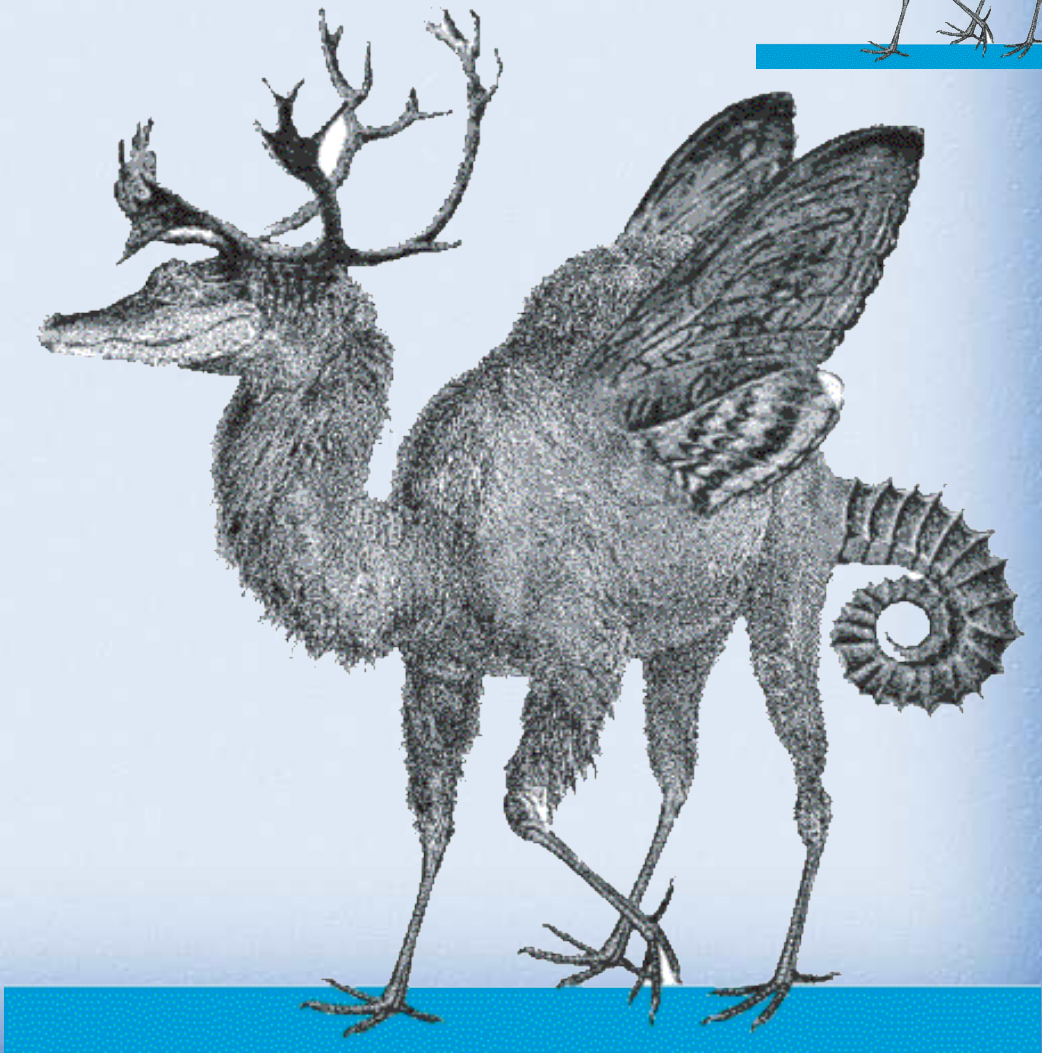


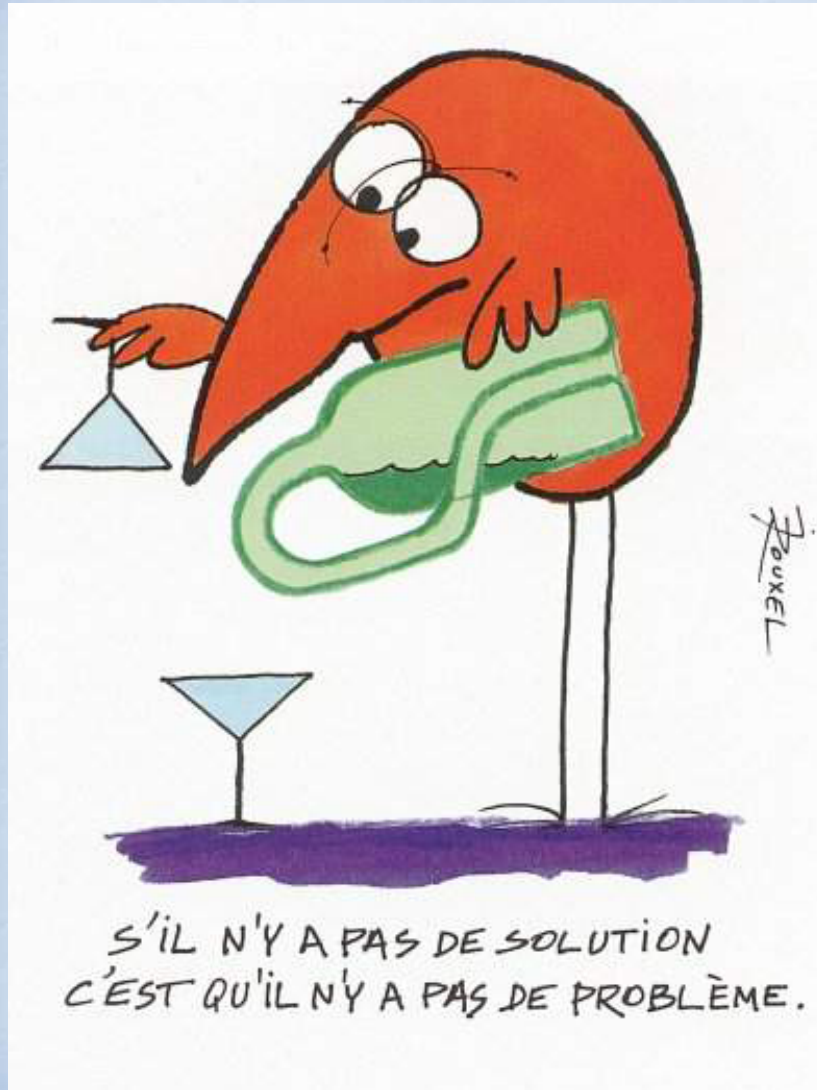
Perl 6.

chris@aperghis.fr

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



Solutions Pell 6



**Perl 6,
une solution
aux problèmes.**



Version de perl.

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 scalaire.

Le type scalaire peut être
Une valeur numérique.
Une chaîne de caractères.
Un booléen.



```
my $chaine = "Ceci est une chaine de caracteres.";
my $entier = 2005;
my $flottant = 3.14159;
my $ref_liste = ["Un", "Deux", "Trois", "Quatre"];
say "Editions.";
say "Chaine : $chaine.";
say "Entier : $entier.";
say "Flottant : $flottant.";
say "Reference de liste : $ref_liste.";
```

```
Chaine : Ceci est une chaine de caracteres..
Entier : 2005.
Flottant : 3.14159.
Reference de liste : Un Deux Trois Quatre.
coruscant:~/Langages/Perl6 chris$
```



A propos du test du type.

La méthode "what" permet de tester le type du scalaire auquel elle est appliquée.

```
my $s1 = 5 < 6;  
my $s2 = "Perl";  
my $s3 = 6 - 5;  
my $s4 = 3.14;  
$s1.WHAT.say;  
$s2.WHAT.say;  
$s3.WHAT.say;  
$s4.WHAT.say;
```

```
Bool()  
