De Perl 5 à Perl 6

Partie 3 : Approfondissements

Par Moritz Lenz - Laurent Rosenfeld

Date de publication : 4 août 2015

Dernière mise à jour : 4 octobre 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 et ajouté des points me paraissant intéressants pour mériter pour ma part de supporter intégralement la responsabilité de toute erreur qui aurait pu s'y glisser.

La **première partie de ce texte** examinait surtout les différences de syntaxe de base entre Perl 5 et Perl 6 et les apports de ces différences. La **seconde partie** se penchait plus sur les notions complètement nouvelles de Perl 6. La présente troisième partie approfondit certaines notions importantes (en particulier les regex et les grammaires), introduit quelques autres nouveautés et se termine par un long chapitre sur le Perl 6 idiomatique.

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 - 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- Annexe 1: ce qui change entre Perl 5 et Perl 6
- Les regex et grammaires de Perl 6 : une expressivité sans précédent
- De Perl 6 à Perl 6 - Annexe 2 : les nouveautés de Perl 6

1 - Les regex, le retour

1-1 - Résumé

```
# reconnaissance normale:
if 'abc' ~~ m/../ {
    say $/;                 # ab
}

# reconnaissance avec le modificateur implicite :sigspace
if 'ab cd ef'  ~~ ms/ (..) ** 2 / {
    say $0[1];                  # cd
}

# substitution avec le modificateur :sigspace
my $x = "abc     defg";
$x ~~ ss/c d/x y/;          # ou : $x ~~ s:s:samespace/c d/x y/;
say $x;                     # abx     yefg
```

Nous avons déjà couvert **les bases des regex dans la première partie**, voici quelques éléments complémentaires utiles (mais pas forcément très structurés).

1-1-1 - La reconnaissance

Il n'est pas nécessaire d'écrire des grammaires pour reconnaître des motifs, la syntaxe traditionnelle m/.../ fonctionne toujours, et elle a vu arriver dans sa fratrie une petite sœur, la syntaxe ms/.../, qui implique le modificateur :sigspace permettant (comme vous vous en souvenez sans aucun doute) une reconnaissance littérale des espaces en les remplaçant dans la regex par la règle <.ws> (« ws » pour *whitespace*).

Par défaut, cette règle reconnaît \s+ (un ou plusieurs espaces) si elle est encadrée par des caractères de type mot (ceux qui reconnaissent \w, c'est-à-dire les caractères alphanumériques plus le « _ ») , et \s* (0 ou plusieurs espaces) dans le cas contraire.

Dans les substitutions, le modificateur :samespace fait en sorte que les espaces reconnus avec la règle ws sont préservés. De même, le modificateur :samecase (en abrégé :ii, car c'est une variante de :i) préserve la casse (la distinction capitale/minuscule).

```
my $x = 'Abcd';
$x ~~ s:ii/^../foo/;
say $x;                     # Foocd
$x = 'ABC'
$x ~~ s:ii/^../foo/;
say $x                      # FOO
```

C'est utile si l'on veut renommer globalement le module Toto en Titi, mais les variables d'environnement sont souvent entièrement écrites en lettres capitales. Le modificateur :ii préserve automatiquement la casse.

Cette option copie les informations de casse caractère par caractère. Mais il existe une version plus intelligente : combinée au modificateur :sigspace (ou :s en abrégé), ce modificateur cherche à trouver un motif dans l'information de casse de la chaîne source. Les motifs reconnus sont .lc, .uc, .lc.ucfirst, .uc.lcfirst, et .lc.capitaliz (Str.capitalize met en lettre capitale le premier caractère de chaque mot). Si ce motif est trouvé, il est également appliqué à la chaîne de substitution.

```
my $x = 'The Quick Brown Fox';
$x ~~ s :s :ii /brown.*/perl 6 developer/;
# $x vaut maintenant: 'The Quick Perl 6 Developer'
```

1-1-2 - Utilisations de l'objet reconnu

L'objet reconnu, stocké dans la variable $/, contient des informations non seulement sur ce qui a été reconnu, mais aussi sur ce qui précède et ce qui suit :

```
if 'abcdef' ~~ / de / {
    say ~$/;            # de
    say $/.prematch;    # abc
    say $/.postmatch;   # f
    say $/.from;        # 3
    say $/.to;          # 5
};
```

1-1-3 - Les regex nommées

Les regex nommées permettent de construire des regex complexes avec des briques de construction simples.

Considérons par exemple une regex pour détecter un nombre en virgule flottante. Cela pourrait s'écrire :

```
my regex float {
     <[+-]>?        # signe optionnel
     \d*            # chiffres avant la virgule, optionnel (en anglais)
     '.'            # séparateur décimal (la "virgule")
     \d+            # chiffres après la virgule
     [              # exposant optionnel
        e <[+-]>?  \d+
     ]?
}
```

Mais il peut être préférable de créer des règles pour reconnaître chaque partie de l'ensemble :

```
my token signe { <[+-]> }
my token decimal { \d+ }
my token exposant { 'e' <signe>? <decimal> }
my regex float {
    <signe>?
    <decimal>?
    '.'
    <decimal>
    <exposant>?
}
```

Les règles (*tokens*) ci-dessus peuvent être réutilisées pour définir une regex reconnaissant un entier.

```
my regex entier {
    <signe>?
    <decimal>
}
```

Mais l'avantage se manifeste surtout quand les regex deviennent plus complexes. Par exemple, dans la regex *float*, on peut vouloir rendre la « virgule » et les nombres après la virgule optionnels s'il y a un exposant :

```
my regex float {
    <signe>?
    [
    || <decimal>?  '.' <decimal> <exposant>?
    || <decimal> <exposant>
    ]
}
```

Les grammaires s'appuient fortement sur ce mécanisme de construction de briques élémentaires destinées à être combinées pour former des expressions plus complexes.

1-1-4 - Les alternatives

Les alternatives utilisent le caractère barre verticale « | » comme en Perl 5, mais sa sémantique a changé. Au lieu d'examiner séquentiellement les possibilités et de prendre la première reconnaissance obtenue comme en Perl 5, toutes les possibilités sont examinées en parallèle et c'est le motif reconnu le plus long qui est retenu.

```
'aaaa' ~~ m/ a | aaa | aa /;
say $/;                         # imprime aaa
```

Ce changement peut paraître minime, mais il a des conséquences considérables, et c'est crucial pour créer des grammaires extensibles. Comme l'analyse lexicale et syntaxique d'un programme Perl 6 utilise une grammaire rédigée en Perl 6, ce changement est responsable du fait que, dans l'expression `++$a`, l'opérateur `++` est interprété comme un seul lexème et non comme une répétition du préfixe `+`.

L'ancien style séquentiel de Perl 5 reste disponible avec l'utilisation de ||, par exemple :

```
grammar Math::Expression {
    token value {
        | <number>
        | '('
          <expression>
          [ ')' || { fail("Parenthèse non fermée") } ]
    }

    # ...
}
```

La construction `{ ... }` exécute une fermeture, et le fait d'appeler `fail` dans cette fermeture entraîne l'échec de l'expression. Il est certain que cette branche ne sera appelée que si l'expression précédente (ici le «) ») échoue, si bien que l'on peut s'en servir pour émettre un message d'erreur utile lors de l'analyse lexicale et syntaxique.

