fonction d'une expression entre parenthèses qui autrement aurait été interprétée comme la liste complète des arguments de la fonction. (Voir les exemples de Termes et opérateurs de listes (leftward).)

L'opérateur unaire "\" crée une référence vers ce qui le suit. Voir *perlref*. Ne confondez pas ce comportement avec celui de la barre oblique inversée (backslash) à l'intérieur d'une chaîne bien que les deux formes proposent une sorte de protection contre l'interprétation de ce qui les suit.

2.7 Opérateurs d'application d'expressions rationnelles

L'opérateur binaire "=~" applique un motif de reconnaissance à une expression scalaire. Plusieurs opérations cherchent ou modifient la chaîne $_ par défaut. Cet opérateur permet d'appliquer cette sorte d'opérations à d'autres chaînes. L'argument de droite est le motif de recherche, de substitution ou de remplacement. L'argument de gauche est ce qui est supposé être cherché, substitué ou remplacé à la place de la valeur par défaut $_. Dans un contexte scalaire, la valeur retournée indique généralement le succès de l'opération. Le comportement dans un contexte de liste dépend de chaque opérateur. Pour plus de détails, voir Opérateurs apostrophe et type apostrophe (§**??**) et, pour des exemples d'utilisation, voir *perlreftut*.

Si l'argument de droite est une expression plutôt qu'un motif de recherche, de substitution ou de remplacement, il est interprété comme un motif de recherche lors de l'exécution.

L'opérateur binaire "!~" est exactement comme "=~" sauf que la valeur retournée est le contraire au sens logique.

2.8 Opérateurs type multiplication

L'opérateur binaire "*" multiplie deux nombres.

L'opérateur binaire "/" divise deux nombres.

L'opérateur binaire "%" calcule le modulo de deux nombres. Soit deux opérandes entiers donnés `$a` et `$b` : si `$b` est positif alors `$a % $b` vaut `$a` moins le plus grand multiple de `$b` qui n'est pas plus grand que `$a`. Si `$b` est négatif alors `$a % $b` vaut `$a` moins le plus petit multiple de `$b` qui n'est pas plus petit que `$a` (c.-à-d. le résultat est plus petit ou égal à zéro). Remarquez que lorsque vous êtes dans la portée de `use integer`, "%" vous donne accès directement à l'opérateur modulo tel qu'il est défini par votre compilateur C. Cet opérateur n'est pas très bien défini pour des opérandes négatifs mais il s'exécute plus rapidement.

L'opérateur binaire "x" est l'opérateur de répétition. Dans un contexte scalaire, il retourne une chaîne constituée de son opérande de gauche répété le nombre de fois spécifié par son opérande de droite. Dans un contexte de liste, si l'opérande de gauche est entre parenthèses ou est une liste formée par `qw/chaîne/`, il répète la liste. Si l'opérande de droite est nul ou négatif, il retourne un chaîne vide ou une liste vide, selon le contexte.

```
print '-' x 80;             # affiche une ligne de '-'

print "\t" x ($tab/8), ' ' x ($tab%8);      # tab over

@ones = (1) x 80;           # une liste de quatre-vingt 1.
@ones = (5) x @ones;        # place tous les éléments à 5.
```

2.9 Opérateurs type addition

L'opérateur binaire "+" retourne la somme de deux nombres.

L'opérateur binaire "-" retourne la différence de deux nombres.

L'opérateur binaire "." concatène deux chaînes.

2.10 Opérateurs de décalages

L'opérateur binaire "<<" retourne la valeur de son opérande de gauche décalée vers la gauche d'un nombre de bits spécifié par son opérande de droite. Les arguments devraient être des entiers. (Voir aussi Arithmétique entière.)

L'opérateur binaire ">>" retourne la valeur de son opérande de gauche décalée vers la droite d'un nombre de bits spécifié par son opérande de droite. Les arguments devraient être des entiers. (Voir aussi Arithmétique entière.)

Notez que "<<" et ">>" en Perl sont implémentés en utilisant directement les opérateurs "<<" et ">>" du C. Si `use integer` (voir Arithmétique entière) est actif alors ce sont des entiers signés du C qui sont utilisés et sinon ce sont des entiers non signés. De plus, ces opérations ne génèrent jamais d'entiers plus grands que la limite imposée par le type d'entiers utilisé pour compiler Perl (32 ou 64 bits).

Le résultat obtenu lors d'un dépassement de capacité des entiers n'est pas défini puisqu'il ne l'est pas non plus en C. En d'autres termes, en supposant des entiers sur 32 bits, le résultat de `1 << 32` est indéfini. Un décalage d'un nombre négatif de bits n'est pas défini non plus.

2.11 Opérateurs unaires nommés

Les différents opérateurs unaire nommés sont traités comme des fonctions à un argument avec des parenthèses optionnelles. Si un opérateur de liste (print(), etc.) ou un opérateur unaire (chdir(), etc.) est suivi d'une parenthèse ouvrante, l'opérateur et ses arguments entre parenthèses sont considérés comme de priorité la plus haute exactement comme n'importe quel appel à une fonction. Par exemple, puisque les opérateurs unaires nommés sont plus prioritaires que || :

```
chdir $foo    || die;       # (chdir $foo) || die
chdir($foo)   || die;       # (chdir $foo) || die
chdir ($foo)  || die;       # (chdir $foo) || die
chdir +($foo) || die;       # (chdir $foo) || die
```

mais puisque * est plus prioritaire que les opérateurs unaires nommés :

```
chdir $foo * 20;    # chdir ($foo * 20)
chdir($foo) * 20;   # (chdir $foo) * 20
chdir ($foo) * 20;  # (chdir $foo) * 20
chdir +($foo) * 20; # chdir ($foo * 20)

rand 10 * 20;       # rand (10 * 20)
rand(10) * 20;      # (rand 10) * 20
rand (10) * 20;     # (rand 10) * 20
rand +(10) * 20;    # rand (10 * 20)
```

Du point de vue des priorités, les opérateurs de test sur fichiers comme `-f`, `-M`, *etc.* sont traités comme des opérateurs unaires nommés mais ils ne suivent pas les règles liées aux parenthèses autour des arguments des fonctions. Cela signifie, par exemple, que `-f($file).".bak"` est équivalent à `-f "$file.bak"`.
Voir aussi Termes et opérateurs de listes (leftward) (§**??**).

2.12 Opérateurs de comparaisons

L'opérateur binaire "<" renvoie vrai si son opérande gauche est numériquement et strictement plus petit que son opérande droit.
L'opérateur binaire ">" renvoie vrai si son opérande gauche est numériquement et strictement plus grand que son opérande droit.
L'opérateur binaire "<=" renvoie vrai si son opérande gauche est numériquement plus petit ou égal à son opérande droit.
L'opérateur binaire ">=" renvoie vrai si son opérande gauche est numériquement plus grand ou égal à son opérande droit.
L'opérateur binaire "lt" renvoie vrai si son opérande gauche est alphabétiquement et strictement plus petit que son opérande droit.
L'opérateur binaire "gt" renvoie vrai si son opérande gauche est alphabétiquement et strictement plus grand que son opérande droit.
L'opérateur binaire "le" renvoie vrai si son opérande gauche est alphabétiquement plus petit ou égal à son opérande droit.
L'opérateur binaire "ge" renvoie vrai si son opérande gauche est alphabétiquement plus grand ou égal à son opérande droit.

2.13 Opérateurs d'égalité

L'opérateur binaire "==" renvoie vrai si l'opérande gauche est numériquement égal à l'opérande droit.
L'opérateur binaire "!=" renvoie vrai si l'opérande gauche n'est pas numériquement égal à l'opérande droit.
L'opérateur binaire "<=>" renvoie -1, 0 ou 1 selon que l'opérande gauche est numériquement et respectivement plus petit, égal ou plus grand que l'opérande droit. Si votre plate-forme reconnaît NaN (*not-a-number*, n'est-pas-un-nombre) comme valeur numérique, son utilisation par "<=>" retourne undef. NaN n'est ni plus grand, ni plus petit, ni égal à quoi que ce soit (même pas NaN). NaN != NaN retourne vrai comme le fait NaN != n'importe quoi d'autre. Si votre plate-forme ne reconnaît pas NaN alors NaN est simplement une chaîne de caractères qui a 0 pour valeur numérique.

```
perl -le '$a = "NaN"; print "NaN est reconnu" if $a == $a'
perl -le '$a = "NaN"; print "NaN n'est pas reconnu" if $a != $a'
```

L'opérateur binaire "eq" renvoie vrai si l'opérande gauche est égal alphabétiquement à l'opérande droit.
L'opérateur binaire "ne" renvoie vrai si l'opérande gauche n'est pas égal alphabétiquement à l'opérande droit.
L'opérateur binaire "cmp" renvoie -1, 0 ou 1 selon que l'opérande gauche est alphabétiquement et respectivement plus petit, égal ou plus grand que l'opérande droit.
"lt", "le", "ge", "gt" et "cmp" utilisent l'ordre de tri (collation) spécifié par le locale courant si `use locale` est actif. Voir *perllocale*.

2.14 Opérateur et bit à bit

L'opérateur binaire "&" renvoie le résultat d'un ET bit à bit entre ses opérandes. (Voir aussi Arithmétique entière et Opérateurs bit à bit sur les chaînes.)

Notez que "&" a une priorité plus faible que les opérateurs de comparaison si bien que les parenthèses sont indispensables dans un test comme ci-dessous :

```
print "Pair\n" if ($x & 1) == 0;
```

2.15 Opérateurs ou et ou exclusif bit à bit

L'opérateur binaire "|" renvoie le résultat d'un OU bit à bit entre ses deux opérandes. (Voir aussi Arithmétique entière et Opérateurs bit à bit sur les chaînes.)

L'opérateur binaire "^" renvoie le résultat d'un OU EXCLUSIF (XOR) bit à bit entre ses deux opérandes. (Voir aussi Arithmétique entière et Opérateurs bit à bit sur les chaînes.)

Notez que "|" et "^" ont une priorité plus faible que les opérateurs de comparaison si bien que les parenthèses sont indispensables dans un test comme ci-dessous :

```
print "faux\n" if (8 | 2) != 10;
```

2.16 Et logique style C

L'opérateur binaire "&&" calcule un ET logique rapide. Cela signifie que si l'opérande gauche est faux, l'opérande droit n'est même pas évalué. Le contexte scalaire ou de liste se propage vers l'opérande droit si il est évalué.

2.17 Défini-ou logique style C

Bien que n'ayant pas d'équivalent direct en C, l'opérateur Perl `//` s'apparente à un ou style C. En fait, c'est exactement la même chose que || sauf qu'il ne teste pas la véracité de son opérande gauche mais le fait qu'il soit défini. Donc, `$a // $b` est similaire à `defined($a) || $b` (sauf qu'il retourne la valeur de `$a` au lieu de la valeur de `defined($a)`) et est exactement équivalent à `defined($a) ? $a : $b`. C'est très pratique pour donner des valeurs par défaut à des variables. On peut aussi tester si au moins l'une des variables `$a` ou `$b` est définie en écrivant `defined($a // $b)`.

2.18 Ou logique style C

L'opérateur binaire "||" calcule un OU logique rapide. Cela signifie que si l'opérande gauche est vrai, l'opérande droit n'est même pas évalué. Le contexte scalaire ou de liste se propage vers l'opérande droit si il est évalué.

