Journées Perl 2016
Paris, 24 et 25 juin 2016

Perl 6 – Un langage résolument moderne

Laurent Rosenfeld

- **Perl 6 : nouvelle mouture du langage Perl (sortie en déc. 2015)**
- **Il reste dans l'esprit de Perl 5 :**
  - Fais ce que je veux dire (DWIM)
  - Il y a plusieurs façons de le faire (TIMTOWTDI)
  - Les choses faciles doivent être simples et les choses difficiles doivent être possibles

- **Il est en même temps radicalement nouveau**
  - Syntaxe proche de Perl 5, mais rompant avec la compatibilité ascendante
  - Langage multiparadigme
    - Programmation procédurale, fonctionnelle ou orientée objet
  - Typage « graduel » : vous choisissez le typage dont vous avez besoin :
    - typage statique fort (strict) ou typage dynamique
  - Système de programmation objet particulièrement moderne
  - Refonte des expressions régulières et extension à des grammaires en bonne est due forme
    - Un programme Perl 6 est compilé avec une grammaire en Perl 6
  - Système de parallélisme et de programmation concurrente de haut niveau

Rakudo : première version de production de Perl 6 : décembre 2015

- Distributions disponibles sous Linux, Max et Windows
- Contient :
  - La machine virtuelle MoarVM et le compilateur,
  - Des modules et un installateur de module, Panda
  - Un débogueur et un interpréteur à la ligne de commande, REPL
  - Une documentation abondante
  - Téléchargement : `http://rakudo.org/downloads/star/`

Installation de Rakudo-Star : `http://rakudo.org/how-to-get-rakudo/`

- Sous Windows ou Mac : fichier .msi ou .dmg
- Sous Linux : utiliser un package ou télécharger les sources et compiler les exécutables

```
$ wget http://rakudo.org/downloads/star/rakudo-star-2016.01.tar.gz
$ tar xzf rakudo-star-2016.01.tar.gz
$ cd rakudo-star-2016.01
$ perl Configure.pl --backend=moar --gen-moar
$ make
$ make rakudo-test
$ make rakudo-spectest
$ make install
```

Combien fait l'opération suivante : 0,3 - 0,2 – 0,1 ?

- En Perl 5 :

```
$ perl -E 'say 0.3 - 0.2 - 0.1;'
-2.77555756156289e-17
```

- En Ruby :

```
$ irb
irb(main):001:0>  puts 0.3 - 0.2 - 0.1
-2.7755575615628914e-17
```

- En Python 3 :

```
>>> print (0.3 - 0.2 - 0.1)
-2.7755575615628914e-17
```

  NB : en Python 2 : -2.77555756156e-17

- Java et TCL : comme Ruby et Python 3 : -2.7755575615628914e-17
- En C (gcc) :

```
$ ./sum_fl.exe
  -2.775558e-17
```

  NB : avec des `long double`, résultat plus proche : 1.122486e-317, mais toujours pas 0

Ces résultats ne sont pas dus à une faiblesse de ces langages

- Ce sont nos ordinateurs qui ne savent pas compter correctement...
- … et ça fait plus de 50 ans que ça dure !

En Perl 6 (test fait avec la boucle REPL) :

```
> my $l'addition-devrait-être-zéro = .3 - .2 - .1;
0
> say sprintf "%.50f", $l'addition-devrait-être-zéro;
0.00000000000000000000000000000000000000000000000000
```

On peut même comparer le résultat avec 0 :

```
> say "Vrai" if $l'addition-devrait-être-zéro == 0;
Vrai
```

Ne faites pas ce genre de comparaison avec les langages précédents

Perl 6 fait vraiment ce que nous voulons dire (DWIM) :

- J'attends que 0,3 – 0,2 – 01 soit égal à 0...
- … et c'est bien le cas avec Perl 6 !
- Nos ordinateurs comptent mal, mais pas avec Perl 6 ! C'est cela un langage moderne.
- En interne, valeurs `Rat` stockées comme des paires numérateur – dénominateur :

```
> say .3.nude
(3 10)
```