Il existe d'autres manières d'écrire des alternatives. Par exemple, si l'on « interpole » un tableau, la recherche portera sur une alternative de ses valeurs :

```
$_ = '12 oranges';
my @fruits = <pomme orange banane kiwi>;
if m:i:s/ (\d+) (@fruits)s? / {
    say "Vous avez $0 $1s, j'en ai { $0 + 2 }. Vous avez perdu.";
}
```

Ce qui imprime :

```
Vous avez 12 oranges, j'en ai 14. Vous avez perdu.
```

Il existe encore une autre construction qui retient automatiquement la plus longue alternative reconnue : les multiregex. On peut les écrire soit sous la forme `multi token nom` ou utiliser un `proto`:

```
grammar Perl {
    ...
    proto token sigil { * }
    token sigil:sym<$> { <sym> }
    token sigil:sym<@> { <sym> }
    token sigil:sym<%> { <sym> }
    ...

   token variable { <sigil> <twigil>? <identifier> }
```

Cet exemple montre des tokens multiples appelés sigil, paramétrés par sym. Quand le nom sigil est utilisé, tous ces tokens sont reconnus dans une alternative. On peut penser que c'est une façon mal commode d'écrire une alternative, mais elle présente un avantage considérable sur la syntaxe '$'|'@'|'%' : elle est facile à étendre :

```
grammar AddASigil is Perl {
    token sigil:sym<!> { <sym> }
}
# waou, nous avons une grammaire Perl 6 avec un sigil supplémentaire!
```

De même, on peut remplacer des alternatives existantes :

```
grammar SigilBizarre is Perl {
    token sigil:sym<$> { '°' }
}
```

Dans cette grammaire, le sigil pour les variables scalaires est « ° », si bien que quand la grammaire recherche un sigil, elle cherche un « ° » au lieu d'un « $ », mais le compilateur sait toujours que c'est la regex sigil:sym<$> qui a été reconnue.

Dans le chapitre suivant, nous verrons le développement d'une vraie grammaire utilisable avec Rakudo.

1-1-5 - Voir aussi

Les regex et grammaires de Perl 6 : une expressivité sans précédent.

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

2 - Une grammaire pour du (pseudo) XML

2-1 - Résumé

```
grammar XML {
    token TOP   { ^ <xml> $ };
    token xml   { <text> [ <tag> <text> ]* };
    token text {  <-[<>&]>* };
    rule tag   {
        '<'(\w+) <attributes>*
        [
            | '/>'                  # a single tag
            | '>'<xml>'</' $0 '>'   # an opening and a closing tag
        ]
    };
    token attributes { \w+ '="' <-["<>]>* '"' };
};
```

2-2 - Description

Cette leçon montre le développement d'une vraie grammaire capable d'analyser du XML de base et fonctionnant sur Rakudo.

Le lecteur qui aurait quelque peu oublié (ou n'aurait pas lu) le sous-chapitre sur les grammaires de la première partie de ce document est invité à en relire la section 8.2.3 consacrée aux **Regex nommées et grammaires** avant de poursuivre.

2-2-1 - Notre version du XML

À cette fin, nous allons adopter une version simple du XML : elle consiste en texte ordinaire et en balises imbriquées qui peuvent éventuellement avoir des attributs. Voici quelques tests pour déterminer ce qui est du XML valide et ce qui n'en est pas.

```
my @tests = (
    [1, 'abc'                          ],       # 1
    [1, '<a></a>'                      ],       # 2
    [1, '..<ab>foo</ab>dd'             ],       # 3
    [1, '<a><b>c</b></a>'              ],       # 4
    [1, '<a href="foo"><b>c</b></a>'],       # 5
    [1, '<a empty="" ><b>c</b></a>' ],       # 6
    [1, '<a><b>c</b><c></c></a>'    ],       # 7
    [0, '<'                            ],       # 8
    [0, '<a>b</b>'                     ],       # 9
    [0, '<a>b</a'                      ],       # 10
    [0, '<a>b</a href="">'             ],       # 11
    [1, '<a/>'                         ],       # 12
    [1, '<a />'                        ],       # 13
);

my $count = 1;
for @tests -> $t {
    my $s = $t[1];
    my $M = XML.parse($s);
    if !($M  xor $t[0]) {
        say "ok $count - '$s'";
    } else {
        say "not ok $count - '$s'";
    }
    $count++;
}
```

C'est une liste de chaînes XML bien formées (celles qui sont associées dans le plan de test à un premier champ égal à 1) et mal formées (celles associées à un 0), et un petit script de test qui exécute ces tests en appelant la fonction XML.parse($string). Par convention, la règle qui reconnaît ce que la grammaire est censée reconnaître s'appelle TOP.

(On constatera à la vue du test 1 que nous n'imposons pas l'existence d'une balise racine unique, mais il serait très simple d'ajouter cette restriction.)

2-2-2 - Développement de la grammaire

L'essence du XML est indubitablement l'imbrication correcte des balises. Commençons donc par le test 2. Ajoutons ceci au début du script de test :

```
grammar XML {
    token TOP   { ^ <tag> $ }
    token tag   {
        '<' (\w+) '>'
        '</' $0   '>'
    }
};
```

Exécutons le script :

```
$ ./perl6 xml-01.pl
not ok 1 - 'abc'
ok 2 - '<a></a>'
not ok 3 - '..<ab>foo</ab>dd'
not ok 4 - '<a><b>c</b></a>'
not ok 5 - '<a href="foo"><b>c</b></a>'
not ok 6 - '<a empty="" ><b>c</b></a>'
```

```
not ok 7 - '<a><b>c</b><c></c></a>'
ok 8 - '<'
ok 9 - '<a>b</b>'
ok 10 - '<a>b</a'
ok 11 - '<a>b</a href="">'
not ok 12 - '<a/>'
not ok 13 - '<a />'
```

Donc, cette simple règle reconnaît correctement une paire balise de début/balise de fin, et rejette à juste titre les quatre échantillons de XML incorrect (tests 8 à 11).

Il devrait être facile de passer avec succès le premier test. Essayons ceci :

```
grammar XML {
   token TOP   { ^ <xml> $ };
   token xml   { <text> | <tag> };
   token text  { <-[<>&]>*  };
   token tag   {
       '<' (\w+) '>'
       '</' $0   '>'
}
};
```

(Souvenez-vous, `<-[...]>` est la négation d'une classe de caractères.)

Et, à l'exécution :

```
$ ./perl6 xml-03.pl
ok 1 - 'abc'
not ok 2 - '<a></a>'
(résultats identiques pour les autres tests)
```

Ah, zut, pourquoi diable le second test échoue-t-il maintenant ? Nous devons ici fournir une explication complémentaire au lecteur : le test figurant ci-dessus a été exécuté lors de la rédaction initiale de ce document à la fin 2008, à un moment où Rakudo n'implémentait pas encore la règle de la reconnaissance la plus longue, mais une règle de la première reconnaissance trouvée (comme Perl 5). Du coup, dans l'alternative `<text> | <tag>` , le programme n'essayait même pas de reconnaître `<tag>` dès lors qu'il avait reconnu `<text>`. Inverser l'ordre de l'alternative pourrait permettre de résoudre le problème.

