Programmation éditoriale
II. Introduction à Perl

Éric Guichard*

1er juin 2009

Ce document est une initiation à Perl pour *non-informaticiens*. Il ne se confond donc pas avec les multiples documentations, aides, tutoriels et conseils, en ligne ou imprimés ([2, 1], etc.). J'ai donc ici l'unique intention de détailler des choses très simples et de proposer quelques approches, méthodes et astuces pour les personnes désireuses de se familiariser dans la gaité et l'humour à la programmation. Le prochain tutoriel proposera une introduction à la programmation de pages web dynamiques[1].

Note : la table des matières est à la fin du document.

Références

[1] E. CASTRO – *Perl et CGI pour le world wide web*, PeachPit, Berkeley (CA), Paris, 2001.

[2] R. L. SCHWARTZ, T. PHOENIX & B. D. FOY – *Learning Perl*, O'Reilly, 2008, Fifth Edition.

Une **excellente introduction** dans Linux Magazine (incontournable) :
http://articles.mongueurs.net/magazines/dossiers/01
Ou, du même auteur : http://sylvain.lhullier.org/publications/intro_perl/index.html
Sinon, toujours en français mais plus détaillé :
http://perl.enstimac.fr
Et partout ailleurs sur le web. Par exemple :
http://www.perl.com/pub/q/documentation

*Maître de conférences à l'ENSSIB, responsable de l'équipe *Réseaux, Savoirs & Territoires* de l'École normale supérieure.

[1] Le précédent tutoriel consistait en une initiation (sans douleur) à Unix/Linux.

1 Confort minimal

1.1 Prolégomène

Je conseille fortement de lire et relire cette documentation et celle de *Linux Magazine*. De même pour *Learning Perl*, si vous l'avez sous la main. N'hésitez-pas non plus à lancer (et modifier) les petits programmes que je propose. En informatique, c'est en faisant des erreurs et en multipliant les essais qu'on apprend.

Nous allons utiliser Perl dans une situation simple, qui combine écriture du programme dans un fichier et son lancement dans une fenêtre *terminal*. Autrement dit, vous avez besoin d'avoir deux *fenêtres* ouvertes et deux seulement. Par la suite, je note `pgm1.pl` le fichier dans lequel est écrit le programme.

Pour plus d'informations sur le *terminal*, cf. `http://barthes.enssib.fr/cours/Linux/Linux-1.pdf`. Je rappelle que sur un Mac, le terminal se trouve dans (Applications -> Utilitaires -> Terminal.app) et que tous les logiciels Linux y fonctionnent (souvent) très bien, quand ils ne sont pas déjà pré-installés.

Voici le contenu de ce fichier intitulé `pgm1.pl` :

```
print ("Bonjour \n");
```

En l'occurrence, dans une fenêtre *terminal*, nous saisirons la commande
`perl pgm1.pl`
suivie d'une pression sur la touche *passage à la ligne* (aussi appelée *retour-chariot*) pour transmettre cette commande à la machine et le résultat s'affichera dans la même fenêtre :
`Bonjour`

1.2 Déjà des problèmes ?

Et si ça ne fonctionne pas ? Voici les trois principales causes d'insuccès :

1. Perl n'est pas installé : on vous répond alors `perl : Command not found`. Parfois, c'est moins grave qu'il n'y paraît : Perl est installé, mais son chemin d'accès n'est pas mémorisé par le système. Cela arrive fréquemment sur Windows. En ce cas, il faut préciser où est Perl. Exemple de solution (sans garantie) : au lieu de `perl pgm1.pl`, saisissez
   `C :\Program Files\Perl\perl.exe pgm1.pl`
   Note optimiste : Perl est pré-installé sur Mac (OS X) et Linux.
2. Le fichier `pgm1.pl` n'est pas à l'endroit prévu : dans le *terminal*, à coups de `cd` (*change directory*), positionnez-vous dans le dossier contenant ce fichier `pgm1.pl` (cf. ici encore ma documentation Linux). Pour vérifier votre localisation, saisissez la commande :
   `ls pgm1.pl`
   Si vous obtenez la réponse
   `pgm1.pl`
   (avec éventuellement des choses bizarres devant), c'est tout bon.
   Sinon, on vous répondra :
   `ls : pgm1.pl : No such file or directory.`

3. Votre système d'exploitation a rajouté une extension à votre nom de fichier. Windows s'amuse souvent à faire ce genre de choses. Par exemple vous croyez avoir nommé votre fichier `pgm1.pl`, et il s'appelle en fait `pgm1.pl.txt`, ou pire `pgm1.pl.rtf`, voire `pgm1.pl.doc` !
   D'où le conseil suivant.

1.3 Maîtriser l'écriture avec les *éditeurs*

Demandez au système d'exploitaiton de ne pas ajouter d'extensions incongrues à vos noms de fichiers, sinon de les afficher en clair. Comprenez aussi que les formats de fichier type Word *ne sont pas les bons*, et trouvez sur le web des **logiciels qui vous permettent d'écrire en mode *texte***.

On appelle de tels logiciels des *éditeurs*, et il y en a des centaines. En plus, ils sont souvent gratuits. Sur Windows, *TeXnicCenter* fonctionne très bien (`http://www.texniccenter.org/`). Sur Mac ou Linux, *nedit, Aquamacs, emacs, kwrite, Dashcode, Bbedit* (ce dernier payant sur Mac) sont aussi d'excellents éditeurs. Un éditeur se repère au fait qu'il colorie souvent les commandes importantes du programme que vous écrivez. Plus précisément, il propose une aide visuelle à la syntaxe qui dépend du fichier que vous rédigez (un programme Perl, un texte en LaTeX, etc.).

