

Perl 6

chris@aperghis.fr

<http://www.dil.univ-mrs.fr/~chris/Telechargement.php>



Perl 6 existe !
Je l'ai utilisé...



Sites



Au début, il n'y avait rien. Enfin, ni plus ni moins de rien qu'ailleurs.



www.parrot.org



www.rakudo.org



La version de Perl6



Commande :

```
perl6 -v
```

```
This is Rakudo Perl 6, version 2011.07 built on parrot 3.6.0 0
```

```
Copyright 2008-2011, The Perl Foundation
```



Les blocs.

Contrairement à d'autres langages, il n'y a en Perl 6 aucune contrainte liée à l'ouverture d'un bloc. Certaines instructions nécessitent l'ouverture d'un bloc qui leur est attaché.



```
if (condition) {  
  Bloc Vrai  
}  
else {  
  Bloc Faux  
}
```

Bloc
conditionnel

```
NEXT {  
  Corps du bloc  
}
```

Bloc étiqueté

```
while (condition) {  
  Corps de la boucle  
}
```

Bloc boucle

```
{  
  Corps du bloc  
}
```

Bloc banalisé

**Blocs a
intérêt
réduit.**

```
sub sp( ) {  
  Corps du sous programme  
}
```

Bloc sous programme

```
for @l -> $i {  
  Corps de la boucle  
}
```

Bloc for

**Blocs
intéressants**



Visibilité des variables.



En Perl 6 toute variable doit être déclarée. Une variable déclarée privée (`my $var`) ne sera donc vue que à l'intérieur du bloc dans lequel elle aura été déclarée.

```
my $x = 5;
my $y = 5;
{
  my $y = 3;
  say "Variable x dans le bloc : $x";
  say "Variable y dans le bloc : $y";
}
say "Variable y hors du bloc : $y";
```

```
Variable x dans le bloc : 5
Variable y dans le bloc : 3
Variable y hors du bloc : 5
```

La directive `"my"` permet de déclarer une variable lexicale locale.

Une alternative est de la déclarer par le directive `"our"` qui permet d'en faire une variable globale partagée.

L'intérêt d'une variable globale est de pouvoir l'utiliser partout sans besoin de la transmettre et d'en garder la trace.

Ce n'est toutefois pas toujours la meilleure chose à faire.

A noter qu'on appelle aussi une telle variable `"Variable de package"` car elle est visible de n'importe quel module.



Non déclaration de variables.



Si une variable est utilisée sans déclaration préalable, cela génère une erreur.

```
my $x = 5;  
say "Variable y: $y";
```

```
===SORRY!===  
Symbol '$y' not predeclared in <anonymous> (/Users/chris/p.p6:2)
```



Non déclaration de variables.



Le programme suivant ne peut pas s'exécuter car il génère des erreurs.

```
my $x = 5;
{
  my $y = 7;
  {
    my $z = 9;
    say $x; # Affiche 5.
    say $y; # Affiche 7.
    say $z; # Affiche 9.
  }
  say $x; # Affiche 5.
  say $y; # Affiche 7.
  say $z; # Génère une erreur de non définition.
}
say $x; # Affiche 5.
say $y; # Génère une erreur de non définition.
```



Les blocs.



En Perl 6, tout bloc, quel que soit sa place, est une fermeture, c'est à dire un morceau de code lié à l'environnement lexical dans lequel il est défini.

C'est ainsi qu'un bloc qui a été mémorisé et qui est ultérieurement rappelé hors de son contexte d'origine conserve les valeurs initiales de son environnement, même si ce dernier n'est plus accessible.

```
my $x = -> $a, $b { say "Argument 1 : $a - Argument 2: $b"; }  
$x(10, 20);  
$x(3.14159265, 2.71828183);  
$x("Un", "Deux");
```

```
Argument 1 : 10 - Argument 2: 20  
Argument 1 : 3.14159265 - Argument 2: 2.71828183  
Argument 1 : Un - Argument 2: Deux
```



La notion de fermeture.



