I. INTRODUCTION

A. Qu'est ce que Perl ?

P.E.R.L. signifie Practical Extraction and Report Language. Que l'on pourrait (essayer de) traduire par « langage pratique d'extraction et d'édition ».
Créé en 1986 par Larry Wall (ingénieur système). Au départ pour gérer un système de « News » entre deux réseaux.

C'est :

- Un langage de programmation
- Un logiciel gratuit (que l'on peut se procurer sur Internet notamment)
- Un langage interprété :
  ⇒ pas de compilation
  ⇒ moins rapide qu'un programme compilé
  ⇒ chaque « script » nécessite d'avoir l'interpréteur Perl sur la machine pour s'exécuter.

Pourquoi Perl est devenu populaire :

- portabilité : Perl existe sur la plupart des plateformes aujourd'hui (Unix, NT, Windows, Mac, VMS, Amiga, Atari ...)
- gratuité : disponible sur Internet (ainsi qu'un nombre impressionnant de librairies et d'utilitaires)
- simplicité : Quelques commandes permettent de faire ce qu'un programme de 500 lignes en C ou en Pascal faisait.
- robustesse : Pas d'allocation mémoire à manipuler, chaînes, piles, noms de variables illimités...

B. Quelle utilisation ?

A l'origine Perl a été créé pour :
1) manipuler des fichiers (notamment pour gérer plusieurs fichiers en même temps),
2) manipuler des textes (recherche, substitution),
3) manipuler des processus (notamment à travers le réseau).

⇨ Perl était essentiellement destiné au monde UNIX

Pourquoi utilise t'on Perl aujourd'hui ?
1) générer, mettre à jour, analyser des fichiers HTML (notamment pour l'écriture de CGI),
2) accès « universel » aux bases de données,
3) conversion de formats de fichiers.

⇨ Perl n'est plus lié au monde UNIX

Perl n'est pas fait pour :

- écrire des interfaces interactives (mais il existe maintenant le module Tk, qui le permet, sur Unix ou Windows),
- le calcul scientifique (Perl n'est pas compilé : problème de performances si l'on veut par exemple faire des multiplications de matrices, mais là encore il existe le module PDL).

Deux versions

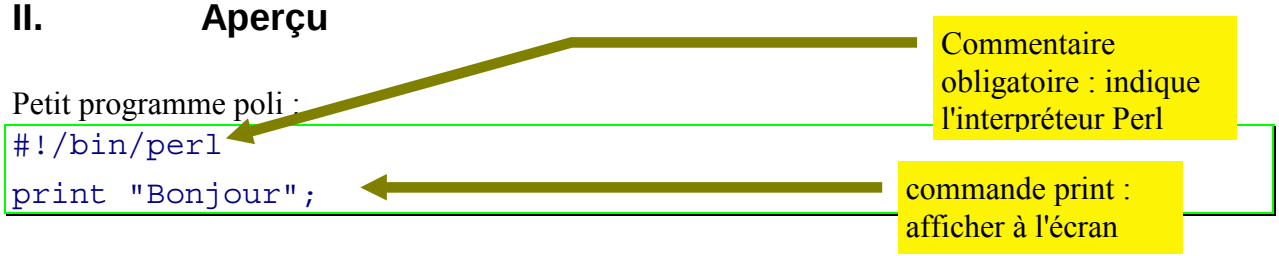
Trois versions principales de Perl :

- Perl 4 : La vieille version qui n'intégrait pas la programmation objet (parfois encore utilisée)
- Perl 5.6 : L'ancienne version
- Perl 5.8 : La nouvelle version qui intègre Unicode (encodage de caractères internationaux)

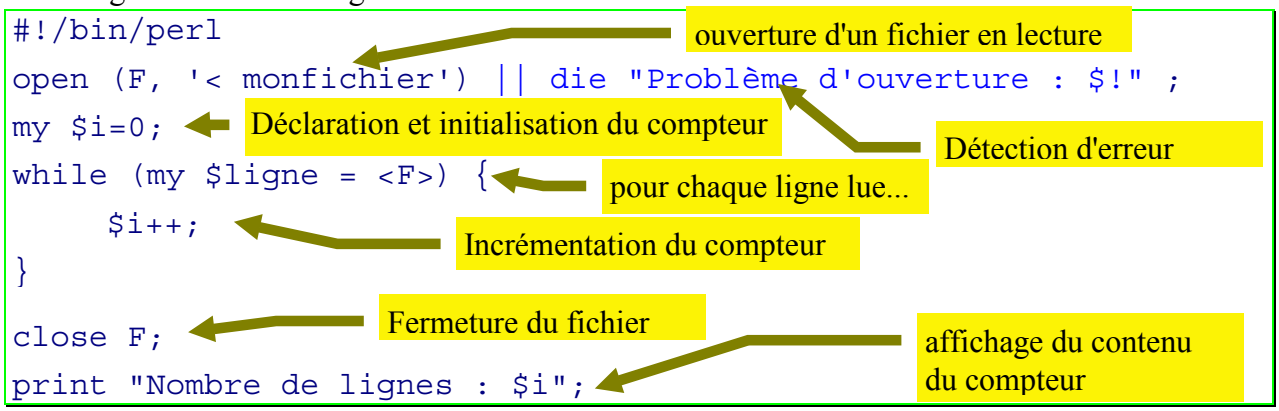
=> À venir: Perl 6.0 qui sera une réécriture complète du langage.