Ex. :
print "Bonjour" ;
$a = 1 ;

1.4 Convention

Dans tout ce qui suit, la succession des lignes écrites avec une police à espacement fixe (`comme cela`) peuvent être copiées dans le fichier `pgm1.pl` et vous pouvez voir leur effet en saisissant la commande `perl pgm1.pl` dans une fenêtre terminal. Suivant les cas, le résultat s'affichera à l'écran, ou dans un fichier.

N'oubliez pas de sauvegarder le fichier `pgm1.pl` après chaque modification ; sinon elles ne seront pas prises en compte.

2 Les fichiers

2.1 Écrire dans un fichier

L'écriture d'un résultat dans un fichier est simple : on ouvre un fichier, on lui donne un nom simple pour mieux le «tenir» (le *filehandle*, de préférence en majuscules), qui est un alias et on y inscrit ce que l'on désire.

Exemple :

Le symbole > est important !

```
open (F, ">resu.txt") ;
print F "Bonjour \n" ;
close (F) ;
```

Ceci crée un fichier *resu.txt* qui contient le mot *Bonjour* suivi d'un espace et d'un retour-chariot : le **\n** signifie *ajoutez un symbole de fin de ligne à la suite de ce qui précède*.

La mention `close (F) ;` n'est pas indispensable, mais est vivement conseillée et on s'efforcera de *fermer* un fichier ouvert dès qu'on n'en a plus l'usage.

2.2 Lecture ligne à ligne : Perl *et* Unix !

Vous l'avez compris, ce qui distingue une ligne de la suivante est ce `\n` : symbole cabalistique tout à fait compréhensible mais quasiment invisible.

En informatique, ce *passage explicite à la ligne* est de toute première importance. C'est lui qui permet de repérer la fin de la ligne[2].

Nous devons comprendre qu'ici, la littératie de l'imprimé et nos pratiques de lecture déteignent sur la façon dont sera *lu* un fichier électronique : il sera parcouru ligne après ligne et de gauche à droite.

Mais le marqueur de fin de ligne pose aussi quelques soucis : déjà, il y a de fortes chances pour qu'il soit difficile à *attraper*[3] ; ensuite, il *dénature* le contenu intuitif de la ligne (n'oublions pas que les ordinateurs ont une intelligence nulle).

Par exemple, supposons qu'une ligne soit composée des trois prénoms
`Marie Jessica Aurélie`
(séparés comme indiqué par des espaces), et que nous voulions y repérer les prénoms se terminant par «ie». Nous ne trouverons pas *Aurélie*, car la fin d'*Aurélie* n'est pas *ie* mais *ie**n*.

La solution pour cela est la commande `chomp`, qui supprime le dernier caractère de la ligne *à condition* qu'il soit un séparateur de ligne.

Elle est donc plus sélective que la commande `chop`.

2.3 Application : lire un fichier

Un exemple valant mieux qu'un long discours...

Le chemin d'accès `/home/etc.` est bien entendu fantaisiste : modifiez-le à votre convenance (cf. la commande `pwd`).

```
open (F,"/home/chezmoi/dossier_test/monfichier");
while (<F>)
      {
      chomp;
      print $_;
      #toutes les lignes seront affichées à l'écran
      #sans ce fameux retour-chariot
      #Faire d'autres choses...
      }
close (F);
```

Remarques :

1. le `$_` est une variable interne à Perl, qui se confond ici avec la ligne courante. Perl facilitant l'écriture implicite, nous aurions pu écrire aussi :
   `print; #au lieu de print $_;`
   ce qui aurait produit la même chose.
2. Vous l'avez compris, le dièse (`#`) sert de commentaire dans un programme Perl : tout ce qui suit jusqu'à la fin de la ligne n'est pas pris en compte.

[2] Je néglige le fait que certains programmes (de l'ancien temps) ne puissent accepter des lignes de grande longueur : Perl accepte des lignes de longueur quasi-infinie.

[3] D'autant qu'il varie selon les systèmes d'exploitation : LF pour Linux/unix, CR pour Mac, CR/LF (les deux) pour Windows.

2.4 Lire et écrire

Il est aisé (et fréquent) de combiner la lecture d'un fichier et l'écriture de résultats dans un autre fichier. Gardons en tête le fait que pour traiter efficacement et sans surprise un fichier, il faut *supprimer* les caractères de fin de ligne. En revanche, pour que le fichier résultat soit un tant soit peu lisible par les humains que nous sommes, il faudra les *rajouter*.

Voici un exemple. Pour qu'il ne soit pas trivial, je précise que parmi les variables prédéfinies par Perl (comme `$_`), existe la variable `$.` qui donne le numéro de la ligne lue. Avec un *éditeur*, constituez un fichier simple de quelques lignes. Appelez-le *monfichier.txt* ; Nous allons le recopier dans un autre fichier, en précisant à chaque fois le numéro de la ligne :

```
open (F,"/home/chezmoi/dossier_test/monfichier.txt"};
open (RESU,">/home/chezmoi/dossier_test/resultat.txt"};
while (<F>)
     {
     chomp;
     print RESU $_;
     #imprime dans RESU la ligne SANS le retour-chariot
     print RESU " se trouve en ligne $. \n";
     }
close (F);
close (RESU);
```

Si, par exemple, la cinquième ligne du fichier *monfichier.txt* est :
La nuit est bien noire ce soir
La cinquième ligne du fichier *resultat.txt* sera :
La nuit est bien noire ce soir se trouve en ligne 5
Et derrière le chiffre «5», nous trouverons un espace, et un passage à la ligne.