Les variables et identifiants

- Remarque sur la variable `$l'addition-devrait-être-zéro`
  - Doit être déclarée avec le mot-clef `my` (ou un autre déclarateur)
  - Contient une apostrophe et des tirets
  - Contient des caractères accentués (ou des lettres unicodes quelconques), par exemple :

```
> my $φ = (5**.5 + 1)/2;      # nombre d'or
1.61803398874989
> say "Le nombre d'or est égal à $φ.";
Le nombre d'or est égal à 1.61803398874989.
```

  - « $ » pour des variables scalaires (valeurs uniques) ;
  - « @ » pour les tableaux (listes de valeurs indexées par des entiers) ;
  - « % » pour des hachages (ensembles de paires clef-valeur)
  - Un tableau ou un hachage conserve son sigil quand on accède à un élément :

```
> my @tableau = 1, 2, 4, 6;
[1 2 4 6]
> say @tableau[2];
4
> my %hachage = jan => 1, fév => 2, mar => 3;
fév => 2, jan => 1, mar => 3
> say %hachage{"fév"};
2
```

Portée des variables

- Le mot-clef `my` introduit une variable lexicale, c'est-à-dire locale au bloc de code dans laquelle elle se trouve (essentiellement comme en Perl 5)

```
{
    my Str $var = "chaîne de caractères";
    say $var;           # affiche "chaîne de caractères"
}
say $var; # ERREUR: $var n'est plus accessible
```

Typage des données

On peut déclarer une variable avec un *type* imposant des contraintes :

```
my Int $c;
$c = 4;                 # tout va bien, $c vaut 4
say $c.WHAT             # (Int)
$c = 4.2;               # ERREUR : Type check failed in assignment to $c...
```

- *Notez l'opérateur « `.` » utilisé pour l'appel d'une méthode (et non plus « `->` »)*

Mais la déclaration n'est pas indispensable, Perl 6 « se débrouille » pour trouver le type si nécessaire :

```
my $d = 4.2;            # Pourrait aussi s'écrire : my $d = Rat.new(42, 10);
say $d.WHAT;            # (Rat)
say '$d = ', $d.numerator, " / ",
    $d.denominator;     # affiche : $d = 21 / 5
```

C'est ce que l'on appelle le « typage graduel », combinaison de typage statique et dynamique

Les types

Les variables `$c` et `$d` sont assimilables à des objets et peuvent invoquer des méthodes (ce ne sont pas vraiment des objets, c'est du *boxing*, mais, en interne, c'est par un système à objets que sont définis les types).

Les types les plus courants n'ont pas besoin d'un constructeur. Par exemple, il y a plusieurs manière de créer des nombres de type `Complex` :

```
my $z1 = 5 + 3i;
say $z1.WHAT;                    # (Complex)
my $z2 = Complex.new(4, 9);      # 4+9i
my Complex $z3 = 2 + 5i;
say $z1 + $z2 + $z3;             # affiche : 11+17i
```

Une simple affectation suffit, mais c'est un peu plus compliqué avec un type date :

```
my $d = Date.new(2015, 12, 24); # Veille de Noël : 2015-12-24
say $d.year;                    # 2015
say $d.month;                   # 12
say $d.day;                     # 24
say $d.day-of-week;             # 4  (donc, jeudi)
my $n = Date.new('2015-12-31'); # Saint-Sylvestre
say $n - $d;                    # 7 (delta 7 jours)
say $n + 1;                     # 2016-01-01
```

Mais la syntaxe avec le constructeur `new` était ici nécessaire, sinon on a une simple soustraction :

```
> my $d = 2015-12-25;  # Non, ce n'est pas Noël
1978
```

Le type chaîne de caractère (`Str`)

- Généralement délimitées par des apostrophes (ou *single quotes*) ou des guillemets, avec le même sémantique d'interpolation des variables qu'un Perl 5

```
my $user = 'Larry';
say "Bonjour, $user !";    # Bonjour Larry !
say 'Bonjour, $user !';    # Bonjour $user !
```

