Introduction à Perl

Sébastien Rimour

Module de bioinformatique 2005-2006

Introduction à Perl – Sébastien Rimour

I. Introduction

A. Qu'est ce que Perl ?

- Perl signifie Practical Extraction and Report Language que l'on pourrait essayer de traduire par «langage pratique d'extraction et d'édition ». Il a été crée en 1986 au départ pour gérer un système de News entre deux réseaux.
- Il est disponible gratuitement. Les versions les plus utilisées actuellement sont les versions 5.6 et 5.8. Les principaux apports de la version 5.8 sont une meilleure gestion des caractères internationaux (standard Unicode), un support des architectures 64 bits, ainsi qu'une nouvelle implémentation de la programmation par thread.
- C'est un langage interprété (avec une phase interne de pré-compilation)

  ⇨ moins rapide qu'un programme compilé
  ⇨ nécessité d'avoir l'interpréteur Perl sur sa machine

B. A quoi sert Perl ?

A l'origine, Perl à été crée pour écrire des scripts d'administration système, manipuler des fichiers textes (recherche, substitution…), manipuler des processus. Aujourd'hui, on l'utilise essentiellement pour :

- Générer, mettre à jour, analyser des fichiers HTML, et notamment pour l'écriture de script CGI (interface entre un serveur Web et une application).
- Accéder à des bases de données.
- Convertir des formats de fichiers.

C. Pourquoi utilise-t-on Perl en bioinformatique ?

Les besoins de cette discipline imposent aux bioinformaticiens (essentiellement des biologistes de formation) la maîtrise d'un langage de programmation. Depuis quelques années, Perl s'est imposé comme le langage le plus utilisé en bioinformatique :

- Il est relativement simple à apprendre et à maîtriser.
- Il est parfaitement adapté aux besoins de la discipline : traitement de fichiers texte, recherche de motifs, extraction d'information à partir de fichiers d'annotations, conversion de format…

D. Premier programme :

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

- Sous UNIX, la première ligne d'un script Perl doit contenir le chemin d'accès à l'interpréteur Perl présent sur la machine.
- Le seul type de commentaire existant est le # : tout ce qui suit est ignoré jusqu'à la fin de la ligne.
- Chaque instruction doit être terminée par un point virgule.

E. Exécution d'un programme Perl

- Sous UNIX, sous réserve que l'utilisateur dispose des droits d'exécution du fichier, il suffit de taper le nom du fichier précédé éventuellement de son chemin d'accès (si ce chemin ne se trouve pas dans la variable d'environnement PATH) :

  `./monProgramme.pl`
  (fichier dans le répertoire courant)
- Sous DOS/Windows : on appelle explicitement l'interpréteur Perl suivi du nom du fichier :

  `perl monProgramme.pl`

On peut spécifier plusieurs options à l'interpréteur Perl :
-c : vérification de la syntaxe du programme sans l'exécuter
-w : recherche de certaines erreurs de programmation avant d'exécuter (on préfère à cette option l'emploi de la directive `use warnings;` en tête de fichier)
-v : indique la version de l'interpréteur Perl utilisé
-d : mode débugage
…
Exemple :
sous DOS/Windows : `perl –d monProgramme.pl`
sous UNIX, on indique l'option sur la première ligne du programme :
`#!/usr/local/bin/perl –d`

F. Un exemple un peu plus complexe

```
#!/usr/bin/perl

use strict ;
use warnings ;

# Affichage de la séquence « inverse complémentaire » d'une séquence
# entrée par l'utilisateur

# declaration des variables
my ($adn, $revcom) ;
```



```
# Lecture de la séquence au clavier
print "Entrez la sequence : " ;
$adn = <STDIN>;

# copie de la sequence et inversion
$revcom = reverse $adn;

# on effectue les substitutions
$revcom =~ tr/ACGTacgt/TGCAtgca/;

# affichage du resultat
print "Voici la sequence inverse complementaire:\n\n";

print "$revcom\n";
```

Ce programme affiche la séquence inverse complémentaire d'une séquence entrée au clavier (cf. VI F).

Remarque :
Il est fortement conseillé de placer en tête de tout programme Perl les directives **use strict;** et **use warnings;**. La première rend obligatoire la déclaration des variables avant leur utilisation (cf. VII A), la seconde recherche de certaines erreurs de programmation avant d'exécuter (cf. I E).

II. Les identificateurs

- Un nom de variable est toujours précédé par $ (pour désigner un scalaire), @ (pour désigner un tableau complet), % (pour désigner une table de hachage) ou & (pour désigner les appels de sous programmes). (voir partie III Les structures de données)
- 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 identificateur peut contenir un caractère numérique. Par exemple, il est possible d'avoir un nom de variable tel $nom2.
- Enfin, le caractère de soulignement (_) est permis. Aussi, on peut définir des variables comme $nombre_de_jours.

Exemples de noms de variables valides :

```
$nom1
$_nom4
$_NOM_4
...
```

III. Les structures de données

Les variables en Perl ne sont pas typées, mais il existe trois grands types de structures de données : les scalaires, les tableaux et les tableaux associatifs (ou tables de hachage). L'affectation se fait en général par l'opérateur = .

A. Les scalaires :

Leur identificateur doit toujours commencer par **$**

Nombres :
```
$unEntier                = 123;
$unFlottant              = 123.45;
$unFlottantScientifique  = 123.45e10;
$unNombre                = 1_234_567; #les _ sont ignorés
$unHexa                  = 0xffff;
$unOctal                 = 0755;
```

Chaînes de caractères :
Elles sont encadrées par des apostrophes(' '), des guillemets (" "), des accents graves(` `) ou des <<.

- 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 l'interprétation de variables internes.