Remarques

1. La commande `print` sans alias de nom de fichier derrière, affiche donc ce que l'on désire *sur l'écran du terminal* ; suivie d'un alias (ex. : `print RESU`), elle inscrit ce que l'on désire *dans le fichier qui a cet alias*.

   L'alias est ce fameux *filehandle*.

2. Les guillemets doubles permettent de combiner du texte *et* des variables (telles que `$.`, `$_`, etc.) ; seront écrits le texte *et* la valeur de la variable (et non pas les symboles qui la composent). Ceci n'aurait pas été le cas si nous avions utilisé des guillemets simples (apostrophes) : alors, les variables n'auraient pas été *interpolées*.

   Les guillemets simples «'» sont appelés *quotes* en anglais. Les guillemets doubles «"», *double quotes*.

3 Les variables

Il y a dans Perl trois grands types de variables : les variables scalaires (*scalar*), les listes (*array*) et les tableaux indexés (*hash*).

Il est conseillé de préciser le statut relatif des variables, mais ce n'est pas obligatoire pour les programmes simples : une variable peut être (définie comme) *locale* (`my`) ou *globale* (`our`).

Une variable sans précision est par défaut considérée comme globale.

3.1 Variables scalaires

Une variable dite *scalaire* est la plus petite *boîte* imaginable. Il n'est pas nécessaire de préciser la *destinée* de la variable (si elle contiendra du texte, un nombre, une commande, etc.). C'est d'ailleurs une des forces de Perl.

Une variable scalaire se repère au signe `$` qui précède son intitulé :

```
my $couleur="vert";
my $a = 3.5;
my $phrase ="le petit chat est rouge";
```

Il est aisé de réaliser des opérations sur les variables. Exemples (simples) :

```
my $phrase ="le petit chat est $couleur";
my $b= 2*$a; #$b vaut évidemment 7
$phrase = $phrase x 2;
#$phrase contient alors deux fois le texte initial;
```

Remarque

Au début, évitez de modifier les variables prédéfinies de Perl (comme `$_`, `$.`, etc. Vous risqueriez de drôles de surprises.

3.2 Listes

3.2.1 Approche élémentaire

Une liste est une succession (une... liste) de variables. On peut l'assimiler à un vecteur. Une liste se repère au signe `@` qui précède son intitulé :

```
my @liste_a_moi=(2, huit, chat, rouge, $phrase);
```

Cette liste est composée de 4 éléments, dont le premier a pour indice (numéro) 0. Ici, le dernier a donc l'indice (je dis aussi *dossard* ou *index*) 3. Pour récupérer le second élément (*huit*), c'est tout simple, si on se souvient du fait qu'une liste est composée de variables scalaires : le «nom» de cet élément commencera par $, il aura celui de la liste, et il sera repéré par son index.

En informatique, pour des raisons tenant aux mathématiques et à la logique, le premier élément a souvent pour numéro de *dossard* 0.

```
my @liste=(2, huit, chat, rouge, $phrase);
my $second= $liste[1]; #$second vaut huit
print $second;
#ou
print $liste[1],"\n";
#pour récupérer $phrase:
my $nouveau= $liste[4];
```

Une liste peut être assez implicite :

```
my @liste_a_moi=(1..10); #contient 1, 2, 3,... 10
@liste_a_moi=(a..z,2, huit, chat);
```

Le premier élément d'une liste ayant pour numéro 0, le dernier aura pour numéro la longueur de la liste -1. Perl a une variable spéciale pour le référencer : `$#liste;`

On peut réaliser une multitude de traitements sur une liste. Je renvoie aux documentations, et ne présente ici que l'instruction la plus fréquente : **foreach**.

3.2.2 Deux exemples avec *foreach*

1. Cas le plus simple :

```
my @liste_a_moi=(2, huit, chat, rouge, $phrase);
foreach my $nimportequoi (@liste_a_moi)
     {
     #la variable $nimportequoi est automatiquement
     #créée et va prendre successivement
     #toutes les valeurs des élements de la liste.
     print $nimportequoi,"\n";
     #on affiche à l'écran ces éléments;
     }
```

2. Si on veut attraper les éléments pairs de la liste.

Dans cet exemple, on utilise le signe `%`, qui signifie «modulo».

```
foreach my $dossard (0..$#liste_a_moi)
     {
     print $liste_a_moi[$dossard],"\n" if $dossard % 2 ==1;
     #Un élément de la liste sera donc imprimé si le reste
     #de la division par deux de son numéro (index)
     #vaut 1 (rappel: le second élément a pour numéro 1).
     }
```

Remarque

Si on est exigeant en matière syntaxique, on enchâsse les éléments textuels de la liste en des apostrophes (*quotes*) :

Ou si les éléments de la liste l'imposent (fragments de phrase, etc.).

```
@liste_a_moi=(a..z,'2', 'le chat');
```

3.2.3 Liens entre listes et variables

1. Construire une liste à partir d'une variable : la commande `split`.
   On choisit les termes qui vont servir de séparateurs et on obtient une liste. Dans l'exemple qui suit, les séparateurs sont les espaces, cités entre les deux signes `/` :

```
$phrase="voici quelques mots";
@liste_a_moi=split(/ /,$phrase);
#@liste_a_moi est alors ('voici','quelques','mots');
#On le vérifie ainsi:
foreach my $cequimepasseparlatete (@liste_a_moi)
     {
     print $cequimepasseparlatete,"\n";
     }
```

