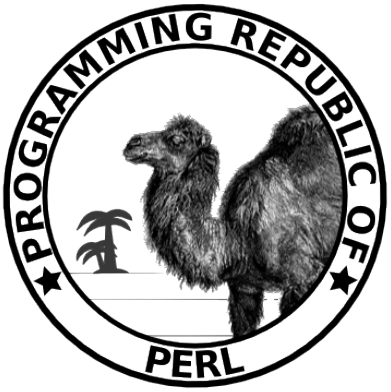
Introduction (rapide) à Perl



Luc DIDRY
Julien VAUBOURG

LP ASRALL
Année universitaire 2009-2010

Université Nancy 2
IUT Nancy-Charlemagne

Table des matières

1 Qu'est-ce que Perl ?

Perl a été créé par Larry Wall au milieu des années 80 parce que *awk* montrait ses limites dans le travail qu'il voulait effectuer.

L'orthographe Perl (avec une majuscule) est le plus souvent employée pour parler du langage tandis que *perl* (en minuscules) fera référence à l'interpréteur.

L'orthographe PERL (comme s'il s'agissait d'un acronyme) est incorrecte.

L'interpréteur Perl compile et exécute le programme en une seule étape. C'est pourquoi les messages d'erreur comportent souvent :

```
Execution of programme aborted due to compilation errors.
```

2 Intérêt de Perl pour un administrateur systèmes et réseaux

Perl a longtemps été *le* langage couteau suisse des adminSys.
Si Python lui fait de plus en plus concurrence sur ce créneau, il reste encore très utilisé et il n'est pas rare d'avoir besoin d'adapter un script Perl provenant d'un précédent adminSys.

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

3 Document de base

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

Les lignes autres que le *shebang* (`#!/usr/bin/perl`), sont appelées des *pragmas*. Ce sont des 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. Ces avertissements ne modifient pas le déroulement du programme mis à part quelques plaintes de temps en temps.

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), ce qui permet souvent d'avoir un code plus compréhensible et plus efficace.

Dans les extraits de code présenté ici, les deux *pragmas* sont positionnés.

4 Les variables

En Perl, les variables ne sont pas typées.

Il y a basiquement trois structures de données en Perl : - les variables scalaires [1] ; - les tableaux ; - les tables de hashage.

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

Par défaut, en Perl, toutes les variables sont globales, il est donc possible d'y accéder depuis tout endroit du programme. On peut créer des variables locales en les déclarant avec `my`.

Cependant, l'usage du *pragma* `strict` nous force à déclarer toutes les variables avec `my`. Une variable ne sera donc globale que si on prend le soin de la déclarer en dehors de toute boucle ou sous-programme.

4.1 Variables scalaires

Exemple de déclarations de variables scalaires :

```
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 lors de l'exécution du programme) :

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

Le `$` devant le nom de la variable est appelé un *sigil*. Il existe d'autres *sigils* en Perl : `@` pour les tableaux, et `%` pour les tables de hachage.

4.2 Tableaux

4.2.1 Déclaration

On peut donner des valeurs à un tableau de deux manières différentes : soit en passant les paramètres séparés par des virgules (et en mettant les chaînes de caractères entre guillemets), soit en utilisant `qw` qui permet de s'affranchir des guillemets et des virgules. Par contre si une chaîne contient une espace, il faudra remettre les guillemets.

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

NB : en informatique, les éléments d'un tableau sont comptés à partir de 0. Les éléments d'un tableau de N éléments sont donc numérotés de 0 à N - 1.

4.2.2 Utilisation

Le *sigil* du tableau passe de `@` à `$` lorsqu'on accède à un de ses éléments.

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

On voit ici que @tab ne renvoit pas forcément la même chose selon le contexte dans lequel on l'utilise. On parle ici de contexte de liste et contexte de scalaire [2].

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

```
# Contexte de scalaire :
#   42 + 4 = 46
$nombre = 42 + @tab;
# Contexte de liste :
#   on recopie (42, 42, "autre", "blip blop") dans @copie_tab
@copie_tab = @tab;
```

4.2.3 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")
$last = pop(@tab);
# On se contente de supprimer le dernier élément
pop(@tab);
```

— `push` ajoute au contraire des éléments au tableau.

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

Les deux derniers manipulent les tableaux par leur début :

— `shift` renvoie la valeur du premier élément du tableau et supprime ce premier élément

```
@tab   = qw(valeur 42 autre);
# $first vaut "valeur" et @tab vaut (42 "autre")
$first = shift(@tab);
# On se contente de supprimer le premier élément
shift(@tab);
```

— `unshift` ajoute au contraire des éléments au tableau.

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

4.3 Tables de hachage (tableaux associatifs)

Une table de hachage 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 avec les tableaux, elles sont repérées par un *nom*.

2. On l'a vu précédemment, le scalaire c'est une seule valeur, la liste c'est logiquement plusieurs valeurs.

4.3.1 Déclaration

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

Il existe d'autres manières d'affecter des valeurs à une table de hachage mais celle-ci a l'avantage d'être la plus lisible.

