

De Perl 5 à Perl 6

Partie 1 : les bases du langage




Par Moritz Lenz - [Laurent Rosenfeld](#) 

Date de publication : 22 octobre 2014

Dernière mise à jour : 6 novembre 2015

DÉBUTANT

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

Perl 6 garde une forte ressemblance avec Perl 5, c'est indiscutablement un langage de la même famille, mais c'est réellement un nouveau langage de programmation. Autant les différentes versions de Perl 5 s'étaient efforcées de conserver dans toute la mesure du possible la rétrocompatibilité avec les versions antérieures, avec Perl 4 et même les précédentes versions (avec les avantages et les inconvénients que cela peut présenter), autant Perl 6 a décidé de s'affranchir des exigences de la rétrocompatibilité pour créer un nouveau langage réellement très ambitieux, entièrement et radicalement moderne, et encore plus puissant et plus expressif.

Cette première partie examine surtout les différences de syntaxe de base entre Perl 5 et Perl 6 et les apports de ces différences. Les parties suivantes (voir les **partie 2** et **partie 3**) se pencheront plus sur les notions complètement nouvelles de Perl 6.

Laurent R.

*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 - Deuxième partie : les nouveautés
- De Perl 5 à Perl 6 - Troisième Partie : approfondissements
- De Perl 5 à Perl 6 - Annexe 1: Ce qui change entre Perl 5 et Perl 6
- Les regex et grammaires de Perl 6 : une expressivité sans précédent

1 - Introduction.....	5
1-1 - Ce qu'est ce document et ce qu'il n'est pas.....	5
1-2 - Installer Perl 6 sur votre ordinateur.....	6
2 - Commentaires, chaînes de caractères, tableaux, tables de hachage.....	6
2-1 - Résumé.....	6
2-2 - Description.....	7
2-2-1 - Commentaires.....	7
2-2-2 - Chaînes de caractères.....	8
2-2-3 - Tableaux.....	9
2-2-4 - Tables de hachage.....	9
2-3 - Remarques.....	10
2-4 - Voir aussi.....	11
3 - Les types.....	11
3-1 - Exemples.....	11
3-2 - Description.....	11
3-3 - Introspection.....	12
3-4 - Motivation.....	12
3-4-1 - Fiabilité de programmation.....	12
3-4-2 - Optimisation.....	12
3-4-3 - Extensibilité.....	12
3-5 - Voir aussi.....	12
4 - Principales structures de contrôle.....	13
4-1 - Résumé.....	13
4-2 - Description.....	13
4-2-1 - Branchements.....	13
4-2-2 - Boucles.....	14
4-3 - Voir aussi.....	15
5 - Fonctions et signatures.....	15
5-1 - Résumé.....	15
5-2 - Description.....	16
5-2-1 - Paramètres nommés.....	16
5-2-2 - Paramètres optionnels.....	17
5-2-3 - Interpolation.....	18
5-2-4 - Fonctions multiples.....	18
5-3 - Motivation.....	19
5-4 - Voir aussi.....	19
6 - Objets et classes.....	19
6-1 - Résumé.....	19
6-1-1 - Méthodes.....	20
6-1-2 - Attributs.....	21
6-1-3 - Héritage.....	21
6-1-4 - Rôles et composition.....	22
6-1-5 - Voir aussi.....	22
7 - Les contextes.....	23
7-1 - Résumé.....	23
7-2 - Description.....	23
7-2-1 - Le contexte d'arbre.....	23
7-3 - Motivation.....	24
7-4 - Voir aussi.....	24
8 - Regex (ou règles).....	24
8-1 - Résumé.....	24
8-2 - Description.....	25
8-2-1 - Nettoyage de la syntaxe.....	26
8-2-2 - L'objet reconnu.....	27
8-2-3 - Regex nommées et grammaires.....	28
8-3 - Motivation.....	29
8-4 - Voir aussi.....	29
9 - Conclusion de cette première partie du tutoriel.....	29

10 - Remerciements.....	30
-------------------------	----

1 - Introduction

Perl 6 est actuellement sous-documenté ⁽¹⁾ (à part les spécifications, qui sont plus destinées aux concepteurs du compilateur qu'à l'utilisateur ordinaire, et ne sont pas forcément d'un abord facile). La relative faiblesse de la documentation n'est pas une surprise parce qu'écrire un compilateur pour Perl 6 paraît bien plus urgent que d'écrire de la documentation destinée à l'utilisateur. Les ressources d'un projet Open Source ne disposant pas du financement d'une grande entreprise ou d'une riche fondation étant ce qu'elles sont, il est souvent nécessaire de faire des choix.

Malheureusement, cela signifie qu'il n'est pas facile d'apprendre Perl 6, et qu'il vous faut une motivation de fer pour essayer de l'apprendre à partir des spécifications, des canaux IRC ou des suites de tests. Surtout si vous n'êtes pas un virtuose de la langue de Shakespeare.

1-1 - Ce qu'est ce document et ce qu'il n'est pas

Cet article est la première partie d'un document essayant de commencer à fournir de la documentation en français sur le sujet. L'auteur de ces lignes est bien conscient que certains utilisateurs de Perl 6 n'ont jamais utilisé Perl 5, mais il suppose tout de même que les premiers utilisateurs de Perl 6 seront tout de même majoritairement d'anciens utilisateurs de Perl 5. Cela permet de se concentrer sur les différences entre Perl 5 et Perl 6 et donc d'apporter relativement plus d'informations qu'un document de même taille qui devrait tout reprendre à partir de zéro.

Ce tutoriel est divisé en une série de leçons. Chaque leçon portera sur un domaine assez limité et essaiera d'expliquer deux ou trois points essentiels avec de courts exemples. Nous essaierons aussi de montrer pourquoi certaines choses ont changé entre Perl 5 et Perl 6, et pourquoi c'est important.

Nous espérons aussi que la connaissance que vous tirerez des présentes leçons vous aidera à lire et comprendre les synopsis, qui constituent la source canonique de toute connaissance de Perl 6.

La définition du langage Perl 6

Après un processus initial de demandes de changement (RFC) ouvert à tous (361 demandes ont été reçues, au lieu de la vingtaine attendues) lancé en 2000, Perl 6 a été initialement défini par Larry Wall dans une série de documents de conception de haut niveau intitulés les Apocalypses (mot qu'il faut comprendre au sens du terme grec original de révélation, et non au sens commun de catastrophe ou même « fin du monde » qu'il a parfois, en particulier en français). Chacune des Apocalypses est un document historique, figé dans le temps et immuable (au point que, du fait du processus de maturation de la définition du langage au fur et à mesure de la rédaction des Apocalypses, certaines des dernières Apocalypses sont parfois en partie contradictoires avec les premières). Une autre série de documents, les Exégèses (mot signifiant « commentaires » s'appliquant généralement aux textes sacrés), formaient une sorte d'erratum aux Apocalypses afin de tracer les modifications nécessaires pour rendre le tout cohérent.

Du coup, il a fallu rédiger une nouvelle série de documents de spécifications plus techniques, les Synopsis (en anglais, The Synopses). Les Synopsis évoluent avec le temps et tiennent compte des découvertes acquises (et éventuellement des difficultés rencontrées) dans le processus de maturation du langage et de ses premières implémentations. Les Synopsis constituent la vraie définition du langage (autrement dit, s'il y a désaccord entre les Apocalypses et les Synopsis, ce sont les Synopsis qui font foi).

(1) C'était vrai en anglais au moment où nous avons rédigé cet article en 2014, mais la situation s'est bien améliorée depuis, la documentation en anglais s'est considérablement étoffée depuis. Cela reste vrai dans les autres langues, notamment en français, langue dans laquelle il n'existe aujourd'hui que très peu de documentation sur le sujet.

Cela dit, pour prétendre être un vrai compilateur Perl 6, le critère principal est de passer avec succès l'intégralité de la suite de tests dérivée des Synopsis (aux dernières nouvelles, la suite comptait plus de 40 000 tests unitaires). Il y a un peu de flou sur le sujet, mais il semble que les meilleures implémentations disponibles en 2014 passent entre 85 et 90 % des tests de la suite.