Que se passe-t-il si le séparateur est vide ? Tous les termes de la liste sont coupés par... rien, et on obtient tous les caractères de `$phrase`. Cet exemple, *a priori* stupide, est **fort utile pour repérer les erreurs dans les OCR** (reconnaissance optique des caractères), où les confusions entre les «1» (un) et les «l» (lettre L), entre les «O» et les «0» sont fréquentes.

```
@liste=split(//,"voici quelques mots");
foreach my $u (@liste)
     {
     print "début",$u,"fin\n";
     }
```

2. Construire une variable à partir d'une liste : la commande `join`.
   Le principe est le même : on *joint* les termes de la liste par un séparateur.

```
@liste=('voici','quelques','mots');
$liant=" TT ";
$a= join($liant,@liste);
print $a,"\n";
#Plus rapide:
print join(" TT ",@liste);
```

3.3 Les tableaux indexés ou *hash(es)*

On peut voir un tableau indexé comme une double liste, ou comme une liste de listes. Cela peut être un vrai tableau (une matrice), ou un objet plus complexe (tri-dimensionnel, etc.). Cependant, le *hash* renvoie à du bon sens. Aussi, avant d'en détailler la syntaxe et l'usage, je vais commencer par un petit détour.

Je préfère le terme anglais *hash* car certaines documentations françaises traduisent *list* par «tableau». Par suite, on ne sait jamais si un tableau est indexé ou pas...

3.3.1 Décrire un tableau

Par exemple, comment repérer simplement le tableau suivant ?

```
0     1     Bonjour
2     3     4
1     0     $phrase
```

Il suffit de décrire les valeurs du tableau en fonction de leurs positions : par exemple, en ligne 1 et colonne 1, j'ai `0`. En ligne 2 et colonne 3, j'ai `4`.

On a alors envie d'écrire :

```
$tableau{ligne1col1}= 0;
...
$tableau{ligne2col3}= 4;
```

etc.

En d'autres termes, pour chaque *clé* (*lignexcoly*), on définit une *valeur*. Nous remarquons là que n'importe quel tableau, aussi complexe soit-il, peut être décrit ainsi, et donc être *réduit* à un tableau à deux colonnes : la première pour les clés, la seconde pour leurs valeurs. Et la matrice précédente peut être ainsi décrite :

clé	*valeur*
ligne1col1	0
ligne1col2	1
...	
ligne2col3	4
...	
ligne3col3	$phrase

C'est exactement le sens d'un *hash* : une succession de clés et de valeurs.

3.3.2 Définir un *hash*

Un *hash* est repéré par le symbole %, et il y a mille et une manières de le définir :

1. comme nous venons de le faire : en écrivant
   ```
   $tableau{ligne1col1}= 0;
   #etc.
   ```
   Ceci a défini le *hash* intitulé `%tableau`.

   Il aurait été sage d'écrire auparavant dans le programme : `my %tableau;`

2. Un peu comme une liste :
   ```
   %tableau =(ligne1col1, 0, ligne1col2, 1); #etc.
   ```
   Bien sûr, on a avantage à l'écrire plus clairement :
   ```
   %tableau =('ligne1col1', '0',
   'ligne1col2','1');
   ```
3. Et la façon la plus courante depuis la version 5 de Perl :
   ```
   %tableau =(ligne1col1 =>0, ligne1col2 => 1);
   ```

Un *hash* est donc une double liste : liste de *clés*, et listes de *valeurs* associées. C'est un peu comme une fonction mathématique : à chaque clé est associée une valeur et une seule. Bien sûr, plusieurs clés peuvent avoir la même valeur.

Mais l'analogie s'arrête là : les objets ressemblant le plus à des fonctions mathématiques sont les (sous-)routines, définies par le mot-clé `sub`.

De ce fait, il est aisé de repérer les deux listes qui composent le *hash* : elles s'appellent tout simplement *keys* et *values*. Exemples :

```
@liste_de_mes_cles= keys %tableau;
@ma_seconde_colonne= values %tableau;
```

```
foreach my $bazar (keys %tableau)
     {
     print $bazar, " ", $tableau{$bazar},"\n";
     }
```

Remarque

Un tableau indexé est un instrument très puissant. On peut par exemple fabriquer des tableaux de tableaux. **Cependant, un *hash* est toujours désordonné** : les valeurs sont certes bien attachées à leurs clés, mais les listes des unes comme des autres sont toujours renvoyées *dans un ordre peu compréhensible*. Il existe des solutions, en demandant à trier (par ordre alphabétique, numérique, ou construit par ses soins) ces clés ou valeurs. Exemple :

```
foreach my $bazar (sort keys %tableau)
     {
     print $bazar, " ", $tableau{$bazar},"\n";
     }
```

Question : que se passe-t-il si j'écris ?
`%tableau = @liste;`
ou l'inverse ?
`@liste= %tableau;`

Réponse : ce que le bon sens nous indique... Ce qui est agréable avec Perl, c'est qu'on peut oser un peu tout ce qu'on veut. Quitte à multiplier les essais, et à consulter les documentations.

3.3.3 Construction d'un *hash* au fil de l'eau

Ce point précise comment définir implicitement un *hash*. Supposons par exemple que nous voulions repérer tous les caractères d'une phrase et seulement eux (pensons ici encore à l'OCR). Il suffit de faire comme suit :

```
$phrase="Quoi? Vous voulez la réponse à 2+2?";
foreach my $bazar (split(//,$phrase)) # je vais vite...
     {
     $tableau{$bazar}=1;
     }
```

On aurait pu faire plus simple avec une liste (commande `push`), mais l'idée est ici de montrer comment définir des *ensembles*.

Par ce moyen, on a donc créé un *hash* nommé `%tableau` qui contient comme clés les caractères de la phrase (lettres, espace et ponctuation), et comme valeurs le nombre *1*. Auparavant, il n'existait pas. Et maintenant, ses clés et valeurs sont exactement ce que nous avons voulu qu'elles soient.

