

# De Perl 5 à Perl 6 - annexe 1

## Ce qui change entre Perl 5 et Perl 6



Par Laurent Rosenfeld 

Date de publication : 3 septembre 2015

Dernière mise à jour : 6 novembre 2015

DÉBUTANT

Ce document fait suite à une série de trois articles décrivant les principaux changements entre la version 5 de Perl, une vénérable dame qui a commencé sa carrière il y a plus de 20 ans (en 1994), et la nouvelle mouture, Perl 6, radicalement nouvelle et bien plus moderne et plus expressive, qui devrait sortir en version de production avant la fin de l'année 2015.

Cette série d'articles décrivait en détail de nombreuses différences entre les deux versions du langage et de nombreuses nouveautés de Perl 6.

La présente annexe 1 vise à résumer succinctement les différences syntaxiques et sémantiques afin de constituer une référence de poche (incomplète), une sorte d'« antisèche » permettant au lecteur de retrouver rapidement un élément de syntaxe qui lui échapperait.

L'**annexe 2** examinera les nouveautés du langage, et elle est à notre humble avis bien plus importante pour le lecteur qui désire apprendre cette nouvelle version du langage.

*Une discussion sur ce tutoriel est ouverte sur le forum Perl à l'adresse suivante : **Commentez** .*

En complément sur Developpez.com

- De Perl 5 à Perl 6 - Partie 2 : les nouveautés, un tutoriel de Moritz Lenz et Laurent Rosenfeld
- De Perl 5 à Perl 6 - Partie 1 : les bases du langage, un tutoriel de Moritz Lenz et Laurent Rosenfeld
- De Perl 5 à Perl 6 - Partie 3 : Approfondissements
- Les regex et les grammaires de Perl 6 : une expressivité sans précédent
- De Perl 5 à Perl 6 - Annexe 2 : les nouveautés du langage


Introduction.....	5
1 - Syntaxe générale.....	5
1-1 - Commentaires.....	5
1-2 - Appels de méthode.....	6
1-3 - Espaces blancs.....	6
1-4 - Démêler le vrai du faux.....	7
1-5 - Les sigils.....	7
1-6 - Accès aux valeurs d'un tableau.....	7
1-7 - Accès aux valeurs d'un hachage.....	8
1-8 - Créations de références.....	8
1-9 - Déréférencement.....	8
1-10 - Documentation intégrée (Pod).....	9
2 - Les opérateurs.....	9
2-1 - Opérateurs de comparaison pour les tris (cmp et <=>).....	10
2-2 - Opérateurs de liaison et opérateur de reconnaissance intelligente (ou smart match, ~~).....	10
2-3 - Opérateurs bit à bit.....	10
2-4 - Opérateur flèche (->).....	11
2-5 - La « grosse virgule » (=>).....	11
2-6 - Opérateur de concaténation.....	12
2-7 - Opérateur ternaire « ? : » remplacé par « ?? !! ».....	12
2-8 - Opérateur de répétition (de liste ou de chaîne).....	12
2-9 - Opérateurs de citation et assimilés.....	12
2-10 - Opérateurs d'entrées-sorties (E/S ou IO).....	14
2-11 - Deux points (« .. ») et trois points (« ... », création d'intervalles et opérateur flip-flop.....	14
2-12 - Voir aussi.....	15
3 - Instructions composées.....	15
3-1 - Conditions.....	15
3-1-1 - Conditions if elsif else unless.....	15
3-1-2 - Branchements multiples given-when.....	16
3-2 - Boucles.....	16
3-2-1 - Boucles while et until.....	16
3-2-2 - Boucles for et foreach.....	17
3-2-3 - Instructions de contrôle de flux.....	17
3-3 - Voir aussi.....	18
4 - Fonctions internes.....	18
4-1 - Fonctions utilisant des blocs de code nus.....	18
4-2 - Fonctions delete et exists.....	18
4-3 - Liste de fonctions.....	18
4-3-1 - Généralités.....	18
4-3-2 - Liste alphabétique de fonctions Perl 5 et équivalences.....	19
5 - Expressions régulières (regex).....	33
5-1 - Les opérateurs =~ et !~ deviennent ~~ et !~~.....	33
5-2 - Déplacement des modificateurs.....	34
5-3 - Espaces dans les motifs.....	34
5-4 - Utiliser le modificateur :P5 ou l'adverbe :Perl5.....	34
5-5 - Les reconnaissances spéciales utilisent souvent une syntaxe avec des chevrons <>.....	35
5-6 - Reconnaissance du motif le plus long dans les alternatives.....	35
5-7 - Voir aussi.....	35
6 - Pragmas.....	35
6-1 - Modules strict et warnings.....	35
6-2 - Module autodie.....	36
6-3 - Modules base et parent.....	36
6-4 - Modules bigint, bignum et bigrat.....	36
6-5 - Module constant.....	36
6-6 - Module mro.....	37
6-7 - Module utf8.....	37
6-8 - Module vars.....	37
7 - Options de la ligne de commande.....	37

8 - Opérations sur des fichiers.....	37
8-1 - Lire les lignes d'un fichier dans un tableau.....	37
9 - Variables spéciales.....	38
9-1 - Variables d'environnement.....	38
9-1-1 - Chemin de la bibliothèque des modules Perl.....	38
9-2 - Variables spéciales générales.....	39
9-3 - Variables associées aux expressions régulières.....	39
9-4 - Variables associées aux descripteurs de fichiers.....	40
10 - Modules.....	40
10-1 - Importation de fonctions depuis un module.....	40
10-2 - Modules standard.....	42
10-2-1 - Module Data::Dumper.....	42
10-2-2 - Module Getopt::Long.....	43
11 - Traduction automatique de Perl 5 en Perl 6.....	44
11-1 - Blue Tiger.....	44
11-2 - Perlito.....	44
11-3 - MAD.....	44
11-4 - Module Inline::Perl5.....	44
11-5 - Module Inline::Perl6.....	45
12 - Remerciements.....	45

## Introduction

La présente annexe 1 du tutoriel *De Perl 5 à Perl 6* décrit les changements qu'il va falloir apporter à un programme Perl 5 pour qu'il fonctionne en Perl 6. Il peut donc servir de guide de traduction (assez complet, mais non exhaustif) de Perl 5 en Perl 6.

Le but est de permettre au lecteur de « traduire » un programme Perl 5 en Perl 6 (bien que ce ne soit pas toujours une bonne idée, il vaut souvent mieux réécrire, mais ne soyons pas dogmatiques sur ce point, les situations réelles ne sont pas toujours si simples). Tout au moins, nous espérons être utile au lecteur qui se dit : « Ah, zut, ça, je sais le faire en Perl 5, comment faire en Perl 6 ? »

Les sources employées pour la rédaction de ce document sont multiples : outre le contenu du tutoriel dont nous reprenons quelques parties (il y aura quelques redites, mais très peu, espérons-nous, il y aura surtout beaucoup de points de détail nouveaux), nous avons abondamment utilisé la  **documentation officielle de Perl 6** en anglais, qui s'est considérablement étoffée depuis la parution de la première partie du tutoriel (il existe encore quelques lacunes, que nous signalerons à l'occasion, mais elle est maintenant assez complète). Certains passages sont des adaptations en français, voire occasionnellement de simples traductions, de la documentation officielle, mais celle-ci étant anonyme, je ne peux rendre à ses auteurs l'hommage qu'ils mériteraient sinon.

### Support de l'Unicode en Perl 6

*Lorsque nous avons rédigé les trois parties du tutoriel (juin à décembre 2014), nous avons signalé que la version de Rakudo Star disponible à l'époque ne supportait pas l'Unicode, si bien que tous nos exemples évitaient soigneusement les lettres françaises accentuées, trémas, cédilles, guillemets français, etc., même dans les chaînes de caractères ou les commentaires.*



*La situation a bien évolué depuis, et Rakudo / Perl 6 supporte maintenant (août 2015, et probablement depuis au moins mars 2015) des caractères non ASCII (caractères accentués, signes diacritiques, symboles divers, émoticônes (smileys), lettres étrangères, etc.) aussi bien dans les chaînes de caractères et les commentaires que dans les identifiants de variables, de méthodes ou de fonctions, et ce, sans la moindre déclaration préalable.*

*Le lecteur pourra constater que de nombreux exemples de ces annexes illustrent cette faculté.*

## 1 - Syntaxe générale

### 1-1 - Commentaires

```
use v6; # utilisation de Perl 6

# ceci est un commentaire (même chose qu'en Perl 5)

#{ Ceci
est un commentaire multiligne.
Ce commentaire est ouvert avec une accolade ouvrante.
Il faut le fermer avec une accolade fermante.
Si on l'avait ouvert avec un crochet ouvrant,
il faudrait le fermer avec un
crochet fermant. }

my $pi #`{{{
Commentaire multiligne utilisant plusieurs accolades (ou crochets)
→ Ce commentaire est ouvert avec trois accolades ouvrantes.
→ Il faut le fermer avec trois accolades fermantes }.
}}}= 3.14159;
```

```
say "Pi is: $pi"; # Affiche : 3.14159
```

Voir aussi le chapitre **Commentaires** de la première partie du tutoriel.

## 1-2 - Appels de méthode

Pour les appels de méthodes, la flèche « `->` » est remplacée par le point « `.` » (la concaténation, qui utilisait le point, est maintenant faite avec le tilde « `~` »).

```
$personne->nom # Perl 5
$personne.nom # Perl 6

# Appel d'une méthode dont le nom n'est connu qu'à l'exécution:

$objet->$methodname(@args); # Perl 5
$objet."$methodname"(@args); # Perl 6
```

## 1-3 - Espaces blancs

Perl 5 est très laxiste sur les espaces blancs : on peut en mettre ou ne pas en mettre presque partout.

Perl 6 ne veut pas limiter la créativité du programmeur, mais le besoin de pouvoir établir une grammaire cohérente, déterministe et extensible travaillant en une seule passe et fournissant des messages d'erreurs qui soient une réelle aide au programmeur a conduit à adopter un compromis. Il en résulte qu'il y a quelques endroits où les espaces blancs sont obligatoires et d'autres où ils sont proscrits.

### Pas d'espace autorisé avant la parenthèse ouvrant une liste d'arguments

```
substr ($s, 4, 1); # Perl 5 (en Perl 6 ce code essaierait de passer un
# seul argument de type List à substr)
substr ($s, 4, 1); # Perl 6
substr $s, 4, 1; # Perl 6 - le + simple est d'omettre les parenthèses
```

### Espace obligatoire après les mots-clefs

```
my($alpha, $bravo); # Perl 5. En Perl 6, essaie d'appeler
# la fonction my()

my ($alpha, $bravo); # Perl 6

if($a < 0) { ... } # Perl 5, erreur en Perl 6
if ($a < 0) { ... } # Perl 6
if $a < 0 { ... } # Perl 6, plus idiomatique sans parenthèses

while($x-- > 5) { ... } # Perl 5, erreur en Perl 6
while ($x-- > 5) { ... } # Perl 6
while $x-- > 5 { ... } # Perl 6, plus idiomatique sans parenthèses
```

### Pas d'espace autorisé après un opérateur préfixé ou avant un opérateur postfixé ou postcircfixé (y compris pour indices des tableaux et des hachages)

```
$seen {$_} ++; # Perl 5
%seen{$_}++; # Perl 6
```

À noter cependant que l'on peut utiliser l'opérateur *unspace* « `\` » pour ajouter des espaces (et même des commentaires) presque n'importe où :

```
# Perl 5
my @books = $xml->parse_file($file) # commentaire quelconque
```

```
->findnodes("/library/book");

# Perl 6
my @books = $xml.parse-file($file)\           # commentaire quelconque
    .findnodes("/library/book");
```

## 1-4 - Démêler le vrai du faux

Perl 5 et Perl 6 ont à peu près la même notion du vrai et du faux (les nombres 0, 0.0 et la chaîne vide sont faux, presque tout le reste est vrai), à cette différence près que, contrairement à Perl 5, Perl 6 considère la chaîne "0" comme vraie.

Perl 6 possède un type booléen (Bool) définissant des valeurs True et False.

Il n'y a pas de valeur undef en Perl 6. Une variable déclarée, mais non initialisée, est évaluée à son type. Mais le type sans valeur définie renvoie une valeur fausse dans un contexte booléen, si bien qu'une variable non initialisée renvoie une valeur fausse, comme le fait undef en Perl 5.

```
my $x; say $x;           # affiche: (Any)
my Int $i; say $i;       # affiche: (Int)
say "Défini" if $i;       # n'affiche rien, $i est évalué à False
say "Non défini" unless $i; # affiche "Non défini"
```

## 1-5 - Les sigils

- En Perl 5, les tableaux et les hachages ont des *sigils* (signes précédant le nom des variables, comme \$, @, %, etc.) qui changent selon la manière dont on accède à la variable. Ce n'est plus le cas en Perl 6, les sigils sont invariants, on peut considérer qu'ils font partie du nom de la variable.
- Le sigil \$ est utilisé pour les variables scalaires (« une seule chose ») et n'est plus utilisé pour accéder aux éléments individuels d'un tableau ou d'un hachage.
- Le sigil @ est utilisé pour accéder aux variables de type tableau (par exemple : @mois, @mois[2], @mois[2, 4]) et n'est plus utilisé pour les tranches de hachage.
- Le sigil % est utilisé pour accéder aux variables de type hachage (par exemple : %calories, %calories<pomme>, %calories<poire prune>) et n'est plus utilisé pour les tranches clef-valeur de tableaux.
- Le sigil & est utilisé de façon cohérente (et sans l'aide d'un antislash « \ ») pour prendre une référence sur une fonction ou un opérateur nommé sans l'invoquer.

```
my $sub = \&toto; # Perl 5
my $sub = &toto;  # Perl 6

callback => sub { say @_ } # Perl 5: ne peut passer directement la sub
callback => &say           # Perl 6: & transforme une fonction en nom
```

Voir aussi le **chapitre 2** de la première partie de ce tutoriel.