En fait ce problème est aujourd'hui réglé et Rakudo utilise la règle de la reconnaissance la plus longue depuis (au moins) janvier 2013. Avec une version plus récente de Rakudo (fin 2014 dans le cas ci-dessous), la grammaire donnée ci-dessus fonctionne maintenant correctement :

```
$ ./perl6 xml-03.pl
ok 1 - 'abc'
ok 2 - '<a></a>'
(résultats identiques pour les autres tests)
```

Mais, même si le test ci-dessus est bon, ce n'est pas encore ce qu'il nous faut : nous ne voulons pas reconnaître soit du pur texte soit des balises, mais toutes sortes de combinaisons arbitraires de texte et de balises :

```
token xml   { <text> [ <tag> <text> ]*  };
```

Les deux premiers tests fonctionnent.

Le troisième test, `..<ab>foo</ab>dd`, introduit du texte entre les balises ouvrante et fermante, il nous faut donc autoriser cela. Mais ce n'est pas seulement du texte que l'on peut trouver entre les balises ouvrante et fermante, aussi du XML quelconque. Appelons donc la règle (1) `<XML>` :

```
token tag   {
```

```
    '<' (\w+) '>'
    <xml>
    '</' $0   '>'
}

./perl6 xml-05.pl
ok 1 - 'abc'
ok 2 - '<a></a>'
ok 3 - '..<ab>foo</ab>dd'
ok 4 - '<a><b>c</b></a>'
not ok 5 - '<a href="foo"><b>c</b></a>'
(résultats identiques pour les autres tests)
```

Nous pouvons maintenant nous tourner vers les attributs (les trucs du genre href="foo") :

```
token tag   {
    '<' (\w+) <attribute>* '>'
    <xml>
    '</' $0   '>'
};
token attribute {
    \w+ '="' <-["<>]>* \"
};
```

Mais ceci ne nous permet pas de passer avec succès de nouveaux tests, en raison des espaces entre le nom de la balise et l'attribut. Plutôt que d'ajouter \s+ ou \s* un peu partout, nous allons transformer cette règle de type token en rule (ce qui implique le modificateur :sigspace qui permet de ne pas tenir compte des espaces :

```
rule tag   {
    '<'(\w+) <attribute>* '>'
    <xml>
    '</'$0'>'
};
token attribute {
    \w+ '="' <-["<>]>* \"
};
```

Et là, seuls les deux derniers tests ne sont toujours pas couronnés de succès :

```
ok 1 - 'abc'
ok 2 - '<a></a>'
ok 3 - '..<ab>foo</ab>dd'
ok 4 - '<a><b>c</b></a>'
ok 5 - '<a href="foo"><b>c</b></a>'
ok 6 - '<a empty="" ><b>c</b></a>'
ok 7 - '<a><b>c</b><c></c></a>'
ok 8 - '<'
ok 9 - '<a>b</b>'
ok 10 - '<a>b</a'
ok 11 - '<a>b</a href="">'
not ok 12 - '<a/>'
not ok 13 - '<a />'
```

Ces deux tests contiennent des balises non imbriquées fermées par une simple barre oblique :

```
# ...
    [1, '<a/>'                      ],      # 12
    [1, '<a />'                     ],      # 13
```

Pas de difficulté particulière pour ajouter ce cas de figure à la règle tag :

```
rule tag   {
    '<'(\w+) <attribute>* [
        | '/>'
        | '>' <xml> '</'$0'>'
```

```
        ]
};
```

Maintenant, tous les tests fonctionnent.

Bien sûr, dans un cas réel, la suite de tests devrait être bien plus complète, et nous n'avons examiné qu'un (petit) sous-ensemble de vrai XML. La grammaire présentée reste donc relativement simpliste, mais cela donne une bonne idée de la méthode.

2-2-3 - Aller plus loin dans le hacking

Jouer avec des grammaires est bien plus amusant que lire comment jouer avec elles. Voici donc quelques améliorations que vous pourriez vouloir mettre en œuvre :

- du texte pur contient des entités telles que & ;
- je ne sais pas si les noms des balises XML peuvent commencer par un chiffre, mais la grammaire actuelle (à l'époque de l'écriture de l'article) l'autorise. Vous pourriez vérifier la spécification XML et, si besoin, adopter cette grammaire ;
- du texte pur peut contenir des blocs du genre <![CDATA[...]]> , dans lesquels les balises de type XML sont ignorées et les caractères tels que < sont ignorés et n'ont pas besoin d'un caractère d'échappement ;
- du XML réel autorise des préambules du type <?xml version="0.9" encoding="utf-8"?> nécessitant une balise racine contenant le reste (il faudra peut-être modifier des cas de test) ;
- vous pourriez essayer de mettre en place un bel affichage de code XML en analysant récursivement l'objet $/. (Ce n'est pas du tout trivial, il se peut que vous deviez contourner des bogues Rakudo et peut-être introduire de nouvelles captures.)

De grâce, ne postez pas de réponse à ces défis dans vos commentaires. Laissez aux autres l'occasion de s'amuser autant que vous avez pu le faire vous-même.

2-3 - Voir aussi

Les regex et grammaires de Perl 6 : une expressivité sans précédent

http://laurent-rosenfeld.developpez.com/tutoriels/perl/perl6/annexe-01/#L5

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

3 - Les types « sous-ensembles » (subset)

3-1 - Résumé

```
subset Squares of Real where { .sqrt.Int**2 == $_ };

multi sub square_root(Squares $x --> Int) {
    return $x.sqrt.Int;
}
multi sub square_root(Real $x --> Real) {
    return $x.sqrt;
}
```

3-1-1 - Description

- Les programmeurs Java tendent à considérer qu'un type est soit une classe, soit une interface (c'est-à-dire une classe dépourvue d'une partie de ses moyens), mais cette vision serait très limitée pour Perl 6. Un type

est plus généralement une contrainte sur les valeurs que peut prendre un container (disons, par exemple, une variable, un objet ou l'attribut d'un objet). Une contrainte classique est par exemple : « Ceci est un objet de la classe X ou d'une classe qui hérite de X. » Perl 6 a aussi des contraintes du type : « Cette classe ou cet objet remplit le rôle Y », ou encore : « Ce bout de code renvoie vrai pour les objets globaux au paquetage (objets *our*). Cette dernière forme est la plus générale est s'appelle un type de genre sous-ensemble. Par exemple pour définir un type sous-ensemble Nb_pair des nombres pairs :

```
subset Nb_pair of Int where { $_ % 2 == 0 }
# Nb_pair peut maintenant être utilisé comme n'importe quel autre nom de type

my Nb_pair $x = 2;
my Nb_pair $y = 3; # erreur de type
```

Il est également possible d'utiliser des sous-types anonymes dans les signatures :

```
sub toto (Int where { ... } $x) { ... }
# ou, en plaçant la variable au début:
sub toto ($x of Int where { ... } ) { ... }
```

3-1-2 - Motivation

Autoriser des contraintes de type arbitraire sous la forme de code permet une extensibilité sans limites : si vous n'aimez pas le système de type en vigueur, ou s'il vous manque quelque chose, vous pouvez juste le remplacer ou ajouter ce qui vous manque en créant vos types de genre sous-ensemble.

