

De Perl 5 à Perl 6 - Annexe 2

Les nouveautés de Perl 6



Par Laurent Rosenfeld 

Date de publication : 24 décembre 2015

Dernière mise à jour : 17 juin 2018

DÉBUTANT

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

Cette série d'articles décrivait en détail de nombreuses différences et de nombreuses nouveautés de Perl 6. L'**Annexe 1** récapitulait les différences entre Perl 5 et Perl 6. Nous voulons dans la présente annexe résumer les nouveaux éléments syntaxiques et sémantiques afin de constituer une référence de poche (incomplète), une sorte d'« antisèche » permettant au lecteur de retrouver rapidement un élément de syntaxe qui lui échapperait.

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

En complément sur Developpez.com

- De Perl 5 à Perl 6 - Partie 2 : les nouveautés, un tutoriel de Moritz Lenz et Laurent Rosenfeld
- De Perl 5 à Perl 6 - Partie 1 : les bases du langage, un tutoriel de Moritz Lenz et Laurent Rosenfeld
- De Perl 5 à Perl 6 - Partie 3 : Approfondissements

- [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 puissance expressive sans précédent](#)
- [Objets, classes et rôles en Perl 6 - Tutoriel de programmation orientée objet](#)

1 - Les types.....	6
1-1 - L'intérêt des types.....	6
1-2 - Types les plus courants.....	7
1-3 - Types sous-ensembles.....	9
1-4 - Quelques types particuliers et utiles.....	10
1-4-1 - Les paires et énumérations.....	10
1-4-1-1 - Les paires.....	10
1-4-1-2 - Les énumérations.....	11
1-4-1-3 - Une énumération booléenne.....	12
1-4-2 - Les jonctions.....	12
1-4-2-1 - Les différents types de jonctions.....	12
1-4-2-2 - Quelques utilisations typiques des jonctions.....	13
1-4-2-3 - Précautions avec les opérateurs négatifs.....	13
1-4-3 - Collections (ensembles, sacs et assortiments).....	14
1-4-3-1 - Opérateurs renvoyant une valeur booléenne.....	15
1-4-3-2 - Opérateurs renvoyant un ensemble ou un sac.....	16
1-5 - Voir aussi.....	16
2 - Structures de contrôle.....	16
2-1 - Voir aussi.....	18
3 - Fonctions et signatures.....	18
3-1 - Paramètres et signatures.....	18
3-2 - Paramètres nommés.....	18
3-3 - Paramètres optionnels.....	19
3-4 - Fonctions multiples.....	20
3-5 - Voir aussi.....	21
4 - Programmation fonctionnelle en Perl 6.....	21
4-1 - Nouveaux opérateurs de listes.....	21
4-1-1 - Programmation par listes ou par flux de données.....	25
4-1-2 - Nouvelles fonctions génériques abstraites.....	26
4-2 - Les métaopérateurs.....	27
4-2-1 - Le métaopérateur de réduction.....	27
4-2-2 - Le métaopérateur croix.....	28
4-2-3 - Les hyperopérateurs « et ».....	29
5 - Créer de nouveaux opérateurs.....	30
5-1 - Qu'est-ce qu'un opérateur et comment en créer un ?.....	30
5-1-1 - Précédence.....	31
5-1-2 - Associativité.....	32
5-1-3 - Notation postcirconfixée et circonfixée.....	32
5-1-4 - Surcharger les opérateurs existants.....	32
5-1-5 - Un exemple plus complet.....	33
6 - Programmation orientée objet.....	34
6-1 - Utiliser des objets.....	35
6-1-1 - Objets-types.....	35
6-2 - Classes.....	36
6-2-1 - Attributs.....	36
6-2-2 - Méthodes.....	37
6-2-3 - Objet self.....	38
6-2-4 - Méthodes privées.....	38
6-2-5 - Subméthodes.....	39
6-2-6 - Héritage.....	39
6-2-7 - Construction d'objet.....	40
6-2-8 - Clonage d'objets.....	40
6-3 - Les rôles.....	41
6-3-1 - Application de rôles.....	41
6-3-2 - Promotion automatique des rôles (punning).....	42
6-4 - Documentation complémentaire sur les objets en Perl 6.....	43
7 - Les regex et les grammaires.....	43
7-1 - Les regex de Perl 6.....	43

7-1-1 - Conventions lexicales.....	43
7-1-2 - Littéraux.....	44
7-1-3 - Métacaractères et classes de caractères.....	44
7-1-3-1 - Caractère d'échappement et classes de caractères prédéfinies.....	45
7-1-3-2 - Propriétés Unicode.....	45
7-1-3-3 - Classes de caractères énumérées et intervalles.....	46
7-1-4 - Quantificateurs.....	46
7-1-4-1 - Avidité et frugalité des quantificateurs.....	47
7-1-5 - Alternatives (reconnaître ceci ou cela).....	47
7-1-6 - Ancres.....	48
7-1-7 - Regroupements et captures.....	48
7-1-7-1 - Regroupements.....	48
7-1-7-2 - Captures.....	49
7-1-7-3 - Regroupements sans capture.....	49
7-1-7-4 - Captures nommées.....	49
7-1-8 - Sous-règles ou règles nommées.....	50
7-1-9 - Adverbes.....	50
7-1-9-1 - Adverbes de regex.....	51
7-1-9-1-1 - L'adverbe « ratchet » (pas de retour arrière).....	51
7-1-9-1-2 - L'adverbe sigspace (espaces blancs significatifs).....	52
7-1-9-2 - Les adverbes de reconnaissance.....	52
7-1-10 - Regarder devant et derrière (assertions).....	52
7-1-10-1 - Assertions avant.....	53
7-1-10-2 - Assertions arrière.....	53
7-2 - Les grammaires.....	53
7-2-1 - Les « briques » de construction d'une grammaire.....	53
7-2-2 - Créer une grammaire.....	54
7-2-2-1 - Syntaxe de définition d'une grammaire.....	54
7-2-2-2 - Héritage de grammaires.....	55
7-2-3 - Utiliser une grammaire.....	55
7-2-4 - Les classes et objets d'actions.....	55
7-2-4-1 - Exécuter du code lors d'une reconnaissance.....	55
7-2-4-2 - Autres façons d'exécuter du code dans une grammaire.....	56
7-2-5 - Un exemple simple de grammaire : validation de noms de modules.....	56
7-2-5-1 - La grammaire de validation.....	57
7-2-5-2 - Ajout d'un objet d'actions.....	58
7-2-5-3 - Autres exemples de grammaires.....	58
7-2-6 - Héritage, grammaires mutables et perspectives.....	59
8 - Multitâche, parallélisme, concurrence et programmation asynchrone.....	59
8-1 - Interface de haut niveau.....	59
8-1-1 - Les promesses (objets de type « Promise »).....	59
8-1-2 - Les fournisseurs (objets de type « Supply »).....	61
8-1-3 - Les canaux (objets de type « Channel »).....	61
8-1-4 - Processus asynchrones (objets de type « Proc::Asyn »).....	62
8-2 - Interface de bas niveau.....	62
8-2-1 - Les threads.....	62
8-2-2 - Ordonnanceurs (objets ayant le rôle « Scheduler »).....	63
8-2-3 - Verrous (objets de type « Lock »).....	63
8-3 - Parallélisme, asynchronisme et concurrence.....	64
8-3-1 - Parallélisme.....	64
8-3-1-1 - Promesses et parallélisme de tâches.....	64
8-3-1-2 - Parallélisme de données.....	65
8-3-1-3 - Méthodes race et hyper pour paralléliser le pipeline.....	65
8-3-2 - Asynchronisme.....	66
8-3-2-1 - Le module Proc::Async.....	66
8-3-2-2 - Flux de données asynchrone multiples : les fournisseurs.....	67
8-3-2-3 - Le bloc react.....	68
8-3-3 - La concurrence.....	69

8-3-3-1 - Utilisation d'un surveillant (monitor).....	69
8-3-3-2 - Utilisation d'un acteur (actor).....	70
9 - Conclusion.....	70
10 - Remerciements.....	71

1 - Les types

1-1 - L'intérêt des types

Perl 6 est typé, mais n'oblige personne à déclarer systématiquement le type de sa variable (les variables peuvent avoir des contraintes de typage, mais ce n'est pas indispensable). D'une certaine façon, tout est objet et a un type. Déclarer le type de ses variables ou objets permet de bénéficier de l'aide du compilateur qui fera un bon nombre de vérifications à notre place et détectera éventuellement pour nous certaines de nos erreurs que nous pourrions ainsi éviter et corriger.

Cela permet aussi d'utiliser les nombreuses méthodes associées aux types ainsi que les opérateurs surchargés par les classes et rôles correspondant à ces types. Par exemple, le type Date permet les opérations suivantes (parmi de nombreuses autres) :

```
#!/usr/bin/perl6
use v6;          # utilisation de Perl 6

my $d = Date.new(2012, 12, 24); # Réveillon de Noël
say $d;                # 2012-12-24
say $d.year;           # 2012
say $d.month;          # 12
say $d.day;            # 24
say $d.day-of-week;    # 1 (donc, lundi)
my $n = Date.new('2012-12-31'); # Réveillon de la Saint-Sylvestre
say $n - $d;           # 7
say $n + 1;            # 2013-01-01
```

On voit que la manipulation des dates est grandement facilitée.

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

```
'a string'          # Str (chaîne de caractères)
'Wall' => 'Larry'    # Pair (une paire)
2                   # Int (nombre entier, précision arbitraire)
7/3                 # Rat (nombre rationnel, une paire contenant
                    # un numérateur et un dénominateur)
3.14                # Rat (nombre rationnel, ici paire 314 et 100)
sqrt(2)             # Num (nombre à virgule flottante)
3 + 2i              # Complex (nombre complexe)
True                # Bool (booléen), de même que False
{ say "Bonjour" }   # Block (bloc de code ayant sa portée lexicale)
<1 2 3>.any          # Junction (superposition logique de valeurs)
41, 4, 45, 7        # List (liste séquentielle de valeurs)
/ ^ ab /            # Regex
```

Chacun de ces types correspond soit à une classe soit à un rôle, et permet d'utiliser un grand nombre de méthodes (correspondant au type ou éventuellement héritées) ou d'opérateurs surchargés facilitant grandement le travail, comme le montre l'exemple du type Date ci-dessus, et ce, souvent sans même avoir besoin d'utiliser une syntaxe orientée objet particulière pour la déclaration : ainsi, dans l'exemple de nombre complexe ci-dessus, il suffit de déclarer une variable \$z avec la valeur 3 + 2i pour construire l'objet nombre complexe, c'est équivalent à écrire \$z = Complex.new(3, 2), et c'est tout de même plus direct et plus parlant.

De même :

```
my $val = 5;
say $val.WHAT;          # (Int) -> Perl 6 a reconnu un Int
say $val.is-prime;      # 5 nombre premier? - imprime True
say is-prime($val);     # idem, True
say ($val+1).is-prime;  # imprime False
say is-prime($val + 1); # idem, False
```

On peut restreindre le type de valeurs que peut contenir une variable (ou un groupe de variables) en ajoutant un nom de type à sa déclaration.

```
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..."
my Str @b;          # tableau de chaînes de caractères
```

De même, les types sont utilisés dans les signatures des fonctions (voir le chapitre **Fonctions et signatures** du tutoriel et le paragraphe **3 Fonctions et signatures** du présent document) et même du programme principal si on utilise une fonction MAIN (voir le chapitre sur **la fonction MAIN** du tutoriel). Les types permettent également une validation des arguments reçus par une fonction et la détermination de la fonction voulue dans le cas de fonctions multiples ou multi subs (voir le chapitre **Créer ses propres opérateurs** du tutoriel et le paragraphe **5 Créer de nouveaux opérateurs** du présent document).

Pour connaître le type d'un objet :

```
say "foo".WHAT; # donne le type exact de "foo": -> Str() ou (Str)

# Pour savoir si la valeur est compatible avec un type donné
# (même par le biais d'un héritage):
if $x ~~ Int {
    say 'La variable $x contient un entier';
}
```

1-2 - Types les plus courants

Voici les types les plus couramment employés (même si c'est pour certains souvent implicitement).

Type	Classe ou rôle	Nature
Array	Classe	Tableau : suite énumérée de valeurs
Bag	Classe	Ensemble immuable d'objets distincts ayant une pondération entière (§ 1.4.3)
BagHash	Classe	Ensemble variable d'objets distincts ayant une pondération entière (§ 1.4.3)
Blob	Rôle	Buffer immuable pour objet binaire (<i>Binary Large Object</i>)
Block	Classe	Objet de code ayant sa propre portée lexicale
Bool	Classe	Booléen logique (deux valeurs possibles : <i>True</i> et <i>False</i>)
Buf	Rôle	Buffer variable pour objet binaire
Complex	Classe	Nombre complexe
Date	Classe	Date calendaire
DateTime	Classe	Date calendaire avec l'heure (Int), minute (Int) et seconde (éventuellement fractionnaire)
Duration	Classe	Durée, longueur d'intervalle de temps
Enum	Classe	Paire immuable de clef-valeur
Exception	Classe	Événement anormal susceptible d'interrompre le flot du programme
FatRat	Classe	Nombre rationnel de précision arbitraire (composé d'un numérateur et d'un dénominateur entiers chacun de précision arbitraire)
Grammar	Classe	Groupe de regex nommées formant une grammaire formelle (§ 7.2 Les grammaires)
Hash	Classe	Hachage ou table de hachage (parfois table associative), table de correspondance entre des chaînes et des valeurs énumérées
IO	Rôle	Objet associé aux entrées-sorties
IO::Handle	Classe	Fichier ou flux de données ouvert
Int	Classe	Entier (de précision arbitraire)
Iterator	Classe	Générateur d'une suite de valeurs
Junction	Classe	Superposition logique de valeurs (§ 1.4.2 Les jonctions)
List	Classe	Suite de valeurs
Macro	Classe	Sous-routine exécutée à la compilation
Match	Classe	Résultat de la reconnaissance d'une regex (§ 7.1 Les regex de Perl 6)
Method	Classe	Fonction membre d'une classe (§ 6.2 Classes)
Mix	Classe	Ensemble immuable d'objets distincts ayant une pondération réelle (§ 1.4.3)
MixHash	Classe	Ensemble variable d'objets distincts ayant une pondération réelle (§ 1.4.3)
Num	Classe	Nombre en virgule flottante
Numeric	Rôle	Nombre ou objet pouvant agir comme un nombre
Pair	Classe	Paire clef-valeur (1.4.1.1 Les paires)
Parameter	Classe	Élément d'une signature
Parcel	Classe	Séquence immuable de valeurs. Pourrait disparaître prochainement en faveur du type List dans le cadre d'une remise à plat de la modélisation des tableaux et listes.
Range	Classe	Intervalle de valeurs ordonnées (en principe consécutives)
Rat	Classe	Nombre rationnel, paire numérateur-dénominateur (numérateur en précision arbitraire, dénominateur en précision limitée)
Real	Rôle	Nombre non complexe
Regex	Classe	Regex, motif de reconnaissance d'une chaîne de caractères
Routine	Classe	Objet de code ayant sa propre portée lexicale et une gestion des valeurs de retour

Set	Classe	Ensemble immuable d'objets distincts et uniques (§ 1.4.3)
SetHash	Classe	Ensemble variable d'objets distincts et uniques (§ 1.4.3)
Signature	Classe	Motif de la liste des paramètres d'une fonction
Stash	Classe	Tables des symboles déclarés avec <i>our</i> .
Str	Classe	Chaîne de caractères
Sub	Classe	Subroutine (routine), fonction
Thread	Classe	Exécution en parallèle de code de bas niveau (§ 8.2.1 Les threads)
X::...	Classe	Ensemble de plus de 100 types relatifs aux erreurs et exceptions

1-3 - Types sous-ensembles

Un type n'est pas simplement soit une classe, soit un rôle, c'est plus généralement une contrainte sur les valeurs que peut prendre un *container* (par exemple une variable, un objet ou l'attribut d'un objet).

On peut créer ses propres « sous-types », ou plus exactement des types de genre sous-ensemble (*subsets*), spécialisant les types existants.

Par exemple, au sein des entiers naturels, on peut définir le sous-ensemble `Nb_pair` des nombres pairs à l'aide des mots-clefs `subset ... of TypeX where` :

```
subset Nb_pair of Int where { $_ %% 2 } # ou : ... where { $_ % 2 == 0 }
# Nb_pair peut maintenant être utilisé comme un autre nom de type

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

Le type sous-ensemble hérite des caractéristiques (opérateurs, routines, méthodes, contraintes) du type dont il est le sous-ensemble, mais il s'y ajoute les contraintes propres à sa définition.

De même, il est possible de définir des sous-ensembles de nos propres classes. Soit par exemple une classe `Personne` et des instances de cette classe d'âges variés :

```
class Personne {
    has Int $.age;
    has Str $.nom;
}

my $toto = Personne.new(age => 14, nom => "Toto");
my $titi = Personne.new(age => 25, nom => "Titi");
my $tutu = Personne.new(age => 70, nom => "Tutu");
```

On peut dès lors définir des sous-ensembles classes d'âges :

```
subset Enfant of Personne where *.age < 16;
subset Adulte of Personne where *.age >= 16 and *.age < 66;
subset Senior of Personne where *.age >= 66;
```

Cela fait, nous pouvons maintenant utiliser ces sous-ensembles pour la répartition des fonctions ou méthodes multiples ou multi (voir [Créer ses propres opérateurs](#) et § [3.4](#) ci-dessous) selon les « sous-types » :

```
multi affiche_nom(Enfant $personne) { say "Le petit {$personne.nom}" }
multi affiche_nom(Adulte $personne) { say $personne.nom }
multi affiche_nom(Senior $personne) { say "Le sage {$personne.nom}" }

affiche_nom($toto);      # -> Le petit Toto
affiche_nom($titi);      # -> Titi
affiche_nom($tutu);      # -> Le sage Tutu
```

Les types sous-ensembles ne sont pas des classes

À noter que nous avons des types sous-ensembles très utiles en termes de contraintes et de signatures de fonctions ou méthodes, **Multi** ou non, mais les sous-types **Enfant**, **Adulte** et **Senior** n'accèdent pas pour autant au rang de classes ou de sous-classes. S'il était tout à fait possible et naturel ci-dessus de déclarer une variable entière paire à l'aide du type sous-ensemble **Nb_paire**, il n'est en revanche pas possible de déclarer un objet à l'aide d'un type sous-ensemble comme ceci :

Erreur

```
my $tata = Enfant.new(age => 12, nom => "Tata"); # ERRONÉ
```

La classe s'appelle bien **Personne**, et non **Enfant**, et c'est bien ce nom de classe qu'il faut utiliser pour la déclaration d'une de ses instances.

Mais il reste possible d'ajouter le typage (et la vérification de type associée) à la déclaration de l'objet :

```
my Enfant $tata = Personne.new(age=> 12, nom => "Tata");
```

On peut enfin utiliser des sous-types anonymes dans les signatures des fonctions :

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

1-4 - Quelques types particuliers et utiles

Les types numériques (Num, Rat, Int, FatRat, Real, Complex) sont intuitivement faciles à comprendre, de même que des types scalaires tels que Str, Bool, etc. ou également certains types plus complexes comme Date ou DateTime.

Certains types complexes comme Regex ou Grammar seront abordés dans des chapitres dédiés plus détaillés.

Seuls quelques types composites un peu inhabituels qu'il est utile de connaître seront abordés ici.

1-4-1 - Les paires et énumérations

Les paires et énumérations sont toutes deux des paires de clef-valeur.

Le type **Pair** représente un type variable (modifiable) de paire alors que le type **Enum** représente un type immuable (constante) de paire.

1-4-1-1 - Les paires

Il existe plusieurs formes syntaxiques pour construire une paire :

```
my $x = 'clef' => 'valeur'; # mot-clef Pair non requis  
my $y = :clef<valeur>; # idem, crée la paire 'clef' => 'valeur'  
  
# Raccourci créant une paire à partir d'une variable et de sa valeur :  
my $le_nombre = 42;  
my $z = :$le_nombre; # crée la paire $z : le_nombre => 42  
  
# Autres notations abrégées:
```

```
my $bool_vrai =:vrai;      # crée la paire vrai => True
my $bool_faux =:!faux;     # crée la paire faux => False
```

Le type `Pair` est surtout utilisé pour trois choses :

- en Perl 6, un hachage est en fait une liste de paires (donc une liste de clefs-valeurs) :

```
my %bool_fr = ($bool_vrai, $bool_faux); # crée le hachage:
                                           # faux => False, vrai => True
```

- les paires servent aussi à désigner les arguments nommés des fonctions (voir § 3.2) :

```
sub ttc(Real :$prix-unitaire, Int :$quantité, Rat :$tva) {
    return $prix-unitaire * $quantité * (1 + $tva);
}
say "prix = ", ttc(prix-unitaire => 7.00, quantité => 5, tva => 0.20);
# affiche: prix = 42
```

- la notation par paire sert également pour les opérateurs de test de fichiers :

```
say "Vrai" if "bin".path ~~ :e; # -> Vrai (le répertoire bin existe)
# fonctionne aussi directement avec la variable par défaut $_ :
given "bin".path {
    say "Vrai" if :e;           # -> Vrai
}
```

1-4-1-2 - Les énumérations

Techniquement, le type `enum` représente une (seule) paire immuable clef-valeur. Cependant, par extension, le mot désigne aussi informellement une liste de paires immuables clef-valeur. Les énumérations permettent d'employer un ensemble de symboles pour représenter un ensemble de valeurs. Chaque association d'une énumération est une paire de constantes déclarée avec le mot-clé `enum` et de type `Enum`. Chaque `enum` associe une clef d'`enum` avec une valeur d'`enum`. Sémantiquement, une `enum` au sens large est donc en quelque sorte un hachage de constantes.

On peut par exemple définir une énumération des mois de l'année :

```
my enum mois (jan => "01", fév => "02", mar => "03",
              avr => "04", mai => "05", jun => "06",
              jui => "07", aoû => "08", sep => "09",
              oct => "10", nov => "11", déc => "12");
say mois.enums; # imprime aoû => 08, avr => 04, déc => 12, ...

for mois.enums -> $x { say $x.kv} # ou: say .kv for mois.enums;
avr 04
jan 01
nov 11
jui 07
oct 10
sep 09
fév 02
mar 03
jun 06
aoû 08
déc 12
mai 05
```

Si les clefs sont spécifiées sans les valeurs, ces dernières prennent automatiquement les valeurs 0, 1, 2, etc.

Par exemple, la ligne de code suivante :

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

est essentiellement du sucre syntaxique pour :

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

1-4-1-3 - Une énumération booléenne

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

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

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

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

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

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

Voir aussi le chapitre sur les **énumérations** du tutoriel.

1-4-2 - Les jonctions

Les *jonctions* sont des valeurs composites superposant une ou plusieurs valeurs non ordonnées (elles ont été initialement été appelées *superpositions*, ce mot peut encore être trouvé dans des documents un peu anciens). Les opérations sur les jonctions s'exécutent séparément sur tous les éléments de la jonction (peut-être même en parallèle dans des *threads* différents si le compilateur et la machine virtuelle le supportent) et une nouvelle jonction du même type est assemblée à partir des résultats partiels.

1-4-2-1 - Les différents types de jonctions

Il y a quatre types de jonctions, qui ne diffèrent que quand elles sont évaluées en contexte booléen. Les types disponibles sont `any`, `all`, `one` et `none`. En contexte booléen :

- une jonction de type `any` retournera une valeur vraie si *l'une au moins des valeurs* de la jonction est vraie ;
- une jonction de type `all` renverra vrai si *toutes les valeurs* de la jonction sont vraies ;
- une jonction de type `one` sera vraie si *une (et une seule) valeur* est vraie ;
- une jonction de type `none` sera évaluée à vraie si *aucune des valeurs* n'est vraie.

En Perl 6, les opérateurs `|`, `^` et `&` ne sont pas des opérateurs booléens binaires (comme en Perl 5 ou en C), mais des opérateurs infixés de jonctions ou même des constructeurs de jonctions :

Type	Opérateur infixé
any	
one	^
all	&

1 | 2 | 3 est la même chose que any(1..3) , any(1, 2, 3) ou any<1 2 3>.