Str()  
Int()  
Rat()
```



Les opérateurs de coercition.



Les opérateurs de coercition ('+', '~', '?').
Ils vont contraindre un changement de contexte dans
l'évaluation d'une valeur scalaire.

```
my $a;  
$a = ?50 ;  
("Le type de la variable a est : " ~  
$a.WHAT) .say;  
$a = +'50' ;  
("Le type de la variable a est : " ~  
$a.WHAT) .say;  
$a = ~5E2 ;  
("Le type de la variable a est : " ~  
$a.WHAT) .say;
```

```
Le type de la variable a est : Bool()  
Le type de la variable a est : Num()  
Le type de la variable a est : Str()
```



Opérateur ternaire.

C'est l'opérateur ?? !!

Il évalue soit le premier opérande (qui suit le ??)
soit le second (qui suit le !!)

En fonction du résultat ("vrai" ou "faux") d'une expression booléenne.

```
$pronom = ($sexe == "fille") ?? "Elle" !! "Il";
```

```
my $sexe = "fille";  
my $pronom = ($sexe eq "fille")??"Elle"!!"Il";  
say $pronom, " est present", $pronom eq "Elle"??"e"!!"", ".";  
$sexe = "garcon";  
$pronom = ($sexe eq "fille")??"Elle"!!"Il";  
say $pronom, " est present", $pronom eq "Elle"??"e"!!"", ".";
```

Elle est presente.

Il est present.



Les listes.

