

## Brève introduction au langage PERL.

O. Pilloud - HB9CEM.- hb9cem@uska.ch

Le langage de programmation **PERL** a ses origines dans le système d'exploitation **UNIX**. Ce système d'exploitation a non seulement été conçu par des professionnels de l'informatique, mais aussi pour des professionnels de l'informatique. Ici, pas question de belles fenêtres, d'icônes et d'animations qui ne servent qu'à surcharger les CPUs, l'efficacité prime.<sup>1</sup>

L'efficacité commence par l'utilisation de formats ouverts, où la transmission de l'information utile prime (c'est-à-dire sans petits icônes et aides visuelles de tout cotés). La compatibilité est également un facteur d'efficacité, en effet combien d'heures sont consacrées aux conversions d'un format à un autre, et même d'une version à l'autre d'un même produit (par exemple Word). Ceci renforce et confirme l'utilisation d'un format commun : les fichiers ASCII.

Dans le domaine de l'efficacité, une action importante est d'automatiser les tâches répétitives et ne nécessitant pas nécessairement l'intervention d'un humain, comme par exemple la conversion d'informations de formatage dans un fichiers ASCII, ou la manipulation de colonnes de chiffres dans des fichiers statistiques ou de mesures.

Tout ceci a poussé lesdits professionnels de l'informatique à créer des outils pour aider à ces tâches, et a donné lieu à des utilitaires tels que **SED**, puis **AWK**.

**AWK**, dont ils existe deux autres versions, soit **NAWK** (New **AWK**) et **GAWK** (**GNU AWK**), est un langage de programmation expressément conçu pour le traitement de fichiers ASCII (un exemple de fichier ASCII pourrait être un log de tous vos contacts radio ou une page web en html).

La philosophie de **AWK** et de ses dérivés, est un programme en 3 parties. Premièrement, une partie du programme est exécutée à titre préparatif. Puis la seconde partie du programme est exécutée, effectuant des opérations (normalement séquentiellement) sur toutes les lignes d'un fichier de données, puis la partie finale du programme est exécutée, pour les opérations finales.

Parallèlement, le langage de programmation **C**, utilisé pour l'écriture initiale du système d'exploitation **UNIX** est devenu extrêmement populaire. En fait sa popularité s'est étendue jusqu'à l'évolution vers le langage **C++** largement utilisé de nos jours, mais ceci est une autre histoire.

Le troisième volet de cette introduction, est à trouver dans la fantastique versatilité d'un outil de programmation peu connu en dehors du monde **UNIX** : les *expressions régulières*, soit *regular expressions* en anglais ou encore *RegExs* en abrégé. Il s'agit d'un jeu de *wild cards*, tels les caractères '\*' ou '?' du monde **DOS**, mais permettant une flexibilité sans commune mesure avec ces derniers. En fait il existe peu d'experts en expressions régulières, non parce qu'elles sont difficiles à maîtriser, mais parce que leur versatilité est telle qu'il est impossible de tout connaître sur ce sujet. Nous en reparlerons plus bas au travers de quelques exemples simples.

Venons-en finalement à notre sujet, soit le langage **PERL**, qui représente la synthèse de ces trois éléments. L'oeuvre de Larry Wall, le langage **PERL** a ses experts, purs génies, tels Randal Schwartz, Tom Christiansen et les nombreux gourous qui fréquentent le newsgroup : *comp.lang.perl.misc*.

---

1. C'est malheureusement de moins en moins vrai, les systèmes UNIX essayant maintenant de rivaliser en inefficacité avec Windoze.

Les lettres P.E.R.L. sont censées signifier *Practical Extraction and Report Language*, (Langage pratique d'extraction - de données - et de génération de rapports) mais Larry Wall lui-même n'hésite pas à se parodier en proclamant que ce sigle signifie *Pathologically Eclectic Rubbish Lister*, ce qui en bon français (vaudois) pourrait se traduire par : programme pathologiquement éclectique de création de fichiers de chenit (chenit étant un mot vaudois dont la traduction difficile pourrait être 'amas d'objets de peu de valeur').

