De Perl 5 à Perl 6

Partie 2 : les nouveautés

Par Moritz Lenz - Laurent Rosenfeld

Date de publication : 4 janvier 2015

Dernière mise à jour : 6 janvier 2019

DÉBUTANT

Cette série d'articles est une traduction ou plutôt une adaptation assez libre **d'une série de blogs en anglais** de Moritz Lenz, qui m'a aimablement donné l'autorisation de faire cette adaptation en français. L'essentiel du contenu technique provient du texte de Moritz Lenz, qui mérite tout le crédit pour son travail, mais j'ai suffisamment réécrit ce texte et ajouté des points me paraissant intéressants pour mériter pour ma part de supporter intégralement la responsabilité de toute erreur qui aurait pu s'y glisser.

La **première partie de ce texte** examinait surtout les différences de syntaxe de base entre Perl 5 et Perl 6 et les apports de ces différences. La présente seconde partie se penche plus sur les notions complètement nouvelles de Perl 6.

N'hésitez pas à laisser un commentaire : **Commentez**

Laurent R.

En complément sur Developpez.com
- De Perl 5 à Perl 6 - Partie 1 : les bases du langage
- De Perl 5 à Perl 6 - Partie 3 : approfondissements

- De Perl 5 à Perl 6- Annexe 1: Ce qui change entre Perl 5 et Perl 6

1 - Les jonctions

1-1 - Résumé

```
my $x = 4;
if $x == 3|4 {
    say '$x vaut soit 3 soit 4'
}
say ((2|3|4)+7)        # équivalent à 9|10|11 ou any(9, 10, 11)
```

1-2 - Description

Les jonctions sont des superpositions de valeurs non ordonnées. Les opérations sur les jonctions s'exécutent séparément sur tous les éléments de la jonction (peut-être même en parallèle dans des *threads* différents si le compilateur le supporte) et une nouvelle jonction du même type est assemblée à partir des résultats partiels.

Il y a quatre types de jonctions, qui ne diffèrent que quand elles sont évaluées en contexte booléen. Les types disponibles sont any, all, one et none. En contexte booléen, une jonction de type any retournera une valeur vraie si l'une au moins des valeurs de la jonction est vraie ; une jonction de type all renverra vrai si toutes les valeurs de la jonction sont vraies, et ainsi de suite. Les opérateurs |, ^ et & ne sont plus des opérateurs booléens binaires, mais des opérateurs infixés de jonctions ou même des constructeurs de jonctions :

Type	Opérateur infixé
any	|
one	^
all	&

1 | 2 | 3 est la même chose que any(1..3).

```
my Junction $jour_semaine = any <lundi mardi mercredi
                                 jeudi vendredi samedi dimanche>;
if $jour eq $jour_semaine {    # on suppose que $jour a été initialisé
    say "Rendez-vous $jour à 20 h pour dîner";
}
```

Dans cet exemple, l'opérateur eq est appelé avec chaque paire $jour_semaine, 'lundi', jour_semaine, 'mardi', etc. et le résultat est stocké dans une nouvelle jonction de type any. Dès que le résultat est connu (dans ce cas, dès que l'une des comparaisons a renvoyé *True*) l'exécution des autres comparaisons peut être interrompue par le compilateur.

Ce sont les jonctions qui permettent de comparer simplement une variable à plusieurs valeurs simultanément, comme cela a été abordé dans**la première partie de ce tutoriel** :

```
say "$_ est un nombre de Fibonacci"
     for grep { $_ == any(<1 2 3 5 8 13>)}, 1..20;
     # Noter la virgule après le bloc du grep en Perl 6
```

Ce qui imprime bien :

```
1 est un nombre de Fibonacci
2 est un nombre de Fibonacci
3 est un nombre de Fibonacci
5 est un nombre de Fibonacci
8 est un nombre de Fibonacci
13 est un nombre de Fibonacci
```

En Perl 5 et dans la quasi-totalité des autres langages de programmation, il aurait fallu comparer la variable avec chacune des valeurs individuellement :

Code Perl 5
```
say "$_ est un nombre de Fibonacci"
     for grep { $_ == 1 or $_ == 2 or $_ == 3 or $_ == 5
     or $_ == 8 or $_ == 13} 1..20;
     # pas de virgule après le bloc du grep en Perl 5
```

On peut aussi utiliser une jonction sur un tableau calculé précédemment. Si nous avons calculé les premiers nombres de Fibonacci et les avons stockés dans le tableau @fib, la syntaxe suivante évite de coder une boucle imbriquée :

```
my @fib = <1 2 3 5 8 13>; # initialisation figurant le calcul de @fib
say "$_ est un nombre de Fibonacci" for grep { $_ == any(@fib)}, 1..20;
```

ce qui affiche la même chose que précédemment. En Perl 5, il aurait fallu une seconde boucle imbriquée, ou éventuellement stocker les nombres de Fibonacci dans un hachage (voire, pour les versions suffisamment récentes, utiliser l'opérateur *smart match* ~~, mais il vaut sans doute mieux éviter, dans la mesure où celui-ci est maintenant déprécié sur les dernières versions de Perl 5). À l'heure où nous écrivons, il est encore prématuré de faire un benchmark pour comparer les deux solutions. En théorie au moins, les différentes comparaisons peuvent être exécutées dans des threads différents tirant parti des processeurs multicore ou de plates-formes ayant des processeurs multiples. Nous ne pouvons dire si cette optimisation est implantée à ce jour.

Bien entendu, si nous ne voulions tester qu'un seul nombre de Fibonacci, nous n'avons plus besoin du grep :

```
my @fib = <1 2 3 5 8 13>;
say "Oui" if 8 == any(@fib); # Affiche "Oui"
```

Ceci fonctionne non seulement pour des opérateurs, mais aussi pour des fonctions.

```
if 2 == sqrt(4 | 9 | 16) {
    say "Oui"; # imprime Oui, puisque 2 est bien la racine de 4
}
```

Pour que cela soit possible, les jonctions ont une place légèrement à part dans la hiérarchie habituelle des types :

```
                    Mu
                   /  \
                  /    \
               Any     Junction
              /  |  \
        Tous les autres types
```

Si vous voulez écrire une fonction qui prend en paramètre une jonction et ne lance pas automatiquement des threads, vous devez décrire le paramètre de la fonction comme soit *Mu*, soit *Junction*.

```
sub dump_yaml(Junction $stuff) {
    # nous espérons que Yaml peut représenter des jonctions ;-)
    ....
}
```


Attention : les jonctions peuvent (ou pourraient) présenter un comportement paraissant contraire à l'intuition avec des opérateurs négatifs tels que !=, ne, !~, etc. Avec des types qui ne sont pas des jonctions, les expressions $a != $b et !($a == $b) veulent toujours dire la même chose. D'après la description que nous avons faite jusqu'ici des jonctions, il pourrait en aller différemment si l'une des variables est une jonction ; nous pourrions avoir le comportement hypothétique suivant :

FAUX
```
# Attention: comportement hypothétique non vérifié en Perl 6
my Junction $b = 3 | 2;
my $a = 2;
say "Oui" if   $a != $b ;       # imprimerait Oui
```

FAUX
```
say "Oui" if !($a == $b);        # n'imprime rien
```

La logique paraît imparable : 2!= 3 est vrai. Donc, $a != 2|3 est également vrai. D'un autre côté, $a == $b renvoie une valeur booléenne unique (vraie), et la négation de cette valeur est donc False.

Ce comportement a été jugé trop contre-intuitif, et il a donc été décidé que, par convention, l'expression $a !op $b est réécrite internement !($a op $b). La négation externe impose une évaluation en contexte booléen de l'expression.

```
my $mot = "oui";
my @negations = <non ne ni pas personne jamais nul>;
if $mot !eq any @negations {
    say "oui n'est pas une négation";
}
```

Sans ce cas particulier, une expression du type $mot ne any @mots serait considérée vraie pour pratiquement tous les cas non triviaux (du moment par exemple que le tableau @mots contient au moins deux mots différents).

Pour que cela fonctionne de façon homogène, l'opérateur ne est considéré comme la négation de !eq.

D'une façon générale, il est plus lisible d'utiliser un opérateur de comparaison positif et une jonction négative :

```
my $mot = 'oui';
my @negations = <non ne ni pas personne jamais nul>;
if $mot eq none @negations {
    say "oui n'est pas une négation";
}
```

1-3 - Motivation

Perl essaie d'être proche des langues naturelles, dans lesquelles on dit souvent des choses du genre : « le $résultat est $ceci ou $cela », plutôt que « le $résultat est $ceci ou le $résultat est cela ». La plupart des langages de programmation n'autorisent que la seconde forme (traduite dans le langage en question), ce qui donne une sensation de maladresse. Avec Perl 6, on peut aussi utiliser la première forme, plus intuitive.

Cela permet d'écrire très rapidement des comparaisons qui nécessiteraient des boucles explicites :

Perl 5
```
# code Perl 5:
my @items = get_data();
my $all_non_neg = 1;
for (@items){
    if ($_ < 0) {
        $all_non_neg = 0;
        last;
    }
}
if ($all_non_neg) { ... }
```

Ou, si vous connaissez le module List::MoreUtils :

Perl 5
```
use List::MoreUtils qw(all);
```

Perl 5
```
my @items = get_data;
if (all { $_ >= 0 } @items) { ...  }
```

En Perl 6, c'est on ne peut plus simple :

```
my @items = get_data();
if all(@items) >= 0 { ... }
```

1-3-1 - Un mot d'avertissement

Beaucoup de gens sont excités par la puissance des jonctions et tendent parfois à leur en demander trop.

Les jonctions ne sont pas des ensembles. Si vous essayez d'extraire des éléments d'une jonction, c'est probablement que vous vous y prenez mal. Utilisez plutôt un Set.