Cela rend plus facile l'extension de bibliothèques : au lieu de « mourir » sur des donnés qui ne peuvent être gérées, les fonctions ou méthodes peuvent déclarer leurs types de telle façon que les données invalides soient rejetées par le mécanisme de sélection des fonctions de type *multi*. Si nous désirons traiter des données qui étaient rejetées jusqu'à présent, il nous suffit d'ajouter une fonction *multi* ayant le même nom et capable d'accepter ce genre de données. Par exemple, une bibliothèque mathématique traitant des nombres réels peut être étendue de cette façon pour aussi traiter des nombres complexes.

3-1-3 - Voir aussi

http://laurent-rosenfeld.developpez.com/tutoriels/perl/perl6/annexe-02/#L1-3

4 - Citation et analyse lexicale

4-1 - Résumé

```
my @animaux = <chien chat tigre>
# ou
my @animaux = qw/chien chat tigre/;
# ou

my $interface = q{eth0};
my $ips = q :s :x /ifconfig $interface/;

# -----------

sub if {
    warn "if() calls a sub\n";
}
if();
```

4-1-1 - Description

4-1-1-1 - Citation

Perl 6 a un mécanisme puissant de citation de chaînes de caractères, l'utilisateur peut contrôler très précisément les caractéristiques de sa chaîne.

Perl 5 a les guillemets simples (apostrophes), les guillemets (doubles) et l'opérateur de citation qw(...) (guillemets simples, mots séparés par les espaces), ainsi que les opérateurs de citation q(...) et qq(...) qui sont essentiellement des synonymes des guillemets simples et doubles.

Perl 6, pour sa part, définit un opérateur de citation nommé Q, qui peut accepter divers modificateurs placés entre l'opérateur et les éléments à citer. Le modificateur :b (*b* pour *backslash*) permet l'interpolation des séquences d'échappement avec barre oblique inverse, comme par exemple \n. Le modificateur :s permet l'interpolation des variables scalaires. Le modificateur :c permet l'interpolation de fermetures ("1 + 2 = { 1 + 2 }") et ainsi de suite. Le modificateur :w divise les données en entrée en mots, comme qw/.../ en Perl 5.

Il est possible de combiner ces différents modificateurs à son gré. Par exemple, si l'on désire une forme de qw/.../ qui interpole les seules valeurs scalaires et rien d'autre. Aucune difficulté :

```
my $gourmandise = "miel";
my @liste = Q :w :s/lait toast $gourmandise barres\nobliques\tinverses\nbizarres/;
say @liste[*-1];      # imprime : barres\nobliques\tinverses\nbizarres
say @liste[2];        # imprime : miel
```

Voici une liste des modificateurs disponibles reprise de la Synopsis 02. À noter que ces modificateurs ont aussi des noms longs fournis ci-dessous.

Nom court	Nom Long	Signification
:q	:single	Interpole \\, \q et \'
:b	:backslash	Interpole les autres séquences d'échappement comme \n, \t, etc.
:x	:exec	Exécute une commande système et renvoie le résultat
:w	:words	Divise le contenu en mots séparés par des espaces
:ww	:quotewords	Divise le contenu en mots séparés par des espaces, avec protection des contenus entre guillemets
:s	:scalar	Interpole les variables scalaires ($var)
:a	:array	Interpole les variables de tableaux (@var[])
:h	:hash	Interpole les tables de hachage (%var{})
:f	:function	Interpole les appels de fonction (&fonc())
:c	:closure	Interpole les fermetures (blocs de type {code})
:qq	:double	Interpole avec :s, :a, :h, :f, :c, :b
	:regex	Analyse comme une regex

Il existe des formes abrégées qui peuvent faciliter la vie :

q	Q:q
qq	Q:qq
m	Q:regex

Il est également possible d'omettre le premier signe deux-points (:) si le symbole de citation est dans sa forme brève et de l'écrire sous une forme abrégée en un seul mot, ce qui permet de retrouver plus ou moins les opérateurs de citation de Perl 5 :

Symbole	Abréviation de ...
qw	q:w
Qw	Q:w
qx	q:x
Qc	Q:c

Et ainsi de suite.

À noter cependant que qw et qx, par exemple, qui sont équivalents respectivement à q:w et à q:x, n'interpolent pas les variables, puisqu'ils n'ont pas de modificateur permettant l'interpolation des variables (par exemple :s ou :qq). Pour obtenir l'interpolation des variables, il faut utiliser qqw ou qqx :

```
# qqw = qw avec interpolation des variables
my $a = 42;
my @list = qqw{$a b c};
say @list;                          # 42 b c

# qqx = qx avec interpolation des variables
my $world = "tout le monde";
say qqx{echo "Bonjour $world"};  # Bonjour tout le monde
```

Toutefois, il existe une forme qui ne fonctionne pas, et certains programmeurs Perl 5 le regretteront peut-être : il n'est pas possible d'écrire qw(...) avec des parenthèses, parce que Perl l'interprète comme un appel de fonction. Cela dit, il suffit d'ajouter un espace entre le mot-clef qw et la parenthèse pour obtenir le comportement désiré.

4-1-2 - Motivation

Diverses combinaisons des modificateurs d'opérateurs de citation existent en interne, par exemple q:w pour analyser <...>, et :regex pour m/.../. Il semble naturel de mettre ces constructions à la disposition de l'utilisateur, qui y gagne en flexibilité, et peut facilement écrire des macros fournissant un raccourci pour la sémantique de citation voulue.

Et si vous limitez la spécificité des mots-clefs, vous avez bien moins de problèmes de compatibilité ascendante si vous désirez modifier ce que vous considérez comme un mot-clef.

4-1-3 - Voir aussi

http://design.perl6.org/S02.html#Literals

http://laurent-rosenfeld.developpez.com/tutoriels/perl/perl6/annexe-01/#L2-9

5 - Le métaopérateur de réduction

5-1 - Résumé

```
say [+] 1, 2, 3;     # 6
say [+] ();          # 0
say [~] <a b>;       # ab
say [**] 2, 3, 4;    # 2417851639229258349412352 (= 2**(3**4) = 2**81)

[\+] 1, 2, 3, 4      # 1, 3, 6, 10
[\**] 2, 3, 4        # 4, 81, 2417851639229258349412352

if [<=] @list {
    say "Liste en ordre ascendant";
}
```

5-1-1 - Description

Le métaopérateur de réduction [...] peut travailler sur n'importe quel opérateur infixé associatif et le transformer en opérateur de liste. Tout se passe comme si l'opérateur en question était placé entre chaque élément de la liste, si bien que [op] $i1, $i2, @reste renvoie le même résultat que si l'on avait $i1 op $i2 op @reste[0] op @reste[1] ....

Ce métaopérateur assure à peu près le même rôle que la fonction reduce que nous avons décrite **ici** dans notre tutoriel sur la programmation fonctionnelle en Perl, mais avec des fonctionnalités supplémentaires et la possibilité de travailler directement sur un opérateur interne.