On remarque que même en plaisantant, Larry arrive à placer le mot 'éclectique' qui, comme nous allons le voir résume bien les vastes capacités de son langage de programmation qui peut quasiment tout faire d'un fichier ASCII.

Deux citations qu'il m'a été donné d'entendre, de la bouche de mes collègues, nous serviront de transition vers notre description de **PERL** :

- **PERL** est votre ami.
- **PERL** fait simplement ce qui est naturel, les choses compliquées sont plus difficiles.

Il est vrai que pour le programmeur, même moyen, **PERL**, ayant été conçu par un programmeur, fait d'abord ce qui est naturel. C'est d'ailleurs normal si l'on considère que **PERL** est l'aboutissement d'une évolution des utilitaires et *scripting languages* de **UNIX**, ayant évolués par l'action de programmeurs soucieux d'efficacité, et non de professeurs concernés par la qualité de leur enseignement (**Modula2** par exemple) ou de commerciaux déchaînés désirant remplacer la fonctionnalité par l'esthétique (pas besoin d'exemple).

Attention, ce petit article n'est pas un tutorial pour **PERL**. Il ne s'adresse qu'à des lecteurs ayant déjà quelques notions de programmation, et son but n'est que de présenter quelques exemples pratiques et fonctionnels pour donner envie au lecteur de s'intéresser de plus près à un langage qui est en train de devenir le langage prépondérant dans le monde **UNIX**, mais qui est aussi disponible dans les autres systèmes d'exploitation.

Avant de voir des exemples de programmes, voici quelques liens qui permettront de charger des versions de **PERL** pour différentes plateformes (différentes sortes d'ordinateurs).

- **UNIX** et **LINUX** et **Win32** : <http://www.perl.com/pub/language/info/software.html>
- **MAC** : <http://www.macperl.com/>

Bien entendu, l'environnement de base pour **PERL** est **UNIX**, et dans ce contexte, **UNIX** ou **LINUX** sont idéaux. N'ayant aucune expérience avec les Macs de chez Apple, je ne peux offrir aucun conseil. Quant aux machines animées par les différents win-machins à Billou, je ne suis qu'une victime, peut-être consentante, mais j'ai une ligne de défense valable puisque **PERL** y est utilisable.

Cet article n'est donc pas vraiment un tutorial, mais plutôt destiné, tel un apéritif à vous mettre l'eau à la bouche. Je pense qu'il est donc approprié à ce point de vous connecter, de *download* l'une ou l'autre de ces versions de **PERL** afin d'essayer les quelques exemples ci-dessous.

Il nous faudra aussi un fichier de données sur lequel appliquer nos programmes, et je propose à cette fin de *download* (télécharger) le fichier de démonstration data.zip (1k) disponible à [http://www.pilloud.net/op\\_web/tek.html#03](http://www.pilloud.net/op_web/tek.html#03). Les lecteurs travaillant sous **UNIX** voudront bien utiliser le fichier [http://www.pilloud.net/op\\_web/data\\_ux.tar](http://www.pilloud.net/op_web/data_ux.tar) (10k).

**Note** : le fichier de données ainsi que tous les exemples de ce texte sont disponibles *zippés* dans le fichier : demoperl.zip (3k), également accessible sur le lien ci-dessus.

Le petit fichier de données, comporte 3 sections qui nous permettront de mettre en évidence différentes possibilités de **PERL**.

La première chose à faire est de trouver comment exécuter un programme **PERL** dans l'environnement que vous utilisez. S'il s'agit d'**UNIX** pas de problème : vos programmes commenceront par : `#!/usr/bin/perl` ce qui sera reconnu par votre OS.

S'il s'agit de **MAC** ou de **WINDOZE**, il vous faudra expérimenter ; enfin dans une fenêtre **DOS** (Command prompt), il suffit de donner à votre programme un nom qui se termine par l'extension **.pl** et que la ligne ci-dessus y figure en première place, comme sous UNIX.