Une liste est une collection d'objets scalaires.
La numérotation des éléments commence à 0.



```
my @liste = ("Zero", "Un", "Deux", "Trois", "Quatre", "Cinq");  
say "Premier element : ", @liste[0];  
say "Second element : ", @liste[1];  
say "Troisieme element : ", @liste[2];  
say "Quatrieme element : ", @liste[3];  
say "Cinquieme element : ", @liste[4];  
say "Sixieme element : ", @liste[5];  
my $nb_e = @liste.elems;  
say "Nombre d'element : $nb_e";
```

```
Premier element : Zero  
Second element : Un  
Troisieme element : Deux  
Quatrieme element : Trois  
Cinquieme element : Quatre  
Sixieme element : Cinq  
Nombre d'element : 6
```

Plusieurs représentations possibles :

```
@1 = (1, 2, 3, 4);  
@1 = 1, 2, 3, 4;  
@1 = <1 2 3 4>;
```



Itération sur les éléments d'une liste



En perl 6-, la boucle "for" est un itérateur.

```
my @liste = ("Zero", "Un", "Deux", "Trois", "Quatre", "Cinq");  
for @liste -> $n {  
    say $n;  
}
```

Zero
Un
Deux
Trois
Quatre
Cinq

Pour récupérer les n premiers éléments d'une liste, on utilise la notation n .

```
my @l = (1, 2, 3, 4, 5, 6, 7, 8, 9);  
for @l[^5] -> $n {  
    say $n;  
}
```

1
2
3
4
5



Transformations

La méthode `split` permet de transformer une chaîne de caractères en une liste.

```
my $ch = "Zero Un Deux Trois Quatre Cinq";  
my @l = $ch.split(" ");  
for @l -> $n{  
    say "$n";  
}
```

```
Zero  
Un  
Deux  
Trois  
Quatre  
Cinq
```



La méthode `join` effectue l'opération inverse.

```
my @liste = ("Zero", "Un", "Deux", "Trois", "Quatre", "Cinq");  
my $ch = @liste.join(" ");  
say $ch;
```

```
Zero Un Deux Trois Quatre Cinq
```

Ce qui nous permet d'afficher une liste de manière présentable.

```
my @liste = ("Zero", "Un", "Deux", "Trois", "Quatre", "Cinq");  
say @liste.join(", ");
```

```
Zero, Un, Deux, Trois, Quatre, Cinq
```



Opérations aléatoires dans une liste.

La méthode "pick" nous permet de récupérer de manière aléatoire un des éléments de la lista à laquelle elle est appliquée.

```
my @liste = ("Zero", "Un", "Deux", "Trois", "Quatre", "Cinq");  
my $val = @liste.pick;  
say $val;
```

Trois

Cinq

Zero

Avec l'argument "*", la méthode "pick" mélange aléatoirement les éléments de la liste dans une nouvelle structure.

```
my @liste = ("Zero", "Un", "Deux", "Trois", "Quatre", "Cinq");  
my @aleat = @liste.pick(*);  
say @aleat.join(" ");
```

Trois Zero Un Deux Cinq Quatre



Les opérateurs de liste.



L'opérateur **xx** permet de multiplier les listes.

```
@liste = "Pom" xx 3;
```

```
@liste est maintenant : ("Pom", "Pom", "Pom");
```

On dispose aussi de l'opérateur d'assignement **xx=** :

```
@liste = ("Un", "Deux");
```

```
@liste xx= 3;
```

```
@liste est maintenant ("Un", "Deux", "Un", "Deux", "Un", "Deux").
```

L'opération est identique à :

```
@liste = (@liste, @liste, @liste);
```

L'opérateur d'énumération **..** retourne une liste de valeur :

```
@chiffres = 0 .. 9;
```



Exemples d'opérateurs de liste.

```
my @l1 = "Pom" xx 3;  
say "Liste 1 : ", @l1.join(" ");  
my @l2 = ("Un", "Deux");  
say "Liste 2 : ", @l2.join(" ");  
@l2 xx= 3;  
say "Nouvelle liste 2 : ", @l2.join(" ");  
my @l3 = ("Un", "Deux");  
@l3 = (@l3, @l3, @l3);  
say "Liste 3 : ", @l3.join(" ");  
my @chiffres = 0 .. 9;  
say "Chiffres : ", @chiffres.join(" ");
```

```
Liste 1 : Pom Pom Pom  
Liste 2 : Un Deux  
Nouvelle liste 2 : Un Deux Un Deux Un Deux  
Liste 3 : Un Deux Un Deux Un Deux  
Chiffres : 0 1 2 3 4 5 6 7 8 9
```



Réduction de liste.



Tout opérateur infixé positionné entre crochets [].
Deviens un opérateur sur l'ensemble des éléments d'une liste.

[+] 1, 2, 3 va calculer $1 + 2 + 3$

[*] 1,2,3 va calculer $1 * 2 * 3$

[**] 4, 3, 2 Va élever à la puissance sous la forme $4 ** (3 ** 2)$

```
my @a;  
@a = [+] 1, 2, 3;  
("[+] 1, 2, 3 donne "~ @a).say;  
@a = [*] 1..10;  
("[*] 1..10 donne "~ @a).say;  
@a = [**] 4, 3, 2;  
("[**] 4, 3, 2 donne "~ @a).say;
```

```
[+] 1, 2, 3 donne 6  
[*] 1..10 donne 3628800  
[**] 4, 3, 2 donne 262144
```



Test sur les éléments d'une liste.



