Initiation au Perl

 v1.0.1, Septembre 1999.

Ce document fournit des informations de base sur la programmation en Perl. Il est fait pour être lu tout en ayant un clavier près de soi.

Table des matières

1 Informations sur ce document

1.1 Auteur et Copyright

1.2 Améliorations de ce document

Si vous souhaitez améliorer ce document en y ajoutant des paragraphes ou tout simplement des corrections judicieuses, vous pouvez m'envoyer un mail (*tyndiuk@ftls.org*) en m'indiquant les modifications à apporter.

1.3 Remerciements

Merci à toutes les personnes ayant apportées leurs critiques constructives sur ce document qui m'ont permis de l'améliorer...
Et tout particulièrement à ? ? ? ? pour la version premieère version Post Script...

1.4 Nouvelles versions

Toutes nouvelles versions de ce documents ainsi que d'autres sont / seront disponibles prioritèrement sur le site de l'auteur : *http ://www.ftls.org/*.

2 Introduction

Le terme Perl est en fait un acronyme. Il vient de la compression du nom anglophone "Practical Extraction and Report Language", écrit par Larry WALL (*larry@wall.org*). Perl est un langage de programmation orienté objet qui permet l'extraction et le traitement rapide d'informations.

L'avantage du Perl est que le programme est un langage en script et non un code source à compiler ce qui permet de porter votre programme directement sur toutes les plates-formes supportées (Unix, Windows, Mac...) sans autre opération qu'une simple copie de fichier.
Un script Perl est à la base interprété, mais cette période d'interprétation n'est que passagère. En fait, il y avant chaque exécution une compilation transparente pour l'utilisateur. L'exécutable est généré en mémoire puis exécuté à partir de là. Ce qui augmente légèrement le temps d'exécution par rapport à un programme équivalent en C, mais c'est souvent bien plus rapide à écrire, plus simple à maintenir et à porter...

Ce petit tutorial s'adresse avant tout aux débutants, il a pour but de vous introduire aux bases du langage Perl, donc n'abordera pas toutes les spécificités de ce langage...
Il n'y a pas besoin d'avoir des connaissances en programmation pour commencer, mais des notions de programmation (surtout en shell Unix) sont des atouts...

3 Syntaxe de base et premier Programme

La syntaxe du Perl est très souple comme vous pourrez le remarquer dans la suite de ce document...

3.1 Premier programme :

Voici votre premier programme Perl...

```
#!/usr/local/bin/perl
#
# programme tout bete
#
print "Salut le monde.\n";      # Affiche Salut le monde.
```

Dans la suite nous détaillerons chaque partie.

3.1.1 La première ligne

Pour ceux qui ne connaissent pas les scripts Unix, vous devez vous demander ce qu'est cette première ligne. Il faut savoir que sous Unix tous les scripts doivent commencer par une ligne du type #!/répertoire/programme qui indique à la machine quel interpréteur utiliser quand le programme est exécuté. Cette ligne doit donc contenir le chemin de Perl sur votre machine, en général dans /usr/local/bin/perl ou /usr/bin/perl...

3.2 Exécuter un script Perl

Etant donné que le chemin d'accès au compilateur est spécifié au début du programme (#!/usr/bin/perl), il suffit de taper le nom du programme. Attention, si vous êtes sous Unix vous devez avant le rendre exécutable de la façon suivante : *chmod u+x <nom du programme>*

Il est aussi possible d'utiliser le compilateur à même la ligne de commande comme suit :
/usr/bin/perl <nom du programme>

3.2.1 Les arguments de l'interpréteur

Il est possible de passer un certain nombre de paramètre à l'interpréteur Perl. Les arguments peuvent être combinés et placés en argument de la commande 'perl' (perl -s file.pl) ou dans la première ligne du programme (#!/usr/bin/perl -s)

Voici quelques une des plus utilisées :

```
-0digits : specifie le separateur de champs en octal. La valeur 00 permettra le lire le fichier en mode pa
-a : provoque l'auto split quand il est utilise avec les parametres -n et -p.
-c : verifie la syntaxe du programme sans l'executer.
-d : execute le programme PERL en mode debug.
-Dnombre : positionne les flags de debug.
-ecommande : permet de rentrer une commande sans passer par un fichier Perl et de l'executer.
-iextension : le fichier utilise avec la construction <> sera edite comme fichier d'entree/sortie.
-Idirectory : indique ou sont les fichiers d'inclusion.
-loctnum : permet de terminer les lignes par le caractere octnum en octal.
-n : execute une boucle sur tout le fichier pour executer le programme sans afficher le resultat.
-p : execute une boucle sur tout le fichier pour executer le programme en affichant le resultat.
-P : le programme est passe au preprocesseur C avant d'etre execute.
-s : indique que les options passees entre le nom du programme et les arguments du programme ne sont pas a
-S : Perl utilise la variable PATH pour rechercher le script a executer.
-u : provoque un core dump apres la compilation du script.
-U : permet de faire des operations non securisees.
-v : affiche la version du compilateur PERL.
-w : affiche des messages d'attention (warning) sur les identificateurs qui sont utilises une seule fois o
-xrepertoire : le script est compris dans un message (ou un courrier) le programme ne commencera qu'apres
```

3.3 Les Commentaires

Comme dans tous programmes, on peut insérer des commentaires. En Perl il faut le précéder du symbole # : tout ce qui suivra jusqu'à la fin de la ligne sera ignoré. La seule manière d'étendre un commentaire sur plusieurs lignes est de commencer chaque ligne par #.

3.4 Les commandes intégrées

Comme dans tous langages de programmation, Perl offre un ensemble de commandes déjà définies, ces commandes peuvent prendre un ou plusieurs arguments...
commande arg1, arg2 ; ou *commande(arg1, arg2) ;*

Dans notre exemple ont utilise la commande *print*, qui a pour but d'afficher un texte sur la sortie standard...
print "Salut le monde.\n" ;

4 Les Variables et Types de données

Perl a trois structures de données : les scalaires, les tableaux de scalaires, et les tableaux associatifs de scalaires, appelés "hachages".

4.1 Les noms de variables

Comme dans tous les langages, des règles précises doivent être suivies dans la sélection des noms de variables :

- Un nom de variable est toujours précédé par $ (pour désigner un scalaire ou une référence à un élément de tableau), @ (pour désigner un tableaux complet), % (pour désigner un hachages) ou & (pour désigner les appels de sous programmes).