- L'opérateur de concaténation est le tilde « ~ » :

```
say "Hello" ~ "World !";   # Hello World !
```

- Il existe de nombreuses fonctions ou méthodes travaillant sur les chaînes :

```
my $nom = "Charlie";
say flip $nom;  # eilrahC (syntaxe fonctionnelle)
say $nom.flip;  # eilrahC (syntaxe de méthode)
say uc $nom;    # CHARLIE
say $nom.uc;    # idem
say $nom.chars; # 7 (nombre de caractères)
say substr $nom, 2, 3; # arl (sous-chaîne)
say "Je suis " ~ $nom; # Je suis Charlie (concaténation)
```

- La plupart des sous-routines internes acceptent une syntaxe de fonction ou de méthode
- On peut les combiner à volonté, notamment pour clarifier la précédence :

```
> say flip "Charlie".substr(2, 3).uc
LRA
```

Les types numériques

- Nous avons vu les types `Int`, `Rat` (rationnel) et `Complex`. Il y a aussi un type `Num` :

```
say 42.WHAT;           # (Int)
say 42.7.WHAT;         # (Rat)
say 2.sqrt.WHAT;       # (Num)
say 5.log.WHAT;        # (Num)
say 1e17.WHAT;         # (Num)
```

- Nombreuses fonctions ou méthodes pour travailler sur les nombres :

```
say 19.is-prime;       # True (19 est premier)
say 42.7.denominator; # 10
say 42.7.nude;         # (427 10), c-à-d 427/10
```

- On peut définir des « sous-types » ou sous-ensembles de types existants :

```
subset Impair of Int where { $_ % 2 };      # non divisibles par 2
                       # Impair est maintenant un sous-type
my Impair $x = 3;      # OK
my Impair $y = 2;      # ERREUR : Type check failed in assignment to $y...
```

Les tableaux

- Variables contenant des listes mutables de valeurs (comme en Perl 5)
- le sigil « @ » est conservé lors de l'accès à des éléments individuels :

```
my @nombres_shadoks = ['GA', 'BU', 'ZO', 'MEU'];
say @nombres_shadoks[1];     # imprime: BU
```

- Si les éléments sont sans espace on peut utiliser l'opérateur de citation de liste `<...>` :

```
> my @nombres_shadoks = <GA BU ZO MEU>;
[GA BU ZO MEU]
> say @nombres_shadoks.elems; # nombre d'éléments
4
> my $dernier = pop @nombres_shadoks;
MEU
> say @nombres_shadoks;
[GA BU ZO]
> say elems @nombres_shadoks; # nombre d'éléments
3
> say "Les shadoks ont perdu leur dernier chiffre: $dernier";
Les shadoks ont perdu leur dernier chiffre: MEU
> push @nombres_shadoks, $dernier;
[GA BU ZO MEU]
```

On remarque ci-dessus l'utilisation des fonctions `pop` et `push` connues en Perl 5

Les hachages

- Ensemble de paires clef-valeur accessibles via la clef.
- On peut en construire avec une syntaxe de liste :

```
> my %capitales = <Italie Rome Allemagne Berlin Espagne Madrid>;
Allemagne => Berlin, Espagne => Madrid, Italie => Rome
```

- … ou utiliser « `=>` », l'opérateur « virgule grasse »

```
my %capitales = (Italie => "Rome", Allemagne => "Berlin", Espagne =>
"Madrid");
```

- Pour ajouter un élément :

```
%capitales{"France"} = "Paris";
```

ou utiliser l'opérateur de citation :

```
%capitales<France> = "Paris";
```

ou utiliser la fonction `push` :

```
push %capitales, (Danemark => "Copenhague");
%capitales.push: (USA => "Washington");
say %capitales.elems;   # 6 (nombre de paires)
```