Exemple :

```
my $nom = "Christian.";
$fermeture = { print $nom; }

.....
my $nom = "Jacques.";
$fermeture (); # Imprime "Christian."
```

Le fait que tout bloc soit une fermeture implique qu'il est possible de lui transmettre des arguments.

```
my $block;
{
  my $nom = "Christian";
  $block = { say $nom };
}

my $nom = "Jacques";
$block ();
```

Christian

Bloc exécuté
immédiatement.

```
{
  print "Bonjour.\n";
}
```

Bloc mémorisé pour
exécution ultérieure.

```
$fermeture = {
  print "Adieu.\n";
}
```



Fermeture simple.

Création d'un compteur au moyen d'un bloc utilisé comme une fermeture.



```
my $compte;  
{  
  my $i;  
  $compte = { say "valeur du compteur : ", ++$i };  
}  
  
$compte ();  
$compte ();  
$compte ();  
$compte ();
```

```
valeur du compteur : 1  
valeur du compteur : 2  
valeur du compteur : 3  
valeur du compteur : 4
```



Fermetures multiples.



On va stocker dix fermetures dans une liste, chaque élément `$i` de la liste contient le multiplicande `$i` en tant que variable lexicale.

```
my @fois = ();
for 1..10 -> $i{
  my $t = $i;
  @fois[$i] = sub ($a) { return $a * $t; };
}
say "Le produit de 3 et de 4 est egal a : ", @fois[3](4);
say "Le produit de 5 et de 30 est egal a : ", @fois[5](30);
say "Le produit de 9 et de 8 est egal a : ", @fois[9](8);
```

```
Le produit de 3 et de 4 est egal a : 12
Le produit de 5 et de 30 est egal a : 150
Le produit de 9 et de 8 est egal a : 72
```



Le bloc "for".



Le bloc rattaché à l'instruction "for" se comporte comme une structure admettant un (des) paramètre(s).

Dans le cas le plus simple, l'itération sur une liste, le paramètre est l'invariant de boucle spécifié ou un invariant implicite.

```
my @n = (1, 2, 3, 4, 5, 6, 7, 8, 9);  
  
for @n -> $i {  
    say "Valeur : $i";  
}
```

```
my @n = (1, 2, 3, 4, 5, 6, 7, 8, 9);  
  
for @n {  
    print "Valeur : ";  
    .say;  
}
```

```
Valeur : 1  
Valeur : 2  
Valeur : 3  
Valeur : 4  
Valeur : 5  
Valeur : 6  
Valeur : 7  
Valeur : 8  
Valeur : 9
```

```
Valeur : 1  
Valeur : 2  
Valeur : 3  
Valeur : 4  
Valeur : 5  
Valeur : 6  
Valeur : 7  
Valeur : 8  
Valeur : 9
```





Plusieurs invariants dans le bloc "for".

Il est aussi possible de transmettre plus d'un invariant au bloc de la boucle.

```
my @n = (1, 2, 3, 4, 5, 6, 7, 8, 9);  
  
for @n -> $a, $b, $c {  
  say "Valeurs : $a $b $c";  
}
```

```
Valeurs : 1 2 3  
Valeurs : 4 5 6  
Valeurs : 7 8 9
```

Mais dans ce cas, le compte doit tomber juste sous peine d'erreur.

```
my @n = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9);  
  
for @n -> $a, $b, $c {  
  say "Valeurs : $a $b $c";  
}
```

```
Valeurs : 0 1 2  
Valeurs : 3 4 5  
Valeurs : 6 7 8
```

```
Not enough positional parameters passed; got 1 but expected 3  
in <anon> at line 5:/Users/chris/p.p6  
in main program body at line 1:src/metamodel/RoleToInstanceApplier.nqp
```





Variables positionnelles dans le bloc "for".

Un bloc lié à une boucle for peut faire référence à des variables positionnelles.

```
my @l = (1, 2, 3, 4, 5, 6, 7, 8, 9);  
  
for @l {  
    say "$^a $^b $^c";  
}
```

```
1 2 3  
4 5 6  
7 8 9
```

```
my @l = (1, 2, 3, 4, 5, 6, 7, 8, 9);  
  
for @l {  
    say "$^a $^b $^c";  
}
```

```
3 2 1  
6 5 4  
9 8 7
```

Ici aussi, le compte doit tomber juste sous peine d'erreur.