## 1-6 - Accès aux valeurs d'un tableau

Les opérations d'accès aux valeurs d'un tableau ne modifient plus le sigil « @ », quel que soit le mode d'accès.

```
# accès à un élément d'un tableau
say $mois[2]; # Perl 5
say @mois[2]; # Perl 6 - @ au lieu de $

# tranche d'un tableau
say join ', ', @mois[6, 8..11]; # Perl 5 et Perl 6

# tranche par clef-valeur
say join ', ', %mois[6, 8..11]; # Perl 5
say join ', ', @mois[6, 8..11]:kv; # Perl 6 - @ au lieu de %;
                                   # utilisation de l'adverbe :kv
```

## 1-7 - Accès aux valeurs d'un hachage

Les opérations d'accès aux valeurs d'un hachage ne modifient plus le sigil « % », quel que soit le mode d'accès.

```
say $calories{"pomme"}; # Perl 5
say %calories{"pomme"}; # Perl 6 - % au lieu de $

say $calories{pomme}; # Perl 5
say %calories<pomme>; # Perl 6 - chevrons; % au lieu de $

# Tranches
say join ', ', @calories{'poire', 'prune'}; # Perl 5
say join ', ', %calories{'poire', 'prune'}; # Perl 6 - % au lieu de @
say join ', ', %calories<poire prune>;      # Perl 6 (version + concise)

# Tranches clefs-valeurs
say join ', ', %calories{'poire', 'prune'}; # Perl 5
say join ', ', %calories{'poire', 'prune':kv}; # Perl 6 - avec:kv
say join ', ', %calories<poire prune>:kv;    # Perl 6 (plus propre)
```

## 1-8 - Créations de références

Les créateurs de références vers des tableaux [...] et des hachages {...} restent inchangés.

En Perl 5, les références vers des tableaux, hachages ou fonctions sont renvoyées lors de leur création. Les références vers des variables ou fonctions existantes sont créées à l'aide de l'opérateur « \ ».

En Perl 6, il en va de même pour les tableaux, hachages et fonctions anonymes. Mais les références vers des fonctions nommées sont générées en préfixant le nom de la fonction avec le sigil « & ». On utilise le contexte item pour générer des références vers des variables nommées.

```
my $aref = [ 1, 2, 9 ]; # Perl 5 et Perl 6
my $href = { A => 98, Q => 99 }; # Perl 5 et Perl 6

my @aaa = <1 4 6>;
my $aref = \@aaa; # Perl 5
my $aref = item(@aaa); # Perl 6
# ou :
my $aref = @aaa.item; # Perl 6, notation objet/méthode

my $href = \%hhh; # Perl 5
my $href = item(%hhh); # Perl 6

my $sref = \%foo; # Perl 5
my $sref = &foo; # Perl 6
```

## 1-9 - Déréférencement

En Perl 5, la syntaxe pour déréférencer une référence entière est d'utiliser le sigil du type voulu et des accolades autour de la référence.

En Perl 6, les accolades sont remplacées par des parenthèses.

```
# Perl 5
say ${$scalar_ref};
say @{$arrayref };
say keys %{$hashref };
say &{$subref };

# Perl 6
say ($$scalar_ref);
say (@$arrayref );
```



```
say keys %($hashref);
say      &($subref);
```

À noter qu'en Perl 5 comme en Perl 6, les accolades ou parenthèses sont souvent optionnelles, mais leur omission peut rendre le code moins clair.

En Perl 5, l'opérateur flèche « -> » est utilisé pour un accès unique à une référence composée ou pour appeler une fonction à l'aide de sa référence. En Perl 6, c'est l'opérateur point « . » qui joue ce rôle :

```
# Perl 5
say $arrayref->[7];
say $hashref->{'Tartempion'};
say $subref->($toto, $titi);

# Perl 6
say $arrayref.[7];
say $hashref.{'Tartempion'};
say $subref.($toto, $titi);
```

## 1-10 - Documentation intégrée (Pod)

La documentation *Pod* (qui s'appelait POD, *Plain Ol' Documentation*, tout en lettres capitales, en Perl 5) a subi des changements entre Perl 5 et Perl 6. La principale différence est qu'il faut insérer un Pod entre des directives `=begin pod` et `=end pod`. Plus généralement, le Pod de Perl 6 est conçu pour être plus régulier et uniforme, un peu plus compact et nettement plus expressif que le POD de Perl 5, tout en limitant dans la mesure du possible les différences.

Les autres modifications sont plus des détails qui peuvent néanmoins s'avérer irritants quand on est habitué au POD de Perl 5.

Le mieux est sans doute d'utiliser l'interpréteur Perl pour vérifier son Pod, en utilisant l'option `--doc` de la ligne de commande, par exemple `perl6 --doc pod_quelconque.pod`, ce qui affichera tout problème sur la sortie d'erreur. (Selon l'emplacement où Perl 6 est installé et la façon dont il a été installé, il peut être nécessaire de spécifier l'emplacement de Pod::To::Text.)

La documentation Pod est un point suffisamment important pour mériter ce paragraphe, mais nous ne pouvons vraiment pas entrer dans les détails ici.

Pour plus de détails sur le Pod de Perl 6, voir la [🚩Synopsis S26](#).

## 2 - Les opérateurs

Les opérateurs suivants ne sont pas modifiés :

,	Séparateur de liste
+ - * /	Addition, soustraction, multiplication et division numériques
= += -= *= **=	Affectation (avec éventuellement addition, soustraction, etc.)
%	Modulo numérique (reste de la division entière)
**	Élévation à la puissance
++ --	Incréméntation et décréméntation numériques
&&    ^	Opérateurs booléens, haute précédence
not and or xor	Opérateurs booléens, basse précédence
//	« Défini ou » logique. Il existe aussi une version de basse précédence, <code>orelse</code> .
== != < > <= >=	Comparaisons numériques
eq ne lt gt le ge	Comparaisons de chaînes de caractères

Les règles de précédence et d'associativité ont peu changé pour les opérateurs qui ont été maintenus entre les deux versions de Perl. Un tableau de précédence est donné au chapitre **Précédence des opérateurs** du tutoriel. Voir aussi [http://...#Operator\\_Precedence](http://...#Operator_Precedence).

## 2-1 - Opérateurs de comparaison pour les tris (cmp et <=>)

En Perl 5, ces opérateurs renvoient -1, 0 ou 1. En Perl 6, ils renvoient `Order::Increase`, `Order::Same`, or `Order::decrease`. Cela ne fait une différence que si nous voulons écrire notre propre fonction personnalisée de comparaison.

L'opérateur `cmp` de Perl 5 est remplacé par `leg`, qui force une comparaison en contexte de chaîne.

L'opérateur `<=>` est inchangé et force une comparaison en contexte numérique.

En Perl 6, l'opérateur `cmp` se comporte soit comme `cmp`, soit comme `<=>`, selon le type de ses arguments.

## 2-2 - Opérateurs de liaison et opérateur de reconnaissance intelligente (ou smart match, ~~)

Les opérateurs de liaison (pour les regex) `<=~>` et `<!~>` sont remplacés respectivement par `<~~>` et `<!~~>`.

L'opérateur `~~` de reconnaissance intelligente de Perl 5 n'a pas changé sur le fond, mais les règles gouvernant ce qui est reconnu dépendent du type des arguments, et, les types ayant été très approfondis en Perl 6, ces règles sont loin d'être les mêmes en Perl 5 et en Perl 6.

Surtout, tout semble indiquer que l'opérateur de comparaison intelligente fonctionne de façon satisfaisante en Perl 6, alors que son implémentation en Perl 5 laisse suffisamment à désirer pour que son utilisation soit dépréciée (ou plus exactement qu'il soit déclaré « expérimental ») dans les dernières versions de Perl 5.

Pour plus de détails sur le fonctionnement de `<~~>`, voir [Smart Matching](#).

## 2-3 - Opérateurs bit à bit

Comme en Perl 5, les opérateurs `<!>` et `<->` assurent respectivement la négation logique et arithmétique (opérateurs non bit à bit).

En Perl 5, le comportement des opérateurs `&` `|` `^` dépendait du type des arguments : par exemple, `31 | 33` ne renvoie pas la même chose que `"31" | "33"`.

En Perl 6, ces opérateurs à un seul caractère ont été supprimés et sont remplacés par des opérateurs à deux caractères qui forcent le contexte approprié.

```
# Opérateurs infixés (deux arguments, un de chaque côté de l'opérateur)
+& +| +^ Et, Ou, Ou exclusif: Numérique
~& ~| ~^ Et, Ou, Ou exclusif: Chaîne de caractères
?& ?| ?^ Et, Ou, Ou exclusif: Booléen

# Opérateurs préfixés (un argument, après l'opérateur)
+^ Non: Numérique
~^ Non: Chaîne de caractères
?^ Non: Booléen (même chose que l'opérateur !)
```

Les opérateurs de décalage de bits `<<` et `>>` sont remplacés par `+<` et `+>` :

```
say 42 << 3; # Perl 5
say 42 +< 3; # Perl 6
```

## 2-4 - Opérateur flèche (->)

Il est beaucoup moins fréquent en Perl 6 qu'en Perl 5 d'utiliser des références, si bien que l'on a aussi moins souvent besoin d'un opérateur de déréférencement. Mais en cas de besoin, l'opérateur de déréférencement est le point « . » et non plus la flèche « -> » (voir aussi § 2.9), et le point remplace aussi la flèche pour les appels de méthodes. Donc, une construction \$tabl\_ref->[3] de Perl 5 devient \$tabl\_ref.[3] en Perl 6.

## 2-5 - La « grosse virgule » (=>)

En Perl 5, « => » agissait comme une virgule et transformait aussi ce qui précédait en une chaîne de caractères.

En Perl 6, « => » est l'opérateur de paire (type Pair), ce qui est assez différent en principe, mais fonctionne de la même façon dans un bon nombre de situations.

Par exemple, pour initialiser un hachage ou passer des arguments à une fonction attendant une référence à un hachage, l'utilisation est la même :

```
# Marche en Perl 5 et en Perl 6
my %hachage = ( AAA => 1, BBB => 2 );
prends_le_butin( 'diamants', { niveau => 'élevé', nombre => 9 } );
# Noter les accolades, le second argument est un hashref
```

Mais si vous utilisiez cet opérateur pour ne pas devoir mettre de guillemets sur une partie d'une liste, ou pour passer une liste plate d'arguments de type CLEF, VALEUR, CLEF, VALEUR, alors il y a de bonnes chances qu'utiliser ainsi => ne fonctionne pas comme prévu. La solution de contournement consiste à remplacer la grosse virgule par une virgule ordinaire et à ajouter des guillemets ou des apostrophes à son argument de gauche. Ou vous pouvez modifier l'API de la fonction pour qu'elle accepte un hachage en argument.

Une meilleure solution à long terme est de modifier l'API de la fonction pour qu'elle accepte des paires. Mais cela oblige à modifier d'un seul coup tous les appels de la fonction.

```
# Perl 5
sub prends_le_butin {
    my $butin = shift;
    my %options = @_;
    # ...
}
prends_le_butin( 'diamants', niveau => 'élevé', nombre => 9 );
# Noter: pas d'accolade cette fois

# Perl 6, API d'origine
sub prends_le_butin ( $butin, *%options ) {
    # L'astérisque * signifie de prendre tous les params
    ...
}
prends_le_butin( 'diamants', niveau => 'élevé', nombre => 9 );
# Noter: pas d'accolade dans cette API

# Perl 6, API modifiée pour spécifier les options valides
# Les deux-points avant les sigils signifient que l'on attend une paire
# dont la clef a le même nom que la variable
sub prends_le_butin ( $butin, :$niveau?, :$nombre = 1 ) {
    # Cette version va vérifier les arguments inattendus!
    ...
}
prends_le_butin( 'diamants', nouveau => 'élevé' );

# Génère une erreur en raison de la faute
# d'orthographe sur le nom du paramètre (nouveau)
```

## 2-6 - Opérateur de concaténation

L'opérateur point « . » étant maintenant un opérateur d'appel de méthode et de déréférencement, l'opérateur de concaténation n'est plus le point, mais le tilde « ~ ».

De même, l'opérateur d'affectation-concaténation n'est plus « .= », mais « ~= ».

## 2-7 - Opérateur ternaire « ? : » remplacé par « ?? !! »

Cet opérateur utilise désormais deux points d'interrogation au lieu d'un seul et deux caractères points d'exclamation au lieu du deux-points :

```
my $résultat = ( $note > 60 ) ? 'Réussi' : 'Loupé'; # Perl 5
my $résultat = ( $note > 60 ) ?? 'Réussi' !! 'Loupé'; # Perl 6
```

## 2-8 - Opérateur de répétition (de liste ou de chaîne)

En Perl 5, x est l'opérateur de répétition. En contexte scalaire ou si l'opérande à sa gauche n'est pas entre parenthèses, x renvoie une chaîne de caractères. En contexte de liste et si l'argument à gauche est entre parenthèses, il répète la liste. Il fallait simplifier ces règles alambiquées.