Il n'est cependant pas impossible que l'on voie réellement utiliser en production des implémentations des versions de Perl 6 fonctionnelles, mais n'implémentant pas une ou deux fonctions plus difficiles à mettre en œuvre (ou ayant été jugées moins prioritaires). Les fonctionnalités ayant pris du retard sont la paresse (évaluation retardée), la gestion du parallélisme et certains opérateurs particuliers. La paresse est assez difficilement négociable (c'est un objectif majeur), mais les coroutines et les threads pourraient peut-être attendre un deuxième temps au besoin, ou du moins n'être implémentées qu'en partie) sans que cela nuise réellement au reste du langage et à son exploitabilité.

Pour faciliter la lecture, aucune leçon ne devrait être trop longue, nous avons essayé de l'éviter.

Il se peut que ces leçons s'avèrent trop courtes pour vous permettre d'apprendre vraiment un nouveau langage de programmation, mais nous espérons qu'elles vous fourniront au moins une bonne base de travail et une bonne image du langage vous permettant d'en apprécier la beauté.

Le présent texte n'est pas un guide pour convertir des programmes Perl 5 en Perl 6 (mais **l'annexe 1 de cette série** essaie d'offrir ce service). Ce n'est pas non plus une liste exhaustive des différences entre les deux langages. Il n'est pas orienté vers l'état actuel des implémentations, mais s'appuie sur le langage idéal défini par les spécifications.

Les exemples de code de ce texte ont été testés avec Rakudo Star (ou Rakudo *), qui est sans doute l'implémentation la plus avancée de Perl 6. La distribution se compose de Rakudo Perl 6, un compilateur Perl 6 utilisant la machine virtuelle Parrot, la machine virtuelle MoarVM ou la machine virtuelle Java (JVM), de Parrot et/ou MoarVM et d'un certain nombre de modules.

1-2 - Installer Perl 6 sur votre ordinateur

Si vous désirez utiliser Perl 6 ou simplement tester, nous vous conseillons de télécharger Rakudo Star à **cette adresse**. Pour Windows, téléchargez un installateur Microsoft (MSI). Il faut choisir entre l'installateur pour la machine virtuelle Parrot et l'installateur pour MoarVM. À l'heure où nous mettons à jour cet article (septembre 2015), il est recommandé d'utiliser MoarVM qui est bien plus complet et plus abouti. Il suffit de double-cliquer sur le fichier MSI d'installation, comme pour installer la plupart des programmes Windows.

Il existe des packages d'installation pour diverses autres plateformes : Cygwin, Fedora, Debian, OpenSuse, FreeBSD, Mac, etc. Certains sont plus à jour que d'autres. Pour d'autres plateformes, il faut télécharger le *tar* des sources et construire l'exécutable soi-même (ce qui implique, au minimum de disposer du compilateur GCC, de *make*, de *git*). La situation évoluant assez rapidement, nous ne pouvons en dire plus ici, il faut consulter le **site de Rakudo**.

Vous pouvez aussi consulter la **FAQ Perl 6 de Developpez.com**.

À noter qu'il est tout à fait possible de faire cohabiter Perl 5 et Perl 6 sur la même machine.

2 - Commentaires, chaînes de caractères, tableaux, tables de hachage

2-1 - Résumé

```
use v6; # utilisation de Perl 6
```

```
# ceci est un commentaire

#`{ Ceci
est un
commentaire multiligne }

my $cinq = 5;
print "une chaine avec interpolation, comme en Perl $cinq\n";
say 'say() ajoute un caractere fin de ligne, comme en Perl 5 (à partir de Perl 5.10)';

my @array = 1, 2, 3, 'toto';
my $sum = @array[0] + @array[1];
if $sum > @array[2] {
    say "non execute";
}
my $nombre_d_elements = @array.elems;      # ou: +@array
my $dernier = @array[*-1];

my %hash = un => 1, deux => 2, trois => 3;
say %hash{'deux'};      # 2
say %hash<trois>;      # 3 pareil, sans guillemets ni apostrophes
# Ceci est une erreur en Perl 6 : %hash{bar}
# (car on cherche ci-dessus à appeler la fonction bar(), qui n'est pas déclarée)
```

2-2 - Description

Perl 6 ressemble à Perl 5, mais en mieux. Les instructions se terminent par des points-virgules. Le point-virgule est cependant optionnel après la dernière instruction d'un bloc et après une accolade fermante à la fin d'une instruction.

2-2-1 - Commentaires

Comme en Perl 5 et dans de nombreux autres langages, le signe croisillon (« # ») marque le début d'un commentaire qui s'étend jusqu'à la fin de la ligne. Autrement dit, tout ce qui suit un croisillon sur une ligne est ignoré par le compilateur.

Contrairement à Perl 5 ⁽²⁾, Perl 6 offre aussi la possibilité de faire un commentaire multiligne, ce qui est bien utile quand on désire désactiver temporairement une portion de code posant un problème (cas assez fréquent dans une séance de débogage). Un commentaire multiligne est introduit par la séquence « #` » suivie d'une accolade, d'un crochet ou d'une parenthèse ouvrant, et le commentaire se termine avec l'accolade, le crochet ou la parenthèse (de même type) fermant.

```
# commentaire uniligne, le compilateur ignore le code de cette ligne

my $pi #`{
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. } = 3.14159;
say "Pi is: $pi";
```

ce qui affiche à l'exécution :

```
Pi is: 3.14159
```

(2) Il y a cependant un moyen simple de le faire en Perl 5 en détournant de leur objet les directives POD « =for » et « =cut »

. Mais ce n'est pas leur objet et cela peut manquer de clarté, notamment pour un débutant.

Si l'on désire mettre en commentaire provisoirement du code, cela n'est pas suffisant, car il risque fortement de contenir une accolade fermante (ou un crochet ou une parenthèse). Qu'à cela ne tienne, on peut doubler l'accolade (ou même la tripler), ou utiliser une combinaison crochet-parenthèse (et ainsi de suite) :

```
use v6;

my $pi #`{{{
Commentaire multiligne:
Ce commentaire est ouvert avec 3 accolades ouvrantes.
Il faut le fermer avec trois accolades fermantes }.
}}} = 3.14159;
say "Pi is: $pi";

my $e #`[
Commentaire multiligne:
Ce commentaire est ouvert avec une accolade {
et un crochet.
Il faut le fermer avec un crochet fermant ]
et une accolade fermante }.
Ainsi: ]] = 2.71828;
say "e vaut: $e";

my $nb_or = #`({
Autre commentaire multiligne.
On a ouvert avec une parenthese ( et une
accolade {, on ferme avec une accolade }
et une parenthese ), comme suit: }) 1.618;
say "Le nombre d'or est: $nb_or";
```

Ce qui affiche :

```
perl6 test_perl6_7.pl
Pi is: 3.14159
e vaut: 2.71828
Le nombre d'or est: 1.618
```

Il existe une troisième sorte de commentaire, les commentaires documentaires. Ils n'ont pas besoin d'être du code valide, mais, contrairement aux deux formes précédentes de commentaires, ils ne sont pas complètement ignorés par le compilateur, qui garde en mémoire leur contenu pour éventuellement le restituer sur demande. Les deux signes « `#=` » introduisent un commentaire qui explique certaines choses à propos de la fonction ou de la méthode définie immédiatement après. Le commentaire placé dans ce contexte est accessible via la méthode d'introspection `WHY` appliquée à ladite fonction ou méthode. En plus de commenter le code comme les autres formes de commentaires, cela peut permettre par exemple de renvoyer à la fonction appelante ou d'imprimer un message explicite sur la syntaxe correcte d'appel.

```
#= ERREUR. Syntaxe d'appel correcte: ...
sub ma_fonction{
    ...
}
```

2-2-2 - Chaînes de caractères

Les chaînes sont mises entre guillemets (ou *double quotes*), auquel cas l'interpolation des variables a lieu, ou entre apostrophes (ou *quotes simples* ou *single quotes*). Le caractère d'échappement barre oblique inversée (« `\` », ou *antislash* ou *backslash*) fonctionne comme en Perl 5.

Cependant, les règles d'interpolation ont quelque peu changé :

```
my $scalaire = 6;
my @tableau = 1, 2, 3;
say "Perl $scalaire";           # imprime: 'Perl 6'
say "Un @tableau[]";           # imprime: 'Un 1 2 3'
say "@tableau[1]";             # imprime: '2'
```



```
say "Code: { $scalaire * 2 }" # imprime: 'Code: 12'
```

Les tableaux et tables de hachage ne sont interpolés que s'ils sont suivis d'un indice (ou d'une invocation de méthode se terminant par des parenthèses (par exemple « une obj.methode() »). En cas d'indice vide, c'est l'ensemble de la structure qui est interpolé.