Déclarations `my`, `state`, `our`, `temp` et `let`.



Ce sont les diverses manières de déclarer les variables.

`my` déclare une variable dans l'environnement lexical courant.

```
my $variable_lexicale;
```

`state`, comme `my`, déclare une variable dans l'environnement lexical courant, mais elle conserve sa valeur d'appel en appel au lieu d'être réinitialisée à chaque entrée dans le bloc,

```
state $variable_lexicale;
```

`our` déclare un alias lexical vers une variable de la table de symbole d'un package.

```
our $variable_de_package;
```

`temp` et `let` ne sont pas des déclarations mais des commandes d'exécution qui mémorisent la valeur d'une variable qui sera récupérée ultérieurement.

Une variable `temp` voit sa valeur restaurée au moment de quitter son bloc.

Une variable `let` conserve la valeur temporaire qu'elle a pu prendre.



Blocs étiquetés.



Un bloc contiendra un certain nombre de descripteurs qui lui sont propres.

Ces descripteurs sont eux mêmes des blocs (des fermetures) liées comme propriétés du bloc qui les contient et sont définis par un nom à l'intérieur du bloc qu'elles affecteront.

```
    NEXT {  
        Instructions  
    }
```

Le nom est une suite de lettres majuscules.

Ces blocs ne sont pas exécutés en séquence. Ils sont mémorisés au moment de la compilation et sont rappelés au moment approprié pendant l'exécution.



Les contrôles.



NEXT : Exécuté entre deux itérations d'une boucle.

LAST : Exécuté à la fin de la dernière itération.

PRE : Exécuté avant tout autre bloc.

POST : Exécuté après tout autre bloc.

CATCH: Récupère les exeptions (il ne peut y avoir qu'un CATCH).

KEEP : Exécuté si le bloc se termine sans exeption ou lorsque toutes les exeptions ont été piégées (s'exécute après le CATCH).

UNDO : Exécuté si le bloc se termine sur une exeption non piégée (s'exécute après le CATCH).



Les sous programmes.



Un sous programme est déclaré au moyen de la directive "sub".
Il admet une liste de paramètres.
Son code est stocké dans un bloc.

```
sub alerte {  
  say "Attention arret du systeme dans 5 minutes !";  
}
```

```
alerte;
```

```
Attention arret du systeme dans 5 minutes !
```

```
sub somme ($a, $b) {  
  say "La somme de $a et de $b est egale a : ", $a + $b;  
}
```

```
somme(15, 70);
```

```
La somme de 15 et de 70 est egale a : 85
```



Les paramètres.



La liste des paramètres est généralement appelée la signature du sous programme.

```
sub divide ($dividende, $diviseur) {  
  my $resultat;  
  given $diviseur {  
    when 0 {$resultat = "Erreur";}   
    default {$resultat = $dividende / $diviseur;}  
  }  
  return $resultat;  
}  
  
say "Le quotient 10 par 5 est : ", divide(10,5);  
say "Le quotient 25 par 0 est : ", divide(25,0);
```

```
Le quotient 10 par 5 est : 2  
Le quotient 25 par 0 est : Erreur
```



Transmission par la liste standard



Sans liste de paramètres explicite, les arguments sont transmis par l'intermédiaire de la liste standard @_.

```
sub prod {  
  my $p;  
  for @_ -> $element {  
    $p *= $element  
  }  
  return $p;  
}  
say ("calcul de prod (1..10) : ", prod (1..10));  
say ("calcul de prod prod (1,5,54,77,9) : ", prod (1,5,54,77,9));
```

```
calcul de prod (1..10) : 3628800  
calcul de prod prod (1,5,54,77,9) : 187110
```



A propos des paramètres.



Il faut noter qu'il n'y a pas assignation des paramètres mais un simple référencement.

Les variables présentes dans la signature sont des références à accès limité en lecture. Il est interdit de les modifier.