```
my $x = 3|4|6;           # $x est une jonction de type any.
                        # Équivalent à $x = any(3, 4, 6)
say "Vrai" if $x == 4;    # -> Vrai (4 est l'une des valeurs possibles)
my $y = any(4, 7, 11);   # Peut aussi s'écrire $y = 4|7|11
say "Vrai" if $x == $y;   # -> Vrai : 4 est commun aux deux jonctions
```

1-4-2-2 - Quelques utilisations typiques des jonctions

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

```
say "$_ est un nombre de Fibonacci"
for grep { $_ == any(<1 2 3 5 8 13>) }, 7..20;
# Affiche : "... est un nombre de Fibonacci" pour 8 et 13
```

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

Code Perl 5

```
say "$_ est un nombre de Fibonacci"
for grep { $_ == 1 or $_ == 2 or $_ == 3 or $_ == 5
or $_ == 8 or $_ == 13 } 1..20;
```

Une jonction de type **all** permet de vérifier aisément une propriété pour la totalité d'un groupe de valeurs :

```
say "Chiffres impairs" if 1 & 3 & 5 & 8 %2; # Faux, n'affiche rien
say "Chiffres impairs" if 1 & 3 & 5 & 7 %2; # -> Chiffres impairs
```

Une jonction de type **any** permet de vérifier si une valeur donnée est présente dans un tableau :

```
my @tableau = <1 3 5 7 >;
say "Trouvé 7" if any(@tableau) == 7; # -> Trouvé 7
say "Trouvé 8" if any(@tableau) == 8; # -> N'affiche rien
```

Dans ce cas, le compilateur *peut en principe* décider de court-circuiter les évaluations successives dès qu'un élément renvoyant une valeur vraie a été trouvé, ce qui rend la recherche optimale (arrêt de la recherche dès que l'élément recherché a été trouvé). L'implémentation de Perl 6 testée (2015-06) ne semble cependant pas le faire.

En dehors d'un contexte booléen, une opération s'applique à tous les éléments d'une jonction et retourne une jonction de même type :

```
say ((2|3|4)+7)          # équivalent à 9|10|11 ou any(9, 10, 11)
```

1-4-2-3 - Précautions avec les opérateurs négatifs

L'utilisation de jonctions avec des opérateurs de comparaison négatifs pourrait être quelque peu déroutante, d'une certaine façon analogue aux difficultés d'interprétation posées par les doubles (ou triples) négations.

Considérons l'exemple suivant :

```
my @tableau = <a b c d e>;
say "Présent" if 'd' eq any(@tableau); # -> Présent, pas de difficulté
say "Absent" if 'd' ne any(@tableau); # -> Comment interpréter ?
```


Si l'on compare la lettre « d » au premier élément du tableau, celui-ci n'est pas égal à « d », et la comparaison *pourrait* logiquement s'arrêter là et renvoyer vrai et, donc, afficher « Absent ».

Ce comportement a été jugé trop contre-intuitif, car la lettre « d » est bien un élément du tableau. Par convention, il a été décidé que, dans un tel cas, l'expression `$a ne $b` est réécrite en interne `not($a eq $b)` par le compilateur et, plus généralement, qu'une expression `$a !op $b` est réécrite `!($a op $b)`. Du coup, l'expression `'d' ne any(@tableau)` est interprétée comme `not('d' eq any(@tableau))` et la sémantique redevient claire : l'expression entre parenthèses renvoie une valeur vraie (« d » est présent dans le tableau), et sa négation une valeur fausse (« d » n'est pas absent) :

```
my @tableau = <a b c d e>;
say "Absent" if 'd' ne any @tableau; # n'affiche rien
# équivalent à :
say "Absent" if not ('d' eq any @tableau); # n'affiche rien
# ou à :
say "Absent" unless 'd' eq any @tableau; # n'affiche rien
```

Pour éviter de se poser des questions, il peut être préférable d'éviter les opérateurs de comparaison négatifs avec les jonctions et de préférer explicitement soit les deux dernières formulations équivalentes de code ci-dessus, soit un opérateur de comparaison positif associé à une jonction négative :

```
say "Absent" if 'd' eq none @tableau; # n'affiche rien
say "Absent" if 'g' eq none @tableau; # -> Absent
```

Voir aussi le chapitre sur les  **jonctions** du tutoriel.

1-4-3 - Collections (ensembles, sacs et assortiments)

Les ensembles, sacs et mélanges sont des collections non ordonnées d'objets uniques et pondérés. Le mot « collection » sera utilisé ici informellement pour les désigner collectivement, sans que ce terme ait un sens officiellement défini en Perl 6.

On a parfois besoin de rassembler des objets dans un contenant sans que l'ordre ait de l'importance. Perl 6 fournit à cette fin six types de collections non ordonnées : `Set` et `SetHash` (ensembles), `Bag` et `BagHash` (sacs), `Mix` et `MixHash` (assortiments). Comme l'ordre est sans importance, ces collections permettent des recherches plus rapides que dans des listes ordonnées, à la manière des tables de hachage.

Si l'on désire une liste d'éléments sans doublons, on peut utiliser les types `Set` (ensemble immuable, une sorte de constante) ou `SetHash` (ensemble mutable ou variable). Pour obtenir une liste sans doublons, mais conservant l'ordre de la liste, consulter la routine (nous utilisons le mot *routine* pour décrire une *subroutine* admettant à la fois une syntaxe de fonction et une syntaxe de méthode) `unique` du type `List`.

Si l'on désire enregistrer le nombre de fois où chaque élément est présent, on peut utiliser un sac (`Bag` ou un `BagHash`). Dans ces sacs, chaque élément est associé à un poids (un décompte sous la forme d'un entier non signé). Les types `Mix` et `MixHash` sont semblables, sauf que leur pondération peut être fractionnaire.

Les types `Set`, `Bag` et `Mix` sont immuables (ce sont des constantes). Il faut utiliser leurs variantes mutables (variables), `SetHash`, `BagHash`, et `MixHash`, si l'on désire pouvoir ajouter ou retirer des éléments après la construction de la collection.

Ces six types de collections possèdent la même sémantique et partagent dans une large mesure les mêmes opérateurs spécifiques.

D'une part, des objets identiques réfèrent au même élément, dont l'identité est déterminée en employant la méthode WHICH (de la même manière que l'opérateur `===` vérifie l'identité). Pour des valeurs de type Str, cela signifie avoir la même valeur. Pour des références vers par exemple des tableaux, cela signifie référer à la même instance d'objet.

D'autre part, ils proposent tous une interface de genre table hachage dans laquelle les éléments de la collection (qui peuvent être des objets de type quelconque) sont les « clefs » et les pondérations associées les « valeurs » :

Type de \$a	Valeur de \$a{\$b} si \$b est un élément	Valeur de \$a{\$b} si \$b n'est pas un élément
Ensemble	True	False
Sac	Un entier strictement positif	0
Assortiment	Un nombre réel non nul	0

Ces types de genre collection offrent toute une série d'opérations ensemblistes communes, comme les unions, intersections, différences symétriques, etc., ainsi que les opérateurs booléens d'appartenance, d'inclusion, etc.

Ces opérateurs peuvent être écrits avec le caractère UTF-8 représentant le symbole mathématique correspondant (comme # ou #) ou avec une version ASCII correspondante : (elem) ou (|), les parenthèses font partie intégrante de l'opérateur.

La plupart du temps, il n'est pas nécessaire d'utiliser explicitement des objets de type collection pour utiliser ces opérateurs ensemblistes, qui fonctionneront sur tout objet dont les arguments sont de type Any (par exemple, les listes, tableaux, etc.), la coercition dans le type désiré se fera au moment voulu.

```
my @tableau = <a b c d e>;      # un simple tableau, pas un Set
say 'c appartient à @tableau' if 'c' # @tableau; # -> Vrai
say "Présent" if 'c' (elem) @tableau;      # idem: -> Présent
```

1-4-3-1 - Opérateurs renvoyant une valeur booléenne

Les opérateurs infixés suivants renvoient une valeur booléenne :

Symbole	ASCII	UTF8	Signification
#	(elem)	\x2208	Appartient à (l'élément à gauche appartient à la collection à droite).
#	!(elem)	\x2209	N'appartient pas à.
#	(cont)	\x220B	Contient (la collection à gauche contient l'élément à droite).
#	!(cont)	\x220C	Ne contient pas.
#	(<=)	\x2286	Est inclus dans ou égal (est un sous-ensemble).
#	!(<=)	\x2288	N'est pas inclus ni égal.
#	(<)	\x2282	Est strictement inclus (est un sous-ensemble strict).
#	!(<)	\x2284	N'est pas strictement inclus.
#	(>=)	\x2287	Est un sur-ensemble (tous les éléments de la collection de droite appartiennent à celle de gauche).
#	!(>=)	\x2289	N'est pas un sur-ensemble.
#	(<)	\x2283	Est un sur-ensemble strict.
#	!(<)	\x2285	N'est pas un sur-ensemble strict.
#	(<+)	\x227C	Est un sous-ensemble, et tous les éléments de la collection de droite sont pondérés au moins autant que leurs homologues de gauche.
#	(>+)	\x227D	Est un sur-ensemble, et aucun élément de droite n'est pondéré plus que son homologue de gauche.

1-4-3-2 - Opérateurs renvoyant un ensemble ou un sac

Les opérateurs infixés suivants renvoient une collection (ensemble, sac ou assortiment). Dans le tableau ci-dessous, les mots « sac » ou « de type sac » (objets ayant le rôle *baggy*) sont employés au sens large et désignent aussi bien les sacs que les assortiments (types Set, SetHash, Mix et MixHash), autrement dit tout type collection ayant une pondération.

Dans certains cas, ces opérateurs ne se comporteront pas exactement de la même manière pour un objet de type Set ou de type Bag (il faudra par exemple additionner les pondérations pour combiner deux collections).

Symbole	ASCII	UTF8	Signification
#	()	\x222A	Union ensembliste. Si l'un des opérandes est de type sac, renvoie un nouveau sac dont chaque élément est pondéré avec le poids le plus fort observé pour cet élément dans les opérandes.
∩	(&)	\x2209	Intersection ensembliste. Si l'un des opérandes est de type sac, renvoie un nouveau sac dont chaque élément est pondéré avec le poids <i>commun</i> le plus fort observé dans les opérandes (autrement, dit, le poids le plus faible observé pour chaque élément).
#	(-)	\x2216	Différence ensembliste (ou complément) : tous les éléments de la collection de gauche qui ne sont pas dans la ou les collection(s) de droite. Si l'opérande de gauche est de type sac, renvoie un nouveau sac dont chaque élément est pondéré avec un poids égal à son poids moins le poids cumulé du même élément dans la ou les collection(s) de droite.
#	(^)	\x2296	Différence ensembliste symétrique (tous les éléments qui sont dans la collection de gauche et pas celle de droite, plus ceux qui sont dans celle de droite et pas celle de gauche. <code>\$a # \$b</code> est équivalent à <code>(\$a # \$b) # (\$b # \$a)</code>).
#	(.)	\x228D	Multiplication de sac : renvoie un sac dont chaque élément est pondéré avec le produit des poids des éléments homologues.
#	(+)	\x228E	Addition de sac : renvoie un sac dont chaque élément est pondéré avec la somme des poids des éléments homologues.

1-5 - Voir aussi

Dans l'introduction initiale de la première partie de ce tutoriel, rédigée à l'été 2014 (mais en partie reprise d'articles plus anciens), nous écrivions que Perl était sous-documenté. En un an, la situation a beaucoup changé et il y a maintenant en anglais une documentation abondante, sans doute pas encore vraiment complète, mais déjà très riche et utilisable.

En ce qui concerne les types, un élément réellement essentiel du langage puisque le type d'une variable ou d'un objet détermine les méthodes ou fonctions que l'on pourra lui appliquer, la documentation est particulièrement dense : voir <http://doc.perl6.org/type.html>.

2 - Structures de contrôle

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-clés if, while, for, etc. et qu'il vaut

même mieux ne pas en mettre (ou, si l'on en met, bien veiller à laisser un espace entre le mot-clef et la parenthèse de la condition, voir **Espaces blancs**).

Les branchements conditionnels sont pratiquement inchangés. Il est toutefois possible en Perl 6 de chaîner les opérateurs sans répéter les variables, comme en mathématiques :

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

La boucle `for` est maintenant utilisée exclusivement pour itérer sur des listes (et la boucle synonyme `foreach` a disparu en Perl 6). Elle utilise par défaut la variable `$_` topicalisée, sauf si une variable de boucle explicite est fournie :

```
for 1..10 {
    say $_; # utilise la variable par défaut $_ comme en Perl 5
}

for 1..10 -> $x {
    say $x; # La variable d'itération est $x ("bloc pointu").
}
```

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.

Il est cependant 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
```

On peut également utiliser plusieurs variables de boucles :

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

À chaque itération, les variables de boucle consommeront chacune un élément de la liste, et la boucle ci-dessus affichera :

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

La boucle `for` « de style C » de Perl 5 a changé de nom et s'appelle désormais `loop` (et c'est la seule construction de boucle nécessitant encore des parenthèses). Ce style de boucle est généralement à utiliser avec parcimonie, dans des cas bien particuliers sur la condition d'arrêt de la boucle ou la façon dont est modifiée la variable de boucle :

```
loop (my $x = 2; $x < 100; $x = $x**2) {
    say $x;
}
# imprime 2, 4, 16
```

Il existe une exception toutefois aux réserves formulées contre l'utilisation de la boucle `loop`, c'est une façon idiomatique de faire une boucle infinie :

```
loop {
```

```
# code de la boucle infinie
}
```

2-1 - Voir aussi

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

3 - Fonctions et signatures

Il reste possible d'utiliser comme en Perl 5 le tableau `@_` pour récupérer les arguments positionnels passés à une fonction, ce qui peut largement suffire pour une fonction simple :

```
sub multiplie {
    my ($x, $y) = @_;
    return $x * $y;
}
```

Mais l'utilisation de `@_` ne fonctionne pas (et c'est logique) si la fonction utilise des signatures (Perl 6 vous reprochera d'essayer de passer outre les signatures et la compilation échouera).

3-1 - Paramètres et signatures

Les fonctions (*subroutines*) sont déclarées avec le mot-clef `sub` et peuvent avoir une signature comportant une liste de paramètres formels, lesquels 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 à des « *traits* » (propriétés définies au moment de la compilation) tels que `is rw` (lecture et écriture, comme lors d'un passage par référence) ou `is copy` (copie locale à la fonction) :

```
sub essaie-de-modifier($toto) {
    # Les traits d'union sont permis dans le nom des identifiants
    # à condition d'être suivis d'une lettre
    $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
```

3-2 - Paramètres nommés

Plutôt que devoir se souvenir de l'ordre des paramètres dans une liste de paramètres positionnels, il est parfois préférable de donner un nom à chaque paramètre (surtout si la liste des paramètres est un peu longue). C'est ce que permettent les paramètres nommés de Perl 6 :

```
my $r = Rectangle.new(
    x      => 100,
    y      => 200,
    hauteur => 23,
    largeur => 42,
    couleur => 'black'
);
```

Pour définir un paramètre nommé dans la signature, 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
```

Cette notation crée en fait automatiquement des paires clef-valeur et cette notation s'appelle syntaxe de paire à deux-points (*colon pair syntax*). L'ordre des paramètres n'a plus d'importance, c'est la correspondance entre le nom du paramètre d'appel et celui de la signature qui établit le lien.

Voir aussi le chapitre **Paramètres nommés** du tutoriel et le chapitre sur les paires (§ 1.4.1.1) du présent document.

3-3 - 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. On dit que la fonction (ou sa liste des paramètres) peut être *variadique*. Autrement dit, il est possible d'utiliser des paramètres optionnels (parfois appelés « *slurpy parameters* », ou paramètres « gobe tout »).

On peut rendre certains paramètres optionnels (généralement les derniers de la liste) en ajoutant un point d'interrogation à leur suite ou en fournissant une valeur par défaut :

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

# Arguments par défaut
sub logarithme($nombre, $base = e) { # e est une constante prédéfinie
                                     # en Perl 6: 2,7182845905...
    return log($nombre) / log($base)
}
say logarithme(4);                 # Second argument par défaut: log népérien
                                   # -> 1.38629436111989
say logarithme(32, 2);             # Second argument explicite: log en base 2
                                   # -> 5
say logarithme(100, 10);           # log en base 10. -> imprime 2
```

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

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

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

```
sub commande-repas($plat-du-jour, *%extras) {
    say "Je désire le $plat-du-jour, mais avec des modifications:";
    say %extras.keys.join(', ');
}
commande-repas('steak', :oignons, :bien-cuit);
```

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

```
sub affiche($scalaire1, @liste, $scalaire2) {
    say "@liste[1], $scalaire2";
}
my @liste = "toto", "titi", "tutu";
affiche(1, @liste, 2); # imprime "titi, 2"
```

Mais il en résulte *a contrario* qu'il n'est par défaut plus possible d'utiliser un tableau comme une liste d'arguments :

```
my @mois = <jan fév mar avr mai jun jui>;
my @val = 3, 2; # deux éléments à partir du quatrième
say splice(@mois, 3, 2); # Comportement correct -> "avr mai"

@mois = <jan fév mar avr mai jun jui>;
say splice(@mois, @val); # ERRONÉ : "mar avr mai jun jui"
```

Le premier argument de la méthode `splice` est censé être un `Int`, et la coercition fait que `splice` reçoit en définitive un seul argument, le nombre d'éléments du tableau `@val` et la fonction retourne tous les éléments à partir du troisième. Il est cependant possible d'obtenir le comportement recherché en préfixant le tableau `@val` de l'hyperopérateur (préfixé « `|` »), qui aplatis les listes, tableaux, paires, énumérations, hachages, etc. en une liste d'arguments :

```
my @mois = <jan fév mar avr mai jun jui>;
say splice(@mois, |@val); # OK : "avr mai"
```

Voir aussi le chapitre sur les **Paramètres optionnels** du tutoriel.

3-4 - Fonctions multiples

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

```
multi subs
multi sub my_splice(@liste, $début) {...};
multi sub my_splice(@liste, $début, $nb_élém) {...};
multi sub my_splice(@liste, $début, $nb_élém, *@remplace) {...};

# À noter: la fonction interne splice n'utilise pas de fonctions
# multiples, mais les paramètres par défaut:
multi sub splice(@list, $start, $elems?, *@replacement);
```

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

Les *fonctions multiples* n'ont pas besoin de se distinguer par leur *arité* (nombre de paramètres), elles peuvent aussi différer par le type des arguments employés. On *pourrait* par exemple vouloir définir une fonction `ajoute` entre nombres complexes et réels :

```
multi sub ajoute(Real $x, Real $y) {return $x + $y}
multi sub ajoute(Complex $z1, Complex $z2) {return $z1.re + $z2.re + i*($z1.im + $z2.im)}
multi sub ajoute(Complex $z, Real $y) {return $z.re + $y + i*$z.im}
multi sub ajoute(Real $x, Complex $z) {return $x + $z.re + i*$z.im}

say ajoute(2, 3.5); # -> 5.5
say ajoute(3+2i, 3-i); # -> 6+1i
say ajoute(3-i, 2); # -> 5-1i
say ajoute(2.5, 3-6i); # -> 5.5-6i

# NB: exemple purement pédagogique de fonctions multiples, Perl 6
# sait additionner les complexes et réels avec l'opérateur + :
```

```
say 2.5 + (3-i); # -> 5.5-1i
```

Voir aussi plus haut l'exemple d'utilisation avec des types sous-ensembles : § 1.3).

3-5 - Voir aussi

Voir le chapitre sur les **Fonctions et signatures** du tutoriel.

4 - Programmation fonctionnelle en Perl 6

Dans son excellent livre **Higher Order Perl (HOP)(HOP)**, Mark-Jason Dominus a montré en détail comment il était possible d'utiliser un modèle de programmation fonctionnelle en Perl 5 et obtenir ainsi une expressivité bien plus grande et même étendre le langage. Ce thème a également fait l'objet sur ce site d'un tutoriel en français et en trois parties consacré à *La programmation fonctionnelle en Perl* :

- **Partie 1 : les opérateurs de listes ;**
- **Partie 2: les fonctions d'ordre supérieur ;**
- **Partie 3 : étendre le langage.**

En Perl 6 comme en Perl 5, les fonctions sont des objets d'ordre supérieur (c'est-à-dire qu'elles peuvent elles-mêmes être passées en paramètre ou en valeur de retour d'une fonction à une autre), et la quasi-totalité de ce qui est décrit dans le livre de Dominus et dans le tutoriel cité ci-dessus peut s'appliquer, aux nuances de syntaxe près, à Perl 6.