2-2-3 - Tableaux

Les variables de type tableau commencent toujours par le *sigil* « @ ». Mais, contrairement à Perl 5, les tableaux conservent le *sigil* « @ » même quand on accède à un élément du tableau à l'aide d'un indice.

```
my @a = 5, 1, 2; # les parenthèses ne sont plus nécessaires
say @a[0];      # Oui, ça commence avec une @
say @a[0, 2];   # les tranches de tableau fonctionnent comme en Perl 5
```

Les listes se construisent avec l'opérateur virgule. « 1, » est une liste, mais « (1) » n'en est pas une.

Comme tout est objet en Perl 6, il est possible d'appeler des méthodes sur les tableaux :

```
my @b = @a.sort;
@b.elems; # nombre d'éléments
if @b > 2 { say "yes" } # fonctionne toujours
@b.end;   # indice du dernier élément. Remplace $#b
my @c = @b.map({$_ * 2}); # Oui, map est aussi une méthode.
```

Il y a une forme plus brève pour remplacer l'ancienne construction de citation `qw / ... /` ;

```
my @methods = <shift unshift push pop end delete sort map>;
```

2-2-4 - Tables de hachage

Vues en contexte de liste, les tables de hachage de Perl 5 sont des listes ayant un nombre pair d'éléments. Dans ce même contexte, les tables de hachage de Perl 6 sont des listes de paires (d'objets de type Pair). La différence peut paraître subtile, mais elle est réelle. Les paires sont utilisées pour faire d'autres choses en Perl 6, comme les arguments nommés d'une fonction, mais nous y reviendrons plus loin.

De même que pour les tableaux, le sigil « % » reste invariant quand on accède à un élément individuel d'un hachage. Et les hachages ont également des méthodes telles que `keys` et `values` que l'on peut invoquer sur eux.

```
my %boissons =
  France => 'Vin',
  Angleterre => 'Tea',
  USA => 'Coke';

say "Boisson favorite en France : ", %boissons{'France'};
say "Pays: ", %boissons.key.sort;
say "Boissons: " %boissons.values.sort;
```

Ce qui affiche :

```
Boisson favorite en France : Vin
Pays: Angleterre France USA
Boissons: Coke Tea Vin
```

À noter que quand on accède aux éléments d'un hachage avec la syntaxe `%hash{...}`, la clef n'est pas mise implicitement entre guillemets comme en Perl 5. En fait, `%hash{toto}` n'accède pas à la clef « toto » du hachage, mais appelle la fonction `toto()`. Le mécanisme de mise implicite entre guillemets ne disparaît pas pour autant, il obéit seulement à une syntaxe modifiée :

```
say %boissons<Angleterre>;      # imprime 'Tea'
```

L'opérateur `each` (partiellement bogué et peu fiable en Perl 5) est remplacé par l'opérateur `pairs`, qui retourne les paires du hachage :

```
my %hash = qw (jan 1 fev 2 mar 3 avr 4 mai 5 jun 6);
say %hash;      # imprime: ("jan" => "1", "fev" => "2", "mar" => "3",
                  # "avr" => "4", "mai" => "5", "jun" => "6").hash

say values %hash;      # imprime 1 2 3 4 5 6
say %hash.keys;        # imprime jan fev mar avr mai jun
say $_ for %hash.pairs;
```

La dernière ligne de code ci-dessus, qui pourrait aussi s'écrire `say $_ for pairs %hash;`, affiche :

```
"jan" => "1"
"fev"  => "2"
"mar"  => "3"
"avr"  => "4"
"mai"  => "5"
"jun"  => "6"
```

ce qui montre bien l'existence réelle de paires, puisque la variable `$_` reçoit bien successivement chaque paire avant de l'afficher.

2-3 - Remarques

La plupart des opérateurs internes existent à la fois sous la forme de méthodes et de fonctions. Par exemple, pour trier un tableau, on peut aussi bien utiliser « `sort @tableau` » que « `@tableau.sort` ». De même, les bouts de code ci-dessus montrent des exemples d'utilisation des opérateurs `values`, `keys` et `pairs` pouvant être employés avec les deux formes syntaxiques.

En Perl 5, les espaces n'ont la plupart du temps pas d'importance. On peut par exemple écrire :

Code Perl 5

```
$ perl -E '@a = qw / 1 2 3 4/; say $a [2]'
3
```

En Perl 6, il n'est pas possible de laisser un espace blanc (ou plusieurs) entre un identifiant et un opérateur postfixé :

Code erroné en Perl 6

```
my @a = qw / 1 2 3 4/; # ERRONÉ: espace après l'opérateur qw
say @a [2]; # erreur (espace entre l'identifiant et l'indice)
say @a[2]; # OK, pas d'espace en trop
```

Toutefois, l'opérateur `unspace` (barre oblique inverse) permet d'ajouter des espaces (et même des commentaires) à peu près n'importe où :

```
my @a = qw / 1 2 3 4/;
say @a\ [2]; # pas d'erreur
```

De même, par exemple pour aligner des opérateurs :

```
say %hash\    { $key };      #OK
say @array\   [ $ix ];       #OK
say $subref\  ( $arg );      #OK
say $nr\      ++;            #OK
$object\     .say();         #OK
# ou même:
$object\     # commentaire
```

```
.say;      #OK
```

Enfin, il faut savoir que les constructions [...] et {...} (juste après un identifiant de variable) ne sont en fait que des invocations de méthodes ayant une syntaxe particulière, mais ne sont pas liées spécifiquement aux tableaux et aux hachages. Ceci signifie qu'elles ne sont pas liées à un *sigil* particulier.

```
my $a = [1, 2, 3];
say $a[2];      # imprime: 3
```

Du coup, il n'y a pas besoin d'une syntaxe de déréférencement particulière. Il est possible de créer des objets qui peuvent agir à la fois comme des tableaux, des hachages ou des fonctions.

2-4 - Voir aussi

<http://perlcabal.org/syn/S02.html>, <http://perlcabal.org/syn/S29.html>

<http://laurent-rosenfeld.developpez.com/tutoriels/perl/perl6/annexe-01/#L1>

3 - Les types

3-1 - Exemples

```
my Int $x = 3;
$x = "toto";      # erreur!
say $x.WHAT;      # 'Int()' ou (Int)

# vérification d'un type:
if $x ~~ Int {
    say '$x contient un entier'
}
```

3-2 - Description



Perl 6 est typé. D'une certaine façon, tout est objet et a un type. Les variables peuvent avoir des contraintes de typage, mais ce n'est pas indispensable, Perl ne se plaint pas si on ne lui donne pas d'information de type (sauf si on lui demande de la faire).

Voici quelques types de base qu'il est bon de connaître :

```
'a string'      # Str (chaîne de caractères)
2               # Int (nombre entier)
3.14            # Rat (nombre rationnel)
(1, 2, 3)       # Seq (suite)
'Wall' => 'Larry' # Pair (une paire)
```

Tous les types internes « normaux » commencent par une lettre capitale. Tous les types normaux héritent de Any et absolument tout hérite de Mu. ⁽³⁾

On peut restreindre le type de valeurs que peut contenir une variable en ajoutant un nom de type à sa déclaration.

(3) Le type Mu est au sommet de la hiérarchie des types, mais ne sert pas directement dans la programmation courante (à part sans doute pour ceux qui programment le compilateur Perl 6 lui-même et la grammaire du langage). Deux types héritent directement de Mu : Any et Junction. On n'utilise en principe pas non plus Any pour déclarer une variable (mais si on déclare une variable sans préciser de type, et sans la définir, Perl lui donnera provisoirement le type Any), mais on peut s'en servir pour déclarer un nouveau type non interne spécifique à une application. Ce point sera complété dans le chapitre sur les  **jonctions** de la  **seconde partie** de cet article.

```
my Numeric $x = 3.4; # $x ne peut prendre que des valeurs numériques
$x = "toto"         # ERREUR: "Type check failed in assignment..."
my Int @a = 1, 2, 3; # tableau d'entiers
@a[1] = 1.1;        # ERREUR: "Type check failed in assignment..."
```

C'est une erreur que d'essayer de mettre dans une variable une valeur du « mauvais » type (c'est-à-dire qui n'est ni du type spécifié, ni de ses sous-types).

Une déclaration de type sur un tableau s'applique à son contenu, si bien que « `my Str @s;` » déclare un tableau qui ne peut contenir que des chaînes de caractères.