- Avec des accents graves :

Les accents graves (` `) permettent l'exécution de la commande shell 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 suivi d'un retour chariot

$repertoire_courant = `pwd`;
```



```
print $repertoire_courant; # Affiche le chemin du repertoire courant

$prenom = <<EOS;     # On choisit par ex EOS (End Of String)
Maligne 1            # Comme chaine de terminaison
Maligne 2
EOS
print $prenom; # affiche : Ma ligne 1
               #           Ma ligne 2
```

Notes :

- Pour utiliser des guillemets ou des apostrophes dans une chaîne de caractères, on utilise \' ou préférentiellement la fonction q :

```
$maChaine= "C\'est aujourd\'hui qu'il s\'en va";
$maChaine= q/C'est aujourd'hui qu'il s'en va/;
```

- l'instruction `print` peut prendre plusieurs arguments séparés par des virgules :

```
print 2+3," une chaine\n"; # affiche : 5 une chaine
```

B. Les listes et les tableaux :

Une liste est un ensemble de valeurs scalaires ordonnées, que l'on peut construire de diverses manières. La plus intuitive est l'énumération des éléments entre parenthèses, en les séparant par des virgules :

```
( 1, "chaine", 0x44, $var )
```

La liste vide est représentée par ( );
Les listes sont stockées dans des variables de type tableau (en fait, liste et tableau sont synonymes en Perl) :

```
@monTableau = ( 1, "chaine", 0x44, $var );
```

Il existe également une autre fonction de construction de tableaux, qui clarifie souvent l'écriture des scripts. C'est la fonction `qw`, qui permet de faire une énumération des éléments de la liste en les séparant uniquement par des espaces, d'où un gain de lisibilité.

```
@monTableau = qw( Facile de construire une liste à 8 éléments )
```

- L'identificateur d'un tableau doit toujours commencer par **@** (lorsqu'on manipule le tableau entier).
- Le premier élément d'un tableau possède l'indice 0, comme en C, C++, Java.
- Pour accéder à l'élément i d'un tableau, on utilise `$nomTableau[i]` (qui est donc un scalaire).
- On peut également manipuler des parties de tableaux : `@monTableau[1..5]` désigne le sous-tableau constitué des éléments 1 à 5, alors que `@monTableau[1,5]` désigne un tableau constitué des éléments 1 *et* 5.
- La variable `$#nom_tableau` donne l'indice du dernier élément du tableau `@nom_tableau`. Le nombre d'éléments d'un tableau est donc `$#nom_tableau+1`.

Exemple :

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

$nbElements = $#jours +1;
print "Le tableau jours possède $nbElements éléments";
# Affiche : Le tableau jours possède 7 éléments

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

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

Notes :

- L'opérateur **.** permet de concaténer deux chaînes de caractères.
- Chaque type de variable a son propre espace de nommage : `$var` n'a aucun rapport avec `@var`.

C. Les tableaux associatifs :

Les tableaux associatifs sont gérés de manière transparente par Perl. Chacune des données d'un tableau associatif est représentée par une combinaison clé/valeur. On accède ensuite à la valeur cherchée par sa clé.

- L'identificateur d'un tableau associatif doit toujours commencer par **%**
- Les accolades {} servent à rechercher un élément à partir de sa clé. Par exemple, `$jours{"Lu"}` permet l'accéder à l'élément `Lundi`.

Exemple:

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

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

Il est à noter que l'affectation d'un tableau associatif peut ce faire de la façon suivante :

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


D. Quelques fonctions agissant sur les tableaux

```
push(@<nom du tableau>,<liste>)
# Ajoute une liste à la fin du tableau (ou un element simple)

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

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

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

join("Modele",@<nom du tableau>)
# Retourne la chaîne composée des éléments du tableau séparés par le
Modele

sort(@<nom du tableau>)
# Retourne le tableau trie dans l'ordre alphabétique
# Il est aussi possible de définir des fonctions de comparaison
# adaptées au tri souhaité.

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

Perl possède également une fonction pour ajouter ou supprimer des éléments à n'importe quel endroit d'une liste. Il s'agit de la fonction `splice` qui permet d'ajouter des éléments ou d'en remplacer certains par de nouveaux. La fonction prend quatre arguments :

- la liste à modifier
- la position (indice) qui indique l'endroit où commence le remplacement
- le nombre d'éléments à supprimer
- la liste des éléments à insérer

```
@Nums = (1,2,3,4,5);
splice( @Nums, 3, 1, ("chien", "vache") );
print join(" ",@Nums); # affiche :    1 2 3 chien vache 5
```

Lorsqu'on ne supprime pas d'éléments, les nouveaux éléments sont insérés avant l'élément indiqué par le second argument.
Si on ne fournit pas de liste en quatrième argument, la fonction supprime simplement des éléments de la liste.

Pour les tableaux associatifs, il existe deux fonctions :

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

keys(%<nom du tableau>)
# Retourne la liste des clés associées au tableau
```

IV. Les opérateurs

A. Opérateurs d'affectation

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

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

B. Opérateurs arithmétiques

Le Perl utilise les même notations que le langage C :

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

C. Opérateurs logiques et de comparaison

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.

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 inférieur à l'opérande de droite,
0 si elles sont égales,
+1 si l'opérande de gauche est supérieur à l'opérande de droite.

V. Structures de contrôle

A. Condition

Condition simple

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

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

Conditions imbriquées

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

Exemple :

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

- Les accolades sont obligatoires même si les blocs ne sont composés que d'une seule commande.
- Il est cependant possible d'utiliser une autre syntaxe :

commande if (<condition>)

Les exemples suivants sont équivalents :

```
if ( !open(FICHIER, $nomFichier) ) { die "Impossible d'ouvrir
$nomFichier : $!"; }
die " Impossible d'ouvrir $nomFichier : $!" if ( !open(FICHIER,
$nomFichier) );
```