Les opérateurs ||, `//` et `&&` renvoient la dernière valeur évaluée (contrairement aux homologues en C, || et `&&` qui renvoient 0 ou 1). Donc, un moyen raisonnablement portable de trouver le répertoire home peut être :

```
$home = $ENV{HOME} // $ENV{LOGDIR} //
  (getpwuid($<))[7] // die "You're homeless!\n";
```

En particulier, cela signifie que vous ne devriez pas les utiliser pour choisir entre deux agrégats dans une <affectation :>

```
@a = @b || @c;              # c'est pas bon
@a = scalar(@b) || @c;      # voila ce que ça signifie
@a = @b ? @b : @c;          # cela marche très bien par contre
```

Pour remplacer d'une manière plus lisible l'usage de `&&`, `//` et || pour contrôler un flot d'opérations, Perl propose les opérateurs `and`, `err` et `or` (voir plus bas). Le comportement d'évaluation rapide est le même. En revanche, comme la priorité de "and", "err" et "or" est plus faible, vous pouvez les utiliser après les opérateurs de listes sans ajouter de parenthèses :

```
unlink "alpha", "beta", "gamma"
        or gripe(), next LINE;
```

Avec les opérateurs à la C, vous auriez dû l'écrire :

```
unlink("alpha", "beta", "gamma")
        || (gripe(), next LINE);
```

En revanche, l'utilisation de "or" lors d'une affectation ne donne pas ce que vous voulez; voir plus bas.

2.19 Opérateurs d'intervalle

L'opérateur binaire ".." est l'opérateur d'intervalle qui est en fait deux opérateurs totalement différents selon le contexte. Dans un contexte de liste, il renvoie une liste de valeurs commençant à la valeur de son opérande gauche et se terminant à la valeur de son opérande droit (par pas de 1). Si la valeur de gauche est plus petite que la valeur de droite, il retourne une liste vide. C'est pratique pour écrire des boucles `foreach (1..10)` et pour des opérations de remplacement sur des tableaux. Dans l'implémentation actuelle, aucun tableau temporaire n'est généré lorsque l'opérateur d'intervalle est utilisé comme expression de boucles `foreach` mais les vieilles versions de Perl peuvent consommer énormément de mémoire lorsque vous écrivez quelque chose comme :

```
for (1 .. 1_000_000) {
    # code
}
```

L'opérateur d'intervalle fonctionne aussi avec des chaînes de caractères, en utilisant l'auto-incrémentation (voir ci-dessous).

Dans un contexte scalaire, ".." renvoie une valeur booléenne. L'opérateur est bi-stable comme un interrupteur et simule l'opérateur d'intervalle de ligne (virgule) de **sed**, de **awk** et d'autres éditeurs. Chaque opérateur ".." conserve en mémoire son propre état booléen. Il reste faux tant que son opérande gauche est faux. Puis dès que son opérande gauche devient vrai, il reste vrai jusqu'à ce que son opérande droit soit vrai. *APRÈS* quoi, l'opérateur d'intervalle redevient faux. Il ne redevient faux que le prochaine fois que l'opérateur d'intervalle est évalué. Il peut tester l'opérande droit et devenir faux lors de la même évaluation où il devient vrai (comme dans **awk**) mais il retournera encore une fois vrai. Si vous ne voulez pas qu'il teste l'opérande droit avant la prochaine évaluation (comme dans **sed**), utilisez trois points ("...") à la place de deux. Pour tout le reste, "..." se comporte exactement comme "..".

L'opérande droit n'est pas évalué tant que l'opérateur est dans l'état "faux" et l'opérande gauche n'est pas évalué tant que l'opérateur est dans l'état "vrai". La priorité de l'opérateur intervalle est un peu plus basse que celle de || et &&. La valeur retournée est soit la chaîne vide pour signifier "faux" soit un numéro de séquence (commençant à 1) pour "vrai". Le dernier numéro de séquence d'un intervalle est suivi de la chaîne "E0" ce qui ne perturbe pas sa valeur numérique mais vous donne quelque chose à chercher si vous voulez exclure cette dernière valeur. Vous pouvez exclure la première valeur en demandant une valeur supérieure à 1.

Si l'un des opérandes de l'opérateur intervalle pris dans un contexte scalaire est une expression constante, alors cet opérande est considéré comme vrai si il est égal (==) au numéro de ligne courant (la variable `$.`).

Plus précisément, la comparaison réellement effectuée est `int(EXPR) == int(EXPR)` ce qui importe uniquement si vous utilisez une expression non entière. Lorsque `$.` est utilisé implicitement, comme dans expliqué dans la paragraphe précédent, la véritable comparaison utilisée est `int(EXPR) == int($.)`. Plus encore, `"span" .. "spat"` ou `2.18 .. 3.14` dans un contexte scalaire ne font pas ce que vous voulez puisque chacun de leurs opérandes est évalué en utilisant sa représentation entière.

Exemples :

Comme opérateur scalaire :

```
if (101 .. 200) { print; }  # affiche la seconde centaine de lignes
                            # raccourci pour
                            # if ($. == 101 .. $. == 200) ...

next line if (1 .. /^$/);   # saut des lignes d'en-têtes
                            # raccourci pour
                            # ... if ($. == 1 .. /^$/);

s/^/> / if (/^$/ .. eof()); # place le corps comme citation

# analyse d'e-mail
while (<>) {
    $in_header =   1  .. /^$/;
    $in_body   = /^$/ .. eof();
    if ($in_header) {
       # ...
    } else { # in body
       # ...
    }
} continue {
    close ARGV if eof;      # réinitialisation de $. à chaque fichier
}
```

Voici un exemple illustrant les différences entre les deux opérateurs d'intervalle :

```
@lines = ("   - Foo",
          "01 - Bar",
          "1  - Baz",
          "   - Quux");

foreach (@lines) {
  if (/0/ .. /1/) {
    print "$_\n";
  }
}
```

Ce programme n'affiche que la ligne contenant "Bar". En utilisant l'opérateur d'intervalle `...`, il afficherait aussi la ligne contenant "Baz".

Et maintenant quelques exemple en tant qu'opérateur de liste :

```
for (101 .. 200) { print; } # affiche $_ 100 fois
@foo = @foo[0 .. $#foo];    # un no-op coûteux
@foo = @foo[$#foo-4 .. $#foo];      # extraction des 5 derniers items
```

L'opérateur intervalle (dans un contexte de liste) utilise l'algorithme magique d'auto-incrémentation si les opérandes sont des chaînes. Vous pouvez écrire :

```
@alphabet = ('A' .. 'Z');
```

pour obtenir toutes les lettres de l'alphabet anglais ou

```
$hexdigit = (0 .. 9, 'a' .. 'f')[$num & 15];
```

pour obtenir les chiffres hexadécimaux ou

```
@z2 = ('01' .. '31');  print $z2[$mjour];
```

pour obtenir des dates avec des zéros. Si la valeur finale donnée n'est pas dans la séquence produite par l'incrémentation magique, la séquence continue jusqu'à ce que la prochaine valeur soit plus longue que la valeur finale spécifiée.

Comme chaque opérande est évalué sous forme entière, `2.18 .. 3.14` retournera deux éléments dans un contexte de liste.

```
@list = (2.18 .. 3.14); # identique à @list = (2 .. 3);
```

2.20 Opérateur conditionnel

L'opérateur "?:" est l'opérateur conditionnel exactement comme en C. Il travaille presque comme un si-alors-sinon. Si l'argument avant le ? est vrai, l'argument avant le : est renvoyé sinon l'argument après le : est renvoyé. Par exemple :

```
printf "J'ai %d chien%s.\n", $n,
        ($n == 1) ? '' : "s";
```

Le contexte scalaire ou de liste se propage au second ou au troisième argument quelque soit le choix.

```
$a = $ok ? $b : $c;  # un scalaire
@a = $ok ? @b : @c;  # un tableau
$a = $ok ? @b : @c;  # oups, juste un compte !
```

On peut affecter quelque chose à l'opérateur si le second ET le troisième arguments sont des lvalues légales (ce qui signifie qu'on peut leur affecter quelque chose) :

```
($a_or_b ? $a : $b) = $c;
```

Il n'y a aucune garantie que cela contribue à la lisibilité de votre programme.

Puisque l'opérateur produit un résultat affectable, l'usage d'affectation sans parenthèse peut amener quelques problèmes. Par exemple, le ligne suivante :

```
$a % 2 ? $a += 10 : $a += 2
```

signifie réellement :

```
(($a % 2) ? ($a += 10) : $a) += 2
```

au lieu de :

```
($a % 2) ? ($a += 10) : ($a += 2)
```

Cela aurait probablement pu être écrit plus simplement :

```
$a += ($a % 2) ? 10 : 2;
```

2.21 Opérateurs d'affectation

"=" est l'opérateur habituel d'affectation.

Les opérateurs d'affectation fonctionnent comme en C. Donc :

```
$a += 2;
```

est équivalent à :

```
$a = $a + 2;
```

quoique sans dupliquer les éventuels effets de bord que le déréférencement de la lvalue pourraient déclencher, comme par exemple avec tie(). Les autres opérateurs d'affectation fonctionnent de la même manière. Voici ceux qui sont reconnus :

```
**=     +=     *=     &=     <<=     &&=
       -=     /=     |=     >>=     ||=
       .=     %=     ^=             //=
              x=
```

Remarque: bien que regroupés par famille, tous ces opérateurs ont la même priorité que l'affectation.

Au contraire du C, les opérateurs d'affectation produisent une lvalue valide. Modifier une affectation est équivalent à faire l'affectation puis à modifier la variable qui vient d'être affectée. C'est très pratique pour modifier une copie de quelque chose comme dans :

```
($tmp = $global) =~ tr [A-Z] [a-z];
```

De même :

```
($a += 2) *= 3;
```

est équivalent à :

```
$a += 2;
$a *= 3;
```

De manière similaire, une affectation vers une liste dans un contexte de liste produit la liste des lvalues affectées et dans un contexte scalaire produit le nombre d'éléments présents dans l'opérande de droite de l'affectation.

2.22 Opérateur virgule

L'opérateur binaire "," est l'opérateur virgule. Dans un contexte scalaire, il évalue son opérande gauche et jette le résultat puis il évalue son opérande droit et retourne cette valeur. C'est exactement comme l'opérateur virgule du C.

Dans un contexte de liste, c'est tout simplement le séparateur d'arguments de la liste. Il insère ses deux opérandes dans la liste.

L'opérateur => est un synonyme de la virgule sauf qu'il contraint un mot (constitué uniquement de caractères de mots) à sa gauche à être interprété comme une chaîne (depuis la version 5.001). Cela inclut les mots qui, dans un autre contexte, auraient pu être considérés comme des constantes ou des appels de fonctions.

```
use constant FOO => "quelque chose";

my %h = ( FOO => 23);
```

qui est équivalent à :

```
my %h = ("FOO", 23);
```

et *non* à :

```
my %h = ("quelque chose", 23);
```

Si son argument de gauche n'est pas un mot, il est tout d'abord interprété comme une expression puis, ensuite, c'est sa valeur en tant que chaîne de caractères qui est utilisée.

2.23 Opérateurs de listes (Rightward)

Du côté droit d'un opérateur de liste, il a une priorité très basse qui permet de maîtriser toutes les expressions présentes séparées par des virgules. Les seuls opérateurs de priorité inférieure sont les opérateurs logiques "and", "or" et "not" qui peuvent être utiliser pour évaluer plusieurs appels à des opérateurs de liste sans l'ajout de parenthèses supplémentaires :

```
open HANDLE, "filename"
    or die "Can't open: $!\n";
```

Voir aussi la discussion sur les opérateurs de liste dans Termes et opérateurs de liste (leftward).

2.24 Non (not) logique

L'opérateur unaire "not" renvoie la négation logique de l'expression à sa droite. Il est équivalent à "!" sauf sa priorité beaucoup plus basse.