En Perl 6, x répète les chaînes de caractères, quel que soit le contexte.

En Perl 6, le nouvel opérateur xx répète des listes, quel que soit le contexte.

```
# Perl 5
print '-' x 80;           # Affiche une rangée de 80 tirets
@uns = 1 x 80;            # Un tableau contenant un seul élément,
                          # une chaîne de 80 fois le chiffre 1

@uns = (1) x 80;          # Un tableau de 80 chiffres 1
@uns = (5) x @uns;        # Met à 5 tous les éléments de @uns

# Perl 6
print '-' x 80;           # Inchangé
@uns = 1 x 80;            # Inchangé
@uns = 1 xx 80;           # Parenthèses plus nécessaires
@uns = 5 xx @uns;        # Parenthèses plus nécessaires
```

## 2-9 - Opérateurs de citation et assimilés

Il existe un opérateur de citation qui garantit des chaînes absolument littérales, sans aucune interpolation, c'est « Q » ou « #...# » (à supposer que vous sachiez comment obtenir « # » et « # » sur votre clavier). Même les échappements par antislash ne s'appliquent pas. Par exemple :

```
my $littéral = Q{Cela reste une accolade fermante → \};
# renvoie la chaîne : "Cela reste une accolade fermante → \"
```

L'opérateur « q » fait ce à quoi l'on s'attend en venant de Perl 5, à savoir pas d'interpolation à l'exception des séquences d'échappement par antislash. Par exemple :

```
my $var-échap = q{Ce n'est pas une accolade fermante → \} mais ceci → };
renvoie : "Ce n'est pas une accolade fermante → } mais c'est ceci →"
```

Comme en Perl 5, vous pouvez utiliser les apostrophes ou guillemets simples ('...') pour obtenir le même résultat.

L'opérateur « qq » autorise l'interpolation des variables (de même que les guillemets doubles « " »). Cependant, par défaut, seules les variables scalaires sont interpolées. Pour interpoler les autres types de variables

(tableaux, hachages, etc.), il faut les suffixer avec des crochets. Par exemple, `@a = <1 2 3>; say qq/@a[] exemple@exemple.com/;` donne : `1 2 3 exemple@exemple.com`. Toutefois, l'interpolation de hachages donne actuellement un résultat visuellement quelque peu inattendu :

```
my %hachage = 1 => 2, 3 => 4;
say "%a[]";      # affiche : 1      2 3      4.
```

comme si les clefs étaient séparées des valeurs par des tabulations et les valeurs des clefs suivantes par des espaces. Il est également possible d'interpoler dans des chaînes de caractères du code Perl 6 en le mettant entre accolades :

```
my $x = 5;
say "Le double de $x est {$x * 2} et le carré de $x est {$x ** 2}.";
# Imprime : Le double de 5 est 10 et le carré de $x est 25.
say qq/Le double de $x est {$x * 2} et le carré de $x est {$x ** 2}. /;
# idem
```

L'opérateur de citation de mots « `qw` » fonctionne comme en Perl 5 et peut aussi être rendu par `<...>`. Par exemple, `qw/ a b c/` est équivalent à `<a b c>`. Cet opérateur ne fait pas d'interpolation. Il existe une version, `qqw`, qui fait l'interpolation des variables. Par exemple :

```
my $a = 42;          # la variable $a n'a rien de spécial en Perl 6
say qqw/$a b c/;     # -> 42 b c
```

L'opérateur de citation du shell (appel de fonctions système) est `qx`, comme en Perl 5, mais il faut noter que les accents graves (ou *backticks*) ``...`` ne fonctionnent pas en Perl 6 et que les variables ne sont pas interpolées dans les chaînes `qx`. Pour interpoler les variables, il suffit de remplacer `qx` par `qqx`.

L'opérateur `qr` n'existe plus en Perl 6, mais les regex de Perl 6 offrent des possibilités bien plus puissantes, comme le mécanisme des regex nommées.

L'opérateur de translittération `tr///` n'est pas bien documenté à l'heure actuelle, mais il semble fonctionner comme en Perl 5, avec cependant cette différence que les intervalles de caractères s'écrivent « `a..z` », donc avec l'opérateur standard d'intervalle (au lieu de « `a-z` » en Perl 5). Il existe une version méthode de `tr///`, mieux documentée, qui s'appelle `.trans`. Cette méthode utilise des listes de paires :

```
$x.trans(['a'..'c'] => ['A'..'C'], ['d'..'q'] => ['D'..'Q'], ['r'..'z'] => ['R'..'Z']);
```

Les deux côtés de chaque paire sont interprétés comme le ferait `tr///`. Cela donne une grande flexibilité. Voir <http://...#Transliteration> pour une description complète.

L'opérateur `y///`, qui était en Perl 5 un simple synonyme de `tr///`, a été abandonné en Perl 6.

Les documents « ici même » (*heredocs*) sont spécifiés de façon un peu différente en Perl 6 : il faut utiliser le mot-clé `:to` immédiatement après l'opérateur de citation, par exemple, la séquence `q:to/FIN` commence un document se terminant par `FIN`. Les échappements et l'interpolation dépendent de l'opérateur de citation utilisé (citation littérale avec `Q`, échappement des antislashes avec `q` et interpolation des variables avec `qq`).

Ce paragraphe ne fait que résumer les principaux opérateurs de citation et assimilés ressemblant à ceux de Perl 5. Le chapitre [Citation et analyse lexicale](#) du tutoriel explique plus en détail la logique de fonctionnement du très puissant mécanisme sous-jacent de citation de chaînes de caractères, permettant à l'utilisateur de maîtriser très finement les caractéristiques de sa chaîne.

Pour encore plus de détails, voir <http://doc.perl6.org/language/quoting>.

## 2-10 - Opérateurs d'entrées-sorties (E/S ou IO)

Comme l'opérateur « diamant » (ou « carreau ») `<...>` est un opérateur de citation de mots (cf. § 3.9 ci-dessus), il n'est plus possible d'utiliser `<>` pour lire les lignes d'un fichier. Pour ce faire, il faut soit créer un objet IO à partir d'un nom de fichier, soit utiliser un descripteur de fichier (*filehandle*) ouvert et, dans les deux cas, utiliser la méthode `.lines` sur l'objet ou le descripteur :

```
# Objet IO :
my @tableau-de-lignes = "filename".IO.lines; # remarquer la concision
# Descripteur de fichier (ou FH) :
my $fh = open "filename", :r;                # :r -> FH en lecture
my @a = $fh.lines;
```

Pour lire un fichier ligne à ligne itérativement :

```
for 'gigantesque-csv'.IO.lines -> $line {
    # Traiter $line
}
```

À noter que la syntaxe utilisant `for` ne serait pas recommandée en Perl 5 pour un très gros fichier, car cela impliquerait de stocker l'ensemble du fichier dans un tableau temporaire et pourrait saturer la mémoire ; en Perl 6, cela ne pose aucun problème grâce à la *paresse* de l'opérateur : la ligne n'est lue que quand on en a besoin et il n'y a jamais plus d'une ligne en mémoire. Noter également l'utilisation ici d'un « bloc pointu » avec la syntaxe `->` (voir le chapitre sur les **Boucles** du tutoriel).

Pour avaler d'un seul coup (*to slurp*) tout un fichier dans un scalaire, il faut utiliser la méthode `.slurp`. Par exemple :

```
my $x = "filename".IO.slurp;

# Ou, façon procédurale (non objet) encore plus simple :
my $x = slurp "filename";

# Ou, avec un FH :
my $fh = open "filename", :r;
my $x = $fh.slurp;
```

Le descripteur de fichier en entrée magique `ARGV` est remplacé par `$*ARGFILES` et le tableau `@ARGV` des arguments de la ligne de commande par `@*ARGS`.

Pour écrire dans un fichier :

```
my $fh = open "filename", :w; # :w -> ouverture en écriture
$fh.say("données, trucs, bidules");
$fh.close;
```

Voir aussi § 9. Pour plus de détails sur les entrées-sorties, voir <http://.../language/io>.

## 2-11 - Deux points (« .. ») et trois points (« ... », création d'intervalles et opérateur flip-flop

En Perl 5, le « `..` » était deux opérateurs complètement différents, selon le contexte. En contexte de liste, c'est l'opérateur bien connu de création d'intervalles. Il y a beaucoup à dire sur les intervalles en Perl 6 (par exemple, il est possible d'inclure ou non les bornes de l'intervalle dans la liste créée, ainsi `1..^5` exclut la borne supérieure de l'intervalle et est donc équivalent à `1..4`), mais un intervalle de Perl 5 ne devrait pas *nécessiter* de traduction en Perl 6.

En contexte scalaire, `..` et `...` étaient en Perl 5 les opérateurs *flip-flop*, beaucoup moins utilisés. Ils ont été remplacés en Perl 6 par les opérateurs `ff` et `fff`.

Ces deux opérateurs sont évalués à faux jusqu'à ce que `$_` soit reconnu par le premier argument, puis à vrai jusqu'à ce que le second argument reconnaisse `$_`. L'exemple ci-dessous illustre la différence assez subtile entre les deux. À noter qu'il existe également des versions `^ff`, `^fff`, `ff^`, `fff^`, `ff^` et `fff^` qui renvoient faux pour la borne du premier ou du second argument où se trouve l'accent circonflexe.

```
# Dès que le premier argument reconnaît $_, ff renvoie vrai puis
# évalue le second. Ci-dessous, AB reconnaît B et ff renvoie faux
for <X Y Z AB C D B E F> {
    say $_ if /A/ ff /B/; # imprime seulement "AB"
}

# fff n'évalue pas le second argument pour la valeur de $_ qui
# a été reconnue par le premier argument
for <X Y Z AB C D B E F> {
    say $_ if /A/ fff /B/; # Imprime "AB", "C", "D" et "B"
}

# ^fff renvoie faux quand le premier argument reconnaît $_ puis
# renvoie vrai jusqu'à ce que le second argument reconnaisse $_
for <X Y Z AB C D B E F> {
    say $_ if /A/ ^fff /B/; # Imprime "AB", "C" et "D", mais pas "B"
}

# fff^ renvoie faux dès que le second argument reconnaît $_
for <X Y Z AB C D B E F> {
    say $_ if /A/ fff^ /B/; # Imprime "AB", "C" et "D", mais pas "B"
}
```

## 2-12 - Voir aussi

Voir aussi le chapitre **Opérateurs** de la seconde partie du tutoriel et celui sur la **création de nouveaux opérateurs** dans l'[annexe 2](#).

## 3 - Instructions composées

### 3-1 - Conditions

#### 3-1-1 - Conditions if elsif else unless

Très peu de changements sur les opérateurs conditionnels. Les parenthèses autour de la condition sont maintenant optionnelles, mais si elles sont utilisées, elles doivent être séparées du mot-clef par au moins un espace. Lier une expression conditionnelle à une variable a un peu changé :

```
if (my $x = faire_truc()) {...} # Perl5, marche aussi en Perl 6
if faire_truc() -> $x {...}    # Perl 6, idiomatique
```

On peut toujours utiliser la forme de Perl 5 avec l'opérateur `my` en Perl 6, mais la portée de la variable s'étendra alors au bloc extérieur.

La condition `unless` n'autorise qu'un seul bloc en Perl 6 ; elle n'autorise pas de clause `else` ou `elsif`.

Les opérateurs `if`, `elsif`, `else` et `unless` permettent aussi de créer des opérations conditionnelles, comme en Perl 5 :

```
next if $ligne =~ /\^s+#/;
say "Je l'emporte !" if gagné();
```

Comme en Perl 5, cette syntaxe de modification d'instruction ne permet pas de clause `else`. Cette syntaxe est également utilisable avec les opérateurs de boucle `for` et `while`.

## 3-1-2 - Branchements multiples given-when

La construction `given-when` est analogue à une chaîne d'instructions `if-elsif-else` ou à la construction `switch-case` d'un langage comme le C. Sa structure générale est la suivante :

```
given EXPR {  
    when EXPR { ... }  
    when EXPR { ... }  
    default { ... }  
}
```

Dans sa forme la plus simple, la construction est la suivante :

```
given $valeur {  
    when "une reconnaissance" {  
        faire_une_chose();  
    }  
    when "une autre reconnaissance" {  
        faire_autre_chose();  
    }  
    default {  
        faire_la_chose_par_defaut();  
    }  
}
```

C'est simple en ce sens qu'une valeur scalaire est reconnue dans les instructions `when`. Plus généralement, les reconnaissances sont en fait des reconnaissances intelligentes (*smart matches*) sur la valeur en entrée si bien que des recherches utilisant des entités plus complexes telles que des types ou des regex peuvent être réalisées au lieu d'une simple égalité de valeurs scalaires.

```
for 42, 43, "toto", 44 {  
    when Int { .say } # .say sans invoquant imprime la valeur de $_  
    default { say "Pas un Int" }  
}  
# -> 42 43 Pas un Int 44
```

## 3-2 - Boucles

### 3-2-1 - Boucles while et until

Peu de changements. Les parenthèses autour de la condition sont maintenant optionnelles, mais si elles sont utilisées, elles doivent être séparées du mot-clef par au moins un espace. Lier l'expression conditionnelle à une variable a un peu changé :

```
while (my $x = faire_truc()) {...} # Perl 5, marche aussi en Perl 6  
while faire_truc() -> $x {...} # Perl 6
```

On peut toujours utiliser la forme de Perl 5 avec l'opérateur `my` en Perl 6, mais la portée de la variable s'étendra au bloc extérieur.