Certains types représentent en fait toute une famille de types plus spécifiques. Par exemple, les entiers (type `Int`), les rationnels (type `Rat`) et les nombres à virgule flottante (type `Num`) respectent tous le type `Numeric`.

3-3 - Introspection

Il est possible de connaître le type exact d'une chose en appelant la méthode `.WHAT` :

```
say "foo".WHAT; # Str() ou (Str)
```

Toutefois, si vous désirez savoir si quelque chose est compatible avec un type spécifique, il existe une autre manière de le faire qui prend en compte les héritages de types et est donc recommandée dans la plupart des cas :

```
if $x ~~ Int {
    say 'La variable $x contient un entier';
}
```

3-4 - Motivation

Il n'est pas facile de résumer brièvement l'ensemble du système de typage, mais il y a d'excellentes raisons pour lesquelles nous avons besoin d'un système de typage.

3-4-1 - Fiabilité de programmation

Si l'on a déclaré une chose d'un certain type, on peut être sûr de pouvoir faire certaines opérations dessus. Il n'y a pas besoin de vérifier.

3-4-2 - Optimisation

Si l'on a les informations de type à la compilation, certaines optimisations sont possibles. Il n'y a en principe pas de raison que Perl soit plus lent que le C.

3-4-3 - Extensibilité

Avec les informations de type et les répartitions multiples, il est possible de raffiner certains opérateurs pour des types particuliers.

3-5 - Voir aussi

L'[annexe 2](#) de cette série présente **un chapitre bien plus détaillé sur les types** et leur utilisation.

http://perlcabal.org/syn/S02.html#Built-In_Data_Types,

4 - Principales structures de contrôle

4-1 - Résumé

```
if $pourcent > 100 {
    say "drôle d'arithmétique";
}
for 1..3 {
    # utilise $_ comme variable de boucle
    say 2 * $_;
}
for 1..3 -> $x {
    # variable de boucle explicite
    say 2 * $x;
}

while $chose.is_wrong {
    $chose.essaie_de_corriger;
}

die "Access denied" unless $password eq "Secret";
```

4-2 - Description

La plupart des structures de contrôle de Perl 6 ressemblent à celles de Perl 5. La principale différence visuelle est qu'il n'y a pas besoin de mettre entre parenthèses la condition suivant les mots-clefs `if`, `while`, `for`, etc.

À la vérité, il est même recommandé de ne pas mettre les conditions entre parenthèses. La raison est que tout identifiant immédiatement suivi (c'est-à-dire sans espace) d'une parenthèse ouvrante est interprété comme un appel de fonction, si bien que « `if($x < 3)` » va essayer d'appeler une fonction nommée `if`. Ajouter un espace entre le `if` et la parenthèse ouvrante résout le problème, mais il est plus sûr de ne simplement pas mettre de parenthèses.

4-2-1 - Branchements

L'instruction `if` est essentiellement inchangée, vous pouvez toujours y ajouter des opérations de branchement `elsif` ou `else`. L'instruction `unless` est toujours là, mais il n'est pas autorisé d'y adjoindre un `else`.

```
my $mouton = 42;
if $mouton == 0 {
    say "Comme on s'ennuie";
} elsif $mouton == 1 {
    say "Un mouton solitaire";
} else {
    say "Un troupeau, super!";
}
```

Il est toujours possible de placer un `if` ou un `unless` après une instruction, en tant que modificateur d'instruction :

```
say "Vous avez gagné!" if $reponse == 42;
```

Perl 6 offre des constructions dont pratiquement tout programmeur (quel que soit le langage) a dû rêver au moins une fois dans sa vie, surtout quand il débutait. Si l'on veut comparer trois variables, en Perl 5 (et dans la quasi-totalité des autres langages), il faut faire deux comparaisons distinctes reliées par un opérateur booléen :

Code Perl 5

```
my $c = my $d = my $e = 10;
print "Vrai\n" if $c == $d and $d == $e;
print "Vrai\n" if $c < 13 and $c > 7;
```

Nous en avons tous rêvé, Perl 6 l'a fait. En Perl 6, on peut chaîner les opérateurs et écrire beaucoup plus simplement et surtout plus naturellement : ⁽⁴⁾

Le rêve du programmeur

```
my $c = my $d = my $e = 10;
say "Vrai" if $c == $d == $e; # imprime Vrai
say "Vrai" if 13 > $c > 7;   # imprime Vrai
```

4-2-2 - Boucles

On peut modifier l'exécution d'une boucle avec les opérateurs `next` et `last`, comme en Perl 5.

La boucle `for` est maintenant utilisée exclusivement pour itérer sur des listes. Elle utilise par défaut la variable `$_` localisée, sauf si une variable de boucle explicite est fournie :

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

La construction `-> $x { ... }` s'appelle un « *pointy block* » (un « bloc pointu ») et est analogue à une fonction (ou fermeture) anonyme ou à un lambda en Lisp. Par défaut, le paramètre d'un bloc pointu est un alias en lecture seule (non modifiable) des valeurs successives du tableau.

Code erroné

```
my @list = 1..10;
for @list -> $x {
    $x++; # erreur ! $x est en lecture seule
}
```

Il est possible de rendre les valeurs modifiables en utilisant un *doubly pointy block* (bloc « doublement pointu ») :

```
my @list = 1..10;
for @list <-> $x {
    $x++;
}
say $_ for @list; # imprime les nombres de 2 à 11
```

Vous pouvez même utiliser plusieurs variables de boucle :

```
for 0..5 -> $pair, $impair {
    say "Pair: $pair \t Impair: $impair";
}
```

Ce qui affiche bien :

```
Pair: 0      Impair: 1
Pair: 2      Impair: 3
Pair: 4      Impair: 5
```

Cette syntaxe offre aussi un bon moyen d'itérer sur les couples clef/valeur de hachages :

```
my %h = a => 1, b => 2, c => 3;
for %h.kv -> $clef, $valeur {
    say "$clef: $valeur";
}
```

Ce qui affiche :

⁽⁴⁾ Voir aussi les chapitres sur les **jonctions** et sur les **opérateurs modifiés** (§ 12).

```
a: 1
b: 2
c: 3
```

La boucle `for` de style C a changé de nom et s'appelle désormais `loop` (et c'est la seule construction de boucle nécessitant encore des parenthèses). Plus encore qu'en Perl 5, ce style de boucle est à réserver à des cas bien particuliers sur la condition d'arrêt de la boucle ou la façon dont est modifiée la variable de boucle :

Boucle C de style C

```
loop (my $x = 2; $x < 100; $x = $x**2) {
    say $x;
}
```

Ce qui affiche :

```
2
4
16
```

4-3 - Voir aussi

http://perlcabal.org/syn/S04.html#Conditional_statements

<http://laurent-rosenfeld.developpez.com/tutoriels/perl/perl6/annexe-01/#L3>

<http://laurent-rosenfeld.developpez.com/tutoriels/perl/perl6/annexe-02/#L2>

5 - Fonctions et signatures

5-1 - Résumé

Fonctions et signatures

```
# sub sans signature, à la Perl 5
sub imprime_arguments {
    say "Arguments:";
    for @_ {
        say "\t$_";
    }
}

# Signature avec une arité et des types immuables:
sub distance(Int $x1, Int $y1, Int $x2, Int $y2) {
    return sqrt ($x2-$x1)**2 + ($y2-$y1)**2;
}
say distance(3, 5, 0, 1);

# Arguments par défaut
sub logarithme($nombre, $base = 2.7183) {
    return log($nombre) / log($base)
}
say logarithme(4);           # Utilise le second argument par défaut
say logarithme(4, 2);        # Second argument explicite

# arguments nommés
sub fais_le(:$quand, :$quoi) {
    say "doing $quoi at $quand";
}
fais_le(quoi => 'machin', quand => 'tout de suite'); # fait machin tout de suite
fais_le(:quand<midi>, :quoi('bidule')); # 'fait bidule à midi'
# illégal: fais_le("truc", "now")
```

5-2 - Description

Les fonctions (*subroutines*) sont déclarées avec le mot-clef `sub` et peuvent avoir une liste de paramètres formels ⁽⁵⁾, comme en C, en Java ou dans la plupart des autres langages. Ces paramètres peuvent avoir optionnellement des contraintes de type.