Notons, pour les intéressés que **PERL** est un langage à la fois interprété et compilé. Il n'y a pas d'étape intermédiaire de *linking*, mais la syntaxe est entièrement vérifiée avant exécution. De plus, les messages d'erreurs, lors de cette phase de compilation ne sont pas entièrement dénués de sens, comme c'est le cas de la plupart des autres langages, mais essaient vraiment de vous donner des informations utiles quant à la source de l'erreur (sans toutefois toujours y arriver !).

A titre d'essai, le petit programme suivant devrait vous permettre de vérifier que la mise en oeuvre de **PERL** sur votre machine est adéquate. Utilisez un éditeur de texte ASCII pour le créer, sauvez-le sous un nom de votre choix, par exemple `hello.pl` (`.pl` comme **PERL**), puis essayez de l'exécuter en tapant :

```
essai.pl
```

Vous devriez obtenir le message suivant sur votre écran :

```
Hello, world
```

Voici le programme :

```
#!/usr/bin/perl
#
print ("Hello, world\n");
```

A ce point les experts en **C** ne pourront que remarquer la totale similitude avec leur langage favori ; pour les affichages, **PERL** fait grand usage des conventions de **C**, en particulier dans l'usage de commandes telles que `print` et `printf`.

Une fois ce petit programme fonctionnel, nous pouvons passer à la suite de cette démonstration.

Pour la suite nous allons assumer que vous avez le fichier de données mentionné ci-dessus et que vous l'avez appelé `data.txt`. Ce fichier ne sera jamais modifié par nos petits programmes de démonstration.

Il n'est peut-être pas inutile à ce stade de voir de quoi est constitué ce fichier de démonstration. Il comporte 3 sections distinctes :

- Premièrement, quelques lignes de texte, dans le but de voir comment on peut traiter des lignes de texte.
- Deuxièmement, un bout de log, tel qu'un radio-amateur aurait pu le créer, ou tel qu'un programme de log un tant soit peu performant pourrait le fournir.
- Troisièmement, quelques colonnes de chiffres (mesures sur un transistor), car l'une des applications de **PERL** est de traiter de telles données.

Dans tous les exemples suivants, il serait possible de créer directement le fichier de sortie, cependant, comme nous traitons des fichiers ASCII, c'est-à-dire affichables, il est utile de les afficher à l'écran pour vérification visuelle. Ceci étant fait, il est ensuite aisé de le rediriger, dans la tradition **UNIX**, vers un fichier de sortie.

Par exemple, pour le petit programme de test ci-dessus, affichant `Hello, world`, on peut exécuter :

```
perl hello.p > myout.txt
```

Le fichier `myout.txt` devrait être créé automatiquement et contenir le texte : `Hello, World`.

Ceci étant entendu, commençons notre parcours dans les rudiments de **PERL**, en ayant un sentiment quand même un peu ému pour Larry Wall (une bibliographie de **PERL** est proposée en fin de cet article).

**PERL** comporte une grande quantité d'opérateurs utiles, parmi ceux-ci, le *diamant* siège tout en haut. Appelé le *diamant* de par son apparence (`<>`), cet opérateur signifie "pour chaque ligne du fichier d'entrée". En voici un exemple, basé sur notre fichier de données de démonstration. Ce programme crée un fichier de sortie qui ne comporte que les lignes qui contiennent les chiffres 599.

```
#!/usr/bin/perl
#
while (<>) {
    if (/599/) { print };
}
```

En assumant toujours que le fichier d'entrée est appelé `data.txt` et que ce programme est appelé `demo1.pl`, on l'exécute en tapant :

```
demo1.pl data.txt
```

si l'on désire créer un fichier de sortie, on peut exécuter :

```
demo1.pl data.txt > out.txt
```

et le fichier `out.txt` contient les 4 lignes du fichier `data.txt` qui comportent les chiffres 599.

**Discussion** : ce programme comporte plusieurs choses intéressantes, premièrement la ligne : `#!/usr/bin/perl` qui en **UNIX** invoque **PERL**. Dans les autres environnements, elle est simplement ignorée puisqu'elle commence par `#` soit le caractère qui définit un commentaire.