1-4 - Voir aussi

http://perlcabal.org/syn/S03.html#Junctive_operators

http://laurent-rosenfeld.developpez.com/tutoriels/perl/perl6/annexe-02/#L1-4-2

2 - Comparaisons

2-1 - Résumé

```
"ab"    eq      "ab"    ; # Vrai
"1.0"   eq      "1"     ; # Faux (comparaison de chaîne)
"a"     ==      "b"     ; # échec, car "a" n'est pas numérique
"1"     ==      1.0     ; # Vrai (comparaison numérique)
1       ===     1       ; # Vrai
[1, 2]  ===     [1, 2]  ; # Faux
$x = [1, 2];            ; #
$x      ===     $x      ; # Vrai
$x      eqv     $x      ; # Vrai
[1, 2]  eqv     [1, 2]  ; # Vrai
1.0     eqv     1       ; # Faux
                        ; #
'abc'   ~~      m/a/    ; # Reconnaît l'objet, vrai en contexte booléen
'abc'   ~~      Str     ; # Vrai
'abc'   ~~      Int     ; # Faux
Str     ~~      Any     ; # Vrai
Str     ~~      Num     ; # Faux
1       ~~      0..4    ; # Vrai
-3      ~~      0..4    ; # Faux
```

2-2 - Description

2-2-1 - Opérateurs existants en Perl 5

Perl 6 conserve les opérateurs de comparaison de chaînes de caractères (eq, lt, gt, le, ge ne ; l'opérateur cmp est maintenant remplacé par leg) qui évaluent leurs opérandes en contexte de chaînes de caractères. De même, tous les opérateurs de comparaison numérique de Perl 5 restent en vigueur.

Le seul changement notable, déjà mentionné dans la **première partie** de ce document, est que ces opérateurs peuvent désormais être chaînés. Pour comparer l'égalité de trois termes, il faut au moins deux comparaisons en Perl 5 :

Code Perl 5
```
my $c = my $d = my $e = 5;
print "Vrai\n" if $c == $d and $d == $e; # imprime Vrai
print "Vrai\n" if 5 < 7 and 7 < 9 and 9 < 14 and 14 < 17; # Vrai aussi
```

En Perl 6, on peut écrire directement :

```
my $c = my $d = my $e = 5;
say "Vrai" if $c == $d == $e;       # syntaxe illégale en Perl 5
say "Vrai" if 5 < 7 < 9 < 14 < 17;  # les 2 lignes impriment aussi vrai
```

2-2-2 - Nouveaux opérateurs

Comme les objets sont maintenant plus que des références « bénies » (blessed), une nouvelle méthode de comparaison est nécessaire. `===` renvoie vrai pour des valeurs identiques. Pour des valeurs non modifiables comme des nombres ou des chaînes (Strings), l'opérateur `===` revient à un test d'égalité habituel, mais pour d'autres objets, il ne renvoie True que si les deux variables réfèrent au même objet (comme une comparaison d'adresse mémoire en C++).

L'opérateur d'équivalence `eqv` teste si deux choses sont équivalentes, c'est-à-dire si elles ont le même type et la même valeur. Dans le cas de variables non scalaires (de type Array ou Hash), `eqv` compare le contenu. Deux structures de données construites de la même façon et avec les mêmes valeurs sont équivalentes.

2-2-3 - Smart match (comparaison intelligente)

Perl 6 a un opérateur de « comparaison universelle », qui se nomme « *smart match* » (que l'on peut traduire par « comparaison intelligente ») et se note `~~`. (1) Il est asymétrique et, d'une façon générale, le type de l'opérande de droite détermine le type de comparaison qui est effectué.

Pour les types non modifiables, c'est une simple comparaison d'égalité. Une comparaison avec un type renvoie vrai si le type est correct. Une comparaison avec une regex reconnaît la regex. Dans le cas d'une comparaison entre un scalaire et objet de type Range (intervalle), la comparaison vérifie si le scalaire fait partie de l'intervalle.

Il y a d'autres formes plus avancées de correspondance : par exemple, on peut vérifier si une liste d'arguments (Capture) correspond à la liste de paramètres (Signature) d'une fonction, ou appliquer un opérateur de test de fichier (comme le test -e de Perl 5).

Pour résumer, il faut se souvenir que toute question de type « est-ce que $x ressemble à $y » se formulera sous la forme d'un smart match en Perl 6.

2-3 - Voir aussi

http://perlcabal.org/syn/S03.html#Smart_matching

3 - Contenants et valeurs

3-1 - Résumé

```
my ($x, $y);
$x := $y;
```

```
$y = 4;
say $x;             # imprime 4
if $x =:= $y {
    say '$x et $y sont des noms différents pour la même chose'
}
```

Perl 6 fait la distinction entre les contenants (*containers*) et les valeurs qui peuvent être stockées dans ces contenants.

Une variable scalaire ordinaire est un contenant et peut avoir des propriétés telles que des contraintes de type ou des contraintes d'accès (par exemple, elle peut être en lecture seule), et elle peut être *aliassée* à d'autres contenants.

Mettre une valeur dans un contenant s'appelle une affectation (*assignment*) et se fait avec l'opérateur `=` (comme en Perl 5). Créer un alias entre deux contenants revient à établir une liaison (*binding*) et se fait avec l'opérateur `:=` ; on dira alors que les variables sont *liées*.

```
my @a = 1, 2, 3;
my Int $x = 4;
@a[0] := $x;      # @a[0] et $x sont maintenant la même variable
@a[0] = 'Foo';    # Erreur: 'Type check failed'
```

Les types tels que `Int` ou `Str` ne sont pas modifiables, ce qui veut dire que les objets de ces types ne peuvent être modifiés. Mais cela n'empêche pas de modifier les variables (c'est-à-dire les contenants) dans lesquelles ils sont stockés :

```
my $a = 1;
$a = 2;   # rien de surprenant ici
```

La liaison entre deux contenants peut être effectuée au moment de la compilation à l'aide de l'opérateur `::=`.

Il est possible de vérifier si deux variables sont liées avec l'opérateur `=:=`.

3-2 - Motivation

L'importation et l'exportation de fonctions, de types et de variables se font par l'utilisation d'alias. Plutôt que de recourir à une magie d'alias implicite et parfois difficile à comprendre, Perl 6 offre un opérateur explicite simple.

3-3 - Voir aussi

http://perlcabal.org/syn/S03.html#Item_assignment_precedence

4 - Changements par rapport aux opérateurs de Perl 5

4-1 - Résumé

```
# opérateurs binaires (bit à bit)
# note: 5 vaut 0b0101 et 3 vaut 0b0011
5   +| 3;        # 7 (0b111)
5   +^ 3;        # 6 (0b110)
5   +& 3;        # 1 (0b001)
"b" ~| "d";      # 'f'

# Concaténation de chaînes de caractères
'a' ~ 'b';       # 'ab'

# tests de fichiers
if '/etc/passwd'.path ~~ :e { say "le fichier existe" }

# répétition
```

```
'a' x 3;        # 'aaa'
'a' xx 3;       # 'a', 'a', 'a'

# opérateur conditionnel ternaire
my ($a, $b) = 2, 2;
say $a == $b ?? 2 * $a !! $b - $a; # imprime 4

# comparaisons en chaîne
my $angle = 1.41;
if 0 <= $angle < 2 * pi { ... }
```

4-2 - Description

Les opérateurs numériques (+, -, /, *, **, %) restent tous inchangés.

Comme les opérateurs |, ^ et & servent désormais à construire des jonctions (§ 1), les opérateurs binaires n'ont plus la même syntaxe. Ils contiennent maintenant un préfixe de contexte : par exemple, le OU binaire s'écrit maintenant +| en contexte numérique (2), et ~^ est le complément sur une chaîne. Les opérateurs de décalage de bits ont changé de la même façon : le décalage à gauche (anciennement <<) devient +<.

L'opérateur de concaténation de chaîne est désormais ~ (le point . étant utilisé pour les invocations de méthodes).

Les opérateurs de tests sur les fichiers sont maintenant en notation par paire clef-valeur (objet de type Pair).La clef contient l'opérateur et la valeur le résultat du test. Par exemple, l'opérateur -e de Perl 5 est remplacé par :e (dans une paire clef-valeur :key<value>, :key renvoie vrai si la valeur est vraie). Si le nom du fichier n'est pas contenu dans $_, alors il peut être fourni avec l'opérateur de comparaison intelligente et utilisant la syntaxe : $filename.path ~~ :e.

L'opérateur de répétition x est maintenant disjoint en deux opérateurs distincts : x réplique les chaînes de caractères et xx les listes.

L'opérateur conditionnel ternaire précédemment appelé $condition ? $true : $false, s'appelle désormais $condition ?? $true !! $false.

Les opérateurs de comparaison peuvent s'enchaîner, si bien que l'on peut écrire :

```
if $a < $b < $c {...}
```

et obtenir exactement ce que l'on veut dire (même sens qu'en mathématiques).

De même pour des comparaisons lexicographiques :

```
say "Vrai" if 'abc' le 'bc' le 'efgh'; # imprime Vrai
```

4-3 - Précédence des opérateurs

Ce texte n'est pas un document de référence, mais il paraît utile de donner ici la précédence (priorité d'exécution) des différents opérateurs. Cela permet en outre de mentionner l'existence d'opérateurs que nous n'avons pas cités ici, mais qui pourront éventuellement intéresser le lecteur. De plus, cela sera utile pour une meilleure compréhension d'un prochain chapitre (§ 6, **Créer ses propres opérateurs**, et, plus précisément la section 6.2.1).

Le tableau suivant résume la précédence des opérateurs inclus dans Perl 6, classés depuis ceux ayant la précédence la plus forte vers ceux ayant la plus faible.