On peut se servir de l'opérateur de réduction pour effectuer des tests sur une liste donnée.

```
my @liste1 = 1, 1, 2, 3, 5, 8, 10;  
my @liste2 = 9, 3, 4, 1, 16, 30, 11;
```

```
("La liste 1 est ", ([<=] @liste1)??""!!"non ", "triee.").say;  
("La liste 2 est ", ([<=] @liste2)??""!!"non ", "triee.").say;
```

```
La liste 1 est trie.  
La liste 2 est non trie.
```

```
my @liste1 = 10, 10, 10, 10, 10, 10;  
my @liste2 = 9, 3, 4, 1, 16, 30, 11;
```

```
("Les elements ", ([==] @liste1)??"sont"!!"ne sont pas", " tous identiques.").say;  
("Les elements ", ([==] @liste2)??"sont"!!"ne sont pas", " tous identiques.").say;
```

```
Les elements sont tous identiques.  
Les elements ne sont pas tous identiques.
```



Les paires.

Une paire est un simple doublet clé, valeur.
Il y a deux syntaxes possibles pour les représenter.

```
my $paire1 = 'Un' => '1';  
my $paire2 = :Deux('2');  
say "Premiere paire : $paire1";  
say "Seconde paire : $paire2";
```

```
Premiere paire : Un      1  
Seconde paire  : Deux   2
```



Les hashes.

Un hash est un ensemble de valeurs scalaires auxquelles on accède au moyen d'une clé.

```
my %couples = (  
  'Romeo' => 'Juliette',  
  'Cyrano' => 'Roxane',  
  'Protis' => 'Gyptis',  
);  
for ('Romeo', 'Cyrano', 'Protis') -> $lui {  
  say "$lui est avec ", %couples{$lui}, "."  
}
```

```
Romeo est avec Juliette.  
Cyrano est avec Roxane.  
Protis est avec Gyptis.
```



Itération sur un hash.

la méthode "kv" permet de récupérer la liste des doublets et, par là même, d'explorer le hash.

```
my %couples = (  
  'Romeo' => 'Juliette',  
  'Cyrano' => 'Roxane',  
  'Protis' => 'Gyptis',  
);  
for %couples.kv -> $lui, $elle {  
  say "$lui est avec $elle."  
}
```

```
Romeo est avec Juliette.  
Cyrano est avec Roxane.  
Protis est avec Gyptis.
```



Itération sur les clés d'un hash.

la méthode "keys" permet de récupérer la liste des clés et d'accéder aux éléments du hash.

```
my %couples = (  
  'Romeo' => 'Juliette',  
  'Cyrano' => 'Roxane',  
  'Protis' => 'Gyptis',  
);  
  
for %couples.keys -> $lui {  
  say "$lui est avec %couples{$lui}."  
}
```

```
Romeo est avec Juliette.  
Cyrano est avec Roxane.  
Protis est avec Gyptis.
```



Une opération originale.

Inversion des clés et des valeurs.

```
my %epouse = (  
  'Romeo'   => 'Juliette',  
  'Cyrano'  => 'Roxane',  
  'Protis'  => 'Gyptis',  
);  
say "Liste des conjointes."  
for %epouse.kv -> $lui, $elle {  
  say "  L'epouse de $lui est $elle."  
}  
  
my %epoux = %epouse.invert;  
say "Liste des conjoints."  
for %epoux.kv -> $elle, $lui {  
  say "  L'epoux de $elle est $lui."  
}
```

Liste des conjointes.

L'epouse de Romeo est Juliette.

L'epouse de Cyrano est Roxane.

L'epouse de Protis est Gyptis.

Liste des conjoints.

L'epoux de Juliette est Romeo.

L'epoux de Roxane est Cyrano.

L'epoux de Gyptis est Protis.



Affichage. une méthode .perl permet de présenter les structures en mode "Perl", c'est à dire de manière particulièrement lisible.



```
my %epouse = (  
    'Romeo' => 'Juliette',  
    'Cyrano' => 'Roxane',  
    'Protis' => 'Gyptis',  
);  
my @epoux = %epouse.keys;  
say "Affichage du hash : ";  
say "    ", %epouse;  
say "    ", %epouse.perl;  
say "Affichage e la liste : ";  
say "    ", @epoux;  
say "    ", @epoux.perl
```

```
Affichage du hash :  
Romeo JulietteCyrano RoxaneProtis Gyptis  
{"Romeo" => "Juliette", "Cyrano" => "Roxane", "Protis" => "Gyptis"}  
Affichage e la liste :  
RomeoCyranoProtis  
["Romeo", "Cyrano", "Protis"]
```



Les séquences.