- Un nom de variable peut être composé de lettres minuscules ou majuscules. Par contre, le langage est sensible à la casse, ce qui signifie que $Nom, $nom et $NOM sont toutes des variables différentes ;

- Un identifiant peut contenir un caractère numérique. Par exemple, il est possible d'avoir un nom de variable tel $nom2 ;

- Finalement, le caractère de soulignement (_) est permis. Aussi, on peut définir des variables comme $nombre_de_jours.

L'affectation, se fait en général grâce à l'opérateur '='
$jours = "Lundi" ; # Affecte à la variable $jours la chaîne Lundi.
@jours = ("Lun", "Mar", "Mer") # Affecte au tableau @jours les chaînes "Lun" "Mar"...

4.2 Les scalaires

Perl est un langage contextuellement polymorphe, on n'a pas besoin de définir le type d'une variable, les variables scalaires peuvent contenir des formes variées de données singulières, comme des nombres, des chaînes, et des références. La conversion d'une forme à l'autre est en général transparente.
Les valeurs scalaires sont toujours désignées par un '$', même si l'on se réfère à un scalaire qui fait partie d'un tableau.

Exemple :

```
$a = 1;
$b = $a + 1; implique $b = 2
mais
$a = "1"; # La chaine de caracteres 1
$b = $a + 1; implique $b = 2
Cela peut surprendre au debut, mais s'avere tres pratique...
```

Il n'existe pas non plus de type "booléen", une valeur scalaire est interprétée comme VRAIE (TRUE) si ce n'est pas une chaîne vide ou le nombre 0 (ou sous équivalent sous forme de chaîne, “0”).

4.2.1 Les numériques :

Il existe plusieurs format de type numérique, la conversion est aussi transparente dans le sans entier -> flottant court -> flottant long... dans le sens inverse, il faut les caster, mais nous verront cela dans la suite du ce tutorial.

Exemples de numériques :

```
2345
12345.67
.23E-10
0xffff              # hexa
0377                # octal
```

4.2.2 Les chaîne de caractères

Nous avons vu précédemment qu'il était possible d'assigner une chaîne de caractères à une variable. Mais il existe plusieurs façons d'affecter ce type de valeur.
Avec des apostrophes :
Lorsqu'un texte est donné entre apostrophes (' ') à une variable, il est affecté tel quel, les variables contenues ne sont donc pas interprétées.

Avec des guillemets :
Les guillemets fonctionnent de la même façon que les apostrophes, sauf qu'ils permettent la résolution de variables internes.

Avec des accents graves :
Les accents graves (‘ ‘) permettent l'exécution de la commande qu'ils délimitent et remplacent l'espace occupé par la valeur de retour ainsi obtenue.

Avec citation orientée ligne << :
Les citations orientées (<<) permettent de définir une chaîne de caractères sur plusieurs lignes et pouvant contenir des simples ou des doubles cotes (' ou "). Après un << vous spécifiez une chaîne pour terminer le matériel cité, et toutes les lignes qui suivent la ligne courante jusqu'à la chaîne de terminaison forment la valeur de l'élément. Exemple :

```
#!/usr/bin/perl

$prenom = 'Patric';
print $prenom;
# Affiche Patric

print '$prenom\n';
# Affiche $prenom\n

print "$prenom\n";
# Affiche Patric\n

$repertoire_courant = `pwd`;
# Execute la commande pwd (Commande Unix connand le repertoire courant)
print $repertoire_courant;
```

```
# Affiche le chemin du repertoire courant.

$prenom = <<EOF;
Maligne 1
Maligne 2
EOF
print $prenom;
# Affiche Maligne 1# Maligne 2
```

Les caractères spéciaux En Perl comme dans d'autre langage, il existe une série de caractères particuliers qui sont interprétées comme le montre le tableau suivant :

```
\t      tabulation
\n      nouvelle ligne
\r      retour ligne
\f      form feed
\v      tabulation verticale
\b      retour arriere
\a      beep
\e      escape
\0dd caractere octal
\xdd caractere hexadecimal
\cx caractere de controle
\l      force le caractere suivant en minuscule
\u      force le caractere suivant en majuscule
\L      force la suite en minuscules jusqu'au caractere \E
\U      force la suite en majuscule jusqu'aux caracteres \E
\Q      force la suite jusqu'a \E a etre interpretee en expression reguliere
\E      marque de fin de minuscule/majuscule/quote
\\      empeche l'analyse de \
```

4.3 Les Listes

Une liste est une série de variables scalaires. L'avantage de la liste est que l'espace mémoire requis pour emmagasiner l'information est allouée dynamiquement et libérée dès la fin de l'utilisation. Aussi, il n'y a pas de perte de ressources inutile par la réservation de mémoires qui ne seront jamais réutilisées.

Exemple :
print ("toto1\n", "toto2\n") ;

4.4 Les tableaux

Un tableau est en fait une liste réutilisable, c'est-à-dire qui reste accessible en mémoire pendant toute l'exécution du programme. Les tableaux autorisent un accès indexé par des entiers {0,1,2,3,...}. Le premier élément de @jours est $jours[0], le second est $jours[1] et ainsi de suite.
Deux symboles importants sont à considérer en ce qui concerne les tableaux :

@ (le symbole "at") :

Ce symbole réfère au tableau ou une tranches de tableau.

$ (le signe dollars) :

Utilisé avec les crochets ([]), il permet d'accéder à un élément particulier du tableau (le premier étant toujours 0).

Exemple :

```
@jours = ("Lun", "Mar", "Mer", "Jeu", "Ven", "Sam", "Dim");
# Defini le taleau jour.

@WeekEnd = @jours[5,6];
# Affect a tableau @weekEnd ("Sam", "Dim")

print $jours[0]."\n";
# Affiche Lun.
print $WeekEnd[0]."\n";
# Affiche "Sam"
```

Note :
La variable $#nom_tableau ; donne l'indice du dernier élément du tableau (@nom_tableau). Le premier indice du tableau est 0. Le nombre d'éléments d'un tableau est donc $#nom_tableau+1 ;

4.5 Les tableaux associatifs

Les tableaux associatifs, aussi appelés table de "hachages" sont accessibles via une clé autre qu'un chiffre représentant leur index. Dans cette optique, chacune des données d'un tableau associatif est représentée par une combinaison clé/valeur.
Deux symboles importants sont à considérer en ce qui concerne les tableaux :

% (le symbole de pour cent) :
Ce symbole réfère au tableau entier. Par exemple, %jours spécifie tout le tableau jours.
{} (les braquettes) :
Serve à indiquer une clé. Par exemple, $jours{"Lu"} définit une clé nommée Lu.