La colonne de gauche (A) donne l'associativité lorsqu'il y a plus de deux termes ayant la même précédence (voir aussi § 6.2.2) :

- **G** : associatif à gauche (par ex. : $c - $d - $e s'interprète comme ($c - $d) - $e) ;
- **D** : associatif à droite (par ex. : $c ** $d ** $e équivaut à $c ** ($d ** $e)) ;
- **N** : chaînage d'opérateurs interdit ou n'ayant pas de sens (donc, illégal) ;
- **C** : chaînage (par ex. : $c < $d < $e équivaut à ($c < $d) and ($d < $e)) ;
- **X** : création de liste (par exemple une jonction : 9|10|11 ou any(9, 10, 11)) ;

A	Niveau	Exemples
N	Termes	42 3.14 "eek" qq["foo"] $x :!verbose @$array
G	Méthode postfixée	.meth .+ .? .* .() .[] .{} .<> .« » .:: .= .^ .:
N	Auto-incrémentation	++ --
D	Exponentielle	**
G	Symbole unaire	! + - ~ ? | || +^ ~^ ?^ ^
G	Multiplications	* / % %% +& +< +> ~& ~< ~> ? & div mod gcd lcm
G	Additions	+ - +| +^ ~| ~^ ?| ?^
G	Réplication	x xx
X	Concaténation	~
X	*et* dans une jonction	&
X	*ou* dans une jonction	| ^
G	Unaire nommé	Temp let
N	Infixé structuré	but does <=> leg cmp .. ..^ ^.. ^..^
C	Infixé chaîné	!= == < <= > >= eq ne lt le gt ge ~~ === eqv !eqv
X	*et* booléen à haute précédence	&&
X	*ou* booléen à haute précédence	|| ^^ // min max
D	Conditionnel	?? !! ff fff
D	Affectation	= => += -= **= xx= .=
G	Unaire	so not
X	Opérateur virgule	, :
X	Liste infixée	Z minmax X X~ X* Xeqv ...
D	Liste préfixée	print push say die map substr ... [+] [*] any Z=
X	*et* de basse précédence	and andthen
X	*ou* de basse précédence	or xor orelse
X	Séquenceur	<==, ==>, <<==, ==>>
N	Terminateur	; {...}, unless, extra),], }

4-4 - Les métaopérateurs

Les opérateurs manipulent des données. Les métaopérateurs manipulent des opérateurs.

4-4-1 - Les opérateurs de réduction

Les métaopérateurs de réductions prennent une liste (et un opérateur) en entrée et renvoient un scalaire. Ils obtiennent ce résultat en appliquant l'opérateur à chaque paire d'éléments de la liste en entrée.

```
my @nombres = 1..5;
my $somme = [+] @nombres;       # 15:  1 + 2 + 3 + 4 + 5
my $produit = [*] @nombres;     # 120: 1 * 2 * 3 * 4 * 5
```

Les crochets [ ] transforment un opérateur manipulant des scalaires en un opérateur de réduction agissant sur une liste. Ils peuvent également être utilisés avec les opérateurs de comparaison :

```
my $x = [<] @y;    # vrai si les éléments de @y sont triés par ordre ascendant
my $z = [>] @y;    # vrai si les éléments de @y sont triés par ordre descendant
```

Voir aussi **http://laurent-rosenfeld.developpez.com/tutoriels/perl/perl6/approfondissements/#L5**

4-4-2 - Les hyperopérateurs

Les opérateurs de réduction prennent en entrée une liste et renvoient un scalaire, les hyperopérateurs appliquent l'opération spécifiée à chaque élément de la liste et renvoient la liste transformée (un peu comme un map en Perl 5). On construit les hyperopérateurs avec les guillemets français « ». Si votre éditeur ou votre clavier ne permet pas d'utiliser les guillemets français, il est possible d'utiliser les symboles ASCII << et >> à la place.

Saisir des caractères Unicode dans un éditeur

Si l'éditeur de texte utilisé le permet, voici les points de code à utiliser pour saisir les guillemets français « » :

Symbole	Point de code	Équivalent ASCII
«	U+00AB	<<
»	U+00BB	>>

La façon de saisir ces points de code varie selon l'éditeur utilisé.

Sous VIM, par exemple, on saisit un caractère Unicode (en mode d'insertion) en tapant Ctrl-V (noté ^V), puis la lettre u, puis la valeur hexadécimale du code voulu. Par exemple, la lettre grecque λ (lambda) s'obtient avec la combinaison de touches ^Vu03BB. De même, Le guillemet français ouvrant « s'obtiendra avec la combinaison ^Vu00AB (et le fermant » avec ^Vu00BB).

Sous Emacs, la lettre λ s'obtient avec la combinaison suivante : Ctrl-x 8 <CR> 3bb <CR>. (<CR> signifie ici la touche « Entrée » et les espaces ont été ajoutés à des seules fins de clarté, mais ne doivent pas être saisis)

Pour d'autres éditeurs, le lecteur est invité à consulter la documentation de son éditeur.

```
my @a = 1..5;
my @b = 6..10;
my @c = 5 <<*>> @b;
say @c;              # imprime 30 35 40 45 50
my @d = @a >>*<< @b;
say @d;              # imprime 6 14 24 36 50
# @d = 1*6, 2*7, 3*8, 4*9, 5*10
```

À remarquer qu'à l'époque où nous avions écrit initialement ce texte (automne 2014), l'utilisation des guillemets français ne marchait pas correctement à cause apparemment d'un support encore limité aux caractères Unicode (dans la version de Rakudo / Perl 6 utilisée à l'époque), ainsi que nous l'avions dit dans une note de bas de page. C'est la raison pour laquelle nos exemples de tests utilisaient tous les symboles ASCII << et >>. Depuis, la situation a bien progressé et de nouveaux tests (août 2015) montrent que cela fonctionne maintenant parfaitement avec les versions plus récentes de Perl 6 :

```
my @a = 1..5;
my @b = 6..10;
```

```
my @c = 5 «*» @b;
say @c;              # imprime 30 35 40 45 50 (5*6, 5*7, 5*8 ...)
my @d = @a »*« @b;
say @d;              # imprime 6 14 24 36 50 (1*6, 2*7, 3*8, ...)
```

On peut également utiliser les hyperopérateurs avec des opérateurs unaires :

```
my @a = 2, 4, 6;
say -<< @a;          # imprime  -2 -4 -6
```

Les hyperopérateurs unaires renvoient toujours une liste de la même taille que la liste en entrée. Les hyperopérateurs infixés ont un comportement différent selon la taille de leurs opérandes :

```
@a >>+<< @b;    # @a et @b doivent avoir la même taille
@a <<+<< @b;    # @a peut-être plus petit
@a >>+>> @b;    # @b peut-être plus petit
@a <<+>> @b;    # L'un ou l'autre peut être plus petit, Perl fera ce que vous voulez dire (DWIM)
```

Les hyperopérateurs fonctionnent aussi avec les opérateurs d'affectation :

```
@x >>+=<< @y    # Même chose que @x = @x >>+<< @y
```

4-4-3 - L'opérateur croix (cross)

L'opérateur croix utilise la lettre capitale X. La croix retourne une liste de toutes les listes possibles combinant les éléments de ses opérandes.

```
my @a = 1, 2;
my @b = 3, 4;
my @c = @a X @b;  # (1,3), (1,4), (2,3), (2,4)
```

L'opérateur croix peut également être utilisé comme un métaopérateur et appliquer l'opérateur qu'il modifie à chaque combinaison d'éléments de ses opérandes :

```
my @a = 3, 4;
my @b = 6, 8;
say @a X* @b;     # imprime 18 24 24 32
```

Voir aussi **http://laurent-rosenfeld.developpez.com/tutoriels/perl/perl6/approfondissements/#L6**

4-5 - Motivation

Certains de ces changements visent à un meilleur codage de Huffman, c'est-à-dire donner un nom court aux opérateurs souvent utilisés (comme le . pour les appels de méthodes) et un nom plus long aux opérateurs moins souvent utilisés (comme les opérateurs binaires, par exemple ~& pour le ET binaire pour les chaînes).

L'enchaînement des opérateurs de comparaison est une façon de rendre le langage plus naturel et de permettre des choses qui sont utilisées communément en notation mathématique.

Les métaopérateurs permettent une grande expressivité sur les listes.

4-6 - Voir aussi

http://perlcabal.org/syn/S03.html#Changes_to_Perl_5_operators

http://doc.perl6.org/language/operators

http://laurent-rosenfeld.developpez.com/tutoriels/perl/perl6/annexe-01/#L2

http://laurent-rosenfeld.developpez.com/tutoriels/perl/perl6/annexe-01/#L4

http://laurent-rosenfeld.developpez.com/tutoriels/perl/perl6/annexe-02/#L4

5 - Paresse

5-1 - Résumé

```
my @integers = 0..*;
for @integers -> $i {
    say $i;
    last if $i % 17 == 0;
}

my @even := map { 2 * $_ }, 0..*;
my @stuff := gather {
    for 0 .. Inf {
        take 2 ** $_;
    }
}
```

5-2 - Description

La paresse est une vertu pour le programmeur Perl 5. En Perl 6, même les listes sont paresseuses. (3)

En l'occurrence, le mot *paresseux* signifie que l'évaluation d'une liste est retardée au maximum. Si l'on écrit ce genre de code :

```
my @a := map BLOCK, @b;
```

le bloc n'est pas exécuté immédiatement. Ce n'est que quand on accède aux éléments de @a que le map exécute le bloc et fournit à @a les éléments nécessaires.

À noter l'utilisation de l'opérateur de liaison := au lieu de celui affectation. Affecter à un tableau peut conduire à une évaluation avide ou gourmande (à moins que le compilateur ne sache que la liste sera infinie ; les détails exacts de comment ce sera fait sont encore susceptibles de changer), mais l'opérateur de liaison ne le fait jamais.

La paresse permet de traiter des listes infinies. Tant que l'on n'applique pas un traitement à tous ses arguments, les listes infinies n'occupent en mémoire que l'espace requis pour stocker les éléments qui ont déjà été évalués.

Il y a cependant quelques pièges à éviter. Déterminer la longueur d'une liste ou la trier tue la paresse. Si la liste est infinie, ça va boucler indéfiniment, ou ça va échouer si la nature infinie de la liste peut être détectée.