Par défaut, les paramètres sont en lecture seule (comme lors d'un passage de paramètre par valeur). Mais on peut changer cela grâce aux « *traits* » (propriétés définies au moment de la compilation) :

```
sub essaie-de-modifier($toto) {
    # Oui, les traits d'union sont permis dans le nom des fonctions
    $toto = 2;      # interdit
}

my $x = 2;
sub modifie($toto is rw) {
    $toto = 0;      # autorisé
}
modifie($x); say $x; # imprime: 0

sub quox($toto is copy){
    $toto = 3;
}
quox($x); say $x    # à nouveau 0
```

On peut rendre des paramètres optionnels en ajoutant un point d'interrogation à leur suite ou en fournissant une valeur par défaut :

```
sub fonction1($x, $y?) {
    if $y.defined {
        say "Le second paramètre a été fourni et défini";
    }
}

sub fonction2($x, $y = 2 * $x) {
    # ...
}
```

5-2-1 - Paramètres nommés

Si vous appelez une fonction avec la syntaxe suivante :

```
my_fonction($param1, $param2) {
    # ...
}
```

L'argument `$param1` est lié au premier paramètre formel, et l'argument `$param2` au second paramètre, et ainsi de suite. C'est ce que l'on appelle des paramètres positionnels, l'affectation des paramètres dans la fonction appelée dépend de l'ordre dans lequel la liste de paramètres est passée.

Il est parfois plus facile de se souvenir de noms plutôt que de numéros d'ordre (surtout si la liste des paramètres est un peu longue). C'est pourquoi Perl 6 a des paramètres nommés ⁽⁶⁾ :

```
my $r = Rectangle.new(
    x      => 100,
    y      => 200,
```

⁽⁵⁾ Perl 6 fait en principe la distinction entre les *paramètres* et les *arguments*. Le paramètre est le nom formel qui est attaché à l'argument entrant, alors que l'argument est la valeur qui est attachée ou liée au paramètre formel.

Nous essayons ici de respecter cet usage, mais, dans la pratique, la distinction n'est pas toujours très aisée.

⁽⁶⁾ Il est possible de faire à peu près la même chose en Perl 5 en utilisant un hachage d'arguments nommés.


```

    hauteur => 23,
    largeur => 42,
    couleur => 'black'
);

```

Quand vous voyez ceci, vous savez immédiatement ce que signifie chaque argument spécifique. Pour définir un paramètre nommé, il suffit de le préfixer du signe deux-points (« : ») dans la liste de signature :

```

sub aire(:$largeur, :$hauteur) {
    return $largeur * $hauteur;
}
aire(largeur => 2, hauteur => 3);
aire(hauteur => 3, largeur => 2); # même chose
aire(:hauteur(3), :largeur(2)); # idem

```

Le dernier exemple utilise la syntaxe dite de paire à deux-points (*colon pair syntax*). Si on omet la valeur, cela lui donne une valeur vraie (True) par défaut, et si l'on utilise l'opérateur de négation « ! », la valeur devient fausse (False) :

```

:dessine_perimetre # équivalent à "dessine_perimetre => True"
:!transparent     # équivalent à "transparent => False"

```

Dans la déclaration des paramètres nommés dans la fonction `aire` ci-dessus, le nom de la variable est également le nom de l'argument. Il est cependant possible d'employer un autre nom :

```

sub aire(:largeur($l), :hauteur($h)) {
    return $l * $h;
}
say aire(largeur => 2, hauteur => 3); # imprime 6

```

5-2-2 - Paramètres optionnels

Le fait de donner une signature à une fonction ne signifie pas qu'il faille connaître à l'avance le nombre des arguments. Il est possible d'utiliser des paramètres optionnels (parfois appelés « *slurpy parameters* », ou paramètres « gobe-tout »). Ce genre de paramètre doit être placé après les paramètres réguliers obligatoires et utilise toute la liste restante des arguments :

```

sub ma_fonction ($param1, *@reste) {
    say "Premier: $param1";
    say "Reste: @reste[]";
}
ma_fonction(1, 2, 3, 4);

```

Ce qui affiche :

```

Premier: 1
Reste: 2 3 4

```

Les paramètres nommés par défaut sont déclarés en utilisant un astérisque devant le paramètre hachage :

```

sub commande-repas($plat, *%extras) {
    say "Je désire un $plat, mais avec quelques modifications:";
    say %extras.keys.join(', ');
}

commande-repas('steak', :oignons, :bien-cuit);

```

Ce qui affiche :

```

Je voudrais un steak, mais avec quelques modifications:
oignons, bien-cuit

```

5-2-3 - Interpolation

Par défaut, les tableaux ne sont pas interpolés dans les listes d'arguments (ils ne sont pas « aplatis » en une liste unique). Donc, contrairement à Perl 5, on peut écrire :

```
sub a($scalaire1, @liste, $scalaire2) {
    say $scalaire2;
}

my @liste = "toto", "titi";
a(1, @liste, 2); # imprime 2
```

Mais cela signifie aussi qu'il n'est pas possible, par défaut, d'utiliser un tableau comme une liste d'arguments passés à une fonction :

Ne fait pas ce qui est prévu

```
my @indexes = 1, 4;
say "abcdefgh".substr(@indexes); # pas ce que l'on veut
```

Contrairement à ce que l'on pourrait penser, cela n'affiche pas « bcde » (comme le ferait `say "abcdefgh".substr(1, 4);`), mais « cdefgh ».

Ce qui se passe ici est que le premier argument de la méthode `substr` est censé être un `Int`, et la coercition fait que `substr` reçoit en définitive un seul argument, le nombre d'éléments du tableau `@indexes` et la fonction retourne la fin de la chaîne à partir du troisième caractère.

Pour obtenir le comportement voulu, il faut utiliser le préfixe « | » (ou hyperopérateur) devant le nom du tableau :

```
say "abcdefgh".substr(|@indexes); # imprime bcde
say "abcdefgh".substr(1, 4);      # même chose
```

5-2-4 - Fonctions multiples

Il est possible de définir des fonctions multiples (`multi subs`) ayant le même nom, mais une signature ou liste de paramètres différente, en utilisant le mot-clef `multi` :

multi subs

```
multi sub my_substr($str) { ... } # 1
multi sub my_substr($str, $start) { ... } # 2
multi sub my_substr($str, $start, $end) { ... } # 3
multi sub my_substr($str, $start, $end, $subst) { ... } # 4
```

Lorsque l'on appelle la fonction `my_substr()`, c'est celle ayant le nombre de paramètres correspondant à l'appel qui sera appelée.

Les *multi* n'ont pas besoin de se distinguer par leur arité (nombre de paramètres), elles peuvent aussi différer par le type des arguments employés.

```
multi sub util(Str $s) { say "Utilisation de la chaîne $s" }
multi sub util(Int $i) { say "Utilisation de l'entier $i" }

util("x");      # Utilisation de la chaîne x
util(2);        # Utilisation de l'entier 2
```

5-3 - Motivation

Personne ne doute de l'utilité des signatures explicites de fonctions : moins de frappes, moins de vérifications en double des arguments, et code autodocumenté. L'intérêt des paramètres nommés a déjà été discuté.

Ces signatures permettent aussi une introspection parfois bien utile. Par exemple, si l'on passe un bloc de code ou une fonction à la méthode `Array.sort`, le programme applique automatiquement une **transformation de Schwartz** aux données. Ce genre d'optimisation serait impossible en Perl 5, parce que l'absence de signatures explicites signifie que la fonction `sort` ne peut pas savoir combien d'arguments le bloc de code attend.

Les *multi* sont très utiles, car elles permettent de surcharger les opérateurs internes pour de nouveaux types. Supposons que l'on désire une version de Perl 6 localisée pour manipuler correctement les chaînes de caractères en langue turque, laquelle présente des règles inhabituelles pour les conversions de casse (mises en minuscules ou en capitales).

Au lieu de modifier le langage Perl, il suffit d'introduire un nouveau type de chaîne de caractères, `TurkishStr`, et d'ajouter des fonctions multiples remplaçant les fonctions internes pour ce nouveau type :

```
multi uc(TurkishStr $s) { ... }
```

Tout ce qui reste à faire est de s'assurer que les chaînes de caractères aient bien le type qui correspond à la langue dans laquelle elles sont écrites, et ensuite d'utiliser la fonction `uc` de la même façon que la fonction interne normale.

Comme les opérateurs sont aussi des fonctions, cette approche fonctionne également pour les opérateurs.