Pour voir le contenu de ce tableau, il nous suffit d'écrire :

```
foreach $u (keys %tableau)
     {
     print "le caractère $u apparaît dans $phrase \n";
     }
```

ou, plus simplement :

```
print join("TTT",keys %tableau);
```

Comme je le disais, cette liste est bien désordonnée. Vous voulez un tri ?

```
print join("TTT",sort keys %tableau);
```

Remarques

1. Le «Q» apparaît avant le «a», le «à» après le «z». Effectivement, ce tri simple (*sort*) dépend du classement informatique (d'abord les nombres, ensuite les Majuscules, après les minuscules, puis les caractères accentués). Il est possible de produire des tris plus subtils.
2. J'ai été un peu vite en besogne, mais c'est pour que vous preniez conscience de tous les implicites rendus possibles par Perl. Voici ce que donnerait le même programme détaillé :

```
$phrase="Quoi? Vous voulez la réponse à 2+2?";
@liste= split(//,$phrase);
foreach my $bazar (@liste)
     {
     $tableau{$bazar}=1;
     }
foreach $u (sort keys %tableau)
     {
     print "le caractère $u apparaît dans la ligne \n";
     }
```

On peut évidemment améliorer un tel tableau, par exemple en comptant le nombre d'apparitions des caractères. Comment faire ? Ici encore, il est inutile de *prédéfinir* le spectre du tableau. Nous allons le construire au fur et à mesure :

```
$tableau{$bazar}=$tableau{$bazar} + 1;
#ou mieux:
$tableau{$bazar}+=1;
#j'ajoute 1
#ou encore
$tableau{$bazar}++;
```

Synthèse

```
my %tableau;
$phrase="Quoi? Vous voulez la réponse à 2+2?";
@liste= split(//,$phrase);
foreach my $bazar (@liste)
     {
```

Le `my %tableau` sert à réinitialiser le tableau, si jamais vous l'avez déjà utilisé dans un exemple précédent. j'aurais pu écrire : `undef %tableau;`

```
      $tableau{$bazar}++;
      }
foreach $u (sort keys %tableau)
      {
      print "le caractère $u apparaît $tableau{$u} fois \n";
      }
```

3.4 Vive l'autonomie

À partir de maintenant, vous avez de solides compétences : vous pouvez vous plonger dans les documentations et dictionnaires relatifs à Perl, explorer les limites d'interpolation des variables, découvrir seul(e) les instructions de contrôle (`if, while, until...` vous connaissez déjà `foreach`), tester par vous-même ce qui est syntaxiquement possible et ce que produisent ces syntaxes.

4 Motifs

J'appelle motif (textuel) toute forme graphique (donc textuelle) composée de tout et n'importe quoi : lettres, nombres, signes de ponctuation, etc. La traduction anglaise de motif est *pattern*. Un motif peut servir à décrire les mots commençant par une majuscule ou finissant par *aît*, des url (`<a href="...">`, des nombres (comme 2,5 ou -16.87), etc.

4.1 Substitution

L'opération la plus simple consiste à réaliser une sorte de recherche/remplacement. Elle s'applique à des variables et sa syntaxe est la suivante (`s` comme *substitute*) :

```
s/recherche/remplace/;
```

En pratique, on précise la variable sur laquelle opérer la transformation, suivie de l'opérateur `=~` (un signe «égal» suivi d'un signe «tilde»). Cet opérateur (le *bind*) est essentiel. Exemple :

Le «tilde» s'obtient avec les touches «Alt-n» (Mac) ou «Alt_Gr-2» (Linux, alors suivies d'un espace pour visualiser ce *flottant*).

```
my $a="Bonjour tout le monde";
$a=~s/o/w/;
print $a,"\n";
#réponse: Bwnjour tout le monde
```

La subtitution accepte divers paramètres. Le plus fréquent est le `g` : il garantit que le remplacement va s'opérer partout où c'est possible.

```
$a="Bonjour tout le monde";
$a=~s/o/w/g;
print $a,"\n";
#réponse: Bwnjwur twut le mwnde
```

On a envie de faire plus sophistiqué :

```
$a=~s/jou/soi/g;
#donnera: Bonsoir tout le monde
```

Voire de tenter un *joker*. C'est possible, et en Perl (comme souvent en informatique), l'invocation de n'importe groupe de caractères (vide inclus) se fait via la succession des deux caractères `.*` : Le `.` signifie *n'importe quel caractère* (sauf le passage à la ligne) et le `*` signifie *apparaissant 0, 1 ou plusieurs fois*.

Un exemple valant mieux que de longs discours :

```
$a="Bonjour tout le monde";
$a=~s/j.*r/soir/g;
#ce qui commence par j et finit par r sera remplacé par soir
print $a,"\n";
#réponse: Bonsoir tout le monde
```

4.2 Repérage d'un motif

On peut aussi repérer un tel motif. Cela se fait avec le `m` (comme *match*, correspondance), et la syntaxe est la même. Souvent, un repérage est associé à une condition :

```
$a="Bonjour tout le monde";
if ($a=~m/jour/)
     {
     #si $a contient "jour"
     print "il doit fait jour car j'ai entendu Bonjour\n";
     }
else
     {
     #sinon
     print "il doit faire nuit\n";
     }
```

La plupart du temps, on oublie le `m` :

```
$a="Bonjour tout le monde";
if ($a=~/j.*r/)
     {
     print "il doit fait jour\n";
     }
else
     {
     print "il doit faire nuit\n";
     }
```

Remarque

Bien sûr, il faut être prudent : on peut avoir des surprises car notre *joker* est gourmand par nature. Exemple :

```
$a="Bonjour tout le monde";
$a=~s/j.*u/soi/g;
print $a,"\n";
#réponse: Bonsoit le monde
```

4.3 Variables prédéfinies associées

Quand un tel repérage est fait, avec les commandes `s` ou `m`, Perl garde en mémoire le motif sélectionné (variable `$&`), ce qui le précède (`$``) et ce qui le suit (`$´`). L'usage de ces trois variables est déconseillé par certains experts car elles ralentissent l'exécution du programme. Mais je les trouve fort utiles pour les débutants.