Perl 6 permet donc de la même manière d'utiliser le paradigme de la programmation fonctionnelle. Bien mieux, Perl 6 a intégré dans le cœur du langage de nombreux concepts additionnels issus de la programmation fonctionnelle :

- les blocs de code sont des fermetures (ou des lambda au sens Lisp) ;
- listes paresseuses ;
- itérateurs ;
- nouvelles fonctions ou nouveaux opérateurs de listes implémentant des opérations de type *reduce* ou *combine* ;
- curryfication ;
- etc.

Perl 6 offre également la possibilité de combiner de façon très simple des opérateurs existants pour en créer de nouveaux surpuissants.

4-1 - Nouveaux opérateurs de listes

La programmation fonctionnelle en Perl 5 s'appuyait notamment sur les nombreux opérateurs de listes du langage comme `sort`, `join`, `split`, `reverse`, `each`, `keys`, `values` et surtout, en particulier, `map`, `grep` et `for`. Ceux-ci permettaient notamment de créer des fonctions génériques abstraites pour étendre le langage.

Perl 6 ajoute de nombreux nouveaux opérateurs ou fonctions sur les listes. On peut notamment citer les suivants :

X	L'opérateur <i>croix</i> ou <i>cross</i> (X) renvoie un produit cartésien entre deux ou plusieurs listes, c'est-à-dire une liste de 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. <i>Voir aussi 4.2.2 Le métaopérateur croix.</i>
Z / zip / roundrobin	L'opérateur <i>zip</i> (Z) associe deux ou plusieurs listes en intercalant les éléments de chaque liste : <code>my @c = 1, 2, 3 Z <a b c>; # -> 1, a, 2, b, 3, c</code>

	<p>Les routines (c'est-à-dire fonctions ou méthodes) <code>zip</code> (fermeture éclair) et <code>roundrobin</code> (tourniquet) effectuent la même opération sur deux ou plusieurs listes (la première s'arrête dès qu'une liste est épuisée, la seconde continue avec les éléments des listes plus longues).</p> <pre>print " (\$_) " for zip <a b c>, <d e f>; # -> (a d) (b e) (c f)</pre>
<code>assuming</code>	<p>La méthode <code>assuming</code> appliquée à des objets de type <code>code</code> permet de <i>curryfier</i> (voir des explications complémentaires sur ce terme et cette technique dans ce chapitre d'un tutoriel sur la programmation fonctionnelle en Perl) une fonction, c'est-à-dire de créer une autre fonction « se souvenant » des paramètres passés à la création et prenant donc moins de paramètres à l'exécution. Par exemple, la fonction interne <code>substr</code> peut prendre classiquement trois paramètres : une chaîne de caractères, la position du début et la longueur de la sous-chaîne à extraire. On peut créer une « variante » curryfiée <code>f</code> de <code>substr</code> opérant toujours sur la même chaîne : <code>my &f = &substr.assuming("Je pense, donc je suis");</code>. La fonction <code>f</code> peut maintenant extraire une partie de la phrase qu'elle a mémorisée : <code>say f(3, 5); # -> pense.</code> (Équivalent à <code>say substr("Je pense, donc je suis", 3 5);</code>.) Cela peut économiser de la frappe, mais c'est surtout utile pour, par exemple, éviter de passer toujours les mêmes paramètres à une fonction de rappel appelée récursivement.</p>
<code>* Whatever</code>	<p>L'opérateur « étoile-<i>Whatever</i> » est une autre façon de <i>curryfier</i> une fonction (ou une expression) : <code>my &tiers = * / 3; say tiers(126); -> 42.</code> L'astérisque, nommé <i>Whatever</i> (valeur quelconque donnée), est une marque substitutive (<i>placeholder</i>) de l'argument qui sera passé en paramètre. La fonction <code>tiers</code> est une fermeture (elle « se souvient » de la valeur du diviseur).</p> <p>La curryfication avec l'« étoile-<i>Whatever</i> » est plus générale que celle avec la méthode <code>assuming</code>, car elle permet de <i>curryfier</i> facilement autre chose que le premier argument : <code>say ~(1, 3).map: 'hi' x *; # -> hi hihhi.</code></p> <p>On peut construire une liste « infinie » paresseuse des nombres de Fibonacci comme suit : <code>my @fib = 0, 1, *+* ... *; say @fib[7]; #-> 13.</code></p>
<code>reduce</code>	<p>Applique la fonction de rappel définie dans le premier paramètre aux premier et deuxième éléments de la liste définie dans le second paramètre, puis au résultat de cette opération et au troisième élément de la liste, et ainsi de suite, et renvoie un élément unique qui pourra être, par exemple, le produit ou la somme des éléments, ou l'élément le plus grand ou le plus petit de la liste. On peut par exemple calculer la somme des nombres de 1 à 10 comme suit :</p> <pre>my \$somme_1_10 = (1..10).reduce: * + *; # -> 55</pre> <p>et définir une fonction factorielle comme suit :</p> <pre>sub fact(Int \$i) {(1..\$i).reduce: * * *}</pre>
<code>gather / take</code>	<p>Un bloc <code>gather</code> renvoie une liste paresseuse. Quand on a besoin d'un élément de cette liste, le bloc est exécuté jusqu'à ce que l'opérateur <code>take</code> produise un élément utilisable. Par exemple, le code suivant renvoie une liste des triples des nombres pairs compris entre 1 et 10 : <code>my @liste = gather { for 1..10 {take 3 * \$_ if \$_ %%2} }; # -> 6, 12, 18 ... 30.</code> Voir aussi le chapitre du tutoriel sur Gather/take et un exemple plus complet.</p>
<code>combinations</code>	<p>Méthode ou fonction renvoyant les combinaisons des éléments d'une liste en entrée :</p> <pre>print map {" \$_ "}, .join(" ") for <a b c>.combinations(2);</pre> <pre># -> a b a c b c ></pre>

	<pre>print " (\$_)" for (5..7).combinations(2); # -> (5 6) (5 7) (6 7) print " (\$_)" for (5..7).combinations(2..3); # -> (5 6) (5 7) (6 7) (5 6 7)</pre>
permutations	<p>Méthode ou fonction renvoyant les permutations des éléments d'une liste en entrée :</p> <pre>print " (\$_)" for (5..7).permutations; # (5 6 7) (5 7 6) (6 5 7) (6 7 5) (7 5 6) (7 6 5)</pre>
rotor	<p>La méthode rotor est invoquée sur une liste et renvoie une suite de listes dans laquelle chaque sous-liste est composée d'éléments de l'invoquant. Si le paramètre @cycle passé est composé d'un seul entier, chaque sous-liste aura un nombre d'éléments égal à cet entier. Si un paramètre booléen :partial existe et est vrai, la dernière sous-liste sera renvoyée, même si elle ne contient pas le nombre voulu d'éléments.</p> <pre>say ('a'..'g').rotor(3).join(' '); # a b c d e f say ('a'..'g').rotor(3, :partial).join(' '); # a b c d e f g</pre> <p>Si un élément du premier paramètre @cycle passé en paramètre est de type Pair, alors la clef spécifie la longueur des sous-listes et la valeur l'intervalle entre les sous-listes. Si la valeur est négative, les listes se chevauchent :</p> <pre>say ('a'..'h').rotor(2 => 1).join(' '); # a b d e g h say ('a'..'h').rotor(3 => -1).join(' '); # a b c c d e e f g</pre> <p>S'il y a plusieurs entiers passés en paramètre, rotor boucle sur cette liste d'entiers pour déterminer la longueur de chaque sous-liste :</p> <pre>say ('a'..'h').rotor(2, 3).join(' '); # a b c d e f g say ('a'..'h').rotor(1 => 2, 3).join(' '); # a d e f g</pre>
classify	<p>La routine classify transforme une liste en un hachage de tableaux selon une fonction de répartition (ou de « mappage »). En sortie, chaque clef du hachage porte l'un des noms donnés par la fonction de répartition et pointe vers un tableau contenant les valeurs ayant satisfait la condition correspondante.</p> <pre>say classify { \$_ % 2 ?? 'pair' !! 'impair' }, (1, 7, 6, 3, 2); # pair => 6 2, impair => 1 7 3 say ('hello', 1, 22/7, 42, 'world').classify: { .Str.chars }; # 1 => 1, 2 => 42, 5 => hello world, 8 => 3.142857 # NB : classés selon le nombre de caractères de la chaîne</pre>
comb	<p>Renvoie toutes les reconnaissances possibles (gourmandes) d'un motif et d'une chaîne de caractères :</p> <pre>say comb /a bc/, "abcbcabcb"; # -> a bc bc a bc say comb /a+/, "aaaaaaaaaaaaa"; # -> /a+/aaaaaaaaaaaaa</pre>
... (infixé)	<p>Opérateur de séquence pour produire des listes paresseuses (et potentiellement infinies) :</p> <pre>say 1 ... 4; # 1 2 3 4 - liste linéaire say 4 ... 1; # 4 3 2 1 say 'a' ... 'e'; # a b c d e say 0, 6 ... 42; # 0 6 12 18 24 30 36 42 - encore linéaire say 2, 4, 8 ... 128; # 2 4 8 16 32 64 128 - suite exponentielle say 1,3 ... 2e32; # 1 3 5 7 9 11 13 (...) 197 199 ... - paresseuse</pre> <p>Avec l'opérateur étoile-Whatever *, la liste devient « infinie » :</p> <pre>say 2, 5 ... *; # 2 5 8 11 14 (...) 293 296 299 ...</pre> <p>Une manière particulièrement simple de générer une suite de Fibonacci infinie, mais paresseuse :</p>

	<pre>say 1, 1, * + * ... *; # 1 1 2 3 5 8 13 21 34 55 89 144 (...) 31940434634990099905 51680708854858323072 83621143489848422977 ...</pre>
unique / squish	<p>La routine unique prend en entrée une liste et renvoie une liste d'éléments distincts (sans doublons). L'ordre de la liste dédoublonnée est tel que c'est le premier élément de chaque élément en doublon qui est conservé :</p> <pre>say <a b b c c b a>.unique; # -> a b c say unique <c a b b c c b a>; # -> c a b</pre> <p>Le paramètre optionnel :as permet de normaliser temporairement les éléments de la liste avant de tester leur unicité :</p> <pre>say <a A B b c b C>.unique :as(&lc); # -> a B c</pre> <p>La méthode ou fonction squish fait à peu près la même chose, mais élimine seulement les doublons adjacents dans la liste en entrée.</p> <pre>say <a a b b b c c>.squish; # -> a b c say <a b b c c b a>.squish; # -> a b c b a</pre> <p>La méthode squish admet aussi le paramètre optionnel :as de normalisation. La méthode squish est préférable à la méthode unique si l'on sait que la liste en entrée est triée, de sorte que les doublons sont forcément adjacents, car elle peut être beaucoup plus rapide.</p>
sort	<p>La fonction ou méthode sort (qui existe bien sûr en Perl 5, mais bénéficie de quelques améliorations en Perl 6) trie les éléments d'une liste. Par défaut, l'ordre est lexicographique (et non numérique) :</p> <pre>sort <a A b C a C D q a>; # -> A C C D a a a b q <a abc ba bac bcd ac abcd>.sort; # -> a abc abcd ac ba bac bcd sort <1 4 10 30 5 3 20>; # -> 1 10 20 3 30 4 5 (! lexicographique)</pre> <p>Si l'on passe en paramètre une fonction acceptant un seul paramètre, cette fonction sert à normaliser les éléments et est appliquée une seule fois à chaque élément de la liste pour le tri, avec mise en cache du résultat normalisé (« transformation de Schwartz » implicite) :</p> <pre>sort &uc, <a A b C a C D q a>; # -> a A a a b C C D q say (3, -4, 7, -1, 2, 0).sort: *.abs; # -> 0 -1 2 3 -4 7</pre> <p>Si l'on passe une fonction acceptant deux paramètres, alors celle-ci est invoquée pour comparer chaque paire de paramètres et doit dans ce cas retourner les valeurs Order::Increase, Order::Same ou Order::Decrease.</p> <pre>say (3, -4, 7, -1, 2, 0).sort: { \$^b leg \$^a }; # 7 3 2 0 -4 -1</pre> <p><i>Voir aussi les chapitres Twigils et Sort du tutoriel.</i></p>
first / first-index / last-index	<p>La fonction ou méthode first renvoie le premier élément d'une liste satisfaisant une condition passée en paramètre :</p> <pre>say first * > 5, 1, 3, 3.14159, 42; # -> 42 say (5, 1, 3, pi, 42).first: * > 5; # -> 42 say first /roi/, <un deux trois quatre cinq>; # -> trois</pre> <p>La fonction first renvoie une erreur si aucun élément ne satisfait la condition :</p> <pre>say first /^roi/, <un deux trois quatre cinq>; # -> Nil</pre> <p>Les fonctions ou méthodes first-index et last-index sont analogues et renvoient, respectivement, l'indice dans le tableau du premier ou du dernier élément remplissant la condition reçue en paramètre.</p>

Ces nouveaux opérateurs sont indiscutablement très pratiques et peuvent, en tant que tels, faciliter la vie de tout programmeur. Il n'y aurait cependant pas lieu de s'attarder sur eux plus que sur d'autres s'ils ne contribuaient à rendre le langage Perl 6 foncièrement plus expressif et plus extensible en élargissant les possibilités de programmation fonctionnelle par rapport à Perl 5.

4-1-1 - Programmation par listes ou par flux de données

Le tutoriel **La programmation fonctionnelle en Perl - Partie 1 : les opérateurs de liste** montre comment il est possible d'enchaîner plusieurs opérateurs de listes pour former une sorte de pipeline de données, dans lequel les données en entrée sont manipulées par un premier opérateur de liste qui fournit les données modifiées qu'il produit en entrée à un second opérateur, lequel effectue une nouvelle transformation des données pour les fournir éventuellement à un troisième opérateur de liste, et ainsi de suite le cas échéant. Ceci met en place un modèle de programmation par flux de données (*dataflow programming*) ou pipeline de données. Cette approche présente l'avantage de diviser un problème relativement complexe en une série de tâches plus simples.

L'exemple classique, voire canonique, de ce genre de processus en Perl 5 est la **transformation de Schwartz**, qui enchaîne un `map`, un `sort` et un `map` pour trier des données : le premier `map` enrichit les données avec le critère de tri, le `sort` effectue le tri, et le dernier `map` remet les données au format d'origine. Le lecteur intéressé est invité à consulter le lien ci-dessus, la mise en place d'une transformation de Schwartz étant moins d'actualité en Perl 6 dans la mesure où elle est automatiquement mise en œuvre par le langage quand la fonction de comparaison utilisée par le `sort` ne prend qu'un seul paramètre (voir un exemple à la fin de ce chapitre).

Quelques-uns des exemples d'utilisation des opérateurs de listes donnés dans le tableau ci-dessus utilisent de façon très modérée cette technique. Ainsi, les exemples :

```
# fonction combinations
print map {" $_ "}, .join('|') for <a b c>.combinations(2);
# -> a|b a|c b|c >

# fonction rotor
say ('a'..'g').rotor(3).join('|');          # a b c|d e f
```

enchaînent ainsi plusieurs opérateurs de listes, dans ces cas dans le seul but de formater le résultat de la fonction étudiée.

Voici, transposé en Perl 6, un exemple inspiré du tutoriel sur la programmation fonctionnelle en Perl 5 mentionné ci-dessus (**Trier un tableau selon l'ordre de tri d'un autre tableau**). On a en entrée un tableau d'employés et un tableau de salaires :

```
my @noms = <marie nicolas isabelle yves>;
my @salaires = <1500 950 1700 2000>;
```

Le premier nom reçoit le premier salaire (Marie reçoit 1500), et ainsi de suite. On désire trier les employés selon le montant de leur salaire. La structure de données avec deux tableaux n'est pas vraiment appropriée. Il est préférable de commencer par construire une table de hachage, par exemple avec l'opérateur `zip`, puis de trier ce hachage selon le salaire :

```
my %noms-sal = @noms Z @salaires;
# -> isabelle => 1700, marie => 1500, nicolas => 950, yves => 2000
.say for sort { %noms-sal{$^a} <=> %noms-sal{$^b} }, keys %noms-sal;
# -> nicolas marie isabelle yves
```

Les variables `^a` et `^b` utilisées ici sont des paramètres positionnels autodéclarés nommés. Ce genre de paramètre utilise le twigil `^` (voir le chapitre du tutoriel sur les **Twigils**). Cela signifie que ce sont des paramètres positionnels du bloc courant qui n'ont pas besoin d'être déclarés dans la signature. Les variables sont alimentées par ordre lexicographique (pseudoalphabétique) :

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

Cette notion généralise le cas particulier des variables spéciales `$a` et `$b` de Perl 5.

Il est même possible de se passer du hachage intermédiaire, de faire les deux opérations en une seule ligne enchaînant les instructions individuelles et d'afficher les paires employé-salaire :

```
.say for sort {$^a.value <=> $^b.value}, %(@noms Z @salaires);
# nicolas => 950
# marie => 1500
# isabelle => 1700
# yves => 2000
```

Ou d'utiliser la transformation de Schwartz implicite, ce qui implique de reformater les salaires sur le même nombre de chiffres pour que la comparaison lexicographique soit équivalente à la comparaison numérique :

```
.say for sort {sprintf "%04d", $_.value}, %(@noms Z @salaires);
# même résultat
```

Les nouvelles fonctions de liste de Perl 6 étendent considérablement ce que l'on peut faire avec ce modèle de programmation.

4-1-2 - Nouvelles fonctions génériques abstraites

Pour quiconque n'a pas un peu l'habitude de la programmation fonctionnelle, la fonction `reduce` décrite au § 4.1 ci-dessus ne paie peut-être pas vraiment de mine. Elle applique une fonction aux deux premiers éléments d'une liste pour obtenir un premier résultat, puis la même fonction au résultat et au troisième élément de la liste, et ainsi de suite. Que peut-on faire avec cela ? L'exemple (nécessairement un peu simpliste) donné ci-dessus donne une première idée :

```
my $somme_1_10 = (1..10).reduce: * + *; # -> 55
```

Pas de quoi s'extasier, cependant. On peut généraliser ou abstraire l'idée en en faisant une fonction somme réutilisable :

```
sub somme (*@a) { @a.reduce: * + * }
say somme 1..10;          # -> 55
say somme 1..20;          # -> 210
my @tableau = <5 7 6 8 4>
say somme @tableau;       # -> 30
say somme <1 7 6 8>;       # -> 22
```

Bien sûr, on peut créer une fonction `produit` sur le même modèle. Mais `reduce` permet bien d'autres choses, comme déterminer l'élément le plus grand d'un tableau :

```
say "max = ", <1 5 7 4 3 14 12 5>.reduce: {$^a > $^b ?? $^a !! $^b};
# -> max = 14
```

On pourra dès lors créer des fonctions `max` et `min` :

```
sub max (*@a) { @a.reduce: {$^a > $^b ?? $^a !! $^b} }
say max <2 4 8 6 9>;      # -> 9
say max 4, 6, 8, 17, 5;   # -> 17
sub min (*@a) { @a.reduce: {$^a < $^b ?? $^a !! $^b} }
say min <2 4 8 6 9>;      # -> 2
```

Ou, si l'on préfère une notation syntaxique fonctionnelle :

```
sub max (*@a) { reduce {$^a > $^b ?? $^a !! $^b}, @a }
sub min (*@a) { reduce {$^a < $^b ?? $^a !! $^b}, @a }
```

On voit dès lors qu'il est facile d'enrichir le langage en créant en quelques lignes de code toute une bibliothèque de fonctions de listes :

```
sub somme      (*@a) {reduce * + *, @a }
sub produit   (*@a) {reduce * * *, @a }
sub max-num   (*@a) {reduce {$^a > $^b ?? $^a !! $^b}, @a}
sub min-num   (*@a) {reduce {$^a < $^b ?? $^a !! $^b}, @a}
sub avg       (*@a) {@a.elems ?? (somme @a) / @a.elems !! Nil} # moyenne
sub variance  (*@a) {my $moy = avg @a; avg (map {($_ - $moy)**2}, @a )}
sub std-dev   (*@a) {sqrt variance @a }; # écart-type
sub max-str   (*@a) {reduce {$^a gt $^b ?? $^a !! $^b}, @a}
sub min-str   (*@a) {reduce {$^a gt $^b ?? $^b !! $^a}, @a}
sub longest   (*@a) {reduce {length $^a > length $^b ?? $^a !! $^b}, @a}
sub shortest  (*@a) {reduce {length $^b > length $^a ?? $^a !! $^b}, @a}
sub concat    (*@a) {reduce * ~ *, @a}
# etc.
```

Les métaopérateurs permettent d'aller encore plus loin et de le faire souvent plus simplement.

4-2 - Les métaopérateurs

Les opérateurs manipulent des données (variables, valeurs, tableaux, etc.), les métaopérateurs manipulent des opérateurs.

4-2-1 - Le métaopérateur de réduction

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` décrite ci-dessus, 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.

Le calcul d'une somme décrit au § 4.1.2 ci-dessus peut être réécrit comme suit :

```
my $somme_1_10 = [+] 1..10; # -> 55. NB: espace nécessaire après [+]
```

On pourrait de même réécrire la fonction `somme` :

```
sub somme (*@a) { [+] @a }
say somme 1..10;      # -> 55
```

mais il n'est pas certain qu'il soit encore utile d'écrire une fonction dédiée, tant l'utilisation directe du métaopérateur de réduction est aisée.

Il est de même possible de créer une fonction factorielle comme suit :

```
sub fact(Int $x) {
  [*] 1..$x;
}
my $c = fact(10);      # -> 3628800
```

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, et contrairement à la fonction `reduce` de Perl 6 vue précédemment, le métaopérateur [...] respecte l'associativité

de l'opérateur concerné, en sorte que `[/] 1, 2, 3` est interprété comme `(1 / 2) / 3` et `[-] 4, 3, 2` comme `(4 - 3) - 2` (opérateurs associatifs à gauche), alors que `[**] 1, 2, 3` est correctement interprété comme `1 ** (2 ** 3)` (associatif à droite). Le chapitre **Précédence des opérateurs** du tutoriel donne un tableau récapitulant notamment l'associativité des différents opérateurs de Perl 6.