```
open(FICHIER, $fichier) or die "Impossible d'ouvrir $nomFichier : $!";
```

La dernière syntaxe est la plus utilisée dans ce cas précis (la variable spéciale $! contient le message standard d'erreur indiquant la cause du problème).

- Perl utilise des nombres pour représenter les valeurs true et false : toute valeur égale à zéro représente la valeur false, et toute valeur différente de zéro vaut true (attention donc à une condition du type : `if ($a=4) ...` qui sera évaluée comme vraie quelle que soit la valeur de a).

B. Boucle for

Le Perl dispose de la structure for, qui est identique à celle du C.

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

Exemple :

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

C. Boucle foreach

foreach VAR (ARRAY) { <action(s)> }

permet d'affecter successivement à la variable VAR tous les éléments du tableau ARRAY. La variable VAR est implicitement locale à la boucle et retrouve sa valeur initiale - si elle existait - à la sortie de la boucle.

Exemple :

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

D. 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
$in = <STDIN>; # Lit le mot entré
chomp($in);    # Enlève 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 : ";
$in = <STDIN>; # Relit le mot entré
chomp($in);    # Enlève le retour chariot
}
```


E. 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 vraie. Si vous utilisez *until* les actions sont exécutées *jusqu'à ce que* la condition soit vraie.

Exemple :

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

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

F. 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 (au test), et par exemple, l'incrémentation est effectuée dans le cas d'une boucle for.

`redo` Force à recommencer une itération de la boucle (avec les mêmes conditions).

`goto` LABEL Saut du programme à une étiquette spécifiée (LABEL). Cette instruction existe, mais il est déconseillé de l'utiliser si l'on veut programmer proprement...

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

Exemple:

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

Note :
Il n'existe pas d'instruction Switch comme en C, mais il est possible de l'émuler de différentes manières :

```
SWITCH : {
if  (EXPR1) { …; last SWITCH; }
if  (EXPR2) { …; last SWITCH; }
if  (EXPR3) { …; last SWITCH; }
…;
}
```

VI. Les opérations sur les chaînes de caractères

A. Opérateurs

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

```
$a = $b . $c;             # Concaténation de $b et $c
$a = $b x $c;             # $b répété $c fois
$a = substr($b, 5, 10);   # Sous-chaine de $b composée de 10 caractères
                          # à partir de la 5e position
$a = lc($b);              # Transforme en minuscule $b (Lower Case)
$a = uc($b);              # Transforme en majuscule $b (Upper Case)
$a = lcfirst($b);         # Transforme en minuscule le premier caractère
                          # de $b
$a = ucfirst($b);         # Transforme en majuscule le premier caractère
                          # de $b
$a = length($b);          # Renvoie le nombre de caractères de $b
```

la fonction `join` :

La fonction `join` prend les éléments d'une liste et les convertit en une chaîne de caractères. Les éléments sont ajoutés 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
```

B. Expressions régulières

Les expressions régulières sont une caractéristique de Perl qui rendent ce langage particulièrement adapté au traitement des fichiers texte.

Une expression régulière est une suite de caractères suivant une certaine syntaxe qui permet de décrire le contenu d'une chaîne de caractère, afin de tester si cette dernière correspond à un motif, d'en extraire des informations ou bien d'y effectuer des substitutions.
Une expression régulière est contenue par des slash "/".

C. Appartenance

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

Exemple :

```
$phrase =~ /le/      # vrai si $phrase contient le motif "le"
$phrase !~ /le/      # vrai si $phrase ne contient pas le motif "le"

$phrase = "Mon langage préféré est perl.";
if ($phrase =~ /perl/) {
   print "Bravo\n";
}
# Condition vraie.
```

D. Notions avancées

L'exemple précédent montre une limite par exemple si "linux" est écrit "Linux" le test sera faux. C'est pour cela qu'il existe des modificateurs et des métacaractères qui permettent 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 : *=~ /motif/modificateur*.

i	Reconnaissance de motif indépendamment de la casse (majuscules/minuscules) (Ignore case)
m	Permet de traiter les chaînes multi-lignes. Les caractères "^" et "$" reconnaissent alors n'importe quel début ou fin de ligne plutôt que début ou fin de chaîne.
s	Permet de traiter une chaîne comme une seule ligne. Le caractère "." reconnaît alors n'importe quel caractère... même une fin de ligne qui normalement n'est pas reconnue.
x	Augmente la lisibilité 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 **métacaractères** comme leurs noms l'indiquent sont des caractères intégrés au motif. Voici certain de ces caractères spéciaux, et leurs significations.

```
.     # Reconnaît n'importe quel caractère, excepté les retours chariot.
^     # Reconnaît le début d'une ligne ou d'une chaîne.
$     # Reconnaît la fin d'une ligne ou d'une chaîne.
*     # Reconnaît 0 fois ou plus le dernier caractère.
+     # Reconnaît 1 fois ou plus le dernier caractère.
```

```
?       # Reconnaît 0 fois ou 1 fois le dernier caractère.
{n}     # Reconnaît n fois exactement le dernier caractère.
{n,}    # Reconnaît au moins n fois le dernier caractère.
{n,m}   # Reconnaît au moins n fois mais pas plus de m fois le dernier caractère.
\       # Annule le sens du caractère spécial qui suit.
|       # Alternative (« ou »)
()      # Groupement ou modèle
[]      # Classe de caractères
```

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....)
```

