Introduction au langage Perl

Sébastien Jeudy

(Version V1)

Objectif :

Acquérir les bases du langage de programmation Perl.

Sommaire :

1. Introduction

Perl (www.perl.org) est un langage de programmation **créé par Larry Wall en 1987** pour traiter facilement de l'information de type textuel.

Ce **langage, à la fois compilé et interprété,** s'inspire des structures de contrôle du langage C mais aussi des **langages de scripts Sed, Awk et Shell** (reprise de nombreux principes et syntaxes).

Il prend en charge les **expressions régulières dans sa syntaxe même**, permettant ainsi directement des actions sur des séquences de texte.

Une association, **The Perl Foundation** (www.perlfoundation.org), s'occupe de son devenir et, entre autres, de son éventuel passage de la **version 5.x (la plus répandue, traitée ici)** à la version 6 (rupture de compatibilité, non traitée ici).

Le statut du langage est celui de **logiciel libre**, distribué sous double licence Artistic License et GPL.

Perl a longtemps été le **langage "couteau suisse" des administrateurs systèmes et réseaux**.

Si Python lui fait de plus en plus concurrence sur ce créneau, il **reste encore très utilisé**.

Il peut tout aussi bien servir à écrire des scripts "quick and dirty" (à jeter après utilisation), que des applications complexes grâce aux **très nombreux modules Perl présents sur le MetaCPAN** (https://metacpan.org).

2. Premier programme (en Unix / Linux)

Dans un éditeur de texte brut (fichier « exemple.pl ») :

```
#!/usr/bin/perl
print "Hello World!\n";
```

Dans le terminal :

```
perl exemple.pl
```

Ou :

chmod a+x exemple.pl

./exemple.pl

L'extension « .pl » est traditionnelle pour les programmes Perl mais pas obligatoire.

Le langage Perl est **sensible à la casse** et **ses instructions se terminent par un point-virgule**.

Toute ligne commençant par **'#' est considérée comme une ligne de commentaire** (non analysée par l'interpréteur).

L'interpréteur **Perl compile et exécute le programme en une seule étape** (ne réanalyse pas ses instructions chaque fois qu'il les exécute). C'est pourquoi les messages d'erreur comportent souvent :

« Execution of ./exemple.pl aborted due to compilation errors. »

Fonctionne aussi : perl -e 'print "Hello World!\n";'

3. Document de base

```
#!/usr/bin/perl
use warnings;
use strict;
```

Les lignes autres que le shebang (#!/usr/bin/perl) sont appelées des **pragmas : indications données au compilateur** lui précisant quelque chose à propos du code.

« use warnings » permet d'obtenir des avertissements de la part de Perl lorsqu'il rencontre des éléments suspects dans le programme (messages ne modifiant pas le déroulement du programme).

Perl est un langage extrêmement permissif mais l'emploi du pragma **« use strict » permet de s'imposer une certaine discipline** (déclaration préalable des variables entre autres), donnant un **code plus compréhensible et plus efficace**.

Idéalement, ces deux pragmas doivent être positionnés.

4. Les variables

En Perl, **les variables ne sont pas typées** et il y a basiquement trois structures de données :

- **les variables scalaires**
- **les tableaux**
- **les tables de hashage (tableaux associatifs)**

Une variable scalaire est le nom d'un emplacement ne contenant qu'une seule valeur, au contraire des tableaux ou des tables de hashage.

Par défaut, toutes les variables sont globales, il est donc possible d'y accéder depuis tout endroit du programme.

On peut **créer des variables locales** – dans un bloc de code entre {} – **en les déclarant avec « my »**.
L'usage du pragma **« use strict » force aussi à déclarer toutes les variables avec « my »**.

Une variable ne sera donc globale que si on prend soin de la déclarer en dehors de toute boucle ou sous-programme (fonction).

5. Variables scalaires

Déclaration :

```
my $var = "string";
my $var2 = 42;
my $var3;
```

Une fois la variable déclarée avec « my », il n'est plus nécessaire d'utiliser « my » (cela créerait même un avertissement / warning lors de l'exécution du programme) :

```
my $foo = "bar";
$foo = "baz";
```

Le '$' devant le nom de la variable (déclaration et utilisation) **est appelé un sigil**. Il existe d'autres sigils en Perl : '@' pour les tableaux et '%' pour les tables de hashage.