D'une façon générale, toutes les conversions vers des scalaires (comme List.join) sont *avides*, c'est-à-dire non paresseuses.

La paresse empêche les calculs inutiles et peut donc améliorer considérablement les performances tout en conservant du code simple.

Quand on lit un fichier ligne à ligne en Perl 5, on évite généralement d'utiliser la syntaxe for (<HANDLE>), parce que cela a pour effet de lire l'ensemble du fichier en mémoire avant de commencer à itérer sur son contenu. Avec la paresse, ce n'est plus un problème :

```
my $file = open '/etc/passwd';
```

```
for $file.lines -> $line {
    say $line;
}
```

Comme file.lines est une liste paresseuse, les lignes ne sont lues depuis le disque qu'en cas de besoin (hormis, bien sûr, l'utilisation interne de mémoire tampon pour réduire le nombre d'appels système).

5-2-1 - Gather/take

Une construction très utile pour créer des listes paresseuses est gather { take }. Elle s'utilise comme suit :

```
my @list := gather {
    while True {
        # calculs;
        take $result;
    }
}
```

Le bloc gather renvoie une liste potentiellement paresseuse (selon le contexte). Quand on a besoin d'éléments de la liste, le bloc est exécuté jusqu'à ce que take produise un élément utilisable. L'opérateur take fonctionne pratiquement comme un return et les éléments fournis par take sont utilisés pour construire la liste @list. Quand il y a besoin de plus d'éléments, l'exécution du bloc reprend après le dernier take.

La construction gather { take } a une portée dynamique, si bien qu'il est possible d'appeler take en-dehors de la portée lexicale du bloc gather :

```
my @list = gather {
    for 1..10 {
        do_some_computation($_);
    }
}

sub do_some_computation($x) {
    take $x * ($x + 1);
}
print " ", @list[$_] for 1..7; # imprime  6 12 20 30 42 56 72

# Liste infinie paresseuse :
my $list = lazy gather {
    for 1..Inf {
        do_some_computation($_);
    }
}
```

À noter que gather peut aussi agir sur une seule instruction au lieu d'un bloc :

```
my @list = gather for 1..10 {
    do_some_computation($_);
}
```

5-2-2 - Un exemple complet : parcours de deux arbres binaires

Considérons le problème suivant, posé sur le site *Rosetta Code* (**http://rosettacode.org/wiki/Same_Fringe**). Il s'agit de comparer les feuilles de deux arbres binaires pour déterminer s'ils ont la même liste de feuilles lors d'un parcours de gauche à droite. La structure (et l'équilibre) des arbres n'a pas d'importance, seuls comptent le nombre, l'ordre et la valeur des feuilles. Il est précisé que la solution doit être de préférence « paresseuse », c'est-à-dire qu'il faut lire une par une chaque feuille de chaque arbre binaire pour les comparer, et arrêter la comparaison dès qu'une différence est trouvée ; autrement dit, la solution simple consistant à parcourir récursivement chaque arbre complètement pour établir deux listes de feuilles puis comparer les listes n'est pas jugée acceptable.

La solution proposée en Perl 6 (**http://rosettacode.org/wiki/Same_Fringe#Perl_6**) utilise gather { take } pour parcourir paresseusement les deux arbres et tient en cinq lignes de code :

```
sub fringe ($tree) {
    multi sub fringey (Pair $node) { fringey $_ for $node.kv; }
    multi sub fringey ( Any $leaf) { take $leaf; }
    gather fringey $tree;
}
sub samefringe ($a, $b) { fringe($a) eqv fringe($b) }
```

Nous invitons le lecteur à visiter ce site et comparer les solutions à ce problème dans différents langages de programmation. La solution en Perl 6 est la plus courte de celles présentées. Les solutions en Java et en C, par exemple, prennent chacune plus de 20 fois plus de code. (4) Cela illustre la grande expressivité de Perl 6.

5-2-3 - Contrôler la paresse

La paresse peut aussi parfois présenter des problèmes (et quiconque a essayé d'apprendre le langage Haskell a pu se rendre compte de l'étrangeté de son système d'entrées/sorties due au fait que Haskell soit à la fois paresseux et sans effet de bord). Parfois, on ne veut pas que le code soit paresseux. Dans ce cas, il suffit de préfixer les instructions avec le mot-clef eager (*avide*) :

```
my @list = eager map { $bloc_ayant_des_effets_de_bord }, @list;
```

Inversement, seules les listes sont paresseuses par défaut. Mais l'on peut construire des scalaires paresseux avec le mot-clef lazy :

```
my $ls = lazy { $calcul_long_et_coûteux };
```

Le scalaire $ls ne sera calculé que s'il est réellement utilisé par la suite.

5-3 - Motivation

En informatique, les problèmes peuvent pour la plupart être décrits comme des arbres de combinaisons possibles parmi lesquelles on cherche une solution. L'une des clefs pour construire un algorithme efficace est non seulement de trouver une manière efficace de trouver la solution, mais aussi d'être en mesure de ne construire que les parties intéressantes de l'arbre. C'est ce que font les algorithmes ou heuristiques de type *branch and bound* (séparation et évaluation) utilisés par exemple en recherche opérationnelle pour résoudre des problèmes de classe NP.

Les listes paresseuses permettent de définir cet arbre récursivement et de l'explorer pour trouver la solution : bien utilisées, elles ne construiront que les parties de l'arbre que vous avez réellement besoin d'utiliser.

D'une façon générale, la paresse rend la programmation plus facile parce qu'il n'y a pas besoin de savoir si le résultat d'un calcul sera vraiment utilisé : il suffit de le rendre paresseux et, s'il n'est en définitive pas utilisé, il ne sera simplement pas exécuté. Si l'on en a besoin, il sera exécuté au moment où c'est nécessaire, mais on n'aura rien perdu.

5-4 - Voir aussi

http://perlcabal.org/syn/S02.html#Lists

6 - Créer ses propres opérateurs

6-1 - Résumé

```
use v6;
```

```
multi sub postfix:<!>(Int $x) {
    my $factorielle = 1;
    $factorielle *= $_ for 2..$x;
    return $factorielle;
}

say 5!;          # imprime 120
```

6-2 - Description

Les opérateurs sont des fonctions ayant des noms inhabituels et quelques propriétés supplémentaires telles que la précédence (priorité d'exécution) et l'associativité. Perl 6 utilise le plus souvent une notation dite infixée, du type terme opérateur terme, dans laquelle terme peut éventuellement être lui-même précédé d'un opérateur préfixé ou suivi d'opérateurs postfixés ou postcirconfixés.

Exemple d'opérateur	Notation
1 + 1	infix (infixée)
-1	prefix (préfixée)
$x++	postfix (postfixée)
<a b c >	circumfix (circonfixée)
@a[1]	postcircumfix (postcirconfixée)

Les noms des opérateurs ne se limitent pas à des caractères spéciaux, ils peuvent contenir à peu près n'importe quoi à l'exception des espaces blancs.

Le nom complet d'un opérateur est son type, suivi d'un caractère deux points (« : ») et d'une chaîne de caractères ou d'une liste de symboles. Par exemple infix:<+> est l'opérateur d'addition utilisé dans l'expression 1 + 2. Ou encore postcircumfix:<[ ]>, l'opérateur utilisé dans @array[0].

Vous en savez déjà assez pour définir un nouvel opérateur :

```
multi sub prefix:<%> (Int $x) {    # opérateur double
    2 *  $x;
}
say % 35; # imprime 70
```

Mais ce nouvel opérateur peut aussi s'appeler autrement :

```
multi sub prefix:<deux_fois> (Int $x) {
    2 *  $x;
}
say deux_fois 35; # imprime 70
```

De même, il est possible d'utiliser d'autres délimiteurs pour encadrer le nom de l'opérateur. Les différentes déclarations suivantes indiquent théoriquement toutes le même opérateur d'addition :

```
infix:<+>
infix:<<+>>
infix:«+»
infix:('+')
infix:("+")
```

Apparemment, seules les deux premières syntaxes étaient réellement implémentées dans la version de Rakudo Star utilisant la machine virtuelle Parrot à l'heure où nous écrivions la version initiale de ce document (fin 2014). Avec la machine virtuelle MoarVM en septembre 2015, cela fonctionne correctement :

```
multi sub prefix:«triple» (Int $x) {3 *  $x}
say triple 14;          # -> 42
```

```
multi sub prefix:('tiers') (Int $x) {$x / 3}
say tiers 126;          # -> 42
```

6-2-1 - Précédence

Dans une expression comme $d = $a + $b * $c, la multiplication entre $b et $c est effectuée avant la somme de ce produit avec $a. L'opérateur infix:<*> a une précédence ou priorité d'exécution supérieure à celle de l'opérateur infix:<+>, et c'est pourquoi l'expression est évaluée comme si elle était écrite $a + ($b * $c), conformément aux conventions mathématiques usuelles (voir le tableau des précédences des opérateurs internes de Perl 6 au § **4.3**).

Lorsque l'on définit un nouvel opérateur, il est généralement important de définir sa précédence par rapport aux opérateurs existants :

```
multi sub infix:<toto> is   equiv(&infix:<+>) { ... }

mutli sub infix:<titi> is tighter(&infix:<+>) { ... }

mutli sub infix:<tata> is  looser(&infix:<+>) { ... }
```

Par exemple, on peut écrire ce qui suit :

```
multi sub infix:<double_somme> (Int $x, Int $y) is equiv(&infix:<+>) {
     2 * ($x + $y)
}
say 4 double_somme 5;       # imprime 18

multi sub infix:«3s» (Int $x, Int $y) is equiv(&infix:<*>) {
     3 * ($x + $y)
}
say 9 3s 5;                 # imprime 42
```

6-2-2 - Associativité

L'associativité détermine comment s'évalue la priorité de plus de deux opérateurs ayant la même précédence.