Exemple :

```
%jours = ("Lu", "Lundi",        # Definition du tableau associatif %jours
        "Ma", "Mardi",
        "Me", "Mercredi",
        "Je", "Jeudi",
        "Ve", "Vendredi",
        "Sa", "Samedi",
        "De", "Dimanche");

print "$jours{'Ma'}\n";
# Affiche Mardi
```

Il est a note que l'affectation d'un associatif associatif peut ce faire de la façon suivant :

```
%jours = ("Lu" => "Lundi",      # Definition du tableau associatif %jours
        "Ma" => "Mardi",
        "Me" => "Mercredi",
        "Je" => "Jeudi",
        "Ve" => "Vendredi",
        "Sa" => "Samedi",
        "De" => "Dimanche");
```

4.6 Les variables spéciales

En Perl il existe un certain nombre de variables spéciales. La plupart des noms ont un mnémonique acceptable, ou équivalent dans un des shells. Néanmoins si vous souhaitez utiliser des descripteurs longs vous avez juste à utiliser *use English ;*.

```
<htmlurl url="http://www.perl-gratuit.com/traduction/traduits/perlvar.html" name="Voir la liste">
```

5 Les Opérateurs

5.1 Opérateurs d'affectation

Comme on l'a vu dans les exemples précédents, l'opérateur d'affectation est le '='.

```
$a = <arg>      # Assigne quelque chose a $a
$a = $b;        # Assigne $b a $a
$a += $b;       # Ajoute $b a $a
$a -= $b;       # Soustrait $b a $a
$a .= $b;       # Concatene $b a $a
```

5.2 Opérateurs arithmétiques

Le Perl utilise les même notations que le langage C :

```
$a = 1 + 2;     # Ajoute 1 a 2, et l'affecte a $a
$a = 3 - 4;     # Soustrait 4 a 3, et l'affecte a $a
$a = 5 * 6;     # Multiplie 5 et 6, et l'affecte a $a
$a = 7 / 8;     # Divise 7 par 8, et affecte 0,875 a $a
$a = 9 **5;     # Eleve 9 a la cinquieme puissance
$a = 5 % 2;     # Le reste de 5 divise par deux (division euclidienne)
++$a;   # Incrementation de $a, puis on retourne la valeur $a
$a++;   # On retourne la valeur $a puis incrementation de $a
--$a;   # Decrementation de $a, puis on retourne la valeur $a
$a--;   # On retourne la valeur $a puis decrementation de $a
```

5.3 Opérateurs sur les chaînes de caractères

Voir chapitre *La gestion / modifications des chaînes : Opérateurs sur les chaînes*

5.4 Opérateurs logiques et de comparaison

5.4.1 Opérateurs logiques

```
&& ET logique   Ex: ($a && $b)
|| OU logique   Ex: ($a || $b)
!  NON  Ex: ! ($a)
```

Les fonctions logiques && et || n'évaluent pas la seconde condition si la première suffit à déterminer la solution.

5.4.2 Opérateurs de comparaison

nombres chaînes

égalité == eq

inégalité != ne

plus grand que > gt

plus grand ou égal >= ge

plus petit que < lt

plus petit ou égal <= le

comparaison avec
résultat signé <=> cmp

Le résultat des opérations <=> et cmp est :
-1 si l'opérande de gauche est inférieure à l'opérande de droite,
0 si elles sont égales,
+1 si l'opérande de gauche est supérieure à l'opérande de droite.

5.5 Opérateurs sur les tableaux

```
scalar(@<nom du tableau>)
        # Retourne le nombre d'elements contenus par le tableau

shift(@<nom du tableau>)
        # Retourne et retire le premier element du tableau

pop(@<nom du tableau>)
        # Retourne et retire le dernier element du tableau

unshift(@<nom du tableau>,<liste>)
        # Ajoute une liste au debut du tableau

push(@<nom du tableau>,<liste>)
        # Ajoute une liste a la fin du tableau

join("Model",@<nom du tableau>)
        # Retourne la chaine composees des elements du tableau separee par le Model

sort(@<nom du tableau>)
        # Retourne le tableau trier dans l'ordre alphabetique
        # Il est aussi possible de definir des fonctions de comparaison adaptees au trie souhaite.

reverse(@<nom du tableau>)
        # Retourne l'inverse de l'ordre du tableau
```

Pour les tableau associatifs, il existe deux opérateurs :

```
values(%<nom du tableau>)
        # Retourne une liste des valeurs associees au tableau

keys(%<nom du tableau>)
```

```
        # Retourne une liste des cles associees au tableau
```

6 Les boucles et conditions

6.1 Condition

6.1.1 Condition simple

```
if ( <condition> ) { <action(s)> }

if ( <condition> ) { <action(s)> }
else { <action(s)> }
```

6.1.2 Conditions imbriquées

```
if ( <condition> ) { <action(s)> }
elsif ( <condition> ) { <action(s)> }
elsif ( <condition> ) { <action(s)> }
....
else { <action(s)> }
```

Exemple :

```
if ($jours eq "Lundi") {
        print "Debut de semaine\n";
} elsif ( ($jours eq "Samedi") || ($jours eq "Dimanche") ) {
        print "C'est le week-end\n";
} else {
        print "Courant de semaine\n";
}
```

6.2 Boucle for

Le Perl dispose de la structure for (pour), qui est une copie de celle du C.

for (<initialisation> ; <test> ; <incrémentation>) { <action(s)> }

Exemple :

```
for ($i = 0; $i < 10; $i++) {
        # commence a $i = 0
        # Continue tant que $i < 10
        # Incremente $i avant de repeter.
        print $i."\n";
}
```

6.3 Boucle foreach

La structure foreach (pour chaque) permet de parcourir tous les éléments d'un tableau ou d'une structure listée quelconque (comme un fichier). Elle se présente de la façon suivant :

foreach <élément> (<liste ou tableau>) { <action(s)> }

Exemple :

```
foreach $jour (@jours) {
        print $jour."\n";  # affiche chaque element contenu dans @jours.
}
```

6.4 Boucle while

Le Perl dispose de la structure while (tant que), qui est une copie de celle du C.

while (<condition>) { <action(s)> }

Exemple :

```
print "Mot de passe? "; # Demande un mot de passe
$in = chop(<STDIN>);    # Lit le mot entre + Enleve le retour chariot de la fin de la ligne
while ($in ne "Fred")   # Tant que le mot n'est pas bon
{
        print "Erreur. Essaye encore une fois : ";
        $a = chop(<STDIN>);     # Relit le mot entre + Enleve le retour chariot
}
```

6.5 Boucle do

La boucle do peut être utilisée de 2 façons différentes suivant que l'on désire qu'elle s'exécute *jusqu'à ou tant que* une condition soit valide.

do { <action(s)> } while_ou_until (<condition>) while_ou_until est soit le mot-clé while, soit until. Si vous utilisez *while* les actions sont exécutées *tant que* la condition est vrai. Si vous utilisez *until* les actions sont exécutées *jusqu'à ce que* la condition soit vrai.