2.25 Et (and) logique

L'opérateur binaire "and" renvoie la conjonction logique des deux expressions qui l'entourent. Il est équivalent à && sauf sa priorité beaucoup plus basse. Cela signifie qu'il est rapide: l'expression de droite est évaluée uniquement si celle de gauche est vraie.

2.26 Ou (or), ou exclusif (xor) et défini-ou (err) logiques

L'opérateur binaire "or" renvoie la disjonction logique des deux expressions qui l'entourent. Il est équivalent à || sauf sa priorité beaucoup plus basse. C'est pratique pour contrôler une suite d'opérations :

```
print FH $data              or die "Can't write to FH: $!";
```

Cela signifie qu'il est rapide: l'expression de droite est évaluée uniquement si celle de gauche est fausse. À cause de sa priorité, vous devriez l'éviter dans les affectations et ne l'utiliser que pour du contrôle d'opérations.

```
$a = $b or $c;              # bug: c'est pas bon
($a = $b) or $c;            # voila ce que ça signifie
$a = $b || $c;              # il vaut mieux l'écrire ainsi
```

Au contraire, lorsque l'affectation est dans un contexte de liste et que vous voulez utiliser "||" pour du contrôle, il vaut mieux utiliser "or" pour que l'affectation soit prioritaire.

```
@info = stat($file) || die;     # holà, sens scalaire de stat !
@info = stat($file) or die;     # meilleur, @info reçoit ce qu'il faut
```

Bien sûr, il est toujours possible d'utiliser les parenthèses.

L'opérateur binaire "err" est l'équivalent de `//` (c'est comme un "ou" binaire sauf qu'il teste si son opérande gauche est défini au lieu de tester si il est vrai). Il y a deux moyens de mémoriser "err" : soit parce que de nombreuses fonctions retourne `undef` en cas d'**err**eur, soit comme une sorte de correction (`$a=($b err 'défaut')`).

L'opérateur binaire "xor" renvoie le ou exclusif des deux expressions qui l'entourent. Il ne peut évidemment pas être rapide.

2.27 Opérateurs C manquant en Perl

Voici ce qui existe en C et que Perl n'a pas :

& unaire
: Opérateur adresse-de. (Voir l'opérateur "\" pour prendre une référence.)

*** unaire**
: Opérateur contenu-de ou de déréférencement. (Les opérateurs de déréférencement de Perl sont des préfixes typés : $, @, % et &.)

(TYPE)
: Opérateur de conversion de type (de cast).

2.28 Opérateurs apostrophe et apparentés

Bien qu'habituellement nous pensions aux apostrophes (et autres guillemets) pour des valeurs littérales, en Perl, elles fonctionnent comme des opérateurs et proposent différents types d'interpolation et de capacités de reconnaissance de motif. Perl fournit des caractères standard pour cela mais fournit aussi le moyen de choisir vos propres caractères. Dans la table suivante, le `{}` représente n'importe quelle paire de délimiteurs que vous aurez choisie.

```
Standard  Générique    Signification       Interpolation
   ''       q{}         Littérale            non
   ""      qq{}         Littérale            oui
   ``      qx{}          Commande            oui*
           qw{}        Liste de mots         non
   //       m{}   Reconnaissance de motif    oui*
           qr{}           Motif              oui*
            s{}{}       Substitution        oui*
           tr{}{}     Translittération       non (mais voir plus bas)
   <<EOF                 here-doc            oui*

 * sauf si les délimiteurs sont de simples apostrophes (').
```

Les délimiteurs qui ne marchent pas par deux utilisent le même caractère au début et à la fin par contre les quatre sortes de parenthèses (parenthèses, crochets, accolades et inférieur/supérieur) marchent par deux et peuvent être imbriquées ce qui signifie que :

```
q{foo{bar}baz}
```

est la même chose que :

```
'foo{bar}baz'
```

Remarquez par contre que cela ne fonctionne pas toujours. Par exemple :

```
$s = q{ if($a eq "}") ... }; # Mauvais
```

est une erreur de syntaxe. Le module `Text::Balanced` (disponible sur CPAN et intégré à la distribution Perl depuis la version 5.8) est capable de gérer cela correctement.

Il peut y avoir des espaces entre l'opérateur et le caractère délimiteur sauf lorsque `#` est utilisé. `q#foo#` est interprété comme la chaîne `foo` alors que `q #foo#` est l'opérateur `q` suivi d'un commentaire. Ses arguments sont alors pris sur la ligne suivante. Cela permet d'écrire :

```
s {foo}  # Remplace foo
  {bar}  # par bar.
```

Le séquences d'échappement suivantes sont disponibles dans les constructions qui font de l'interpolation et dans les translittérations :

```
\t          tabulation             (HT, TAB)
\n          nouvelle ligne         (LF, NL)
\r          retour chariot         (CR)
\f          page suivante          (FF)
\a          alarme (bip)           (BEL)
\e          escape                 (ESC)
\033        caractère en octal     (ESC)
\x1B        caractère hexadécimal  (ESC)
\x{236a}    caractère hexadécimal long (SMILEY)
\c[         caractère de contrôle  (ESC)
\N{nom}     caractère Unicode nommé
```

Note : la séquence \v (pour une tabulation verticale - VT - ASCII 11) qui existe en C et dans d'autres langages n'est pas reconnue en Perl.

Les séquences d'échappement suivantes sont disponibles dans les constructions qui font de l'interpolation mais pas dans les translittérations :

```
\l          convertit en minuscule le caractère suivant
\u          convertit en majuscule le caractère suivant
\L          convertit en minuscule jusqu'au prochain \E
\U          convertit en majuscule jusqu'au prochain \E
\E          fin de modification de casse
\Q          désactive les méta-caractères de motif jusqu'au prochain \E
```

Si `use locale` est actif, la table de conversion majuscules/minuscules utilisée par `\l`, `\L`, `\u` et `\U` est celle du locale courant. Voir *perllocale*. En Unicode (par exemple avec `\N{}` ou avec des caractères larges codés en hexadécimal avec une valeur supérieure à 0x100), la table de conversion majuscules/minuscules utilisée par `\l`, `\L`, `\u` et `\U` est définie par les conventions Unicode. Pour la documentation de `\N{nom}`, voir *charnames*.

Tous les systèmes utilisent le `"\n"` virtuel pour représenter une terminaison de ligne appelée "newline" ou "nouvelle ligne". Il n'existe pas de caractère physique invariant pour représenter ce caractère "newline". C'est une illusion qu'essayent conjointement de maintenir le système d'exploitation, les pilotes de périphériques, la bibliothèque C et Perl. Tous les systèmes ne lisent pas `"\r"` comme le CR ASCII ni `"\n"` comme le LF ASCII. Par exemple, sur Mac, ils sont inversés et sur des systèmes sans terminaison de ligne, écrire `"\n"` peut ne produire aucun donnée. En général, utilisez `"\n"` lorsque vous pensez "newline" pour votre système mais utilisez le littéral ASCII quand voulez un caractère exact. Par exemple, de nombreux protocoles réseaux attendent et préfèrent un CR+LF (`"\015\012"` ou `"\cJ\cM"`) comme terminaison de ligne. La plupart acceptent un simple `"\012"` et ne tolèrent que très rarement un simple `"\015"`. Si vous prenez l'habitude d'utiliser `"\n"` sur les réseaux, vous aurez des problèmes un jour.

Dans les constructions qui font appel à de l'interpolation, les variables commençant par `"$"` ou `"@"` sont interpolées. Les variables utilisant des indices comme dans `$a[3]` ou `$href->{key}[0]` sont aussi interpolées comme le sont les tranches de tableaux ou de tables de hachage. Par contre, les appels de méthodes comme `$obj->meth` ne le sont pas.

Un tableau (ou une tranche de tableau) est interpolé par la suite de valeurs dans l'ordre séparées par la valeur de `$"`, ce qui est équivalent à `join $", @tableau`. Les tableaux dont le nom utilisent des caractères spéciaux comme `@+` ne sont interpolés que si on entoure le nom par des accolades `@{+}`.

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

Les motifs sont sujets à un niveau supplémentaire d'interprétation en tant qu'expression rationnelle. C'est fait lors d'une seconde passe après l'interpolation des variables si bien qu'une expression rationnelle peut-être introduite dans un motif via une variable. Si ce n'est pas ce que vous voulez, utilisez `\Q` pour interpoler une variable littéralement.

Mis à part ce qui précède, il n'y pas de multiples niveaux d'interpolation. En particulier et contrairement à ce qu'attendraient des programmeurs shell, les accents graves (ou backticks) *NE* sont *PAS* interpolées à l'intérieur des guillemets et les apostrophes n'empêchent pas l'évaluation des variables à l'intérieur des guillemets.

2.29 Opérateurs d'expression rationnelle

Nous discuterons ici des opérateurs style apostrophe qui implique une reconnaissance de motif ou une action s'y rapportant.

?MOTIF?

C'est la même chose qu'une recherche par `/motif/` sauf qu'elle n'est reconnue qu'une seule fois entre chaque appel à l'opérateur reset(). C'est une optimisation pratique si vous ne recherchez que la première occurrence de quelque chose dans chaque fichier ou groupes de fichiers par exemple. Seuls les motifs `??` locaux au package courant sont réinitialisés par reset().

```
while (<>) {
    if (?^$?) {
                        # ligne blanche entre en-tête et corps
    }
} continue {
    reset if eof;       # réinitialisation de ?? pour le fichier suivant
}
```

Cet usage est plus ou moins désapprouvé et pourrait être supprimé dans une version future de Perl. Peut-être vers l'année 2168.

m/MOTIF/cgimosx

/MOTIF/cgimosx

Effectue le recherche d'un motif d'expression rationnelle dans une chaîne et, dans un contexte scalaire, renvoie vrai en cas de succès ou faux sinon. Si aucune chaîne n'est spécifiée via l'opérateur =~ ou l'opérateur !~, c'est dans la chaîne $_ que s'effectue la recherche. (La chaîne spécifiée par =~ n'est pas nécessairement une lvalue – cela peut être le résultat de l'évaluation d'une expression.) Voir aussi *perlre*. Voir *perllocale* pour des informations supplémentaires qui s'appliquent lors de l'usage de `use locale`.

Les options (ou modificateurs) sont :

```
c   Ne pas réinitialiser la position de recherche lors d'un échec avec /g.
g   Recherche globale, c.-à-d. trouver toutes les occurrences.
i   Reconnaissance de motif indépendamment de la casse (majuscules/minuscules).
m   Traitement de chaîne comme étant multi-lignes.
o   Compilation du motif uniquement la première fois.
s   Traitement de la chaîne comme étant une seule ligne.
x   Utilisation des expressions rationnelles étendues.
```

Si "/" est le délimiteur alors le `m` initial est optionnel. Avec le `m` vous pouvez utiliser n'importe quelle paire de caractères ni alphanumériques ni blancs comme délimiteur. C'est particulièrement pratique pour les chemins d'accès Unix qui contiennent des "/" afin d'éviter le LTS (leaning toothpick syndrome). Si "?" est le délimiteur alors la règle "ne-marche-qu-une-fois" de `?MOTIF?` s'applique. Si "'" est le délimiteur alors aucune interpolation n'est effectuée sur le MOTIF.

MOTIF peut contenir des variables qui seront interpolées (et le motif recompilé) chaque fois que la recherche du motif est effectuée sauf si le délimiteur choisi est une apostrophe. (Remarquez que `$(`, `$)` et `$|` ne peuvent pas être interpolés puisqu'ils ressemblent à des tests de fin de chaîne.) Si vous voulez qu'un motif ne soit compilé qu'une seule fois, ajoutez `/o` après le dernier délimiteur. Ceci évite des coûteuses recompilations lors de l'exécution et c'est utile lorsque le valeur interpolée ne doit pas changer pendant la durée de vie du script. Par contre, ajouter `/o` suppose que vous ne changerez pas la valeur des variables présentes dans le motif. Si vous les changez, Perl ne s'en apercevra même pas. Voir aussi qr/CHAINE/imosx (§**??**).