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, 6, 8;      # "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 existe une forme particulière de cet opérateur qui utilise un antislash comme ceci : `[+]`. Elle renvoie les résultats des évaluations partielles intermédiaires. Par exemple, `[+] 1..3` renvoie la liste `1, 1+2, 1+2+3`, ce qui donne bien sûr `1, 3, 6`.

De même, en utilisant cette forme du métaopérateur de réduction avec l'opérateur de concaténation :

```
say [\~] '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 :

```
say [\**] 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) :

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

Ou même :

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

4-2-2 - Le métaopérateur croix

L'opérateur croix (ou *cross*) `X` a été décrit au début du tableau des opérateurs de listes du § 4.1. Il renvoie un produit cartésien entre deux ou plusieurs listes, c'est-à-dire une liste de 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 :

```
print "($_) " for <a b> X 1,2 X <x y>;
# -> (a 1 x) (a 1 y) (a 2 x) (a 2 y) (b 1 x) (b 1 y) (b 2 x) (b 2 y)
```

Cet *opérateur* `X` peut aussi agir comme un *métaopérateur* : s'il est suivi d'un opérateur infixé, alors le deuxième opérateur est appliqué à tous les éléments des tuples et ce sont les résultats de cette opération sur chaque élément qui sont renvoyés. Par exemple, en postfixant la croix de l'opérateur de concaténation dans l'exemple ci-dessus :

```
# -> a1x a1y a2x a2y b1x b1y b2x b2y
say <a b> X~ 1,2 X~ <x y>;
```

Un bon vieux problème classique de l'algorithmique est la détermination des nombres de Hamming. Il s'agit des nombres de la forme :

$$H = 2^i \cdot 3^j \cdot 5^k \quad \text{avec } i, j, k \geq 0$$

Autrement dit, ce sont les nombres dont tous les diviseurs premiers sont inférieurs ou égaux à 5. Le métaopérateur X, utilisé de deux manières différentes, va permettre de construire aisément une liste de nombres de Hamming.

Combiné à l'opérateur d'exponentiation `**`, ce métaopérateur permet de générer facilement une liste de puissances de 2 :

```
say 2 X** 0..10; # -> 1 2 4 8 16 32 64 128 256 512 1024
```

On peut générer de la même façon les listes de puissances de 3 et de 5. Et on peut utiliser à nouveau le métaopérateur **X**, combiné cette fois à l'opérateur de multiplication *****, sur ces trois listes pour générer des nombres de Hamming :

```
my @h = sort ((2 X** 0..10) X* (3 X** 0..10) X* (5 X** 0..10));
# -> 1 2 3 4 5 6 8 9 10 12 15 16 18 20 24 25 27 30 32 36 40 45 48 ..
```

À remarquer que cette instruction génère environ 1300 nombres de Hamming, mais la liste n'est exhaustive que jusqu'au rang 108 (2025) : le nombre de Hamming suivant ($2048 = 2^{11}$) est forcément manquant de la liste puisque les puissances de deux utilisées ici s'arrêtent à 2^{10} . Il n'est pas bien difficile d'ajuster les puissances pour chacun des trois facteurs premiers afin de trouver par exemple le millionième nombre de Hamming (parmi plus de 7 millions d'autres) :

```
my @h1 = sort ((2 X** 0..300) X* (3 X** 0..183) X* (5 X** 0..130));  
say @h1[99999];  
# -> 519312780448388736089589843750000000000000000000000000000000000000000000000000
```

mais le traitement prend quelques dizaines de minutes.

4-2-3 - Les hyperopérateurs « et »

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

Saisir des caractères Unicode dans un éditeur

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

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



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

Sous VIM, par exemple, on saisit un caractère Unicode (en mode d'insertion) en tapant Ctrl-V (noté ^V), puis la lettre u, puis la valeur hexadécimale du code voulu. Par exemple, la lettre grecque λ (lambda) s'obtient avec la combinaison de touches ^Vu03BB. De même, le guillemet français ouvrant « s'obtient avec la combinaison ^Vu00AB (et le fermant » avec ^Vu00BB). Sous Emacs, la lettre λ s'obtient avec la combinaison suivante : Ctrl-x 8 <CR> 3bb <CR>. (<CR> signifie ici la touche « Entrée » et les espaces ont été ajoutés à des seules fins de clarté, mais ne doivent pas être saisis)

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

```
my @a = 1..5;
my @b = 6..10;
my @c = 5 «*» @b;
say @c;           # imprime 30 35 40 45 50 (5*6, 5*7, 5*8 ...)
my @d = @a »*« @b;
say @d;           # imprime 6 14 24 36 50 (1*6, 2*7, 3*8, ...)
```

On peut également utiliser les hyperopérateurs avec des opérateurs unaires :

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

Les hyperopérateurs unaires renvoient toujours une liste de la même taille que la liste en entrée.

Les hyperopérateurs infixés ont un comportement différent selon la taille de leurs opérandes :

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

L'exemple ci-dessus illustre aussi au passage l'utilisation des chevrons ASCII en lieu et place des guillemets français s'ils ne sont pas facilement disponibles dans l'éditeur utilisé.

Les hyperopérateurs fonctionnent aussi avec les opérateurs d'affectation :

```
@x »+=« @y;      # Même chose que @x = @x »+« @y
```

5 - Créer de nouveaux opérateurs

L'**Annexe 1** du tutoriel passe en revue les **opérateurs** et **fonctions internes** de Perl 6 (du moins ceux ayant plus ou moins un équivalent en Perl 5). Deux listes d'opérateurs ensemblistes essentiellement utilisables sur les listes, tableaux, ensembles, sacs, assortiments, etc. ont été fournies au § 1.4.3. Le chapitre **4.1 Nouveaux opérateurs de listes** présente les nouveaux opérateurs et fonctions de listes. Un tableau de précédences est donné au chapitre **Précédence des opérateurs** du tutoriel. Voir aussi http://...#Operator_Precedence.

Toutes ces listes ne donnent pas un état exhaustif des opérateurs disponibles en Perl 6, mais couvrent une très grosse partie des besoins. Il serait fastidieux de vouloir les énumérer tous ici. Le lecteur est invité à consulter la documentation officielle (<http://doc.perl6.org/routine-operator.html>).

Le chapitre précédent a montré comment **Les métaopérateurs** permettent de combiner des opérateurs pour en créer de nouveaux.

L'objectif du présent chapitre est plutôt de montrer comment construire des opérateurs complètement nouveaux pour étendre le langage.

5-1 - Qu'est-ce qu'un opérateur et comment en créer un ?

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

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

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

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

Dès lors, on peut par exemple définir le nouvel opérateur préfixé double, noté `%`, comme suit :

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

Mais ce nouvel opérateur peut aussi s'appeler autrement :

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

Il est possible d'utiliser d'autres délimiteurs pour encadrer le nom de l'opérateur :

```
multi sub prefix:<triple> (Int $x) { 3 * $x }
say triple 14; # -> 42
multi sub prefix:<'tiers'> (Int $x) { $x / 3 }
say tiers 126; # -> 42
```

5-1-1 - Précédence

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

Lorsque l'on définit un nouvel opérateur, il est généralement important de définir sa précedence, ce qui se fait par rapport aux opérateurs existants :

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

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

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

Par exemple, on peut écrire ce qui suit :

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

multi sub infix:<3s> (Int $x, Int $y) is equiv(&infix:<*>) {
```



```

    3 * ($x + $y)      # triple somme
}
say 9 3s 5;           # imprime 42

```

5-1-2 - Associativité

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

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

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

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

5-1-3 - Notation postcirconfixée et circonfixée

Les opérateurs postcirconfixés sont des invocations de méthodes.

```

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

```

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

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

```

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

```

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

5-1-4 - Surcharger les opérateurs existants

Parfois, le type (préfixé, infixé, etc.) d'un nouvel opérateur suffit à lui seul à le distinguer d'un opérateur existant ayant le même nom. Par exemple, l'opérateur `!` de négation booléenne est préfixé. Il est donc très simple de créer un nouvel opérateur `!` postfixé pour dénoter la factorielle d'un entier :

```

sub postfix:<!> (Int $n) {
    [*] 2..$n
}

```



```
say 10!; # -> 3628800 - Attention : pas d'espace entre 10 et !
```

Voici une version un peu plus prudente vérifiant que le nombre reçu en paramètre est positif, ainsi qu'un exemple de jeu de tests minimal :

```
sub postfix:<!> (Int $n) {
    fail "Le paramètre n'est pas un entier naturel" if $n < 0;
    [*] 2..$n
}

use Test;
isa-ok (-1)!, Failure, "Factorielle échoue pour -1";
ok 0! == 1, "Factorielle 0";
ok 1! == 1, "Factorielle 1";
ok 5! == 120, "Factorielle d'un entier plus grand";
```

Quand le type prévu du nouvel opérateur ne suffit pas à le distinguer d'un opérateur existant, la signature peut apporter le comportement attendu. La plupart des opérateurs existants (voire tous) sont des fonctions ou des méthodes de type *multi*, et il est donc facile d'en faire des versions « sur mesure » pour de nouveaux types. Ajouter une fonction *multi* est la façon la plus courante de surcharger un opérateur :

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

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

5-1-5 - Un exemple plus complet

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

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

```
#!/usr/bin/perl6
use v6;

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

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

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

my $a = 4=>3;          # une paire pour le complexe 4 + 3i
say - $a;              # imprime -4 => -3
my $b = 5=>7;
say $a + $b;          # imprime 9 => 10
```

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

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

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

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

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

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

6 - Programmation orientée objet

Perl 6 a un modèle d'objet bien plus développé que celui de Perl 5 et cela constitue une différence majeure entre les deux versions du langage, même si l'utilisation en Perl 5 du module Moose (ou des modules dérivés simplifiés, Moo, Mo, etc.), nettement inspiré du modèle objet de Perl 6, a permis de se rapprocher en Perl 5 de ce qui existe en Perl 6.

La taille volontairement limitée des chapitres du tutoriel n'a peut-être pas permis de rendre entièrement justice à cet aspect essentiel de Perl 6. Nous allons reprendre la description de la POO en Perl 6 de façon plus détaillée et d'un point de vue entièrement nouveau (ce qui ne vous empêche pas de rejeter un coup d'œil au chapitre du tutoriel consacré aux **Objets et classes**, ne serait-ce qu'en guise d'introduction plus légère).

Perl 6 possède des mots-clefs pour créer des classes, des rôles, des attributs et des méthodes, ainsi que des méthodes et attributs privés encapsulés.

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
```

Dans sa propre conception, Perl 6 est un langage orienté objet (c'est-à-dire qu'il est lui-même construit sur un modèle objet), bien qu'il vous permette d'écrire des programmes dans d'autres styles de programmation (impératif procédural, fonctionnel, par flux de données, par contraintes, déclaratif, voire logique, etc.).

6-1 - Utiliser des objets

Pour utiliser des objets, il faut invoquer des méthodes agissant sur eux. Pour invoquer une méthode sur une expression, il faut la suffixer d'un point suivi du nom de la méthode :

```
say "abc".uc;           # -> ABC
# En l'absence d'invoquant explicite, les méthodes s'appliquent à $_
$_ = "CQFD.";
say .lc;                # -> cqfd.
```

La première ligne ci-dessus appelle la méthode `uc` (mettre en lettres capitales) sur un objet de type `Str`. Pour fournir des arguments à la méthode, il suffit d'ajouter des parenthèses après le nom de la méthode et d'y mettre ces paramètres.

```
my $texte-formaté = "Je suis Charlie".indent(8); # indentation de 8
# produit : "           Je suis Charlie"
```

S'il faut plusieurs arguments, il suffit de les séparer par une virgule :

```
my @mots = <Arnold Schwarzenegger>;
@mots.push("a", "dit : ", $texte-formaté.comb(/\w+/));
# -> Arnold Schwarzenegger a dit : Je suis Charlie
```

Une autre syntaxe d'invocation de méthode consiste à séparer le nom de la méthode de la liste d'arguments par un caractère deux-points :

```
say @mots.join: '--';
# -> Arnold--Schwarzenegger--a--dit :--Je--suis--Charlie
```

Dans la mesure où il faut mettre un « `:` » après la méthode pour lui passer des arguments sans parenthèses, une invocation de méthode non suivie d'un « `:` » ou de parenthèses est sans ambiguïté une méthode sans liste d'arguments :

```
say 4.log;           ; # 1.38629436111989 (logarithme naturel de 4)
say 4.log: +2;        # 2                (logarithme en base 2 de 4)
say 4.log +2;        # 3.38629436111989 (logarithme naturel de 4, plus 2)
```

Beaucoup d'opérations qui n'ont pas l'air d'appels de méthodes (par exemple une reconnaissance intelligente ou l'interpolation d'un objet dans une chaîne) se traduisent en fait par des appels de méthodes sous le capot.

6-1-1 - Objets-types

Les types eux-mêmes sont des objets et il est possible d'obtenir l'*objet-type* en écrivant simplement son nom :

```
my obj-type-int = Int;           # -> (Int)
```

Il est possible de connaître l'objet-type d'un objet quelconque en appelant la méthode `WHAT` (qui est en fait une macro sous la forme d'une méthode) :

```
my obj-type-int = 1.WHAT;        # -> (Int)
```

Il est possible de vérifier si des objets-types (à l'exception de Mu, le type au sommet de la hiérarchie des types dont héritent tous les autres) sont égaux à l'aide de l'opérateur d'identité `===` :

```
sub f(Int $x) {
  if $x.WHAT === Int {
    say 'Vous avez passé un Int';
  }
  else {
    say 'Vous avez passé un sous-type de Int';
  }
}
```

En vérité, dans la majeure partie des cas, la méthode `.isa` (« est un(e) ») sera suffisante et plus simple d'utilisation :

```
sub f($x) {
  if $x.isa(Int) {
    ...
  }
  ...
}
```

La vérification de la compatibilité des sous-types se fait avec l'opérateur de reconnaissance intelligente `~~` (*smart match*) :

```
my Int $i = 5;
say "Compatible avec réel" if $i ~~ Real; # Int est sous-type de Real
# -> "Compatible avec réel" (Int hérite des méthodes de Real)
```

6-2 - Classes

On déclare une classe à l'aide du mot-clef `class`, généralement suivi du nom de la classe :

```
class Voyage {
  # ...
}
```

Cette déclaration crée un objet-type et l'installe dans le paquetage courant et dans la portée lexicale courante sous le nom `Voyage`.

Il est également possible de déclarer des classes de portée lexicale :

```
my class Voyage {
  # ...
}
```

Ceci limite la visibilité de la classe à la portée lexicale courante, ce qui peut s'avérer utile si la classe est un détail de mise en œuvre à l'intérieur d'un module ou d'une autre classe.

6-2-1 - Attributs

Les attributs sont des variables privées qui existent à l'intérieur des membres d'une classe (et que possèdent tous les objets instanciant ladite classe). Ce sont eux qui stockent l'état d'un objet. En Perl 6, tous les attributs sont privés. On les déclare généralement avec le mot-clef `has` et en utilisant le twigil « `!` » :

```
class Voyage {
  has $!point-de-départ;
  has $!destination;
  has @!voyageurs;
  has $!notes;
```

```
}
```

Il n'existe pas en Perl 6 d'attribut public (ou même protégé), mais il existe une manière de générer automatiquement des accesseurs (méthodes d'accès) : il suffit de remplacer le twigil « ! » par le twigil « . » (moyen mnémotechnique : le « . » devrait vous faire penser à un appel de méthode) :

```
class Voyage {
    has $.point-de-départ;
    has $.destination;
    has @!voyageurs;
    has $.notes;
}
```

Ceci fournit par défaut des accesseurs en lecture seule. Pour autoriser des modifications de l'attribut, il faut ajouter le **trait** `is rw` :

```
class Voyage {
    has $.point-de-départ;
    has $.destination;
    has @!voyageurs;
    has $.notes is rw;
}
```

Désormais, une fois un objet `Voyage` créé, ses attributs `.point-de-départ`, `.destination` et `.notes` seront accessibles depuis l'extérieur de la classe via les accesseurs, mais, dans le code ci-dessus, seul l'attribut `.notes` sera modifiable. L'attribut `!voyageurs` reste privé et inaccessible depuis l'extérieur de la classe.

Comme les classes héritent d'un constructeur par défaut de Mu et comme nous avons demandé que des accesseurs soient générés pour nous, notre classe est déjà presque fonctionnelle :

```
# Création d'une nouvelle instance de la classe
my $vacances = Voyage.new(
    point-de-départ => 'Suède',
    destination     => 'Suisse',
    notes           => 'Équipement type camping!'
);

# Utilisation d'un accesseur :
say $vacances.point-de-départ;    # -> Suède

# Utilisation d'un accesseur de type rw pour modifier la valeur:
$vacances.notes = 'Équipement type camping plus lunettes de soleil';
```

À noter que le constructeur par défaut n'alimentera que les attributs qui ont un accesseur, mais il peut initialiser des attributs en lecture seule.

6-2-2 - Méthodes

On déclare une méthode à l'aide du mot-clef `method` à l'intérieur du corps d'une classe :

```
class Voyage {
    has $.départ;
    has $.destination;
    has @!voyageurs;
    has $.notes is rw;

    method ajoute_voyageur($nom) {
        if $nom ne any(@!voyageurs) {
            push @!voyageurs, $nom;
        }
        else {
            warn "$nom est déjà du voyage!";
        }
    }
}
```

```

    }
    method décrire() {
        join " ", "De", $!départ, "à", $!destination,
            "- Voyageurs:", @!voyageurs;
    }
}

```

La classe peut être appelée comme suit :

```

my $week-end-amoureux = Voyage.new(départ => "Paris",
                                     destination => "Londres");
$week-end-amoureux.ajoute_voyageur($_) for <Roméo Juliette Roméo>;
say $week-end-amoureux.perl;
say "Ajoute une note";
$week-end-amoureux.notes = "Eurostar";
say $week-end-amoureux.perl;
say $week-end-amoureux.décrire;

```

Ce qui affiche :

```

Roméo est déjà du voyage! in method ajoute_voyageur at voyage.pl:12
Voyage.new(départ => "Paris", destination => "Londres", notes => Any)
Ajoute une note
Voyage.new(départ => "Paris", destination => "Londres", notes => "Eurostar")
De Paris à Londres - Voyageurs: Roméo Juliette

```

On constate que le programme avertit à propos du voyageur (Roméo) ajouté par erreur une seconde fois. L'ajout d'une note directement depuis le code appelant ne pose pas de problème, car c'est un attribut public. L'appel de méthode `$week-end-amoureux.perl` renvoie les seuls attributs publics tandis que l'appel de la méthode `décrire` permet aussi de connaître le nom des tourtereaux partant en week-end.

Une méthode peut avoir une signature, tout comme une fonction (*subroutine*). Les attributs sont utilisables dans des méthodes et peuvent toujours être employés avec le twigil « `!` », même s'ils ont été déclarés avec le twigil « `.` » (comme dans la méthode `décrire` de l'exemple ci-dessus). La raison en est qu'en fait, le twigil « `.` » déclare un twigil « `!` » et génère en outre un accesseur. Autrement dit, les attributs publics ne sont rien d'autre que des attributs privés dotés d'un accesseur public.

Il y a une différence subtile mais importante entre, par exemple, `$!départ` et `$.départ` dans la méthode `décrire`. Le premier effectue toujours une simple recherche sur la valeur de l'attribut. Cela ne coûte pas grand-chose et vous savez que c'est l'attribut déclaré dans la classe. La seconde syntaxe, avec le « `.` », est en fait un appel de méthode et peut donc être redéfinie (*overriden*) dans une classe fille. Il faut par conséquent utiliser uniquement `$.départ` si l'on désire explicitement autoriser une redéfinition de la méthode.

6-2-3 - Objet self

À l'intérieur d'une méthode, il est possible d'utiliser le terme `self` qui est lié à l'objet invoquant, c'est-à-dire à l'objet sur lequel la méthode a été invoquée. `self` peut être utilisé pour appeler d'autres méthodes sur l'invoquant. À l'intérieur d'une méthode, la syntaxe `$.point-de-départ` est équivalente à `self.point-de-départ`.

6-2-4 - Méthodes privées

Les méthodes déclarées avec un point d'exclamation « `!` » sont *privées*, c'est-à-dire qu'elles ne peuvent être invoquées que depuis l'intérieur de la classe (et non de l'extérieur). On les appelle avec un point d'exclamation au lieu d'un simple point :

```

method !action-privée($x) {
    ...
}
method publique($x) {

```

```
if self.précondition {
    self!action-privée(2 * $x)
}
}
```

Les méthodes privées ne sont pas héritées dans les classes filles.

6-2-5 - Subméthodes

Une *subméthode* (*submethod*) est une méthode publique qui n'est pas héritée dans les classes filles. Leur dénomination émane du fait qu'elles sont sémantiquement équivalentes à des fonctions (*subroutines*), mais invoquées avec une syntaxe de méthode.

Les subméthodes sont utiles pour accomplir des tâches de construction et de destruction d'objets, ainsi que pour des tâches qui sont si spécifiques à un type donné que les sous-types devront certainement les redéfinir.

Par exemple, le constructeur par défaut `new` appelle la subméthode `BUILD` sur chaque classe de la chaîne d'héritage :

```
class Point2D {
    has $.x;
    has $.y;

    submethod BUILD(:$!x, :$!y) {
        say "Initialise Point2D";
    }
}

class Point2DInversible is Point2D {
    submethod BUILD() {
        say "Initialise Point2DInversible";
    }
    method inverse {
        self.new(x => - $.x, y => - $.y);
    }
}

my $pt_inv = Point2DInversible.new(x => 1, y => 2);
say $pt_inv.inverse.perl;
```


Ce qui affiche :

```
Initialise Point2D
Initialise Point2DInversible
Point2DInversible.new(x => -1, y => -2)
```

6-2-6 - Héritage

Les classes peuvent avoir des *classes mères* (ou super-classes).

```
class Enfant is Parent1 is Parent2 { }
```

Si l'on appelle sur une classe fille une méthode qui n'est pas définie dans cette classe fille, alors c'est une méthode ayant le même nom dans l'une des classes mères qui sera appelée, si elle existe. L'ordre dans lequel les classes mères sont consultées s'appelle l'ordre de résolution des méthodes (*method resolution order* ou MRO). Perl 6 utilise la  **méthode C3 de résolution**. Il est possible de connaître cet ordre pour un type donné grâce à un appel à sa métaclasse :

```
say Int.^mro;      # (Int) (Cool) (Any) (Mu)
```

Si une classe ne spécifie pas de classe mère, alors la classe `Any` est sa classe mère par défaut. Toutes les classes héritent directement ou indirectement de `Mu`, la racine de la hiérarchie des types.