- Les fonctions ou méthodes `kv`, `keys` et `values` permettent de récupérer des listes de paires, de clefs ou de valeurs :

```
> say %capitales.kv
(France Paris Allemagne Berlin Italie Rome USA Washington Danemark […] )
> say %capitales.keys;
(France Allemagne Italie USA Danemark Espagne)
> say %capitales.values;
(Paris Berlin Rome Washington Copenhague Madrid)
```

Les opérateurs les plus courants (par ordre de précédence)

`.method`	Méthode post-fixée
`++ --`	Auto-incrémentation, auto-décrémentation (préfixées ou post-fixées)
`**`	Exponentielle
`! + - ~ ? |`	Symboles unaires : négation logique, plus, moins, concaténation, coercition booléenne, flatten
`* / % %% div gcd lcm`	Multiplication, division, modulo, divisibilité, division entière, PGDC, PPMC
`+ -`	Addition, soustraction
`x xx`	Réplication de chaîne (renvoie une chaîne), réplication d'élément (renvoie une liste)
`~`	Concaténation de chaînes
`~~ != == < <= > >= eq ne lt le gt ge`	Opérateur de comparaison intelligente, opérateurs de comparaisons numériques, opérateurs de comparaisons de chaînes
`&&`	*et* booléen (de haute précédence)
`||`	*ou* booléen (de haute précédence)
`?? !!`	Opérateur conditionnel ternaire
`= => += -= **= xx= .=`	Opérateurs d'affectation
`so not`	Booléens unaires de basse précédence : coercition booléenne et non logique
`and andthen`	*et* booléen (de basse précédence) – `andthen` : renvoie le premier élément non défini
`or xor orelse`	*ou* booléen (de basse précédence) – `orelse` : défini ou, comme //, mais de basse priorité

Nous verrons plus loin que l'on peut construire ses propres opérateurs

- **Les opérateurs – on en a tous rêvé, Perl 6 le fait :**
  - On peut chaîner les opérateurs de comparaison logique :

```
say "Valeurs dans l'ordre" if $u < $v < $x < $y < $z;
# équivaut à : ... if $u < $v and $v < $x and $x < $y and $y < $z
```

  - De même, les jonctions permettent d'écrire des comparaisons très concises :

```
my $x = 7;
say "Trouvé" if $x == 4|6|9|7|15;      # -> Trouvé
# Peut aussi s'écrire :
say "Trouvé" if $x == any <4 6 9 7 15>;
# équivaut à : ... if ($x == 4) or ($x == 6) or ($x == 9) or ...
```

- **Le métaopérateur de réduction**
  - Les opérateurs travaillent sur des données, les métaopérateurs travaillent sur des opérateurs
  - Le métaopérateur de réduction « [...] » transforme un opérateur infixé associatif quelconque en un opérateur de liste renvoyant un scalaire :

```
> say [+] 1, 2, 3, 4;
10
```

  - De même, la factorielle peut se calculer ainsi :

```
my $fact10 = [*] 1..10;   # -> 3628800
```

- **L'hyperopérateur « » (peut aussi s'écrire : << >>)**
  - Un hyperopérateur applique une opération à chaque membre d'une liste et renvoie une liste

```
say 5 «*» <3 4 5 6>;   # -> (15 20 25 30)
# Opération membre à membre avec deux ou plusieurs listes:
my @x = ('a'..'e') «~» (3..7)  «~» ('v'..'z'); # -> [a3v b4w c5x d6y e7z]
```

Quizz

- ## Exercice 1 : PPMC des nombres de 1 à 20
  - L'opérateur infixé `lcm` renvoie le PPMC de deux nombres. Calculer le plus petit entier positif divisible par tous les nombres de 1 à 20

- ## Exercice 2 : Calculer la somme des chiffres de factorielle 100

- ## Exercice 3 : carré de la somme moins somme des carrés
  - Trouver la différence entre le carré de la somme des 100 premiers entiers naturels et la somme des carrés des 100 premiers entiers.