La plupart des opérateurs ne prennent que deux arguments. Mais dans une expression comme $c = 1 / 2 / 4, c'est l'associativité de l'opérateur qui décide dans quel ordre le résultat est évalué. L'opérateur infix</> est associatif à gauche, ce qui veut dire que l'expression est évaluée de la façon suivante : (1 / 2) / 4, ce qui donne 1/8. Si elle était associative à droite, cela donnerait 1 / (2 / 4) = 1/2.

On voit que la différence est cruciale. Pour un opérateur associatif à droite comme infix:<**> (puissance), l'expression 2 ** 2 ** 4 est comprise comme 2 ** (2 ** 4), soit 65 536. Si l'associativité était à gauche, on obtiendrait 256. L'associativité des opérateurs internes de Perl a été donnée au § **4.3** du présent document.

Perl 6 offre plusieurs associativités : none interdit le chaînage des opérateurs ayant la même précédence (par exemple, 2 <=> 3 <=> 4 est proscrit). Et infix:<,> a une associativité de liste. 1, 2, 3 se traduit en infix:<,>(1; 2; 3). Enfin, il y a l'associativité chaînée : $a < $b < $c se traduit en ($a < $b) && ($b < $c).

6-2-3 - Notation postcirconfixée et circonfixée

Les opérateurs postcirconfixés sont des invocations de méthodes.

```
class OrderedHash is Hash {
     method postcircumfix:<{ }>(Str $key) {
          ...
     }
}
```

Si on invoque cette méthode avec un appel du genre $objet{$truc}, $truc sera passé en argument à la méthode et $objet sera accessible via $self.

Les opérateurs circonfixés impliquent généralement une syntaxe différente (par exemple : my @list = <a b c>;), et sont donc implémentés sous la forme de macros :

```
macro circumfix:«< >»($text) is parsed / <-[>]>+ / {
    return $text.comb(rx/\S+/);
}
```

Le trait is parsed est suivi d'une regex qui analyse tout ce qui se trouve entre les délimiteurs. S'il n'y a pas de règle de ce type fournie, alors c'est analysé comme du code Perl ordinaire (mais ce n'est généralement pas ce que vous voulez quand vous introduisez une nouvelle syntaxe). Str.comb recherche les occurrences d'une regex et renvoie une liste des textes reconnus.

6-2-4 - Surcharger les opérateurs existants

La plupart des opérateurs existants (voire tous) sont des fonctions ou des méthodes de type multi, et il est donc facile d'en faire des versions « sur mesure » pour de nouveaux types. Ajouter une fonction multi est la façon de surcharger un opérateur :

```
class MyStr { ... }
multi sub infix:<~>(MyStr $this, Str $other) { ... }
```

Ceci signifie qu'il est possible d'écrire des objets qui se comportent exactement comme les objets « spéciaux » tels que Str, Int, etc.

6-2-5 - Un exemple plus complet

Il est par exemple possible de définir une addition membre à membre entre des paires. Ceci permettrait par exemple de manipuler des nombres complexes. En fait, il existe une classe interne Complex définissant le type complexe (et c'est évidemment elle qu'il faudrait utiliser pour faire des opérations sur les nombres complexes), mais cela va nous permettre de donner un exemple assez complet et parlant de surcharge d'opérateurs arithmétiques.

Voici par exemple comment calculer l'opposé d'un nombre complexe opérateur (- unaire), additionner et multiplier des nombres complexes (définis comme des paires exprimant les formes cartésiennes de nombres complexes) :

```
use v6;

multi sub prefix:<-> (Pair $x) is equiv(&prefix:<->) {
    # opposé d'un nombre complexe
    - $x.key => - $x.value;
}

multi sub infix:<+> (Pair $x, Pair $y) is equiv(&infix:<+>) {
    # somme de nombres complexe
    my $key = $x.key + $y.key;
    my $val = $x.value + $y.value;
    return $key=>$val
}

multi sub infix:<*> (Pair $x, Pair $y) is equiv(&infix:<*>) {
    # produit de nombres complexes
    my $key = $x.key * $y.key - $x.value * $y.value;
    my $val = $x.key * $y.value + $x.value * $y.key;
    return $key=>$val
}

my $a = 4=>3;          # une paire pour le complexe 4 + 3i
```

```
say - $a;               # imprime -4 => -3
my $b = 5=>7;
say $a + $b;            # imprime 9 => 10

my $c = 3.5 => 1/3;     # NB: Perl 6 stocke en interne le rationnel
                        # (classe Rat) 1/3, pas 0.333...
my $d = 1/2 => 2/3;     # idem pour 2/3
say $c + $d;            # imprime 4.0 => 1.0;
say $c +(1/2=>4/3);     # imprime 4.0 => <5/3>;

say $a * $b;            # imprime -1 => 43
say $c * $d;            # imprime <55/36> => 2.5
say $c + $a * $b;       # imprime 2.5 => <130/3>;
                        # bien comme : say $c + ($a * $b);
```

Le dernier exemple montre que les règles de précédence usuelles en mathématiques sont bien respectées (la multiplication est exécutée avant l'addition).

L'utilisation de la surcharge d'opérateurs existants nécessite quelques précautions : dans l'exemple ci-dessus, s'il existait déjà un opérateur `+` ou `*` sur les paires (ou sur un type dont les paires héritent dans la hiérarchie des types), en créer un nouveau générerait une ambiguïté que le compilateur ne pourrait résoudre. On aurait alors un message du type :

```
Ambiguous call to 'infix <+>'; these signatures all match:
:(Pair $x, Pair $y)
:(Pair $x, Pair $y)
in any at ...
```

Il faut donc s'assurer que les signatures permettent au compilateur de choisir la bonne `multi sub`. Nous n'avons pas ce problème ici, car ces opérateurs ne sont pas définis pour des paires. Dans l'hypothèse où ce serait le cas, le problème se résoudrait simplement si, au lieu d'utiliser directement des paires, on créait un type héritant de `Pair` et redéfinissant ces opérateurs pour les objets de la nouvelle classe.

6-3 - Motivation

Permettre à l'utilisateur de déclarer de nouveaux opérateurs et de surcharger des opérateurs existants rend les types définis par l'utilisateur aussi puissants que les types internes. Si les opérateurs internes sont insuffisants, on peut les remplacer par d'autres, plus adaptés à la situation, sans pour autant à avoir à changer quoi que ce soit au compilateur.

Cela réduit aussi le fossé entre « utiliser un langage », et « étendre ou modifier le langage ».

6-4 - Voir aussi

http://perlcabal.org/syn/S06.html#Operator_overloading

http://laurent-rosenfeld.developpez.com/tutoriels/perl/perl6/annexe-02/#L5

Si l'aspect technique vous intéresse et si vous voulez comprendre comment Perl 6 parvient à mettre en œuvre ces surcharges d'opérateur et autres modifications de grammaires, consultez le site suivant: **http://perlgeek.de/en/article/mutable-grammar-for-perl-6**.

7 - La fonction MAIN

7-1 - Résumé

```
$ # programme Perl lance.pl:
$
```

```
$ cat lance.pl
#!/usr/bin/perl6
sub MAIN($path, :$force, :$recursive, :$home = '~/') {
    # commencer le traitement ici
}

$ # ligne de commande shell
$ ./lance.pl --force --home=/home/autre_utilisateur fichier_a_traiter
```

7-2 - Description

Appeler une fonction et lancer un programme Unix depuis la ligne de commande du shell se ressemblent beaucoup : il est possible d'utiliser des arguments positionnels, optionnels et nommés.

Il est possible d'en tirer parti, parce que Perl 6 peut analyser la ligne de commande à votre place en en faisant un appel de fonction. Le script est lancé normalement (c'est le moment où il récupère les arguments de la ligne de commande stockés dans @*ARGS) et la fonction MAIN est immédiatement appelée si elle existe, avec lesdits arguments.

Si la fonction MAIN ne peut être appelée parce que les arguments de la ligne de commande du shell ne concordent pas avec ses paramètres formels, le programme meurt en imprimant un message généré automatiquement rappelant la syntaxe d'appel.

Considérons par exemple le programme (pas très utile) suivant qui imprime le produit des deux nombres entiers qui lui sont passés en argument, et le double de l'entier passé en argument s'il n'y en a qu'un seul :

```
#!/usr/bin/perl6

sub MAIN ( Int :$nombre_1, Int :$nombre_2 = 2) {
    my $result = $nombre_1 * $nombre_2;
    say $result;
}
```

Essayons de l'exécuter avec différents arguments passés à la ligne de commande :

```
~ # appel correct: arguments nombre_1 et nombre_2 sont des Int
~ ./mult_ou_double.pl --nombre_1=7 --nombre_2=5
35

~ # correct: avec des arguments nommés, l'ordre importe peu
~ ./mult_ou_double.pl --nombre_2=7 --nombre_1=5
35

~ # correct (avec second argument remplacé par la valeur 2 par défaut)
~ ./mult_ou_double.pl --nombre_1=7
14

~ # incorrect: le nom de l'argument est inconnu
~ ./mult_ou_double.pl --nombre=7
Usage:
  mult_ou_double.pl [--nombre_1=<Int>] [--nombre_2=<Int>]

~ # incorrect: le second argument n'est pas un entier
~ ./mult_ou_double.pl --nombre_1=7 --nombre_2=5.5
Usage:
  mult_ou_double.pl [--nombre_1=<Int>] [--nombre_2=<Int>]

~ # incorrect: le second argument n'est pas numérique
~ ./mult_ou_double.pl --nombre_1=7 --nombre_2=string
Usage:
  mult_ou_double.pl [--nombre_1=<Int>] [--nombre_2=<Int>]
```

On constate que la signature de la fonction MAIN permet de valider les arguments passés au programme ou de les rejeter. À noter que cela marche aussi dans une fenêtre de commande Windows (mais les utilisateurs Windows

utilisent plutôt rarement la ligne de commande). Si la validation des arguments échoue, la fonction USAGE est appelée, si elle existe, pour informer l'utilisateur de la syntaxe d'appel. Si la fonction USAGE n'existe pas, un message décrivant la syntaxe d'appel est généré à la compilation à partir de la signature de la fonction (ou des fonctions) MAIN, comme on peut le voir dans les trois derniers exemples ci-dessus.