C'est une construction extrêmement puissante qui promeut l'opérateur + au rang d'une fonction somme, l'opérateur ~ à celui d'un join (avec des séparateurs vides), et ainsi de suite.

Il est par exemple possible de créer une fonction factorielle comme suit :

```
sub fact(Int $x){
    [*] 1..$x; # NB: espace nécessaire entre '[*]' et liste de valeurs
}
my $c = fact(10);     # 3628800
```

Ce métaopérateur a une certaine parenté avec la fonction List.reduce et, si vous avez au moins un peu pratiqué la programmation fonctionnelle, vous avez sans doute rencontré les fonctions reduce, fold, foldl ou foldr (par exemple en Lisp ou en Haskell). Contrairement à ce qui se passe dans ces langages, le métaopérateur [...] respecte l'associativité de l'opérateur concerné, en sorte que [/] 1, 2, 3 est interprété comme (1 / 2) / 3 (associatif à gauche) , alors que [**] 1, 2, 3 est correctement interprété comme 1 ** (2**3) (associatif à droite).

Comme pour tous les autres métaopérateurs, les espaces blancs sont interdits à l'intérieur des crochets : vous pouvez écrire [+], mais pas [+]. (Cela contribue à lever les ambiguïtés avec les tableaux.)

Comme les opérateurs de comparaison peuvent être chaînés, vous pouvez aussi écrire des choses du style :

```
say "true" if [<] 1, 3, 5, 9;       # "true"

if    [==] @nums { say "Tous les nombres de @nums sont identiques" }
elsif [<]  @nums { say "le tableau @nums est dans un ordre ascendant strict" }
elsif [<=] @nums { say "le tableau @nums est dans un ordre ascendant"}
```

Il n'est cependant pas possible de réduire l'opérateur d'affectation :

```
my @a = 1..3;
[=] @a, 4;   # Ne peut réduire "=" parce que les affectations de listes sont trop délicates
```

5-1-1-1 - Obtenir des résultats partiels

Il existe une forme particulière de cet opérateur qui utilise un antislash comme ceci : [\+]. Elle renvoie les résultats des évaluations partielles. Par exemple, [\+] 1..3 renvoie la liste 1, 1+2, 1+3, ce qui donne bien sûr 1, 3, 6.

Par exemple, en utilisant cette forme du métaopérateur de réduction avec l'opérateur de concaténation :

```
[\~] 'a' .. 'd'      # <a ab abc abcd>
```

Comme les opérateurs associatifs à droite s'évaluent de droite à gauche, on obtient aussi les résultats partiels dans cet ordre :

```
[\**] 1..3;          # 3, 2**3, 1**(2**3), soit 3, 8, 1
```

Il est possible d'enchaîner plusieurs opérateurs de réduction (qui s'exécuteront alors de droite à gauche) :

```
[~] [\**] 1..3;      # "381"
```

Ou même :

```
[\~] [\**] 1..3;      # "3 38 381"
```

5-1-2 - Motivation

Les programmeurs sont paresseux et ne veulent pas écrire une boucle juste pour appliquer un opérateur binaire à tous les éléments d'une liste. La méthode List.reduce de Perl 6 fait à peu près la même chose, mais sa syntaxe n'est pas aussi concise que le métaopérateur de réduction : [+] @list devrait s'écrire : @list.reduce(&infix:<+>). En outre, avec reduce, il faut s'occuper soi-même de l'associativité de l'opérateur, alors que le métaopérateur s'en charge tout seul.

Si vous n'êtes pas convaincu, jouez un peu avec lui (sous Rakudo, par exemple), c'est assez amusant.

5-1-3 - Voir aussi

http://design.perl6.org/S03.html#Reduction_operators, **http://www.perlmonks.org/?node_id=716497**

http://laurent-rosenfeld.developpez.com/tutoriels/perl/perl6/les-nouveautes/#L4-4-1

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

6 - Le métaopérateur X (croix ou cross)

6-1 - Résumé

```
for <a b> X 1..3 -> $a, $b {
    print "$a: $b   ";
}
# imprime: a: 1  a: 2  a: 3  b: 1  b: 2  b: 3

.say for <a b c> X 1, 2;
# imprime: a 1 a 2 b 1 b 2 c 1 c 2
# (avec chaque élément à la ligne)
```

6-2 - Description

Le métaopérateur croix X renvoie le produit cartésien de deux ou de plusieurs listes, c'est-à-dire qu'il renvoie tous les tuples possibles dans lesquels le premier élément est un élément de la première liste, le second élément un élément de la seconde liste, et ainsi de suite.

Voici un exemple avec trois listes :

```
for <a b> X 1,2 X <x y> -> $x, $y, $z {
    say "$x, $y, $z";
}
```

Ce qui imprime :

```
a, 1, x
a, 1, y
a, 2, x
a, 2, y
b, 1, x
b, 1, y
b, 2, x
b, 2, y
```

Si le métaopérateur X est suivi d'un opérateur infixé, alors cet opérateur est appliqué à tous les éléments des tuples et ce sont les résultats de cette opération sur chaque élément qui sont renvoyés :

```
say <a b> X~ 1,2;       #  'a1', 'a2', 'b1', 'b2'

say 1, 2 X+ 3, 4, 20;   #  1+3, 1+4, 1+20, 2+3, etc., soit 4 5 21 5 6 22
```

6-2-1 - Motivation

Il est assez fréquent de devoir itérer sur toutes les combinaisons possibles de deux ou plusieurs listes, et l'opérateur croix permet de condenser cette opération en une seule itération, ce qui simplifie la syntaxe et réduit le niveau d'indentation.

L'utilisation de ce métaopérateur permet parfois d'éliminer complètement les boucles.

6-2-2 - Voir aussi

http://design.perl6.org/S03.html#Cross_operators

http://laurent-rosenfeld.developpez.com/tutoriels/perl/perl6/les-nouveautes/#L4-4-3

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

7 - Exceptions et exceptions de contrôle

7-1 - Résumé

```
try {
    die "Oh non";

    CATCH {
        say "Il y a eu une erreur: $!";      # $! vaut "Oh non"
    }
```

```
}
```

7-2 - Description

Contrairement à ce que leur nom pourrait laisser entendre, les exceptions n'ont rien d'exceptionnel. En fait, elles font partie intégrante du flot d'exécution normal des programmes en Perl 6.

Les exceptions sont générées soit à la suite d'erreurs implicites (par exemple l'invocation d'une méthode inexistante, l'utilisation d'une variable non initialisée ou un contrôle de type renvoyant une erreur), soit explicitement en appelant die ou d'autres fonctions.

Quand une exception se produit, le programme recherche des blocs CATCH ou try dans la chaîne des appels de fonctions, remontant toute la pile (donc, sortant de force de toutes les fonctions appelées jusqu'ici). Si l'on ne trouve aucun CATCH ou try, le programme se termine et affiche, espère-t-on, un message d'erreur utile. Dans le cas contraire, le message d'erreur est stocké dans la variable spéciale $! et le bloc CATCH s'exécute (dans le cas d'un try sans bloc CATCH, le bloc try renvoie Any).