Si l'évaluation du MOTIF est la chaîne vide, la dernière expression rationnelle *reconnue* est utilisée à la place. Dans ce cas, seuls les modificateurs `g` et `c` du motif vide sont pris en compte - les autres modificateurs seront ceux de

l'expression rationnelle utilisée. Si aucun motif n'a été précédemment reconnu, le motif utilisé (silencieusement) sera le motif vide (qui est toujours reconnaissable).

Notez qu'il est possible que Perl confonde `//` (l'expression rationnelle vide) et `//` (l'opérateur défini-ou). Habituellement, Perl se débrouille assez bien pour cela mais quelques cas pathologiques peuvent se présenter, comme, par exemple, `$a///` (est-ce `($a) / (//)` ou `$a // /` ?) ou `print $fh //` (est-ce `print $fh(//` ou `print($fh //` ?). Dans tous ces exemples, Perl choisira l'opérateur défini-ou. Pour choisir l'expression rationnelle vide, il vous suffit d'ajouter des parenthèses ou un espace pour lever l'ambiguïté ou encore préfixer votre expression rationnelle vide par un `m` (`//` devient donc `m//`).

Si l'option `/g` n'est pas utilisée, dans un contexte de liste, `m//` retourne une liste constituée de tous les sous-motifs reconnus par des parenthèses dans le motif (c.-à-d. `$1`, `$2`, `$3`...). (Remarquez que `$1`, `$2`, etc. sont aussi mises à jours ce qui diffère du comportement de Perl 4.) Lorsqu'il n'y a pas de parenthèses dans le motif, la valeur retournée est la liste `(1)` en cas de succès. Qu'il y ait ou non de parenthèses, une liste vide est retournée en cas d'échec.

Exemples :

```
open(TTY, '/dev/tty');
<TTY> =~ /^y/i && foo();    # appel de foo si désiré

if (/Version: *([0-9.]*)/) { $version = $1; }

next if m#^/usr/spool/uucp#;

# le grep du pauvre
$arg = shift;
while (<>) {
    print if /$arg/o;       # compile une seule fois
}

if (($F1, $F2, $Etc) = ($foo =~ /^(\S+)\s+(\S+)\s*(.*)/))
```

Ce dernier exemple découpe $foo en trois parties (le deux premiers mots et le reste de la ligne) et les affecte à $F1, $F2 et $Etc. La condition est vraie si au moins une des variables est affectée, c.-à-d. si le motif est reconnu.

Le modificateur `/g` spécifie une recherche globale du motif – c'est à dire la recherche d'autant de correspondances que possible dans la chaîne. Le comportement dépend du contexte. Dans un contexte de liste, c'est la liste de toutes les sous-chaînes reconnues par les sous-motifs (entre parenthèses) du motif qui est retournée. En l'absence de parenthèses, c'est la liste de toutes les chaînes correspondant au motif qui est retournée, comme si il y avait des parenthèses autour du motif lui-même.

Dans un contexte scalaire, chaque exécution de `m//g` trouve la prochaine correspondance et retourne vrai si il y a correspondance et faux si il n'y en a plus. La position après la dernière correspondance peut-être lue et modifiée par la fonction pos(); voir pos in *perlfunc*. Normalement, un échec de la recherche réinitialise la position de recherche au début de la chaîne sauf si vous ajoutez le modificateur `/c` (par ex. `m//gc`). La modification de la chaîne sur laquelle à lieu la recherche réinitialise aussi la position de recherche.

Vous pouvez mélanger des reconnaissances `m//g` avec des `m/\G.../g` où `\G` est l'assertion de longueur nulle qui est reconnue à la position exacte où, si elle existe, s'est arrêtée la précédente recherche `m//g`. Sans le modificateur `/g`, l'assertion pourra encore s'ancrer sur pos(), mais cette reconnaissance n'est évidemment essayée qu'une seule fois. L'application de `\G` sans `/g` sur une chaîne de caractères sur laquelle aucune reconnaissance utilisant `/g` n'a encore été faite, revient à utiliser l'assertion `\A` pour reconnaître le début de la chaîne. Notez aussi que, actuellement, `\G` ne fonctionne bien que si il est utilisé au tout début du motif.

Exemples :

```
# contexte de liste
($one,$five,$fifteen) = (`uptime` =~ /(\d+\.\d+)/g);

# contexte scalaire
$/ = "";
while (defined($paragraph = <>)) {
    while ($paragraph =~ /[a-z]['")]*[.!?]+['")]*\s/g) {
        $sentences++;
    }
}
print "$sentences\n";

# utilisation de m//gc avec \G
$_ = "ppooqppqq";
while ($i++ < 2) {
```

```
    print "1: '";
    print $1 while /(o)/gc; print "', pos=", pos, "\n";
    print "2: '";
    print $1 if /\G(q)/gc;  print "', pos=", pos, "\n";
    print "3: '";
    print $1 while /(p)/gc; print "', pos=", pos, "\n";
}
print "Final: '$1', pos=",pos,"\n" if /\G(.)/;
```

Le dernier exemple devrait afficher :

```
1: 'oo', pos=4
2: 'q', pos=5
3: 'pp', pos=7
1: '', pos=7
2: 'q', pos=8
3: '', pos=8
Final: 'q', pos=8
```

Remarquez que la reconnaissance finale reconnaît `q` au lieu de `p`, comme l'aurait fait une reconnaissance sans `\G`. Notez aussi que cette reconnaissance finale ne met pas à jour `pos` (`pos` n'est mis à jour qu'avec `/g`). Si vous obtenez un `p` lors de cette reconnaissance finale, cela indique que vous utilisez une vieille version de Perl (avant 5.6.0).

Une construction idiomatique pratique pour des analyseurs à la `lex` est `/\G.../gc`. Vous pouvez combiner plusieurs expressions rationnelles de ce type pour traiter une chaîne partie par partie en faisant différentes actions selon l'expression qui est reconnue. Chaque expression essaye de correspondre là où la précédente s'est arrêtée.

```
$_ = <<'EOL';
     $url = new URI::URL "http://www/";   die if $url eq "xXx";
EOL
LOOP:
   {
     print(" chiffres"),       redo LOOP if /\G\d+\b[,.;]?\s*/gc;
     print(" minuscule"),      redo LOOP if /\G[a-z]+\b[,.;]?\s*/gc;
     print(" MAJUSCULE"),      redo LOOP if /\G[A-Z]+\b[,.;]?\s*/gc;
     print(" Capitalisé"),     redo LOOP if /\G[A-Z][a-z]+\b[,.;]?\s*/gc;
     print(" MiXtE"),          redo LOOP if /\G[A-Za-z]+\b[,.;]?\s*/gc;
     print(" alphanumérique"), redo LOOP if /\G[A-Za-z0-9]+\b[,.;]?\s*/gc;
     print(" autres"),         redo LOOP if /\G[^A-Za-z0-9]+/gc;
     print ". C'est tout !\n";
   }
```

Voici la sortie (découpée en plusieurs lignes) :

```
autres minuscule autres minuscule MAJUSCULE autres
MAJUSCULE autres minuscule autres minuscule autres
minuscule minuscule autres minuscule minuscule autres
MiXtE autres. C'est tout !
```

q/CHAINE/

'CHAINE'

Une apostrophe pour une chaîne littérale. Un backslash (une barre oblique inverse) représente un backslash sauf si il est suivi par le délimiteur ou par un autre backslash auquel cas le délimiteur ou le backslash est interpolé.

```
$foo = q!I said, "You said, 'She said it.'"!;
$bar = q('This is it.');
$baz = '\n';                # une chaîne de deux caractères
```

qq/CHAINE/

"CHAINE"

Des guillemets pour une chaîne interpolée.

```
$_ .= qq
 (*** La ligne précédente contient le gros mot "$1".\n)
           if /\b(tcl|java|python)\b/i;      # :-)
$baz = "\n";                # une chaîne d'un caractère
```

qr/CHAINE/imosx

Cette opérateur considère *CHAINE* comme une expression rationnelle (et la compile éventuellement). *CHAINE* est interpolée de la même manière que *MOTIF* dans `m/MOTIF/`. Si "'" est utilisé comme délimiteur, aucune interpolation de variables n'est effectuée. Renvoie une expression Perl qui peut être utilisée à la place de l'expression `/CHAINE/imosx`.

Par exemple :

```
$rex = qr/ma.CHAINE/is;
s/$rex/foo/;
```

est équivalent à :

```
s/ma.CHAINE/foo/is;
```

Le résultat peut être utilisé comme motif dans une recherche de correspondance :

```
$re = qr/$motif/;
$string =~ /foo${re}bar/;   # peut être interpolée dans d'autres motifs
$string =~ $re;             # ou utilisée seule
$string =~ /$re/;           # ou de cette manière
```

Puisque Perl peut compiler le motif lors de l'exécution de l'opérateur qr(), l'utilisation de qr() peut augmenter les performances dans quelques situations, notamment si le résultat de qr() est utilisé seul :

```
sub match {
  my $patterns = shift;
  my @compiled = map qr/$_/i, @$patterns;
  grep {
      my $success = 0;
      foreach my $pat @compiled {
          $success = 1, last if /$pat/;
      }
      $success;
  } @_;
}
```

La précompilation du motif en une représentation interne lors du qr() évite une recompilation à chaque fois qu'une recherche `/$pat/` est tentée. (Notez que Perl a de nombreuses autres optimisations internes mais aucune n'est déclenchée dans l'exemple précédent si nous n'utilisons pas l'opérateur qr().)

Les options sont :

```
i   Motif indépendant de la casse (majuscules/minuscules).
m   Traitement de chaîne comme étant multi-lignes.
o   Compilation du motif uniquement la première fois.
s   Traitement de la chaîne comme étant une seule ligne.
x   Utilisation des expressions rationnelles étendues.
```

Voir *perlre* pour de plus amples informations sur la syntaxe correcte de CHAINE et pour une description détaillée de la sémantique des expressions rationnelles.

qx/CHAINE/

‘CHAINE‘

Une chaîne qui est (éventuellement) interpolée puis exécutée comme une commande système par `/bin/sh` ou équivalent. Les jokers, tubes (pipes) et redirections sont pris en compte. L'ensemble de la sortie standard de la commande est renvoyé ; la sortie d'erreur n'est pas affectée. Dans un contexte scalaire, le résultat est retourné comme une seule chaîne (potentiellement multi-lignes) ou undef si la commande échoue. Dans un contexte de liste, le résultat est une liste de lignes (selon la définition des lignes donnée par $/ ou $INPUT_RECORD_SEPARATOR) ou la liste vide si la commande échoue.