Toute tentative visant à le faire se solde par une erreur (exception).

```
sub mod ($a, $b) {  
    $a = 10;  
    $b = 20  
}
```

```
my $x = 100;  
my $y = 200;  
mod($x, $y);
```

Cannot modify readonly value

in '&infix:<=>' at line 1:src/metamodel/RoleToInstanceApplier.nqp

in 'mod' at line 4:/Users/chris/p.p6

in main program body at line 9:/Users/chris/p.p6



Modification des paramètres.



La propriété "rw" permet de marquer le paramètre concerné comme modifiable dans le corps du sous programme auquel il est transmit.

```
sub mod ($a is rw, $b is rw) {  
    $a = 111;  
    $b = 222  
}  
  
my $x = 100;  
my $y = 200;  
say "Avant appel."  
say "Valeur de x : $x, valeur de y : $y";  
mod($x, $y);  
say "Apres appel."  
say "Valeur de x : $x, valeur de y : $y";
```

```
Avant appel.  
Valeur de x : 100, valeur de y : 200  
Apres appel.  
Valeur de x : 111, valeur de y : 222
```



Propriété "copy".



Par défaut, les paramètres sont des alias vers ceux passés dans la liste d'appel, ce qui implique un passage par référence.
Pour des raisons de sécurité ils sont marqués "constants" et ne sont par là même non modifiables.

La propriété "copy" permet de spécifier que le passage de paramètre doit se faire par valeur.

```
sub puiss ($x1 is copy, $x2 is copy) {  
    return ($x1 ** $x2)  
}  
  
my $a = 2;  
my $b = 10;  
say "$a a la puissance $b vaut ", puiss($a,$b);
```

```
2 a la puissance 10 vaut 1024
```



Transmission de listes.



Le sigil de la variable indique le type à utiliser.
Si on spécifie une liste (@) on contraint l'appel à se faire par une liste.
Tout autre type générera une erreur.

```
sub prod (@l) {  
  my $p;  
  for @l -> $element {  
    $p *= $element  
  }  
  return $p;  
}  
say ("Calcul de prod (1..10) : ", prod (1..10));  
say ("Appel avec un scalaire : ", prod (5));
```

```
Calcul de prod (1..10) : 3628800  
Nominal type check failed for parameter '@l'; expected Positional but got  
Int instead  
  in 'prod' at line 4:/Users/chris/p.p6  
  in main program body at line 11:/Users/chris/p.p6
```



Banalisation du type.

Si on désire que le type transmis par la liste soit indifférent, le sigil doit être un \$ en raison du référencement automatique.



```
my $x = 10;
my @l = (1, 2, 3, 4, 5);
my %couples = (
  'Romeo' => 'Juliette',
  'Cyrano' => 'Roxane',
  'Protis' => 'Gyptis',
);

sub code ($p) {
  say ("Type du parametre transmis : ", $p.WHAT);
}

code ($x);
code (@l);
code (1..10);
code (%couples);
```

```
Type du parametre transmis : Int()
Type du parametre transmis : Array()
Type du parametre transmis : Range()
Type du parametre transmis : Hash()
```



Transmission de plusieurs listes.



C'est ce mécanisme qui va permettre de transmettre plusieurs listes sans qu'il y ait, comme en Perl 5, aplatissage des listes.

```
my @l1 = (1, 2, 3, 4, 5);  
my @l2 = <A B C D E>;  
my @l3 = <Un Deux Trois Quatre Cinq>;  
  
sub code (@x, @y, @z) {  
    say ("Premiere liste : ", @x.join(" "));  
    say ("Seconde liste : ", @y.join(" "));  
    say ("Troisieme liste : ", @z.join(" "));  
}  
code(@l1, @l2, @l3);
```

```
Premiere liste : 1 2 3 4 5  
Seconde liste : A B C D E  
Troisieme liste : Un Deux Trois Quatre Cinq
```



Transmission de code.