Jusqu'à présent, les exceptions peuvent toujours paraître exceptionnelles, mais la gestion des erreurs est une partie intégrante de toute application non triviale. Bien plus, même de simples instructions return ou next peuvent générer une exception !

C'est ce que l'on appelle des *exceptions de contrôle*, et il est possible de les gérer avec des blocs CONTROL ou lors de chaque déclaration de fonction.

Considérons cet exemple :

```
use v6;

sub s {
    my $block = -> { return "block"; say "Toujours ici" };
    $block();
    return "sub";
}

say s();    # block
```

Ici, le « bloc » renvoie une exception de contrôle de flux, ce qui entraîne non seulement la sortie du bloc courant (et, donc, la non-impression du message « Toujours ici »), mais aussi la sortie de la fonction, où elle est gérée par la déclaration sub s... . La valeur de retour, ici une chaîne de caractères, est renvoyée à l'appelant est imprimée à l'écran par l'instruction say de la dernière ligne.

En ajoutant un bloc CONTROL {…} dans la portée dans laquelle le $block est appelé, celui-ci va gérer l'exception de contrôle.

Contrairement à ce qui se passe avec d'autres langages de programmation, les blocs CATCH/CONTROL sont dans la portée dans laquelle l'erreur est détectée (et non à l'extérieur), ce qui donne accès complet aux variables lexicales concernées et permet à la fois de générer des messages d'erreurs utiles et d'empêcher des blocs DESTROY de s'exécuter avant que l'erreur ne soit traitée.

7-2-1 - Exceptions non générées

Perl 6 supporte l'exécution multitâche et, en particulier, la parallélisation automatique. Pour s'assurer que l'ensemble des *threads* (ou fils d'exécution) ne souffrent pas de la défaillance d'un seul *thread*, une forme d'exception « douce » a été inventée.

Quand une fonction appelle fail($obj), elle renvoie une forme spéciale de undef, qui contient la charge utile $obj (habituellement un message d'erreur) et la trace arrière (nom du fichier et numéro de ligne). Traiter cette valeur indéfinie spéciale sans vérifier si elle est indéfinie entraîne la génération normale d'une exception.

```
my @files = </etc/passwd /etc/shadow inexistant>;
my @handles = hyper map { open($_) }, @files; # Implémentation de hyper non finalisée
```

Dans cet exemple (théorique), l'opérateur hyper dit à map de paralléliser ses actions dans toute la mesure du possible. Quand l'ouverture du fichier inexistant échoue, une instruction usuelle du genre die "No such file or directory" arrêterait l'exécution de toutes les autres opérations d'ouverture de fichiers. Mais comme l'échec de l'ouverture d'un fichier appelle l'instruction fails "No such file or directory" (au lieu de « die »), cela donne à l'appelant la possibilité de vérifier le contenu de @handles et il a toujours accès à l'intégralité du message d'erreur.

Si vous n'aimez pas les exceptions douces, vous pouvez utiliser le pragma use fatal; au début du programme pour faire en sorte que toutes les exceptions de type fail soient générées immédiatement.

7-3 - Motivation

Un bon langage de programmation a besoin d'exceptions pour gérer les conditions d'erreur. Toujours vérifier les valeurs de retour est une tâche pénible et s'oublie facilement.

Comme les exceptions traditionnelles peuvent empoisonner le parallélisme implicite, il fallait une solution qui combine le meilleur des deux mondes : ne pas tuer tout sur-le-champ, et ne pas perdre non plus d'information.

8 - Perl 6 idiomatique

8-1 - Résumé

```
# Créer un hachage à partir d'une liste de clefs et de valeurs :
# solution 1: tranches
my %hash; %hash{@keys} = @values;
# solution 2: métaopérateurs
my %hash = @keys Z=> @values;

# Créer un hachage à partir d'un tableau, en
# affectant une valeur vraie à chaque élément du tableau
my %exists = @keys X=> True;

# limiter une variable à une fourchette de valeurs (0 à 10)
my $x = -2;
$x = 0 max $x min 10;    # $x vaut maintenant 0

# pour le débogage: affiche le contenu d'une variable
# ainsi que son nom sur STDERR
note :$x.perl;

# tri non sensible à la casse
say @list.sort: *.lc;

# attributs obligatoires
class Quelquechose {
    has $.required = die "Attribut 'required' est obligatoire";
}
Quelquechose.new(required => 2); # pas d'erreur
Quelquechose.new()               # BOUM
```

8-2 - Description

Apprendre les spécifications d'un langage ne suffit pas pour devenir productif avec lui. On a besoin de savoir comment résoudre des problèmes spécifiques. Les constructions communément utilisées, ou idiomes, permettent de ne pas devoir réinventer la roue chaque fois que l'on rencontre un problème.

Voici quelques idiomes communs de Perl 6 relatifs pour l'essentiel aux structures de données.

8-2-1 - Hachages

```
# Créer un hachage à partir d'une liste de clefs et de valeurs :
# solution 1: tranches
my %hash; %hash{@keys} = @values;
# solution 2: métaopérateurs
my %hash = @keys Z=> @values;
```

La première solution est la même qu'en Perl 5 : affectation à une tranche. La seconde solution utilise l'opérateur Z (*zip,* fermeture éclair) qui associe deux listes en intercalant les éléments de chaque liste comme une fermeture éclair :

```
my @c = 1, 2, 3 Z 10, 20, 30;   # 1, 10, 2, 20, 3, 30
```

La construction Z=> est un métaopérateur qui combine zip avec => (l'opérateur de construction de paires). Si bien que 1, 2, 3 Z=> 10, 20, 30 est évalué à : 1 => 10, 2 => 20, 3 => 30. L'affectation de cette liste à un hachage transforme le tout en un hachage.

Pour les tests d'existence, les valeurs du hachage n'ont généralement pas d'importance, pourvu qu'elles soient évaluées à Vrai dans un contexte booléen. Dans ce cas, voici une jolie façon d'initialiser un hachage à partir d'un tableau ou d'une liste de clefs :

```
my %exists = @keys X=> True;
```

qui utilise le métaopérateur croix pour affecter la valeur True à chaque élément du tableau @keys.

8-2-2 - Les nombres

Parfois, on a besoin d'un nombre appartenant à une fourchette prédéfinie (par exemple pour s'en servir d'indice pour un tableau).

En Perl 5, on se retrouve souvent à faire quelque chose du genre : $a = $b > $haut? $haut : $b, et une autre condition du même type pour la limite basse.

En Perl 6, l'utilisation des opérateurs infixés max et min simplifient considérablement la syntaxe :

```
my $in-range = $bas max $x min $haut;
```

parce que $bas max $x renvoie le plus grand des deux nombres, assurant que le bas de la fourchette est respecté (et de même avec min pour le haut de la fourchette).

Comme min et max sont des opérateurs infixés, on peut également écrire :

```
my $x = 15;
$x max= 0;
$x min= 10; # $x vaut maintenant 10
```

8-2-3 - Débogage

En Perl 5, le module Data::Dumper permet de visualiser les structures de données et objets complexes. En Perl 6, la méthode .perl fait de même. Dans les deux cas, on génère du code qui reproduit la structure de données d'origine aussi fidèlement que possible.