La façon de lire sur un descripteur de fichier ligne par ligne a changé (*voir aussi* § 3.10).

En Perl 5, on lisait le fichier dans une boucle `while` avec l'opérateur diamant (« `<...>` »). Utiliser une boucle `for` au lieu d'un `while` était généralement une erreur parce que cela chargeait l'ensemble du fichier en mémoire et pouvait entraîner une saturation de la mémoire.

En Perl 6, l'instruction `for` est paresseuse, si bien que nous pouvons lire le fichier ligne par ligne dans une boucle `for` en utilisant la méthode `.lines` :



```
while (<$IN_FH>) { } # Perl 5
for $IN_FH.lines { } # Perl 6
```

### 3-2-2 - Boucles for et foreach

À noter que les mots-clefs `for` et `foreach` sont synonymes en Perl 5 et ne jouent aucun rôle dans la distinction entre la boucle `for` dite de style C (avec trois expressions) et la forme plus « perliste » avec itérateur de liste.

La forme de style C avec trois expressions utilise maintenant le mot-clef `loop` et ne subit pas d'autre changement. Les parenthèses restent nécessaires :

```
for ( my $i = 1; $i <= 10; $i++ ) { ... } # Perl 5
loop ( my $i = 1; $i <= 10; $i++ ) { ... } # Perl 6
```

La forme `for` ou `foreach` de Perl 5 avec itérateur de liste s'appelle désormais uniquement `for` (`foreach` n'est plus un mot-clef). Les parenthèses sont optionnelles.

La variable d'itération, si elle existe, a été déplacée après la liste et un opérateur flèche (bloc pointu) :

```
for 1..10 -> $x {
    say $x;      # imprime les nombres de 1 à 10
}
```

La variable d'itération est maintenant toujours lexicale. Utiliser `my` n'est ni nécessaire, ni même autorisé dans cette forme de type « bloc pointu ».

En Perl 5, la variable d'itération était un alias en lecture et écriture sur l'élément courant de la liste, ce qui pouvait être dangereux (risque de modification de la liste par effet de bord).

En Perl 6, cet alias est par défaut en lecture seule, pour des raisons de sécurité, sauf si vous remplacez la flèche `->` par une double flèche `<->` :

```
for my $cli (@clients) {...} # Perl 5; lecture et écriture
for @clients -> $cli {...}  # Perl 6; lecture seule
for @clients <-> $cli {...}  # Perl 6; lecture et écriture
```

Si la variable *topicale* par défaut `$_` est utilisée et doit être en lecture et en écriture, alors il faut utiliser `<->` et spécifier explicitement `$_` :

```
for (@clients) {...} # Perl 5; variable $_ par défaut
for @clients {...}  # Perl 6; $_ en lecture seule
for @clients <-> $_ {...} # Perl 6; $_ est en lecture et écriture
```

### 3-2-3 - Instructions de contrôle de flux

Les instructions `next`, `last` et `redo` sont inchangées.

Il n'y a plus de bloc `continue`. À la place, utilisez un bloc `NEXT` à l'intérieur du corps de la boucle.

```
# Perl 5
my $str = '';
for (1..5) {
    next if $_ % 2 == 1;
    $str .= $_;
}
continue {
    $str .= ':'
}
```

```
# Perl 6
my $str = '';
for 1..5 {
    next if $_ % 2 == 1;
    $str ~= $_;
    NEXT {
        $str ~= ':' ; # noter l'opérateur de concaténation
    }
}
```

### 3-3 - Voir aussi

Voir aussi les chapitres consacrés aux **Principales structures de contrôle** et aux **Comparaisons** du tutoriel.

## 4 - Fonctions internes

### 4-1 - Fonctions utilisant des blocs de code nus

Les fonctions internes qui acceptaient un bloc de code nu suivi, sans virgule, du reste des arguments doivent maintenant avoir une virgule après le bloc de code et avant la suite des arguments, par exemple `map`, `grep` ou `sort` :

```
my @results = grep { $_ eq "bars" } @foo;      # Perl 5
my @results = grep { $_ eq "bars" }, @foo;      # Perl 6

my @sorted = sort { $a <=> $b }, @unsorted;      # Perl 5
my @sorted = sort { $^a <=> $^b }, @unsorted;    # Perl 6
my @sorted = @unsorted.sort: { $^a <=> $^b };    # Perl 6, autre méthode
```

### 4-2 - Fonctions delete et exists

Les deux fonctions `delete` et `exists` sont transformées en des *adverbes* (arguments nommés pour des fonctions, ou parfois sur d'autres objets, préfixés par le signe deux-points) opérant sur les indices de tableaux ou clefs de hachages.

```
# delete

my $valeur-effacée = delete $hash{$clé}; # Perl 5
my $valeur-effacée = %hash{$clé}:delete; # Perl 6

my $valeur-effacée = delete $array[$i]; # Perl 5
my $valeur-effacée = @array[$i]:delete; # Perl 6

# exists

say "L'élément existe" if exists $hash{$key}; # Perl 5
say "L'élément existe" if %hash{$key}:exists; # Perl 6

say "L'élément existe" if exists $array[$i]; # Perl 5
say "L'élément existe" if @array[$i]:exists; # Perl 6
```

## 4-3 - Liste de fonctions

### 4-3-1 - Généralités

La plupart des fonctions peuvent aussi avoir une syntaxe de méthode. Ainsi, dans le cas de la première fonction ci-dessous (`abs`), les syntaxes suivantes sont possibles :

```
my $abs = abs($x); # ou sans parenthèses : my $abs = abs $x;
my $abs = $x.abs;  # syntaxe de méthode
```

Toutefois, dans certains cas, il faut parfois prendre quelques précautions avec la syntaxe de méthode pour éviter des problèmes de précédence. Ainsi, comme l'opérateur de méthode est prioritaire sur l'opérateur unaire `-`, on a :

```
my $val = -5.abs; # ERRONÉ : évalué comme -(5.abs) et retourne -5
my $val = (-5).abs; # Correct: retourne 5
```

Les appels de méthodes internes de Perl 6 sans invoquant travaillent presque toujours sur la variable par défaut `$_` :

```
my $val = .abs; # équivalent à : my $val = $_.abs;
```

## 4-3-2 - Liste alphabétique de fonctions Perl 5 et équivalences

La liste ci-dessous est longue, mais n'est pas tout à fait exhaustive. Nous passons en revue les fonctions courantes, mais laisserons de côté quelques fonctions spécialisées, très rarement utilisées par les programmeurs généralistes, relatives au système, aux réseaux, sockets, IPC System V, qui se trouvaient dans le *core* de Perl 5 pour des raisons historiques (à l'époque où Perl a été conçu comme un outil d'administration système Unix).

Un certain nombre de ces fonctions fortement spécialisées ont été reléguées non sans raison dans des modules spécialisés.

### Documentation officielle Perl 6 encore en évolution

*Cette section du présent document, plus encore que le reste de ces annexes, s'appuie fortement sur la documentation officielle Perl 6 en anglais, telle qu'elle existe au moment où nous écrivons (août-sept 2015). Or la documentation Perl 6 n'est à l'heure actuelle pas entièrement figée, elle évolue régulièrement et va encore évoluer.*

*Il est généralement peu probable que les syntaxes présentées changent significativement. Cependant, là où nous indiquons, à l'heure où nous écrivons, que la documentation est incomplète, voire inexistante, sur tel ou tel point de détail, il est fort possible que ce ne soit plus le cas dans quelques semaines quand vous lirez ces lignes. N'hésitez pas à consulter la [documentation officielle Perl 6](#) en cas de besoin.*

Fonction Perl 5	Fonction Perl 6	Remarques
<code>abs</code>	<code>abs</code>	Pas de différence. Syntaxe de méthode possible (voir § 5.3.1).
<code>atan2</code>	<code>atan2</code>	Syntaxe de méthode possible : <code>my \$tg = 1.atan2;</code> <code># pi/4.</code>
<code>binmode</code>	N/A	Remplacé notamment par le mode <code>:bin</code> lors de l'ouverture du descripteur de fichier (ou d'un socket, d'un pipe, etc.).
<code>bless</code>	<code>bless</code>	Le constructeur de classes <code>new</code> supplante <code>bless</code> dans la plupart des cas. Reste utilisable pour des

		constructeurs de bas niveau.
break	N/A	Pour sortir d'un bloc given, voir proceed et succeed.
caller	callframe	La fonction callframe renvoie un objet de type CallFrame. Pour en extraire des informations utiles, utiliser la méthode callframe().annotations. Consulter la documentation.
chdir	chdir	Semble fonctionner comme en Perl 5.
chmod	chmod	Fonctionne comme en Perl 5, sauf que les nombres octaux sont représentés différemment (0o755 au lieu de 0755).
chomp	chomp	Comportement modifié : laisse l'argument d'origine inchangé et renvoie la valeur modifiée (sans le retour à la ligne). Syntaxe de méthode possible.
chop	chop	Comportement modifié : laisse l'argument d'origine inchangé et renvoie la valeur modifiée (sans le dernier caractère de la chaîne). Syntaxe de méthode possible.
chown	N/A	Fonction reléguée à un éventuel module.
chr	chr	Analogue à Perl 5 : coercion de l'argument en un entier, et utilisation de cet entier comme point de code Unicode pour renvoyer le caractère voulu. Syntaxe de méthode possible : chr(65); # "A" est équivalent à 65.chr; # "A".
close	close	Ferme un descripteur de fichier comme en Perl 5. Syntaxe de méthode possible. close \$fh et \$fh.close font la même chose.
closedir	N/A	N'existe actuellement pas. Pourrait être implémentée sous la forme d'une méthode de la classe IO::DIR.
continue	N/A	Au lieu d'un bloc continue, utiliser un bloc NEXT (voir § 4.2.3). Voir aussi

		les fonctions <code>proceed</code> et <code>succeed</code> .
<code>cos</code>	<code>cos</code>	Syntaxe de méthode possible : <code>cos(pi/3) # 0.5</code> et <code>(pi/3).cos # 0.5</code> sont équivalents.
<code>defined</code>	<code>defined</code>	Fait sans doute ce que vous attendez, mais, techniquement, renvoie <code>False</code> sur l'objet-type non initialisé et <code>True</code> sinon. Syntaxe de méthode possible.
<code>delete</code>	<code>(:delete)</code>	N'existe plus sous forme de fonction, mais sous la forme de l'adverbe <code>:delete</code> : <code>my \$deleted_value = @array[\$i]:delete;</code>
<code>die</code>	<code>die</code>	Fonctionne comme en Perl 5, mais le mécanisme d'exception de Perl 6 donne bien plus de flexibilité et d'expressivité.
<code>do</code>	<code>do</code>	Comme en Perl 5. Un espace est nécessaire entre <code>do</code> et l'accolade ouvrante du bloc qui suit.
<code>each</code>	<code>(.kv)</code>	L'équivalent le plus proche est l'appel de méthode <code>%hash.kv</code> . Par exemple : <code>for %hash.kv -&gt; \$k, \$v { utiliser \$k et \$v }.</code>
<code>eof</code>	<code>(.eof)</code>	N'existe pas comme fonction, mais seulement comme méthode.
<code>eval</code>	<code>EVAL</code>	À noter qu' <code>EVAL</code> ne fait pas de gestion d'exception.
<code>exec</code>	N/A	Les fonctions <code>shell</code> et <code>run</code> remplacent la fonction <code>system</code> , mais il n'y a pas d'équivalent pour <code>exec</code> , sans doute très rarement utilisée en fait. Facile à émuler, par exemple avec : <code>shell(\$command);exit();</code>
<code>exists</code>	<code>(:exists)</code>	En Perl , ce n'est pas une fonction, mais un adverbe (voir § <a href="#">5.2</a> ).
<code>exit</code>	<code>exit</code>	Peu documenté, mais semble faire la même chose qu'en Perl 5.
<code>exp</code>	<code>exp</code>	Syntaxe de méthode possible, mais l'ordre des arguments peut être

		trompeur : say 2.exp(3); # 9, soit 3**2.
<u>FILE</u>	\$?FILE	RAS.
fork	N/A	N'existe pas comme fonction interne, mais disponible via l'interface NativeCall. Par exemple : use NativeCall; sub fork returns int32 is native { * }; say fork;.
Formats	N/A	Les formats (adaptés à des technologies d'impression aujourd'hui dépassées, mais parfois encore occasionnellement utiles pour des rapports) ont été abandonnés en tant que <i>core</i> Perl 6. Ils pourront faire l'objet de modules spécifiques.
getc	getc	Syntaxe de méthode possible : \$filehandle.getc.
glob	N/A	Utilisé dans un exemple dans la synopsis S32. Pas implémenté pour l'instant, semble-t-il.
gmtime	N/A	N'existe plus comme fonction interne, mais l'équivalent se trouve dans la classe DateTime. Par exemple : my \$gmtime = DateTime.now.utc;.
goto	goto	Décrit dans la synopsis S04, mais pas complètement documenté.
grep	grep	Comme en Perl 5, mais la forme avec bloc nécessite maintenant une virgule après le bloc (§ 5.1). Syntaxe de méthode possible : @foo = @bar.grep(/^f/);.
hex	(:16)	La fonction hex est remplacée par l'adverbe :16. Par exemple, :16("aF") renvoie 175.
index	index	Syntaxe de méthode possible : "Bonjour".index("jou") # 3.
int	truncate	La fonction int est remplacée par la fonction truncate, également utilisable avec une syntaxe de méthode. La méthode .int fait la même chose