Exemple :

```
do {
        $line = <STDIN>;
} while ($line ne "");
# Lit l'entree standard tant que $line n'est pas vide.

do {
        $line = <STDIN>;
} until ($line eq "");
# Lit l'entree standard jusqu'a ce que $line soit vide.
```

6.6 Contrôle du flux dans les boucles

L'exécution normale d'une boucle peut être modifiée par une intervention externe. Perl offre des possibilités intéressantes à ce niveau. Voici les principales commandes disponibles :

last

Envoie l'exécution à la première instruction après la boucle.
En d'autres termes, on force ce passage comme dernier passage.

next

Force un nouveau passage immédiat de la boucle.
Aussi, même si le passage courant n'est pas complètement terminé, l'exécution est renvoyée au début de la boucle.

redo

Force à recommencer une itération de la boucle.

goto LABEL

Force l'envoie à une étiquette spécifiée (LABEL). Cette instruction existe, mais il est déconseillé de l'utilisée si l'on veut programmer proprement...

De plus lorsque plusieurs boucles sont imbriquées, il est possible de définir des étiquettes pour indiquer à quel boucle appartient last, next ou redo...

Exemple :

```
LABEL:
# Definition de l'etiquettes
while ($line1 = <FILE1>) {
        while ($line2 = <FILE2>) {
                if (line2 eq "fin") {
                        last LABEL;
                }
        }
}
```

7 La gestion / modifications des chaînes de caractères

7.1 Opérateurs sur les chaînes de caractères

Les opérations sur les chaînes de caractères, sont simplifiée en Perl :

```
$a = $b . $c;   # Concatenation de $b et $c
$a = $b x $c;   # $b repete $c fois
$a = substr($b, 5, 10); # Garde que les caracteres entre la 5{deg} et la 10{deg} position
$a = lc($b);    # Transforme en minuscule $b
$a = uc($b);    # Transforme en majuscule $b
$a = lcfirst($b);       # Transforme en minuscule le premier caractere de $b
$a = ucfirst($b);       # Transforme en majuscule le premier caractere de $b
$a = length($b);        # Renvoie le nombre de caracteres de $b
```

7.1.1 fonction join

La fonction join prend les éléments d'une liste et les convertit en une chaîne de caractère. Les éléments sont ajouter les uns à la suite des autres en utilisant le séparateur 'sep'.
join(<sep>, <@tab>) ;

Exemple :

```
@tab = ('Mot1', 'Mot2', 'Mot3', 'Mot4');
$resultat = join(" , ", @tab);
print $resultat;
        # Affichera Mot1 , Mot2 , Mot3 , Mot4
```

7.2 Expressions régulières

Une des possibilités les plus utiles en Perl (si ce n'est LA plus utile) est la manipulation des chaînes de caractères. Au coeur du systèmes, les Expressions Régulières.

Une expression régulière est contenue par des slash "/", est sont utilisées pour vérifier la présence ou l'absence d'un motif dans une chaîne de caractère.

7.3 Appartenance

Grâce à l'utilisation de l'opérateur =~ ou !~ ont peut tester si un motif appartient ou pas à une chaîne de caractère.

Exemple :

```
$phrase =~ /le/ # renverra 1 ou nul suivant que $phrase contient ou pas le motif "le"
$phrase !~ /le/ # renverra nul ou 1 suivant que $phrase contient ou pas le motif "le"

$phrare = "Mon OS prefere est linux.";
if ($phrase =~ /linux/) {
        print "Bravo\n";
}
# Condition vrai.
```

7.4 Notions avancées

L'exemple précédant montre une limite par exemple si "linux" est écrit "Linux" le teste sera faut. C'est pour cela qu'il existe des modificateurs et des caractères spéciaux permettant d'accroître l'efficacité des expressions régulières...

Les modificateurs permettent de modifier l'interprétation des expressions régulières. Ce modificateur est placé après le / de fermeture exemple = */motif/modificateur*.

```
i       Reconnaissance de motif independamment de la casse (majuscules/minuscules).
m       Permet de traiter les chaines multi-lignes. Les caracteres ``^'' et ``$'' reconnaissent alors n'im
s       Permet de traiter une chaine comme une seule ligne. Le caractere ``.'' reconnait alors n'importe q
x       Augmente la lisibilite de vos motifs en autorisant les espaces et les commentaires.
```

Exemple :

```
if ($phrase =~ /linux/i) {
        # Sera vrai pour linux, LINUX, Linux, LinuX, lINUx...
        print "Bravo\n";
}
```

Les caractères spéciaux comme leurs noms l'indiquent sont des caractères intégrés au motif. Voici certain de ces caractères spéciaux, et leurs significations.

```
.       # Reconnait n'importe quel caractere, excepte les retours chariot.
^       # Reconnait le debut d'un ligne ou d'une.
$       # Reconnait la fin d'une ligne ou d'une chaine.
*       # Reconnait 0 fois ou plus le dernier caractere.
+       # Reconnait 1 fois ou plus le dernier caractere.
?       # Reconnait 0 fois ou 1 fois le dernier caractere.
{n}     # Reconnait n fois exactement le dernier caractere.
```

```
{n,}    # Reconnait au moins n fois le dernier caractere.
{n,m}   # Reconnait au moins n fois mais pas plus de m fois le dernier caractere.
\       # Annule le sens caractere special qui suit.
|       # Alternative
()      # Groupement ou modele
[]      # Classe de caracteres
```

Les classes de caractères [] (crochet) sont utilisées pour assortir n'importe lequel des caractères qu'il contiennent. A l'intérieur des crochets, "-" signifie : entre et "^" signifie : pas.

```
[ftls]  # Soit f ou t ou l ou s
[^ftls] # Soit pas f ou pas t ou pas l ou pas s
[a-z]   # Toutes les lettres de  a a z inclus
[^a-z]  # Pas de lettre minuscule
[a-zA-Z]        # Pas de lettres
[a-z]+  # N'importe quelle enchainement de lettres
```