Quizz : solutions

- ## Exercice 1 : PPMC des nombres de 1 à 20
  - L'opérateur infixé `lcm` renvoie le PPMC de deux nombres. Calculer le plus petit entier positif divisible par tous les nombres de 1 à 20
  - Le méta-opérateur `[…]` permet d'appliquer `lcm` aux nombres de l'intervalle 1..20 :

```
> say [lcm] 1..20;
232792560
```

- ## Exercice 2 : Calculer la somme des chiffres de factorielle 100

- ## Exercice 3 : carré de la somme moins somme des carrés
  - Trouver la différence entre le carré de la somme des 100 premiers entiers naturels et la somme des carrés des 100 premiers entiers.

Quizz : solutions

Exercice 1 : PPMC des nombres de 1 à 20

- L'opérateur infixé `lcm` renvoie le PPMC de deux nombres. Calculer le plus petit entier positif divisible par tous les nombres de 1 à 20
- Le méta-opérateur `[…]` permet d'appliquer `lcm` aux nombres de l'intervalle 1..20 :

```
> say [lcm] 1..20;
232792560
```

Exercice 2 : Calculer la somme des chiffres de factorielle 100

- Nous utilisons deux fois le métaopérateur de réduction `[…]` : une première fois avec la multiplication pour calculer 100 ! et une seconde fois pour la somme des chiffres du résultat.

```
> say [+] split '', [*] 2..100;
648
```

Exercice 3 : carré de la somme moins somme des carrés

- Trouver la différence entre le carré de la somme des 100 premiers entiers naturels et la somme des carrés des 100 premiers entiers.

Quizz : solutions

Exercice 1 : PPMC des nombres de 1 à 20

- L'opérateur infixé `lcm` renvoie le PPMC de deux nombres. Calculer le plus petit entier positif divisible par tous les nombres de 1 à 20
- Le méta-opérateur `[…]` permet d'appliquer `lcm` aux nombres de l'intervalle 1..20 :

```
> say [lcm] 1..20;
232792560
```

Exercice 2 : Calculer la somme des chiffres de factorielle 100

- Nous utilisons deux fois le métaopérateur de réduction `[…]` : une première fois avec la multiplication pour calculer 100 ! et une seconde fois pour la somme des chiffres du résultat.

```
> say [+] split '', [*] 2..100;
648
```

Exercice 3 : carré de la somme moins somme des carrés

- Trouver la différence entre le carré de la somme des 100 premiers entiers naturels et la somme des carrés des 100 premiers entiers.
- Métaopérateur `[]` pour la somme des 100 premiers nombres, hyperopérateur « » pour calculer les carrés des 100 premiers entiers et `[]` pour la somme de ces carrés :

```
say ([+] 1..100)**2 - [+] (1..100) «**» 2;
25164150
```

Les conditions

- Condition `if / elsif /else` classique. La condition ne nécessite pas de parenthèses :

```
if $âge < 12 {
    say "Enfant"
} elsif $âge < 18 {
    say "Adolescent"
} else {
    say "Adulte"
}
```

- La forme dite d'instruction modifiée ou condition postfixée est possible :

```
> say "Bienvenue sur ce site" if $âge >= 18;
Bienvenue sur ce site
```

- On retrouve aussi la forme négative `unless` (équivalente à `if not`):

```
say  "Désolé : réservé aux adultes !" unless $âge >= 18;
```

- La construction `given … when` correspond au `switch` d'autres langages :

```
my Int $âge = 18;
given $âge {
    when /\D/  { say "âge contient des non-chiffres" };
    when *..^0 { say "âge négatif ? Hum..." };
    when 0..2  { say "Bébé" };
    when 3..12 { say "Enfant" };
    when *..17 { say "Adolescent" };
    when   18  { say "tout jeune adulte" };
    default    { say "Adulte"}
}
```

Les boucles

- La boucle la plus commune est la boucle `for` :

```
> for 1..10 -> $val { print $val * 2, " "};
0 2 4 6 8 10 12 14 16 18 20 >
```