5-4 - Voir aussi

<http://perlcabal.org/syn/S06.html>

 <http://laurent-rosenfeld.developpez.com/tutoriels/perl/perl6/annexe-02/#L3>

6 - Objets et classes

6-1 - Résumé

```
class Forme {
    method aire { ... }    # littéralement trois points '...'
    has $.couleur is rw;
}

class Rectangle is Forme {
    has $.largeur;
    has $.hauteur;

    method aire {
        $!largeur * $!hauteur;
    }
}

my $x = Rectangle.new(
    largeur => 30.0,
    hauteur => 20.0,
    couleur => 'noir',
);
say $x.aire;           # imprime 600
say $x.couleur;       # imprime 'noir'
$x.couleur = 'bleu';
```

Perl 6 a un modèle d'objet bien plus développé que celui de Perl 5. Il possède des mots-clefs pour créer des classes, des rôles, des attributs et des méthodes, ainsi que des méthodes et attributs privés encapsulés. En fait, il se rapproche beaucoup plus du module Moose de Perl 5 (lequel module s'est inspiré du système d'objets de Perl 6).

Il y a deux façons de déclarer des classes.

```
Class NomClasse;  
# la définition de la classe commence ici
```

La première commence par la déclaration `Class NomClasse;` et s'étend jusqu'à la fin du fichier. Dans la seconde forme, le nom de la classe est suivi d'un bloc, et tout ce qui se trouve dans ce bloc constitue la définition de la classe :

```
Class NomClasse {  
    # définition de la classe dans ce bloc  
}  
# autres définitions de classes ou code autre
```

6-1-1 - Méthodes

On déclare les méthodes avec le mot-clef `method`. À l'intérieur d'une méthode, il est possible d'utiliser le terme `self` pour se référer à l'objet sur lequel la méthode a été appelée (l'*invocateur*).

On peut également donner un autre nom à l'invocateur en ajoutant un premier paramètre à la liste de signature et en le postfixant avec un caractère deux-points, « `:` ».

```
method faire ($x: $a, $b, $c) { ... } # $x représente l'invocateur
```

On peut appeler une méthode publique avec la syntaxe `$object.method` si la méthode ne prend pas d'argument et avec les syntaxes `$object.method(@args)` (notation pointée) ou `$object.method: @args` (notation indirecte) si elle doit recevoir des arguments.

```
class SomeClass {  
    # ces 2 méthodes ne font rien d'autre que retourner l'invocateur  
    method foo {  
        return self;  
    }  
    method bar(SomeClass $s: ) {  
        return $s;  
    }  
}  
my SomeClass $x .= new;  
$x.foo;           # même chose que $x  
$x.bar;           # pareil  
$x.foo.bar ;     # idem
```

La syntaxe `my SomeClass $x .= new;` n'est en fait qu'un raccourci pour `my SomeClass $x = SomeClass.new;`. Cela marche parce la déclaration de type fait de la variable un objet de type `SomeClass`, qui est un objet représentant la classe.

Les méthodes peuvent recevoir des arguments supplémentaires, comme les fonctions.

Les méthodes privées sont marquées comme telles à l'aide d'un point d'exclamation (« `!` »). Elles se déclarent avec la syntaxe `methode!methodname` et s'appellent avec la syntaxe `self!method_name`.

```
class Foo {  
    method !private($toto) {  
        return "Reçu le param $toto";  
    }  
  
    method publique {
```

```

        say self!private("foo");
    }
}
my Foo $x .= new;
$x.publique; # imprime "Reçu le param foo"

```

Les méthodes privées ne peuvent être appelées que depuis l'intérieur de la classe.

6-1-2 - Attributs

Les attributs se déclarent avec le mot-clef `has` et ont un *twigil*, c'est-à-dire un second caractère spécial après le *sigil*. Pour les attributs privés, c'est le point d'exclamation (« ! »), pour les attributs publics, c'est le point (« . »). Les attributs publics ne sont autre chose que des attributs privés dotés d'un accesseur public. Donc, pour modifier un attribut, il faut utiliser le sigil « ! » pour accéder à l'attribut réel, et non l'accesseur, sauf si l'accesseur est doté de la propriété `is rw`.

```

class SomeClass {
    has $!a;
    has $.b;
    has $.c is rw;

    method set_stuff {
        $!a = 1;    # OK, écriture dans l'attribut depuis la classe
        $!b = 2;    # idem
        $.b = 3;    # ERREUR, ne peut écrire sur un accesseur RO
    }

    method do_stuff {
        # on peut utiliser le nom public ou le nom privé
        # $!b et $.b sont vraiment la même chose
        return $!a + $!b + $.c;
    }
}

my $x = SomeClass.new;
say $x.a;    # ERREUR!
say $x.b;    # OK
$x.b = 2;    # ERREUR!
$x.c = 3;    # OK

```

6-1-3 - Héritage

L'héritage est réalisé à l'aide du *trait* `is`.

```

class Foo is Bar {
    # la classe Foo hérite de la classe Bar
    ...
}

```

Toutes les règles communes d'héritage s'appliquent. Les méthodes sont d'abord recherchées dans la classe directe de l'objet, puis, si elle n'y est pas trouvée, dans la classe parente (et ainsi de suite récursivement). De même, le type d'une classe enfant est conforme à celui d'une classe parente :

```

class Bar { }
class Foo is Bar { }
my Bar $x = Foo.new(); # ok, car Foo ~~ Bar

```

Dans cet exemple, le type de l'objet `$x` est `Bar`, et il est licite de lui affecter un objet de type `Foo`, car « tout `Foo` est un `Bar` ».

Les classes peuvent hériter d'autres classes multiples :

```

class ArrayHash is Hash is Array {

```

```
...  
}
```

Cela dit, l'héritage multiple pose de multiples problèmes, et est souvent déconseillé. Les rôles constituent souvent un choix plus fiable.

6-1-4 - Rôles et composition

En général, le monde n'est pas hiérarchique, et il est donc souvent difficile de tout faire entrer dans une hiérarchie d'héritage. C'est l'une des raisons pour laquelle Perl 6 instaure des rôles ⁽⁷⁾. Un rôle regroupe des comportements qui peuvent être partagés par différentes classes. Un rôle est assez semblable à une classe, mais la grande différence est qu'il n'est pas possible d'instancier des objets directement à partir de rôles. En outre, la composition de rôles multiples ayant les mêmes noms de méthodes génère des conflits au lieu de se résoudre silencieusement à l'une d'entre elles, comme c'est le cas avec l'héritage multiple.

Tandis que les classes ont pour principal objectif la conformité des types, les rôles sont en Perl 6 surtout un moyen de réutiliser du code.

```
role Dessinable {  
    has $.couleur is rw;  
    method dessiner { ... }  
}  
class Forme {  
    method aire { ... }  
}  
  
class Rectangle is Forme does Dessinable {  
    has $.largeur;  
    has $.hauteur;  
    method aire {  
        $!largeur * $!hauteur;  
    }  
    method dessiner() {  
        for 1..$.hauteur {  
            say 'x' x $.largeur;  
        }  
    }  
}  
  
Rectangle.new(largeur => 8, hauteur => 3).dessiner;
```

Ce qui affiche le rectangle :

```
~ perl6 test_rectangle.pl  
xxxxxxx  
xxxxxxx  
xxxxxxx
```

6-1-5 - Voir aussi

<http://perlcabal.org/syn/S12.html> <http://perlcabal.org/syn/S14.html> <http://www.jnthn.net/papers/2009-yapc-eu-roles-slides.pdf> http://en.wikipedia.org/wiki/Perl_6#Roles

Annexe 2 : Programmation objet en Perl 6

(7) De même que Java instaure des interfaces, qui jouent à peu près le même... rôle.

7 - Les contextes

7-1 - Résumé

```
my @a = <a b c>;
my $x = @a;
say $x[2];           # c
say (~2).WHAT;       # (Str)
say +@a;              # 3
if @a < 10 { say "petit tableau"; }
```

7-2 - Description

Si l'on écrit une ligne comme celle-ci

```
$x = @a;
```

en Perl 5, `$x` contient moins d'information que `@a` - `$x` contient seulement le nombre d'éléments du tableau `@a`. pour préserver toute l'information, il faut prendre une référence explicite : `$x = \@a`.