Pour facilité l'utilisation des classe de caractères il existe un certain nombre de groupe près définit.

```
\w      [a-zA-Z0-9_]    Reconnait un caractere de "mot" (alphanumerique plus "_").
\W      [^a-zA-Z0-9_]   Reconnait un caractere de non-"mot".
\d      [0-9]   Reconnait un chiffre.
\D      [^0-9]  Reconnait un caractere autre qu'un chiffre.
\s                      Reconnait un caractere d'espacement (espace, tab, retour chariot, etc...)
\S                      Reconnait un caractere autre qu'un espacement.
```

La barre verticale représente "ou", et les parenthèses sont utilisées pour grouper :

```
voiture|moto    # Soit une voiture soit une moto
(ht|f)tp        # Soit http soit ftp
(to)+           # Un ou plusieurs to (to, toto, tototo, totototo....)
```

Il existe d'autre caractères spéciaux :

```
\b      Reconnait la limite d'un mot
\B      Reconnait autre chose qu'une limite de mot
\A      Reconnait uniquement le debut de la chaine
\Z      Reconnait uniquement la fin de la chaine (ou juste avant le caractere de nouvelle ligne final)
\z      Reconnait uniquement la fin de la chaine
```

Lorsque l'on utilise les Groupements (), les valeurs de ces groupements sont mémorisées est peuvent être rappelée par les variables $1,...,$9, ou utilisé dans l'expression de la substitution même avec \1,...,\9.

Exemple :

```
if ($time = /Time: (..):(..):(..)/) {
        $heures = $1;
        $minutes = $2;
        $secondes = $3;
}
```

7.5 Substitution

Perl sait aussi effectuer des substitutions basées sur ces expressions. On peut donc le faire avec la fonction s. (s'ajoute devant le premier / de l'expression régulière). Encore une fois, on utilise l'opérateur = .

Exemple :

```
$phrase = "Mon OS prefere est linux.";
$phrase =~ s/linux/Linux/;
print $phrase."\n";
        # affichera "Mon OS prefere est Linux."
```

L'opération de substitution ainsi utilisée ne permet de ne remplacer que le premier occurrence de la chaîne. Pour pouvoir remplacer toutes les occurrence, il suffit d'ajouter le modificateur "g" après de "/". De plus les modificateurs vu précédaments sont toujours valides.

Exemple :

```
$phrase = "Mon OS prefere est LINUX, linux, LinuX...";
$phrase =~ s/linux/Linux/gi;
print $phrase."\n";
        # affichera "Mon OS prefere est Linux, Linux, Linux..."
```

Il est aussi possible d'utiliser la mémorisations des groupements dans les substitutions.

Exemple :

```
$phrase = "Mon OS prefere est Linux";
$phrase =~ s/(linux)/Linux/gi;
print "Mon OS prefere est".$1."\n";
        # affichera "Mon OS prefere est linux..."
```

Après une recherche, les variables $‘ ou $& ou $’ (lecture seule) contiennent respectivement, la partie avant la chaîne trouvée, la chaîne trouvée, et la partie après la chaîne trouvée.

7.6 Traduction

Il existe aussi un fonction traduction (tr) qui permet le remplacement caractère par caractère. Attention, la plupart des caractères spéciaux ne sont pas reconnu par tr.

Exemple :

```
$phrase =~ tr/abc/edf/  # Traduit les a en e, des b en d, les c en f.
$phrase =~ tr/a-z/A-Z/; # Traduit les minuscules en majuscules.
```

7.7 Eclatement

La fonction split permet décomposer une chaîne en une liste d'éléments en la coupant aux motif définit. Sa syntaxe est la suivant @tableau = split(/motif/, $chaine) ;

Exemple1 :

```
$phrase = "12 15 25";
($v1, $v2) = split(/\s+/, $phrase);
        #On a alors $v1 = "12", $v2 = "15"
```

Exemple2 :

```
$phrase = "Mon OS prefere est Linux";
@list = split(/\s+/, $phrase);
```

```
foreach $mot (@list) {
        print $mot."\n";
}
```

Ce programme donnera le résultat suivant :
Mon
OS
préféré
est
Linux

8 Les procédures

Perl comporte déjà des fonction intégrées, mais on a aussi besoin de créer ses fonctions propres, appelées procédures. Elles peuvent être placées n'importe où dans le programme, mais il est plutôt préférable des les placer toutes au début ou à la fin. Une procédure a la forme suivante :

```
sub ma_procedure {
        print "Merci d'avoir executer ma procedure.\n";
}
```

Dans la suite du programme, elle peut être exécutée par l'appel suivant :
&ma_procedure ;

8.1 Variables locales

Lorsque l'on crée un procédure, il est préférable que les variables de la utilisée dans la procédure n'interfère pas avec celle contenue dans le reste du programme. On définit donc des variables local de la manière suivante :

local(<$var1>, <$var2>, ... , <$varN>) ;
ou
my(<$var1>, <$var2>, ... , <$varN>) ;

Ces variables locale ne sont définie qu'à l'intérieur de la procédure, et sont réinitialisée à chaque appels.

8.2 Paramètres

Il est souvent utile de pouvoir passer des paramètres à une procédure, ont utilise donc la syntaxe suivante lors de l'appel :

```
&ma_procedure;          # Appel sans parametre
&ma_procedure($p1);     # Appel la routine avec 1 parametre
&ma_procedure($p1,$p2); # Appel la routine avec 2 parametres
```

Quand on appel la procédure, tous les paramètres passés sont stockés dans une variable (tableau) spéciale @_ . Ainsi on peut les utiliser soit directement à partir de cette variable, ou les affecter à une variable locale.

Exemple 1 :

```
sub ma_procedure {
        print "Les parametres sont : "; print @_; # Affiche tous les parametres
        print "\n\n";
        print "Le 1er parametre est : $_[0]\n";
        print "Le 2em parametre est : $_[1]\n";
}
```

Exemple 2 :

```
sub ma_procedure {
        my($p1, $p2) = @_;
        print "Le 1er parametre est : $p1\n";
        print "Le 2em parametre est : $p2\n";
}
```

8.3 Renvoi des valeurs

Le résultat d'une procédure est toujours la dernière variable évaluée. Mais pour la lisibilité, on préfère souvent utiliser la fonction "return" qui sort de la procédure en renvoyant le résultat.