La correspondance entre les options de la ligne de commande et la fonction se fait comme suit :

-nom	:nom
-nom=valeur	:nom<valeur> (ou :nom=valeur)
Souvenez-vous : <...> est analogue à qw(...)	
--hackers=Larry,Damian,Tom	:hackers=Larry,Damian,Tom
--bon-langage	:bon-langage
--bon_langage=Perl	:bon_langage<Perl>
--lang_moins_bons=PHP_Java	:lang_moins_bons<PHP_Java>
+truc	:!truc
+truc=OK	:truc<OK> but false

La syntaxe $x = $obj but false signifie que $x est une copie de l'objet, mais renvoie Bool::False dans un contexte booléen.

Donc, dans les cas simples (et même nettement moins simples), il n'est pas nécessaire de créer une fonction spécifique de traitement des arguments passés au programme, il suffit d'utiliser la signature de la fonction MAIN à cet effet.

Il est également possible de déclarer MAIN comme un ensemble de fonctions multis :

```
multi MAIN (Int $i) {...}    # mon_progr.pl 1
multi MAIN (Rat $i) {...}    # mon_progr.pl 1/2
multi MAIN (Num $i) {...}    # mon_progr.pl 1e6
multi MAIN (Str $i) {...}    # mon_progr.pl toto
```

Le compilateur sélectionnera la version de MAIN dont la signature correspond aux paramètres reçus, si possible.

7-3 - Motivation

La motivation semble évidente. Cela rend les choses simples plus faciles, les choses semblables semblables, et, bien souvent, permet de réduire le traitement de la ligne de commande à une seule ligne de code: la signature de la fonction MAIN.

7-3-1 - Voir aussi

http://perlcabal.org/syn/S06.html#Declaring_a_MAIN_subroutine décrit la spécification.

8 - Les twigils

8-1 - Résumé

```
class Foo {
    has $.bar;
    has $!baz;
}

my @stuff = sort { $^b[1] <=> $^a[1]}, ([1, 2], [0, 3], [4, 8]);
my $block = { say "This is the named 'foo' parameter: $:foo" };
$block(:foo<bar>);
```

```
say "This is file $?FILE on line $?LINE"

say "A CGI script" if %*ENV.exists('DOCUMENT_ROOT');
```

8-2 - Description

En Perl 5, le *sigil* est le caractère dit « de ponctuation » (en fait, généralement $, @, %, &, *) qui précède le nom d'une variable et donne son type. En Perl 6, on parle de *twigil* quand il y a deux caractères de ce type en tête d'une variable. Cela signifie généralement que la variable n'est pas « normale », mais qu'elle diffère d'une façon ou d'une autre ; par exemple, elle peut avoir une portée différente.

Nous avons déjà rencontré ces *twigils* à propos de la **programmation objet** : les attributs publics et privés des objets ont, respectivement les *twigils* . et !. Ce ne sont pas des variables normales, elles sont liées à self.

Le *twigil* ^ permet de supprimer un cas particulier de Perl 5. Pour pouvoir écrire :

Perl 5
```
# code Perl 5
sort { $a <=> $b } @array;
```

il faut que les variables spéciales $a et $b soient exemptées du pragma strict. Perl 6 propose le concept de *paramètre positionnel autodéclaré nommé*. Ce genre de paramètre utilise le *twigil* ^. Cela signifie que ce sont des paramètres positionnels du bloc courant qui n'ont pas besoin d'être déclarés dans la signature. Les variables sont alimentées par ordre lexicographique (pseudoalphabétique) :

```
my $block = { say "$^c $^a $^b" };
$block(1, 2, 3);          # imprime :  3 1 2
```

Il est donc maintenant possible d'écrire :

```
@list = sort { $^b <=> $^a }, @list; # virgule après le bloc
# ou:
@list = sort { $^toto <=> $^titi }, @list;
```

sans qu'il s'agisse de variables spéciales ou de cas particuliers. Par exemple :

```
#!/usr/bin/perl6

my @list = 1, 5, 6, 7, 4, 0, 67, 44;
my @sorted_list_desc = sort {$^toto <=> $^titi}, @list;
say @sorted_list_desc; # affiche : 67 44 7 6 5 4 1 0
say sort {$^titi <=> $^toto}, @list; # affiche 0 1 4 5 6 7 44 67
```

Et pour conserver la symétrie entre arguments positionnels et nommés, le twigil : fait la même chose pour les paramètres nommés, si bien que ces lignes sont à peu près équivalentes :

```
my $block = { say $:stuff };
# ou
my $sub   = sub (:$stuff) { say $stuff };
```

Le twigil ? représente les variables ou constantes qui sont connues au moment de la compilation. Par exemple, $?LINE représente la ligne de code courante (anciennement __LINE__) et $?DATA est le descripteur de fichier (*file handle*) attaché à la section DATA.

On peut accéder aux variables contextuelles grâce au twigil « * », si bien que les variables $*IN et $*OUT peuvent être modifiées dynamiquement.

Le signe < est un pseudotwigil utilisé dans une construction du genre $<capture>, représentant un raccourci pour $/<capture>, qui accède à l'objet reconnu après le succès d'une regex.

8-3 - Motivation

À la lecture du document perlvar de Perl 5, on constate qu'il y a beaucoup trop de variables spéciales, globales pour la plupart, affectant le fonctionnement des programmes de diverses manières.

Les *twigils* essaient de ramener un peu d'ordre dans ces variables spéciales et permettent aussi d'éliminer le besoin de cas particuliers. Dans le cas des attributs d'objets, elles permettent de raccourcir la notation self.var en $.var (ou @.var, etc.).

L'un dans l'autre, le « bruit de ponctuation » augmente certes quelque peu, mais cela rend les programmes nettement plus cohérents et plus lisibles.

9 - Énumérations

9-1 - Résumé

```
enum bit Bool <False True>;
my $value = $arbitrary_value but True;
if $value {
    say "Yes, it's true";       # sera imprimé
}

enum Day ('Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun');
if custom_get_date().Day == Day::Sat | Day::Sun {
    say "Weekend";
}
```

9-2 - Description

Les énumérations sont des bestioles assez versatiles. Elles sont des classes de bas niveau qui consistent en une liste de constantes, habituellement des entiers ou des chaînes de caractères (mais ça peut être des types arbitraires). Elles permettent d'employer un ensemble de symboles pour représenter un ensemble de valeurs. Chaque association d'une énumération est une paire de constantes déclarée avec le mot-clef *enum* et de type Enum. Chaque *enum* associe une clef d'*enum* avec une valeur d'*enum*. Sémantiquement, une *enum* est donc en quelque sorte un hachage de constantes.

Par exemple, la ligne de code suivante :

```
enum E <a b c>;
```

est essentiellement du sucre syntaxique pour :

```
package E {
        constant a = 0;
        constant b = 1;
        constant c = 2;
    }
```

Lorsque les valeurs ne sont pas spécifiées, elles prennent les valeurs 0, 1, 2, etc.

Pour spécifier des valeurs particulières, il faut par exemple employer la syntaxe suivante :

```
my enum chiffres_romains (I => 1,   V => 5,
```

```
                X => 10,  L => 50,
                C => 100, D => 500,
                M => 1000);
say chiffres_romains.enums.kv;
         # imprime: I 1 V 5 X 10 L 50 C 100 D 500 M 1000
```

9-2-1 - Une enum booléenne

On peut par exemple construire une *enum* booléenne comme suit :

```
my enum ouinon <non oui>; # les constantes oui et non prennent vie
say ouinon.enums;         # imprime ("non" => 0, "oui" => 1).hash
say non.pair;                          # imprime "non" => 0
say oui.pair;                          # imprime "oui" => 1
say "Oui vraiment" if oui;             # imprime Oui vraiment
say "Non" if non;                      # n'imprime rien
say "Non, pas du tout" unless non;     # imprime Non, pas du tout
say oui.kv;                            # imprime oui 1 -- kv = key value
say non.defined;                       # imprime True
say oui.WHAT;                          # imprime (ouinon)

say ouinon.enums.keys;                 # imprime non oui
say ouinon.enums.values;               # imprime 0 1
say ouinon.enums.kv;                   # imprime non 0 oui 1
say ouinon.enums.invert;               # imprime 0 => "non" 1 => "oui"
```

À noter que Perl 6 définit de façon interne l'*enum* booléenne suivante :

```
our enum Bool does Boolean <False True>;
```

Ce qui permet d'écrire directement, sans déclaration préalable :

```
say "Oui vraiment" if True;            # imprime Oui vraiment
say "Non, pas du tout" unless False;   # imprime Non, pas du tout
```

Ces constantes peuvent agir comme des sous-types, des méthodes ou des valeurs normales. On peut les associer à un objet avec l'opérateur but, qui « mixe » l'*enum* dans la valeur.

```
enum Jour ('lun', 'mar', 'mer', 'jeu', 'ven', 'sam', 'dim');
my $x = "aujourd'hui" but Jour::mar;
say $x;                          # imprime aujourd'hui
say $x.Jour;                      # imprime mar
```

On peut également utiliser le nom de type de l'énumération comme une fonction, et fournir la valeur en argument :

```
my $x = "aujourd'hui" but Jour(jeu);
say $x;                          # imprime aujourd'hui
say $x.Jour;                     # imprime jeu
```

Dans une énumération, la valeur de la première constante est 0, la suivante 1, et ainsi de suite, sauf si une valeur différente est fournie par une notation par paire explicite :

```
enum Hackers (:Larry<Perl>, :Guido<Python>, :Paul<Lisp>);
say Larry.pair;                  # imprime : "Larry" => "Perl"
# ou encore avec cette autre syntaxe :
my enum chiffres_romains (I => 1,   V => 5,
                X => 10,  L => 50,
                C => 100, D => 500,
                M => 1000);
say chiffres_romains.enums;
# imprime: ("I" => 1, "V" => 5, "X" => 10, "L" => 50, "C" => 100, "D" => 500, "M" => 1000).hash
```

Il est possible de vérifier si une valeur spécifique a été « mixée » dans l'énumération en utilisant l'opérateur de reconnaissance intelligent (*smart match*) ou la méthode .does :

```
if $today ~~ Day::Fri {
    say "Dieu merci, c'est vendredi!"
}
if $today.does(Fri) { ... }
```

À noter que l'on peut se contenter de spécifier le seul nom de la valeur (par exemple ven) uniquement s'il n'est pas ambigu. Dans le cas contraire, il faut fournir le nom complet Jour::ven.