II. Aperçu

Petit programme poli :



Affichage du nombre de lignes d'un fichier :



F est un descripteur de fichier, que l'on peut appeler comme on veut (l'usage veut que l'on note les descripteurs de fichier en majuscules).
Chaque instruction Perl se termine par un point-virgule.

Cf Perldoc.com perl [1]

III. Types de données

A. Constantes

`1, -12345, 0.1415, 1.6E16` *(signifie 160 000 000 000 000 000)*`, 'cerise', 'a', 'les fruits du palmier sont les dattes'`

[1] Quand vous installez Perl (cf annexe « Installation du programme principal » p. 53), vous avez accès à la documentation en ligne « perldoc *module* ». Ce symbole « Perldoc.com module » signifie que vous pouvez lire cette documentation en ligne, ou y accéder sur le Web (et même en français parfois, cf « Bibliographie » p. 49)

B. Scalaires

Les scalaires sont précédés du caractère $

```
$i = 0; $c = 'a';
$mon_fruit_prefere = 'kiwi';
$racine_carree_de_2 = 1.41421;
$chaine = '100 grammes de $mon_fruit_prefere';
=> '100 grammes de $mon_fruit_prefere'
$chaine = "100 grammes de $mon_fruit_prefere";
=> '100 grammes de kiwi'
```

Attention : Ne mettez pas d'espaces dans les noms de variables. Un nom peut être aussi long qu'on le veut. Dans les dernières versions de Perl, les noms de variables peuvent être accentués:

```
$€_écrit_en_grec = "ευρώ";
```

C. Tableaux, listes

En Perl, les tableaux peuvent être utilisés comme des ensembles ou des listes.
Toujours précédés du caractère « @ »

```
@chiffres = (1,2,3,4,5,6,7,8,9,0);
@fruits = ('amande','fraise','cerise');
@alphabet = ('a'..'z');   Les deux points signifient de "tant à tant"
@a = ('a'); @nul = ();
@cartes = ('01'..'10','Valet','Dame','Roi');
```

on fait référence à un élément du tableau selon son indice par :

```
$chiffres[1] (=> 2)
$fruits[0]  (=> 'amande')
```

REMARQUE : En Perl (comme en C) les tableaux commencent à l'indice 0

On peut affecter un tableau à un autre tableau :

```
@ch = @chiffres;
@alphanum = (@alphabet, '_', @chiffres);
=> ('a','b',...,'z','_','1','2','3','4','5','6','7','8','9','0')
@ensemble = (@chiffres, 'datte', 'kiwi', 12.45);
```

Remarques :
On dispose d'un scalaire spécial : `$#tableau` qui indique le dernier indice du tableau (et donc sa taille - 1) : `$fruits[$#fruits]` (=> 'cerise')
Possibilité de référencer une partie d'un tableau

```
@cartes[6..$#cartes] => ('07','08','09','10','Valet','Dame','Roi')
@fruits[0..1] => ('amande', 'fraise')
```

D. Tableaux indicés (ou associatifs)

Ils sont toujours précédés du caractère % :

```
%prix = ('amande'=>30, 'fraise'=>9, 'cerise'=>25);
```

ou :

```
%prix = (amande=>30, fraise=>9, cerise=>25);
```

On référence ensuite un élément du tableau par : `$prix{'cerise'}` (=> 25)
(ou `$prix{cerise}`)

Exemples:
```
%chiffre = ();
$chiffre{'un'} = 1;   => ou $chiffre{un} = 1;
print $chiffre{'un'};
$var = 'un'; print $chiffre{$var};
```

E. Remarques

1. Perl5 autorise les combinaisons, comme un tableau de tableaux :

```
%saisons = (
'abricot'=>['été'],
'fraise'=> ['printemps','été'],
'pomme'=>  ['automne','hiver'],
'orange'=> ['hiver'],
'cerise'=> ['printemps'],
'amande'=> ['printemps','été','automne','hiver']);
```

ou

```
@departements = (
['Ain', 'Bourg-en-Bresse', 'Rhône-Alpes'],
['Aisne', 'Laon', 'Picardie'],
...
['Yonne', 'Auxerre', 'Bourgogne']);
```