4.3.2 Utilisation

```
print $hash{"cle1"};          # Affiche "valeur"
$hash{"cle1"} = 42;           # Affectation
@tab_cles     = keys %hash;   # @tab_cles vaut "cle1 cle2 cle3"
@tab_valeurs  = values %hash; # @tab_valeurs vaut "42 42 autre"
```

5 Les structures de contrôle

Les structures **if**, **if else**, **if elsif**, **while**, **do while** fonctionnent comme dans la plupart des langages de programmation.

Mais comme Perl, c'est trop la classe, on peut faire des raccourcis marrants qui font gagner du temps.

5.1 Les boucles `for` et `foreach`

`foreach` est en fait un synonyme de `for`, vous pouvez utiliser indifféramment l'une ou l'autre.

Vous pouvez les utiliser pour faire une boucle à la manière du langage `C` :

```
for (my $i = 0; $i <= 42; $i++) {
    # Attention : print n'effectue pas de retour a la ligne
    # On va donc utiliser \n
    print "$i\n";
}
```

Ou vous pouvez l'utiliser d'autres manières

```
# Tableau particulier (suite de nombres)
# Pour $i de 0 à 42, inclus
foreach my $i (0..42) {
    # Affichage identique que celui de la boucle for vue ci-avant
    print "$i\n";
}

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

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

Le second exemple démontre qu'il est aussi possible de remplir un tableau avec les nombres de `n` à `m` de façon automatisée :

```
my @tab = (42..1337);
```

Mais cela fonctionne aussi avec les lettres !

```
my @tab = (a..z);
```

5.2 Les structures `if` et `unless`

Perl possède la structure de contrôle `unless`, qui est le contraire de `if`.

Affichons quelque chose si la variable `$a` n'est pas égale à 42 :

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

Cette syntaxe est équivalente à :

```
unless ($a == 42) {
    print "quelque chose\n";
}
```

L'utilisation de l'une ou l'autre des structures est au libre choix du programmeur mais il est conseillé de les choisir en fonction de la lisibilité que cela apportera au code.

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);
```

Vous remarquerez que cette syntaxe est relativement proche du langage naturel, améliorant de fait sa lisibilité. Cependant, on ne peut l'utiliser 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.

6 Les fonctions

Celles-ci sont appelées sous-routines dans la terminologie Perl.

6.1 Déclaration

On les déclare avec le mot-clé `sub` qui, contrairement à `my` qui n'est obligatoire que si on utilise le pragma `strict`, doit toujours être utilisé.

```
sub division {
    # Premier argument divise 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;
}
```

Vous aurez remarqué que contrairement à bon nombre d'autres langages, Perl n'utilise pas ce qu'appelle les signatures pour ses fonctions : pas de `sub foo(bar) {}` qui assigne automatiquement les arguments passés à la fonction. Il est possible de les utiliser dans les dernières version de Perl, mais s'agissant d'une fonctionnalité récente, vous ne les rencontrerez que rarement (du moins pour l'instant).

Les arguments passés à une fonction constituent le tableau `@_`. On peut utiliser ses éléments de façon classique (`$_[0]`) 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 `$b` les valeurs des deux premiers éléments de `@_`, sans se soucier de connaître le nombre d'éléments de `@_` : si `@_` contient plus de deux éléments, les éléments surnuméraires seront ignorés, s'il en a moins, `$b` (et `$a` aussi si `@_` a une taille nulle) aura pour valeur `undef`.

6.2 Appel

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

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

7 Expressions régulières

Perl est particulièrement réputé pour sa force sur le traitement des expressions régulières.

Pour la syntaxe des expressions régulières, vous pouvez aller sur http://sylvain.lhullier.org/publications/intro_perl/chapitre10.html (vous y trouverez aussi des exercices).

7.1 Recherche

L'expression régulière recherchée est appelée motif. On place le motif entre slashes [3] et on utilise `=~` pour indiquer dans quelle variable on doit rechercher le motif. La recherche de motif renverra `true` ou `false` dans un contexte de scalaire.

```
if ($foo =~ /bar/) {
    print "\"bar\" a ete trouve dans \$foo";
}
# Complètement équivalent
if ($foo =~ #bar#) {
    print '"bar" a ete trouve dans $foo';
}
```

Vous noterez que les `"` et `$foo` n'ont pas été échappés dans le dernier exemple. En effet, la chaîne de caractère a été délimité par des apostrophes et non par des guillemets. Donc les guillemets n'ont pas à être échappés et `$foo` ne sera pas interprété.