Ensuite : `while (<>) { ... }` est une boucle 'tant que' utilisant le *diamant* pour lire successivement toutes les lignes du fichier d'entrée. Le contenu de la parenthèse est la clause du *while*. Brièvement, on peut dire que (`<>`) assigne la ligne lue à la variable par défaut, qui est appelée `$_` et cette variable est analysée pour voir si elle contient 599.

Si c'est le cas, on imprime la variable par défaut (`$_`) qui contient la ligne en question (`print`).

Puisque l'on parle de simplicité, on peut encore dire à ce point que la variable `$`` (dollar-apostrophe inverse) contient tout ce qui se trouve devant le 599 trouvé, et la variable `$'` (dollar-apostrophe) contient tout ce qui se trouve après. Quand au 'mot' 599, il se trouve dans la variable `$&` (dollar-ampersand). Notons en passant que le fichier spécifié sur la ligne de commande a été automatiquement ouvert en lecture. C'est entre autres dans ce genre de choses que réside la puissance de **PERL**.

Passons maintenant à un programme qui va non seulement ne s'occuper que de la première section de notre fichier de données, mais va donner en sortie les lignes contenant le mot 'de' et va également compter le nombre de lignes contenant ce mot.

Voici ce programme que l'on peut appeler de `.pl` :

```
#!/usr/bin/perl
#
while (<>) {
    if (/^\w/) {
        if (/sde\s/) {print; $nb++;}
    }
}
print " ==> Il y a $nb lignes qui contiennent : de\n";
```

**Note** : dans tous ces programmes, le nombre d'espaces et la tabulation n'est pas importante et ne sert que pour aider à la lisibilité du programme.

Le programme ci-dessus est invoqué par :

```
de.pl data.txt
```

et le résultat, qui peut être redirigé vers un fichier texte au moyen du signe de redirection '>' comporte toutes les lignes de la première section qui comportent le mot 'de', ainsi que le nombre de ces lignes.

**Discussion** : nous avons déjà parlé de la boucle `while` avec la *diamant*. Le premier `if` ne garde que les lignes qui commencent par un mot (`^\w`), c'est-à-dire les lignes de la première section, puisque les autres commencent toutes par des espaces (ou un tab).

Puis on cherche la séquence `<espace>de<espace>` afin de ne pas compter les lettres 'de' à l'intérieur d'un mot. La recherche s'effectue par défaut sur la dernière ligne lue (contenue dans la variable par défaut `$_`).

La variable `$n` (qui n'a pas besoin d'être déclarée ou initialisée - **PERL** s'en occupe) compte le nombre de lignes trouvées (`$n++`). Finalement cette valeur est imprimée ; `\n` comme en **C** imprime un retour de ligne.

Nous pouvons maintenant modifier ce petit programme pour en faire quelque chose de plus utile, par exemple un compteur du nombre de mots dans la première section (note : il y en a 134).

Voici ce programme que l'on peut appeler `mots.pl` :

```
#!/usr/bin/perl
#
while (<>) {
    if (/^\w/) {
        @line = split;
        $mots += @line;
    }
}
print "Le nombre de mots est : $mots\n";
```

**Discussion** : comme pour le programme précédent, le premier `if` permet de ne travailler que sur la première section de notre fichier de données.

L'instruction `split` permet de diviser une ligne de données sur un caractère à choix. Cependant, par souci de simplification, l'instruction `split` comporte un certain nombre de valeurs par défaut, qui justement se trouvent être celles dont nous avons besoin ici. La division de la ligne va s'effectuer sur les espaces entre les mots, et la variable `@line`, qui est un tableau (*array*) va contenir autant d'éléments qu'il y avait de mots dans la ligne. Ces mots pourraient être accédés

individuellement, mais ce qui nous intéresse ici est leur nombre. Or le simple fait d'assigner un tableau à une variable 'normale' (scalaire) nous donne le nombre d'éléments dans le tableau. Encore le soucis de simplification et d'efficacité de **PERL**.