9-3 - Motivation

Les énumérations remplacent à la fois la « magie » associée aux variables « *tainted* » et la sémantique particulière de l'expression « 0 but true » de Perl 5, interprétée comme un zéro en contexte numérique, mais comme vrai en contexte booléen, sans que cela ne provoque d'avertissement. Elles permettent aussi de créer un type booléen.

Les énumérations offrent aussi la puissance et la flexibilité d'associer des métadonnées arbitraires utiles pour le débogage et le traçage.

9-3-1 - Voir aussi

http://perlcabal.org/syn/S12.html#Enumerations

http://laurent-rosenfeld.developpez.com/tutoriels/perl/perl6/annexe-02/#L1-4-1

10 - Unicode

10-1 - Description

Bien que Perl 5 soit depuis plus d'une dizaine d'années l'un des meilleurs langages pour gérer l'Unicode (avec de solides améliorations à presque chaque nouvelle version), le modèle Unicode de Perl 5 souffre d'une sérieuse faiblesse : il utilise le même type pour les données binaires et textuelles. Par exemple, si un programme lit 512 octets sur un socket réseau, il s'agit certainement d'une chaîne d'octets. Cependant, en Perl 5, si on appelle la fonction uc sur cette chaîne d'octets, celle-ci sera traitée comme du texte. Il est recommandé de commencer par décoder la chaîne d'octets, mais quand une fonction reçoit une telle chaîne en argument, elle ne peut savoir à coup sûr si celle-ci a été encodée ou non, autrement dit si elle doit être traitée comme un objet binaire ou comme du texte.

Perl 6 offre au contraire le type buf, qui est une simple collection d'octets, et le type Str, qui est une collection de caractères logiques.

L'expression caractère logique reste cependant assez floue. Pour être plus précis, un objet de type Str peut se concevoir à différents niveaux : octet (*Byte*), point de code (ou *Codepoint*, tous les signes auxquels le Consortium Unicode attribue un nombre est un point de code), un graphème (*Grapheme*, les choses qui ont visuellement l'apparence d'un caractère) et les Charlingua (caractères définis par le langage).

Par exemple la chaîne contenant les octets hexadécimaux 61 cc 80 contient (à l'évidence) trois octets, mais peut aussi être considérée comme composée de deux points de code appelés LATIN SMALL LETTER A (U+0041, lettre minuscule latine « a ») et COMBINING GRAVE ACCENT (U+0300, accent grave, diacritique), ou encore comme le graphème « à ».

Il en résulte qu'il n'est simplement pas possible de demander la taille d'une chaîne. Il faut spécifier le type de taille voulu :

```
$str.bytes;
$str.codes;
$str.graphs;
```

Il existe également une méthode nommée chars, qui renvoie la longueur de la chaîne dans le mode Unicode courant (ce mode peut être sélectionné à l'aide d'un pragma, par exemple use bytes;) et donne par défaut le nombre de graphèmes.

En Perl 5, il pouvait occasionnellement arriver de concaténer accidentellement des chaînes d'octets et de caractères. En cas de problème de ce genre en Perl 6, il est facile d'identifier où et quand le problème se produit en surchargeant l'opérateur de concaténation :

```
sub GLOBAL::infix:<~> is deep (Str $a, buf $b)|(buf $b, Str $a) {
   die "Can't concatenate text string «"
       ~ $a.encode("UTF-8")
         "» with byte string «$b»\n";
}
```

10-1-1 - Encodage et décodage

La spécification du système d'entrées-sorties reste très élémentaire et ne définit pas encore de couches d'encodage ou de décodage, ce qui explique pourquoi cette section n'a pas de résumé ni d'exemples réels de code utilisable. La gestion de l'Unicode est un domaine qui avait pris du retard par rapport au reste de la spécification du langage, une bonne partie de ce retard a été rattrapée, mais certains points restent en suspens (ou, du moins, ne sont pas encore documentés).

Il ne fait guère de doute que cela devra être implémenté, et l'on peut imaginer que l'on arrivera peut-être à une syntaxe de ce type :

```
my $handle = open($filename, :r, :encoding<UTF-8>);
```

10-1-2 - Regex et Unicode

Les regex ont des modificateurs qui spécifient le niveau Unicode. Par exemple, m:codes/./ reconnaîtra exactement un point de code. En cas d'absence de ce modificateur, c'est le niveau Unicode courant qui sera utilisé.

Les classes de caractères telles que \w (un caractère texte) obéiront aux normes Unicode. Il existe des modificateurs qui ignorent la casse (:i) ou les accents (:a) et des modificateurs qui peuvent communiquer l'information de casse à la chaîne de substitution (:samecase, et :sameaccent, ou, en résumé sous une forme abrégée, :ii et :aa).

10-1-3 - Motivation

Il est devenu de nos jours assez difficile de traiter correctement les chaînes de texte avec la plupart des outils et des langages programmation. Par exemple, supposons que vous ayez une application Web en Perl 5 et désiriez couper automatiquement les mots longs pour qu'ils n'altèrent pas votre mise en page. Si vous utilisez naïvement substr, vous risquez de couper accidentellement un graphème en deux.

Perl 6 sera sans doute le premier grand langage de programmation supportant la manipulation des chaînes au niveau des graphèmes, ce qui lève pratiquement toutes les inquiétudes à propos de l'Unicode et fait de Perl 6 (avec **les regexes vues dans la première partie de cet article**) le langage le plus puissant pour la manipulation de chaînes.

L'existence de types de données distincts pour les chaînes de texte et d'octets rend le débogage et l'introspection plus faciles.

10-1-4 - Voir aussi

http://perlcabal.org/syn/S32/Str.html

11 - La portée des variables

11-1 - Résumé

```
for 1 .. 10 -> $a {
    # $a visible ici
}
# $a non visible ici

while my $b = lire_info() {
    # $b visible ici
}
# $b toujours visible ici

my $c = 5;
{
    my $c = $c;
    # $c est indefini (undef) ici
}
# $c vaut 5 ici

my $y;
my $x = $y + 2 while $y = calc();
# $x toujours visible
```

11-2 - Description

11-2-1 - La notion de bloc

Intuitivement, un bloc est le plus souvent un fragment de code entre des accolades ouvrante et fermante, mais parfois l'ensemble de l'unité de compilation, c'est-à-dire le fichier entier du programme, ou éventuellement la chaîne de caractères passée à la fonction eval. C'est du moins plus ou moins ce que l'on pense en venant de Perl 5.

En Perl 6, un bloc peut certes être simplement cela, mais c'est aussi beaucoup plus que cela.

Un bloc est un objet de code, nommé ou anonyme, ayant sa propre portée lexicale et dont le but est notamment la réutilisation du code à petite échelle. Il existe une classe Block, héritant de la classe Code, et dont les sous-classes sont par exemple les classes Routines, Sub, Method, Macro, et même Regex (sous-classe de Method).

```
class Block is Code { }
```

Il n'est cependant pas besoin d'utiliser une syntaxe orientée objet pour définir ou utiliser un bloc. Un bloc est syntaxiquement une liste d'instructions incluses dans une paire d'accolades, ce qui nous ramène en définitive très près de la définition intuitive issue de Perl 5 donnée au début de cette section.

En l'absence d'une signature ou d'arguments positionnels de type variables libres (*placeholders*), un bloc utilise l'argument positionnel $_:

```
my $block = { uc $_; };
say $block.WHAT;            # imprime : Block
say $block('hello');        # imprime : HELLO
```

On constate dans l'exemple ci-dessus que le bloc nommé $block est une forme un peu rudimentaire de fonction ou de référence à une fonction. Pas si rudimentaire que cela, cependant, puisqu'on peut lui donner une signature en bonne et due forme :

```
my $add = -> $a, $b { $a + $b };
say $add(38, 4);            # imprime : 42
```

On retrouve ici le « bloc pointu » décrit dans **la première partie de ce tutoriel**. Les arguments sont ici en lecture seule (non modifiables).

Si la signature est introduite avec la construction <-> (bloc « doublement pointu »), alors les paramètres sont en lecture-écriture par défaut. Par exemple, le bloc $swap ci-dessous permet d'intervertir deux variables :

```
my $swap = <-> $a, $b { ($a, $b) = ($b, $a) };
my ($a, $b) = (2, 4);
$swap($a, $b);
say $a;                     # imprime 4
```

Cela dit, les blocs n'ont pas le type Routine. Et il en résulte qu'ils ne connaissent pas la fonction ou méthode return. Un return d'un bloc se trouvant dans une fonction entraîne la sortie de la fonction, et pas seulement du bloc :

```
sub fonction {
    for 1..10 -> $x {
        say $x;
        return if $x == 5;
    }
    say 15;
}
fonction(); # imprime les chiffres de 1 à 5, mais n'imprime pas 15
```

Le return dans le bloc for fait sortir de la fonction, sans imprimer le 15 quand la variable $x atteint la valeur 5. En remplaçant le return par un last, la fonction imprime 15.