L'apostrophe inverse ` n'est pas toujours facile à saisir au clavier... Dans tous les cas, faites la suivre d'un espace, elle risquera moins de disparaître.

Exemple :

```
$a="Bonjour tout le monde";
$a=~s/o/w/;
print "début: $`
motif trouvé: $&
fin: $' \n";
```

J'en profite ici pour montrer la souplesse du `print` et rappeler l'interpolation de variables entre les guillemets.

Le résultat affiché sera :

```
début: B
motif trouvé: o
fin: njour tout le monde
```

On comprend que Perl s'arrête ici au *premier* «o» rencontré. Ce serait l'inverse avec le programme suivant :

```
$a="Bonjour tout le monde";
$a=~s/o/w/g;
print "début: $` motif trouvé: $&
fin: $' \n";
#Résultat affiché ci-dessous:
début: Bonjour tout le m motif trouvé: o
fin: nde
```

5 Les expressions régulières

Elles généralisent ce que j'ai introduit au point 4. Les expressions régulières constituent la force de Perl, au point que beaucoup d'autres *langages de programmation* les ont intégrées (ex. : Php).

En français orthodoxe, une expression régulière est appelée «expression rationnelle». Ses diminutifs anglais sont *regex* ou *regexp*.

Elles ouvrent la porte de la programmation aux experts en sciences humaines. Ici encore, je renvoie aux documentations, bien plus détaillées que ce cours, et me limiterai à quelques exemples.

5.1 Philosophie

Il y a deux façon de considérer les expressions régulières. On peut penser que c'est une horreur sans nom, et qu'il faut des années pour se familiariser avec. L'autre approche est plus empirique : on commence par se familiariser avec les plus simples, et on développe une série de méthodes personnelles. N'hésitez pas à fragmenter votre travail en utilisant plusieurs expressions régulières simples et *lisibles* plutôt que chercher la superbe expression régulière qui pourrait produire en une ligne ce que les autres font en dix.

5.2 Quelques formes courantes d'expressions régulières

Je présente ici un nombre fort réduit de syntaxes d'expressions régulières et de raccourcis des précédentes, en confondant volontairement les deux. Ici encore, la documentation en ligne ou imprimée est exhaustive.

- `\w` un mot : ce qui contient des lettres (non accentuées dans le cas général), des chiffres, et le symbole _
- `\W` tout ce qui n'est pas un mot (souvent la ponctuation, et parfois les lettres accentuées...)
- `\d` un chiffre (entre 0 et 9) ; équivalent à `[0-9]`
- `^` désigne le début de la cible. Impose souvent que l'expression régulière qui le suit apparaisse dès le début de la ligne
- `[des_caractères]` repère l'un des caractères cités entre le `[` et le `]`. Ce type d'expression, comme le suivant, est *souvent suivi d'un* $\star$ ou d'un + (pour signifier : apparaissant 0 ou plusieurs fois, une ou plusieurs fois)
- `[^des_caractères]` exclut tous les caractères cités entre le `[` et le `]`

Voir le point 6.2.1 pour en savoir plus sur la gestion des accents.

5.3 Exemples simples

5.3.1 Usage direct

```
$a="Bonjour tout le monde; il est 17h30";
if ($a =~/\d+h\d+/)
     {
     #si $heure contient des chiffres, un "h" et d'autres chiffres
     $heure =$&;
     print $heure,"\n";
     #affiche 17h30
     }
```

```
if ($a =~/^B.*r/)
     {
     #si $a commence par un B et contient un r
     print "la phrase commence par $& \n";
     #affiche: la phrase commence par Bonjour
     }
```

Attention ! Piège possible `$&` renvoie le dernier motif non vide. Ce n'est pas toujours ce que vous espérez. Exemple :

```
$a="Bonjour tout le monde; il est 17h30";
$heure =$& if ($a =~/\d+h\d+/);
#$heure vaut: 17h30
if ($a =~/^C.*r/)
     {
     #si $a commence par un C et contient un r
     print "la phrase commence par $& \n";
     #affiche: la phrase commence par 17h30
     #(l'ancienne valeur de $& ...)
     }
```

5.3.2 Avec la commande `split`

Nous allons rechercher les «mots» de la phrase *Bonjour ! dit-il à tous. Il est 17h30*. Il nous faut donc utiliser les espaces et la ponctuation comme séparateurs et les intégrer dans une expression régulière. Voir les remarques ci-dessous.

```
my $a="Bonjour! dit-il; il est 17h30...";
my @listemots=split(/[ !;\.-]+/,$a);
print join ("TT", @listemots);
#affiche: BonjourTTditTTilTTilTTestTT17h30
```

Remarques

1. Certains caractères pouvant entrer dans la définition d'expressions régulières doivent être *protégés* : on les précède d'un «\». C'est le cas du «`.`», qui devient «`\.`» dans l'expression régulière. Les autres caractères à protéger sont : `+ ? * ^ $ ( ) [ ] |` et `\`.
2. Comme le tiret intervient aussi dans la formulation d'une expression régulière, il faut lui aussi le protéger, sinon le mettre à la fin de l'expression, comme dans l'exemple précédent.