Par Exemple :

```
($v1, $v2) = &ma_procedure(25, 66);
print "v1: $v1\n;       # 1650
print "v2: $v2\n;       # 91

sub ma_procedure {
        my($p1, $p2) = @_;
        return ($p1*$p2,$p1+$p2) ;
                # renvoie la multiplication et l'addition des 2 arguments.
}
```

9 Les entées - sorties

Perl contient deux commandes essentielles au niveau des fichiers, soit l'ouverture (open) et la fermeture (close). A celles-ci s'ajoutent les opérations de test, de lecture et d'écriture.

Comme nous l'avons vu dans les exemples précédents, par défaut Perl lit et écrit sur les entées - sorties standard. Il est bien sur possible de rediriger

9.1 Manipulation de fichiers

Lorsque l'on veut lire ou écrire dans un fichier, il faut le dire au programme. Pour cela il faut dans un premier temps ouvrir le fichier, lire ou écrire, puis le fermer.

Pour lire un fichier on utilise la commande Open :

```
Ouvre un fichier en lecture uniquement
        syntaxe : open ( descripteur,"<nom_fichier");

Ouvre un fichier en lecture et en ecriture
        syntaxe : open ( descripteur,"+<nom_fichier");
```

```
Ouvre un fichier en ecriture avec ecrasement en cas d'existence prealable du fichier
        syntaxe : open ( descripteur,">nom_fichier");

Ouvre un fichier en ecriture pour ajout en fin de fichier
        syntaxe : open ( descripteur,">>nom_fichier");

Ouvre un fichier en lecture et ecriture avec ecrasement en cas d'existence prealable du fichier
        syntaxe : open ( descripteur,"+>nom_fichier");
```

Le descripteur (ou handle) permet de spécifier sur quel entrée / sortie on agit. Pour lire ou écrire le fichier on utilise les commandes suivantes :

```
Ecrit dans un fichier ouvert
        syntaxe : print descripteur "ce qu'il y a a ajouter";

Lit ligne par ligne dans un fichier
        syntaxe : $ligne = < descripteur >;

Lit et stocke chaque ligne dans le tableau fichier
        syntaxe : @fichier = < descripteur >;
```

Une fois les opérations voulues effectuées ont ferme le fichier :

```
ferme un fichier
        syntaxe : close (descripteur);
```

Par défaut il existe 3 les descripteurs ouvert à l'exécution du programme

```
<STDIN> :       Descripteur d'entree, Entree standard en lecture seul.
<STDOUT>:       Descripteur de sortie. Sortie standard en ecriture seul.
<STDERR>:       Descripteur des messages d'erreurs. Sortie d'erreurs standard en ecriture seul.
```

Lorsque l'on utilise la commande print STDOUT "essais\n" ; est équivalent à print "essais\n" si l'on n'a pas sélectionné une autre sortie par défaut.

La commande *select(descripteur) ;* permet de spécifier une autre sortie par défaut. La commande print "essais\n" ; affectera donc le nouveau descripteur.

La commande Open et Close renvoie en 1 ou 0 en cas de succès ou d'échec, on utilise donc cette propriété pour vérifier qu'il n'y ai pas d'erreur à l'ouverture grâce à la syntaxe suivante :
open(FILE,"$fichier") || die "Erreur de lecture $fichier, Erreur : $!" ;
Affiche le message et sort du programme en cas d'erreur.

Exemple :

```
open(FILE,"<$nom_fichier") || die "Erreur de lecture $nom_fichier, Erreur: $!";
@fichier = <FILE>;
close(FILE);

@result = sort @fichier;

open(FILE,">$nouveau_fichier") || die "Erreur d'ecriture de $nouveau_fichier, Erreur: $!";
print FILE @result;
close(FILE);
```

9.2 Les accès au système de fichiers

Perl offre la possibilité de réaliser un certain nombre d'opérations sur les fichiers et répertoires du disque sans passer par un shell quelconque (utilisation de la commande system ou exec).

```
Efface un ou plusieurs fichiers
        syntaxe : unlink (liste des noms de fichiers);

Renomme ou deplace un fichier
        syntaxe : rename(ancien nom du fichier, nouveau nom du fichier)

Cree un lien symbolique (Unix seulement, ln -s)
        syntaxe : symlink(nom du fichier, nom du lien)

Change de repertoire courant
        syntaxe : chdir(nom du repertoire)

Cree un nouveau repertoire
        syntaxe : mkdir(nom, mode)
                # mode sont les permissions (Unix) a attribuer au repertoire.

Efface un repertoire
        syntaxe : rmdir(nom)
```

9.3 Testes sur les fichiers

Il existe un certain nombre d'opérateur de test sur les fichiers dont voici quelques uns :

```
-e      Le fichier existe
-r      Le fichier est lisible (permission r)
-x      Le fichier est executable (permission x)
-w      Le fichier est accessible en ecriture (permission w)
-d      Le repertoire existe
-T      Le fichier est de type texte
-B      Le fichier est binaire
-f      Fichier texte existant et non vide
-l      Le lien symbolique existe (Unix)
-t      Terminal
-s      Le fichier n'est pas de taille nulle
-z      Le fichier est de taille nulle (! -s)
```

Exemple :

```
if (! -e "$NomFichier") {
        print "Le fichier n'existe pas...\n";
        exit; # Sort du programme.
}
```

9.4 Appels systèmes

Perl permet de faire appel à une commande du système. Pour cela il existe plusieurs solutions :

```
system "nom_de_la commande";

open(descripteur, "nom_de_la commande|"); ....; close(descripteur);
        # Execute la commande et recupere la sortie standard dans le descripteur
```

Exemple :

```
system "rm nomfichier"; sous UNIX
system "del monfichier"; sous DOS
```

9.5 Formatage de la sortie

9.5.1 Utilisation de printf

Il est possible de formater la sortie en utilisant *printf format, valeurs*

Exemple :
printf "%6s %3d %4.2f\n",$s,$d,$f;
%6s sortie caractère sur 6 caractères
%3d sortie d'un entier sur 3 caractères
%4.2f sortie d'un réel sur 4 caractères dont 2 après la virgule

9.5.2 Utilisation de write

Mais il aussi possible prédéfinir des formats. Pour définir un format vous devez utiliser le mot clé format et le définir puis afficher vos résultats au moyen de la commande write.