La ligne `$mots += @line;` qui est une forme raccourcie (comme en **C**) de `$mots = $mots + @line;` ajoute donc à la variable `$mots` (qui n'a ni besoin d'être déclarée, ni initialisée) le nombre d'éléments dans le tableau `@line`.

Finalement, une fois tout le fichier lu, le résultat est affiché. On se doit ici de remarquer que les variables (ici `$mots`) sont 'visibles' à la commande `print` à l'intérieur de chaînes (*string*) encadrées de " ". Ceci est un comportement hérité de **UNIX**. Si cela n'est pas désiré, on utilise des apostrophes simples ( ' ' ).

Passons maintenant au traitement de la seconde section de notre fichier de données. Ici les données sont en colonnes, un format pour lequel **PERL** est particulièrement bien adapté. On se propose ici de trouver tous les contacts en CW (morse) qui nous ont reçu avec un rapport de moins de 599. Nous désirons aussi afficher la ligne d'entête, avec la dénomination des colonnes. Ensuite nous allons demander de l'utilisateur un indicatif, et afficher le numéro de la ligne contenant cet indicatif.

Nous allons pour ce programme utiliser l'instruction `split`, et tester les deux champs concernés avant d'imprimer la ligne en question. Voici ce petit programme que nous pouvons appeler `indic.pl` :

```
#!/usr/bin/perl
#
while (<>) { s/^\s+//;
    if (/^Date/) { print "    ",$_ }; # afficher l'entête
    @line = split;
    if ($line[3] eq "CW") {
        if ($line[6] ne "599") {
            print (++$nb," ",$_);
            $call{$line[4]} = $nb;
        }
    }
}
print "Quel indicatif chercher ? : ";
$ind = <STDIN>;
chop($ind);
$ind =~ tr/a-z/A-Z/;
$nb = $call{$ind};
if ($nb =~ /\d/) { print "L'indicatif $ind se trouve à la ligne $nb.\n";}
else { print "L'indicatif $ind ne se trouve pas dans le tableau.\n";}
```

**Discussion** : l'instruction `s/^\s+//;` fait usage d'une expression régulière simple (`^\s+`) et signifie : substitue le contenu de l'expression régulière par rien. Quant à l'expression régulière, elle signifie : en début de ligne (^) rechercher '1 ou plus' espaces. Le signe `\s` signifiant 'espace, tab ou tout ce qui sépare des mots', alors que le `+` signifie 'au moins 1 fois le signe précédent', ici `\s`. Cette instruction procède donc à l'élimination des espaces en début de ligne.

Puis si cette ligne commence par le mot 'Date' (le carret '^' signifiant 'en début de ligne', elle est affichée précédée de 3 espaces.

Ensuite toutes les lignes du fichier sont examinées tour à tour, divisées en éléments dans un tableau par la commande `split`. Nous pouvons ensuite examiner les éléments de ce tableau qui nous intéressent, soit le champ 3 qui contient le mode de transmission (on commence à compter depuis zéro) et le champ 5 qui contient le rapport reçu. Si cette ligne remplit nos critères de sélection nous pouvons l'imprimer, précédée d'un numéro d'ordre : `print (++$nb, " ", $_)`.

`$nb` est la variable qui compte le nombre de lignes imprimées. Comme elle vaut zéro par défaut, on commence par l'incrémenter avant de l'afficher (`++$nb`). `$_` est la variable qui contient toute la ligne lue, bien sûr.

Puisque nous aurons ensuite besoin de retrouver le numéro d'ordre correspondant à un indicatif, créons un tableau dit associatif pour toutes ces paires de données (`$call{$line[4]} = $nb`).

Un tableau associatif est l'une des particularités très puissante du langage **PERL**. Il crée une table de correspondance entre des paires de données, évitant souvent des boucles de recherche qui seraient autrement indispensables.

La dernière section du programme comporte encore quelques fonctions intéressantes. Pour commencer : `$ind = <STDIN>`. Cette instruction lit un indicatif entré au clavier. On peut noter que le clavier est considéré comme un fichier, puisqu'il est accédé par une variation de l'opérateur *diamant* mais en spécifiant `STDIN` (*Standard Input*).