Il est aussi possible de transmettre un nom de sous programme.

```
my @l1 = (1, 2, 3, 4, 5);  
  
sub appel (&code) {  
    code (@l1);  
}  
  
sub somme (@x) {  
    say ("Somme : ", [+] @x);  
}  
  
appel (&somme)
```

Somme : 15



Paramètres optionnels.



La liste d'appel peut contenir des paramètres optionnels.

Il y a plusieurs manières de déclarer un paramètre comme étant optionnel.

```
$opt = valeur  
$opt?  
$opt? = valeur
```

```
sub param ($x, $y = "a vous")  
{  
    say ("$x $y.");  
}
```

```
param("Bonjour", "Monsieur");  
param("Bonjour", "Madame");  
param("Bonjour");
```

```
Bonjour Monsieur.  
Bonjour Madame.  
Bonjour a vous.
```

```
sub param ($x, $y?) {  
    say ("$x ", $y??$y!!"a vous", " .");  
}
```

```
param("Bonjour", "Monsieur");  
param("Bonjour", "Madame");  
param("Bonjour");
```

```
Bonjour Monsieur.  
Bonjour Madame.  
Bonjour a vous.
```



Inversion de paramètres.



Lorsque la liste de paramètres est longue, il peut se présenter des oublis ou des inversions, ce qui risque de provoquer des erreurs.

```
sub param ($Un, $Deux, $Trois) {  
    say (" Parametre Un : $Un.");  
    say (" Parametre Deux : $Deux.");  
    say (" Parametre Trois : $Trois.");  
}  
say "Premier appel."  
param(1, 2, 3);  
say "Second appel."  
param(1, 3, 2);
```

```
Premier appel.  
Parametre Un : 1.  
Parametre Deux : 2.  
Parametre Trois : 3.  
Second appel.  
Parametre Un : 1.  
Parametre Deux : 3.  
Parametre Trois : 2.
```

Manifestement, le second appel présente une inversion des paramètres.



Les paramètres nommés.



Pour éviter ce problème, il suffit de nommer les paramètres.

```
sub param ($Un, $Deux, $Trois) {  
    say (" Parametre Un : $Un.");  
    say (" Parametre Deux : $Deux.");  
    say (" Parametre Trois : $Trois.");  
}  
say "Premier appel."  
param(Un => 1, Deux => 2, Trois => 3);  
say "Second appel."  
param(Un => 1, Trois => 3, Deux => 2);
```

Premier appel.

Parametre Un : 1.

Parametre Deux : 2.

Parametre Trois : 3.

Second appel.

Parametre Un : 1.

Parametre Deux : 2.

Parametre Trois : 3.

L'inversion des deux paramètres n'a plus d'effet néfaste sur le résultat.

Ici, l'appel peut se faire indifféremment sous forme positionnelle ou de paramètres nommés.



Contraindre les paramètres nommés.



Si on désire contraindre l'appel de se faire par paramètres nommés, il faut faire précéder le nom du paramètre par un ":".

```
sub param (:$Un, :$Deux, :$Trois) {  
  say (" Parametre Un : $Un.");  
  say (" Parametre Deux : $Deux.");  
  say (" Parametre Trois : $Trois.");  
}  
say "Premier appel."  
param(Un => 1, Deux => 2, Trois => 3);  
say "Second appel."  
param(1, 2, 3);
```

Premier appel.

Parametre Un : 1.

Parametre Deux : 2.

Parametre Trois : 3.

Second appel.

Too many positional parameters passed; got 3 but expected 0
in 'param' at line 3:/Users/chris/p.p6
in main program body at line 11:/Users/chris/p.p6



Changement du nom.



La possibilité de transmettre les arguments par nom sous entend qu'ils doivent être considérés comme des interfaces publics du sous programme.

C'est pourquoi, il pourrait être intéressant de publier un paramètre avec un nom en le liant à une variable de nom différent.

```
sub param (:Un($Param1), :Deux($Param2), :Trois($Param3)) {  
  say ("Parametre Un : $Param1.");  
  say ("Parametre Deux : $Param2.");  
  say ( Parametre Trois : $Param3.);  
}
```

```
param(Un => 1, Deux => 2, Trois => 3);
```

```
Parametre Un : 1.  
Parametre Deux : 2.  
Parametre Trois : 3.
```

Ici, les noms publics sont Un, Deux et Trois alors que les noms internes sont Param1, Param2 et Param3.