Il est possible de définir des séquences, c'est à dire des suites arithmétiques ou géométriques.



```
my @pairs := 0, 2 ... *;  
my @impairs := 1, 3 ... *;  
my @puissances2 := 1, 2, 4 ... *;  
  
say "Les dix premiers nombres pairs."  
say @pairs[^10].join(" ");  
say "Les dix premiers nombres impairs."  
say @impairs[^10].join(" ");  
say "Les dix premieres puissances de 2."  
say @puissances2[^10].join(" ");
```

```
Les dix premiers nombres pairs.  
0 2 4 6 8 10 12 14 16 18  
Les dix premiers nombres impairs.  
1 3 5 7 9 11 13 15 17 19  
Les dix premieres puissances de 2.  
1 2 4 8 16 32 64 128 256 512
```



Définition de séquences.

Pour définir une séquence il faut un nombre suffisant de termes.

```
my @suite1 := 1, 3 ... *;  
say "Les dix premieres valeurs de : 1, 3 ... *";  
say @suite1[^10].join(" ");  
my @suite2 := 1, 3, 9 ... *;  
say "Les dix premieres valeurs de : 1, 3, 9 ... *";  
say @suite2[^10].join(" ");
```

```
Les dix premieres valeurs de : 1, 3 ... *  
1 3 5 7 9 11 13 15 17 19  
Les dix premieres valeurs de : 1, 3, 9 ... *  
1 3 9 27 81 243 729 2187 6561 19683
```



Séquences plus élaborées.



On peut aussi définir des séquences par d'une fonction génératrice.

```
my @Fib := 0, 1, -> $a, $b, { $a + $b } ... *;  
say "Les dix premiers termes de la suite de Fibonacci."  
say @Fib[^10].join(" ");
```

```
Les dix premiers termes de la suite de Fibonacci.  
0 1 1 2 3 5 8 13 21 34
```

La notation

-> \$a, \$b { \$a + \$b }

Permet de définir un bloc pointé, en quelque sorte une "Fonction Lambda" qui récupère deux arguments et en retourne la valeur définie, ici leur somme.

La séquence permet de définir le nombre d'arguments à transmettre à partir de la fin de la séquence déjà générée afin de calculer le nouveau terme.

```
my @Fib := 0, 1, * + * ... *;  
say "Le terme d'indice 10 de la suite de Fibonacci : @Fib[10]";
```

```
Le terme d'indice 10 de la suite de Fibonacci : 55
```



Limitation de la séquence.



Ici, la notation

-> \$a { \$a > 10000 }

Permet une nouvelle fois de définir un bloc pointé qui retourne la valeur "Vrai" lorsque la valeur est supérieure à 10000.

```
my @Fib := 0, 1, -> $a, $b, { $a + $b} ...^ -> $a { $a > 10000};  
say "Les termes inferieurs a 10000 de la suite de fibonnacci."  
say @Fib.join(" ");
```

Les termes inferieurs a 10000 de la suite de fibonnacci.

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765

Autre possibilité. TIMTOWTDI

```
my @Fib := 0, 1, * + * ...^ * > 10000;  
say "Les termes inferieurs a 10000 de la suite de fibonnacci."  
say @Fib.join(" ");
```

Les termes inferieurs a 10000 de la suite de fibonnacci.

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765



Lire une valeur.

Le descripteur `$*IN` permet d'accéder à l'entrée standard.
Si on désire poser une question, on dispose de la fonction `prompt`
sous la forme : `my $rep = prompt ("Question ? ");`

```
say "Quel est votre prenom ? ";  
my $prenom = $*IN.get;  
my $ville = prompt ("Ou habitez vous ? ");  
say "Bonjour $prenom de $ville.";
```

```
Bonjour Christian de Marseille.
```



Les intervalles

Il est possible de définir un intervalle de référence et de tester si une valeur donnée par rapport à ses bornes.

```
loop {
  print "Donnez moi une valeur entre 1 et 100 : ";
  my $n = $*IN.get;
  if (1 <= $n >= 100) {
    say "J'ai dit entre 1 et 100 !"
  } else {
    say "C'est bon.";
    last
  }
}
```

```
Donnez moi une valeur entre 1 et 100 : 500
J'ai dit entre 1 et 100 !
Donnez moi une valeur entre 1 et 100 : 50
C'est bon.
```



Les opérateurs bit à bit.



Sur des scalaires.

Opération		Exemple	Résultat
Intersection	+&	42 +& 18;	2
Union	+	42 + 18;	58
Ou exclusif	+^	42 +^ 18;	56
Complément	+^	+^ 42;	- 43

Sur des chaînes.

Opération		Exemple	Résultat
Intersection	~&	"jj" ~& "gg"	"bb"
Union	~	"aa" ~ "bb";	"cc"
Ou exclusif	~^	"GG" ~^ "***";	"mm"

Les opérations de décalage.