		(mais n'existe pas comme fonction).
join	join	Syntaxe de méthode possible : <code>my \$c = @x.join(",");</code> .
keys	keys	Syntaxe de méthode possible : <code>my @a = %hash.keys;</code> .
last	last	Même chose en Perl 5 et Perl 6.
lc	lc	Syntaxe de méthode possible : <code>my \$lc = "BOUM".lc;</code> .
lcfirst	N/A	N'existe pas en Perl 6. À mettre dans un module, peut-être.
length	chars	<code>length</code> est remplacée par <code>chars</code> , habituellement utilisée comme une méthode, mais aussi utilisable avec une syntaxe de fonction.
<code>__LINE__</code>	<code> \$?LINE</code>	RAS.
link	link	Même chose, mais l'ordre des arguments a changé : <code>link(\$original, \$fichier-lié)</code> .
listen	listen	Pas clairement documenté, mais <code>listen</code> devrait être une méthode sur un objet du genre <code>IO::Socket</code> .
local	temp	Le nouveau nom <code>temp</code> exprime plus clairement ce que cela fait.
localtime	N/A	Les fonctionnalités de <code>localtime</code> sont fournies par la classe <code>DateTime</code> .
lock	(lock)	En Perl 6, <code>lock</code> est une méthode de la classe <code>Lock</code> .
log	log	Syntaxe de méthode possible : <code>log(2)</code> est équivalent à <code>2.log</code> .
lstat	N/A	La fonctionnalité devrait être implémentée dans l'une des classes <code>IO</code> , reste à savoir laquelle.
m//	m//	La syntaxe des regex a bien évolué, mais l'opérateur de reconnaissance <code>m//</code> existe toujours. La principale différence est que <code>=~</code> et <code>!~</code> sont remplacés respectivement par <code>~~</code> et <code>!</code> . Voir § 3.2 et § 6.1.
map	map	Comme en Perl 5, mais la forme avec bloc

		nécessite maintenant une virgule après le bloc (voir § 5.1). Syntaxe de méthode possible : <code>my @new = @old.map: { \$_ * 2 }</code> .
<code>mkdir</code>	<code>mkdir</code>	Même chose en Perl 5 et Perl 6.
<code>my</code>	<code>my</code>	Même chose en Perl 5 et Perl 6.
<code>next</code>	<code>next</code>	Même chose en Perl 5 et Perl 6.
<code>no</code>	<code>no</code>	Utilisable en Perl 6 pour les <i>pragmas</i> , mais pas pour les autres modules. Ne semble pas devoir fonctionner avec la forme <code>no VERSION</code> .
<code>oct</code>	<code>(:8)</code>	La fonction <code>oct</code> est remplacée par l'adverbe <code>:8</code> . Par exemple, <code>:8("100")</code> renvoie 64.
<code>open</code>	<code>open</code>	Le changement principal est la façon de préciser le mode d'ouverture : <code>:r</code> , <code>:w</code> , <code>:rw</code> et <code>:a</code> , pour ouvrir des fichiers, respectivement, en modes lecture, écriture, lecture-écriture et ajout en queue de fichier ( <i>append</i> ). Par exemple : <code>open("file", :r);</code> . Il existe des options complémentaires pour gérer les fins de ligne et l'encodage.
<code>opendir</code>	N/A	Pas de fonction interne Perl. Il faut utiliser la classe <code>IO::Path</code> : <code>my \$dir = IO::Path.new("directory");</code> ou, plus simplement, <code>my \$dir = "directory".IO;</code> .
<code>ord</code>	<code>ord</code>	Même chose en Perl 5 et Perl 6. Syntaxe de méthode possible : <code>"Salut".ord; # 83</code> .
<code>our</code>	<code>our</code>	Même chose en Perl 5 et Perl 6.
<code>pack</code>	<code>pack</code>	Même chose en Perl 5 et Perl 6, mais les options de modèles ( <i>templates</i> ) sont actuellement moins nombreuses qu'en Perl 5. La liste des options actuellement documentées se trouve à l'adresse




		<a href="http://doc.perl6.org/routine/unpack">http://doc.perl6.org/routine/unpack</a> .
package	package	<p>La synopsis S10 indique que l'on peut utiliser des packages en Perl 6. Ainsi, <code>package Foo { ... }</code> déclare ce qui se trouve dans le bloc entre les accolades comme un paquetage. Il existe un cas particulier dans lequel une déclaration de la forme <code>package Toto;</code> est la première instruction du fichier, ce qui indique que le reste du fichier est du code Perl 5, mais l'utilité de ce mécanisme est peu claire (peut-être utiliser un module Perl 5 en Perl 6). De toute façon, comme les modules et les classes sont déclarés avec des mots-clé distincts, il est peu probable de devoir utiliser <code>package</code> directement en Perl 6.</p>
<code>__PACKAGE__</code>	<code> \$?PACKAGE</code>	RAS.
pipe	N/A	<p>Sans doute remplacé par quelque chose dans la classe <code>IO::Pipe</code>, mais ce n'est pas clairement documenté.</p>
pop	pop	<p>Même chose en Perl 5 et Perl 6. Syntaxe de méthode possible : <code>my \$x = @a.pop;</code> est équivalent à <code>my \$x = pop @a;</code>.</p>
print	print	<p>En Perl 6, <code>print</code> est une fonction qui écrit par défaut sur la sortie standard. Pour écrire sur un descripteur de fichier (<i>file handle</i>), il faut suffixer celui-ci du caractère deux-points : <code>print \$fh: "Bonjour!"</code>. Le caractère deux-points sert de « marqueur d'invoquant » dans la syntaxe objet indirecte (voir <a href="http://design.perl6.org/S03.html#line_4019">http://design.perl6.org/S03.html#line_4019</a>). Sinon, il est possible d'utiliser une invocation de</p>

<http://>

		méthode : <code>\$fh.print("Bonjour!");</code>
<code>printf</code>	<code>printf</code>	Même chose en Perl 5 et Perl 6. Voir <code>sprintf</code> pour les formats disponibles.
<code>push</code>	<code>push</code>	Même chose en Perl 5 et Perl 6. Syntaxe de méthode possible : <code>@a.push("foo");</code>
<code>rand</code>	<code>rand</code>	Même chose en Perl 5 et Perl 6, mais il n'est plus possible de passer un argument à sa forme fonctionnelle, alors que c'est possible avec la forme appel de méthode. Pour l'équivalent du code Perl 5 <code>rand(100)</code> , il faut donc écrire <code>100.rand</code> . Il est également possible d'obtenir un nombre entier aléatoire avec une instruction de ce type : <code>(^100).pick</code> .
<code>read</code>	<code>(read)</code>	<code>read</code> est une méthode des classes <code>IO::Handle</code> et <code>IO::Socket</code> en Perl 6.
<code>readdir</code>	<code>(dir)</code>	Pas de fonction interne, mais la classe <code>IO::Path</code> fournit la routine <code>dir</code> assurant essentiellement la même fonction.
<code>readline</code>	<code>(.lines)</code>	Pas de fonction interne, mais il est possible d'utiliser la méthode <code>.lines</code> aux mêmes fins.
<code>redo</code>	<code>redo</code>	Même chose en Perl 5 et Perl 6.
<code>ref</code>	N/A	<code>ref</code> n'existe plus en Perl 6, mais pour obtenir le nom d'un type, on peut utiliser <code>\$var.WHAT.perl</code> .
<code>rename</code>	<code>rename</code>	Même chose en Perl 5 et Perl 6.
<code>require</code>	<code>require</code>	Même chose en Perl 5 et Perl 6 pour les modules, mais on ne sait si cela fonctionnera pour les numéros de versions.
<code>return</code>	<code>return</code>	Existe en Perl 6, mais pas clairement documenté.
<code>reverse</code>	<code>reverse</code> / <code>.flip</code>	En Perl 6, <code>reverse</code> n'inverse que les éléments d'une liste. Pour inverser les


		caractères d'une chaîne, utiliser la méthode <code>.flip</code> .
<code>rindex</code>	<code>rindex</code>	Même chose en Perl 5 et Perl 6. Syntaxe de méthode possible : <code>my \$x = "Abracadabrantisque"; say \$x.rindex("a"); say \$x.rindex("a", 5); # 10 5.</code>
<code>rmdir</code>	<code>rmdir</code>	Même chose en Perl 5 et Perl 6. Syntaxe de méthode possible : <code>rmdir "Fic";</code> et <code>"Fic".IO.rmdir;</code> sont équivalents.
<code>s///</code>	<code>s///</code>	La syntaxe des regex a bien évolué, mais l'opérateur de substitution <code>s///</code> existe toujours. La principale différence est que <code>=~</code> et <code>!~</code> sont remplacés respectivement par <code>~~</code> et <code>!~~</code> . Voir § 3.2 et § 6.1.
<code>say</code>	<code>say</code>	En Perl 6, <code>say</code> est une fonction qui écrit par défaut sur la sortie standard. Pour écrire sur un descripteur de fichier ( <i>file handle</i> ), il faut suffixer celui-ci du caractère deux-points : <code>say \$fh: "Bonjour!"</code> . Le caractère deux-points sert de « marqueur d'invoquant » dans la syntaxe objet indirecte (voir <a href="http://design.perl6.org/S03.html#line_4019">http://design.perl6.org/S03.html#line_4019</a> ). Sinon, il est possible d'utiliser une invocation de méthode : <code>\$fh.say("Bonjour!")</code> .
<code>scalar</code>	N/A	Apparemment disparu.
<code>seek</code>	( <code>seek</code> )	Pas de documentation concrète à ce jour, mais listé dans la classe <code>IO::HANDLE</code> .
<code>select</code>	N/A	Disparu (mais pas vraiment regretté).
<code>sem*</code>	N/A	Les fonctions <code>semctl</code> , <code>semget</code> et <code>semop</code> ne font plus partie des fonctions internes. Elles pourraient

		réapparaître dans un module quelque part.
send	(send)	Ce n'est plus une fonction interne, mais elle existe dans la classe IO::Socket.
set*	N/A	Les fonctions setpgrp et setpriority ne sont plus des fonctions internes, mais elles devraient réapparaître dans un module POSIX. La fonction setsockopt n'est pas documentée, mais elle se cache sans doute dans une classe IO.
shift	shift	Même chose en Perl 5 et Perl 6. Syntaxe de méthode possible : shift@a et @a.shift sont équivalents.
sin	sin	Même chose en Perl 5 et Perl 6. Syntaxe de méthode possible : sin 2 et 2.sin sont équivalents.
sleep	sleep	Fonctionne comme en Perl 5, mais a été dépréciée et pourrait disparaître.
sort	sort	La fonction sort existe en Perl 6, mais son utilisation diffère quelque peu. \$a et \$b ne sont plus des variables spéciales (voir \$a et \$b dans le § 10.2) et les fonctions de comparaisons ne retournent plus -1, 0, 1, mais des objets Order::Increase, Order::Same, ou Order::Decrease. Voir <a href="http://doc.perl6.org/routine/sort">http://doc.perl6.org/routine/sort</a> pour les détails. Syntaxe de méthode possible : sort(@a) est équivalent à @a.sort. Si la fonction de comparaison est spécifiée dans un bloc de code anonyme, celui-ci doit être suivi d'une virgule avant la suite des arguments.
splice	splice	Même chose en Perl 5 et Perl 6. Syntaxe de méthode possible : splice(@tabl, 2,3, <M N O P>); est équivalent à

		@tabl.splice(2, 3, <M N O P>);.
split	split	<p>Fonctionne essentiellement comme en Perl 5 avec quelques différences de détail. Pour obtenir le comportement spécial de la chaîne vide, il faut utiliser vraiment une chaîne vide (cela ne marche pas avec un motif vide //). split divisera la chaîne sur une regex si l'on utilise une regex et sur une chaîne littérale si on utilise une chaîne littérale, sans mélange des genres. Les fragments vides ne sont plus supprimés de la liste (voir aussi la fonction  <b>comb</b> qui peut filtrer la liste produite). Syntaxe de méthode possible : "a;b;c".split(";").</p>
sprintf	sprintf	<p>Fonctionne comme en Perl 5. Voici la liste des formats utilisables après le caractère % littéral :</p> <ul style="list-style-type: none"> <li>• <b>c</b> : un caractère avec le point de code donné ;</li> <li>• <b>d</b> : un entier signé, en décimal ;</li> <li>• <b>u</b> : un entier non signé, en décimal ;</li> <li>• <b>o</b> : un entier non signé, en octal ;</li> <li>• <b>x</b> : un entier non signé, en hexadécimal ;</li> <li>• <b>e</b> : un nombre à virgule flottante, notation scientifique ;</li> <li>• <b>f</b> : un nombre à virgule flottante, notation décimale ;</li> <li>• <b>g</b> : un nombre à virgule flottante en notation %e ou %f ;</li> <li>• <b>X</b> : comme x, mais en utilisant des lettres capitales ;</li> <li>• <b>E</b> : comme e, mais en utilisant la lettre capitale E ;</li> </ul>