5.4 Affectation directe

On peut mettre en mémoire ce qui est repéré sans passer par la variable `$&`, si on le met entre parenthèses. La syntaxe est un peu plus délicate. Exemples :

```
$a="Bonjour tout le monde; il est 17h30";
(my $b)= $a =~/(^B.*r)/;
#ce qui suit est mieux:
($b)= ($a =~/(^B.*r)/);
print $b,"\n";
#affiche dans les deux cas: Bonjour
```

Attention : la parenthèse autour de `$b` est essentielle ! Voyons ce qui se passe si on l'oublie :

```
my $b= $a =~/(^B.*r)/;
print $b,"\n";
#affiche 1 (témoin de la correspondance: valeur "vrai")
$b= $a =~/(^C.*r)/;
print $b,"\n";
#N'AFFICHE RIEN!  (pas de correspondance: valeur "faux")

(my $heure)= ( $a =~/(\d+h\d+)/);
print $heure,"\n";
#affiche 17h30
```

On peut aussi être plus direct, en intégrant toutes les occurrences repérées par *les* motifs entre parenthèses dans une liste. Certes, l'expression régulière devient complexe ; mais une solution de simplification sera proposée au point 5.5.

```
($b, $heure)= ( $a =~/(^B.*r).*(\d+h\d+)/);
print $b," ",$heure,"\n";
#affiche Bonjour 7h30, ce qui est normal car le chiffre 1
#a été absorbé par le .*
```

Solution :

```
($b, $heure)= ( $a =~/(^B.*r)[^\d]*(\d+h\d+)/);
#[^\d]* signifie n'importe quoi, tant que ce n'est pas un chiffre
print $b," il est ",$heure,"\n";
#affiche Bonjour il est 17h30
```

Si vous désirez en savoir plus, vous pouvez aussi regarder la documentation sur les variables prédéfinies `$1`, `$2`... `$9`.

5.5 Intégrer une expression régulière dans une variable

Comme tout cela devient compliqué, je propose une solution personnelle pour manipuler plus aisément les expressions régulières en les glissant dans une variable. Placée aux bons endroits, cette dernière sera interprétée comme nous le désirons.

5.5.1 Méthode

Commençons par un exemple simple, où l'expression régulière correspond à une seule variable. Nous devinons qu'il nous faudra *protéger* les caractères spéciaux avec un \ :

```
$a="Bonjour tout le monde; il est 17h30";
my $expr1= "\^B\.*r";
print $& if $a=~/$expr1/;
#renvoie Bonjour
my $expr2= "\\d+h\\d+";
print $& if $a=~/$expr2/;
#renvoie 17h30
```

C'est déjà agréable de pouvoir réduire une expression régulière rarement lisible dans une variable au nom explicite. Cela nous donne alors l'idée de décomposer en morceaux notre précédente expression régulière. Je la rappelle ci-après :

```
(^B.*r)[^\d]*(\d+h\d+)
```

Nous voulons prendre en considération :

1. ce qui commence par un B et finit par un r, à mettre entre parenthèses pour l'intégrer dans une variable ;
2. ce qui, ensuite, ne contient pas de chiffre et dont on se moque (parenthèses inutiles) ;
3. enfin, ce qui correspond à l'heure (qui entrera dans une variable).

La fortune souriant aux audacieux, tentons ce qui suit :

Pour mémo, les noms des variables à retenir (expressions régulières entourées de parenthèses) se terminent, par un *m*.

```
my $regexpdebutm="(^B\.*r)";
my $nondecimal="[\^\\d]*";
my $regexpheurem="(\\d+h\\d+)";
#concaténons les 3 expressions régulières
# Cela se fait avec le point .
my $regexp_pas_belle= $regexpdebutm.$nondecimal.$regexpheurem;
my ($b, $heure)= ($a=~/$regexp_pas_belle/);
print $b," ",$heure,"\n";
#affiche Bonjour 17h30
```

Avouons-le, cette solution n'est pas idéale, mais elle est fort utile quand on se perd avec les protections, surtout quand nous proposons à l'utilisateur de choisir lui-même ses séparateurs.

Vous avez remarqué qu'ici, il n'était pas indispensable de protéger les parenthèses.

5.5.2 Application à une liste complexe

Voyons un dernier exemple avec `split`, qui rend plus lisible la liste des séparateurs à utiliser :

```
my $separateurs="[ #début de la série des séparateurs de mots
, #je n'écris qu'un signe de ponctuation par ligne
; #pour garantir la lisibilité
\.
\(
\)
\?
!
 #ici l'espace
]+"; #fin de ma série des séparateurs de mots

$separateurs=~s/\#.*\n//g; #supprime les commentaires
$separateurs=~s/\n//g; #supprime les passages à la ligne
print $separateurs,"\n";
#affiche [ , ; . ()?! ]+
#certes, il y a des espaces en trop, mais ce n'est pas un souci

my $a="Bonjour à tous! il est 17h30 (enfin, je crois).";
my @listemots=split (/$separateurs/,$a);
foreach my $mot (@listemots)
     {
     print $mot," ";
     }
print "\n";
```

Ce programme affiche à l'écran :

```
Bonjour à tous il est 17h30 enfin je crois
```

6 En savoir plus

Cette partie sert de conclusion : si vous avez lu jusqu'ici cette documentation, et que vous en avez testé tous les exemples, vous pouvez sans souci vous plonger dans celles que je vous ai conseillées au début de cet article.