Comme en Shell Unix / Linux :

" ... " interprète les variables (et les tableaux)

' ... ' n'interprète pas les variables (ni les tableaux)

6. Tableaux

Déclaration :

On peut donner des valeurs à un tableau de deux manières :

- soit **en passant les valeurs en paramètre séparées par des virgules** (et en mettant les chaînes de caractères entre guillemets)
- soit **en utilisant « qw » qui permet de s'affranchir des virgules et des guillemets**

Par contre si une chaîne contient un espace, il faut l'entourer de guillemets.

```
my @tab = ("valeur", 42, "autre", "blip blop");      # à privilégier
my @tab2 = qw(valeur 42 autre "blip blop");
```

Les indices d'un tableau commencent à partir de 0 (les indices d'un tableau de N éléments vont donc de 0 à N – 1).

Utilisation :

Le sigil du tableau **passe de '@' à '$' lorsqu'on accède à un de ses éléments (indice entre [])**.

```
# Attention : c'est bien un '$' pour accéder à l'élément du tableau
print $tab[0];

# On modifie ici la valeur du 1er élément du tableau
$tab[0] = 42;
# $tab[100] = "essai";        # fonctionne (warnings à l'affichage)
```

```
# Affichage du contenu : "4242autreblip blop"
print @tab;

# Affichage du contenu, mais séparé par des espaces :
# "42 42 autre blip blop"
print "@tab";

# Affiche le nombre d'éléments du tableau : 4
print scalar(@tab);
```

```
# Réinitialisation du tableau
@tab = ("valeur", 42, "autre", "blip blop");
```

On voit ici que « @tab » ne renvoie pas forcément la même chose selon le contexte dans lequel on l'utilise. On parle ici de **contexte de liste** (plusieurs valeurs) **et contexte de scalaire** (une seule valeur).

Perl choisira automatiquement la valeur nécessaire selon le contexte dans lequel la variable est utilisée :

```
# Contexte de scalaire :
# 42 + 4 = 46
my $nombre = 42 + @tab;
print "$nombre\n";

# Contexte de liste :
# on recopie ("valeur", 42, "autre", "blip blop") dans @copie_tab
my @copie_tab = @tab;
print "@copie_tab\n";
```

Les opérateurs pop, push, shift et unshift :

Les deux premiers permettent de manipuler aisément les tableaux par leur fin :

« pop » renvoie la valeur du dernier élément du tableau et supprime ce dernier élément :

```
# $last vaut "blip blop" et @tab vaut ("valeur", 42, "autre")
my $last = pop(@tab);
print "$last\n@tab\n";
```

```
# On se contente de supprimer le dernier élément
pop(@tab);
print "@tab\n";
```

« push » ajoute au contraire des éléments au tableau :

```
# @tab vaut ("valeur", 42, "autre", "nouveau")
push(@tab, "autre", "nouveau");
print "@tab\n";
```

Les deux derniers manipulent les tableaux par leur début :

« shift » renvoie la valeur du premier élément du tableau et supprime ce premier élément :

```
@tab = ("valeur", 42, "autre");

# $first vaut "valeur" et @tab vaut (42, "autre")
my $first = shift(@tab);
print "$first\n@tab\n";
```

```
# On se contente de supprimer le premier élément
shift(@tab);
print "@tab\n";
```

« unshift » ajoute au contraire des éléments au tableau :

```
# @tab vaut ("nouveau", 42, "autre")
unshift(@tab, "nouveau", 42);
print "@tab\n";
```

7. Tables de hashage (tableaux associatifs)

Une table de hashage (sigil '%') est une structure de données comme un tableau, en cela qu'elle peut contenir un nombre quelconque de valeurs et les retrouver à la demande.

Cependant, au lieu de repérer les valeurs par un indice numérique comme les tableaux classiques, celles-ci sont **repérées par une clé (nom entre {})**.
Ce sont des **tableaux associatifs, associant des paires de « clé / valeur »**.