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

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

7.2 Substitution

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

$foobar = "foobarbar";
# le g permet de ne pas limiter la substitution a la premiere occurence
$foobar =~ s/bar/foo/g;
# Affiche : foofoofoo (comme le collège)
print $foobar;
```

On peut également cumuler les options `g` et `i` en `gi`. Il en existe d'autres (pour plus amples informations, RTFM [4]).

Sachez aussi qu'un certain nombre de caractères doivent être échappés par un anti-slash (ex : / [5] . ? * etc.).

3. On peut aussi utiliser d'autres caractères mais on utilise traditionnellement les slashes
4. Read The Fabulous Manual
5. Si le délimiteur de la regex est un /

7.3 Les classes de caractères

Les raccourcis suivants peuvent être utiles[6] :
— `\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]`)

7.4 Extraction de sous-chaînes

Allez, parce que Perl est trop puissant, on en remet une couche !

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

ou, et c'est là qu'on voit l'intérêt d'utiliser un autre délimiteur que `/` :

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

8 Consulter l'entrée standard

L'entrée standard, c'est le plus souvent le clavier mais ça peut aussi être un fichier (si vous faites `$ ./mon_prog < mon_fichier` par exemple) ou la sortie d'un autre programme.

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

Cette syntaxe demande à l'utilisateur d'entrer du texte au clavier. Celui-ci peut cesser de fournir les entrées par Ctrl+D. `$foo` contiendra à chaque fois ce qu'a tapé l'utilisateur avant de valider par Entrée. S'il s'agissait d'un fichier, `$foo` contiendra 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 pour valider son entrée, ou le passage à la ligne dans le cas d'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
```

6. C'est faux : ils ne **peuvent** pas être utiles, ils le **sont** !

9 Manipulation de fichiers

9.1 Ouvrir un fichier

Un descripteur de fichier est le nom, dans un programme Perl, d'une connexion d'entrée/sortie entre le processus Perl et le monde extérieur. `STDIN`, vu plus haut, est un descripteur de fichier spécial.

```
my $fichier = "fichier.txt";
# Ouverture en lecture seule
open(my $fh, "<", $fichier)
    or die("Impossible d'ouvrir le fichier $fichier : $!\n");
# Ouverture en ecriture (reinitialise le fichier)
open(my $fh, ">", $fichier)
    or die( ... );
# Ouverture en ecriture (ajoute a la fin du fichier)
open(my $fh, ">>", $fichier)
    or die( ... );

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

Il est également possible d'ouvrir un fichier en lecture/écriture (`+>` : écrasement, `+<` : ajout).

9.2 Consulter un fichier

C'est bien joli d'ouvrir un fichier, encore faut-il s'en servir.

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

9.3 Écrire dans un fichier

```
# Écrit $foo dans le fichier ouvert en écriture
print $fh $foo;
```

9.4 Se placer dans un répertoire

Par défaut, Perl se place dans le répertoire d'appel du script. Il cherchera donc, par exemple, les fichiers qu'on lui dit d'ouvrir dans ce répertoire.

```
chdir("/home/asrall");
```

9.5 Explorer un répertoire

L'opérateur `glob` permet l'expansion de nom de fichier exactement comme dans le shell et donc d'explorer le répertoire :

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

Notez bien que la syntaxe de `glob` n'a rien à voir avec les regex !

10 Fonctions diverses

Rapidement, quelques fonctions que vous aurez sûrement l'occasion d'utiliser :

```
# 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";
}

if(-d $foo) { print "$foo est un répertoire.\n"; }
if(-e $foo) { print "Le fichier $foo existe.\n"; }
if(-f $foo) { print "$foo est un fichier régulier.\n"; }
# Ces fonctions qui se ressemblent sont les mêmes que
# la commande test de bash (man test pour toutes les voir)

# Décompose $foo dans le tableau @tab d'après le motif (ici une 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, etc.)
@foo = sort(@foo);
# Trie le tableau selon l'ordre alphanumérique (1, 2, 10, a, etc.)
@foo = sort { $a <=> $b } @foo;
```

11 La magie de Perl

Perl permet d'économiser énormément de caractères, voici quelques raccourcis très utilisés dans la communauté Perl.

11.1 La variable magique

```
for (1..42) {
    print "$_\n";
}

foreach (@tab) {
    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).

Attention ! L'utilisation de la variable magique n'est pas conseillée pour un programme destiné à être réutilisé !

Utilisez-la dans vos scripts jetables si vous le souhaitez, mais elle nuit à la lisibilité du programme et donc à ses éventuelles adaptations.

11.2 L'opérateur diamant

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

Selon le contexte, le diamant peut représenter deux choses différentes : si un fichier est passé en argument à l'appel du script [7], le diamant le lira ligne à ligne, sinon l'utilisateur sera invité à taper au clavier.

11.3 Les parenthèses

Perl se passe de la plupart des parenthèses des fonctions. Ces trois lignes sont identiques :

```
chomp($_);
chomp $_;
chomp;
```

12 Plus de Perl

- http://perl.developpez.com/cours/?page=SommaireTutoriels#TutorielsDebuter
- http://sylvain.lhullier.org/publications/perl.html
- http://articles.mongueurs.net/
- *Introduction à Perl* de Randal L. Schwartz, Tom Phoenix et Brian D. Foy *O'Reilly* 2006, ISBN : 2-84177-404-X

Les modules additionnels de Perl sont disponibles sur http://metacpan.org.

7. Attention, cette fois ça peut être de la forme `$ ./mon_prog < mon_fichier` comme tout à l'heure ou `$ ./mon_prog mon_fichier`.

8. http://oreilly.com/pub/a/oreilly/perl/usage/