Tous les appels aux méthodes publiques sont virtuels au sens C++ du terme, ce qui signifie que c'est le type réel de l'objet qui détermine quelle méthode invoquer, et non son type déclaré :

```
class Parent {
    method farfouille {
        say "méthode farfouille de la classe mère"
    }
}

class Enfant is Parent {
    method farfouille {
        say "Appel de la méthode farfouille de la classe fille"
    }
}

my Parent $test;      # type déclaré: Parent
$test = Enfant.new;   # type réel: Enfant
$test.farfouille;     # appelle la méthode farfouille de la classe fille
# affiche : Appel de la méthode farfouille de la classe fille
```

6-2-7 - Construction d'objet

Les objets sont généralement créés au moyen d'appels de méthodes, soit sur l'objet-type, soit sur un autre objet de même type.

La classe `Mu` fournit un constructeur, la méthode `new`, qui prend en paramètres des arguments nommés et les utilise pour initialiser les attributs publics :

```
class Point {
    has $.x;
    has $.y = 2 * $!x; # valeur par défaut de $y si non spécifiée
}

my $p = Point.new( x => 1, y => 2 );
#      ^^ méthode héritée de la classe Mu
say "x: ", $p.x;      # -> x: 1
say "y: ", $p.y;      # -> y: 2

my $p2 = Point.new( x => 5 );
# la valeur sert à calculer $y si l'argument $y n'est pas fourni
# value for y.
say "x: ", $p2.x;     # -> x: 5
say "y: ", $p2.y;     # -> y: 10
```

`Mu.new` appelle la méthode `bless` sur son invoquant et passe tous les arguments nommés. Il est donc possible d'utiliser la méthode de bas niveau `bless` pour créer ses propres constructeurs sur mesure, mais cela sortirait du cadre de ce tutoriel.

6-2-8 - Clonage d'objets

La classe mère `Mu`, dont héritent toutes les autres classes, fournit une méthode nommée `clone` qui est quelque peu magique en ce sens qu'elle peut copier des valeurs à partir des attributs privés d'une instance pour créer une nouvelle instance. Cette copie est superficielle, c'est-à-dire qu'elle ne fait que lier les attributs aux valeurs respectives contenues dans l'instance d'origine, elle ne fait pas de copie de ces valeurs contenues.

Comme avec `new`, il est possible de fournir des valeurs initiales pour les attributs publics et, dans ce cas, ces valeurs l'emportent sur celles provenant de l'instance d'origine. Voici l'exemple fourni dans la documentation de la classe `Mu`.


```
class Point2D {
    has ($.x, $.y);
    multi method gist(Point2D:D:) {
        "Point($.x, $.y)";
    }
}

my $p = Point2D.new(x => 2, y => 3);

say $p;                # Point(2, 3)
say $p.clone(y => -5);  # Point(2, -5)
```

6-3 - Les rôles

Par certains aspects, les rôles sont semblables aux classes : ils constituent en effet une collection d'attributs et de méthodes. Mais ils sont différents dans la mesure où les rôles ne décrivent qu'une partie du comportement d'un objet et aussi dans la façon dont les rôles s'appliquent aux classes. Pour le dire autrement, les classes sont censées gérer des instances (des données) et les rôles sont censés gérer un comportement et la réutilisation du code.

Pour affecter un rôle à une classe, on utilise le mot-clef `does` `Nom_du_role` lors de la définition d'une classe.

```
role Sérialisable {
    method sérialise() {
        self.perl; # forme très primitive de sérialisation
    }
    method désérialise($buffer) {
        EVAL $buffer; # opération inverse de .perl
    }
}

class Point does Sérialisable {
    has $.x;
    has $.y;
}

my $p = Point.new(:x(1), :y(2));
my $sérialisé = $p.sérialise;      # méthode fournie par le rôle
my $clone-de-p = Point.désérialise($sérialisé);
say $clone-de-p.x;                 # -> 1
```

Les rôles sont immuables dès que le compilateur a fini d'analyser l'accolade fermante de la déclaration de rôle.

6-3-1 - Application de rôles

L'application d'un rôle diffère sensiblement de l'héritage d'une classe. Quand un rôle est appliqué à une classe, les méthodes de ce rôle sont copiées dans cette classe. Si plusieurs rôles sont appliqués à la même classe, d'éventuels conflits (par exemple des attributs ou des méthodes non multiples ayant le même nom) entraînent une erreur de compilation. Laquelle peut être résolue en fournissant une méthode ayant le même nom dans la classe en question.

Ce comportement est bien plus fiable que celui de l'héritage multiple, dans lequel les conflits ne sont jamais détectés par le compilateur, mais résolus silencieusement en prenant la méthode de la classe mère qui apparaît en premier dans le MRO - ce qui est, ou non, ce que désirait le développeur.

Par exemple, si vous avez découvert une nouvelle façon de conduire des vaches et essayez de la commercialiser comme une nouvelle forme populaire de transport, vous aurez peut-être une classe `Taureau` pour les taureaux que vous élevez, et une classe `Automobile` pour les choses que vous pouvez conduire.

```
class Taureau {
    has Bool $.castré = False;
    method mène {
        # Mène votre taurillon au vétérinaire pour le châtrer
        !$castré = True;
        return self;
    }
}
```

```

    }
}
class Automobile {
    has $.direction;
    method mène ($!direction) { }
}
class Taurus is Taureau is Automobile { }

my $t = Taurus.new;
$t.mène; # Castre $t

```

Avec cette configuration, vos pauvres clients seront dans l'incapacité d'utiliser leur Taurus et vous dans celle de vendre vos produits. Il aurait peut-être été plus judicieux d'utiliser des rôles :

```

role Taurin {
    has Bool $.castré = False;
    method mène {
        # Mène votre taurillon au vétérinaire pour le châtrer
        $!castré = True;
        return self;
    }
}
role Menable {
    has Real $.direction;
    method mène (Real $d = 0) {
        $!direction += $d;
    }
}
class Taurus does Taurin does Menable { }

```

Ce code va avorter avec un message du genre :

```

===SORRY!===
Method 'mène' must be resolved by class Taurus because it exists in
multiple roles (Menable, Taurin)

```

Cette vérification du compilateur vous épargnera, à vous et à vos clients, beaucoup de migraines à rechercher les causes subtiles d'une anomalie. Ici, vous pourriez simplement définir votre classe comme suit :

```

class Taurus does Taurin does Menable {
    method mène ($direction?) {
        self.Menable::mène($direction?)
    }
}

```

6-3-2 - Promotion automatique des rôles (punning)

Toute tentative d'instancier un rôle (ainsi que diverses autres opérations sur les rôles) créera automatiquement une classe portant le même nom que le rôle, ce qui permet d'utiliser de façon transparente un rôle comme s'il s'agissait d'une classe :

```

role Point {
    has $.x;
    has $.y;
    method abs { sqrt($.x * $.x + $.y * $.y) }
}
say Point.new(x => 6, y => 8).abs;

```


Cette création automatique de classe s'appelle *punning* et la classe générée est un *pun*. (Le mot anglais *pun* désigne habituellement un calembour ou un jeu de mots, et *punning*, c'est faire un jeu de mots ; l'auteur de ces lignes ne voit pas de rapport entre la promotion de rôles au rang de classes et l'idée d'un jeu de mots...).

6-4 - Documentation complémentaire sur les objets en Perl 6

Voir aussi cet article plus détaillé sur la programmation objet en Perl 6 :

 **Objets, classes et rôles en Perl 6 - Tutoriel de programmation orientée objet.**

7 - Les regex et les grammaires

Le chapitre **Regex (ou règles)** du tutoriel et le  **chapitre sur les Regex de l'Annexe 1** abordaient les regex de Perl 6 du point de vue des différences avec les expressions régulières de Perl 5, ce qui est bien normal dans le contexte de l'objectif déclaré de ce tutoriel.

Toutefois, les *regex* de Perl 6 sont si différentes des *expressions régulières* de Perl 5 (au point qu'elles n'ont plus le même nom) qu'il nous paraît préférable ici de redéfinir les regex Perl 6 sans le faire à partir de Perl 5 (mais en supposant que le lecteur connaît au moins un peu le mécanisme général de la reconnaissance des motifs par expressions régulières).

Les grammaires de Perl 6 sont un moyen de combiner des regex pour analyser un texte beaucoup moins bien structuré que ce que peuvent reconnaître des expressions régulières ou des regex, ce qui permet par exemple d'effectuer une analyse lexicale et syntaxique d'un document XML, HTML ou JSON, ou même le code source d'un programme informatique : ainsi, un programme Perl 6 est compilé à l'aide d'une grammaire Perl 6 écrite en Perl 6.

Il n'est toutefois pas possible, dans le cadre de ce document, d'entrer dans tout le détail de ce qu'il est possible de faire avec les regex et les grammaires de Perl 6, pour des raisons de place. Aussi ce chapitre est-il en fait un résumé d'un article beaucoup plus complet que nous avons écrit sur le sujet, **Les regex et grammaires de Perl 6 : une puissance expressive sans précédent**, que nous incitons vivement le lecteur intéressé à consulter (éventuellement après avoir lu le présent chapitre qui lui aura fourni une bonne introduction).

7-1 - Les regex de Perl 6

7-1-1 - Conventions lexicales

Perl 6 offre les constructions syntaxiques suivantes pour écrire des regex :

```
m/abc/;      # une regex immédiatement appliquée à $_
rx/abc/;     # un objet de type Regex
/abc/;       # un objet de type Regex
```

Les deux premières syntaxes peuvent utiliser d'autres délimiteurs que la barre oblique :

```
m{abc};     # ou m[abc];
rx{abc};    # ou rx!abc!;
```

À noter cependant que le caractère deux-points (« : ») et les parenthèses ordinaires (« (» et «) ») ne peuvent en principe pas servir de délimiteurs pour des regex.

D'une façon générale, les espaces blancs des motifs sont ignorés par défaut, sauf en cas de l'utilisation (explicite ou implicite) de l'adverbe `:s` ou `:sigspace`, voir § **7.1.9.1.2** plus bas) .

```
say "Reconnu" if "abc" ~~ /a b c /;  # -> "Reconnu"
```

Comme dans le reste de Perl 6, les commentaires commencent habituellement avec le caractère *dièse* ou plus exactement *croisillon* (« # ») et vont jusqu'à la fin de la ligne (sauf si le croisillon est utilisé comme délimiteur, auquel

cas il est nettement préférable de ne pas essayer de s'en servir comme caractère de début de commentaire). Les **commentaires multilignes** sont également possibles.

7-1-2 - Littéraux

Le cas le plus simple de motif de reconnaissance d'une regex est une chaîne constante. Dans ce cas, reconnaître un motif consiste à rechercher le motif comme une sous-chaîne de la chaîne :

```
my $chaîne = "Esperluette"; # le nom parfois donné au signe &;
if ($chaîne =~ m/ perl / {
    say "\$chaîne contient 'perl'"; # esperluette contient 'perl'
}
```

Tous les caractères alphanumériques (Unicode) et le caractère souligné ou *underscore* (« `_` ») sont des reconnaissances littérales. Tous les autres caractères (signes de ponctuation, symboles, etc.) doivent être protégés par le caractère d'échappement *antislash* (« `\` ») ou être cités entre apostrophes (ou guillemets simples) :

```
/ 'deux mots' / # reconnaît 'deux mots', espace blanc compris
/ "a:b" / # reconnaît 'a:b', caractère deux-points compris
/ '#' / # reconnaît le caractère dièse (ou hash)
/ moi\@gmail\.com/ # échappements pour protéger l'@ et le .
```

Lorsqu'ils sont protégés par un caractère d'échappement, les caractères alphanumériques prennent une signification particulière : par exemple, le métacaractère `\d` représente une classe de caractères pouvant signifier un chiffre quelconque (Unicode) ; de nombreux exemples seront donnés plus loin (notamment au § 7.1.3.1).

Les chaînes de caractères sont explorées de gauche à droite, il suffit donc, par exemple, qu'une sous-chaîne soit égale au motif :

```
if 'abcdefg' =~ / de / {
    say ~$/; # de -> motif reconnu
    say $/.prematch; # abc -> ce qui précède le motif reconnu
    say $/.postmatch; # fg -> ce qui suit le motif reconnu
    say $/.from; # 3 -> position du début de la reconnaissance
    say $/.to; # 5 -> position de ce qui suit la reconnaissance
};
```

Les résultats de la reconnaissance sont stockés dans la variable `$/` (représentant le *match object*, que l'on traduira dans ce document par « objet reconnu ») et sont également renvoyés par la reconnaissance. Le résultat est de type `Match` si la reconnaissance a réussi, et `Nil` (équivalent approximatif de `undef` en Perl 5) sinon.

7-1-3 - Métacaractères et classes de caractères

Une *classe de caractères* est un élément de syntaxe des regex qui permet de reconnaître non plus un seul caractère déterminé, mais un caractère appartenant à tout un ensemble de caractères ayant éventuellement des traits communs (reconnaître par exemple l'un quelconque des chiffres de 0 à 9, ou l'un quelconque des caractères alphabétiques minuscules).

Le point (« `.` ») reconnaît tout caractère simple (sauf s'il est précédé d'un caractère d'échappement, auquel cas il reconnaît un point littéral) :

```
'perl' =~ /per./; # Reconnaît toute la chaîne
'perl' =~ / per . /; # Idem (espaces blancs ignorés);
'perl' =~ / pe.l /; # Idem: le . reconnaît le r
'Épelle' =~ / pe.l /; # Idem: le . reconnaît le premier l

'perl' =~ /. per / # Pas de reconnaissance:
# le . ne reconnaît rien avant la chaîne per
```

Contrairement à Perl 5, le point reconnaît aussi toujours le caractère retour à la ligne.

7-1-3-1 - Caractère d'échappement et classes de caractères prédéfinies


Il existe des classes de caractères prédéfinies ayant la forme de l'antislash (ou barre oblique inverse) suivi d'une lettre, par exemple `\w` (caractère alphanumérique, comme en Perl 5). Si la lettre est en majuscule (`\W`), c'est la négation de la classe de caractères correspondant à la même lettre en minuscule (autrement dit, `\W` reconnaît tout caractère non reconnu par `\w`). Voici quelques classes de caractères prédéfinies :

- **caractère alphanumérique** (lettres, chiffres et `_`) : `\w` (complément : `\W`) ; reconnaît par exemple `a`, `C`, `z`, `7`, `0041 A LATIN CAPITAL LETTER A`, `0031 1 DIGIT ONE`, `03B4 δ GREEK SMALL LETTER DELTA` ou `0409 Љ CYRILLIC CAPITAL LETTER LJE` ;
- **caractère numérique** : `\d` et `\D` (chiffre unique, au sens Unicode de chiffre, pas seulement nos chiffres arabes : par exemple, `U+0E53 # THAI DIGIT THREE` (chiffre thaïlandais 3) est reconnu par `\d`) ;
- **espace horizontal** : `\h` et `\H` (espaces blancs, tabulations, `U+00A0 NO-BREAK SPACE`) ;
- **espace quelconque** : `\s`.

Voir aussi des  **exemples plus variés de classes de caractères prédéfinies**.

7-1-3-2 - Propriétés Unicode

Les classes de caractères vues ci-dessus sont pratiques pour des cas courants. L'utilisation des propriétés Unicode permet une approche plus systématique et plus fine. La syntaxe d'appel est de la forme `<:propriété>`, dans laquelle « propriété » peut être un nom court ou long de propriété Unicode. Le sens précis des propriétés Unicode elles-mêmes n'est pas défini par Perl, mais par les normes Unicode.

Voici une liste de quelques propriétés Unicode courantes (voir aussi  **une liste plus complète**) :

Nom court	Nom long	Signification et remarques
L	Letter	Lettre
LC	Cased_Letter	Lettre avec sa casse (distinction capitale/majuscule)
LU	Upper_Cased_Letter ou Upper	Lettre capitale (en majuscule)
LL	Lower_Cased_Letter ou Lower	Lettre bas de casse (en minuscule)
N	Number	Nombre
Nd	Decimal_Number ou Digit	Nombre décimal (chiffre)
P	Punctuation ou Punct	Signe de ponctuation
S	Symbol	Symbole
Sm	Math_Symbol	Symbole mathématique
Sc	Currency_Symbol	Symbole monétaire (par ex. \$, £ ou €).

Par exemple, `<:Lu>` reconnaît une seule lettre capitale (majuscule).

La négation d'une propriété Unicode est obtenue avec la forme `<:!propriété>`, par exemple `<:!Lu>` reconnaîtra tout caractère unique qui n'est pas une lettre capitale.

Il est possible de combiner plusieurs propriétés à l'aide des opérateurs infixés suivants :

Opérateur	Signification	Remarque
+	Union ensembliste	ou logique (or) entre les propriétés
	Union ensembliste	ou logique (or) entre les propriétés
&	Intersection ensembliste	et logique (and) entre les propriétés
-	Différence ensembliste	Ayant la première propriété et pas la seconde
^	Intersection symétrique ensembliste	ou exclusif logique (XOR) entre les propriétés

Par exemple, pour reconnaître soit une lettre minuscule soit un nombre, il est possible d'écrire : `<:Ll+:N>` ou `<:Ll+:Number>` ou encore `<+:Lowercase_Letter+:Number>`.

Il est également possible de grouper des catégories et des ensembles de catégories avec des parenthèses, par exemple :

```
'perl6' ~~ m{\w+(<:Ll+:N>)} # 0 => #6#
```

7-1-3-3 - Classes de caractères énumérées et intervalles

Parfois, les métacaractères et classes de caractères prédéfinies ne suffisent pas. Il est heureusement simple de définir sa propre classe de caractères en plaçant entre `<[...]>` un nombre quelconque de caractères et d'intervalles de caractères (avec deux points « .. » entre les bornes de ces intervalles), avec ou sans espaces blancs :

```
"abacabadabacaba" ~~ / <[ a .. d 1 2 3 ]> / # Vrai
```

Il est possible d'utiliser à l'intérieur des `<...>` les mêmes opérateurs que pour les catégories Unicode (+, |, &, -, ^) pour combiner de multiples définitions d'intervalles ou même de les combiner avec les catégories Unicode ci-dessus.

La négation d'une classe de caractères de ce type s'obtient avec le signe « - » après le chevron ouvrant :

```
say 'pas des guillemets' ~~ / <-[ " ]> + /;
# reconnaît les caractères autres que "
```

Il est assez commun, pour analyser des chaînes délimitées par des guillemets, d'utiliser un motif utilisant des négations de classes de caractères :

```
say 'entre guillemets' ~~ / "'" <-[ " ]> * "'/;
# un guillemet, suivi de non-guillemets, suivi d'un guillemet
```

7-1-4 - Quantificateurs

Un quantificateur permet de reconnaître non pas exactement une fois, mais plutôt un nombre fixe ou variable de fois, l'atome qui le précède. Par exemple, le quantificateur « + » cherche à reconnaître une ou plusieurs fois ce qui précède.

Les quantificateurs ont une précedence plus forte que la concaténation, si bien que `/ab+/` reconnaît la lettre a suivie d'une ou plusieurs fois la lettre b. La situation est inversée avec des apostrophes : `/'ab'+/` reconnaît les chaînes 'ab', 'abab' 'ababab', etc.

Quantificateur	Signification	Remarques ou exemples
+	Un ou plusieurs	Reconnaît l'atome précédant une ou plusieurs fois, sans limite supérieure
*	0 ou plusieurs fois	Par exemple, pour autoriser un espace ou plusieurs espaces optionnel(s) entre a et b : / a \s* b /
?	0 ou 1 fois	Par exemple pour un caractère optionnel unique
** min..max	Nombre arbitraire de fois entre min et max	say Bool('a' ~~ /a ** 2..5/); #-> False say Bool('aaa' ~~ /a ** 2..5/); #-> True
** n	Exactement n fois	say Bool('aaaa' ~~ /a ** 5/); #-> False say Bool('aaaaa' ~~ /a ** 5/); #-> True
%	Quantificateur modifié	Pour faciliter le travail avec les CSV, le modificateur % appliqué à l'un des quantificateurs ci-dessus, permet de spécifier un séparateur qui doit être présent entre les reconnaissances répétées. Par exemple : / a+ % ';' / reconnaît 'a,a' ou 'a,a,a' etc., mais ni 'a,', ni 'a,a,'.

7-1-4-1 - Avidité et frugalité des quantificateurs

Par défaut, les quantificateurs + et * sont *avides* ou *gourmands*, c'est-à-dire qu'ils cherchent la reconnaissance la plus longue possible dans la chaîne. Par exemple :

```
say ~$/ if 'aabaababa' ~~ /.+ b /; # -> aabaabab
```

Ci-dessus, la sous-regex .+ recherche la plus longue chaîne possible de caractères quelconques permettant encore de reconnaître la suite de la regex, ici l'atome b, ce qui peut être le but recherché. Mais il arrive assez fréquemment que cela soit une erreur de débutant et que l'objectif soit plutôt de reconnaître « des caractères quelconques jusqu'au premier b ». Dans ce cas, on préférera utiliser un quantificateur *non gourmand* (ou « frugal »), obtenu en postfixant le quantificateur d'origine avec un point d'interrogation, ce qui donne soit +?, soit *?. Par exemple :

```
say ~$/ if 'aabaababa' ~~ /.+? b /; # -> aab
```

7-1-5 - Alternatives (reconnaître ceci ou cela)

Pour reconnaître une possibilité parmi plusieurs, il faut les séparer par « || » ; la première reconnaissance trouvée (de gauche à droite) l'emporte. Par exemple, les fichiers d'initialisation (du genre config.ini) ont souvent la forme suivante :

```
[section]
clef = valeur
```

Quand on lit une ligne d'un fichier de ce genre, ce peut être soit une section, soit une paire clef-valeur. En première approche, la regex pour lire ce type de fichier pourrait être :

```
/ '[' \w+ ']' || \S+ \s* '=' \s* \S* /
```

C'est-à-dire :

- soit un mot entre crochets ;
- soit une chaîne composée de caractères autres que des espaces blancs, suivie de 0 ou plusieurs espaces, suivis du signe égal « = », suivi à nouveau d'espaces optionnels, suivi d'une autre chaîne composée de caractères autres que des espaces blancs.