En Perl 6, c'est l'inverse qui se produit : par défaut, on ne perd rien, la variable scalaire stocke simplement le tableau. Cela a été rendu possible par l'introduction d'un contexte générique d'*item* (appelé scalaire en Perl 5) et des contextes plus spécialisés, numérique, entier et chaîne de caractères (*string*). Les contextes de liste et vide (*sink* ou *void*) restent inchangés.

Il est possible de forcer le contexte avec une syntaxe spécifique :

Syntaxe	Contexte
~truc	Chaîne de caractères
?truc	Booléen (logique)
+truc	Numérique
-truc	Numérique (et négation)
\$(truc)	Contexte d'item générique
@(truc)	Contexte de liste
%(truc)	Contexte de hachage
truc.tree	Contexte d'arbre (<i>tree context</i>)

7-2-1 - Le contexte d'arbre

Au tout début de Perl 6, de nombreux opérateurs avaient deux versions, une qui retournait une simple liste « aplatie » et une autre qui renvoyait une liste de tableaux.

Ceci est maintenant résolu en renvoyant une liste d'objets de type *Parcel* (paquet), et les parcs peuvent s'aplatir ou non, selon le contexte.

Considérons l'opérateur `Z` (lettre capitale Z, abréviation de *zip*) qui alterne les éléments de deux listes.

```
my @a = <a b c> Z <1 2 3>;
say @a.join;           # imprime : a1b2c3
```

Ce qui se passe ici est que la partie droite de la première instruction ci-dessus a renvoyé :

```
('a', 1), ('b', 2), ('c', 3)
```

et que l'affectation à un tableau, qui fournit un contexte de liste, a aplati les `parcels` internes en une liste unique. Si en revanche on écrit :

```
my @t = (<a b c> Z <1 2 3>).tree;
```

alors le tableau `@t` contient maintenant trois éléments, qui sont eux-mêmes des tableaux de deux éléments et ne s'aplatissent pas. Les instructions suivantes :

```
for @t -> @paire {
    say "premier: @paire[0] second: @paire[1]"
}
```

produisent maintenant l'affichage suivant :

```
premier: a second: 1
premier: b second: 2
premier: c second: 3
```

7-3 - Motivation

L'existence de contextes plus spécifiques permet de remettre à plus tard certaines décisions de conception lorsque l'on écrit une application. Par exemple, il peut paraître prématuré de décider ce qu'une liste doit retourner dans un contexte scalaire - une référence à la liste préserverait l'ensemble de l'information, mais n'est pas très utile dans des comparaisons numériques. D'un autre côté, une représentation sous la forme d'une chaîne de caractères serait le plus utile à des fins de débogage. Si bien que, quel que soit le choix fait, quelqu'un sera déçu.

Les contextes plus spécifiques permettent de ne pas faire ce choix, ils renvoient un format de valeur par défaut plus flexible, et les opérateurs qui n'aiment pas ce format peuvent simplement évaluer l'objet dans un contexte plus spécifique.

Pour certaines choses (comme les objets *Match*), les différents contextes renforcent considérablement leur utilité et leur beauté.

7-4 - Voir aussi

<http://perlcabal.org/syn/S02.html#Context>
[context.html](http://perlcabal.org/syn/S02.html#Context)

<http://perlgeek.de/blog-en/perl-6/immutable-sigils-and->

8 - Regex (ou règles)

8-1 - Résumé

```
# Une grammaire pour analyser syntaxiquement (parser) une URL:
grammar URL {
    token TOP {
        <schema> '://'
        [<ip> | <hostname> ]
        [ ':' <port> ]?
        '/' <path>?
    }
    token byte {
        (\d**1..3) <?{ $0 < 256 }>
    }
    token ip {
        <byte> [\.\ <byte> ] ** 3
    }
    token schema {
```



```

    \w+
}
token hostname {
    (\w+) ( \. \w+ ) *
}
token port {
    \d+
}
token path {
    <[ a..z A..Z 0..9 \- _ . ! ~ * ' ( ) : @ & = + $ , / ] > +
}
}

my $match = URL.parse('http://perl6.org/documentation/');
say $match<hostname>;      # perl6.org

```

8-2 - Description

Les *regex* sont l'un des domaines qui ont été le plus considérablement améliorés et refondus. Elles ne s'appellent plus « expressions régulières », car elles ont été tellement étendues qu'elles sont encore moins régulières que ne l'étaient celles de Perl 5.

Trois profondes améliorations ont été apportées aux regex.

Nettoyage de la syntaxe

De nombreux petits changements rendent les règles plus faciles à écrire. Par exemple, le point (« . ») reconnaît maintenant n'importe quel caractère, l'ancienne sémantique (tout caractère sauf un caractère fin de ligne) est obtenu maintenant avec « \N ».

Les modificateurs sont maintenant placés au début d'une regex et les groupes non capturants s'écrivent [...], ce qui est plus facile à retenir, à écrire et à lire que l'ancien (?:...).

Captures imbriquées et objet reconnu

En Perl 5, une regex telle que (a(b))(c) met ab dans \$1, b dans \$2, et c dans \$3 en cas de reconnaissance du motif. Cela change en Perl 6. Maintenant, \$0 (l'énumération commence à zéro) contient ab, et \$0[0] ou \$/[0][0] contient b. Et \$1 contient c. Ainsi, chaque niveau de parenthésage se retrouve dans un niveau d'imbrication supplémentaire dans l'objet reconnu produit.

Toutes les variables reconnues sont aliassées dans \$/, qui constitue l'objet reconnu (*match object*), et qui contient en fait un arbre entier de reconnaissance.

Regex nommées et grammaires

Il est possible de déclarer les regex avec des noms exactement comme avec des fonctions ou des méthodes. On peut ensuite utiliser ces regex nommées à l'intérieur d'autres règles en utilisant la syntaxe <nom>. Et l'on peut enfin inclure toutes ces regex dans des grammaires, qui sont semblables aux classes et supportent l'héritage et la composition.

Ces changements rendent les regex et les grammaires de Perl 6 bien plus faciles à écrire et à maintenir, et aussi bien plus puissantes.

Toutes ces modifications ont de profondes conséquences, dont on ne pourra ici que gratter la surface.

8-2-1 - Nettoyage de la syntaxe

Les caractères de type alphanumériques (c'est-à-dire les lettres, le souligné « `_` », les chiffres et tous les caractères Unicode) sont reconnus littéralement (ils établissent une correspondance avec eux-mêmes) et peuvent prendre une signification particulière (dite *métasyntaxique*) lorsqu'ils sont préfixés d'une barre oblique inversée (*backslash* ou *antislash*). Pour tous les autres caractères, c'est exactement l'inverse : ce sont des métacaractères sauf s'ils sont préfixés d'un backlash.

Littéral	Métasyntaxique
a b 1 2	\a \b \1 \2
* \: \. \?	* : . ?

Les symboles métasyntaxiques n'ont pas tous une signification (pour l'instant). Il est illicite d'employer ceux qui n'ont pas une signification dûment définie.

Il y a une autre façon de redonner leur sens littéral aux caractères métasyntaxiques des chaînes dans les regex, avec les apostrophes :

```
m/'a texte littéral: $#@!!'/'
```

Le changement de sémantique du point « `.` » a déjà été mentionné, ainsi que la nouvelle construction `[...]` pour les groupes non capturants. Les classes de caractères s'écrivent `<[...]>` et la négation d'une classe `<-[...]>`. `^` et `$` reconnaissent toujours respectivement le début et la fin de la chaîne, alors que l'on emploie `^^` et `$$` pour reconnaître le début et la fin d'une ligne. Cela signifie que les modificateurs `/s` et `/m` sont désormais inutiles et ont disparu.

Les modificateurs (ou adverbess) préfixent désormais la regex et utilisent la notation suivante :

```
if "abc" ~~ m:i/B/ {
    say "Reconnu!";
}
```

... qui se trouve être la même chose que la syntaxe de paire à deux-points (*colon pair*) utilisée pour passer des arguments nommés à des fonctions (§ 5.2.1).

Le comportement introduit en Perl 5 par le modificateur `/x` est maintenant le fonctionnement par défaut ; autrement dit, les espaces blancs sont ignorés.

Les modificateurs ont une forme courte et une forme longue.