Déclaration :

```
my %hash = (
   "cle1" => "valeur",
   "cle2" => 42,
   "cle3" => "autre",
);
```

Il existe d'autres manières d'affecter des valeurs à une table de hashage mais **celle-ci a l'avantage d'être la plus lisible**. Accepte les espaces entre " ... " (clés et valeurs).

Utilisation :

```
print $hash{"cle1"},"\n";          # Affiche "valeur"
$hash{"cle1"} = 42;                # (ré)Affectation d'un élément
# @tab_cles vaut "cle1 cle2 cle3" (ordre quelconque)
my @tab_cles = keys %hash;
# @tab_valeurs vaut "42 42 autre" (ordre des clés)
my @tab_valeurs = values %hash;
print "@tab_cles\n@tab_valeurs\n";
$hash{"autrecle"} = "test";        # Ajout d'un élément
print scalar(%hash),"\n";          # Nombre d'éléments
delete ($hash{"autrecle"});        # Suppression d'un élément
```

8. Les structures de contrôle

Les boucles « for » et « foreach » :

En Perl, « foreach » est un synonyme de « for » (équivalents).

Boucle à la manière du langage C :

```
for (my $i = 1; $i <= 10; $i++) {
   print "$i\n";
}
```

Autre syntaxe (équivalente) :

```
foreach my $i (1..10) {          # aussi avec des lettres (entre '  ')
   print "$i\n";
}
```

Pour les tableaux :

```
foreach my $element (@tab) {
   print "$element\n";
}
```

Pour les tables de hashage (tableaux associatifs) :

```
for my $cle (keys(%hash)) {
   print "$hash{$cle}\n";      # Affiche chaque valeur de la table
}
```

Précision : il est donc aussi possible de remplir un tableau avec des nombres et des lettres (croissants) de façon automatisée...

```
my @tab = (10..20);
my @tab = ('ha'..'pa');
```

Les structures « if » et « unless » :

En Perl, « unless » est le contraire de « if ».

```
if ($a != 42) {
   print "quelque chose\n";
}
# Syntaxe équivalente à :
unless ($a == 42) {
   print "quelque chose\n";
}
```

Il est possible d'utiliser « if » et « unless » d'une autre manière :

```
print "quelque chose\n" if ($a != 42);
print "quelque chose\n" unless ($a == 42);
```

(utilisable que dans le cas où il n'y pas d'action si la condition n'est pas remplie, et où il n'y a qu'une seule action à exécuter)

Et aussi : `if ($a == 42) { print "vaut 42\n"; } else { print "ne vaut pas 42\n"; }`

9. Les fonctions

Sous-routines ou sous-programmes dans la terminologie Perl.

Déclaration de fonctions : mot-clé « sub » obligatoire

```
sub division {
   # Premier argument divisé par le second
   return $_[0] / $_[1];
}
```

Il est aussi **possible de définir une ligne permettant de donner aux arguments des noms** plus explicites :

```
sub division {
   my ($nombre, $diviseur) = @_;
   return $nombre / $diviseur;
}
```

Les arguments passés à une fonction constituent le tableau spécial « @_ ».

On peut donc utiliser ses éléments de façon classique ($_[0], $_[1],...) ou, pour plus de lisibilité, affecter ses éléments à des variables nommées de façon plus explicite.

La syntaxe « **my ($a, $b) = @_; » affecte à $a et $b les valeurs des deux premiers éléments du tableau « @_ »**, sans se soucier de connaître le nombre d'éléments de « @_ » :

Si « @_ » contient plus de deux éléments, les éléments surnuméraires seront ignorés ; s'il en a moins, $b (voire $a si « @_ » a une taille nulle) aura pour valeur « undef ».

Cette syntaxe fonctionne pour tout tableau ou liste de valeurs :

```
my ($var1, $var2) = @tab;
my ($var1, $var2) = ("valeur", 42, "autre", "blip blop");
```

Appel de fonctions :

On utilise l'esperluette **'&' pour indiquer qu'il s'agit d'une fonction** (on peut s'en passer dans certaines circonstances) :

```
print &division(42, 10);        # Affiche 4.2
```

10. Expressions régulières

Perl est **réputé pour les expressions régulières** (regexp).

Rappels : https://fr.wikipedia.org/wiki/Expression_régulière

Recherche :

L'expression régulière recherchée est appelée motif.