		<ul style="list-style-type: none"> <li>• <b>G</b> : comme <b>g</b>, mais avec la capitale <b>E</b> (le cas échéant).  <i>Compatibilité</i> : <b>%i</b> est synonyme de <b>%d</b>, <b>%D</b> est synonyme de <b>%ld</b>, <b>%U</b> est synonyme de <b>%lu</b>, <b>%O</b> est synonyme de <b>%lo</b>, <b>%F</b> est synonyme de <b>%f</b>.            Les options <b>n</b> et <b>p</b> produisent actuellement des exceptions à l'exécution.            Les <i>modificateurs</i> suivants servent à l'interprétation des entiers :           <ul style="list-style-type: none"> <li>• <b>h</b> : comme un « short » natif (typiquement int16) ;</li> <li>• <b>l</b> : comme un « long » natif (int32 ou int64) ;</li> <li>• <b>ll</b> : comme un « long long » natif (int64) ;</li> <li>• <b>L</b> : comme « long long » natif (uint64) ;</li> <li>• <b>q</b> : comme un « quads » natif (int64 ou plus grand),</li> </ul>           mais ils ne font pour l'instant rien (pour la plupart), car leur sémantique n'est pas complètement arrêtée.         </li> </ul>
<code>sqrt</code>	<code>sqrt</code>	Même chose en Perl 5 et Perl 6. Syntaxe de méthode possible : <code>sqrt(4)</code> et <code>4.sqrt</code> sont équivalents.
<code>srand</code>	<code>srand</code>	Même chose en Perl 5 et Perl 6.
<code>stat</code>	N/A	Sera sans doute implémenté dans une classe IO, mais où exactement n'est pas clair pour l'instant.
<code>state</code>	<code>state</code>	Disponible en Perl 6, mais pas clairement documenté pour l'instant.
<code>study</code>	N/A	N'existe plus.
<code>sub</code>	<code>sub</code>	Personne ne sera surpris de savoir qu'il y a encore des fonctions ( <i>subroutines</i> )

		<p>en Perl 6 ! Il y a des nouveautés, notamment la possibilité d'utiliser des signatures pour spécifier les arguments. Toutefois, en l'absence de signature (et seulement dans ce cas), le tableau <code>@_</code> contient les arguments passés à la fonction. Il n'y a donc en principe pas de changement à faire dans le cas d'un portage d'un programme Perl 5 et Perl 6 (mais il n'est pas déraisonnable d'envisager, voire de recommander, l'utilisation des signatures). Pour les détails supplémentaires sur les fonctions, voir <a href="#">Fonctions et signatures</a> dans le tutoriel et <a href="#">Fonctions</a>.</p>
<code>__SUB__</code>	<code>&amp;?ROUTINE</code>	RAS.
<code>substr</code>	<code>substr</code>	Même chose en Perl 5 et Perl 6. Syntaxe de méthode possible : <code>substr("¡hola!", 2, 3)</code> et <code>"¡hola!".substr(2, 3)</code> renvoient "ola".
<code>symlink</code>	<code>(symlink)</code>	Implémenté dans <code>IO::Path</code> en Perl 6. La différence entre Perl 5 et Perl 6 est que l'ordre des arguments a changé : c'est maintenant <code>symlink(\$original, \$fichier-lié)</code> .
<code>sys*</code>	N/A	Les fonctions <code>sysopen</code> , <code>sysread</code> , <code>syswrite</code> et <code>sysseek</code> ne sont plus des fonctions internes, mais sont sans doute implémentées quelque part dans les classes IO.
<code>system</code>	N/A	À remplacer sans doute par les fonctions <a href="#">shell</a> ou <a href="#">run</a> , selon que l'on désire ou non passer par l'interprétation du shell.
<code>tell</code>	<code>tell</code>	Implémenté dans <code>IO::Handle</code> , mais non documenté à ce jour.
<code>tie</code>	N/A	La synopsis <code>S29</code> indique que la fonctionnalité associée à <code>tie</code> est remplacée par les types

		de containers, mais ce que cela signifie en termes pratiques reste obscur.
time	time	La documentation dit : « Renvoie un entier représentant l'instant présent », mais ne précise pas <i>comment</i> cet entier représente l'instant. Actuellement, la fonction paraît renvoyer le nombre de secondes écoulées depuis l'« Époque » <sup>(1<sup>er</sup> janvier 1970)</sup> , comme en Perl 5.
tr///	tr///	Fonctionne comme en Perl 5, sauf que les intervalles sont spécifiés avec l'opérateur d'intervalle de Perl 6 (a..z) et non avec a-z comme en Perl 5. Voir § 3.9 pour plus de détails.
truncate	N/A	Sans doute quelque part dans IO::Handle, mais pas documenté pour l'instant.
uc	uc	Fonctionne comme une fonction et comme une méthode. uc("ha") et "ha".uc renvoient tous deux "HA".
ucfirst	N/A	Supprimé des fonctions internes de Perl 6. La fonction  tc (title case, majuscule à chaque mot) fera bien souvent ce dont vous avez besoin.
umask	(umark)	C'est une méthode de la classe IO. IO.umask renvoie le masque de création de fichiers par l'utilisateur (umask ou « u-masque »).
undef	N/A	N'existe pas en Perl 6. Il n'est pas possible d'« undefinir » une fonction, et la valeur équivalente la plus proche pour une variable est sans doute Nil, mais il est peu probable d'en avoir besoin.
unlink	unlink	Même chose en Perl 5 et Perl 6. Syntaxe de méthode possible : "fichier-à-effacer".IO.unlink;
unpack	unpack	Même chose en Perl 5 et Perl 6, mais les options



		de modèles ( <i>templates</i> ) sont actuellement moins nombreuses qu'en Perl 5. La liste des options actuellement documentées se trouve à l'adresse <a href="http://doc.perl6.org/routine/unpack">http://doc.perl6.org/routine/unpack</a> .
unshift	unshift	Même chose en Perl 5 et Perl 6. Syntaxe de méthode possible : <code>unshift(@a, "blah");</code> est équivalent à <code>@a.unshift("blah");</code> .
untie	N/A	Même remarque que pour tie ci-dessus.
use	use	Disponible en Perl 6, mais curieusement non encore documenté. Semble fonctionner de manière analogue, voire identique, à Perl 5.
values	values	Même chose en Perl 5 et Perl 6. Syntaxe de méthode possible : <code>values %hash</code> est équivalent à <code>%hash.values</code> .
wantarray	N/A	N'existe plus en Perl 6.
warn	warn	La fonction <code>warn</code> déclenche une exception. Pour afficher un message sur la sortie d'erreur ( <code>\$*ERR</code> ), utiliser la fonction <code>note</code> . Pour plus de détails sur les exceptions, voir le chapitre <b>Exceptions</b> du tutoriel.
write	N/A	Les <i>formats</i> (voir ce mot ci-dessus) ont été supprimés du <i>core</i> de Perl 6, la fonction <code>write</code> n'a donc plus lieu d'être.
y///	tr///	En Perl 5, <code>y///</code> était un pur synonyme de <code>tr///</code> , conservé pour des raisons historiques (utilitaire <code>sed</code> ). <code>y///</code> n'existe plus en Perl 6 et est remplacé par <code>tr///</code> .

## 5 - Expressions régulières (regex)

### 5-1 - Les opérateurs `=~` et `!~` deviennent `~~` et `!~~`

En Perl 5, les reconnaissances et les substitutions sur une variable s'effectuent avec l'opérateur de liaison de regex `=~`.

En Perl 6, c'est l'opérateur `~~` de reconnaissance intelligente (*smart match*) qui est utilisé.

```
next if $ligne =~ /statique/ ; # Perl 5
next if $ligne ~~ /statique/ ; # Perl 6

next if $ligne !~ /dynamique/ ; # Perl 5
next if $ligne !~~ /dynamique/ ; # Perl 6

$ligne =~ s/abc/123/;          # Perl 5
$ligne ~~ s/abc/123/;          # Perl 6
```

On peut également utiliser les nouvelles méthodes `.match` and `.subst`. Mais la méthode `.subst` ne modifie pas la chaîne d'origine (elle ne fait que renvoyer la chaîne modifiée si la substitution a réussi).

## 5-2 - Déplacement des modificateurs

Les modificateurs ou *adverbes* sont déplacés de la fin vers le début de la regex. Ceci peut vous obliger à ajouter le `m` optionnel dans un motif de type `/abc/`.

```
next if $ligne =~ /statique/i ; # Perl 5
next if $ligne ~~ m:i/statique/ ; # Perl 6
```

## 5-3 - Espaces dans les motifs

Par défaut, les espaces positionnés dans les motifs ne jouent pas de rôle dans les reconnaissances (autrement dit, ils sont ignorés).

```
say "Vrai" if "abc" ~~ /a b c/; # imprime Vrai (mais émet un warning)
```

Comme en Perl 5, on peut utiliser `\s` pour reconnaître un espace de type quelconque (espace proprement dit, tabulation, saut de ligne, etc.), et `\s+` pour plusieurs. La séquence `' '` permet de reconnaître un seul espace blanc. De même, `\t`, `\h` et `\v` reconnaissent respectivement des tabulations, des espaces horizontaux et des espaces verticaux.

Le modificateur `:s` (ou `:sigspace`) permet de tenir compte littéralement des espaces :

```
say "Vrai" if "abc" ~~ m:s/ab c/; # n'imprime rien, non reconnu
say "Vrai" if "ab c" ~~ m:s/ab c/; # Vrai
```

On peut obtenir le même résultat avec la syntaxe simplifiée `ms/ab c/` pour définir le motif.

Des détails complémentaires sont disponibles dans le paragraphe **Reconnaissance** du tutoriel.

## 5-4 - Utiliser le modificateur `:P5` ou l'adverbe `:Perl5`

Si vous avez une expression régulière Perl 5 complexe à utiliser dans un programme Perl 6, vous pouvez la garder telle quelle en disant à Perl 6 d'utiliser la syntaxe et la sémantique des expressions régulières Perl 5 au moyen du modificateur `:P5` ou de l'adverbe `:Perl5`.

```
next if $ligne =~ m/[aeiou]/ ; # Perl 5
next if $ligne ~~ m:P5/[aeiou]/ ; # Perl 6, avec le modificateur P5
next if $ligne ~~ m:Perl5/[aeiou]/; # Perl 6, avec l'adverbe Perl5
next if $ligne ~~ m/ <[aeiou]> / ; # Perl 6, nouvelle syntaxe des
# classes de caractères
```

## 5-5 - Les reconnaissances spéciales utilisent souvent une syntaxe avec des chevrons <>

Nous ne pouvons les énumérer toutes, mais, souvent, au lieu d'être entourées de parenthèses (...), les reconnaissances spéciales seront entourées de chevrons <...>.

Ainsi, pour les classes de caractères :

Perl 5	Perl 6
[abc]	<[abc]>
[^abc]	<-[abc]>
[a-zA-Z]	<[a-zA-Z]>
[[:upper:]]	<:Upper>
[abc][:upper:]]	<[abc]+:Upper:>

De même pour les assertions avant ou arrière :

Perl 5	Perl 6
(?=[abc])	<?[abc]>
(?=ar?bitrary* pattern)	<!before ar?bitrary* pattern>
(?!=[abc])	<![abc]>
(?!=ar?bitrary* pattern)	<!before ar?bitrary* pattern>
(?<=ar?bitrary* pattern)	<after ar?bitrary* pattern>
(?<!ar?bitrary* pattern)	<!after ar?bitrary* pattern>
/foo\Kbar/	/foo <( bar )>
(?(?{condition})yes-pattern no-pattern)	[ <?{condition}> yes-pattern   no-pattern ]

## 5-6 - Reconnaissance du motif le plus long dans les alternatives

Dans les alternatives en Perl 5, c'est le premier motif reconnu qui gagne.

Avec l'opérateur d'alternative |, Perl 6 examine tous les termes de l'alternative (sans doute en parallèle), et en cas de reconnaissances multiples, c'est le motif le plus long qui est retenu.

Pour obtenir le comportement de Perl 5 (premier motif reconnu), il suffit d'utiliser l'opérateur ||.

## 5-7 - Voir aussi

Voir aussi les chapitres **Regex (ou règles)** et **Les regex, le retour** du tutoriel, le **chapitre sur les regex et les grammaires** de l'**annexe 2** et l'article beaucoup plus complet  **Les regex et les grammaires de Perl 6 : une expressivité sans précédent.**

## 6 - Pragmas

### 6-1 - Modules strict et warnings

Le mode `strict` et les `warnings` sont activés par défaut.

## 6-2 - Module autodie

Les fonctions qui généraient une exception si le module `autodie` était utilisé déclenchent maintenant par défaut une exception, sauf si le programme teste explicitement leur valeur de retour.

```
# Perl 5
open my $i_fh, '<', $fic_in; # échoue silencieusement en cas d'erreur
use autodie;
open my $o_fh, '>', $fic_out; # exception en cas d'erreur

# Perl 6
my $i_fh = open $input_path, :r; # exception en cas d'erreur
my $o_fh = open $output_path, :w; # exception en cas d'erreur
```

## 6-3 - Modules base et parent

Les modules `base` et `parent` sont remplacés en Perl 6 par le mot-clef `is` dans la déclaration de la classe.

```
# Perl 5
package Chien;
use base qw(Animal);

# Perl 6
class Chien is Animal;
```

## 6-4 - Modules bigint, bignum et bigrat

Ces modules sont sans objet en Perl 6.