:$var génère une paire utilisant le nom de la variable (sans le sigil) comme clef et sa valeur comme valeur :

```
my $var = 10;
say :$var.perl;     # affiche: "var" => 10
```

La fonction note écrit sur la sortie en erreur (STDERR) en ajoutant un retour à la ligne. Si bien que note :$var.perl est une bonne façon d'afficher le nom et la valeur d'une variable à des fins de débogage.

8-2-4 - Tri

Comme en Perl 5, la fonction sort prend en paramètre une fonction qui compare deux valeurs, puis assure le tri selon le résultat de cette comparaison. Par rapport à Perl 5, le sort de Perl 6 est un peu plus malin et effectue la transformation pour vous à condition que la fonction en question n'accepte qu'un seul argument.

En général, l'on désire comparer des valeurs transformées, on peut faire ce qui suit en Perl 5 :

Sort - code Perl 5
```
# ATTENTION: code Perl 5
my @sorted = sort { transform($a) cmp transform($b) } @values;

# ou la transformation de Schwartz:
my @sorted = map { $_->[1] }
             sort { $a->[0] cmp $b->[0] }
             map { [transform($_), $_] }
             @values
```

La première solution nécessite de taper deux fois le nom de la fonction de transformation et, surtout, appelle cette fonction deux fois pour chaque comparaison, ce qui peut avoir un impact négatif sur les performances. La seconde solution évite cela en stockant la valeur transformée avec la valeur d'origine, mais c'est pas mal de code à taper.

Perl 6 automatise la seconde solution (et est un peu plus efficace que la transformation de Schwartz naïve en évitant de créer un tableau pour chaque valeur) quand la fonction de transformation a une *arité* de un (c'est-à-dire n'accepte qu'un seul argument, l'arité étant le nombre d'arguments d'une fonction) :

```
my @sorted = sort &transform, @values;
```

8-2-5 - Attributs obligatoires

La manière classique de garantir la présence d'un attribut est de vérifier sa présence dans le constructeur - ou dans tous les constructeurs s'il y en a plusieurs.

Cela fonctionne en Perl 6, mais il est plus facile et plus sûr de rendre la présence obligatoire au niveau de chaque attribut :

```
has $.attr = die "'attr' is mandatory";
```

Ce code exploite le mécanisme de la valeur par défaut. Si une valeur est fournie, le code pour générer la valeur par défaut ne s'exécute pas et la fonction die n'est pas appelée. Si en revanche le constructeur omet de préciser l'attribut, alors une exception est déclenchée.

8-2-6 - Idiomes Perl 5 et idiomes Perl 6

Dans la plupart des exemples ci-dessous, le code montrera :

1. Du code Perl 5 non idiomatique ;
2. Une version Perl 5 plus idiomatique ;
3. Une version Perl 6 traduisant naïvement la version Perl 5 idiomatique ;
4. Une (ou plusieurs) version(s) Perl 6 idiomatique(s).

8-2-6-1 - Itérer sur les indices d'un tableau

```
# Perl 5
for ( my $i=0; $i < @array; $i++ ) {...}
for my $i ( 0 .. $#array ) {...}
# Perl 6
for 0 .. @array.end -> $i {...}
for @array.keys -> $i {...}
```

8-2-6-2 - Choisir un élément aléatoire dans un tableau

```
# Perl 5
$z = $array[ int(rand scalar(@array)) ];
$z = $array[ rand @array ];
# Perl 6
$z = @array[ rand*@array ];
$z = @array.pick;
```

8-2-6-3 - Division entière

```
# Perl 5
$c = ( ($x - ($x % 3) ) / 3 );
$c = int( $x / 3 );
# Perl 6
$c = Int( $x / 3 );
$c = $x div 3;            # opérateur de division entière
```

8-2-6-4 - Imprimer le nombre d'éléments d'un tableau

```
# Perl 5
say scalar @array;
say 0+@array;
# Perl 6
say 0+@array;       # Comme en Perl 5
say +@array;        # + impose un nouveau contexte numérique
say @array.elems;   # la méthode .elems est plus explicite
```

8-2-6-5 - Faire quelque chose une fois sur cinq

```
# Perl 5
if ( ($x/5) == int($x/5) ) {...}
if ( !($x % 5) ) {...}
# Perl 6
if !($x % 5) {...}
if $x %% 5 {...}           # %% signifie "est divisible par"
```

8-2-6-6 - Faire quelque chose n fois (en comptant jusqu'à n-1)

```
# Perl 5
```

```
for ( $_=0; $_ < $n; $_++ ) {...}
for ( 0 .. ($n-1) ) {...}
# Perl 6
for 0 ..^ $n {...}
for ^$n {...}                  # ^9 signifie 0 ..^ 9, soit 0..8
```

8-2-6-7 - Diviser une chaîne de caractères en mots (sur l'espace)

Les appels de méthode « nus » (sans objet préfixant le .) s'appliquent *toujours* à la variable $_ (ce qui élimine la confusion, en Perl 5, sur les fonctions qui travaillent par défaut sur $_ et celles qui ne le font pas).

```
# Perl 5
@words = split /\s+/, $_;
@words = split;     # Comportement par défaut utile, mais peu intuitif
# Perl 6
@words = .split(/\s+/); # split() n'a plus de comportement par défaut
@words = .words;        # La nouvelle méthode .word remplace l'ancien
                        # comportement par défaut de split en Perl 5
```

8-2-6-8 - Diviser une chaîne en une liste de caractères

```
# Perl 5
@chars = map { substr $word, $_, 1 } 0..length($word);
@chars = split '', $word; # Split sur rien
# Perl 6
@chars = $word.split('');
@chars = $word.comb;
```

Remarque : la fonction ou méthode comb prend habituellement comme arguments une regex ou une sous-chaîne, une chaîne de caractères, et optionnellement un nombre maximal de reconnaissances, et renvoie une liste de sous-chaînes reconnues :

```
"tototititatato".comb(/to/).perl;    # affiche ("to", "to", "to")
```

Si aucune regex ou sous-chaîne n'est spécifiée, comb renvoie par défaut une liste des caractères de la chaîne :

```
"tototiti".comb.perl;     # ("t", "o", "t", "o", "t", "i", "t", "i")
```

8-2-6-9 - Boucle infinie

```
# Perl 5
for (;;) {...}     # Prononcé avec un fort accent C
while (1) {...}
# Perl 6
while 1 {...}
loop {...}         # sans limite, loop forme une boucle infinie
```

8-2-6-10 - Renvoyer les éléments uniques d'une liste

```
# Perl 5
my %s, @r; for @a { push @r, $_ if !$s{$_}; $s{$_}++; }; return @r;
my %s; return grep { !$s{$_}++ } @a;     # ou List::MoreUtils::uniq
# Perl 6
my %s; return grep { !%s{$_}++ }, @a;    # ne pas oublier la virgule
return @a.unique;
```

8-2-6-11 - Somme des éléments d'une liste

```
# Perl 5
my $sum = 0; for my $num (@a) { $sum += $num }
my $sum; $sum += $_ for @a;        # ou List::Util::sum
# Perl 6
my $sum = @a.reduce(*+*);
my $sum = [+] @a;                  # [op] applique op à toute la liste
```