On place le motif entre slashes et on utilise l'opérateur « =~ » ("bind" en anglais) pour indiquer dans quelle variable on doit rechercher le motif ("pattern" en anglais).

La recherche de motif renverra "true" ou "false" dans un contexte de scalaire (i.e. une seule valeur).

```
my $foo = "une barre ou deux barres";
if ($foo =~ /bar/) {
   print "\"bar\" a été trouvé dans \$foo\n";
}
# Complètement équivalent :
if ($foo =~ /bar/) {
   print '"bar" a été trouvé dans $foo',"\n";
}
```

```
$foo = "une BARre";
if ($foo =~ /bar/i) {
   print "\"bar\", \"bAr\" ou \"BAR\" a été trouvé dans \$foo\n";
}
```

Le 'i' collé à l'arrière du motif permet de faire une **recherche insensible à la casse**.

Substitution :

Equivalent à l'outil « sed ».

```
my $foobar = "foobarbar";
$foobar =~ s/bar/foo/;
print "$foobar\n";      # Affiche : foofoobar

$foobar = "foobarbar";
# le 'g' permet de ne pas se limiter à la première occurrence
$foobar =~ s/bar/foo/g;
print "$foobar\n";      # Affiche : foofoofoo
```

On peut cumuler les options 'g' et 'i' en 'gi' (et d'autres).

A noter qu'**un certain nombre de caractères doivent être échappés par un antislash** (ex : / ? * ...).

Les classes de caractères :

Les raccourcis suivants peuvent être utiles :

« \d » représente tous les chiffres (équivalent à [:digit:] ou [0-9]).

« \w » représente tous les caractères alphanumériques plus l'underscore, sans les accents (équivalent à [a-zA-Z0-9_]).

« \s » représente tous les caractères d'espacement (espace, passage à la ligne, tabulation, saut de page, retour chariot : équivalent à [\n\t\f\r]).

Extraction de sous-chaînes (entre parenthèses) :

my $url = "http://www.unsite.org/undossier/";

```
my ($site, $dossier) = $url =~ /http:\/\/www.(\w+).org\/(\w+)\//;

print "site : $site\tdossier : $dossier\n";
```

Autre délimiteur que '/' (allège la syntaxe) :

```
($site, $dossier) = $url =~ #http://www.(\w+).org/(\w+)/#;

print "site : $site\tdossier : $dossier\n";
```

11. Consulter l'entrée standard

L'entrée standard, c'est le plus souvent **le clavier**, mais ça peut aussi être **un fichier** (ex : ./mon_prog.pl < mon_fichier) ou la sortie d'**un autre programme** (avec redirection '|').

while (defined(my $foo = <STDIN>)) { print "$foo"; }

Demande à l'utilisateur d'entrer du texte au clavier. Celui-ci peut **cesser les entrées par « Ctrl + D »**. « $foo » contiendra à chaque fois la nouvelle saisie de l'utilisateur avant « Entrée ».

S'il s'agissait d'un fichier (avec : ./mon_prog.pl < mon_fichier), **« $foo » contiendrait les lignes du fichier**, l'une après l'autre.

On utilise **« chomp » pour supprimer le retour à la ligne** qui finalise chaque ligne de texte (quand l'utilisateur tape sur « Entrée », ou le passage à la ligne dans un fichier) :

```
my $foo = "toto\n";
print $foo;      # Affiche "toto" suivi d'un retour à la ligne
```

```
chomp($foo);
print $foo;      # Affiche "toto" sans retour à la ligne
```

Saisie simple :

```
print "Saisissez un texte : ";
chomp(my $saisie = <STDIN>);
print "Vous avez saisi : $saisie\n";
```

12. Manipulation de fichiers

Ouvrir (et fermer) un fichier :

Un descripteur de fichier ("filehandle" en anglais) est le nom, dans un programme Perl, **d'une connexion d'entrée / sortie** entre le processus et l'extérieur (« STDIN », vu plus haut, est un descripteur de fichier spécial).

```
my $fichier = "fichier.txt";          # chemin possible
```

```
# Ouverture en lecture seule
open(my $fh, "<", $fichier)
or die("Impossible d'ouvrir le fichier $fichier : $!\n");

# Ouverture en écriture seule (réinitialise le fichier)
open(my $fh, ">", $fichier)
or die("Impossible d'ouvrir le fichier $fichier : $!\n");

# Ouverture en écriture seule (ajoute à la fin du fichier)
open(my $fh, ">>", $fichier)
or die("Impossible d'ouvrir le fichier $fichier : $!\n");
```

```
# Facultatif (se ferme automatiquement à la fin du programme)
close($fh);
```

Il est également possible d'ouvrir un fichier en lecture / écriture (+> : écrasement du fichier, +< : écrit à la position courante).