Les entiers (type `Int`) ont maintenant une précision arbitraire, de même que le numérateur des rationnels (type `Rat`). Le dénominateur des types `Rat` est limité à  $2^{64}$  ( $2^{64}$ ), après quoi il passe en type `Num` pour ne pas pénaliser les performances. Si vous avez besoin d'un `Rat` avec un dénominateur de précision arbitraire, alors il faut utiliser le type `FatRat`.

## 6-5 - Module constant

En Perl 6, `constant` déclare une variable, au même titre que `my`, sauf que la valeur de la variable est verrouillée à la valeur de son expression d'initialisation (évaluée au moment de la compilation).

Donc, il faut remplacer le `=>` de Perl 5 par un opérateur `=` d'affectation simple et, en principe, ajouter un sigil.

```
use constant DEBUG => 0; # Perl 5
constant $DEBUG = 0;     # Perl 6

use constant pi => 4 * atan2(1, 1); # Perl 5
# pi, e et i sont des constantes prédéfinies en Perl 6

# NB: le sigil $ n'est pas indispensable, un "mot nu" fonctionne aussi:
constant nombre_d_or = (5**(1/2)+1)/2; # (1.61803398874989)
say nombre_d_or ** 2;                  # 2.61803398874989
say 1 / nombre_d_or;                   # 0.61803398874989
# il reste préférable pour permettre l'interpolation dans une chaîne
say "nombre_d_or";                     # nombre_d_or
constant $nombre_d_or = (5**(1/2)+1)/2;
say "$nombre_d_or";                    # 1.61803398874989
```

## 6-6 - Module mro

Sans objet : les appels de méthodes utilisent toujours l' **ordre de résolution de méthode C3**.

## 6-7 - Module utf8

Sans objet : en Perl 6, le code source est encodé par défaut en UTF-8.

## 6-8 - Module vars

L'utilisation de ce module pour déclarer des variables globales est **maintenant déconseillée** en Perl 5. Mieux vaut utiliser une déclaration avec l'opérateur `our`.

Modifiez votre code Perl 5 pour éviter l'utilisation de `use vars` avant de traduire votre code en Perl 6.

## 7 - Options de la ligne de commande

Les options suivantes de la ligne de commande sont inchangées : `-c`, `-e`, `-h`, `-I`, `-n`, `-p`, `-S`, `-T`, `-v` et `-V`.

Les options suivantes appellent des commentaires.

- Options `-a` et `-F` : pas de changement dans les spécifications, mais pas encore implémentées aux dernières nouvelles (à l'heure où nous écrivons) dans Rakudo ; pour l'instant, utilisez la méthode `.split` manuellement.
- Option `-l` (traitement automatique des fins de lignes) : sans objet, c'est maintenant le comportement par défaut.
- Options `-M` et `-m` : seule l'option `-M` est conservée (et l'option « no Module » n'est plus disponible).
- Option `-E` : comme toutes les fonctionnalités permises par `-E` (par rapport à `-e`), `-E` n'a plus d'objet et n'est plus disponible.
- Options de débogage `-d`, `-dt`, `-d:foo`, `-D`, etc. : remplacées par l'option métasyntaxique `++BUG`.
- Option `-s` : sans objet, l'analyse est maintenant faite par la liste de paramètres de la fonction `MAIN` :

```
# Perl 5
#!/usr/bin/perl -s
if ($xyz) { print "$xyz\n" }
# ./exemple.pl -xyz=5      # -> 5

# Perl 6
sub MAIN ( Int :$xyz ) {
    say $xyz if $xyz.defined;
}
# perl6 exemple.p6 --xyz=5  # -> 5
# perl6 exemple.p6 -xyz=5  # -> 5
```

- Option `-t` : les avertissements de type « taint » ne sont pas complètement spécifiés.
- Options `-P`, `-u`, `-U`, `-W` et `-X` : supprimées (voir **Synopsis 19**).
- Option `-w` : sans objet, l'activation des *warnings* est maintenant le comportement par défaut.

## 8 - Opérations sur des fichiers

### 8-1 - Lire les lignes d'un fichier dans un tableau

En Perl 5, une façon idiomatique de lire un fichier dans un tableau peut s'écrire comme suit :

```
open my $fh, "<", "file" or die "$!";
```

```
my @lines = <$fh>;  
close $fh;
```


En Perl 6, c'est plus simple :

```
my @lines = "file".IO.lines;
```

N'essayez **pas** de gober un fichier et de diviser la chaîne résultante sur les caractères de fin de ligne, car ceci vous donnera un tableau dont le dernier élément est un de plus que ce que vous attendez probablement (et c'est plus compliqué que prévu) :

```
# Création du fichier (de 3 lignes) à lire  
# spurt écrit directement dans un fichier ou une chaîne de caractères, sans  
# avoir besoin d'ouvrir puis fermer un descripteur  
spurt "fic-test", q:to/END/;  
Ligne 1  
Ligne 2  
Ligne 3  
END  
# Relecture du fichier  
my @lignes = "fic-test".IO.slurp.split(/\n/);  
say @lignes.elems;    #-> 4 éléments
```

## 9 - Variables spéciales

Les variables spéciales de Perl 5 mentionnées dans ce chapitre sont décrites dans le document  [perlvar](#).

### 9-1 - Variables d'environnement

#### 9-1-1 - Chemin de la bibliothèque des modules Perl

En Perl 5, l'une des variables d'environnement Unix ou Linux utilisables pour spécifier des chemins de recherche supplémentaires pour les modules Perl est `PERL5LIB`.

```
$ PERL5LIB="/some/module/lib" perl program.pl
```

Même chose en Perl 6, si ce n'est qu'il faut changer le numéro de version :

```
$ PERL6LIB="/some/module/lib" perl6 program.p6
```

Comme en Perl 5, si vous ne définissez pas `PERL6LIB` dans le shell, vous devrez sans doute le faire dans le programme Perl en utilisant le pragma `use lib` :

```
use lib '/some/module/lib';
```

À noter que `PERL6LIB` est une facilité temporaire offerte au développeur Perl 6 (contrairement à `PERL5LIB` en Perl5) et il vaut mieux ne pas l'utiliser pour des programmes durables, car il se peut qu'elle soit supprimée. La raison en est que le mécanisme de chargement des modules en Perl 6 n'est plus calqué directement sur les chemins des hiérarchies de dossiers et fichiers des systèmes d'exploitation.

## 9-2 - Variables spéciales générales

Nom court P5	Nom long Perl 5	Perl 6 - Remarques
<code>\$_</code>	<code>\$ARG</code>	Même chose en Perl 6. Seule différence : un appel de méthode sans invoquant travaillera pas défaut sur <code>\$_</code> : <code>.say</code> est équivalent à <code>\$_say</code> ou à <code>say(\$_)</code> .
<code>@_</code>	<code>@ARG</code>	L'utilisation des signatures de fonction est la façon standard de Perl 6 pour récupérer les arguments d'une fonction. Mais si une fonction n'a pas de signature, alors ses arguments se retrouvent dans <code>@_</code> . Contrairement à <code>\$_</code> , <code>@_</code> n'est pas l'argument par défaut des méthodes de listes : <code>@_.shift</code> fonctionne, mais pas <code>.shift</code> .
<code>\$"</code>	<code>\$LIST_SEPARATOR</code>	N'existe pas en Perl 6.
<code>\$\$</code>	<code>\$PID</code> , <code>\$PROCESS_ID</code>	Remplacé en Perl 6 par <code>*\$PID</code> .
<code>\$0</code>	<code>\$PROGRAM_NAME</code>	Remplacé par <code>*\$PROGRAM_NAME</code> . À noter que <code>\$0</code> représente maintenant la première capture d'une regex.
<code>\$a</code> , <code>\$b</code>		<code>\$a</code> et <code>\$b</code> n'ont aucune signification spéciale en Perl 6, et sort ne les utilise pas de façon spéciale comme en Perl 5. Perl 6 propose des paramètres positionnels autodéclarés ( <i>placeholder variables</i> ) utilisant le twigil <code>^</code> (voir le chapitre <b>Twigils</b> du tutoriel), qui généralisent le rôle particulier qu'avaient <code>\$a</code> et <code>\$b</code> en Perl 5. On peut donc écrire : <pre>sort { \$^a cmp \$^z }, 1,5,6,4,2,3; # 1,2,3,4,5,6 sort { \$^g cmp \$^a }, 1,5,6,4,2,3; # 6,5,4,3,2,1 for 1..9 { say \$^c, \$^a, \$^b; last } # 312</pre>
<code>%ENV</code>		Remplacé en Perl 6 par <code>%*ENV</code> .
<code>@INC</code>		Remplacé en Perl 6 par <code>@*INC</code> .
<code>\$^I</code>	<code>\$INPLACE_EDIT</code>	Remplacé en principe par <code>*\$INPLACE_EDIT</code> , mais ne semble pas encore implémenté.
<code>\$^O</code>	<code>\$OSNAME</code>	Remplacé par plusieurs variables spéciales, dont <code>*\$KERNEL</code> et <code>*\$DISTRO</code> , qui admettent les méthodes <code>.version</code> et <code>.name</code> . Ces variables fonctionnent, mais ne sont pour l'instant pas suffisamment documentées. La variable <code>*\$VM</code> donne des informations sur la machine virtuelle employée (par exemple, chez moi, « moar (2015.6) »).
<code>\$^T</code>	<code>\$BASETIME</code>	Remplacé par <code>*\$INITITME</code> , mais avec une sémantique différente.
<code>\$^V</code>	<code>\$PERL_VERSION</code>	Remplacé par <code>\$PERL.version</code> .

## 9-3 - Variables associées aux expressions régulières

Les variables `$``, `$&` et `$'` n'existent plus en Perl 6 (remplacées pour l'essentiel par des variations sur `$/`) et, avec leur élimination, les problèmes de performances qu'elles pouvaient générer en Perl 5 disparaissent également.

Nom court P5	Nom long Perl 5	Perl 6 - Remarques
<code>\$1</code> , <code>\$2</code> , ...		Les variables <code>\$1</code> , <code>\$2</code> , <code>\$3</code> , etc. font la même chose (capture) en Perl 5 et en Perl 6, sauf qu'elles commencent maintenant à <code>\$0</code> . Elles sont des synonymes des éléments

		indités de la variable de reconnaissance <code>\$/</code> . <code>\$0</code> est donc synonyme de <code>\$_[0]</code> , <code>\$1</code> de <code>\$_[1]</code> et ainsi de suite.
<code>\$&amp;</code>	<code>\$MATCH</code> (et <code>\$_^MATCH</code> )	<code>\$/</code> contient maintenant l'objet reconnu. Le <code>\$&amp;</code> de Perl 5 peut donc être obtenu en transformant <code>\$/</code> en chaîne de caractères : <code>~/</code> . <code>\$/Str</code> devrait marcher, mais <code>~/</code> semble avoir la préférence.
<code>\$'</code>	<code>\$PREMATCH</code> (et <code>\$_^PREMATCH</code> )	Remplacé par <code>\$/prematch</code> .
<code>\$'</code>	<code>\$POSTMATCH</code> (et <code>\$_^POSTMATCH</code> )	Remplacé par <code>\$/postmatch</code> .
<code>\$+</code>	<code>\$LAST_PAREN</code>	Cette variable n'existe pas en tant que telle en Perl 6, mais la même information existe dans <code>\$/[*-1]Str</code> .
<code>%+</code>	<code>%LAST_PAREN</code>	Information présente dans <code>\$/{\$match}</code> .

## 9-4 - Variables associées aux descripteurs de fichiers

Nom court P5	Nom long Perl 5	Perl 6 - Remarques
<code>\$ARGV</code>		Le nom du fichier en cours de lecture se trouve dans <code>*\$ARGFILES.filename</code> .
<code>@ARGV</code>		Le tableau <code>@*ARGS</code> contient les arguments de la ligne de commande.
<code>\$,</code>	<code>\$OFS</code> , <code>\$OUTPUT_FIELD_SEPARATOR</code>	Pas d'équivalent évident à l'heure actuelle.
<code>\$.</code>	<code>\$NR</code> , <code>\$INPUT_LINE_NUMBER</code>	Le numéro de la ligne courante est donné par la méthode <code>.ins</code> invoquée sur le descripteur de fichier voulu, par exemple : <code>*\$IN.ins</code> .
<code>\$/</code>	<code>\$RS</code> , <code>\$INPUT_RECORD_SEPARATOR</code>	Utiliser la méthode <code>.nl</code> sur le descripteur de fichier, par exemple : <code>*\$IN.nl</code> .
<code>\$\</code>	<code>\$ORS</code> , <code>\$OUTPUT_RECORD_SEPARATOR</code>	La méthode <code>.nl</code> sur le descripteur de fichier en sortie (par exemple : <code>*\$OUT.nl</code> ) semble fonctionner, mais la documentation n'est pas claire.
<code>\$ </code>	<code>\$OUTPUT_AUTOFLUSH</code>	L'autoflush n'est actuellement pas encore implémenté en Perl 6.