8-2-6-12 - Listes et opérations sur les listes

Parfois, les idiomes de Perl 5 se retrouvent tels quels (ou presque) en Perl 6

```
# Perl 6
@alpha = 'A' .. 'Z';
@a = qw{ alpha bravo charlie }; # Mais il y a aussi: @a = < alpha ...>
%meta = ( foo => 'bar', baz => 'quz' );
@squares = map { $_ * $_ }, @a;           # Noter la virgule après map
@commence_par_un_chiffre = grep { /^\d/ }, @a;        # idem après grep
```

8-2-6-13 - Traiter chaque ligne de STDIN ou d'une liste de fichiers de ligne de commande

```
# Perl 5
for my $file (@ARGV) {
    open my $FH, "<", $file or die "$!";
    while (<$FH>) {...}
}
while (<>) {...}          # Le filehandle vide est magique

# Perl 6
for $*ARGFILES.lines {...}
for lines() {...}         # lines() traite par défaut $fh = $*ARGFILES
```

Remarque : la méthode ou la fonction lines renvoie une liste *paresseuse* de lignes, si bien qu'il n'y a aucun inconvénient d'occupation mémoire à l'utiliser dans une boucle for comme ce serait le cas en Perl 5.

8-2-6-14 - Initialisation d'un hachage avec une constante

```
# Perl 5
my %h;   for (@a) { $h{$_} = 1 }
my %h = map { $_ => 1 } @a;
# Perl 6
my %h = map { $_ => 1 }, @a;
my %h = @a X=> 1;
```

8-2-6-15 - Initialisation d'un hachage avec une énumération

```
# Perl 5
my %h;   for (0..$#a) { $h{ $a[$_] } = $_ }
my $c;   my %h = map { $_ => ++$c } @a;
# Perl 6
my $c;   my %h = map { $_ => ++$c }, @a;
my %h = @a Z=> 1..*;
my %h = @a.pairs».invert;  # si l'on commence à 0
```

Remarque : la méthode invert intervertit clefs et valeurs d'une énumération.

8-2-6-16 - Initialisation d'un hachage à partir de tableaux en parallèle

```
# Perl 5
my %h;   for (@a) { $h{$_} = shift @b }
my %h;   @h{@a} = @b;
# Perl 6
my %h;   %h{@a} = @b;
my %h = @a Z=> @b;
```

8-2-6-17 - Intervertir deux variables

```
# Perl 5
my $temp = $x; $x = $y; $y = $temp;
( $x, $y ) = ( $y, $x );
# Perl 6
( $x, $y ) =   $y, $x;
( $x, $y ) .= reverse;  # .= permet à reverse de modifier la liste sur place
# Marche aussi très bien sur les tableaux:   @a[ $j, $k ] .= reverse;
```

8-2-6-18 - Rotation d'un élément d'un tableau

```
# Perl 5
my $temp = shift @a; push @a, $temp;
push @a, shift @a;
# Perl 6
@a.push: @a.shift;
@a .= rotate;
```

8-2-6-19 - Créer un objet

```
my $chien = new Animal;
my $chien = Animal->new;
#
my $chien = Animal.new;
my Animal $chien .= new;  # $chien est toujours un Animal;
                          # le compilateur peut optimiser!
```

8-2-6-20 - Générer trois exemplaires des éléments supérieurs à 5

Combiner une transformation et une sélection est un idiome relativement avancé de Perl 5. Les nouvelles valeurs renvoyées par if offrent une syntaxe concise.

```
# Perl 5
@z = map { ($_) x 3 } grep { $_ > 5 } @y;     # map et grep
@z = map { $_ > 5 ? ($_) x 3 : () } @y;       # map simulant aussi grep
# Perl 6
@z = map { $_ > 5 ?? ($_) xx 3 !! Nil }, @y;
@z = @y.map: { $_ xx 3 if $_ > 5 };           # !if == liste vide
@z = ($_ xx 3 if $_ > 5 for @y);              # Liste en compréhension
```

La dernière forme ci-dessus est une *liste en compréhension*, un concept populaire dans les langages fonctionnels (une liste en compréhension est une liste construite en filtrant les éléments d'une autre liste).

8-2-6-21 - Générer des entiers aléatoires compris entre 3 et 7

```
# Perl 5
do { $z = int rand 8 } until $z >= 3;
$z = 3 + int rand 5;
# Perl 6
```

```
$z = 3 + Int(5.rand);
$z = (3..7).pick;
```

8-2-6-22 - Compter trois par trois dans une boucle infinie

```
# Perl 5
for ( my $i = 1; ; $i++ ) { my $n = 3 * $i; ... }
for ( my $n = 3; ; $n += 3 ) {...}
#
loop ( my $n = 3; ; $n += 3 ) {...}
for 3, * + 3 ... * -> $n {...}      # ... est l'opérateur "séquence"
for 3, 6, 9 ... * -> $n {...}       # ... devine à partir de l'exemple
```

8-2-6-23 - Boucler sur une fourchette de valeurs, sans les bornes

```
# Perl 5
for my $i ($start .. $end) {next if $i == $start or $i == $end; ... }
for my $i ( ($start+1) .. ($end-1) ) {...}
# Perl 6
for ($start+1) .. ($end-1) -> $i {...}
for $start ^..^ $end -> $i {...}
```

9 - Conclusion

J'espère, cher lecteur, que ce tutoriel en trois parties vous aura ouvert l'appétit pour cette nouvelle version de Perl et vous aura convaincu du gain d'expressivité qu'elle permet. Quelques points importants n'ont été que peu ou pas du tout abordés (le parallélisme, les coroutines, les macros, l'Unicode, etc.), souvent parce que leur implémentation n'est pas encore complètement figée (ou ne l'a été que très récemment), mais le tour d'horizon est néanmoins assez complet et permet déjà d'envisager de faire beaucoup de choses.

Il sera complété avec des annexes sous la forme de référence de poche ou, si l'on préfère, d'« antisèche », c'est-à-dire un résumé assez concis de tous les éléments de la syntaxe de Perl abordés dans les trois parties de ce tutoriel, afin de pouvoir retrouver rapidement tel ou tel point qui serait sorti momentanément de la mémoire. La première de ces annexes, mise en ligne début octobre 2015, couvre **Ce qui change entre Perl 5 et Perl 6**. La seconde traite plus spécifiquement **des nouveautés de Perl 6**.

En février 2015 (à la conférence Fosdem à Bruxelles), Larry Wall a annoncé la sortie d'une version de production de Perl 6 avant la fin de l'année, plus précisément avant Noël. Ce sera à mes yeux un très beau cadeau et j'attends avec impatience ce moment.

10 - Remerciements

Je remercie à nouveau Moritz Lenz de m'avoir aimablement donné l'autorisation de réaliser cette adaptation en français de son travail.

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

1 : Nous appliquons ici la convention de la **première partie de ce document** et utilisons dans ce contexte le mot *règle* pour désigner une regex nommée quelconque, quel que soit le mot-clef (`regex`, `rule` ou `token`) utilisé pour l'introduire.