Ordre des paramètres.



Lorsque qu' une même liste d'appels comporte à la fois des paramètres positionnels et des paramètres nommés, tous les paramètres positionnels doivent impérativement apparaître avant les paramètres nommés.

```
sub param ($P1, :$P2) {  
  say ("Parametre Un : $P1.");  
  say ("Parametre Deux : $P2.");  
}
```

```
param(1, P2 => 2);
```

```
Parametre Un : 1.  
Parametre Deux : 2.
```

```
sub param (:$P1, $P2) {  
  say ("Parametre Un : $P1.");  
  say ("Parametre Deux : $P2.");  
}
```

```
param(1, P2 => 2);
```

===SORRY!===

```
Cannot put required parameter after variadic parameters at line 3,  
near ") { \n sa"
```



Les paramètres absorbants.

C'est la possibilité qui est offerte de stocker la liste des paramètres dans une structure (liste ou hash)
Le nom du paramètre est alors précédé d'une astérisque.

C'est ce qui permet de créer des listes d'arguments de longueur variable, d'où leur nom.



```
sub capitales ($entete, *%europe) {  
  say $entete;  
  for %europe.kv -> $pays, $capitale {  
    say " $pays capitale : $capitale";  
  }  
}
```

```
capitales("Liste des capitales",  
  France => "Paris",  
  Italie => "Rome",  
  Espagne => "Madrid",  
  Autriche => "Vienne",  
  Portugal => "Lisbonne");
```

```
Liste des capitales  
France capitale : Paris  
Italie capitale : Rome  
Espagne capitale : Madrid  
Autriche capitale : Vienne  
Portugal capitale : Lisbonne
```



Les paramètres typés.



La liste de paramètres permet non seulement de spécifier la structure qui est attendue (\$, @, %, &), mais aussi le type de valeur souhaité.

```
sub aff (Str $nom, Numeric $val) {  
    say "Le nombre $nom a la valeur $val";  
}
```

```
aff("Pi", 3.14159265);  
aff(2.71828183, "e");
```

```
Le nombre Pi a la valeur 3.14159265  
Nominal type check failed for parameter '$nom'; expected Str but got Rat  
instead  
in 'aff' at line 1:/Users/chris/p.p6  
in main program body at line 5:/Users/chris/
```



Contraintes supplémentaires.



Il peut être parfois insuffisant de se contenter de décrire les attributs d'un paramètre.

Dans ces conditions, une contrainte additionnelle peut être spécifiée par l'intermédiaire d'un bloc "where".

```
sub note ($val where {0 <= $val <= 20}) {  
  say "Vous avez obtenu la note $val."  
}
```

```
my $x = 14;  
my $y = 25;  
note($x);  
note($y);
```

Vous avez obtenu la note 14.

```
Constraint type check failed for parameter '$val'  
in 'note' at line 3:/Users/chris/p.p6  
in main program body at line 10:/Users/chris/p.p6
```



Les multisub.



Lorsqu'on est en présence de plusieurs programmes portant des noms identiques dans une même application, on parle de définition multiple ou "multisub".

Lorsqu'il est fait référence au multiSub, le système parcourt la liste des sous programmes et cherche celui dont la signature est la plus voisine de celui qui est invoqué. Ces diverses fonctions doivent donc différer par leur liste de paramètres ou par leur signature.

Un multisub est déclaré par l'indicateur "multi".

```
multi fac (0) {return (1)}  
multi fac ($n) {return ($n * fac($n - 1))}  
  
my $x = 10;  
say ("La factorielle de $x est egale a ", fac($x));
```

```
La factorielle de 10 est egale a 3628800
```



La fonction d'ackerman.