Puisque les apostrophes inverses (backticks) n'affectent pas la sortie d'erreur, il vous faut utiliser la (les) syntaxe(s) de redirection du shell (en supposant qu'elle(s) existe(nt)) afin de capter les erreurs. Pour récupérer les sorties STDOUT et STDERR d'une commande :

```
$output = ‘cmd 2>&1‘;
```

Pour récupérer STDOUT et faire disparaître STDERR :

```
$output = `cmd 2>/dev/null`;
```

Pour récupérer STDERR et faire disparaître STDOUT (l'ordre est ici très important) :

```
$output = `cmd 2>&1 1>/dev/null`;
```

Pour échanger STDOUT et STDERR afin de récupérer STDERR tout en laissant STDOUT s'afficher normalement :

```
$output = `cmd 3>&1 1>&2 2>&3 3>&-`;
```

Pour récupérer STDOUT et STDERR séparément, il est plus simple de les rediriger séparément vers des fichiers qui seront lus lorsque l'exécution de la commande sera terminée :

```
system("program args 1>program.stdout 2>program.stderr");
```

L'utilisation de l'apostrophe comme délimiteur protège la commande de l'interpolation normalement effectuée par Perl sur les chaînes entre guillemets. La chaîne est passée tel quelle au shell :

```
$perl_info  = qx(ps $$);            # Le $$ de Perl
$shell_info = qx'ps $$';            # Le $$ du nouveau shell
```

Remarquez que la manière dont la chaîne est évaluée est entièrement dépendante de l'interpréteur de commandes de votre système. Sur la plupart des plates-formes, vous aurez à protéger les méta-caractères du shell si vous voulez qu'ils soient traités littéralement. En pratique, c'est difficile à faire, surtout qu'il n'est pas toujours facile de savoir quels caractères doivent être protégés. Voir *perlsec* pour un exemple propre et sûre d'utilisation manuelle de fork() et exec() pour émuler proprement l'utilisation des backticks.

Sur certaines plates-formes (particulièrement celles style DOS) le shell peut ne pas être capable de gérer des commandes multi-lignes. Dans ce cas, l'usage de passages à la ligne dans la chaîne de commande ne donne pas ce que vous voulez. Vous pouvez évaluez plusieurs commandes sur une seule et même ligne et les séparant par le caractère de séparation de commandes si votre shell le supporte (par ex. `;` pour la plupart des shells Unix; `&` sur le `cmd` shell de Windows NT).

Depuis la version v5.6.0, Perl tente de vider les tampons de tous les fichiers ouverts en écriture avant de lancer la commande mais cela n'est pas supporté sur toutes les plates-formes (voir *perlport*). Pour être plus sûr, vous devriez positionné la variable `$|` ($AUTOFLUSH en anglais) ou appelé la méthode `autoflush()` des objets `IO::Handle` pour chacun des descripteurs ouverts.

N'oubliez pas que certains shells ont quelques restrictions sur la longueur de la ligne de commande. Vous devez vous assurer que votre chaîne n'excède pas cette limite même après interpolation. Voir les notes spécifiques à votre plate-forme pour plus de détails sur votre environnement particulier.

L'utilisation de cet opérateur peut aboutir à des programmes difficiles à porter puisque les commandes shells appelées varient d'un système à l'autre et peuvent parfois ne pas exister du tout. À titre d'exemple, la commande `type` du shell POSIX est très différente de la commande `type` sous DOS. Cela ne signifie pas qu'il vous faut à tout prix éviter cet opérateur lorsque c'est le bon moyen de faire ce que vous voulez. Perl a été conçu pour être un "glue language"... La seule chose qui compte c'est que vous sachiez ce que vous faites.

Voir Opérateurs d'E/S (§**??**) pour des plus amples informations.

qw/CHAINE/

Retourne la liste des mots extraits de CHAINE en utilisant les espaces comme délimiteurs de mots. On peut le considérer comme équivalent à :

```
split(' ', q/CHAINE/);
```

avec comme différence que la liste est générée lors de la compilation, et que, dans un contexte scalaire, c'est le dernier élément de la liste qui est retourné. Donc <l'expression :>

```
qw(foo bar baz)
```

est sémantiquement équivalente à la liste :

```
'foo', 'bar', 'baz'
```

Quelques exemples fréquemment rencontrés :

```
use POSIX qw( setlocale localeconv )
@EXPORT = qw( foo bar baz );
```

Une erreur assez commune consiste à séparer les mots par des virgules ou à mettre des commentaires dans une `qw`-chaîne multi-lignes. C'est la raison pour laquelle `use warnings` ou l'option `-w` (c'est à dire la variable `$^W`) produit un message d'avertissement lorsque CHAINE contient le caractère "," ou le caractère "#".

s/MOTIF/REMPLACEMENT/egimosx

Recherche le motif dans une chaîne puis, s'il est trouvé, le substitue par le texte de REMPLACEMENT et retourne finalement le nombre de substitutions effectuées. Sinon, renvoie faux (en l'espèce, la chaîne vide).

Si aucune chaîne n'est spécifiée via =~ ou !~, la recherche et la substitution s'appliquent à la variable $_. (La chaîne spécifiée par =~ doit être une variable scalaire, un élément d'un tableau ou d'une table de hachage ou une affectation de l'un ou de l'autre... en un mot, une lvalue.)

Si le délimiteur choisi est l'apostrophe, aucune interpolation n'est effectuée ni sur MOTIF ni sur REMPLACEMENT. Sinon, si MOTIF contient un $ qui ressemble plus à une variable qu'à un test de fin de chaîne, la variable sera interpolée dans le motif lors de l'exécution. Si vous voulez que le motif ne soit compilé qu'une seule fois la première fois que la variable est interpolée, utilisez l'option `/o`. Si l'évaluation du motif est la chaîne nulle, la dernière expression rationnelle reconnue est utilisée à la place. Voir *perlre* pour de plus amples informations à ce sujet. Voir *perllocale* pour des informations supplémentaires qui s'appliquent lors de l'usage de `use locale`.

Les options sont :

```
e   Évaluez la partie droite comme une expression.
g   Substitution globale, c.-à-d. toutes les occurrences.
i   Motif indépendant de la casse (majuscules/minuscules).
m   Traitement de chaîne comme étant multi-lignes.
o   Compilation du motif uniquement la première fois.
s   Traitement de la chaîne comme étant une seule ligne.
x   Utilisation des expressions rationnelles étendues.
```

N'importe quel délimiteur (ni blanc ni alphanumérique) peut remplacer les barres obliques (slash). Si l'apostrophe est utilisée, aucune interpolation n'est effectuée sur la chaîne de remplacement (par contre, le modificateur `/e` passe outre). Au contraire de Perl 4, Perl 5 considère l'accent grave (backtick) comme un délimiteur normal ; le texte de remplacement n'est pas évalué comme une commande. Si le MOTIF est délimité par des caractères fonctionnant en paire (comme les parenthèses), le texte de REMPLACEMENT a sa propre paire de délimiteurs qui peuvent être ou non des caractères fonctionnant par paire, par ex. `s(foo)(bar)` ou `s<foo>/bar/`. L'option `/e` implique que la partie remplacement sera interprétée comme une expression Perl à part entière et donc évaluée comme telle. Par contre, sa validité syntaxique est évaluée lors de la compilation. Un seconde option `e` provoquera l'évaluation de la partie REMPLACEMENT avant son exécution en tant qu'expression Perl.

Exemples:

```
s/\bgreen\b/mauve/g;                # ne modifie pas wintergreen

$path =~ s|/usr/bin|/usr/local/bin|;

s/Login: $foo/Login: $bar/; # motif dynamique

($foo = $bar) =~ s/this/that/;      # recopie puis modification

$count = ($paragraph =~ s/Mister\b/Mr./g);  # calcul du nombre de substitution

$_ = 'abc123xyz';
s/\d+/$&*2/e;               # produit 'abc246xyz'
s/\d+/sprintf("%5d",$&)/e;  # produit 'abc  246xyz'
s/\w/$& x 2/eg;             # produit 'aabbcc  224466xxyyzz'

s/%(.)/$percent{$1}/g;      # pas de /e
s/%(.)/$percent{$1} || $&/ge;       # une expression donc /e
s/^=(\w+)/&pod($1)/ge;      # appel de fonction

# expansion des variables dans $_, mais dynamique uniquement
# en utilisant le déréférencement symbolique
s/\$(\w+)/${$1}/g;

# Ajoute un à chaque nombre présent dans la chaîne
s/(\d+)/1 + $1/eg;

# Ceci développe toutes les variables scalaires incluses
# (même les lexicales) dans $_ : $1 est tout d'abord interpolé
# puis évalué
s/(\$\w+)/$1/eeg;
```

```
# Suppression des commentaires C (presque tous).
$program =~ s {
    /\*     # Reconnais le début du commentaire
    .*?     # Reconnais un minimum de caractères
    \*/     # Reconnais la fin de commentaire
} []gsx;

s/^\s*(.*?)\s*$/$1/;  # suppression des espaces aux extrémités de $_ (couteux)

for ($variable) {     # suppression des espaces aux extrémités de $variable (efficace)
    s/^\s+//;
    s/\s+$//;
}

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

Remarquez l'usage de $ à la place de \ dans le dernier exemple. À l'inverse de **sed**, nous utilisons la forme \<*digit*> uniquement dans la partie gauche. Partout ailleurs, c'est $<*digit*>.

Dans certains cas, il ne suffit pas de mettre `/g` pour modifier toutes les occurrences. Voici quelques cas communs :

```
# placer des virgules correctement dans un entier
# (NdT: en français, on mettrait des espaces)
1 while s/(.*\d)(\d\d\d)/$1,$2/g;      # perl4
1 while s/(\d)(\d\d\d)(?!\d)/$1,$2/g;  # perl5

# remplacement des tabulations par 8 espaces
1 while s/\t+/' ' x (length($&)*8 - length($`)%8)/e;
```

tr/LISTERECHERCHEE/LISTEREMPLACEMENT/cds

y/LISTERECHERCHEE/LISTEREMPLACEMENT/cds

Translittère chacune des occurrences des caractères de la liste recherchée par le caractère correspondant de la liste de remplacement. Cette opérateur retourne le nombre de caractères remplacés ou supprimés. Si aucune chaîne n'est spécifié via =~ ou !~, la translittération s'applique à $_. (La chaîne spécifiée par =~ doit être une variable scalaire, un élément d'un tableau ou d'un table de hachage ou une affectation de l'un ou de l'autre... en un mot, une lvalue.)

Un intervalle de caractères peut-être spécifié grâce à un tiret. Donc `tr/A-J/0-9/` effectue les mêmes translittérations que `tr/ACEGIBDFHJ/0246813579/`. Pour les adeptes de **sed**, `y` est fourni comme un synonyme de `tr`. Si la liste recherchée est délimitée par des caractères fonctionnant par paire (comme les parenthèses) alors la liste de remplacement a ses propres délimiteurs, par ex. `tr[A-Z][a-z]` ou `tr(+\-*/)/ABCD/`.

Notez que `tr` ne reconnaît pas les classes de caractères des expressions rationnelles telles que `\d` ou `[:lower:]`. L'opérateur `tr` n'est donc pas équivalent à l'utilitaire tr(1). Si vous devez effectuer des translittérations majuscules/minuscules, voyez du côté de lc in *perlfunc* et uc in *perlfunc*, et, en général, du côté de l'opérateur `s` si vous avez besoin des expressions rationnelles.

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

Les options (ou modificateurs) :

```
c   Complémente la SEARCHLIST.
d   Efface les caractères trouvés mais non remplacés.
s   Agrège les caractères de remplacement dupliqués.
```

Si le modificateur `/c` est utilisé, c'est le complément de la liste recherchée qui est utilisé. Si le modificateur `/d` est spécifié, tout caractère spécifié dans LISTERECHERCHEE et sans équivalent dans LISTEREMPLACEMENT est effacé. (Ceci est tout de même plus flexible que le comportement de certains programme **tr** qui efface tout ce qui est dans LISTERECHERCHEE, point!) Si le modificateur `/s` est spécifié, les suites de caractères qui sont translittérés par le même caractère sont agrégées en un seul caractère.

Si le modificateur `/d` est utilisé, LISTEREMPLACEMENT est toujours interprété exactement comme spécifié. Sinon, si LISTEREMPLACEMENT est plus court que LISTERECHERCHEE, le dernier caractère est répété autant de fois que nécessaire pour obtenir la même longueur. Si LISTEREMPLACEMENT est vide, LISTERECHERCHEE est utilisé à la place. Ce dernier point est très pratique pour comptabiliser les occurrences d'une classe de caractères ou pour agréger les suites de caractères d'une classe.

Exemples :