Les classes de caractères [] (crochet) sont utilisées pour assortir n'importe lequel des caractères qu'elles 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 à z inclus
[^a-z]          # Pas de lettre minuscule
[^a-zA-Z]       # Pas de lettres
[a-z]+          # N'importe quelle enchaînement de lettres
```

Pour faciliter l'utilisation des classes de caractères il existe un certain nombre de groupes prédéfinis :

		équivaut à :
\w	Reconnaît un caractère de "mot" (alphanumérique plus "_").	[a-zA-Z0-9_]
\W	Reconnaît un caractère de non-"mot".	[^a-zA-Z0-9_]
\d	Reconnaît un chiffre.	[0-9]
\D	Reconnaît un caractère autre qu'un chiffre.	[^0-9]
\s	Reconnaît un caractère d'espacement (espace, tab, retour chariot,etc...)	
\S	Reconnaît un caractère autre qu'un espacement.	

Il existe d'autres caractères spéciaux :

\b	Reconnaît la limite d'un mot
\B	Reconnaît autre chose qu'une limite de mot
\A	Reconnaît uniquement le début de la chaîne
\Z	Reconnaît uniquement la fin de la chaîne (ou juste avant le caractère de nouvelle ligne final)
\z	Reconnaît uniquement la fin de la chaîne

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

Exemple :

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

E. 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 langage préféré est perl.";
$phrase =~ s/perl/Perl/;
print $phrase."\n";
# affichera "Mon langage préféré est Perl."
```

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

Exemple :

```
$phrase = "Mon langage préféré est PERL, perl, PerL...";
$phrase =~ s/perl/Perl/gi;
print $phrase."\n";
# affichera "Mon langage préféré est Perl, Perl, Perl..."
```

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

Exemple :

```
$phrase = "Mon langage préféré est perl";
$phrase =~ s/(perl)/Perl/;
print "Mon langage préféré est ".$1."\n";
# affichera "Mon langage préféré est perl..."
```

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

F. Traduction

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

G. Eclatement

La fonction split permet de décomposer une chaîne en une liste d'éléments en repérant un motif défini qui sert de séparateur. Sa syntaxe est la suivante :
@tableau = split(/motif/, $chaine);

Exemple 1 :

```
$phrase = "12 15 25";
($v1, $v2) = split(/\s+/, $phrase);
# Utilise un ou plusieurs (+) espaces (\s) comme séparateur
# On a alors $v1 = "12", $v2 = "15"
```

Exemple 2 :

```
$phrase = "Facile de separer une liste de mots";
@list = split(/\s+/, $phrase);
foreach $mot (@list) {
print $mot."\n";
}
```

Ce programme donnera le résultat suivant :

```
Facile
de
separer
une
liste
de
mots
```

VII. Les fonctions

Les fonctions (ou procédures) peuvent être placées n'importe où dans le programme, mais il est préférable des les placer toutes au début ou à la fin. Une procédure a la forme suivante :

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

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

```
&ma_procedure;
```

Perl ne distingue pas les fonctions suivant le nombre de leurs arguments, d'ailleurs on ne déclare pas explicitement quels sont ces arguments. L'appel suivant est parfaitement correct et provoquera l'exécution de la procédure déclarée plus haut :

```
&ma_procedure("toto",4);
```

Par contre, les paramètres ne seront pas exploités.

A. Variables locales

Lorsque l'on crée une procédure, il est préférable que les variables utilisées dans la procédure n'interfèrent pas avec celles contenues dans le reste du programme. On définit donc des variables locales avec les fonctions `local(…)` ou `my(…)` de la manière suivante :
local(<$var1>, <$var2>, ... , <$varN>);
ou
my(<$var1>, <$var2>, ... , <$varN>);

Ces variables locales ne sont définies qu'à l'intérieur de la procédure, et sont réinitialisées à chaque appel.
La fonction `local` masque une variable uniquement aux yeux du bloc *qui appelle* la procédure concernée, laissant cette variable visible aux procédures *appelées* dans le bloc contenant `local`. La fonction `my` quant à elle, masque une variable à tous les autres blocs du programme, qu'ils soient appelés ou appelants. L'exemple suivant illustre cette différence :

```
sub PrintJ { print "La valeur de J est $J \n" }
sub PrintK { print "La valeur de K est $K \n" }

my($J) = 5 ;
local($K) = 10 ;
&PrintJ ;
&PrintK ;
```

On obtient l'affichage suivant :

```
La valeur de J est
La valeur de K est 10
```

Remarque :
En Perl, il n'est pas nécessaire de déclarer les variables avant leur utilisation et ceci peut être source de nombreuses erreurs de programmation. C'est pourquoi il est possible de placer en tête du fichier source la directive :

```
use strict;
```

Cette directive interdit l'utilisation de toute variable qui n'aurait pas été déclarée au préalable à l'aide de `local` ou `my`. Il est fortement conseillé de l'utiliser.

B. Paramètres

On peut passer un nombre de paramètres quelconque à une procédure :

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

Quand on appelle 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 paramètres sont : "; print @_; # Affiche tous les paramètres
print "\n\n";
print "Le 1er paramètre est : $_[0]\n";
```

```
print "Le 2em paramètre est : $_[1]\n";
}
```

Exemple 2:

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

C. Renvoi des valeurs

La valeur renvoyée par une fonction 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.

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.
}
```

D. Les bibliothèques de fonctions ou Modules

Il est possible de créer des bibliothèques de fonctions réutilisables. En Perl, on les nomme Modules.

1) Création d'un module

Les fonctions doivent être placées dans un fichier portant l'extension .pm (pour Perl Module).

Exemple :
Fichier math.pm

```
sub oppose
{
   my ($nombre)=@_;

   return - $nombre ;
}
1;
```

Lors du chargement d'un module, la dernière valeur évaluée doit toujours avoir la valeur "vrai". C'est pourquoi il est d'usage de toujours placer `1;` en fin de module. Mais on peut également envisager de mettre un test en dernière instruction pour vérifier si les conditions sont réunies pour l'usage du module.

2) Utilisation d'un module

Exemple :

```
use math;           # nom du fichier module sans l'extension
print &oppose(5);   #affiche -5
```

VIII. Les entrées-sorties

Petit programme de base :

```
#! /usr/local/bin/perl
print "entrez votre nom : ";
$nom = <STDIN>;
chomp($nom);                    # supprime le dernier caractère de la
                                # chaine si c'est un retour chariot
print "Bonjour $nom\n";
```

L'opérateur < DESCRIPTEUR > lit les caractères sur le flux correspondant jusqu'au prochain retour chariot, qui sera d'ailleurs inclus dans le résultat.