La fonction d'Ackerman est définie comme suit :

$Ackerman(0, Y) = Y + 1$

$Ackerman(X, 0) = Ackerman(X - 1, 1)$

$Ackerman(X, Y) = Ackerman(X - 1, Ackerman(X, Y - 1))$

```
multi sub ack (0,$n) {return ($n+1)};

multi sub ack ($m, 0) {return ack($m-1, 1)};

multi sub ack ($m,$n) {return ack($m-1, ack($m, $n-1))};

my $r = &ack(3,5);

say ("resultat : $r.");
```

```
resultat : 253.
```



Et pour finir, un grand classique.



"Ciseaux", "Papier", "Rocher"

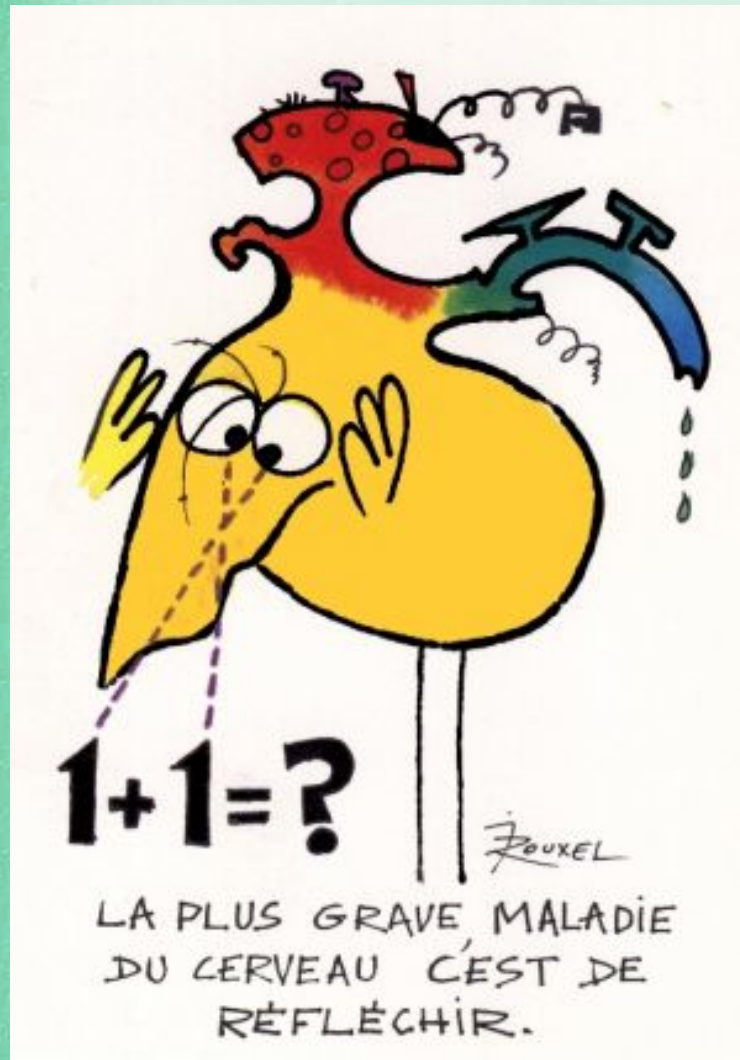
```
multi pcr("Ciseaux", "Papier") {return ("gagne")};
multi pcr("Rocher", "Ciseaux") {return ("gagne")};
multi pcr("Papier", "Rocher") {return ("gagne")};
multi pcr($a, $b where { $a eq $b }) {return ("egalite")};
multi pcr($a, $b) {return ("perdu")};

my @o =< Ciseaux Papier Rocher >;
for @o X @o -> $i, $j {
    say ("$i contre $j, resultat : ", pcr($i,$j));
}
```

```
Ciseaux contre Ciseaux, resultat : egalite
Ciseaux contre Papier, resultat : gagne
Ciseaux contre Rocher, resultat : perdu
Papier contre Ciseaux, resultat : perdu
Papier contre Papier, resultat : egalite
Papier contre Rocher, resultat : gagne
Rocher contre Ciseaux, resultat : gagne
Rocher contre Papier, resultat : perdu
Rocher contre Rocher, resultat : egalite
```



Pour conclure.



Ne réfléchissez plus...

Utilisez Perl 6