Le dernier caractère entré au clavier est un retour chariot (*Enter* ou *Return*), et ce caractère doit être supprimé avant d'effectuer la recherche ; c'est le rôle de l'instruction `chop`. La ligne suivante convertit les éventuelles minuscules en majuscules pour la comparaison.

Puis nous pouvons utiliser l'indicatif entré comme index dans notre tableau associatif pour trouver le numéro de ligne correspondant. Si cette opération retourne un nombre (`\d`), nous avons trouvé et pouvons afficher le résultat, sinon, un message adéquat est imprimé.

Venons-en maintenant à la troisième section de notre fichier de données, qui contient quelques résultats de mesures sur un transistor. Les 3 colonnes représentent la tension de base (*Vb*), le courant de base (*IB*) et le courant correspondant de collecteur (*IC*). Proposons-nous de calculer le gain (*Beta*) de ce transistor et de l'afficher en fonction du courant de collecteur. Ce gain est bien entendu égal au rapport du courant de collecteur et du courant de base (*IC/IB*).

Ceci pourrait servir par exemple à déterminer pour quel courant de collecteur le gain est maximal, ou pour créer un graphe de *Beta* à l'aide d'un programme graphique adéquat (par exemple *xmgr* sous **UNIX**). Cette fois, nous allons créer directement un fichier de sortie appelé `beta.txt`.

Premièrement nous devons ignorer les deux premières sections du fichier de données, puis créer les deux colonnes requises de chiffres. Voici le programme que nous appellerons `beta.pl`:

```
#!/usr/bin/perl
#
open (OUT,">beta.txt");
while (<>) { s/^\s+//;
    if ($third){
        @line = split;
        if (@line < 3) {next}
        print OUT ($line[2],"\t", $line[2]/$line[1],"\n");
    }
}
```

```

    }
    if (/^Vb/) { $third = 1;}
    }
close (OUT);

```

**Discussion** : les instructions `open` puis `close` vers la fin du programme gèrent le fichier de sortie (l'instruction `close` n'est cependant pas nécessaire puisque **PERL** s'en occupe si elle est absente). Puis comme pour l'exemple précédent, nous commençons par supprimer les espaces en début de ligne s'il y en a.

La variable `$third` est un *flag* (drapeau) qui vaut zéro par défaut. La section `if ($third)` du programme n'est donc pas exécutée pour le moment.

L'instruction `if (/^Vb/)` cherche le mot 'Vb' en début de ligne. En effet, ceci est l'entête de la section qui nous intéresse.

Ceci trouvé, nous allons, au prochain passage (à la prochaine ligne) passer par l'instruction `if ($third)`, qui va maintenant effectuer les calculs requis, et en diriger les résultats sur le fichier désigné par `OUT`.

L'instruction `if (@line < 3) {next}` permet de sauter après la fin de la boucle courante (ici `if ($third)`) si le `split` n'a pas fourni 3 champs, soit si l'on *splitte* une ligne vide comme celles en fin de fichier. Cela évite une division par zéro qui est une erreur fatale.

Ici nous imprimerons IC, un tab (`\t`), beta (IC/IB), puis un retour ligne (`\n`).

Bien entendu, l'instruction `if (/^Vb/)` n'est plus exécutée, puisque que ce fichier ne comporte qu'une ligne répondant à ce critère.

Notons que dans la ligne `if (/^Vb/) { $third = 1;}` le '1' pourrait être remplacé par le mot "true" ou même "false" et que le programme fonctionnerait toujours. En effet pour qu'une expression soit fausse, il faut qu'elle soit évaluée à zéro, or ces deux mots ne valent pas spécifiquement zéro. Une autre chose intéressante est de remarquer que **PERL** se comporte intelligemment en présence d'une chaîne ou d'un nombre et qu'une variable peut contenir indistinctement l'un ou l'autre.