Consulter un fichier :

```
# Affiche chaque ligne du fichier
while (defined(my $foo = <$fh>)) { print "$foo"; }
```

Écrire dans un fichier :

```
my $foo = "Quelques lignes\nen plus\n";
# Écrit le contenu de $foo dans le fichier ouvert en écriture
print $fh $foo;
```

Attention : les lectures / écritures déplacent la position courante dans le fichier.

Commandes complémentaires :

Perl dispose aussi de **différentes options de tests sur les fichiers** (idem aux options de tests en Shell Unix / Linux) :
-e, -z, -f, -d, -l, -r, -w, -x, -u, -g,... (ex : if (-e $fichier) ...)

« **unlink** » / « **rename** » **:** supprimer / renommer un fichier
« **copy** » / « **move** » **:** copier / déplacer (renommer) un fichier
« **chdir** » **:** changer de répertoire
« **mkdir** » / « **rmdir** » **:** créer / supprimer un répertoire
« **chmod** » **(Unix / Linux) :** changer les permissions d'un fichier

Explorer un répertoire :

L'opérateur **« glob » permet l'expansion de nom de fichier** exactement comme en Shell et donc d'explorer un répertoire :

```
# Remplit un tableau de tous les noms de fichiers
# du répertoire courant correspondant au motif "*.pl"
my @fichiersPerl = glob("*.pl");
```

(la syntaxe de « glob » n'a **rien à voir avec les regexp**)

13. Fonctions diverses (en vrac)

```
exit;                    # quitte le programme
system ("clear");        # appel de commande système (ici « clear »)

# Si une variable n'a pas été initialisée, elle vaut "undef"
if (defined $foo) {
   print "La variable \$foo a été définie.\n";
} else {
   print "\$foo retourne la valeur undef.\n";
}
```

```
# Décompose $foo dans le tableau @tab d'après le motif
# (ici un espace)
my @tab = split(/ /, $foo);

# Caste $foo en entier
my $numerique = int($foo);

# Retourne un nombre aléatoire entre 0 et $max
my $aleatoire = rand($max);
```

```
# Retourne $foo en majuscules
my $enMajuscules = uc($foo);

# Retourne $foo en minuscules
my $enMinuscules = lc($foo);

# Trie le tableau @foo selon l'ordre asciibétique (1, 10, 2, a,...)
@foo = sort(@foo);

# Trie le tableau selon l'ordre alphanumérique (a, 1, 2, 10,...)
@foo = sort { $a <=> $b } @foo;
```

Quelques raccourcis très utilisés en Perl :

```
for (10..20) {
   print "$_\n";
}
foreach (@foo) {
   print;
}
```

En cas d'absence du nom de variable, « $_ » prend automatiquement le relais.

De plus, en cas d'absence de paramètre, certaines fonctions prennent « $_ » comme valeur par défaut (c'est le cas ici pour le « print » du « foreach »). A éviter pour des questions de lisibilité.

```
while (<>) {
   print;
}
```

Si un fichier est passé en argument à l'appel du script (« ./prog.pl < fichier » ou « ./prog.pl fichier »), l'opérateur diamant '<>' le lira ligne par ligne, sinon l'utilisateur sera invité à taper au clavier.

Transmission de paramètres à l'appel d'un script Perl :

```
my $prem_arg = $ARGV[0];
my $deux_arg = $ARGV[1];
print "arg 1 : $prem_arg\narg 2 : $deux_arg\n";
print "nombre : ",scalar(@ARGV),", tous : @ARGV\n";
```

Perl se passe de la plupart des parenthèses de fonctions :

```
print ("The End\n");
print "The End\n";
```