```
$ARGV[1] =~ tr/A-Z/a-z/;    # tout en minuscule

$cnt = tr/*/*/;             # compte les étoiles dans $_

$cnt = $sky =~ tr/*/*/;     # compte les étoiles dans $sky

$cnt = tr/0-9//;            # compte les chiffres dans $_

tr/a-zA-Z//s;               # bookkeeper -> bokeper

($HOST = $host) =~ tr/a-z/A-Z/;

tr/a-zA-Z/ /cs;             # remplace tous les non alphanumériques
                            # par un seul espace

tr [\200-\377]
   [\000-\177];             # efface le 8ème bit.
```

Si plusieurs caractères de translittération sont donnés pour un caractère, seul le premier est utilisé :

```
tr/AAA/XYZ/
```

translittérera tous les A en X.

Remarquez que puisque la table de translittération est construite lors de la compilation, ni LISTERECHERCHEE ni LISTEREMPLACEMENT ne sont sujettes à l'interpolation de chaîne entre guillemets. Cela signifie que si vous voulez utiliser des variables, vous devez utiliser eval() :

```
eval "tr/$oldlist/$newlist/";
die $@ if $@;

eval "tr/$oldlist/$newlist/, 1" or die $@;
```

<<EOF

Il existe une syntaxe de saisie de chaînes de caractères sur plusieurs lignes basée sur la syntaxe du "here-document" du shell. À la suite d'un <<, vous placez une chaîne de terminaison du texte et toutes les lignes à partir de celle-ci et jusqu'à cette chaîne de terminaison constitueront votre texte. La chaîne de terminaison peut-être un simple identificateur (un mot) ou un texte entre guillemets ou entre apostrophes. Le choix entre les guillemets ou les apostrophes déterminera si l'interpolation sera active ou non, comme pour les chaînes de caractères classiques. Un identificateur aura le même comportement que si il était entre guillemets (et donc l'interpolation sera active). Il ne doit pas y avoir d'espace entre le << et l'identificateur si il n'est pas entre guillemets ou entre apostrophes. (Si il y a un espace, il sera considéré comme l'identificateur vide, qui est valide, et sera reconnu dès la première ligne vide.) Le texte de terminaison doit apparaître tel quel (sans guillemets ni apostrophes ni aucun espace autour) en tant que ligne finale.

```
   print <<EOF;
The price is $Price.
EOF

   print << "EOF"; # Pareil que ci-dessus
The price is $Price.
EOF

   print << `EOC`; # exécute les commandes
echo hi there
echo lo there
EOC

   print <<"foo", <<"bar"; # vous pouvez les empiler
I said foo.
foo
I said bar.
bar

   myfunc(<< "THIS", 23, <<'THAT');
Here's a line
or two.
THIS
and here's another.
THAT
```

N'oubliez pas de mettre un point-virgule à la fin de votre instruction sinon Perl pourrait croire que vous essayez de faire ce qui suit :

```
   print <<ABC
179231
ABC
   + 20;
```

Si vous souhaitez que vos here-docs soient indentés comme le reste de votre code, vous devrez supprimer vous-même les espaces supplémentaires en début de ligne :

```
($quote = <<'FINIS') =~ s/^\s+//gm;
   The Road goes ever on and on,
   down from the door where it began.
FINIS
```

Si vous utilisez un here-doc à l'intérieur d'une construction utilisant des délimiteurs, comme `s///eg`, votre texte doit commencer après le délimiteur final. Donc, n'écrivez pas :

```
s/this/<<E . 'that'
the other
E
 . 'more '/eg;
```

mais écrivez :

```
s/this/<<E . 'that'
 . 'more '/eg;
the other
E
```

Si votre chaîne de terminaison est la toute dernière ligne de votre programme, soyez sûr qu'elle se termine par un saut de ligne sinon Perl produira l'avertissement suivant : **Can't find string terminator "END" anywhere before EOF...**.

De plus, les règles d'interpolation classiques des chaînes de caractères (comme `q{}`, `qq{}` et autres) ne sont pas applicables à la chaîne de terminaison elle-même sauf celle permettant d'insérer le délimiteur lui-même.

```
print << "abc\"def";
testing...
abc"def
```

Pour finir, la chaîne de terminaison ne peut pas être sur plusieurs lignes. En d'autres termes, ça ne peut être qu'une chaîne littérale. Respectez cela et tout ira bien.

2.30 Les détails sordides de l'interprétation des chaînes

Face à quelque chose qui pourrait avoir différentes interprétations, Perl utilise le principe du **FCQJP** (qui signifie Fait Ce Que Je Pense) pour choisir l'interprétation la plus probable. Cette stratégie fonctionne tellement bien que les utilisateurs de Perl soupçonnent rarement l'ambiguïté de ce qu'ils écrivent. Par contre, de temps à autre, l'idée que se fait Perl diffère de ce que l'auteur du programme pensait.

L'objet de cette partie est de clarifier la manière dont Perl interprète les chaînes. La raison la plus fréquente pour laquelle quelqu'un a besoin de connaître tous ces détails est l'utilisation d'une expression rationnelle "velue". Par contre, les premiers pas de l'interprétation d'une chaîne sont les mêmes pour toutes les constructions de chaînes. Ils sont donc expliqués ensemble.

L'étape la plus importante de l'interprétation des chaînes dans Perl est la première exposée ci-dessous : lors d'une interprétation de chaînes, Perl cherche d'abord la fin de la construction puis il interprète le contenu de cette construction. Si vous comprenez cette règle, vous pouvez sauter les autres étapes en première lecture. Contrairement à cette première étape, les autres étapes contredisent beaucoup moins souvent les attentes de l'utilisateur.

Quelques-unes des passes exposées plus loin sont effectuées simultanément. Mais comme le résultat est le même, nous les considérerons successivement une par une. Selon la construction, Perl effectue ou non certaines passes (de une à cinq) mais elles sont toujours appliquées dans le même ordre.

Trouver la fin

La première passe consiste à trouver la fin de la construction qui peut être la suite de caractères `"\nEOF\n"` d'une construction `<<EOF`, le `/` qui termine une construction `qq/`, le `]` qui termine une construction `qq[` ou le `>` qui termine un fileglob commencé par `<`.

Lors de la recherche d'un caractère délimiteur non appairé comme `/`, les combinaisons comme `\\` et `\/` sont sautées. Lors de la recherche d'un délimiteur appairé comme `]`, les combinaisons `\\`, `\]` et `\[` sont sautées ainsi que les constructions imbriquées `[ ]`. Lors de la recherche d'un délimiteur constitué d'une suite de caractères, rien n'est omis.

Pour les constructions en trois parties (`s///`, `y///` et `tr///`), la recherche est répétée une fois supplémentaire.

Lors de cette recherche, aucune attention n'est accordée à la sémantique de la construction, donc :

```
"$hash{"$foo/$bar"}"
```

ou :

```
m/
  bar       #  Ce n'est PAS un commentaire, ce slash / termine m// !
 /x
```

ne constituent donc pas des constructions légales. L'expression se termine au premier `"` ou `/` et le reste apparaît comme une erreur de syntaxe. Remarquez que puisque le slash qui termine `m//` est suivi pas un `ESPACE`, ce n'est pas une constructions `m//x` mais bien un constructions `m//` sans le modificateur `/x`. Donc `#` est interprété comme un `#` littéral.

Notez aussi que rien n'est fait pour détecter une construction du type `\c\` durant cette recherche. Donc le second `\` dans `qq/\c\/` sera vu comme le début de `\/` et le `/` qui suit ne sera pas reconnu comme délimiteur. En remplacement, vous pouvez utiliser `\034` ou `\x1c` à la fin des constructions délimitées.

Suppression des barres obliques inverses (backslash) précédents les délimiteurs

Lors de la seconde passe, le texte entre le délimiteur de départ et le délimiteur de fin est recopié à un endroit sûr et le `\` des combinaisons constituées par un `\` suivi du délimiteur (ou des délimiteurs si celui de départ diffère de celui de fin) est supprimé. Cette suppression n'a pas lieu pour des délimiteurs multi-caractères.

Remarquez que la combinaison `\\` est laissée telle quelle.

À partir de ce moment plus aucune information concernant le(s) délimiteur(s) n'est utilisée.

Interpolation

L'étape suivante est l'interpolation appliquée au texte isolé de ses délimiteurs. Il y a quatre cas différents.

`<<'EOF', m", s"', tr///, y///`

Aucune interpolation n'est appliquée.

`", q//`

La seule interpolation est la suppression du `\` des paires `\\`.

`"", `` , qq//, qx//, <file*glob>`

Les `\Q`, `\U`, `\u`, `\L`, `\l` (éventuellement associé avec `\E`) sont transformés en leur construction Perl correspondante et donc `"$foo\Qbaz$bar"` est transformé en `$foo . (quotemeta("baz" . $bar))` en interne. Les autres combinaisons constituées d'un `\` suivi d'un ou plusieurs caractères sont remplacées par le ou les caractères appropriés.

Soyez conscient que *tout ce qui est entre `\Q` et `\E`* est interpolé de manière classique. Donc `"\Q\\E"` ne contient pas de `\E` : il contient `\Q`, `\\` et `E` donc le résultat est le même que `"\\\\E"`. Plus généralement, la présence de backslash entre `\Q` et `\E` aboutit à des résultats contre-intuitifs. Donc `"\Q\t\E"` est converti en `quotemeta("\t")` qui est la même chose que `"\\\t"` (puisque TAB n'est pas alphanumérique). Remarquez aussi que :

```
$str = '\t';
return "\Q$str";
```

est peut-être plus proche de *l'intention* de celui qui écrit `"\Q\t\E"`.

Les scalaires et tableaux interpolés sont convertis en une série d'opérations de concaténation `join` et `.`. Donc `"$foo XXX '@arr'"` devient :

```
$foo . " XXX '" . (join $", @arr) . "'";
```

Toutes les étapes précédentes sont effectuées simultanément de la gauche vers la droite.

Puisque le résultat de "\Q STRING \E" a tous ses méta-caractères quotés, il n'y aucun moyen d'insérer littéralement un $ ou un @ dans une paire \Q\E : si il est protégé par \, un $ deviendra "\\\$" et sinon il sera interprété comme le début d'un scalaire a interpolé.

Remarquez aussi que l'interpolation de code doit décider où se termine un scalaire interpolé. Par exemple "a $b -> {c}" peut signifier :

```
"a " . $b . " -> {c}";
```

ou :