En Perl 6, tout bloc est une fonction anonyme (sauf s'il est nommé, bien sûr, et encore, même si le bloc est nommé, le code qu'il contient peut être considéré comme une fonction anonyme dotée d'une *coderef* au sens de ce mot en Perl 5). Tout bloc est même (au moins potentiellement, mais en un sens assez profond) une fermeture puisqu'il est une fonction qui peut prendre un cliché de son environnement d'exécution lexical. Cette idée était déjà en partie présente ou sous-jacente à certaines constructions en Perl 5, elle est poussée à ses conclusions ultimes en Perl 6. Si bien qu'en Perl 6, on distingue plusieurs sortes de fonctions : les *subroutines* ou *sub* (introduites avec le mot-clef sub), les méthodes, les macros et les blocs et lambdas.

C'est pourquoi les blocs ont des paramètres, des signatures, etc. Perl 6 supporte les concepts des langages fonctionnels encore plus que Perl 5 ne le faisait, mais il n'est pas intégriste de la programmation fonctionnelle pour autant, puisqu'il existe un type et une classe Block offrant une petite vingtaine de méthodes, on est donc aussi dans un modèle de programmation objet.

11-2-2 - Portée lexicale

En Perl 6, la portée des variables est assez semblable à celle de Perl 5. Un bloc (au sens large, y compris par exemple le bloc définissant une fonction) introduit une nouvelle portée. Un nom de variable est recherché d'abord dans la portée la plus interne ; si elle n'y est pas trouvée, alors il est recherché dans la portée immédiatement plus large, et ainsi de suite. Comme en Perl 5, une variable déclarée avec my est une variable lexicale (« variable locale ») valable et une déclaration utilisant our introduit un alias lexical pour une variable de paquetage (« variable globale »).

Mais il y a des différences assez subtiles. Les variables sont visibles dans le reste du bloc à l'intérieur duquel elles sont déclarées, mais, contrairement à ce qui se passe en Perl 5, la portée des variables déclarées dans l'entête d'un bloc (par exemple dans la condition d'une boucle while) ne se limite pas au bloc qui suit cet entête.

Ainsi, par exemple, à la différence de ce qui se passerait en Perl 5, le programme suivant :

```
use v6;

my @liste = 1..10;
while my $i = shift @liste {
    last if $i > 4;
    say "La valeur du compteur dans la boucle est $i";
}
say "Valeur du compteur hors de la boucle: $i";
```

affiche bien la valeur de la variable compteur après la fin de la boucle :

```
perl6 count.pl
La valeur du compteur dans la boucle est 1
La valeur du compteur dans la boucle est 2
La valeur du compteur dans la boucle est 3
La valeur du compteur dans la boucle est 4
Valeur du compteur hors de la boucle: 5
```

Mais cette règle ne s'applique pas au paramètre d'un « bloc pointu » d'une boucle for. Le programme suivant :

Ne fonctionne pas

```
for 1..5 -> $j {
    say "La valeur du compteur dans la boucle est $j";
}
say "Valeur du compteur hors de la boucle: $j"; # ERREUR
```

produit des erreurs de compilation :

```
===SORRY!=== Error while compiling count.pl
Variable '$j' is not declared
at count.pl:7
```

parce que la variable $j est en réalité le paramètre (autodéclaré) du bloc pointu, et non une variable lexicale déclarée avec my.

Perl 6 recherche systématiquement les noms non qualifiés (variables et fonctions) dans les portées lexicales.

Pour limiter la portée, on peut utiliser les paramètres formels d'un bloc :

```
if calc() -> $result {
    # $result est accessible ici
}
# $result non visible ici
```

Une autre différence est que les variables sont visibles dès qu'elles sont déclarées, sans devoir attendre la fin de l'instruction comme en Perl 5.

```
my $x = .... ;
        ^^^^^
        $x est visible ici en Perl 6
        mais pas en Perl 5
```

Cette différence est sans doute assez subtile, mais conduit à clarifier la syntaxe dans certains cas particuliers. En Perl 5, il est possible (mais sans doute ni très clair, ni très recommandé) d'écrire dans un bloc :

Code Perl 5

```
my $x = $x;           # ne marche pas en Perl 6
```

ce qui a pour effet de déclarer une version locale au bloc de la variable $x et de lui attribuer (sans doute temporairement) la valeur de la variable $x à l'extérieur du bloc. En Perl 6, ceci ne fonctionnera pas, car l'instruction d'affectation ne verra plus la valeur de la variable $x à l'extérieur du bloc. Cela n'empêche en rien de récupérer la valeur de $x à l'extérieur du bloc, mais il faut le faire plus explicitement (et donc bien plus clairement) en utilisant par exemple le pseudopaquetage OUTER présenté un peu plus bas (§ **11.2.5**).

11-2-3 - Portée dynamique

Le qualificatif local de Perl 5 s'appelle désormais temp (ce qui reflète mieux son rôle), et, s'il n'est pas suivi par une initialisation, préserve la valeur que la variable possédait auparavant (donc n'affecte pas la valeur à undef comme en Perl 5).

Il existe aussi un nouveau type de variable à portée dynamique, qui s'appelle variable *hypothétique*, introduite avec l'opérateur préfixé let. Sa sémantique est la suivante : si le bloc sort en exception, alors la valeur précédente de la variable est restaurée ; sinon, la valeur courante est conservée.

11-2-4 - Variables de contexte

Certaines variables spéciales qui sont globales en Perl 5 ($!, $_) deviennent des variables *contextuelles* en Perl 6, ce qui veut dire qu'elles peuvent être transférées entre des portées dynamiques.

Ceci permet de résoudre un vieux problème, assez rare, mais gênant et souvent difficile à identifier, de Perl 5. En Perl 5, il est possible qu'une fonction DESTROY soit appelée à la sortie d'un bloc et qu'elle change accidentellement la valeur d'une variable globale, par exemple celle d'une des variables d'erreur :

Code Perl 5 erroné

```
# Code Perl 5 erroné
sub DESTROY { eval { 1 }; }

eval {
    my $x = bless {};
    die "Mort\n";
};
print $@ if $@;          # ERREUR: n'imprime rien
```

Le problème ne se pose pas en Perl 6 qui n'utilise pas implicitement des variables globales.

(À noter qu'à partir de Perl 5.14, il existe une solution de contournement qui empêche la modification intempestive de la variable $@ et résout donc le problème particulier décrit ci-dessus.)

11-2-5 - Les pseudopaquetages

Si une variable est cachée par une autre variable lexicale de portée plus étroite portant le même nom, il reste possible d'y accéder en utilisant le pseudopaquetage OUTER.

```
my $x = 3;
{
    my $x = 10;
    say $x;              # imprime 10
    say $OUTER::x;       # imprime 3
    say OUTER::<$x>      # imprime 3
}
```

De même, une fonction peut accéder aux variables de la fonction appelante à l'aide des pseudopaquetages CALLER et CONTEXT. La différence est que CALLER n'accède qu'à la portée de la fonction immédiatement appelante, alors que CONTEXT fonctionne plus comme les variables d'environnement Unix et ne devrait être utilisé qu'internement

par le compilateur pour gérer les variables spéciales du genre $_, $! et ainsi de suite. Pour accéder aux variables de la portée dynamique externe, il faut les déclarer avec la syntaxe is context.

11-3 - Motivation

Il est généralement admis que les variables globales sont dangereuses et à la source de nombreux problèmes. Il reste parfois tentant, en Perl 5, de déclarer des variables globales, notamment pour des tableaux ou hachages servant de paramétrage général à une application. Nous avons à notre disposition les ressources pour mettre en œuvre un meilleur mécanisme de contrôle de la portée des variables. Cela permet de n'utiliser des variables globales que pour les données qui sont intrinsèquement globales (comme %*ENV et $*PID).

Les règles régissant la portée lexicale dans les blocs ont été fortement simplifiées.

Voici une citation du document **Perlsyn** de Perl 5 :

Note : le comportement d'une instruction my, state ou our modifiée par un modificateur conditionnel ou une construction de boucle (par exemple my $x if ...) est indéfini. La valeur de la variable peut être undef, la valeur précédemment affectée ou n'importe quoi d'autre. Ne dépendez pas d'un comportement particulier. Les prochaines versions de Perl feront peut-être quelque chose de différent que celle que vous utilisez actuellement.

Nous ne voulons pas d'un comportement aussi compliqué en Perl 6.

11-4 - Voir aussi

S04 discute de la portée des blocs : **http://perlcabal.org/syn/S04.html**.

S02 énumère tous les pseudo-paquetages et explique la portée contextuelle :

http://perlcabal.org/syn/S02.html#Names.

12 - Remerciements

Je remercie **Djibril** et **ClaudeLeloup** pour leur relecture attentive de ce tutoriel et leurs très utiles suggestions d'amélioration. Merci également à **ptonnerre** et à Sébastien Dorey pour les suggestions d'amélioration qu'ils m'ont envoyées après publication et qui m'ont permis d'améliorer le contenu.

1 : Cet opérateur défini lors de l'élaboration de Perl 6 a été jugé suffisamment intéressant pour être introduit en Perl 5.10.1, et n'est donc pas à proprement parler une nouveauté de Perl 6. Toutefois, l'implémentation en Perl 5 ayant posé des problèmes, cet opérateur est considéré comme « expérimental » dans les dernières versions de Perl 5 (5.18 et suivantes) , de même que la construction `given/when` s'appuyant sur cet opérateur. Autrement dit, il est susceptible d'être profondément remanié, voire supprimé de Perl 5.
2 : Ou `~|` en contexte de chaîne de caractères ou `?|` en contexte booléen.
3 : Ce **tutoriel sur la programmation fonctionnelle en Perl** aborde la question de comment réaliser des évaluations paresseuses en Perl 5. En particulier, il propose la construction d'opérateurs paresseux **lazy_map et lazy_grep** en Perl 5. Le lecteur pourra y trouver un éclairage complémentaire sur les avantages de la paresse.
4 : Une discussion supplémentaire en français de cette question et une solution en Perl 5 figurent **dans ce tutoriel**.