Une fois le programme exécuté, un fichier `beta.txt` est créé, qui contient une colonne de valeurs IC et une colonne beta. On peut y voir que pour ce transistor, le beta passe par un maximum pour des valeurs de IC comprises entre 8 et 46 mA.

### Conclusion :

Ces quelques petits exemples font à peine justice à la grande valeur du langage **PERL**. Il n'est pas rare, dans un cadre professionnel, de rencontrer des programmes en **PERL** de plusieurs centaines à quelques milliers de lignes.

Encore un dernier exemple, j'ai récemment dû m'occuper de réorganiser une base de donnée sous **Win95/DOS** contenant plusieurs centaines de fichiers. Le but était de renommer un grand nombre de ces fichiers, en s'assurant qu'il n'y en avait pas à double et que tous les fichiers avaient une structure de noms commune et bien précise.

La commande **DOS** `dir`, redirigée vers un fichier texte a fournit le fichier de données, plusieurs petits programmes **PERL** ont manipulé ce fichier pour implémenter les différentes opérations requises, puis pour générer un fichier **DOS** `.bat` censé exécuter la suite d'opérations de déplacement et de 'renommage' des fichiers à traiter.

Avant d'exécuter ce fichier `.bat`, il fut ainsi possible de le vérifier, de s'assurer qu'il ne comportait pas d'erreur à même de causer la perte de l'un ou l'autre des fichiers d'entrées. Ce programme *batch* finalement exécuté dans une fenêtre **DOS** a fonctionné sans problème.

Si cette suite d'opérations avait dû être effectuée par les moyens mis à disposition par Billou et son OS, il est estimé que cela aurait pris 10 fois plus long, sans compter les risques d'erreurs importants lorsque de nombreuses opérations sont effectuées manuellement ou par des programmes plus complexes et volumineux.

J'espère ainsi vous avoir donné envie de découvrir le langage **PERL** de façon plus approfondie. Ce langage est devenu, ces dernières années l'un des piliers du monde **UNIX**, mais sous-employé, me semble-t-il par les utilisateurs d'autres systèmes d'exploitation.

Il trouve enfin de nombreuses utilisations comme langage de programmation pour l'automatisation de sites web.

### **Bibliographie :**

La plupart des bons livres techniques de programmation et en particulier ceux sur **PERL** sont publiés par **O'REILLY** : <http://perl.oreilly.com/>

Il existe certainement plusieurs bons livres sur **PERL**, mais deux d'entre eux forment la base et les fondations de ce langage :

Introduction en français : **Introduction à PERL**, Randal L. Schwartz, Editions O'Reilly International Thomson, Paris, ISBN 2-84177-005-2.

Le même en anglais : **Learning PERL**, Randal L. Schwartz, Editions O'Reilly & Associates, Inc. ISBN 1-56592-042-2.

L'ouvrage de référence en français : **Programmation en PERL**, Larry Wall, Tom Christiansen & Randal L. Schwartz Editions O'Reilly International Thomson, Paris, ISBN 2-84177-004-4.

Le même en anglais, aussi connu sous le nom de 'Camel Book' (Le livre au chameau - à cause de l'illustration sur sa couverture) **Programming in PERL**, Larry Wall, Tom Christiansen & Randal L. Schwartz, O'Reilly & Associates, Inc. ISBN 1-56592-149-6.

**Note** : le premier de ces livres est idéal pour débiter avec le langage PERL ; quant au second, plus volumineux, c'est plutôt un ouvrage de référence, indispensable aux utilisateurs avancés de PERL.

Finalement si vous désirez en savoir plus sur les expressions régulières, vous trouverez de précieux renseignements dans les livres ci-dessus, dans le plupart des livres consacrés à UNIX, et dans le livre : *Mastering Regular Expressions*, Jeffrey Friedl, O'Reilly & Associates, Inc. ISBN 1-56592-257-3.

**Note** : si vous lisez suffisamment l'anglais pour considérer l'acquisition de ces ouvrages en anglais, je ne peux que vous encourager. En effet, l'informatique en français est comme la cuisine américaine, elle remplit son but, mais toute la magie a disparu !

Revision 2004-09-06, 2007-08-05