A. Manipulation de fichiers

Pour ouvrir un fichier on utilise la commande Open :

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

open ( descripteur,"+<nom_fichier");#Ouvre un fichier en lecture et en
écriture

open ( descripteur,">nom_fichier"); #Ouvre un fichier en écriture avec
écrasement en cas d'existence préalable du fichier

open ( descripteur,">>nom_fichier"); #Ouvre un fichier en écriture pour
ajout en fin de fichier

open ( descripteur,"+>nom_fichier"); #Ouvre un fichier en lecture et
écriture avec écrasement en cas d'existence préalable du fichier
```

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

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

```
$ligne = < descripteur >;  #Lit ligne par ligne dans un
                           #fichier

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

Une fois les opérations voulues effectuées on ferme le fichier :

```
close (descripteur);
```

Par défaut il existe 3 descripteurs ouverts à l'exécution du programme :
<STDIN> : Descripteur d'entrée, Entrée standard en lecture seule.
<STDOUT>: Descripteur de sortie. Sortie standard en écriture seule.
<STDERR>: Descripteur des messages d'erreurs. Sortie d'erreurs standard en écriture seule.

Utiliser 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.

Les commandes Open et Close renvoient 1 ou 0 en cas de succès ou d'échec, on utilise donc cette propriété pour vérifier qu'il n'y a pas d'erreur à l'ouverture ou d'absence du fichier 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 : tri d'un fichier

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

@result = sort @fichier; # tri des lignes par ordre alphabetique

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

B. 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).

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

rename(ancien nom du fichier, nouveau nom du fichier) #Renomme ou
déplace un fichier

symlink(nom du fichier, nom du lien) #Crée un lien symbolique (Unix
seulement, ln -s)

chdir(nom du répertoire) #Change de répertoire courant
```



```
mkdir(nom, mode) # Crée un nouveau répertoire. mode représente les
permissions (Unix) à attribuer au répertoire.

rmdir(nom) #Efface un répertoire
```

C. Tests 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 exécutable (permission x)
-w	Le fichier est accessible en écriture (permission w)
-d	Le répertoire 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.
}
```

D. 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);
# Exécute la commande et récupère la sortie standard dans le descripteur
```

Exemple :

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

E. Formatage de la sortie

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

Utilisation de write

Mais il est 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 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.

- Exemple :
On désire écrire les noms et prénoms des personnes contenus dans un tableau associatif par colonnes avec comme en-tête 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, $prenom # le nom sur 8 caractères et le prénom sur 10
.

foreach $nom (keys %tab) {
$prenom = @tab{$nom};
write;
}
```

IX. Variables et tableaux spéciaux

A. Les arguments de la ligne de commande

Le tableau spécial `@ARGV` contient les paramètres passés au programme par la ligne de commande.
`$ARGV[0]` contient le 1er paramètre
`$ARGV[1]` contient le 2ème paramètre
etc…

Contrairement à d'autres langages, le nom du programme n'est pas contenu dans le tableau `@ARGV`.

B. Variables spéciales

$_	La dernière ligne lue dans un fichier
$!	La dernière erreur, utilisée dans les détections d'erreurs
$1, $2,...	Le contenu de la parenthèse numéro n dans la dernière expression régulière
$0	Le nom du programme
$&	Contient après l'utilisation d'une expression régulière la valeur correspondant au modèle de recherche.
&`	Contient la partie de l'expression complète avant la valeur correspondant au modèle de recherche.
&'	Contient la partie de l'expression complète après la valeur correspondant au modèle de recherche.

C. Tableaux spéciaux

@_	Contient les paramètres passés à une procédure.
@ARGV	Contient les paramètres passés au programme.
%ENV	Table de hachage contenant toutes les variables d'environnement.

X. Exemples

A. Recherche de motifs dans une séquence protéique contenue dans un fichier

```
#!/usr/bin/perl -w
# Recherche de motifs

# Saisie du nom du fichier contenant la sequence proteique
print "Entrez le nom du fichier contenant la sequence : ";

$nomfichier = <STDIN>;

chomp $nomfichier;

# ouverture du fichier
unless ( open(FICPROT, $nomfichier) ) {

    print "Impossible d'ouvrir le fichier \"$nomfichier \"\n\n";
    exit;
}

# Lecture et stockage de toutes les lignes du fichier
# dans le tableau @protein
@protein = <FICPROT>;

# Fermeture du fichier.
close FICPROT;

# On rassemble toutes les lignes en une seule chaine de caracteres
$protein = join( '', @protein);

# On enleve les espaces et retours à la ligne
$protein =~ s/\s//g;

# Boucle principale : recherche du motif
do {
    print "Entrez un motif a : ";

    $motif = <STDIN>;
```



```
    chomp $motif;

    if ( $protein =~ /$motif/ ) {

        print "Je l'ai trouve !\n\n";

    } else {

        print "Je ne l'ai pas trouve.\n\n";
    }

# sortie lorsque l'utilisateur n'entre rien
} until ( $motif =~ /^\s*$/ );
```

B. Fréquence des nucléotides dans une séquence

```
#!/usr/bin/perl

# Saisie de la sequence
print "Entrez la sequence :";

$DNA = <STDIN>;

chomp $DNA;

# Initialisation des compteurs
$count_of_A = 0;
$count_of_C = 0;
$count_of_G = 0;
$count_of_T = 0;
$errors     = 0;

for ( $position = 0 ; $position < length $DNA ; ++$position ) {

    $base = substr($DNA, $position, 1);

    if     ( $base eq 'A' ) {
        ++$count_of_A;
    } elsif ( $base eq 'C' ) {
        ++$count_of_C;
    } elsif ( $base eq 'G' ) {
        ++$count_of_G;
    } elsif ( $base eq 'T' ) {
        ++$count_of_T;
    } else {
        print "!!!!!!!! je ne reconnais pas cette base: $base\n";
        ++$errors;
    }
}

# affichage des resultats
print "A = $count_of_A\n";
print "C = $count_of_C\n";
print "G = $count_of_G\n";
print "T = $count_of_T\n";
print "erreurs = $errors\n";
```