Il existe une autre forme d'alternative, utilisant le séparateur « | » (au lieu de « || »). L'idée est la même, mais c'est la reconnaissance la plus longue (et non plus la première) qui est retenue (voir [un exemple de la différence entre ces deux types d'alternatives](#)). C'est cette règle de la reconnaissance la plus longue qui permet par exemple à la grammaire de Perl 6 de reconnaître des identifiants de variables contenant le caractère « - », sans que celui-ci soit confondu avec la soustraction.

7-1-6 - Ancres

Le moteur de regex essaie de trouver une correspondance dans une chaîne en cherchant de gauche à droite.

```
say so 'Saperlipopette' ~~ / perl /; # Vrai (True, en fait)
#      ^^^^
# (so renvoie une évaluation booléenne, donc, en fait True ou False)
```

Mais ce n'est pas toujours ce que l'on désire. Par exemple, on peut vouloir reconnaître toute la chaîne, ou toute une ligne, ou un ou plusieurs mots entiers, ou attacher de l'importance à l'*endroit* de la chaîne où la reconnaissance se produit. Les ancres (et les assertions) permettent de spécifier *où* la reconnaissance aura lieu.

Il faut que les ancres d'une regex soient reconnues pour que l'ensemble de la regex le soit, mais les ancres *ne consomment pas* de caractère dans la chaîne.

Ancre	Signification	Remarques ou exemples
^	Début de la chaîne	'Saperlipopette' ~~ /perl/; # Vrai 'Saperlipopette' ~~ /^ perl/; # Faux 'perles fines' ~~ /^ perl/; # Vrai
^^	Début de ligne	^^ reconnaît le début de la chaîne ou ce qui suit un caractère de retour à la ligne
\$\$	Fin de ligne	\$\$ reconnaît la fin de la chaîne ou un caractère suivi d'un retour à la ligne
\$	Fin de chaîne	'Carpe diem' ~~ /arpe \$/; # faux 'Carpe diem' ~~ /diem \$/; # vrai

7-1-7 - Regroupements et captures

7-1-7-1 - Regroupements

Les parenthèses permettent de regrouper des éléments d'une regex :

```
/ a || b c / # reconnaît 'a' ou 'bc'
/ ( a || b ) c / # reconnaît 'ac' ou 'bc'
```

La même technique de regroupement peut s'appliquer aux quantificateurs :

```
/ a b+ / # Reconnait un 'a' suivi d'un ou plusieurs 'b'
/ ( a b )+ / # Reconnait une ou plusieurs séquences 'ab'
/ ( a || b )+ / # Reconnait une séquence quelconque de 'a' et de 'b' longue d'au moins un caractère
```


7-1-7-2 - Captures

Les parenthèses ne servent pas seulement à regrouper, elles servent aussi à capturer, c'est-à-dire qu'elles stockent la partie reconnue entre parenthèses dans une variable réutilisable ensuite, ainsi que sous la forme d'un élément de l'objet reconnu.

```
my $str = 'nombre 42';
if $str =~ /'nombre ' (\d+) / {
    say "Le nombre est $0";      # Le nombre est 42
    # ou
    say "Le nombre est $/[0]";   # Le nombre est 42
}
```

S'il y a plusieurs paires de parenthèses, elles sont numérotées de gauche à droite, en partant de zéro.

```
if 'abc' =~ /(a) b (c)/ {
    say "0: $0; 1: $1";        # 0: a; 1: c
}
```

Les variables \$0, \$1, etc. sont en fait des raccourcis. Ces captures sont canoniquement disponibles dans l'objet reconnu \$/ en utilisant celui-ci sous la forme d'une liste, si bien que \$0 est en fait du sucre syntaxique pour \$/[0], \$1 pour \$/[1] et ainsi de suite.

Forcer un contexte de liste à l'objet reconnu permet accéder facilement à tous les éléments :

```
if 'abcdef' =~ /(a) b (c) (d) e (f)/ {
    say $/.list.join: ', '
}                                     # -> a, c, d, f
```

7-1-7-3 - Regroupements sans capture

Les parenthèses assurent une double fonction : elles regroupent des éléments à l'intérieur de la regex et elles capturent ce qui a été reconnu dans la sous-regex entre les parenthèses.

Pour ne conserver que le comportement de regroupement (sans capturer), on peut utiliser des crochets au lieu de parenthèses :

```
if 'abc' =~ / [a|b] (c) / {
    say ~$0;                      # c
}
```

Si l'on n'a pas besoin d'une capture, utiliser des groupes non capturants présente trois avantages : l'intention du développeur est plus claire, il est plus facile de compter les groupes capturants dont on a besoin, et c'est un peu plus rapide.

7-1-7-4 - Captures nommées

Au lieu de numéroter les captures, il est possible de leur donner des noms. La façon générique (et un peu bavarde) de nommer des captures est la suivante :

```
if 'abc' =~ / $<mon_nom> = [ \w+ ] / {
    say ~$<mon_nom>             # abc
}
```

L'accès à une capture nommée, \$<mon_nom>, est en fait un raccourci pour accéder à l'objet reconnu sous la forme d'un hachage, autrement dit : \${ 'myname' } ou \$/<myname>.

Forcer l'objet reconnu dans un contexte de hachage donne un moyen d'accès simple à toutes les captures nommées :

```
if 'décompte=23' =~ / $<variable>=\w+ '=' $<valeur>=\w+ / {
    my %h = %/.hash;
    say %h.keys.sort.join: ', ';      # valeur, variable
    say %h.values.sort.join: ', ';    # 23, décompte
    for %h.kv -> $k, $v {
        say "Trouvé valeur '$v' avec la clef '$k'";
        # Affiche ces deux lignes :
        #   Trouvé valeur 'décompte' avec la clef 'variable'
        #   Trouvé valeur '23' avec la clef 'valeur'
    }
}
```

La section suivante (**Sous-règles ou règles nommées**) offre un moyen souvent plus pratique (et surtout plus puissant, comme nous le verrons plus loin) d'accéder aux captures nommées.

7-1-8 - Sous-règles ou règles nommées

Il est possible de mettre des morceaux de regex dans des sous-règles ou règles nommées, de même que l'on peut mettre des fragments de code dans une fonction (ou *subroutine*) ou une méthode.

```
my regex ligne { \N*\n }
if "abc\ndef" =~ /<ligne> def/ {
    say "Première ligne: ", $<ligne>.chomp;    # Première ligne: abc
}
```

Une regex nommée peut se déclarer avec la syntaxe `my regex nom_regex {corps de la regex }`, et appelée ensuite avec `<nom_regex>`. En outre, invoquer une regex nommée crée *ipso facto* une capture nommée portant le même nom (`$<ligne>` dans l'exemple ci-dessus).

Les regex nommées peuvent être regroupées en grammaires (voir § 7.2) et il est souvent souhaitable de le faire (l'objectif des regex nommées est précisément de construire des grammaires).

Il existe des sous-règles prédéfinies, correspondant plus ou moins aux classes de caractères vues antérieurement, par exemple :

- `ident` : un identifiant ;
- `upper` : un seul caractère capital ;
- `lower` : un seul caractère minuscule ;
- `alpha` : un seul caractère alphabétique ou un caractère souligné « `_` » (pour un caractère alphabétique Unicode sans le caractère souligné, utiliser `<:alpha>` ;
- `digit` : un seul chiffre ;
- `punct` : un seul caractère ponctuation ;
- `alnum` : un seul caractère alphanumérique (équivalent à `<+alpha+digit>`) ;
- `wb` : limite de mot, assertion de longueur nulle ;

Une liste plus complète se trouve dans le [chapitre sur les règles nommées de l'article](#) sur les regex (voir aussi la [Synopsis S05](#)).

7-1-9 - Adverbes

Les *adverbes* (qui correspondent à ce que l'on appelait *modificateurs* en Perl 5) modifient la façon dont fonctionnent les regex et permettent des raccourcis très pratiques pour certaines tâches répétitives.

Il y a deux sortes d'adverbes : les *adverbes de regex* s'appliquent là où la regex est définie et les *adverbes de reconnaissance* là où le motif reconnaît une chaîne.

7-1-9-1 - Adverbes de regex

Les adverbes qui apparaissent au moment de la déclaration d'une regex font partie intégrante de la regex et influent la façon dont le compilateur Perl 6 traduit la regex en code binaire.

Par exemple, l'adverbe `:ignorecase` ou `/i` (ignorer la casse) dit au compilateur d'ignorer les distinctions entre lettres capitales et minuscules. Ainsi, `'a' ~~ /A/` est faux, alors que `'a' ~~ /i A/` est reconnu avec succès.

Les adverbes de regex peuvent être placés avant ou à l'intérieur d'une déclaration de regex, et n'affectent que la partie de la regex qui vient ensuite, lexicalement.

Ces deux regex sont équivalentes :

```
my $rx1 = rx:i/a/;      # avant
my $rx2 = rx/:i a/;     # à l'intérieur
```

Mais celles-ci ne le sont pas :

```
my $rx3 = rx/a :i b/;   # insensible à la casse seulement pour b
my $rx4 = rx/:i a b/;   # complètement insensible à la casse
```

Les crochets et les parenthèses limitent la portée d'un adverbe :

```
/ (:i a b) c /          # reconnaît 'ABc' mais pas 'ABC'
/ [:i a b] c /          # reconnaît 'ABc' mais pas 'ABC'
```

7-1-9-1-1 - L'adverbe « ratchet » (pas de retour arrière)

L'adverbe `:ratchet` ou `/r` ordonne au moteur de regex de ne pas revenir en arrière (*backtrack*). Le mot anglais *ratchet* désigne un cliquet antiretour (comme dans une clef à cliquet), un système mécanique empêchant un dispositif de revenir en arrière (et le forçant donc, implicitement, à aller de l'avant).

Sans cet adverbe, différentes parties d'une regex vont essayer différentes façons de reconnaître une chaîne afin de permettre à d'autres parties de la regex de correspondre. Par exemple, avec la regex `'abc' ~~ /\w+ ./`, la partie `\w+` commence par consommer toute la chaîne, `abc`, puis échoue sur le « `.` » qui suit. Il y a alors un retour arrière (ou retour sur trace), c'est-à-dire que `\w+` abandonne le dernier caractère et ne reconnaît que `ab`, ce qui permet au « `.` » de reconnaître avec succès `c`. Ce processus consistant à abandonner un caractère pour recommencer un nouvel essai de reconnaissance s'appelle retour arrière (parfois retour sur trace) ou *backtracking*.

```
say so 'abc' ~~ / \w+ ./;   # Vrai
say so 'abc' ~~ / :r \w+ ./; # Faux
```

L'utilisation d'un tel « cliquet » (de l'adverbe *ratchet*) peut être une optimisation, car les retours arrière sont souvent coûteux. Mais l'intérêt est surtout que la reconnaissance sans retour arrière correspond étroitement à la façon dont les humains analysent un texte qu'ils lisent. Avec les regex `my regex identifiant { \w+ }` et `my regex keyword { if | else | endif }`, on attend intuitivement que l'identifiant absorbe un mot complet et n'ait pas besoin de restituer la fin de ce mot pour satisfaire la règle suivante. Par exemple, personne ne s'attend à ce que le mot `motif` soit analysé comme l'identifiant `mot` suivi du mot-clef `if` ; on attend plutôt que `motif` soit analysé comme un identifiant et, si l'analyseur attend le mot `if` à sa suite, qu'il échoue plutôt que d'analyser la donnée en entrée différemment de ce que l'on attend.

On peut considérer que le retour arrière est le comportement généralement recherché pour une analyse de bas niveau, caractère par caractère, d'une chaîne de caractères, mais que la recherche avec cliquet (*ratchet*) correspond généralement mieux à ce que l'on désire faire pour l'analyse lexicale ou syntaxique d'un texte structuré.

L'utilisation de la recherche avec cliquet est même si importante pour l'analyse lexicale ou syntaxique que Perl 6 définit un type de regex nommée particulier ayant implicitement la propriété ratchet : le *token* :

```
my token truc { .... }
# raccourci pour :
my regex truc { :r ... }
```

7-1-9-1-2 - L'adverbe sigspace (espaces blancs significatifs)

L'adverbe `:sigspace` ou `:s` rend les espaces blancs significatifs dans une regex (ils ne sont plus ignorés comme dans les exemples jusqu'ici) :

```
say so "J'ai utilisé Photoshop@" ~~ m:i/ photo shop /; # Vrai
say so "J'ai utilisé photo shop"  ~~ m:i:s/ photo shop /; # Vrai
say so "J'ai utilisé Photoshop@"  ~~ m:i:s/ photo shop /; # Faux
```

`m:s/ photo shop /` se comporte comme si l'on avait écrit `m/ photo <.ws> shop <.ws> /`. Par défaut, `<.ws>` assure que les mots sont séparés, si bien que 'a b' sera reconnu pas `<.ws>`, mais pas 'ab'.

Un espace dans une regex se transforme ou non en `<.ws>` selon ce qui précède l'espace. Dans l'exemple ci-dessus, l'espace au début de la regex ne se transforme pas en `<.ws>`, mais l'espace après les caractères le fait. D'une façon générale, la règle est que si un terme peut reconnaître quelque chose, alors un espace suivant ce terme est converti en `<.ws>`.

De même qu'une regex déclarée avec le mot-clef *token* implique l'adverbe `:ratchet`, une regex déclarée avec le mot-clef *rule* implique à la fois les adverbes `:ratchet` et `:sigspace`.

7-1-9-2 - Les adverbes de reconnaissance

Contrairement aux adverbes de regex (§ 7.1.9.1), qui sont liés à la déclaration d'une regex, les adverbes de reconnaissance n'ont un sens qu'au moment où l'on veut faire correspondre une chaîne et une regex.

Ils ne peuvent jamais figurer à l'intérieur d'une regex, mais seulement à l'extérieur de celles-ci, soit dans le cadre d'une reconnaissance `m/.../`, soit comme argument d'une méthode de reconnaissance.

On peut citer les adverbes de reconnaissances suivants (décrits en détail dans [le chapitre sur le sujet](#) de l'article sur les regex et grammaires) :

- `:continue` : permet de spécifier que la recherche doit commencer à la position suivant la fin de la reconnaissance précédente ;
- `:exhaustive` : trouve toutes les correspondances possibles d'une regex (y compris celles qui se chevauchent) ;
- `:global` ou `:g` : trouve chaque reconnaissance possible, mais sans chevauchement (la recherche d'une nouvelle reconnaissance commence à la fin de la reconnaissance précédemment trouvée) ; cet adverbe correspond à peu près au modificateur `g` de Perl 5 ;
- `:pos` ou `:p` : la recherche commence à la position spécifiée dans la chaîne.

7-1-10 - Regarder devant et derrière (assertions)

Les assertions permettent de rechercher des correspondances vers l'avant ou vers l'arrière, mais sans consommer la chaîne cible (en restant à la même position), comme le font les ancres.

7-1-10-1 - Assertions avant

Pour vérifier si un motif apparaît avant un autre motif, on peut utiliser l'assertion avant `before`, de la forme suivante : `<?before motif>`.

Ainsi, pour rechercher la chaîne `toto` immédiatement suivie de la chaîne `titi`, on peut utiliser la regex `rx{ toto <?before titi> }`, par exemple comme suit :

```
say "tototiti" ~~ rx{ toto <?before titi> }; # -> toto
```

Si l'on souhaite au contraire rechercher un motif qui ne soit pas immédiatement suivi par un autre motif, il faut utiliser une assertion avant négative, de la forme suivante : `< !before motif>`.

Par exemple, pour rechercher la chaîne `toto` non immédiatement suivie de la chaîne `titi`, on peut utiliser la regex `rx{ toto < !before titi> }`.

7-1-10-2 - Assertions arrière

Pour vérifier si un motif apparaît après un autre motif, on peut utiliser l'assertion arrière `after`, de la forme suivante : `<?after motif>`, équivalente en sens inverse de l'assertion avant. De même, il existe une assertion arrière négative, de la forme suivante : `< !after motif>`.

7-2 - Les grammaires

Les grammaires sont un outil puissant pour décomposer un texte en éléments individuels et, souvent, renvoyer les structures de données qui ont été créées en interprétant ce texte.

Par exemple, un programme Perl 6 est interprété et exécuté en utilisant une grammaire Perl 6.

Un exemple de portée plus pratique pour un utilisateur courant de Perl 6 est le module **JSON::Tiny** de Perl 6, qui peut désérialiser n'importe quel fichier JSON valide. Le code qui effectue cette désérialisation est écrit en moins de 100 lignes de code simple et facile à étendre.

Les grammaires Perl 6 sont en fait un simple moyen de regrouper des regex, de même que les classes permettent de regrouper des méthodes de code ordinaire (et l'analogie va beaucoup plus loin qu'on pourrait le croire de prime abord, comme on le verra plus loin).

7-2-1 - Les « briques » de construction d'une grammaire

Une grosse partie de ce qu'il y a besoin de savoir et de comprendre pour écrire une grammaire a déjà été vu ci-dessus, en particulier au chapitre **7.1.8 Sous-règles ou règles nommées**, ainsi qu'aux sous-chapitres **7.1.9.1.1 L'adverbe « ratchet » (pas de retour arrière)** et **7.1.9.1.2 L'adverbe sigspace (espaces blancs significatifs)** ci-dessus. Ces sous-règles ou règles nommées constituent les briques élémentaires d'une grammaire. En fait, l'un des principaux buts d'une grammaire est de regrouper des sous-règles ou regex nommées (de types `regex`, `token` et `rule`) dans un espace de noms bien délimité afin d'éviter les collisions de noms d'identifiants avec d'autres regex ailleurs dans le code.

Rappelons que les règles nommées `regex`, `token` et `rule` sont des entités très semblables servant à déclarer une regex nommée sous une forme ressemblant à la définition d'une fonction ou d'une méthode. Dans ce chapitre, nous les appellerons désormais collectivement *règles*, indépendamment du mot-clef utilisé pour les déclarer. On les déclare de la façon suivante :

```
regex ma_regex { ... } # regex ordinaire
```

```
token mon_token { ... } # regex avec adverbe :ratchet implicite
rule ma_règle { ... } # regex avec :ratchet et :sigspace implicites
```

Pour mémoire :

- Le mot-clef `regex` signale une regex ordinaire ;
- Le mot-clef `token` implique l'adverbe `:ratchet` (« cliquet ») implicite, c'est-à-dire que ce genre de règle ne fait pas de retour arrière (pas de *backtracking*) ;
- Le mot-clef `rule` implique implicitement les adverbes `:ratchet` (pas de retour arrière) et `:sigspace` (les espaces du motif ne sont pas ignorés).

Voici un exemple dans lequel on définit une première règle `chiffres`, et l'utilise pour en définir une seconde, `décimal` :

```
my regex chiffres { \d+ }
my regex decimal { <chiffres> \. <chiffres> }
say so " Cet objet coûte 13.45 euros" ~~ /<décimal>; # -> True
# (so renvoie une évaluation booléenne, donc, en fait True ou False)
say ~$/; # -> 13.45
```

On voit ci-dessus qu'une règle peut appeler une autre règle, de même qu'une fonction peut appeler une autre fonction. Une règle peut aussi s'appeler elle-même récursivement. Ce mécanisme capital est le cœur de la puissance des regex de Perl 6 et de leur capacité à créer des grammaires.

7-2-2 - Créer une grammaire

Une grammaire crée un espace de noms propre et s'introduit avec le mot-clef `grammar`.

7-2-2-1 - Syntaxe de définition d'une grammaire

De même qu'une classe peut regrouper des actions (appelées méthodes), une grammaire regroupe des règles nommées :

```
grammar Identité {
  rule nom { Nom '=' (\N+) } # chaîne de caractères quelconques
                                # autres que des retours à la ligne
  rule adresse { Adr '=' (\N+) } # idem
  rule âge { Age '=' (\d+) } # des chiffres

  rule desc {
    <nom> \n
    <âge> \n
    <adresse> \n
  }

  # etc.
}
```

La règle `desc` est ici définie en utilisant d'autres règles (`nom`, `âge` et `adresse`) définies par ailleurs. Cela permet de construire progressivement des niveaux d'abstraction de plus en plus élevés.

Ceci crée un objet de type `Grammar` dont le type dénote le langage en cours d'analyse et dont on peut dériver d'autres grammaires sous la forme de langages étendus.

À remarquer ici qu'il n'est plus nécessaire de déclarer les règles avec l'opérateur `my` comme cela avait été fait jusqu'à présent, parce la grammaire crée l'espace de noms et la portée lexicale nécessaires.

7-2-2-2 - Héritage de grammaires

Une grammaire peut hériter d'une autre grammaire (parente) de la même façon qu'une classe peut hériter d'une autre classe :

```
grammar Message {
    rule texte { <salutation> $<corps>=<ligne>+? <fin> }
    rule salutation { [Salut|Bonjour] $<dest>=\S+? ', ' }
    rule fin { [à|'@'] plus ', ' $<auteur>=.+ }
    token ligne { \N* \n }
}

grammar MessageFormel is Message {
    rule salutation { [Cher|Chère] $<dest>=\S+? ', ' }
    rule fin { Bien cordialement ', ' $<auteur>=.+ }
}
```

Ici, la grammaire `MessageFormel` hérite de la grammaire parente `Message`. Comme pour les méthodes d'une classe, les règles sont héritées de la grammaire parente (et polymorphiques), il n'y a donc pas besoin de définir à nouveau les règles `texte` et `ligne` qui ne changent pas. On ne surcharge que les règles qui changent.

Toutes les grammaires dérivent de la classe `Grammar`, qui fournit entre autres les méthodes communes à toutes les grammaires, comme `.parse` et `.fileparse`, décrites ci-dessous.

La capacité des grammaires d'hériter d'autres grammaires est un instrument extrêmement puissant et un facteur essentiel permettant l'extensibilité du langage Perl 6.