Opération	
A droite	>>
A gauche	<<

Exemple	Résultat
4 << 3;	32
42 >> 1;	2



Exemples.

```
say "42 +& 18 : ", 42 +& 18;
say "42 +| 18 : ", 42 +| 18;
say "42 +^ 18 : ", 42 +^ 18;
say "complement de 42 : ", +^ 42;
say "Intersection de chaines 'jj' ~& 'gg' : ", 'jj' ~& 'gg';
say "Union de chaines 'aa' ~| 'bb' : ", 'aa' ~| 'bb';
say "Disjonction de chaines 'GG' ~^ '**' : ", 'GG' ~^ '**';
```

```
42 +& 18 : 2
42 +| 18 : 58
42 +^ 18 : 56
complement de 42 : -43
Intersection de chaines 'jj' ~& 'gg' : bb
Union de chaines 'aa' ~| 'bb' : cc
Disjonction de chaines 'GG' ~^ '**' : mm
```

```
'jj'      : 0110101001101010 (6A6A)
'gg'      : 0110011101100111 (6767)
intersection : 0110001001100010 (6262) = 'bb'
```

```
'aa'      : 0110000101100001 (6161)
'bb'      : 0110001001100010 (6262)
union      : 0110001101100011 (6363) = 'cc'
```



Les hyper opérateurs.



Ce sont des opérateurs spécifiques des listes représentant une extensions des opérateurs standards.

Tout opérateur a sa version standard et sa version hyper-opérateur.

Un opérateur standard est étendu en hyper-opérateur en étant placé entre >> et <<.

Un hyper-opérateur effectuera l'opération sur l'ensemble des éléments de la liste.

`@somme = @liste_1 >>+<< @liste_2;`



Exemples.

```
my @s;  
my @l1 = (0..10);  
say "l1 : ", @l1.join(" ");  
my @l2 = (100..110);  
say "l2 : ", @l2.join(" ");  
@s = @l1 >>+<< @l2;  
say "s : ", @s.join(" ");  
@s = @l1 >>*<< @l2;  
say "s : ", @s.join(" ");  
@s = @l1 >>+>> 1;  
say "s : ", @s.join(" ");
```

```
l1 : 0 1 2 3 4 5 6 7 8 9 10  
l2 : 100 101 102 103 104 105 106 107 108 109 110  
s : 100 102 104 106 108 110 112 114 116 118 120  
s : 0 101 204 309 416 525 636 749 864 981 1100  
s : 1 2 3 4 5 6 7 8 9 10 11
```



Papier, ciseaux, Rocher.

Réalisation du jeu en utilisant les propriétés offertes par les hyperopérateurs.



```
my @o =< Ciseaux Papier Rocher >;
my @res = <Perd Nul Gagne>;
my %scores = (
  @o[1] => [ @o ],
  @o[0] => [ @o[2,0,1] ],
  @o[2] => [ @o[1,2,0] ]
);
sub pcr($a, $b) {
  return [~] (@res >>xx<< ( %scores{$a} >>eq>> $b));
}
for @o X @o -> $i, $j {
  say ("$i contre $j, resultat : ", pcr($i,$j));
}
```

```
Ciseaux contre Ciseaux, resultat : Nul
Ciseaux contre Papier, resultat : Gagne
Ciseaux contre Rocher, resultat : Perd
Papier contre Ciseaux, resultat : Perd
Papier contre Papier, resultat : Nul
Papier contre Rocher, resultat : Gagne
Rocher contre Ciseaux, resultat : Gagne
Rocher contre Papier, resultat : Perd
Rocher contre Rocher, resultat : Nul
```



Les jonctions.

Au niveau le plus élémentaire, une jonction est une opération logique entre des valeurs.

Opération	
Intersection	&
Union	
Ou exclusif	^

Alors que $||$ est une opération entre deux expressions :

`if ($valeur == 1) || ($valeur == 2) { ... }`

$|$ est une opération entre deux valeurs

`if $valeur == 1 | 2 { ... }`

Ici, le résultat est identique.

Le résultat sera "vrai" si \$valeur est égal à 1 ou à 2 et "Faux" sinon.



Retour de jonction.



Une jonction ne renvoie pas une simple valeur mais une valeur composite contenant tous ses opérandes.

Cette valeur de retour est une jonction qui peut être utilisée partout où une telle opération est utilisable.

```
$jonction = 1 | 2;  
if ($valeur == $jonction) { ... };
```

La variable `$jonction` est utilisée en lieu et place de `1 | 2` et a exactement le même effet.



Exemple.

```
my $a = 1;
my $b = 0;
my $valeur;
"Test de la jonction ($a & $b). ".print;
$valeur = ($a & $b);
if ($valeur) {
    say "Resultat : Vrai"
} else {
    say "Resultat : Faux"
}
"Test de la jonction ($a | $b). ".print;
$valeur = ($a | $b);
if ($valeur) {
    say "Resultat : Vrai"
} else {
    say "Resultat : Faux"
}
```



```
Test de la jonction (1 & 0). Resultat : Faux
Test de la jonction (1 | 0). Resultat : Vrai
```



Utilisation des jonctions.

Basiquement, une jonction est un ensemble non ordonné dont les éléments sont réunis par une opération logique.