C. Génération aléatoire de séquence

```
#!/usr/bin/perl

@bases = ('A','T','G','C');
$longueur=40;
$seq="";
srand(time|$$);

for($i=0;$i<$longueur;$i++)
{
  $seq.= $bases[rand @bases];
}

print $seq;
```

XI. Perl et CGI

Common Gateway Interface (CGI) est un standard pour l'écriture de passerelle entre un serveur Web et une application externe. Un script CGI permet par exemple de récupérer des informations saisies à partir d'un formulaire HTML et de les transmettre à une application. Un exemple classique nous est fourni par les moteurs de recherche, qui à partir de mots clés saisis dans un formulaire, vous affichent l'ensemble des pages au format HTML contenant ces mots clés, la recherche étant effectuée par un programme externe.
Du fait de son aptitude à traiter les chaînes de caractère, Perl est particulièrement adapté à l'écriture de scripts CGI.

A. Premier script CGI

```
#!usr/bin/perl

print "Content-type: text/html\n\n";

open(LISTE, 'ls /home/ |') or die "Impossible d'executer la commande ls:
$!";
print "Liste des utilisateurs : <UL>";
while ($nom =  <LISTE>) {
   print "<LI>$nom";
}
print "</UL>";
```

Ce programme crée un fichier HTML qui ressemble à :

```
Content-type: text/html

Liste des utilisateurs : <UL>
<LI>seb
<LI>benny
```



```
...
<LI>toto
</UL>
```

Et le serveur Web affichera une page qui aura l'aspect suivant :

Liste des utilisateurs :
- seb
- benny
- toto

B. Récupération de données issues d'un formulaire

Dans la plupart des cas, un script CGI sert à récupérer des données issues d'un formulaire. Rien n'oblige le formulaire à être sur le même serveur que le CGI. Par contre, le CGI doit être dans un répertoire spécial du serveur Web (en général `/usr/local/httpd/cgi-bin/prog.pl` sous UNIX) et sera accessible avec l'URL correspondante (en général http://seveur/cgi-bin/prog.pl).

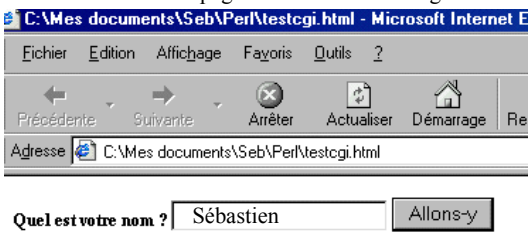
Exemple : le script bonjour.pl

```
#!/usr/bin/perl

use CGI_Lite;  # Utilisation d'un module CGI specifique
$cgi = new CGI_Lite;
%in = $cgi->parse_form_data;     # Lecture des paramètres

print "Content-type: text/html\n\n";
print "<h1>Bonjour $in{'nom'} !</h1>";
```

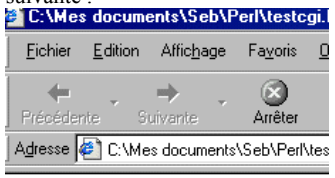
Ce CGI utilise un module spécial (CGI_Lite) qui permet de lire les données d'un formulaire et de les placer dans un tableau associatif %in.
Il reste ensuite à écrire un formulaire qui remplit les données dont on a besoin :

```
<form action = "http://serveur/cgi-bin/bonjour.pl" METHOD=POST>
<b>Quel est votre nom ?</b>
<input type=text name=nom>
<input type=submit value="Allons-y">
</form>
```

Ce formulaire affiche la page suivante sur un navigateur Web :



L'appui sur le bouton lance l'exécution du programme Perl et ce dernier affiche la page suivante :



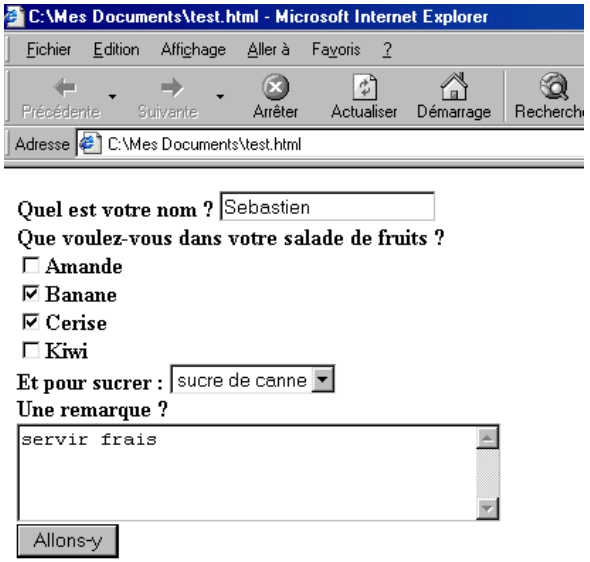
Bonjour Sébastien !