- Dans cette forme dite de « bloc pointu », la variable de boucle `$val` est un alias en lecture seule sur les valeurs de 1 à 10.
- Un bloc doublement pointu permet de modifier `$val` et donc le tableau d'origine :

```
my @tableau = [0..10];
for @tableau <-> $val {
    $val *= 2;
}
say @tableau;      # -> [0 2 4 6 8 10 12 14 16 18 20]
```

- Pour itérer sur les indices, on peut générer une liste d'indice à la volée :

```
my @mois = <none jan fév mar avr mai jun>;
for 1..end @mois -> $num { say "$num \t @mois[$num]" };
```

- On peut toujours utiliser la variable par défaut `$_` :

```
say $_ * 3 for 1..10;
```

- La boucle `while` est classique (parenthèses non requises autour de la condition) :

```
my $var = 0;
while $var < 5 {
    print 3 * ++$var, " ";      # -> 3 6 9 12 15
}
```

- Il existe aussi une boucle `until` classique et une boucle `loop` (boucle de type C)

Fonctions et sous-routines

- Introduites avec le mot clef `sub` :

```
saluer("Maître");   # imprime "Bonjour Maître"
sub saluer ($nom) {
    say 'Bonjour $nom.';
}
saluer(); # ERREUR: Calling saluer() will never work with declared signature
```

- La **signature** impose ici le passage d'un (seul) argument
- Par défaut, les paramètres d'une fonction sont en lecture seule, mais on peut modifier ce comportement avec les « traits » `is rw` (lecture et écriture, modification de l'argument d'appel) ou `is copy` (copie locale modifiable) :

```
my Int $valeur = 5;
ajoute-deux($valeur);
say $valeur; # -> 7
sub ajoute-deux (Int $x is rw) { $x += 2}
```

- La signature permet de vérifier non seulement le nombre d'arguments mais aussi leur type :

```
sub multiplie (Int $x, Int $y) {
    say "Produit = ", $x * $y
}
multiplie(3, 4); # -> Produit = 12
```

- … ou une condition supplémentaire déclenchant une erreur si elle n'est pas satisfaite :

```
sub divise (Numeric $x, Numeric $y where $y != 0) {   # Erreur si $y == 0
    return $x / $y;
}
```

Paramètres nommés

- Les *paramètres positionnels* utilisés jusqu'ici peuvent poser problème s'il y en a beaucoup
- On peut utiliser des *paramètres nommés* insensibles à l'ordre des arguments :

```
sub divise (Numeric :$dividende, Numeric :$diviseur where $diviseur != 0) {
    say $dividende/$diviseur
}
divise(dividende => 12, diviseur => 4);  # -> 3
divise diviseur => 4, dividende => 12 ;  # idem
```

Fonctions multiples

- Le mot-clef `multi` permet de définir des fonctions ayant le même nom mais faisant des choses différentes selon leur signature (nombre et/ou type des arguments)

```
multi salue ($nom)          {say "Bonjour $nom"}
multi salue ($nom, $titre) {say "Bonjour Ô $titre $nom"}
salue("George Lucas");                  # -> Bonjour George Lucas
salue "Yoda", "Maître";                 # -> Bonjour Ô Maître Yoda
```

Créer (ou redéfinir) ses propres opérateurs

- Un opérateur Perl 6 est en fait une fonction ayant un nom et une syntaxe d'appel particuliers
  - Précédence (priorité d'exécution) ;
  - Type de notation syntaxique (préfixée, infixée, postfixée, etc.)
  - Associativité (ordre d'exécution en cas de même précédence)
- On peut créer un nouvel opérateur en précisant au minimum le type de notation
- Utilisons par ex. le petit « ² » en haut à gauche du clavier pour définir l'élévation au carré :

```
sub postfix:<²>(Numeric $n) {$n**2}
say 5²;         # -> 25
say (3 + 4)²;   # -> 49
```