## 10 - Modules

### 10-1 - Importation de fonctions depuis un module

En Perl 5, il est possible d'importer sélectivement seulement certaines fonctions d'un module donné :

Perl 5

```
use NomDeModule qw{toto titi};
```

En Perl 6, c'est dans le module que l'on spécifie les fonctions à exporter, en utilisant lors de la définition de ces fonctions le rôle `is export`, et toutes les fonctions munies de ce rôle seront exportées. Ainsi, le module `Parler` suivant

```
unit module Parler;

sub dit($a) is export {say "La personne dit $a" }
sub crie($a) is export {say "La personne crie $a"}
sub murmure($a) {say "La personne murmure $a" }
```



exporte les fonctions `dit` et `crie`, mais pas `murmure`.

Pour utiliser ce module, il suffit d'insérer la ligne `use Parler;` et les fonctions `dit` et `crie` seront disponibles :

```
use Parler;

dit("Bonjour");           # "La personne dit bonjour"
crie("Au secours!");       # "La personne crie au secours!"
```

Si l'on essaie d'utiliser la fonction `murmure`, une erreur `Undeclared routine` se produit à la compilation.

N'est-il alors plus possible d'importer sélectivement les fonctions comme en Perl 5 ? Si, c'est possible et ce serait dommage de s'en passer, mais c'est un peu plus compliqué parce que cela sort de la sémantique recherchée et oblige de descendre d'un niveau. Pour permettre ce comportement comme en Perl 5, il faut définir au sein du module une fonction `EXPORT` qui spécifie les fonctions à exporter et (dans l'implémentation actuelle de Rakudo) supprimer la déclaration de module `Parler`. (À noter que l'absence de déclaration de module n'est pas entièrement conforme à la *Synopse 11*, mais ça fonctionne.)

Le module `Parler` est maintenant simplement un fichier nommé `Parler.pm` contenant par exemple ce qui suit :

```
use v6;

sub EXPORT(*@import-list) {
    my %exportable-subs =
        '&dire' => &dit,
        '&crie' => &crie,
        ;
    my %subs-to-export;
    for @import-list -> $import {
        if grep $sub-name, %exportable-subs.keys {
            %subs-to-export{$sub-name} = %exportable-subs{$sub-name};
        }
    }
    return %subs-to-export;
}

sub dit($a, $b, $c) { say "Elle dit $a, $b, $c" }
sub crie($a) { say "Elle crie $a" }
sub murmure($z) { say "Elle murmure $z" }
```

Il faut remarquer que les fonctions ne sont plus exportées explicitement par le truchement du rôle `is export`. On définit à la place la fonction `EXPORT` qui spécifie les fonctions du module que l'on désire rendre disponible à l'exportation, puis on alimente un hachage contenant les fonctions qui seront réellement exportées. Le tableau `@import_list` est la liste de fonctions demandées par le code appelant dans sa clause `use ...`, ce qui permet à l'utilisateur du module d'importer sélectivement certaines des fonctions mises à sa disposition.

Pour n'importer que la fonction `dit`, il suffit d'écrire :

```
use Parler <dit>;
dit("Bonjour");           # Elle dit Bonjour
```

On voit donc ici que, bien que la fonction `crie` soit exportable, elle ne sera pas disponible dans le code appelant :

```
use Parler <dit>;
dit("Bonjour");           # Elle dit Bonjour
crie("Au secours");       # "Undeclared routine: crie used at line 3"
```

Cependant, ceci fonctionnera :

```
use Parler <dit crie>;
dit("Bonjour");           # Elle dit Bonjour
```

```
crie("Au secours"); # Elle crie Au secours
```

À noter que la fonction `murmure` ne peut toujours pas être importée, même si elle est ajoutée dans la clause `use`, puisqu'elle ne figure pas dans le hachage `%exportable-sub` :

```
use Parler <dit crie>;
dit("Bonjour"); # Elle dit Bonjour
murmure("Chut!"); # "Undeclared routine: murmure used at line 3"
```

Pour faire fonctionner tout cela, il faut manifestement parcourir un chemin parsemé d'embûches. Dans l'utilisation normale dans laquelle on spécifie les fonctions à importer à l'aide du rôle `is export`, Perl 6 crée automatiquement la fonction `EXPORT` correcte. C'est la façon normale de faire. Bien que cela soit possible, il faut vraiment se demander soigneusement si ça vaut vraiment la peine de se lancer dans sa propre fonction `EXPORT`.

## 10-2 - Modules standard

### 10-2-1 - Module `Data::Dumper`

En Perl 5, le module **`Data::Dumper`** est utilisé pour *sérialiser* des structures de données présentes en mémoire (c'est-à-dire leur donner un format permettant de les stocker dans un fichier ou de les envoyer par un réseau, et pouvoir récupérer le format d'origine ensuite) et aussi, à des fins de débogage, de permettre au programmeur de visualiser la structure de ses données.

En Perl 6, la méthode `.perl` (voir **Partie 3 du tutoriel De Perl 5 à Perl 6**), présente dans tous les objets, permet d'obtenir le même résultat.

```
# Soit la structure de données suivante:
my @array_of_hashes = (
  { NOM => 'pomme', type => 'fruit' },
  { NOM => 'brocoli', type => 'non merci, sans façon' },
);
# Perl 5
use Data::Dumper;
$Data::Dumper::Useqq = 1;
print Dumper \@array_of_hashes; # Noter l'antislash
# Perl 6
say @array_of_hashes.perl; # .perl sur le tableau, pas
                          # sur une référence.
```

Le module `Data::Dumper` de Perl 5 a des options d'appel permettant par exemple de nommer les `VARs`.

En Perl 6, préfixer d'un caractère deux-points («`:`») le sigil d'une variable la transforme en une paire (donnée de type `Pair`), dans laquelle la clef est le nom de la variable et la valeur de la paire le contenu de la variable.

```
# Soient les variables suivantes
# (noms des vars tirés des nombres Shadoks pour changer de toto) ;- )
my ( $ga, $bu, $zo ) = ( 42, 44, 46 );
my @meu = ( 16, 32, 64, 'Stop!' );
# Perl 5
use Data::Dumper;
print Data::Dumper->Dump (
  [ $ga, $bu, $zo, \@meu ],
  [ qw( ga bu zo *meu ) ],
);
# Affiche:
$ga = 42;
$bu = 44;
$zo = 46;
@meu = (
  16,
  32,
```

```

        64,
        'Stop!'
    );
# Perl 6
say [ :$ga, :$bu, :$zo, :@meu ].perl;
# Affiche
["ga" => 42, "bu" => 44, "zo" => 46, "meu" => [16, 32, 64, "Stop!"]]

```

## 10-2-2 - Module Getopt::Long

Le module **Getopt::Long** de Perl 5 n'est plus nécessaire en Perl 6, car l'analyse des arguments et options de la ligne de commande est maintenant réalisée en définissant une fonction **MAIN** (appelée automatiquement au lancement du programme si elle existe).

Les arguments d'appel du programme sont comparés aux paramètres formels de la fonction **MAIN** (autrement dit à la signature de **MAIN**) et cela permet de valider les arguments d'appel et d'initialiser les paramètres du programme aux valeurs voulues (voir aussi [le chapitre sur MAIN du tutoriel](#)).

Voici un exemple :

```

# Perl 5
use 5.010;
use Getopt::longueur;
use utf8;
GetOptions(
    'longueur=i' => \( my $longueur = 24 ), # numeric
    'fichier=s'  => \( my $data = 'fichier.dat' ), # string
    'verbeux'    => \( my $verbeux ), # indicateur booléen
) or die;
say $longueur;
say $data;
say 'Bavard : ', ($verbeux ? 'oui' : 'non') if defined $verbeux;
# perl exemple.pl
24
fichier.dat
perl exemple.pl --fichier=toto --longueur=42 --verbeux
42
toto
Bavard : oui

perl exemple.pl --longueur=abc
Value "abc" invalid for option longueur (number expected)
Died at c.pl line 3.

# Perl 6
sub MAIN ( Int :$longueur = 24, :fichier($data) = 'fichier.dat', Bool :$verbeux ) {
    say $longueur if $longueur.defined;
    say $data     if $data.defined;
    say 'Bavard : ', ($verbeux ?? 'oui' !! 'non');
}
# perl6 exemple.p6
24
fichier.dat
Bavard : non
# perl6 exemple.p6 --fichier=toto --longueur=42 --verbeux
42
toto
Bavard : oui
# perl6 exemple.p6 --longueur=abc
Usage:
  c.p6 [--longueur=<Int>] [--fichier=<Any>] [--verbeux]

```

À remarquer que Perl 6 autogénère un message décrivant la syntaxe d'appel en cas d'erreur lors de l'analyse des arguments de la ligne de commande.

## 11 - Traduction automatique de Perl 5 en Perl 6

Une façon rapide de trouver la version Perl 6 d'une construction Perl 5 est de la faire passer dans l'un des outils de traduction automatique. À noter cependant qu'aucun de ces outils n'est à ce jour complet (en particulier, mais pas seulement, en ce qui concerne les nouvelles fonctionnalités des versions récentes de Perl 5).

### 11-1 - Blue Tiger

Le projet **Blue Tiger** est dédié à la modernisation automatique de code Perl. Il n'existe (pour l'instant) pas d'interface Web, il faut donc l'installer localement pour en tirer parti.

Il contient aussi un programme distinct de traduction des regex Perl 5 en Perl 6.

### 11-2 - Perlito

Le projet **Perlito** est un traducteur en ligne !

Ce projet est une suite de compilateurs croisés, dont l'un effectue **une traduction de code Perl 5 en code Perl 6**. Il dispose d'une interface Web permettant de l'utiliser depuis un navigateur Web sans installation logicielle préalable.

### 11-3 - MAD

Le code de Larry Wall pour traduire de Perl 5 en Perl 6 a beaucoup vieilli et n'est (actuellement) plus viable pour les versions récentes de Perl 5.

MAD (Misch Attribute Decoration) est une option de configuration lorsque l'on construit un exécutable Perl depuis une distribution. Un exécutable Perl analyse et traduit le code source Perl qui lui est présenté en un arbre d'opérations (*op-tree*) et exécute ensuite le programme en parcourant cet arbre. En temps normal, tous les détails de cette analyse sont jetés à l'issue du processus. Si MAD a été activé, l'exécutable Perl sauvegarde les détails de cet arbre dans un fichier XML qui peut ensuite être retraité et transformé en code Perl 6 par un analyseur MAD.

Veuillez consulter le canal IRC `#perl6` pour déterminer la meilleure version de Perl 5 à utiliser pour vos expériences avec MAD. En l'état actuel, attendez-vous à devoir bien bidouiller pour réussir à vous en servir.

### 11-4 - Module Inline::Perl5

Le module **Inline::Perl5** de Perl 6 permet d'utiliser du code Perl 5 à l'intérieur d'un programme Perl 6. Son principal avantage est de permettre d'utiliser les modules Perl 5 du CPAN (y compris des modules XS contenant une partie écrite en C) depuis un programme Perl 6 et donc d'offrir accès à une grande partie de la vaste bibliothèque de modules Perl 5 du CPAN.

Certains modules ou « frameworks » particulièrement complexes ou « magiques » ne fonctionnent sans doute pas encore, mais il est par exemple possible depuis août 2014 d'utiliser un module assez pointu comme DBI depuis Perl 6.

L'outil standard d'installation de modules Perl 6 s'appelle **Panda** (livré en standard avec *Rakudo Star*). La procédure classique d'installation se déroule comme suit :

```
panda install Inline::Perl5
perl Makefile.PL
make
make test
make install
```

Dans le même ordre d'idée, rappelons la possibilité d'utiliser le modificateur `:P5` ou l'adverbe `:Perl5` pour utiliser les expressions régulières de Perl 5 dans un programme Perl 6 (voir § 6.4).

## 11-5 - Module `Inline::Perl6`

Le module `Inline::Perl6` est en quelque sorte la réciproque du module présenté ci-dessus : c'est un module Perl 5 qui permet d'utiliser du code Perl 6 à l'intérieur d'un programme Perl 5. Pour ce faire, il intègre un Perl 6 Rakudo (MoarVM).

On peut imaginer utiliser ce module à deux types d'objectifs.

À des fins essentiellement pédagogiques (par exemple pour s'autoformer), on peut continuer à programmer en Perl 5, ou à utiliser un programme Perl 5 existant, mais commencer à se familiariser « sans douleur » avec les nouveautés syntaxiques ou sémantiques de Perl 6.

L'autre objectif plus pragmatique est de permettre l'utilisation en Perl 5 de certains concepts de Perl 6 absents ou moins puissants en Perl 5, ou plus simplement d'utiliser des méthodes Perl 6.

À noter que l'utilisation de ce module nécessite d'installer sur l'ordinateur une version à jour de Perl 6 Rakudo et du module `Inline::Perl5` pour Perl 6 décrit au § 12.4 ci-dessus.

## 12 - Remerciements

Je remercie les auteurs anonymes de la  **Documentation officielle de Perl 6**, dont j'ai abondamment utilisé le travail.

Je remercie **Djibril**, **f-leb** et **Claude Leloup** pour leur relecture attentive de ce tutoriel et leurs très utiles suggestions d'amélioration.