Toute opération sur la jonction est une opération sur la totalité de l'ensemble.



Fonction	Opérateur	Relation	Signification
all	&	AND	Doit être vrai pour toutes les valeurs.
any		OR	Doit être vrai pour au moins une valeur.
one	^	XOR	Doit être vrai pour exactement une valeur.
none		NOT	Doit être faux pour toutes les valeurs.

Lorsqu'une jonction est déclarée, les éléments de la liste sont remplacés par leur valeur.



Exemple.

```
my $a = 0;
my $b = 1;
my $c = 1;
my $d = 1;
my $valeur = all($a, $b, $c, $d);
if ($valeur) {
    "Resultat all ($a, $b, $c, $d) : Vrai".say
} else {
    "Resultat all ($a, $b, $c, $d) : Faux".say
}
$valeur = any($a, $b, $c, $d);
if ($valeur) {
    "Resultat any ($a, $b, $c, $d) : Vrai".say
} else {
    "Resultat any ($a, $b, $c, $d) : Faux".say
}
```

```
Resultat all (0, 1, 1, 1) : Faux
Resultat any (0, 1, 1, 1) : Vrai
```



Les jonctions dans un contexte booléen.



L'exemple le plus simple est le résultat de l'évaluation des jonctions dans un contexte booléen.

Jonction	Fonction	Signification
true (\$a & \$b)	true (all (\$a,\$b))	Résultat "vrai" si \$a et \$b sont vrais.
true (\$a \$b)	true (any (\$a,\$b))	Résultat "vrai" si \$a ou \$b est "vrai" ou les deux.
true (\$a ^ \$b)	true (one (\$a,\$b))	Résultat "vrai" si \$a ou \$b est "vrai".
	true (none (\$a,\$b))	Résultat "vrai" si \$a et \$b sont faux.

Les opérateurs arithmétiques interagissent sur les jonctions de la même manière que les hyper opérateurs.

`$jonction = any(1, 2);`

`$jonction *= 3;`

`#$jonction est maintenant any (3,6);`



Affichage de la jonction..



Afficher une jonction permet d'en connaître toutes les caractéristiques.

```
my $a = 1; my $b = 2; my $c = 3; my $d = 4;  
my $valeur = all($a, $b, $c, $d);  
say "Jonction : ", $valeur;  
$valeur += 1;  
say "Jonction : ", $valeur;
```

```
Jonction : all(1, 2, 3, 4)  
Jonction : all(2, 3, 4, 5)
```

A noter qu'une variable scalaire non initialisée est considérée comme une jonction vide de type "Any()".

```
my $a;  
say "valeur de a : $a";  
my $b = 0;  
say "valeur de b : $b";
```

```
valeur de a : Any()  
valeur de b : 0
```



Incrémentation d'une jonction.

Lorsqu'on incrémente une jonction, on incrémente ses éléments mais les valeurs de référence ne sont pas affectées.



```
my $a = 0; my $b = 1; my $c = 2; my $d = 3;
my @v = ($a, $b, $c, $d);
my $valeur = all(@v);
"Resultat avant incrementation de la jonction : ".print;
if ($valeur) {
    "Vrai".say
} else {
    "Faux".say
}
"Resultat apres incrementation de la jonction : ".print;
$valeur += 1;
if ($valeur) {
    "Vrai".say
} else {
    "Faux".say
}
```

```
Resultat avant incrementation de la jonction : Faux
Resultat apres incrementation de la jonction : Vrai
```



Incrémentation des valeurs de référence.

Lorsqu'on incrémente les valeurs de référence de la jonction, la jonction n'en est pas affectée.
Les valeurs sont positionnées au moment de la déclaration et ne sont plus remises à jour par la suite.



```
my $a = 0; my $b = 1; my $c = 2; my $d = 3;
my @v;
@v = ($a, $b, $c, $d);
say "Valeur des elements : ", @v.join(" ");
my $valeur = all(@v);
say "Valeur de la jonction : $valeur";
@v = @v >>+<< (1 xx @v.elems);
say "Valeur des elements : ", @v.join(" ");
say "Valeur de la jonction : $valeur";
```

```
Valeur des elements : 0 1 2 3
Valeur de la jonction : all(0, 1, 2, 3)
Valeur des elements : 1 2 3 4
Valeur de la jonction : all(0, 1, 2, 3)
```



Usage de l'opérateur jonction.



La jonction est un opérateur d'une grande efficacité et d'une puissance exceptionnelle lorsqu'il est parfaitement maîtrisé.