- Perl 6 supportant les caractères Unicode, on peut utiliser les lettres grecques, symboles mathématiques, pictogrammes normalisés, etc. Ou un nom de plusieurs lettres comme pour ces opérateurs de conversion entre francs et euros :

```
sub prefix:<f2€> (Numeric $n) {$n/6.55957}
sub prefix:<€2f> (Numeric $n) {$n*6.55957}
say f2€ 6.56;   # -> 1.0000656
say €2f 10;     # -> 65.5957
```

- Les opérateurs internes étant eux-même définis de cette façon, on peut les redéfinir à condition de changer quelque chose à leur définition. Par exemple, l'opérateur « ! » de négation est préfixé, on peut réutiliser le « ! » pour créer un opérateur factorielle postfixé :

```
sub postfix:<!> (Int $n) {
    [*] 2..$n
}
say 20!;      # -> 2432902008176640000
say ! False;  # -> True
```

- Voici un exemple définissant la précédence des opérateurs entre des paires utilisées pour simuler des opérations entre les nombres complexes :

```
multi sub infix:<+> (Pair $x, Pair $y) is equiv(&infix:<+>) {
    return $x.key + $y.key => $x.value + $y.value
}
multi sub infix:<*>  (Pair $x, Pair $y) is equiv(&infix:<*>) {
    return ($x.key * $y.key - $x.value * $y.value)
        => ($x.key * $y.value + $y.key * $x.value);
}
my $a = 4=>3;   # NB: $a et $b n'ont aucune sémantique particulière en Perl 6
my $b = 7=>2;
say $a + $b;           # -> 11 => 5
say $a * $b;           # -> 22 => 29
say $a + $a * $b;      # -> 26 => 32
say $a + ($a * $b);    # -> 26 => 32  (idem: précédence respectée)
say ($a + $a) * $b;    # -> 44 => 58
```

- On retrouve bien la précédence mathématique usuelle des opérateurs « + » et « * »

Conclusion

- Nous n'avons fait qu'un bref tour d'horizon de la syntaxe de Perl 6, mais avons essayé de présenter quelques caractéristiques (métaopérateurs, fonctions multiples, création de nouveaux opérateurs, etc.) le rendant particulièrement puissant et expressif

- Nous avons dû passer sous silence d'autres caractéristiques essentielles :
  - Un système d'objets moderne et riche
  - Des expressions régulières refondues débouchant sur des grammaires
  - Un modèle de programmation fonctionnelle très enrichi, avec en natif le support aux listes paresseuses, les fermetures, les lambdas, la curryfication, etc.
  - Un modèle de programmation parallèle, concurrente et asynchrone de haut niveau
  - Un support natif à des structures de données multidimensionnelles de plus bas niveau
  - un interfaçage particulièrement simple avec des bibliothèques C/C++ ou des modules Perl 5

- Tout n'est pas encore parfait pour autant
  - Les performances se sont considérablement améliorées mais nécessitent encore des progrès
  - Quelques fonctionnalités non encore implémentées (par exemple les macros)

Perl 6 offre déjà une palette de fonctionnalités largement inégalée et des capacités d'extension du langage exceptionnelles qui peuvent en faire le langage des vingt ou trente prochaines années.

Références

- Documentation Perl 6 (en anglais) : https://doc.perl6.org/

- Tutoriel de Naoum Hankache : http://naoumhankache.developpez.com/tutoriels/perl6/perl6intro/

- De Perl 5 à Perl 6 (série de Moritz Lenz et Laurent Rosenfeld):
  - http://laurent-rosenfeld.developpez.com/tutoriels/perl/perl6/les-bases/
  - http://laurent-rosenfeld.developpez.com/tutoriels/perl/perl6/les-nouveautes/
  - http://laurent-rosenfeld.developpez.com/tutoriels/perl/perl6/approfondissements/
  - http://laurent-rosenfeld.developpez.com/tutoriels/perl/perl6/annexe-01/
  - http://laurent-rosenfeld.developpez.com/tutoriels/perl/perl6/annexe-02/