```
"a " . $b -> {c};
```

Dans la plupart des cas, le choix est de considérer le texte le plus long possible n'incluant pas d'espaces entre ses composants et contenant des crochets ou accolades bien équilibrés. Puisque le résultat peut-être celui d'un vote entre plusieurs estimateurs heuristiques, le résultat n'est pas strictement prévisible. Heureusement, il est habituellement correct pour les cas ambigus.

?RE?, /RE/, m/RE/, s/RE/foo/,

Le traitement des \Q, \U, \u, \L, \l et des interpolations est effectué quasiment de la même manière qu'avec les constructions qq// sauf que la substitution d'un \ suivi d'un ou plusieurs caractères spéciaux pour les expressions rationnelles (incluant \) n'a pas lieu. En outre, dans les constructions (?{BLOC}), (?# comment), et #-comment présentes dans les expressions rationnelles //x, aucun traitement n'est effectué. C'est le premier cas ou la présence de l'option //x est significative.

L'interpolation a quelques bizarreries : $|, $(et $) ne sont pas interpolés et les constructions comme $var[QQCH] peuvent être vues (selon le vote de plusieurs estimateurs différents) soit comme un élément d'un tableau soit comme $var suivi par une alternative RE. C'est là où la notation ${arr[$bar]} prend tout sons intérêt : /${arr[0-9]}/ est interprété comme l'élément -9 du tableau et pas comme l'expression rationnelle contenu dans $arr suivie d'un chiffre (qui est l'interprétation de /$arr[0-9]/). Puisqu'un vote entre différents estimateurs peut avoir lieu, le résultat n'est pas prévisible.

C'est à ce moment que la construction \1 est convertie en $1 dans le texte de remplacement de s/// pour corriger les incorrigibles adeptes de *sed* qui n'ont pas encore compris l'idiome. Un avertissement est émis si le pragma use warnings ou l'option **-w** (c'est à dire la variable $^W) est actif.

Remarquez que l'absence de traitement de \\ crée des restrictions spécifiques sur le texte après traitement : si le délimiteur est /, on ne peut pas obtenir \/ comme résultat de cette étape (/ finirait l'expression rationnelle, \/ serait transformé en / par l'étape précédente et \\/ serait laissé tel quel). Puisque / est équivalent à \/ dans une expression rationnelle, ceci ne pose problème que lorsque le délimiteur est un caractère spécial pour le moteur RE comme dans s*foo*bar*, m[foo], ou ?foo? ou si le délimiteur est un caractère alphanumérique comme dans :

```
m m ^ a \s* b mmx;
```

Dans l'expression rationnelle ci-dessus qui est volontairement obscure pour l'exemple, le délimiteur est m, le modificateur est mx et après la suppression des backslash, l'expression est la même que m/ ^ a \s* b /mx. Il y a de nombreuses raisons de vous encourager à ne choisir que des caractères non alphanumériques et non blancs comme séparateur.

Cette étape est la dernière pour toutes les constructions sauf pour les expressions rationnelles qui sont traitées comme décrit ci-après.

Interpolation des expressions rationnelles

Toutes les étapes précédentes sont effectuées lors de la compilation du code Perl alors que celle que nous décrivons ici l'est a priori lors de l'exécution (bien qu'elle soit parfois effectuée lors de la compilation pour des raisons d'optimisation). Après tous les pré-traitements précédents (et évaluation des éventuels concaténations, jointures, changements de casse et applications de quotemeta() déduits), la *chaîne* résultante est transmise au moteur RE pour compilation.

Ce qui se passe à l'intérieur du moteur RE est bien mieux expliqué dans *perlre* mais dans un souci de continuité, nous l'exposons un peu ici.

Voici donc une autre étape où la présence du modificateur //x est prise en compte. Le moteur RE explore la chaîne de gauche à droite et la convertit en un automate à états finis.

Les caractères "backslashés" sont alors remplacés par la chaîne correspondante (comme pour \{) ou génèrent des noeuds spéciaux dans l'automate à états finis (comme pour \b). Les caractères ayant un sens spécial pour le moteur RE (comme |) génèrent les noeuds ou les groupes de noeuds correspondants. Les commentaires (?#...) sont

ignorés. Tout le reste est converti en chaîne littérale à reconnaître ou est ignoré (par exemple les espaces et les commentaires `#` si le modificateur `//x` est présent).

Remarquez que le traitement de la construction `[...]` est effectué en utilisant des règles complètement différentes du reste de l'expression rationnelle. Le terminateur de cette construction est trouvé en utilisant les mêmes règles que celles utilisées pour trouver le terminateur d'une construction délimitée (comme `{}`). La seule exception est le `]` qui suit immédiatement le `[` et qui est considéré comme précédé d'un backslash. De manière similaire, la construction `(?{...})` n'est explorée que pour vérifier que les parenthèses, crochets et autres accolades sont bien équilibrés.

Il est possible d'inspecter la chaîne envoyée au moteur RE ainsi que l'automate à états finis résultant. Voir les arguments `debug`/`debugcolor` de la directive `use re` et/ou l'option Perl **-Dr** dans Options de Ligne de Commande in *perlrun*.

Optimisation des expressions rationnelles

Cette étape n'est citée que dans un souci de complétude. Puisque elle ne change en rien la sémantique, les détails de cette étape ne sont pas documentés et sont susceptibles d'évoluer. Cette étape est appliquée à l'automate à états finis généré lors des étapes précédentes.

C'est lors de cette étape que `split()` optimise silencieusement `/^/` en `/^/m`.

2.31 Opérateurs d'E/S

Il y a plusieurs opérateurs d'E/S (Entrée/Sortie) que vous devez connaître.

Une chaîne entourée d'apostrophes inversées (accents graves) subit tout d'abord une substitution des variables exactement comme une chaîne entre guillemets. Elle est ensuite interprétée comme une commande et la sortie de cette commande est la valeur du pseudo-littéral comme avec un shell. Dans un contexte scalaire, la valeur retournée est une seule chaîne constituée de toutes les lignes de la sortie. Dans un contexte de liste, une liste de valeurs est retournée, chacune des ces valeurs contenant une ligne de la sortie. (Vous pouvez utilisez `$/` pour utiliser un terminateur de ligne différent.) La commande est exécutée à chaque fois que le pseudo-littéral est évalué. La valeur du statut de la commande est retournée dans `$?` (voir *perlvar* pour l'interprétation de `$?`). À l'inverse de **csh**, aucun traitement n'est appliqué aux valeurs retournées – les passages à la ligne restent des passages à la ligne. A l'inverse de la plupart des shells, les apostrophes inversées n'empêchent pas l'interprétation des noms de variables dans la commande. Pour passer littéralement un $ au shell, vous devez le protéger en le préfixant par un backslash (barre oblique inversée). La forme générale des apostrophes inversées est `qx//`. (Puisque les apostrophes inversées impliquent toujours un passage par l'interprétation du shell, voir *perlsec* pour tout ce qui concerne la sécurité.)

Dans un contexte scalaire, évaluer un filehandle entre supérieur/inférieur produit le ligne suivante de ce fichier (saut à la ligne inclus si il y a lieu) ou `undef` à la fin du fichier. Lorsque `$/` a pour valeur `undef` (c.-à-d. le mode *file slurp*) et que le fichier est vide, `''` est retourné lors de la première lecture puis ensuite `undef`.

Normalement vous devez affecter cette valeur à une variable mais il y a un cas où une affectation automagique a lieu. Si et seulement si cette opérateur d'entrée est la seule chose présente dans la condition d'une boucle `while` ou `for(;;)` alors la valeur est automagiquement affectée à la variable `$_`. (Cela peut vous sembler bizarre, mais vous utiliserez de telles constructions dans la plupart des scripts Perl que vous écrirez.) La variable $_ n'est pas implicitement local-isée. Vous devrez donc le faire explicitement en mettant `local $_;`.

Les lignes suivantes sont toutes équivalentes :

```
while (defined($_ = <STDIN>)) { print; }
while ($_ = <STDIN>) { print; }
while (<STDIN>) { print; }
for (;<STDIN>;) { print; }
print while defined($_ = <STDIN>);
print while ($_ = <STDIN>);
print while <STDIN>;
```

et celle-ci a un comportement similaire mais sans utiliser $_ :

```
while (my $line = <STDIN>) { print $line }
```

Dans ces constructions, la valeur affectée (que ce soit automagiquement ou explicitement) est ensuite testée pour savoir si elle est définie. Ce test de définition évite les problèmes avec des lignes qui ont une valeur qui pourrait être interprétée comme fausse par perl comme par exemple "" ou "0" sans passage à la ligne derrière. Si vous voulez réellement tester la valeur de la ligne pour terminer votre boucle, vous devrez la tester explicitement :

```
while (($_ = <STDIN>) ne '0') { ... }
while (<STDIN>) { last unless $_; ... }
```

Dans tous les autres contextes booléens, *<filehandle>* sans un test explicite de définition (par `defined`) déclenchera un message d'avertissement si le pragma `use warnings` ou l'option `-w` (c'est à dire la variable `$^W`) est actif.

Les filehandles STDIN, STDOUT, et STDERR sont prédéfinis. (Les filehandles `stdin`, `stdout`, et `stderr` fonctionnent aussi exceptés dans les packages où ils sont interprétés comme des identifiants locaux.) Des filehandles supplémentaires peuvent être créés par la fonction open(). Voir *perlopentut* et open in *perlfunc* pour plus de détails.

Si <FILEHANDLE> est utilisé dans un contexte de liste, une liste constituée de toutes les lignes est retournée avec une ligne par élément de la liste. Il est facile d'utiliser de grande quantité de mémoire par ce moyen. Donc, à utiliser avec précaution.

<FILEHANDLE> peut aussi être écrit `readline(*FILEHANDLE)`. Voir readline in *perlfunc*.

Le filehandle vide <> est spécial et peut être utiliser pour émuler le comportement de **sed** et de **awk**. L'entrée de <> provient soit de l'entrée standard soit de tous les fichiers listés sur la ligne de commande. Voici comment ça marche : la première fois que <> est évalué, on teste le tableau @ARGV et s'il est vide alors `$ARGV[0]` est positionné à "-" qui lorsqu'il sera ouvert lira l'entrée standard. Puis le tableau @ARGV est traité comme une liste de nom de fichiers. La boucle :

```
while (<>) {
    ...                     # code pour chaque ligne
}
```

est équivalent au pseudo-code Perl suivant :

```
unshift(@ARGV, '-') unless @ARGV;
while ($ARGV = shift) {
    open(ARGV, $ARGV);
    while (<ARGV>) {
        ...         # code pour chaque ligne
    }
}
```

sauf qu'il est moins volumineux et qu'il marche réellement. Il décale vraiment le tableau @ARGV et stocke le nom du fichier courant dans la variable $ARGV. Il utilise aussi en interne le filehandle *ARGV* – <> est simplement un synonyme de <ARGV> qui est magique. (Le pseudo-code précédent ne fonctionne pas car il tient pas compte de l'aspect magique de <ARGV>.)

Vous pouvez modifier @ARGV avant la premier <> tant que vous y laissez la liste des noms de fichiers que vous voulez. Le numéro de ligne (`$.`) augmente exactement comme si vous aviez un seul gros fichier. Regardez l'exemple de `eof` pour savoir comment le réinitialiser à chaque fichier.

Si vous voulez affecter à @ARGV votre propre liste de fichiers, procédez de la manière suivante. La ligne suivante positionne @ARGV à tous les fichiers de type texte si aucun @ARGV n'est fourni :

```
@ARGV = grep { -f && -T } glob('*') unless @ARGV;
```

Vous pouvez même les positionner avec des commandes pipe (tube). Par exemple, la ligne suivante applique automatiquement **gzip** aux arguments compressés :

```
@ARGV = map { /\.(gz|Z)$/ ? "gzip -dc < $_ |" : $_ } @ARGV;
```

Si vous voulez passer des options à votre script, vous pouvez utiliser l'un des modules Getopts ou placer au début de votre script une boucle du type :

```
while ($_ = $ARGV[0], /^-/) {
    shift;
    last if /^--$/;
    if (/^-D(.*)/) { $debug = $1 }
    if (/^-v/)     { $verbose++  }
    # ...           # autres options
}
```

```
while (<>) {
    # ...           # code pour chaque ligne
}
```

Le symbole <> retournera `undef` à la fin des fichiers une seule fois. Si vous l'appelez encore une fois après, il supposera que vous voulez traiter une nouvelle liste @ARGV et, si vous n'avez pas rempli @ARGV, il traitera l'entrée STDIN.

Si la chaîne entre supérieur/inférieur est une simple variable scalaire (par ex. <$foo>) alors cette variable doit contenir le nom du filehandle à utiliser comme entrée ou son typeglob ou une référence vers un typeglob. Par exemple :