C. *Un exemple un peu plus complexe*

Utilisation de différents «tags» pour saisir des informations.
Un problème se pose avec les tags de type «sélection multiple » : nous avons plusieurs valeurs dans une même variable... Pour résoudre ce problème le module CGI_Lite offre une fonction $cgi->get_multiple_values ($in{'*champ*'}) qui permet de retourner sous forme de tableau les différentes valeurs du champ. On va créer un CGI qui «crée » une salade de fruits avec les paramètres de l'utilisateur.
Voici tout d'abord le formulaire qui permet de saisir les informations :

```
<form action="salade.pl" METHOD=POST>
<b>Quel est votre nom ?</b>
<input type=text name=nom>
<br>
<b>Que voulez-vous dans votre salade de fruits ?<br>
<input type=checkbox name=fruit value="amande">Amande<br>
<input type=checkbox name=fruit value="banane">Banane<br>
<input type=checkbox name=fruit value="cerise">Cerise<br>
<input type=checkbox name=fruit value="kiwi">Kiwi<br>

Et pour sucrer :
<select name=sucre>
<option checked>sucre vanille
<option>sucre de canne
<option>rien du tout
</select>
<br>
Une remarque ? <br>
<textarea name=remarques cols=40 rows=4></textarea>
<br>
<input type=submit value="Allons-y">
</form>
```

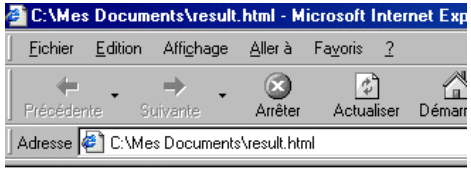
Ce qui donne sur un navigateur web :



Le script Perl permet d'exploiter ces données :

```
#!/bin/perl
use CGI_Lite;              # Module CGI_Lite
$cgi = new CGI_Lite;       # On crée un objet de type CGI
%in = $cgi->parse_form_data;     # On lit les données

print "Content-type: text/html\n\n";  # Indispensable: header

# Les champs du formulaire sont maintenant dans le tableau associatif
%in
# On fait reference a un champ par $in{'champ'}
# ATTENTION: Certains champs ont plusieurs valeurs
# (select multiple, cases a cocher...)
# dans ce cas on fait reference aux valeurs dans un tableau
# obtenu par $cgi->get_multiple_values($in{'champ'})

print "Bonjour $in{'nom'} !<p>\n";
print "Voici donc une petite salade de fruits :\n";
print "<br>Composee de <ul>\n";

# Pour toute valeur
foreach $f ($cgi->get_multiple_values($in{'fruit'})) {
    print "<li>$f\n";
}
print "</ul>\n";
print "Et pour la douceur un petit peu de $in{'sucre'}<br>\n";
if (exists($in{'remarques'}== { # Si le champ a ete rempli
    print "Nous avons tenu compte de votre remarque<br>\n";
    print "<b>$in{'remarques'}</b>\n";
}
```

Ce script provoque l'affichage de la page web suivante :



XII. Les références et la programmation orientée-objet

Une des limitations de Perl 4 venait du fait qu'il était impossible de manipuler des structures de données plus évoluées que de simples tableaux sans passer par des astuces plus ou moins efficaces. Une des grandes innovations de Perl 5 a été l'introduction des références, qui permettent de travailler sur des structures de données plus complexes.

A. Principe

Une référence est un scalaire qui pointe sur une structure de données (scalaire, tableau, tableau associatif, fonction). Il est possible de construire des tableaux de références, donc des tableaux multidimensionnels. Perl garde une trace du nombre de références sur un objet. Une fois que celui-ci atteint 0, l'objet est détruit et la mémoire libérée.

B. Déclaration – Syntaxe

Soit `@tab` un tableau. On peut alors obtenir sa référence avec l'expression suivante :
`$ref=\@tab;`
Si on exécute la ligne `print $ref;` on obtient quelque chose du type :
`ARRAY(0x91a9c)`

Il y a deux manières d'utiliser ensuite cette référence. On peut la déréférencer pour la manipuler comme un tableau normal :

```
@new = @$ref;
print $new[0];
```

ou bien on peut accéder directement aux données en remplaçant le nom du tableau par la référence :

```
print $$ref[0];
# ou bien
print $ref->[0];
```

C. Création de références anonymes

Il est possible de créer des références sur des structures anonymes en vue d'obtenir des structures de données plus complexes. Voici par exemple comment créer un tableau de tableaux :

```
@liste = ( ["toto", "tutu", "tata"] ,
           ["truc", "much"],
           ["patati", [patata] );
```

`@liste` est une liste de références sur des tableaux anonymes. L'affectation d'un élément s'opère à la manière d'un tableau à deux dimensions. L'affectation d'une liste complète implique l'utilisation de références.

```
# remplacer "truc" par "machin"
@liste[1][0]="machin";
# remplacer ["patati","patata] par ["bla","blabla"]
@autreListe = ("bla","blabla");
$Liste[2] = \@autreListe;
```

L'utilisation de tableaux à deux dimensions se fait de façon naturelle :

```
@tab = ( [ 0,1],    [1,2] ) ;
print $tab[1][1] ;  #affiche 2
```

D. Les objets

Perl n'est pas à proprement parler un langage orienté-objet, mais l'apparition des références dans la version 5, associée à certaines fonctions spécifiques, permet de manipuler des objets.

- Un exemple limité à l'usage classique des packages :

Si l'on considère une classe d'objet comme une structure de données accompagnée de procédures qui régissent le comportement de ses membres potentiels, alors en Perl on peut essayer d'utiliser les packages.
Dans l'exemple suivant, on désire gérer des *objets* d'une *classe* appelée *voiture* caractérisés chacun par une marque et une couleur, à tout moment on souhaite connaître le nombres de voitures en cours.

```
#!/usr/local/bin/perl
use voiture;
$voit1 = voiture::nouvelle('verte','citroen');
$voit2 = voiture::nouvelle('bleue','renault');
$voit3 = voiture::nouvelle('rouge','citroen');
printf (" %s \n", voiture::couleur($voit1));
printf (" %s \n", voiture::marque($voit2));
printf (" %s \n", voiture::couleur($voit3));
printf (" %d \n", voiture::total());
```

```
package voiture;
BEGIN { @liste=();}
sub nouvelle {
    my ($color,$marque) = @_;
    my $objptr = {};
    $objptr->{'couleur'}=$color;
    $objptr->{'marque'}=$marque;
    push(@liste,$objptr);
    return $objptr;
}
sub couleur {
    my ($objptr) = @_;
    return ($objptr->{'couleur'});
}
sub marque {
    my ($objptr) = @_;
    return ($objptr->{'marque'});
}
sub total {
    return($#liste+1);
}
1  #  Tous les package doivent renvoyer 1 (vrai) pour que
   #  l'instruction use nom_package réussisse
```