- Description d'un format :
format nom_du_format =
descriptif de la sortie
.

nom_du_format :
2 types de nom sont possibles, pour l'entête et pour le corps de l'affichage. (NOM_TOP et NOM)
N'oubliez pas le caractère . (point) qui termine la définition d'un format. Ce point doit être en première colonne dans votre fichier.
Ne pas mettre de commentaires comme ici dans la définition d'un format ceux-ci seraient affichés lors de l'affichage.

descriptif de la sortie :
Ce descriptif comprend 2 parties essentielles :

1. le format d'affichage
Cette partie permet de connaître la façon dont vont être affichées les données (l'endroit, la manière, la taille ...) La taille est définie par le nombre de signes présents après le signe @ ou ^

@>>>> indiquera que la valeur sera affichée sur 4 caractères justifiés à droite
@<<<< même chose mais justifié à gauche
@|||| même chose mais centré
@###.## champ numérique avec 2 chiffres après la virgule
@* champ multiligne, on ne s'occupe pas de la taille du champ à afficher
^>>> affichera un champ sur plusieurs lignes de 3 caractères lignes vides incluses
^>>> même chose mais sans les lignes vides
^>>> même chose mais répétera l'affichage jusqu'à trouver une ligne vide

2. les variables à afficher
$variables1,$variables2, ...

- Affichage à l'écran :
Le nom du format sera STDOUT qui par défaut est la sortie standard.
STDOUT_TOP sera le nom du format pour l'entête.

- Affichage dans un fichier :
Le nom du format sera le nom du descripteur du fichier (par exemple DESC).
DESC_TOP sera le nom du format pour l'entête.

- Quelques exemples :

* On désire écrire les noms et prénoms des personnes contenus dans un tableau associatif par colonnes avec comme entête les mots Nom Prénom et ceci à l'écran.

```
format STDOUT_TOP = # Permet d'écrire la première ligne
Nom Prénom # La première ligne
.
format STDOUT = # Permet d'écrire les données formatées
@>>>>>>>> @>>>>>>>>> # Format d'affichage des 2 champs justifié à droite
$nom, %tab{$nom} # le nom sur 8 caractères et le prénom sur 10
.
foreach $nom (keys %tab) {
$prenom = @tab{$nom};
write;
}
```

10 Quelques fonction intégrées utiles

Voici quelques intégrées autre que celle vu dans les chapitres précédant qui peuvent vous être utile.

10.1 chop

Permet de supprimer le caractère de retour chariot à la fin d'un ligne.

syntaxe : chop($ma_chaine);

10.2 rand(?)

Permet de générer un nombre aléatoire comprit entre 0 et $Nb

syntaxe : rand($Nb);

10.3 time et localtime

La commande 'time' renvoie le nombre de seconde écoulées depuis le 1er janvier 1970 jusqu'à maintenant.

syntaxe : $temps = time;

Grâce à la commande 'localtime' ont peut convertir cette valeur pour connaître le jour, le mois, l'année, l'heure... renvoyée par time.
syntaxe : ($sec,$min,$heure,$jour,$mois,$annee,$jour_se,$jour_an,$ete) = localtime(temps);
ou
$sec, $min, $heure : les secondes, minutes, heures
$jour, $mois, $annee : le jour, le mois, l'année courante
$jour_se, $jour_ann : le jour de la semaine (0-6), de l'année (0-364)
$ete : heure été ou heure hivers.

10.4 grep

Permet d'extraire un élément d'une liste qui correspond à un motif spécifié.

syntaxe : @resultat = grep(/motif/, @ma_liste) ;

10.5 crypt

Permet de crypter une chaîne de caractère à l'aide de l'algorithme DES. Cette fonction est généralement utilisée pour crypté des mots de passe. Attention ce codage n'est pas réversible, pour tester si la chaîne cryptée est la même que l'original, il faut crypté l'original est comparé...

syntaxe : $resultat = crypt($original, $maniere) ;

$manière est une chaîne de 2 caractères indiquant à l'algorithme une clé de cryptage. Souvent fournit par la commande suivante : *substr($original, 0, 2) ;*

11 Exemples

11.1 Bonjour

```
#!/usr/bin/perl

if ($#ARGV < 0) {
        print "Quel est votre prenom ? ";
        $prenom = <STDIN>;
        chop ($prenom);
} else {
        $prenom = $ARGV[0];
}
        print "Bonjour $prenom\n";
```

11.2 Calcule la valeur de Pi

```
#!/usr/bin/perl

# Calcule la valeur de Pi avec une precision superieure
# ou egale a 0.1^precision en utilisant la limite de
# la somme pour i variant entre 0 et l'infini de
# (-1^i / 2i+1) * [4 * (1/5)^(2i+1) - (1/239)^(2i+1)]
# qui est egale a Pi/4.
# A chaque iteration pour le calcul de la somme, on
# conserve la puissance negative impaire courante de 5
# et 239. Pour la puissance de -1, on utilise la parite
# de l'iteration.

# Test des arguments et affichage du resultat.

if ($#ARGV < 0) { &usage(); }

for ($i=0; $i < $#ARGV+1; $i++) {
        print &Pi($ARGV[$i])."\n";
}
```

```
# Renvoie la valeur de Pi avec la precision voulue. sub Pi {
        my($precision)=@_;
        my($i, $somme, $somme_precedente, $puissance_de_cinq,
                $puissance_de_deux_cent_trente_neuf, $epsilon);
        $somme = 0;
        $somme_precedente = 1;
        $puissance_de_cinq = 1/5;
        $puissance_de_deux_cent_trente_neuf = 1/239;
        $epsilon = 0.1;

        for ($i = 1; $i < $precision; ++$i) {
                $epsilon /= 10;
        }

        for ($i = 0; abs($somme - $somme_precedente) > $epsilon; $i++) {
                $somme_precedente = $somme;
                $somme += ( ( $i % 2 == 0 ) ? 1 : -1 ) / (2 * $i + 1) *
                        ( 4 * $puissance_de_cinq - $puissance_de_deux_cent_trente_neuf );
                $puissance_de_cinq /= 25;
                $puissance_de_deux_cent_trente_neuf /= 57121;
        }
        return $somme * 4;
}

sub usage {
        print STDERR "Usage $0 ENTIER...\n";
        print STDERR "Calcule la valeur de Pi\n";
        print STDERR "avec une precision superieure ou egale a 0.1^precision\n";
        exit;
}
```