Forme brève	Forme longue	Signification
:i	:ignorecase	Ignorer la casse (anciennement /i)
:m	:ignoremark	Ignorer les marques (accents, trémas, cédilles, tildes...)
:g	:global	Reconnaissance globale (/g)
:s	:sigspace	Reconnaissance littérale des espaces
:P5	:Perl5	Retour à une syntaxe compatible à Perl 5
:4x	:x(4)	Quantificateur : 4 occurrences (marche avec d'autres nombres)
:3rd	:nth(3)	Troisième occurrence
:ov	:overlap	Comme :g, mais tient compte des chevauchements de reconnaissance
:ex	:exhaustive	Reconnaissance de toutes les manières possibles
	:ratchet	Pas de retour arrière ou retour sur trace (<i>backtracking</i>)

Le :sigspace (ou :s) nécessite une petite explication. Il remplace tous les espaces du motif par <.ws> (c'est-à-dire qu'il appelle la règle ws sans conserver le résultat). Cette règle peut être annulée. Par défaut, il reconnaît un ou plusieurs espaces blancs s'il est encadré de caractères alphanumériques, et zéro ou plusieurs blancs dans le cas contraire.

(Il y a d'autres nouveaux modificateurs, mais sans doute pas aussi importants que ceux énumérés ci-dessus.)

8-2-2 - L'objet reconnu

Chaque reconnaissance génère un objet reconnu (*match object*), qui est contenu dans la variable spéciale \$/. C'est une chose versatile. En contexte booléen, il retourne Bool::True (vrai) si la reconnaissance a réussi. En contexte de chaîne de caractères, il renvoie la chaîne reconnue. En contexte de liste, il renvoie les chaînes capturées dans l'ordre des captures, et en contexte de hachage, il renvoie des captures nommées. Les méthodes .from et .to donnent respectivement la position dans la chaîne du début et de la fin de la reconnaissance.

```
if 'abcdefg' =~ m/(.()) (e | bla ) $<foo> = (.) / {
    say $/[0][0];           # d
    say $/[0];              # cd
    say $/[1];              # e
    say $/<foo>              # f
}
```

\$0, \$1, etc. sont simplement des alias pour \$/[0], \$/[1], etc. De même, \$<x> est un alias vers \$/<x> et \$/'x'.

À noter que tout ce que l'on peut accéder avec \$/[...] et \$/{...} est à nouveau un objet reconnu (ou une liste d'objets reconnus). Cela permet de vrais arbres d'analyse syntaxique (*parsing*) avec des règles.

8-2-3 - Regex nommées et grammaires

Les regex peuvent être utilisées de la façon traditionnelle, `m/.../`, ou être déclarées comme des fonctions ou des méthodes :

```
regex a { ... }
token b { ... }
rule c { ... }
```

La différence est qu'un token (unité lexicale) implique le modificateur `:ratchet` (ce qui signifie pas de retour arrière, comme si le motif avait le modificateur `(?> ...)` en Perl 5) et qu'une rule (règle) implique à la fois `:ratchet` et `:sigspace`.

Pour appeler une telle règle (nous appellerons les trois entités ci-dessus règles, indépendamment du mot-clef utilisé pour les déclarer), on met son nom entre des chevrons (signes inférieur à « < » et supérieur à « > », ou *angle brackets*) : `<a>`. Ceci ancre implicitement la règle à sa position actuelle dans la chaîne de caractères et place le résultat dans l'objet reconnu à l'emplacement `$/<a>`, autrement dit, c'est une capture nommée. Il est également possible d'appeler une règle sans capturer le résultat en préfixant son nom d'un point : `<.a>`.

Une grammaire est un groupe de règles permettant de construire progressivement un système de reconnaissance de texte beaucoup moins bien structuré que ce que l'on peut reconnaître avec les expressions régulières (même étendues) de Perl 5.

Voici par exemple une grammaire sommaire décrivant l'identité d'une personne (adaptée de la Synopsis [S05](#) du langage) :

```
grammar Personne {
  rule nom { Nom '=' (\N+) }
  rule age { Age '=' (\d+) }
  rule adr { Adr '=' (\N+) }
  rule desc {
    <nom> \n
    <age> \n
    <adr> \n
  }
  # etc.
}
```

L'exemple fourni au début de ce chapitre (§ 8.1) donne un exemple de grammaire (sans doute quelque peu simpliste) permettant de reconnaître une URL. Autant il n'était pas vraiment recommandé d'utiliser les expressions régulières de Perl 5 pour reconnaître une URL, autant les grammaires permettent d'effectuer une réelle analyse lexicale et syntaxique du texte analysé pour détecter ou confirmer une URL. Et même s'attaquer à des problèmes bien plus complexes (analyser du HTML, du XML et même du code source informatique).

Les grammaires fonctionnent de façon semblable aux classes. En particulier, les grammaires peuvent hériter des règles d'une autre grammaire, en surcharger d'autres, et ainsi de suite.

```
grammar Courriel {
  rule texte { <debut> $<corps>=<ligne>+? <salutation> }
  rule debut { [Salut|Bonjour|Hi|Hey] $<to>=<S+? '&', '> }
  rule salutation { A plus '<from>=<.+ }
  token ligne { \N* \n }
}
grammar CourrielOfficiel is Courriel {
  rule debut { Cher $<to>=<S+? '<from>=<.+ }
  rule salutation { Bien cordialement '<from>=<.+ }
}
```

8-3 - Motivation

Les regex de Perl ont dans une large mesure révolutionné la partie reconnaissance de texte de la programmation informatique. Au point que de nombreux langages de programmation « concurrents » de Perl se doivent d'annoncer qu'ils font (presque) aussi bien que les expressions régulières de Perl ou qu'ils sont capables d'utiliser la bibliothèque PCRE (Perl Compatible Regular Expressions). Et la réalité est que cet avantage considérable dont disposait Perl sur la plupart des autres langages pour l'analyse textuelle s'est fortement amenuisé ces dernières années.

Par ailleurs, les regex de Perl 5 sont souvent devenues difficiles à lire. Les grammaires vous autorisent et encouragent à diviser vos regex en petits fragments plus faciles à lire (et à utiliser). Les captures nommées permettent aux règles de s'autodocumenter, et beaucoup de choses sont bien plus cohérentes que précédemment.

Surtout, les grammaires de Perl 6 sont si puissantes qu'elles vous permettent de procéder à une analyse lexicale et syntaxique d'à peu près n'importe quel langage de programmation (y compris naturellement Perl 6 lui-même). Ceci rend la grammaire de Perl 6 plus facile à maintenir que celle de Perl 5, écrite en C. Il se peut même que, bientôt, voie le jour un compilateur C utilisant des grammaires Perl, au lieu de *lex* et *yacc* (ou de *flex* et *bison*), ou les équivalents pour d'autres langages) pour compiler du code C ou autres.

La **partie 3** du présent tutoriel approfondira les regex et les grammaires, et fournira notamment un exemple de **grammaire pour analyser du XML** simplifié.

8-4 - Voir aussi

Un article beaucoup plus complet (et en français) sur  **Les regex et grammaires de Perl 6 : une expressivité sans précédent.**

<http://perlcabal.org/syn/S05.html>

<http://perlgeek.de/en/article/mutable-grammar-for-perl-6>

<http://perlgeek.de/en/article/longest-token-matching>

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

<http://laurent-rosenfeld.developpez.com/tutoriels/perl/perl6/approfondissements/#L1>

<http://laurent-rosenfeld.developpez.com/tutoriels/perl/perl6/approfondissements/#L2>

 <http://laurent-rosenfeld.developpez.com/tutoriels/perl/perl6/annexe-02/#L7>

9 - Conclusion de cette première partie du tutoriel

Le présent document aborde les bases de Perl 6. Nous espérons qu'il vous aura donné envie de découvrir ce nouveau langage de programmation et d'aller plus loin.

La suite de ce tutoriel se trouve dans les articles suivants :

De Perl 5 à Perl 6 - Partie 2 : les nouveautés ;

De Perl 5 à Perl 6 - Partie 3 : approfondissements.

Par ailleurs, l'**annexe 1 du présent tutoriel De Perl 5 à Perl 6** décrit plus spécifiquement les changements de syntaxe qu'il va falloir apporter à un programme Perl 5 pour qu'il fonctionne en Perl 6. L'**annexe 2 de ce même tutoriel De Perl 5 à Perl 6** présente de façon détaillée de nombreuses nouveautés du langage Perl 6.

10 - Remerciements

Je remercie **Djibril** et **Claude Leloup** pour leur relecture attentive de ce tutoriel et leurs très utiles suggestions d'amélioration. Je remercie également **grondilu** de m'avoir signalé une erreur dans l'un de mes exemples de code après publication.