- Les regex et les grammaires de Perl 6 : une puissance expressive sans précédent : http://laurent-rosenfeld.developpez.com/tutoriels/perl/perl6/regex-grammaire-puissance/

- Deux articles dans *GNU Linux Magazine France* : numéro 193 (mai 2016) et numéro 194 (juin 2016)

Très bref aperçu des grammaires de Perl 6

- Les **briques de base** : les regex nommées ou « règles » au sens large
  - Ce sont des regex ayant une syntaxe de fonction ou de méthode ;
  - Trois sortes : *regex*, *token* et *rule*. Exemples de déclaration :

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

  - Exemple d'utilisation pour reconnaître un nombre décimal :

```
my regex chiffres { \d+ }
my regex décimal { <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
```

  - Les règles de type *token* ou *rule* ne font pas de retour arrière :

```
my token lettres { abc .+ g };
say "abcdefg" ~~ /<lettres>/ ?? "Réussit" !! "Échoue"; # ->  Échoue
```

  - Mais ça marcherait avec un quantificateur non gourmand (ou *frugal*) :

```
my token lettres { abc .+? g };
say "abcdefg" ~~ /<lettres>/ ?? "Réussit" !! "Échoue"; # ->  Réussit
```

  - (Mais ça ne marcherait de nouveau pas avec une *rule*, à cause des espaces dans le motif.)
  - En gros :
    - Une *regex* fait ce qu'on attend d'une expression régulière (analyse de bas niveau) ;
    - Une *token* va servir à l'analyse *lexicale* (division du texte en symboles ou lexèmes)
    - Une *rule* servira plutôt à l'analyse *syntaxique* (rapport entre les lexèmes, contexte)

Créer une grammaire

- Une grammaire regroupe des règles comme une classe regroupe des méthodes:

```
grammar Identité {
    rule nom     { Nom '=' (\N+) } # caractères non retours à la ligne
    rule adresse { Adr '=' (\N+) } # idem
    rule âge     { Age '=' (\d+) } # des chiffres
    rule desc {
        <nom>     \n
        <âge>     \n
        <adresse> \n
    }
    # etc.
}
```

- Une grammaire peut hériter d'une autre grammaire (de même que les classes)

```
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>=.+ }
}
```

Une grammaire simple pour valider des noms de modules Perl

- Un nom de module peut se décomposer en identifiants séparés par des paires de caractères deux-points : « :: », par exemple List::Util ou List::MoreUtils
  - Un identifiant commence par un caractère alphabétique ou un « _ » suivis d'alphanumériques
  - Certains modules n'ont qu'un identifiant (par ex. Memoize), donc pas de « :: »
  - D'autres ont des noms « à rallonge » : Regexp::Common::Email::Address.
- Définissons les règles *identifiant* et *séparateur* (et *TOP*) :

```
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 simplement quelques noms de modules :

```
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";
}
```

- Seul `Super::Nouveau::Module` sera reconnu

- Parfois, on résume les noms de modules en remplaçant les « :: » par des tirets.
  - Un seul changement à faire pour propager la modification à l'ensemble :

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

- Reconnaître ne suffit pas toujours, il faut parfois exécuter du code dans une grammaire
  - Exécuter du code au sein même d'une regex (ici par exemple pour déboguer) :

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

  - Définir une méthode au sein de la grammaire (oui, une grammaire est une classe) :

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

  - Fournir un *objet d'actions* associé à une règle : avertir sur les noms de modules trop longs

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

  - Syntaxe d'appel avec l'objet d'action :

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

- 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 un avertissement :

```
> perl6 grammaire_nom_module.pl
Nom de module très long! Peut-être le réduire ?
  in method TOP at perl6_grammaire_module.pl:15
「Mon::Module::Super::Cali::Fragi::Listi::Cexpi::Delilicieux」
 identifiant => 「Mon」
 séparateur => 「::」
 identifiant => 「Module」
 séparateur => 「::」
 identifiant => 「Super」
(...)
 identifiant => 「Delilicieux」
```