6.1 Bibliothèques et modules

Il existe des milliers de bibliothèques (ou modules) dans le moteur même de Perl, et vous n'avez donc pas à les télécharger. Très bien écrits, ces modules vous déchargent de l'écriture de dizaines ou de centaines de lignes de code.

En général, pour les solliciter, il vous suffit de saisir *au début de votre programme* la commande `use` suivie du nom du module. Exemples :

```
use locale;
use strict;
use CGI::Carp;
```

Vous n'avez pas à comprendre comment ces modules sont écrits. Il vous suffit de savoir ce qu'ils font. Ils sont aussi documentés : saisissez par exemple dans un terminal la commande `perldoc locale`.

6.2 Application : créer vite le lexique d'un fichier

6.2.1 Première étape

Nous avons vu précédemment comment repérer les mots d'un fichier, en prenant comme séparateurs la ponctuation et les espaces. *A priori*, cette démarche est inutile, puisque l'expression régulière `\W` remplit cette fonction. Or, cela n'est pas satisfaisant avec les caractères accentués. Exemple :

```
my $a="Bonjour à tous!";
my @listemots=split (/\W/,$a);
print join("T",@listemots),"\n";
#affiche BonjourTTTtous
```

Non seulement le *à* a disparu de nos mots, mais en plus, des mots vides apparaissent. Cet exemple semble prouver que notre démarche précédente était la bonne. Mais il y a une solution plus rapide :

```
use locale;
my @listemots=split (/\W/,$a);
print join("T",@listemots),"\n";
#affiche BonjourTàTtous
```

locale est un élément de votre système d'exploitation qui précise votre environnement linguistique et votre encodage. Saisissez `man locale`, `locale` et `perldoc locale` pour en savoir plus. Pour imposer un environnement en *latin1*, saisissez la commande suivante dans une fenêtre *terminal* : `LANG= fr_Fr`

6.2.2 Le programme final

```
use locale;
open (F, "le_fichier_dont_je_veux_le_lexique");
open (G,">lexique.txt");
while (<F>)
     {chomp;
     $_=lc($_); #met tout en minuscules(lowercase)
     @l=split(/\W/);
     foreach $u (@l)
          {next if $u eq ""; #conseillé
          $freq{$u}++;
          }
```

```
     }
close (F);
foreach my $u (sort tridecroissant keys %freq)
     {print G "\"$u\" a pour fréquence $freq{$u}\n";
     }
close (G);

sub tridecroissant #voir le paragraphe routines
     {$freq{$b} <=> $freq{$a};}
```

6.3 Routines

Ce sont en quelque sorte des fonctions ; elles peuvent s'appliquer à plusieurs variables, de types scalaires ou listes ; mais alors les scalaires doivent être citées en premier (sinon la variable la plus sophistiquée absorbe la plus simple). En général, les routines renvoient des variables scalaires ou des listes, mais il y a moyen de se débrouiller pour qu'elles renvoient aussi des *hash*es.

On peut définir des fonctions qui travaillent sur des tableaux indexés, mais c'est plus technique : voir les *références*.

Je donne quelques exemples simples, qui évoquent en même temps la syntaxe propre aux (sous-)routines.

6.3.1 Exemple élémentaire

```
sub maximum
     {
     my ($a, $b) =@_;
     my $max;
     if ($a < $b)
          {
          $max=$b;
          }
     else
          {
          $max=$a;
          }
     return $max;
     }
```

Explication rapide et remarque

`@_` est une variable prédéfinie qui reçoit tous les paramètres de la fonction définie. La ligne contenant `return` n'est pas indispensable si ce qu'on attend est la dernière variable utilisée.

Par convention, on glisse toutes les routines à la fin du programme. Mais ce n'est pas obligatoire. En général, je glisse la définition des routines dans un autre document, que j'appelle avec la fonction `require`.

On appelle ainsi la routine constituée :

```
my $chose=2;
my $c=maximum ($chose,8);
```

```
print $c,"\n";
#affiche 8
```

6.3.2 Exemple utilisant une liste

```
my $a=12;
@liste_de_nombres=(-2,13,6,8);
my $m=maximum2 ($a,@liste_de_nombres),"\n";
print $m,"\n";
#affichera 13

sub maximum2
     {
     my ($a, @liste) =@_;
     foreach my $c (@liste)
          {
          if ($a < $c)
               {
               $max=$c;
               $a=$max;
               }
          else
               {
               $max=$a;
               }
          }
     return $max;
     }
```

6.3.3 Routine de tri

Cf. le point 6.2.2 et les documentations. Pour trier dans l'ordre croissant des fréquence le programme de 6.2.2 :

```
sub tricroissant
     {
     $freq{$a} <=> $freq{$b};
     }
```

Bien entendu, il faudra modifier la ligne `foreach` :

```
foreach my $u (sort tricroissant keys %freq)
```

6.4 Toujours de l'audace

Une idée pour intégrer un *hash* dans une routine : supposez que vous avez un *hash* qui s'appelle `%tableau`, et que vous voulez l'appeler par son nom, mais

que cela vous apparaît difficile. Pourquoi ne pas tenter de créer une variable qui contient le mot *tableau* et de la précéder d'un `%` ? Le risque que Perl interprète la chose comme vous l'imaginez est grand.

Exemple :

```
%tableau= (France => Paris, Allemagne => Berlin);
$mot="tableau";
print %$mot, "\n";
#affiche: FranceParisAllemagneBerlin
print $$mot{Allemagne};
#affiche: Berlin
```

Si vous aimez ce genre d'interprétations, vous adorerez les références.

7 Conclusion

Vous avez désormais tous les moyens d'améliorer vos connaissances de Perl en vous plongeant dans les documentations précitées.

Enjoy !

Table des matières