11.3 Triangle de Pascal

```
#!/usr/bin/perl

if ($#ARGV < 0) { &usage(); }

$Nb = $ARGV[0];
$tab[0][0] = 1;

# Calcul du triangle
for ($n = 1; $n < $Nb+1; ++$n) {
        for ($k=1; $k < $Nb+1; ++$k) {
                $tab[$n][$k] = $tab[$n-1][$k] + $tab[$n-1][$k-1];
        }
}

# Affichage du resultat
for ($n = 1; $n < $Nb+1; ++$n) {
        for ($k = 1; $k < $Nb+1; ++$k) {
                if ($tab[$n][$k] != 0) {
                        $s .= $tab[$n][$k]." ";
                }
        }
```

```
        $s .= "\n";
}

print $s;

sub usage {
        print STDERR "Usage $0 ENTIER...\n";
        print STDERR "Affiche le triangle de Pascal de profondeur ENTIER.\n";
        exit;
}
```

11.4 Compteur

```
#!/usr/bin/perl

$count_file = "compteur.txt";
open(COUNT,"<$count_file") || die("Erreur de lecture de $count_file, Erreur: $!\n");
$count = <COUNT>;
close(COUNT);
chop ($count);
$count++;
open(COUNT,">$count_file") || die("Erreur d'ecriture de $count_file, Erreur: $!\n");
print COUNT $count."\n";
close(COUNT);

print $count."\n";
```

11.5 Recherche d'une occurence dans un fichier

```
#!/usr/bin/perl

$fichier = "monfichier.txt";
open(FILE,"<$fichier") || die("Erreur de lecture de $fichier, Erreur: $!\n");

print("Quel mot rechercher : ");
$cherche = <STDIN>;
chop($cherche);
$ligne = 0;
$sum = 0;
$cond = 0;
while(<FILE>){
        $cond=$sum;
        $ligne +=1;
        $sum += (s/\b$cherche\b/$cherche/g);
        if($cond != $sum){
                print("Ligne $ligne ==> ");
                print("$sum\n");
        }
}
close(FILE)
print("-----------------------------------------------------\n");
print("Votre fichier contient $sum fois le mot '$cherche'\n");
```

11.6 Date en français

```
#!/usr/bin/perl

@WeekDays = ('Dimanche', 'Lundi', 'Mardi', 'Mercredi', 'Jeudi', 'Vendredi', 'Samedi');
@Months = ('Janvier', 'Fevrier', 'Mars', 'Avril', 'Mai', 'Juin',
        'Juillet', 'Aout', 'Septembre', 'Octobre', 'Novembre', 'Decembre');

($Sec,$Min,$Hour,$Day,$Month,$Year,$Week_Day) = (localtime);
$Year += 1900;

print $WeekDays[$Week_Day]." ".$Day." ".$Months[$Month]." ".$Year."\n";
        # Affiche Lundi 1 Janvier 1999

$Month++;
if ($Day < 10) { $Day = "0".$Day; }
if ($Month < 10) { $Month= "0".$Month; }
print "$Day/$Month/$Year\n";
        # Affiche 01/01/1999

if ($Sec  < 10) { $Sec = "0".$Sec; }
if ($Min  < 10) { $Min = "0".$Min; }
if ($Hour < 10) { $Hour = "0".$Hour; }

print "$Hour:$Min:$Sec\n";
        # Affiche 23:15:50
```

12 Exercices

Pour apprendre un langage la meilleure méthode est de l'utiliser.

Je vous conseillerez donc en premier temps de reprendre tous les exemples des chapitres précédant et de regarder les résultats, de les modifier pour obtenir un le résultat que vous désirez...

Voici quelques exemples de programmes que vous deviez pouvoir réaliser :

- Manipulation des nombres et conditions
  - Programme d'extraction des racines d'un polynôme du second degré.
- Manipulation de fichier et conditions
  - Programme comptant les lignes non vide d'un fichier texte.
- Expressions régulières et appartenance
  - Programme comptant les lignes d'un fichier contenant X ou x.
  - Programme comptant les lignes d'un fichier contenant un chiffre.
  - Programme comptant les lignes d'un fichier contenant un mot demande en argument.
- Expressions régulières et substitution
  - Programme remplaçant tous les A ou a par Z et z.
  - Programme supprimant tous les chiffres d'un fichier.
- Expressions régulières
  - Programme affichant dans un tableau les informations (Login, Nom, répertoire Home) des utilisateurs de votre système, C'est information sont contenues dans le fichier '/etc/password' et sont sous la forme suivant :
    Login :Password :IDU :IDG :Nom Prénom :Répertoire Home : Shell
    exemple : (ftls :FMn6Faux8CNk :500 :500 :Frederic TYNDUIK :/home/ftls :/bin/bash)
    Faire une version utilisant split, et une utilisant les expressions régulières (avec $1, $2)...

Il n'y a pas les solutions sur le site pour éviter de vous influencer, car il existe plusieurs manière pour arriver au résultat, la seule façon de savoir si votre programme fonctionne correctement est de le tester...

13 Conlusion

Ce document ne se veut qu'un bref aperçu de Perl. S'il vous a donné envie de découvrir ce langage, alors je considérerai avoir atteint mon objectif en l'écrivant.

Si vous avez des commentaires sur ce document, n'hésitez pas à me les faire parvenir à *tyndiuk@ftls.org*...

Il ne vous reste plus qu'à essayer par vous même et à concevoir vos premiers programmes...

13.1 Trouver d'autres informations sur Perl.

Il existe de nombreux endroits où trouver de la documentation à propos de Perl :

13.1.1 Votre machine

En effet, la source la plus complète se trouve sur votre propre ordinateur mais malheureusement en anglais pour l'instant... La commande *perldoc* vous donne une quantité non négligeable d'informations sur les commandes Perl. Au prompt, tapez *perldoc* pour en savoir davantage.

13.1.2 Sur le Web

De nombreux sites proposent des documentations sur Perl en particulier :
- *Perl-Gratuit.com* (En français)
- *http ://www.perl.com/* (Site officielle du Perl en Anglais)
- *http ://www.perl.com/CPAN/* (Documentation officielle du Perl + exemples + modules... mais en Anglais)

13.1.3 Sur le Usenet (News)

- *fr.comp.lang.perl* (En français)
- *alt.comp.lang.perl* (En anglais)

13.1.4 Les livres

Il suffit d'aller dans votre librairie préférée et vous trouverez sûrement un livre sur Perl.

De plus il arrive à certaine revue de presse de proposer des articles sur Perl.