7-2-3 - Utiliser une grammaire

Il est possible d'analyser une chaîne de caractères avec une grammaire en appelant la méthode `.parse` sur cette grammaire et en passant optionnellement en paramètre un *objet d'actions* (voir § XXXX ci-dessous). De même, la méthode `.parsefile` permet d'analyser un fichier.

```
MaGrammaire.parse($chaîne, :actions($objet-action))
MaGrammar.parsefile($nom-fic, :actions($objet-action))
```

Les méthodes `.parse` et `.parsefile` sont ancrées au début et à la fin du texte, et échouent si la fin du texte n'est pas atteinte.

En principe, il ne faut utiliser une grammaire que pour effectuer l'analyse lexicale et syntaxique proprement dite du texte. Pour extraire des données complexes, il est recommandé d'utiliser un objet d'actions en conjonction avec la grammaire.

7-2-4 - Les classes et objets d'actions

7-2-4-1 - Exécuter du code lors d'une reconnaissance

Quand une grammaire analyse avec succès un texte, elle renvoie un arbre syntaxique d'objets reconnus. Plus cet arbre est profond (et il le devient souvent très rapidement), et plus il y a de branches dans cet arbre, plus il devient difficile d'explorer cet arbre pour y trouver l'information que l'on recherche.

Pour éviter de devoir procéder à cette exploration de l'arbre des reconnaissances, on peut fournir un *objet d'action*. Après chaque analyse réussie d'une règle nommée de la grammaire, celle-ci cherche à invoquer une méthode de cet objet d'action portant le même nom que la règle, en lui fournissant en argument positionnel l'objet reconnu qui vient d'être créé. Cette méthode, si elle existe, peut notamment servir à construire un arbre syntaxique abstrait (*Astract*

Syntax Tree ou AST) ou à faire toutes sortes d'autres choses dont on pourrait avoir besoin pour la suite. Si cette méthode n'existe pas, cette étape est simplement ignorée.

Voici un exemple minimaliste et assez artificiel d'une grammaire et d'actions travaillant de concert :

```
#!/usr/bin/perl6
use v6;

grammar GrammaireTest {
    token TOP { ^ \d+ $ }
}

class ActionsTest {
    method TOP($/) {
        $/.make(2 + ~$/.);
    }
}

my $actions = ActionsTest.new;
my $reconnu = GrammaireTest.parse('40', :$actions);
say $reconnu;           # -> #40#
say $reconnu.made;      # -> 42
```

L'objet `$actions` de la classe `Actionstest` est instancié puis passé en argument lors de l'appel de la méthode `.parse`. Quand la règle `TOP` reconnaît l'argument, la grammaire appelle automatiquement la méthode `TOP` en lui passant l'objet reconnu en argument.

La méthode `make` de la classe `Match` alimente la structure `$/made` (l'utilisateur décide son contenu, mais ce sera souvent un arbre syntaxique abstrait) avec son argument.

7-2-4-2 - Autres façons d'exécuter du code dans une grammaire

Dans les exemples ci-dessus, les méthodes-actions sont définies dans une classe d'actions distincte de la grammaire proprement dite, et c'est généralement la voie à suivre dans toute grammaire un tant soit peu étoffée.

Pour des cas simples, il est toutefois également possible de définir des méthodes au sein même de la grammaire :

```
grammar toto {
    regex titi { <.configurer> blah blah }
    method configurer {
        # faire quelque chose ici
    }
}
```

Oui, les grammaires peuvent définir des méthodes (et elles peuvent même utiliser des rôles), ce sont vraiment des classes...

Il est également possible d'exécuter du code au sein même d'une règle en l'insérant entre des accolades :

```
grammar toto {
    regex titi { blah blah { say "Je suis arrivé ici" } blah blah }
}
```

Si la portion du motif qui précède le bloc de code est reconnue, alors ce bloc est immédiatement exécuté.

7-2-5 - Un exemple simple de grammaire : validation de noms de modules

L'objectif de cet exemple de grammaire est de valider un nom de module Perl.

Le nom d'un module Perl peut se décomposer en identifiants séparés par des paires de caractères deux-points : « :: », par exemple **List::Util** ou **List::MoreUtils** (les exemples de noms de modules fournis ici sont des modules Perl 5). Un identifiant doit commencer par un caractère alphabétique ou un caractère souligné, suivi de zéro, un ou plusieurs caractères alphanumériques.

Rien de bien complexe jusqu'ici, mais ceci se complique quelque peu du fait que certains modules ont un seul identifiant (**Memoize**), et donc pas de caractères deux-points, et que d'autres peuvent avoir des noms « à rallonge » : **Regexp::Common::Email::Address**.

7-2-5-1 - La grammaire de validation

Il suffit, par exemple, de définir une règle identifiant garantissant les règles de nommage ci-dessus et une règle séparateur, et de les combiner adéquatement dans une grammaire.

```
grammar Valide-Nom-Module {
    token TOP { ^ <identifiant> [ <séparateur> <identifiant> ]* $ }
    token identifiant {
        <[A..Za..z_]>      # 'mot' commençant par un caractère
                           # alphabétique ou un caractère souligné
        <[A..Za..z0..9]>*  # 0 ou plusieurs caractères alphanumériques
    }
    rule séparateur { '::' } # paire de caractères deux-points
}
```

On peut maintenant tester cette grammaire avec quelques noms de modules valides ou non :

```
for <Super:Nouveau::Module Super.Nouveau.Module
    Super::6ouveau::Module Super::Nouveau::Module> -> $nom {
    my $reconnu = Valide-Nom-Module.parse($nom);
    say "nom\t", $reconnu ?? $reconnu !! "Nom de module invalide";
}
```

Ce qui affiche :

```
Super:Nouveau::Module  Nom de module invalide
Super.Nouveau.Module   Nom de module invalide
Super::6ouveau::Module Nom de module invalide
Super::Nouveau::Module #Super::Nouveau::Module#
identifiant => #Super#
séparateur  => #::#
identifiant => #Nouveau#
séparateur  => #::#
identifiant => #Module#
```

Seul le nom de module valide a été reconnu, les trois autres ont été rejetés à juste titre.

Parfois, les noms de modules sont résumés en remplaçant les paires de caractères deux-points par des tirets. Par exemple, le nom officiel est **Regexp::Common::Email::Address** et peut aussi s'écrire **Regexp-Common-Email-Address**. Si l'on désire valider cette seconde écriture, il suffit de modifier le séparateur pour qu'il autorise également un tiret :

```
rule séparateur { '::' || '-' } # deux car. deux-points ou tiret
```

La grammaire ainsi modifiée reconnaît maintenant aussi le nom « Super-Nouveau-Module ».

Il a suffi de modifier la règle `séparateur` pour que la modification se propage à toute la grammaire, jusqu'à la règle `TOP`.

7-2-5-2 - Ajout d'un objet d'actions

La grammaire ci-dessus peut déterminer si un nom de module Perl est valide ou non.

On désire maintenant ajouter un avertissement si le nom du module est trop long (plus de 5 identifiants) ; dans ce cas, le nom du module sera toujours valide, mais l'on pourra peut-être conseiller à l'auteur du module d'essayer de choisir un nom plus court.

Il suffit par exemple d'ajouter une classe d'actions `Valide-Nom-Module-Actions` définie comme suit :

```
class Valide-Nom-Module-Actions {
    method TOP($/) {
        if $<identifiant>.elems > 5 {
            warn "Nom de module très long! Peut-être le réduire ?\n"
        }
    }
}
```

La définition de la classe n'a rien de particulier, c'est une classe Perl 6 ordinaire. La particularité importante est que la seule méthode définie ici a le même nom qu'une des règles de la grammaire (en l'occurrence, la règle d'entrée dans la grammaire, `TOP`). L'avertissement sera envoyé si le nombre d'identifiants dépasse 5, mais cela n'empêchera pas de valider le nom du module.

La syntaxe d'appel de la grammaire est modifiée comme suit :

```
my $reconnu = Valide-Nom-Module.parse($nom, :actions(Valide-Nom-Module-Actions));
```

Le résultat est le même que précédemment si l'on appelle la grammaire avec le nom de module « `Super::Nouveau::Module` » (ou « `Super-Nouveau-Module` »), ce qui est rassurant.

Mais avec un nom de module à la Mary Poppins :

```
my $nom = "Mon::Module::Super::Cali::Fragi::Listi::Cexpi::Delilicieux";
my $reconnu = Valide-Nom-Module.parse($nom, :actions(Valide-Nom-Module-Actions));
say $reconnu if $reconnu;
```

on obtient l'avertissement :

Nom de module très long ! Peut-être le réduire ?

```
> perl6 grammaire_nom_module.pl
in method TOP at perl6_grammaire_module.pl:15
#Mon::Module::Super::Cali::Fragi::Listi::Cexpi::Delilicieux#
identifiant => #Mon#
séparateur => #::#
identifiant => #Module#
(...)
identifiant => #Delilicieux#
```

suivi de l'affichage de l'objet reconnu.

Cet exemple de validation d'un nom de module est très librement inspiré d'une idée provenant de l'article [How to create a grammar in Perl 6](#) de David Farrell.

7-2-5-3 - Autres exemples de grammaires

Les documents en français ci-dessous fournissent d'autres exemples de construction détaillée de grammaires un peu plus complexes (mais restant assez simples) :


- Une  **grammaire pour analyser du JSON** ;
- Une  **grammaire pour analyser du (pseudo) XML**.
-  **Reconnaître une URL**.

Dans un genre nettement plus ambitieux, le lecteur intéressé pourra aussi consulter la  **Grammaire de Perl 6**.

7-2-6 - Héritage, grammaires mutables et perspectives

La possibilité d'hériter d'une grammaire offre une puissance d'expression insoupçonnée et des perspectives immenses : il est possible, par exemple dans le cadre d'un module, d'écrire une « sous-grammaire » ou grammaire fille de la grammaire de Perl 6 afin de surcharger un opérateur, d'ajouter une fonctionnalité ou même de modifier un élément de la syntaxe, et de faire tourner un programme Perl avec le même compilateur Perl 6 utilisant cette syntaxe localement modifiée.

C'est grâce à ce mécanisme sous-jacent que la syntaxe de Perl 6 est dynamique et, par exemple, qu'il est facile de définir ses propres opérateurs, comme cela a été décrit au § [5 Créer de nouveaux opérateurs](#).

Il est même possible d'effectuer des modifications de bas niveau de la grammaire de Perl 6 pour aller plus loin et étendre encore plus le langage (voir  [Modifications de la grammaire et extensibilité du langage](#)).

Les possibilités d'extension sont réellement extraordinaires et stupéfiantes et font de Perl 6 un langage profondément malléable susceptible de rester à la pointe de la technologie des langages pour des décennies.

8 - Multitâche, parallélisme, concurrence et programmation asynchrone

Comme beaucoup de langages modernes, Perl 6 a été conçu pour permettre le parallélisme, c'est-à-dire l'exécution de plusieurs choses en même temps, et la programmation asynchrone (parfois appelée programmation événementielle), c'est-à-dire qu'un événement ou un changement dans une partie d'un programme peut conduire à un autre événement ou changement dans une autre partie d'un programme, de façon asynchrone par rapport au flot normal d'exécution d'un programme.

Perl cherche à offrir une interface modulaire et cohérente de haut niveau au parallélisme, indépendante de la façon dont une machine virtuelle pourrait mettre en œuvre cette interface pour un système d'exploitation donné, au moyen de couches de fonctionnalités décrites ci-dessous.

De plus, certaines fonctionnalités de Perl peuvent opérer implicitement de façon asynchrone. Aussi, pour permettre une interopérabilité maîtrisée et prévisible de ces fonctionnalités, le code utilisateur devrait, dans la mesure du possible, utiliser ces interfaces de haut niveau et éviter d'utiliser les interfaces de bas niveau (*threads*, ordonnanceurs, verrous, etc.). Outre le fait qu'ils ne fonctionnent pas de la même façon dans les différents systèmes d'exploitation ou environnements d'exécution, ces mécanismes de bas niveau sont notoirement très délicats à programmer et à maîtriser dans la plupart des langages de programmation qui les mettent en œuvre (et les *threads* de Perl 5 ne font pas exception).

8-1 - Interface de haut niveau

8-1-1 - Les promesses (objets de type « Promise »)

Une *promesse* (Promise) représente le résultat d'un calcul qui n'est peut-être pas terminé (voire n'est même pas commencé) au moment où la promesse est obtenue. Les promesses offrent l'essentiel de ce dont a besoin du code utilisateur pour opérer de façon parallèle ou asynchrone.

Un programme peut tenir (keep) sa promesse ou la rompre (break). La promesse peut avoir les statuts (status) « planifié » (Planned), tenu (Kept) ou rompu (Broken).

```
my $p1 = Promise.new;
say $p1.status;           # -> Planned
$p1.keep('résultat');
say $p1.status;           # -> Kept
say $p1.result;           # -> résultat

my $p2 = Promise.new;
$p2.break('Oh non, pas vrai');
say $p2.status;           # -> Broken
say $p2.result;           # -> "Oh non, pas vrai"
```

Une bonne partie de la puissance des promesses provient du fait qu'il est possible de les combiner, par exemple de les enchaîner :

```
my $promesse1 = Promise.new();
my $promesse2 = $promesse1.then(
    -> $v { say $v.result; "Second Résultat" }
);
$promesse1.keep("Premier Résultat");
say $promesse2.result;    # Premier Résultat \n Second Résultat
```

Ici, le then ordonnance le code (de la promesse 2) en sorte qu'il s'exécute quand la promesse 1 sera tenue ou rompue et renvoie à son tour une nouvelle promesse qui sera elle-même tenue (ou rompue si le code échoue). L'instruction keep fait passer le statut de la première promesse à tenu (Kept), ce qui autorise le démarrage de la seconde promesse.

Il est possible d'ordonner une promesse pour qu'elle démarre plus tard :

```
my $promesse1 = Promise.in(5);
my $promesse2 = $promesse1.then(-> $v { say $v.status; 'Second Résultat' });
say $promesse2.result;
```

La méthode in crée une nouvelle promesse et ordonnance une nouvelle tâche chargée d'appeler keep au plus tôt dans le nombre de secondes spécifié dans l'argument, retournant un nouvel objet de type promesse.

Une utilisation fréquente des promesses est de lancer un fragment de code et de tenir la promesse quand il se termine avec succès, ou de la rompre s'il échoue :

```
my $promesse = Promise.start(
    { my $i = 0; for 1 .. 10 { $i += $_ }; $i }
);
say $promesse.result;    # 55
```

Le résultat de la promesse est ici la valeur renvoyée par le bloc de code. Si le bloc de code renvoie une erreur (la promesse est donc rompue), alors la méthode cause renvoie l'objet de type Exception qui a été généré.

```
my $promesse = Promise.start({ die "Promesse rompue" });
try $promesse.result;
say $promesse.cause;     # -> Promesse rompue in block ...
```

On a besoin de ce genre de construction suffisamment souvent pour qu'il ait été jugé utile d'offrir également une syntaxe fonctionnelle :

```
my $promesse = start {
    my $i = 0;
    for 1 .. 10 {
        $i += $_
    };
};
```

```

    $i
}
my $résultat = await $promesse;
say $résultat;      # 55

```

Appeler la fonction `await` équivaut presque à invoquer la méthode `result` sur l'objet promesse renvoyé par `start`, sauf qu'elle peut prendre une liste de promesses et renvoyer le résultat de chacune d'elles :

```

my $p1 = start {
    my $i = 0;
    for 1 .. 10 {
        $i += $_
    };
    $i
};
my $p2 = start {
    my $i = 0;
    for 1 .. 10 {
        $i -= $_
    };
    $i
};
my @résultat = await $p1, $p2;
say @résultat;      # 55 -55

```

8-1-2 - Les fournisseurs (objets de type « Supply »)

Un *fournisseur* (Supply) est un mécanisme de flux de donnée asynchrone qui peut être consommé simultanément par un ou plusieurs consommateurs d'une manière analogue aux événements dans d'autres langages de programmation. On peut s'en servir pour mettre en place des schémas de programmation événementielle.

Dans sa forme la plus élémentaire, un fournisseur est un flux de messages créés avec la méthode `Emit` et auquel des tâches peuvent s'abonner en invoquant la méthode `Tap`.

```

my $fournisseur = Supply.new();

$fournisseur.tap( -> $v { say $v });

for 1 .. 10 {
    $fournisseur.emit($_);
}

```

La méthode `tap` renvoie un objet de type `Tap` (« robinet », « vanne ») qui peut être fermé avec la méthode `close` lorsque l'on n'est plus intéressé par les événements.

Il est possible de filtrer ou de transformer un objet de type `Supply` à l'aide des méthodes `grep` et `map`, de même que ces opérateurs peuvent filtrer ou transformer une liste. Dans ce cas, `grep` renvoie un nouveau fournisseur tel que seuls les événements pour lesquels la condition du `grep` a renvoyé vrai sont émis en aval, et `map` renvoie un nouveau fournisseur émettant les événements transformés.

8-1-3 - Les canaux (objets de type « Channel »)

Un *canal* (Channel) est une queue ou file d'attente qui peut avoir plusieurs lecteurs et plusieurs rédacteurs, analogue à une file *FIFO* (premier entré, premier sorti) qui autoriserait les communications interprocessus. Il s'agit véritablement d'une file d'attente en ce sens que chaque élément envoyé dans le canal ne sera disponible qu'à un seul lecteur (celui qui aura lu le canal le premier). Si l'on désire que plusieurs lecteurs puissent recevoir le même élément, alors il faut utiliser un fournisseur.

Un élément est posté dans un canal au moyen de la méthode `send`, et la méthode `receive` extrait un élément de la queue (ou se bloque jusqu'à ce qu'un nouvel élément soit envoyé si la queue est vide).

```
my $canal = Channel.new;
$canal.send('Canal Un');
say $canal.receive; # 'Canal Un'
```

Si un canal a été fermé avec la méthode `close`, alors toute tentative d'envoi avec la méthode `send` provoquera une exception de type `X::Channel::SendOnClosed`, et un `receive` sur ce canal provoquera une exception de type `X::Channel::ReceiveOnClosed` s'il n'y a plus d'éléments dans la queue.

8-1-4 - Processus asynchrones (objets de type « `Proc::Asyn` »)

Le module `Proc::Asyn` utilise les fonctionnalités présentées (promesses, fournisseurs, etc.) ci-dessus pour interagir avec un programme extérieur.

```
my $proc = Proc::Asyn.new('echo', 'toto', 'titi');

$proc.stdout.tap(-> $v { print "Sortie: $v" });
$proc.stderr.tap(-> $v { print "Erreur: $v" });

say "Démarre...";
my $promesse = $proc.start;

await $promesse;
say "Fin.";

# Sortie:
# Démarre...
# Sortie: toto titi
# Fin.
```

Le chemin du programme externe et les éventuels arguments sont fournis au constructeur, mais le programme ne démarrera que lors de l'exécution de l'instruction `start`, qui renvoie une promesse qui sera tenue à la fin de l'exécution du programme externe. Les sorties standard et d'erreur du programme sont disponibles sous la forme d'objets de type `Supply` au moyen des méthodes `stdout` et `strerr` sur lesquelles on peut se brancher par une instruction `tap`.

Pour écrire sur l'entrée standard du programme externe, il faut fournir l'adverbe `!w` au constructeur et utiliser les méthodes `write`, `print` ou `say`.

8-2 - Interface de bas niveau

8-2-1 - Les threads

L'interface de bas niveau pour la concurrence est le `Thread`. Un *thread* peut être considéré comme un bout de code qui sera peut-être finalement exécuté sur un processeur, mais les détails de cette exécution dépendent presque exclusivement de la machine virtuelle ou du système d'exploitation.

Il est possible de créer un *thread* et de l'exécuter plus tard :

```
my $thread = Thread.new(code => { for 1 .. 10 -> $v { say $v } });
# ...
$thread.run;
```

Ou de le créer et de le lancer dans la foulée :

```
my $thread = Thread.start({ for 1 .. 10 -> $v { say $v } });
```

Dans les deux cas, on peut attendre la fin de l'exécution du code du *thread* avec la méthode `finish` qui est bloquante tant que le *thread* n'est pas terminé.

```
$thread.finish;
```

Il n'existe pas d'autre moyen de synchronisation ou de partage des ressources, et c'est la principale raison pour laquelle les *threads* ne trouvent généralement pas leur place sous cette forme de l'interface de bas niveau dans du code utilisateur.

8-2-2 - Ordonnanceurs (objets ayant le rôle « Scheduler »)

Les ordonnanceurs sont une API fournie par des classes qui peuvent assurer le rôle *Scheduler*. Le but de cette interface est de déterminer quelles ressources utiliser pour exécuter une tâche donnée et quand. Les API de concurrence de haut niveau sont construites pour la plupart sur un ordonnanceur, si bien que du code utilisateur n'a généralement pas à s'en préoccuper, mais certaines méthodes des classes *Proc::Async*, *Promise* et *Supply* permettent de fournir explicitement un ordonnanceur.

Actuellement, l'ordonnanceur global par défaut est disponible dans la variable `*$SCHEDULER`.

L'interface de base d'un ordonnanceur est la méthode *cue* (en anglais, le mot *cue* désigne la phrase finale de la tirade d'un acteur qui constitue le signal d'une réplique pour un autre comédien) :

```
method cue(&code, Instant :$at, :$in, :$every, :$times = 1; &catch)
```

Ceci va ordonnancer `&code` de la manière déterminée par les adverbes passés en paramètre en utilisant le schéma d'exécution mis en œuvre par l'ordonnanceur. Par exemple :

```
my $i = 0;
my $annulation = $*SCHEDULER.cue({ say $i++; }, every => 2 );
sleep 20;
```

Si la variable `*$SCHEDULER` contient bien les variables par défaut, ceci va imprimer les nombres 0 à 10 toutes les dix secondes approximativement (en fonction des tolérances d'ordonnancement du système d'exploitation). Ici, le code sera ordonné pour s'exécuter jusqu'à la fin ; cependant, la méthode *cue* renvoie un objet de type *Cancellation* (ici nommé `$annulation`) qui peut être utilisé pour annuler l'exécution avant la fin normale en y appliquant la méthode *cancel*.

Même si l'interface *Scheduler* offre des fonctionnalités un peu plus riches que celles des *Threads*, toutes les fonctionnalités sont disponibles dans les API de haut niveau, et il ne devrait en général pas y avoir de raison d'utiliser des ordonnanceurs dans du code utilisateur.