```
$fh = \*STDIN;
$line = <$fh>;
```

Si ce qui est entre supérieur/inférieur n'est ni un filehandle, ni une simple variable scalaire contenant un nom de filehandle, un typeglob ou une référence à un typeglob, il est alors interprété comme un motif de nom de fichier à appliquer et ce qui est retourné est soit la liste de tous les noms de fichiers ou juste le nom suivant dans la liste selon le contexte. La distinction n'est faite que par la syntaxe. Cela signifie que <$x> est toujours la lecture d'une ligne à partir d'un handle indirect mais que <$hash{key}> est toujours un motif de nom de fichiers. Tout cela parce que $x est une simple variable scalaire alors que `$hash{key}` ne l'est pas – c'est un élément d'une table de hachage. Même <$x > (notez l'espace supplémentaire) sera traité comme `glob("$x ")` et non comme `readline($x)`.

Comme pour les chaînes entre guillemets, un niveau d'interprétation est appliqué au préalable mais vous ne pouvez pas dire <$foo> parce que c'est toujours interprété comme un filehandle indirect (explications du paragraphe précédent). (Dans des versions antérieures de Perl, les programmeurs inséraient des accolades pour forcer l'interprétation comme un motif de nom de fichier: <${foo}>. De nos jours, il est considéré plus propre d'appeler la fonction interne explicitement par `glob($foo)` qui est probablement la meilleure façon de faire dans le premier cas.) Exemple :

```
while (<*.c>) {
    chmod 0644, $_;
}
```

est équivalent à :

```
open(FOO, "echo *.c | tr -s ' \t\r\f' '\\012\\012\\012\\012'|");
while (<FOO>) {
    chomp;
    chmod 0644, $_;
}
```

sauf que la recherche des fichiers est faite en interne par l'extension standard `File::Glob`. Bien sûr, le moyen le plus court d'obtenir le résultat précédent est :

```
chmod 0644, <*.c>;
```

Un motif de noms de fichier évalue ses arguments uniquement lorsqu'il débute une nouvelle liste. Toutes les valeurs doivent être lues avant de recommencer. Dans un contexte de liste, cela n'a aucune importance puisque vous récupérez toutes les valeurs quoi qu'il arrive. Dans un contexte scalaire, par contre, l'opérateur retourne la valeur suivante à chaque fois qu'il est appelé ou la valeur `undef` à la fin. Comme pour les filehandles un `defined` automatique est généré lorsque l'opérateur est utilisé comme test d'un `while` ou d'un `for` - car sinon certains noms de fichier légaux (par ex. un fichier nommé *0*) risquent de terminer la boucle. Encore une fois, `undef` n'est retourné qu'une seule fois. Donc si vous n'attendez qu'une seule valeur, il vaut mieux écrire :

```
($file) = <blurch*>;
```

que :

```
$file = <blurch*>;
```

car dans le dernier cas vous aurez soit un nom de fichier soit faux.

Si vous avez besoin de l'interpolation de variables, il est définitivement meilleur d'utiliser la fonction glob() parce que l'ancienne notation peut être confondu avec la notation des filehandles indirects.

```
@files = glob("$dir/*.[ch]");
@files = glob($files[$i]);
```

2.32 Traitement des constantes

Comme en C, Perl évalue un certain nombre d'expressions lors de la compilation lorsqu'il peut déterminer que tous les arguments d'un opérateur sont statiques et n'ont aucun effet de bord. En particulier, la concaténation de chaînes a lieu lors de la compilation entre deux littéraux qui ne sont pas soumis à l'interpolation de variables. L'interprétation du backslash (barre oblique inversée) a lieu elle aussi lors de la compilation. Vous pouvez dire :

```
'Now is the time for all' . "\n" .
    'good men to come to.'
```

qui sera réduit à une seule chaîne de manière interne. Vous pouvez aussi dire :

```
foreach $file (@filenames) {
    if (-s $file > 5 + 100 * 2**16) {  }
}
```

le compilateur pré-calculera la valeur représentée par l'expression et l'interpréteur n'aura plus à le faire.

2.33 No-ops (opération vide)

Perl n'a pas officiellement d'opérateur no-op (une opération vide) mais les constantes `0` et `1` sont des cas spéciaux évitant de produire un avertissement si elles sont utilisées dans un contexte vide si bien que vous pouvez écrire :

```
1 while foo();
```

2.34 Opérateurs bit à bit sur les chaînes

Les chaînes de bits de longueur arbitraire peuvent être manipulées par les opérateurs bit à bit (`~ | & ^`).

Si les opérandes d'un opérateur bit à bit sont des chaînes de longueurs différentes, les opérateurs | et ^ agiront comme si l'opérande le plus court était complété par des bits à zéro à droite alors que l'opérateur **&** agira comme si l'opérande le plus long était tronqué à la longueur du plus court. Notez que la granularité pour de telles extensions ou troncatures est d'un ou plusieurs octets.

```
# Exemples ASCII
print "j p \n" ^ " a h";            # affiche "JAPH\n"
print "JA" | "  ph\n";              # affiche "japh\n"
print "japh\nJunk" & '_____';       # affiche "JAPH\n";
print 'p N$' ^ " E<H\n";            # affiche "Perl\n";
```

Si vous avez l'intention de manipuler des chaînes de bits, vous devez être certain de fournir des chaînes de bits : si un opérande est un nombre cela implique une opération bit à bit **numérique**. Vous pouvez explicitement préciser le type d'opération que vous attendez en utilisant `""` ou `0+` comme dans les exemples ci-dessous :

```
$foo =  150  |  105;        # produit 255  (0x96 | 0x69 vaut 0xFF)
$foo = '150' |  105;        # produit 255
$foo =  150  | '105';       # produit 255
$foo = '150' | '105';       # produit la chaîne '155' (si en ASCII)

$baz = 0+$foo & 0+$bar;     # les deux opérandes explicitement numériques
$biz = "$foo" ^ "$bar";     # les deux opérandes explicitement chaînes
```

Voir vec in *perlfunc* pour savoir comment manipuler des bits individuellement dans un vecteur de bits.

2.35 Arithmétique entière

Par défaut Perl suppose qu'il doit effectuer la plupart des calculs arithmétiques en virgule flottante. Mais en disant :

```
use integer;
```

vous dites au compilateur qu'il peut utiliser des opérations entières (si il le veut) à partir de ce point et jusqu'à la fin du bloc englobant. Un BLOC interne peut contredire cette commande en disant :

```
no integer;
```

qui durera jusqu'à la fin de ce BLOC. Notez que cela ne signifie pas que tout et n'importe quoi devient entier mais seulement que Perl peut utiliser des opérations entières si il le veut. Par exemple, même avec `use integer`. `sqrt(2)` donnera toujours quelque chose comme `1.4142135623731`.

Utilisés sur des nombres, les opérations bit à bit ("&", "|", "^", "~", "<<", et ">>") produisent toujours des résultats entiers. (Mais voir aussi Opérateurs bit à bit sur les chaînes.) Par contre, `use integer` a encore une influence sur eux. Par défaut, leurs résultats sont interprétés comme des entiers non-signés. Par contre, si `use integer` est actif, leurs résultats sont interprétés comme des entiers signés. Par exemple, `~0` est habituellement évalué comme un grande valeur entière. Par contre, `use integer; ~0` donne -1 sur des machines à complément à deux.

2.36 Arithmétique en virgule flottante

Bien que `use integer` propose une arithmétique uniquement entière, il n'y a aucun moyen similaire d'imposer des arrondis ou des troncatures à un certain nombre de décimales. Pour arrondir à un nombre de décimales précis, la méthode la plus simple est d'utiliser sprintf() ou printf().

Les nombres en virgule flottante ne sont qu'une approximation de ce que les mathématiciens appellent les nombres réels. Il y a infiniment plus de réels que de flottants donc il faut arrondir quelques angles. Par exemple :

```
printf "%.20g\n", 123456789123456789;
#         produit 123456789123456784
```

Tester l'égalité ou l'inégalité exacte de nombres flottants n'est pas une bonne idée. Voici un moyen (relativement coûteux) pour tester l'égalité de deux nombres flottants sur un nombre particulier de décimales. Voir le volume II de Knuth pour un traitement plus robuste de ce sujet.

```
sub fp_equal {
    my ($X, $Y, $POINTS) = @_;
    my ($tX, $tY);
    $tX = sprintf("%.${POINTS}g", $X);
    $tY = sprintf("%.${POINTS}g", $Y);
    return $tX eq $tY;
}
```

Le module POSIX (qui fait partie de la distribution standard de perl) implémente ceil(), floor() et un certain nombre d'autres fonctions mathématiques et trigonométriques. Le module Math::Complex (qui fait partie de la distribution standard de perl) définit de nombreuses fonctions mathématiques qui fonctionnent aussi bien sur des nombres réels que sur des nombres complexes. Math::Complex n'est pas aussi performant que POSIX mais POSIX ne peut pas travailler avec des nombres complexes.

Arrondir dans une application financière peut avoir de sérieuses implications et la méthode d'arrondi utilisée doit être spécifiée précisément. Dans ce cas, il peut être plus sûr de ne pas faire confiance aux différents systèmes d'arrondi proposés par Perl mais plutôt d'implémenter vous-mêmes la fonction d'arrondi dont vous avez besoin.

2.37 Les grands nombres

Les modules standard Math::BigInt et Math::BigFloat fournissent des variables arithmétiques en précision infinie et redéfinissent les opérateurs correspondants. Au prix d'un peu d'espace et de beaucoup de temps, ils permettent d'éviter les embûches classiques associées aux représentations en précision limitée.

```
use Math::BigInt;
$x = Math::BigInt->new('123456789123456789');
print $x * $x;

# affiche +15241578780673678515622620750190521
```

Il existe plusieurs modules vous permettant d'effectuer vos calculs avec une précision arbitraire ou illimitée (limitée uniquement par le temps de calcul et la mémoire disponible). Il existe aussi des modules non standard qui fournissent des implémentations plus rapides passant par des bibliothèques C externes.

En voici une courte liste non-exhaustive :

```
Math::Fraction          grand rationnels exacts comme 9973 / 12967
Math::String            traite les chaînes comme des nombres
Math::FixedPrecision    calcul avec précision arbitraire
Math::Currency          pour les calculs financiers
Bit::Vector             manipulation rapide de vecteurs de bits (en C)
Math::BigIntFast        Extension de Bit::Vector pour les grands nombres
Math::Pari              donne accès à la bibliothèque C Pari
Math::BigInteger        utilise une bibliothèque C externe
Math::Cephes            utilise la bibliothèque C Cephes (pas de grands nombres)
Math::Cephes::Fraction  rationnels via la bibliothèque Cephes
Math::GMP               encore une autre bibliothèque C externe
```

Il ne vous reste plus qu'à choisir.

3 TRADUCTION

3.1 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/.

3.2 Traducteur

Traduction initiale et mise à jour vers 5.8.8 : Paul Gaborit <Paul DOT Gaborit AT enstimac DOT fr>.

3.3 Relecture

Personne pour l'instant.

4 À propos de ce document

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

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

Si vous avez des remarques concernant ce document, en premier lieu, contactez la traducteur (vous devriez trouver son adresse électronique dans la rubrique TRADUCTION) et expliquez-lui gentiment vos remarques ou critiques. Il devrait

normalement vous répondre et prendre en compte votre avis. En l'absence de réponse, vous pouvez éventuellement me contacter.

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