Tester que l'intersection de deux ensembles de valeurs est vide s'écrit en utilisant la notion de jonction :

```
if all ($a, $b, $c) == none ($x, $y, $z) {  
    ...  
};
```

Dans un langage plus traditionnel, cette opération aurait du s'écrire :

```
if ($a != $x) and ($a != $y) and ($a != $z) and  
    ($b != $x) and ($b != $y) and ($b != $z) and  
    ($c != $x) and ($c != $y) and ($c != $z) {  
    ...  
};
```



Application.

```
my $a1 = 1; my $b1 = 2; my $c1 = 3; my $d1 = 4;
my $a2 = 5; my $b2 = 6; my $c2 = 7; my $d2 = 8;
"Test de l'intersection de : $a1, $b1, $c1, $d1".say;
"et de                : $a2, $b2, $c2, $d2".say;
if (all($a1, $b1, $c1, $d1) == none ($a2, $b2, $c2, $d2)) {
    say "L'intersection des deux ensembles est vide."
} else {
    say "L'intersection des deux ensembles n'est pas vide."
}
say "";
my $A1 = 1; my $B1 = 2; my $C1 = 3; my $D1 = 4;
my $A2 = 5; my $B2 = 6; my $C2 = 1; my $D2 = 8;
"Test de l'intersection de : $A1, $B1, $C1, $D1".say;
"et de                : $A2, $B2, $C2, $D2".say;
if (all($A1, $B1, $C1, $D1) == none ($A2, $B2, $C2, $D2)) {
    say "L'intersection des deux ensembles est vide."
} else {
    say "L'intersection des deux ensembles n'est pas vide."
}
```



Exécution du programme précédent.

```
Test de l'intersection de : 1, 2, 3, 4  
et de                    : 5, 6, 7, 8  
L'intersection des deux ensembles est vide.
```

```
Test de l'intersection de : 1, 2, 3, 4  
et de                    : 5, 6, 1, 8  
L'intersection des deux ensembles n'est pas vide.
```



Créer ses propres opérateurs

Il arrive, lorsqu'on développe des programmes que l'on ait envie de créer ses propres opérateurs.

Perl 6 propose cette particularité dont la puissance n'est limitée que par l'imagination de celui qui va devoir les concevoir.

Un opérateur se définit comme suit :

```
sub x:<y> (z) { ... }
```

x est la catégorie de l'opérateur : infix, prefix, postfix, circumfix or postcircumfix.
y est la représentation de l'opérateur, n'importe quel caractère y compris Unicode).
z la représentation du ou des opérateurs.

Les opérateurs unaires peuvent être définis comme préfixés ou postfixés (prefix, postfix).

Les opérateurs binaires sont définis comme infixés (infix).

Les opérateurs encadrés, comme les balises html par exemple (<!-- -->), sont définis comme circumfixés (circumfix).

L'option " postcircumfix " est utilisée lorsqu'un postfix est nécessaire.



Exemples simple.



```
sub infix:<~||~> (Int $n, Str $s) {  
  return $s xx $n;  
};  
my $a = 2 ~||~ "cou";  
say $a;
```

```
cou cou
```

Il est possible de créer un opérateur plus complexe,
Par exemple un opérateur +/- auquel sera transmis une valeur numérique et qui nous retournera une jonction de type "any" qui va contenir la valeur positive et la valeur négative du nombre considéré.

```
sub prefix:<+/-> (Int $n) {  
  return +$n|-$n;  
};  
my $a = +/-5;  
say $a;
```

```
any (5, -5)
```

```
sub prefix:<+/-> (Int $n) {  
  return +$n|-$n;  
};  
my $x = 5;  
if ($x == +/-5) {  
  say "C'est bon."  
}
```

```
C'est bon.
```



Autres exemples.

Pour apporter notre pierre au concours \$A++,
créons l'opérateur "Plus Plus".

```
sub postfix:< Plus>($n) {  
    return ($n + 0.5) ;  
};
```

```
my $A = 10;  
say $A Plus Plus;
```

11

Les opérateurs "CopyLeft" et "CopyRight"

```
sub prefix:<:C>( $n) {  
    return "CopyLeft $n"  
};  
sub prefix:<C:>( $n) {  
    return "CopyRight $n"  
};  
say (:C "Chris");  
say (C: "Chris");
```

CopyLeft Chris
CopyRight Chris

L'opérateur ! (factorielle).

```
sub prefix:<!>(Int $n) {  
    if ($n == 1) { return 1; }  
    return $n * !($n-1) ;  
};  
say ("Factorielle 10 : ",!10);
```

```
sub prefix:<!>(Int $n) {  
    [*] 1..$n  
};  
say ("Factorielle 10 : ",!10);
```

Factorielle 10 : 3628800



Un petit exemple d'utilisation des hashes et des jonctions.

Compter les voyelles présentes dans une phrase.

```
my %voy;  
"Quelle est votre phrase ?".say;  
my $line = $*IN.get;  
for $line.split("") -> $l {  
  %voy{$l}++ if ($l eq any (<a e i o u>));  
}  
for %voy.kv -> $k, $v {  
  "$k: $v".say;  
}
```

```
je me figure ce zouave, jouant du xylophone en buvant du whisky...  
e: 7  
i: 2  
u: 6  
o: 4  
a: 3
```