8-2-3 - Verrous (objets de type « Lock »)

La classe *Lock* fournit un mécanisme de bas niveau pour protéger les données partagées dans un environnement concurrent et permet d'établir une « programmation à fil sécurisé » (*thread-safe programming*) dans les API de haut niveau. Ces verrous correspondent à ce que l'on appelle parfois des *Mutex* (verrous d'exclusion mutuelle) dans d'autres langages de programmation. Comme les classes de haut niveau *Channel*, *Promise* et *Supply* mettent en place ce genre de verrous lorsque c'est nécessaire, le code utilisateur n'en aura sans doute presque jamais besoin..

La principale interface de la classe *Lock* est la méthode *protect* qui assure qu'un bloc de code (généralement appelé *section critique*) n'est exécuté que dans un *thread* à la fois :

```
my $verrou = Lock.new;
my $a = 0;
await (^10).map: {
  start {
    $verrou.protect({
      my $r = rand;
```

```


        sleep $r;
        $a++;
    });
}
say $a; # 10

```

Comme `protect` va bloquer tout *thread* attendant d'entrer dans la section critique, il convient de faire en sorte que celle-ci soit aussi brève que possible.

8-3 - Parallélisme, asynchronisme et concurrence

Il y a beaucoup de confusion entre les termes parallélisme, asynchronisme et concurrence. Ce sont des notions différentes, on les utilise dans des contextes différents, et ils nécessitent des solutions distinctes. Il importe de bien comprendre ces distinctions pour concevoir des solutions multiprocesseurs adaptées et efficaces.

Le texte de la présente section **8.3 Parallélisme, asynchronisme et concurrence** est dans une très large mesure inspiré d'une  **série de diapositives** de Jonathan Worthington, chef de l'équipe de développement de Rakudo Perl 6, créateur et architecte de MoarVM.

8-3-1 - Parallélisme

Pendant des décennies, on a accru la vitesse des processeurs en réduisant la taille des composants afin que, ceux-ci étant de plus en plus proches, ils communiquent plus rapidement entre eux. Nous en sommes à faire des transistors ne comportant que quelques dizaines d'atomes. On se rapproche d'une limite physique difficile à franchir. La solution retenue depuis quelques années consiste à mettre plusieurs processeurs dans la même machine ou, ce qui revient essentiellement au même, à créer des processeurs multicœurs. Encore faut-il exploiter ces CPU ou cœurs multiples.

Le *parallélisme* consiste à décider de faire plusieurs choses en même temps, dans l'espoir d'aboutir plus rapidement à la solution. C'est une *décision* de conception : le parallélisme ne découle pas du problème à résoudre, mais de la solution que l'on décide d'y appliquer.

Soit le problème de comparer deux fichiers JSON. On peut écrire un programme comme celui-ci :

```

use JSON::Tiny;
sub MAIN($fic1, $fic2) {
    my $arbre1 = from-json(slurp($fic1));
    my $arbre2 = from-json(slurp($fic2));
    say $arbre1 eqv $arbre2
        ?? 'Les fichiers contiennent du JSON identique'
        !! 'Les fichiers contiennent du JSON différent';
}

```

Ce programme effectue séquentiellement plusieurs tâches, dont deux au moins (les deux tâches `from-json` d'analyse syntaxique des fichiers JSON) n'ont aucune dépendance entre elles. On devrait donc pouvoir gagner du temps en les exécutant en parallèle, mais cela peut poser quelques questions :

- comment décider du bon nombre de tâches à exécuter en parallèle ?
- comment attendre correctement la fin des tâches et obtenir des résultats corrects ?
- comment gérer les exceptions avec des tâches en parallèle ?

8-3-1-1 - Promesses et parallélisme de tâches

Les *promesses* (promises) de Perl 6 offrent une solution à ce problème :

```

use JSON::Tiny;
sub MAIN($fic1, $fic2) {

```



```
my $analyse1 = start from-json(slurp($fic1));
my $analyse2 = start from-json(slurp($fic2));
my ($arbre1, $arbre2) = await $analyse1, $analyse2 ;
say $arbre1 eqv $arbre2
    ?? 'Les fichiers contiennent du JSON identique'
    !! 'Les fichiers contiennent du JSON différent';
}
```

Le mot-clef `start` ordonnance la tâche sur les *threads* disponibles. Le choix du nombre de « travailleurs » se fera en fonction du matériel, de la mémoire, etc. L'instruction `await` peut attendre un nombre quelconque de promesses. L'attente est efficace et ne consomme pratiquement pas de ressources. Si l'une des tâches génère une exception, celle-ci est automatiquement retransmise à la tâche en attente.

C'est du *parallélisme de tâches* : on détecte à l'analyse des tâches indépendantes et l'on s'arrange pour les exécuter en parallèle.

8-3-1-2 - Parallélisme de données

Voici un cas un peu plus complexe : paralléliser un programme qui doit analyser des milliers de fichiers de données provenant de toutes les stations météo d'un réseau mondial, filtrer ceux qui sont relatifs à l'Europe et déterminer le lieu où la température est la plus élevée à un moment donné en Europe. Le programme pourrait ressembler à ceci :

```
sub MAIN($data-dir) {
    my $noms-fic = dir($data-dir);
    my $data     = $noms-fic.map(&slurp);
    my $analysés = $data.map(&analyse-données-météo);
    my $européen = $analysés.grep(*.continent eq 'Europe');
    my $max      = $européen.max(by => *.temp-moyenne);
    say "$max.place() est le plus chaud!";
}
# Fonctions accessoires omises
```

Cette approche est séquentielle : on prend chaque fichier l'un après l'autre pour le faire passer dans un pipeline de données : chargement en mémoire, analyse syntaxique du JSON, analyse des données météorologiques, filtrage sur le continent. Le modèle de programmation par flux de données (pipeline de données) employé ici se prête généralement bien au parallélisme.

Ici, on recherche du *parallélisme de données*. Il y a de très nombreuses données et l'on désire leur appliquer le même traitement. Le but est donc de partitionner les données. On désire donc exécuter le pipeline de données en parallèle sur plusieurs *threads* puis collecter les résultats. Mais comment distribuer le travail puis recueillir les résultats de façon fiable ? Comment gérer les exceptions ? Que faire s'il est important que l'ordre des résultats reflète celui des données en entrée ?

8-3-1-3 - Méthodes `race` et `hyper` pour paralléliser le pipeline

La méthode `race` (« course ») permet de paralléliser le pipeline de traitement :

```
sub MAIN($data-dir) {
    my $noms-fic = dir($data-dir).race(batch => 10);
    my $data     = $noms-fic.map(&slurp);
    my $analysés = $data.map(&analyse-données-météo);
    my $européen = $analysés.grep(*.continent eq 'Europe');
    my $max      = $européen.max(by => *.temp-moyenne);
    say "$max.place() est le plus chaud!";
}
# Fonctions accessoires omises
```

L'invocation à la deuxième ligne ci-dessus de la méthode `race` avec le paramètre nommé `batch => 10` oblige le pipeline à s'exécuter en parallèle. Dès qu'il y a suffisamment de fichiers disponibles, des *threads* seront lancés pour traiter le pipeline par lots de 10 éléments à la fois.

La méthode `race` accepte deux paramètres nommés : `race(batch => 32, degree => 4)` va traiter des lots de 32 éléments à la fois et lancer quatre travailleurs en parallèle. Les résultats sont produits dans l'ordre d'arrivée (de complétion des calculs). Il existe une autre méthode, `hyper`, qui respecte d'ordre des données en entrée. `hyper(batch => 64, degree => 2)` traitera des lots de 64 éléments, lancera deux travailleurs en parallèle et assurera que les résultats soient produits dans l'ordre relatif des données en entrée. Invoquées sans arguments, les méthodes `race()` et `hyper()` déterminent elles-mêmes la meilleure taille de lot et le nombre optimal de travailleurs en parallèle en fonction de ce qu'elles « savent » de l'architecture machine et du volume des données.

À noter que les méthodes `race` et `hyper` n'ont été implémentées dans Rakudo/Perl 6 que très récemment et nécessitent une version publique de Rakudo datant au plus tôt de septembre 2015.

8-3-2 - Asynchronisme

L'objectif de l'asynchronisme est de réagir à des événements qui auront lieu à l'avenir, sans que l'on puisse savoir ni contrôler quand ils auront lieu. On peut citer les exemples suivants :

- complétion de processus lancés en parallèle ;
- réponses à des requêtes Web ;
- connexions entrantes sur un serveur ;
- interaction avec un utilisateur dans une interface graphique (GUI) ;
- signaux.

Dans certains cas, il est possible de lancer une opération qui s'achèvera dans le futur et de se mettre en attente jusqu'à ce que ce soit le cas. Mais ce n'est pas toujours ce dont on a besoin, ou ça ne fonctionne pas forcément très bien avec des opérations en grand nombre.

Supposons que l'on désire copier de façon fiable toute une série de fichiers sur de multiples serveurs. On peut utiliser le programme `scp` (copie à distance sécurisée) dans une boucle :

```
for @téléchargements -> $fic-info {  
    run('scp', $fic-info.local, $fic-info.distant);  
}
```

8-3-2-1 - Le module `Proc::Async`

Mais si l'on désire accélérer le traitement en lançant quatre processus en parallèle, on peut réécrire le code en utilisant le module `Proc::Async` (module standard de Perl 6) :

```
for @téléchargements -> $fic-info {  
    my $proc = Proc::Async.new(run('scp',  
        $fic-info.local, $fic-info.distant);  
    await $proc.start;  
}
```

On retrouve ici la fonction `await` parce que `$proc.start` renvoie une promesse.

Pour aller plus vite, on peut essayer d'insérer toutes les promesses dans un tableau et de les attendre toutes :

```
my @travaux;  
for @téléchargements -> $fic-info {  
    my $proc = Proc::Async.new(run('scp',  
        $fic-info.local, $fic-info.distant);  
    push, @travaux, $proc.start;  
}  
await @travaux;
```

Mais cela risque bien sûr de surcharger le réseau et, en définitive, de ralentir le traitement (et peut-être de gêner d'autres utilisateurs). Il est peut-être préférable, selon la configuration matérielle, de ne traiter que quatre processus scp à la fois :

```
my @travaux;
for @téléchargements -> $fic-info {
    my $proc = Proc::Async.new(run('scp',
        $fic-info.local, $fic-info.distant);
    push, @travaux, $proc.start;
    if @working == 4 {
        await Promise.anyof(@travaux);
        @travaux .= grep({ !$_ });
    }
}
await @travaux;
```

Si le tableau `@travaux` atteint quatre éléments, le processus attend qu'une promesse soit tenue et fait un `grep` sur rompu (`unkept`).

8-3-2-2 - Flux de données asynchrone multiples : les fournisseurs

Les opérations asynchrones simples ne produisent qu'un seul résultat. Mais certaines sources de données asynchrones produisent de nombreuses valeurs au fil du temps. Voici quelques exemples de flux de données asynchrones :

- notifications de modification de fichier ;
- requêtes entrantes sur un serveur ;
- paquets de données arrivant sur un socket ;
- événements d'une interface graphique.

En Perl 6, un flux de données asynchrone s'appelle un fournisseur (`Supply`) - on aurait aussi pu l'appeler en français « producteur », mais le modèle utilisé ici ne correspond pas exactement au schéma classique producteur(s)-consommateur(s) (même s'il a de nettes ressemblances), autant éviter les ambiguïtés en choisissant un autre nom. Certains fournisseurs sont infinis (ils émettent sans discontinuer), d'autres sont finis, soit parce qu'ils ont terminé leur émission, soit en raison d'une erreur.

Un fournisseur de notifications de changements de fichiers va déclencher l'exécution automatique d'une suite de tests. On désire scruter un répertoire de tests (et optionnellement des répertoires sources) et l'on désire n'effectuer qu'un test à la fois. Le module standard `IO::Notification` permet de détecter des modifications de fichiers : la méthode `watch-path` renvoie un objet de type `Supply` sur lequel il est possible de « se brancher » au moyen de la méthode `tap` :

```
my $changements = IO::Notification.watch-path($répertoire-de-test);
$changements.tap({ say 'ça a changé !'; });
```

Dans le pipeline de données vu précédemment, la fonction `max` « aspirait » les données météorologiques (mode *pull*) à travers le pipeline. Un fournisseur est aussi un pipeline, mais qui fonctionne selon une logique inverse : les valeurs sont injectées dans le pipeline (mode *push*) au fur et à mesure de leur production par la source asynchrone. Les méthodes sont familières, mais on est ici en mode asynchrone. Cela signifie que l'on peut utiliser des fonctions comme `map` ou `grep` pour projeter et filtrer les données qui arrivent de façon asynchrone. Par exemple, il est possible de filtrer les noms de fichier selon leur extension :

```
my $changements = IO::Notification.watch-path($répertoire-source);
my $code        = $changements.grep(*.path ~~ /<.pm .p6> $/);
$code.tap({say 'Un fichier source a changé !';});
```

C'est bien pratique, mais beaucoup de gens ne résolvent pas tous leurs problèmes de listes en utilisant `grep`, `map` et les autres opérateurs d'ordre supérieur. Certains problèmes sont plus faciles à exprimer avec des boucles `for` ou

des conditions `if`, et ainsi de suite. Mais une boucle `for` est une construction synchrone et bloquante. Comment faire avec des données asynchrones ?

Perl 6 a une construction de boucle asynchrone nommée `whenever` (« lorsque ... », « à chaque fois que ... ») Le corps de la boucle s'exécute quand une valeur arrive :

```
whenever IO::Notification.watch-path($répertoire-de-test) {  
    lance-test-le-cas-échéant('Un test a changé');  
}
```

Comme c'est une boucle comme une autre, il est possible d'utiliser `LAST` pour décider comment gérer la fin de la séquence.

8-3-2-3 - Le bloc `react`

Une boucle `whenever` peut résider dans un bloc fournisseur (qui peut émettre des valeurs) ou dans un bloc `react` (qui fonctionne comme l'entrée dans une boucle événementielle) :

```
my $code = supply {  
    whenever IO::Notification.watch-path($répertoire-source) {  
        emit .path if .path ~~ /<.pm .p6> $/;  
    }  
}  
  
react {  
    whenever $code -> $path {  
        say "Le fichier de code $path a changé!";  
    }  
}
```

Pour revenir à l'exécution automatique de tests, on peut peut-être lancer des tests lors de modifications dans les répertoires de tests et de sources :

```
sub MAIN ($test-dir, *@répertoires-sources) {  
    react {  
        whenever IO::Notification.watch-path($test-dir) {  
            lance-test-le-cas-échéant('Un test a changé');  
        }  
        for @répertoires-sources -> $rep {  
            whenever IO::Notification.watch-path($dir) {  
                lance-test-le-cas-échéant('Un source a changé');  
            }  
        }  
        # ...  
    }  
}
```

Les notifications peuvent arriver sur des *threads* différents, mais un seul *thread* peut se trouver dans un état `supply/react` à un moment donné, il est donc possible en toute sécurité d'écrire :

```
sub lance-test-le-cas-échéant($raison) {  
    state $test-en-cours = False;  
    unless $test-en-cours {  
        say "Tests en cours ($raison)";  
        $test-en-cours = True;  
        whenever lance-tests() {  
            print "\n\n";  
            $test-en-cours = False;  
        }  
    }  
}
```

Pour lancer l'exécution, on retrouve `Proc::Async`. Ici, le programme affiche un texte indenté sur `STDOUT` et écarte silencieusement `STDERR`. Il renvoie une promesse, car `whenever` peut aussi travailler contre une promesse.

```
sub lance-tests() {
    my $runner = Proc::Async.new('prove ...');
    whenever $runner.stdout -> $output {
        print $output.indent(2);
    }
    whenever $runner.stderr { } # écarté
    return $runner.start;
}
```

Perl 6 rend l'asynchronisme explicite. Il fournit un soutien sous la forme d'un langage structuré, pour éviter un enchevêtrement de fonctions de rappel, et il guide les programmeurs vers des *threads* implicites sûrs et fiables.

8-3-3 - La concurrence

La concurrence désigne une compétition d'accès (en lecture et/ou en écriture) à des ressources partagées. Des passagers s'enregistrant à un vol ne doivent pas pouvoir choisir la même place assise. La classe simpliste suivante alloue des sièges aux passagers :

```
class Vol {
    has %!sièges;
    submethod BUILD(:@num-sièges) {
        %!sièges{@num-sièges} = False xx *;
    }
    method choisit-siège($siège, $id-passager) {
        die "Ce siège n'existe pas" unless %!sièges{$siège}:exists;
        die "Siège pris!" if %!sièges{$siège};
        %!sièges{$siège} = $id-passager;
    }
}
```

Il y a une situation de compétition d'accès dans les deux dernières lignes de la méthode `choisit-siège` : si deux *threads* sont dans cette méthode au même moment, ils peuvent tous les deux penser que le siège `$siège` est libre et lui attribuer leur passager respectif.

8-3-3-1 - Utilisation d'un surveillant (monitor)

Une solution consiste à utiliser un « surveillant » (`monitor`) : un surveillant est une classe qui garantit qu'un seul thread peut exécuter une méthode sur une instance donnée à un instant donné. Du coup, le second passager désirant choisir un siège va devoir attendre que le premier ait terminé de choisir le sien. Les modifications à apporter au code sont d'une simplicité biblique :

```
use OO::Monitors;

monitor Vol {
    has %!sièges;
    submethod BUILD(:@num-sièges) {
        %!sièges{@num-sièges} = False xx *;
    }
    method choisit-siège($siège, $id-passager) {
        die "Ce siège n'existe pas" unless %!sièges{$siège}:exists;
        die "Siège pris!" if %!sièges{$siège};
        %!sièges{$siège} = $id-passager;
    }
}
```

Le terme `monitor` utilisé pour déclarer la classe crée un nouveau type de paquetage fourni par le module `OO::Monitors`.

Fort bien, mais il subsiste une difficulté : si l'on utilise un surveillant dans une application Web asynchrone, en cas de contention, l'un des *threads* va se bloquer de façon synchrone en attendant que l'autre *thread* libère la ressource en sortant du surveillant.

8-3-3-2 - Utilisation d'un acteur (actor)

On peut mieux faire en utilisant un acteur (**actor**) au lieu d'un surveillant :

```
use OO::Actors;

actor Vol {
    has %!sièges;
    submethod BUILD(:@num-sièges) {
        %!sièges{@num-sièges} = False xx *;
    }
    method choisit-siège($siège, $id-passager) {
        die "Ce siège n'existe pas" unless %!sièges{$siège}:exists;
        die "Siège pris!" if %!sièges{$siège};
        %!sièges{$siège} = $id-passager;
    }
}
```

Un acteur place les invocations de méthodes arrivantes dans une queue (file d'attente) de traitement. Une invocation de méthode sur un acteur renvoie une promesse que l'invoquant attend :

```
await $vol.choisit-siège($siège, $id-passager);
```

Le *thread* en cours d'exécution est maintenant libre de traiter d'autres requêtes en attendant la libération de la ressource.

Les acteurs et les moniteurs placent le contrôle de la concurrence en dehors du code utilisateur, qui peut oublier la cuisine technique de bas niveau et se concentrer pleinement sur la logique fonctionnelle et/ou métier du problème.

9 - Conclusion

La présente Annexe 2 conclut une série de cinq articles consacrés à Perl 6. Ces cinq articles représentent ensemble plus de 300 pages de texte au format A4 ou environ 100 000 mots, ce qui représente l'équivalent d'un bon gros livre (sans compter les articles annexes tels que celui sur [les regex et les grammaires](#) ou celui sur les **Objets, classes et rôles en Perl 6 - Tutoriel de programmation orientée objet**). Cela donne un panorama assez complet de Perl 6 (plus complet même sur certains aspects que la documentation officielle actuelle de Perl 6), mais celui-ci ne saurait être exhaustif pour autant.

La taille imposante de la présente annexe reflète le très grand nombre de nouveautés introduites par Perl 6, qui en font un langage non seulement résolument moderne, mais aussi profondément évolutif et capable de s'adapter à de nouveaux besoins très récents ou même à venir. Si l'on ne devait en retenir que quelques-unes des nouveautés concrètes les plus révolutionnaires et les plus porteuses d'avenir, on pourrait mettre en avant les suivantes :

- **les regex et surtout les grammaires** (§ 7), en particulier la possibilité d'ajouter dynamiquement des nouveaux éléments syntaxiques à la grammaire Perl 6 existante, rendant le langage intrinsèquement malléable et évolutif ;
- **un nouveau système de programmation orientée objet** (§ 6) particulièrement flexible, puissant et expressif ;
- **les fonctions multiples** (§ 5) et la capacité de créer dynamiquement de nouveaux opérateurs ou de surcharger des opérateurs existants ;
- un modèle de **programmation fonctionnelle** (§ 4) très enrichi avec en particulier le support aux listes paresseuses ;

- un modèle de **programmation parallèle et concurrente** (§ 8) de haut niveau, fiable, facile à utiliser et extrêmement prometteur.

Avec une telle abondance de nouveautés révolutionnaires, nous pensons que Larry Wall a bien raison de dire que la perspective n'est pas que Perl 6 soit prochainement « le langage du mois ou même de l'année », mais de **faire de Perl 6 le langage des 30 ou 40 prochaines années**.

Nous espérons que ce vaste tutoriel incitera le lecteur à découvrir ce nouveau langage plein de promesses. À l'heure où nous écrivons cette conclusion (novembre 2015), la première version de production de Perl 6 doit sortir d'ici quelques semaines (avant la fin 2015). Les versions de développement disponibles actuellement sont presque entièrement fonctionnelles et faciles à installer, empresses-vous de les découvrir.

Le présent document présentant des informations encore très difficiles à trouver et parfois même disponibles pratiquement nulle part ailleurs (même en anglais), nous le mettons en licence *Creative Common* « Attribution CC BY » afin d'autoriser quiconque le désirerait à en reprendre librement le contenu, tout en souhaitant si possible que la source et son auteur soient cités. Si vous souhaitez contribuer au rayonnement de Perl 6, n'hésitez pas à l'utiliser, le diffuser et même le copier !

10 - Remerciements

Je remercie les auteurs anonymes de  **la documentation officielle Perl 6** dans laquelle j'ai abondamment puisé pour la rédaction de certaines parties du présent document.

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