Dans l'exemple précédent, la *classe voiture* est un package, le constructeur de package BEGIN est utilisé pour initialiser la liste des *objets* à vide.
La fonction *nouvelle* est le constructeur de la *classe*, elle reçoit en arguments la couleur et la marque d'un *objet* et retourne la référence de l'*objet* instancié. La référence de l'*objet* est ici une référence sur un tableau associatif anonyme ($objptr = {}). Les fonctions *couleur* et *marque* sont les *méthodes d'instance* de la *classe voiture*. Les fonctions *nouvelle* et *total* sont des *méthodes de classe*.
Si le début du code qui utilise la *classe voiture* s'apparente à du code orienté objet (appel au constructeur notamment), l'utilisation des méthodes s'en éloigne. La référence à l'objet ($voit3 par exemple) n'a pas de souvenir du type de l'objet référencé et on ne peut l'utiliser qu'en citant le nom du package.
Perl propose donc un type de références particulier, il va permettre d'appeler les méthodes directement à partir de la référence à l'objet. Dans l'exemple précédent on souhaite plutôt écrire $voit3->couleur() et non pas voiture::couleur($voit3).

- Référencer une classe :

Dans l'exemple de la classe *voiture* les références $voit1, $voit2 et $voit3 n'ont pas connaissance de la classe d'appartenance des objets qu'elles référencent. La fonction bless est utilisée pour celà, on peut maintenant modifier le constructeur de la classe *voiture* comme suit:

```
sub nouvelle {
    my ($classe,$color,$marque) = @_; # le nom de la classe
                                      # est le 1er argument
    my $objptr = {};
    $objptr->{'couleur'}=$color;
    $objptr->{'marque'}=$marque;
    bless $objptr;                    # référence la classe
    push(@liste,$objptr);
    return $objptr;
}
```


Un affichage de la valeur de $objptr indique explicitement la classe de l'objet référencé (*voiture=HASH(0xca454)*).
Le programme qui utilise la classe *voiture* peut maintenant être modifié de la manière suivante :

```
#!/usr/local/bin/perl
use voiture;
$voit1 = voiture->nouvelle('verte','citroen');
$voit2 = voiture->nouvelle('bleue','renault');
$voit3 = voiture->nouvelle('rouge','citroen');
printf (" %s \n", $voit1->couleur());
printf (" %s \n", $voit2->marque());
printf (" %s \n", $voit3->couleur());
printf (" %d \n", voiture->total());
```

Il convient ici de noter la forme d'appel des méthodes de classe (voiture->nouvelle()) et la forme d'appel des méthodes d'instance ($voit1->couleur()). Les méthodes de classe s'appellent en précédant la méthode par le nom de la classe, et, le nom de la classe est alors passé en premier argument (ce qui justifie le changement intervenu dans notre classe *voiture*).
Les méthodes d'instance reçoivent comme premier argument une référence d'objet, il n'est donc pas nécessaire de modifier le code de notre classe *voiture*.

- Hériter d'une classe :

Perl autorise de construire de nouvelles classes à partir de classes existantes en utilisant des techniques d'héritages :
- la liste @ISA indique à un package (à une classe) quels sont les ancêtres dont il hérite;
- de manière à pouvoir être hérité, le constructeur d'une classe de base doit rendre une référence *blessee* de la classe spécifiée (de la classe dérivée).

Pour construire une classe appelée *location* (pour gérer les voitures de location) héritant de notre classe *voiture*, on peut procéder comme suit :

```
package location;
use voiture;        # inclure voiture.pm pour la compile
@ISA = ("voiture"); # hériter des méthodes de la classe
                    # voiture.
sub loc {
  my ($classe, $couleur, $marque, $nbjour, $pu) = @_;
#  appel du constructeur hérité de la classe de base, il
#  rendra un objet de classe location
  my $locptr = $classe->nouvelle($couleur,$marque);
  $locptr->{'pu'}=$pu;
  $locptr->{'nbjour'}=$nbjour;
  return($locptr);
}
sub tarif {
    my ($locptr) = @_;
    return ($locptr->{'pu'} * $locptr->{'nbjour'});
}
```

Il convient de bien noter que la classe *voiture* doit être modifiée pour que le constructeur *nouvelle()* rende un objet de la classe *location* et non pas un objet de la classe *voiture* :

```
sub nouvelle {
    my ($classe,$color,$marque) = @_; # le nom de la classe
                                      # est le 1er argument
    my $objptr = {};
    $objptr->{'couleur'}=$color;
    $objptr->{'marque'}=$marque;
```

```
    bless $objptr,$classe;  <==================
    push(@liste,$objptr);
    return $objptr;
}
```

Un programme utilisant la classe *location* peut s'utiliser comme dans l'exemple ci-dessous :

```
#!/usr/local/bin/perl
use location;
$voit1 = location->loc('verte','peugeot',10,100);
print $voit1;
print $voit1->couleur();
print $voit1->tarif();
```

XIII. Où trouver de la documentation sur Perl ?

- www.perl.org Toutes sortes d'infos sur Perl
- http://perldoc.perl.org/ La documentation officielle
- www.cpan.org Comprehensive Perl Archive Network, on y trouve l'ensemble des modules additionnels du langage (dont BioPerl)

Introduction à Perl pour la bioinformatique, James Tisdall, 1re édition, juillet 2002
ISBN : 2-84177-206-3, 400 pages
Un excellent ouvrage d'où sont tirés certains exemples de ce cours.

Mastering Perl for Bioinformatics, James Tisdall, September 2003
0-596-00307-2, 396 pages
La suite logique du précédent, bientôt en français.