Et l'on accédera à la région du Finistère, ou à la préfecture d'Ille et Vilaine par
`$departements[29 - 1][2], $departements[35 - 1][1]`
(On retranche un car l'indice des tableaux commence toujours à 0)

2. Pas de booléens (comme en langage C)

On utilise des entiers sachant que 0 est évalué comme étant faux (en fait il s'agit de la chaîne de caractère vide) et 1 comme étant vrai.

```
my $deux plus grand que un = (2 > 1);
if ($deux plus grand que un) {
     print "Ok !";
}
```

Le programme répondra Ok !

3. Tableaux à deux dimensions

On peut utiliser les tableaux indicés pour simuler des tableaux à 2 (ou n) dimensions :

```
%table_multiplication = ('1,1'=>1, '1,2'=>2, ...,
'9,8'=>72,'9,9'=>81);

$traduction{'amande','anglais'} = 'almond';
$traduction{'amande','italien'} = 'amoria';
$traduction{'cerise','anglais'} = 'cherry';
```

On peut également utiliser les tableaux de tableaux de Perl 5 pour le faire :

```
@table_multiplication = (
[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],   # Multiplié par 0
[ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9],   # Multiplié par 1
[ 0, 2, 4, 6, 8,10,12,14,16,18],   # Multiplié par 2
...
[ 0, 9,18,27,36,45,54,63,72,81]); # Multiplié par 9
```

On référencera alors 2*6 par $table_mult[6] [2]

Cf. Perldoc.com *perllol*

4. Références

On peut déclarer des références vers des objets (scalaires, tableaux …) :

Ex :

```
@fruits = ('banane', 'pomme');
$f = \@fruits; # $f est une référence vers le tableau "@fruits"
print $f->[1]; # Affichera "pomme"
$f->[1] = 'golden'; # Modifie le contenu du tableau "@fruits"
```

On peut aussi créer directement la référence et le tableau:

```
$f = ['banane', 'pomme'];
```

⇨ On utiliser les crochets au lieu des parenthèses

⇨ Ceci crée un tableau “anonyme” (il n’a pas de nom mais est référencé par “$f”)

De la même manière, on peut créer une référence vers un tableau indicé:

```
$p = {'banane'=>7, 'pomme'=>4};
```

⇨ On utiliser les accolades au lieu des parenthèses

⇨ Ceci crée un tableau indicé “anonyme”

Cf. Perldoc.com *perlreftut*

IV. Expressions

A. *Opérateurs*

De nombreux opérateurs existent en Perl (hérités la plupart du temps du langage C). Voici les principaux :

1. Opérateurs arithmétiques

```
$a = 1; $b = $a;      les variables a, et b auront pour valeur 1
$c = 53 + 5 - 2*4;    => 50
```

Plusieurs notations pour incrémenter une variable

`$a = $a + 1;` ou `$a += 1;` ou encore `$a++;` => addition

Même chose pour * (multiplication), - (soustraction), / (division), ** (exponentielle)

```
$a *= 3; $a /= 2; $a -= $b; ...
```

% : modulo (`17 % 3` =>2)

2. Chaînes de caractères

`.` concaténation

```
$c = 'ce' . 'rise';        (=> $c devient 'cerise')
$c .= 's';                 (=> $c devient 'cerises')
```

`x` réplique

```
$b = 'a' x 5; => 'aaaaa'
$b = 'jacqu' . 'adi' x 3; => 'jacquadiadiadi'
$b = 'assez ! '; $b x= 5; => 'assez ! assez ! assez ! assez ! assez ! assez !'
```

3. Parenthèses

Comme dans tous les langages de programmation, les parenthèses peuvent être utilisées dans les expressions :

```
$x = 2 * (56 - 78);
```

B. Comparaisons

1. de chiffres

Ce sont les opérateurs habituels :
>, >=, <, <=, ==, !=
respectivement: supérieur à, supérieur ou égal, inférieur à, inférieur ou égal, égalité, différent

Attention : = est une affectation, == est une comparaison
`if ($a = 2)` => sera toujours vrai !
Il aurait fallu écrire: `if ($a == 2)`

2. de chaînes

`gt, ge, lt, le, eq, ne`
respectivement: supérieur à (selon l'ordre alphabétique), supérieur ou égal, inférieur à, inférieur ou égal, égalité, différent

Attention ! Ne pas confondre la comparaison de chaînes et d'entiers
`'b' == 'a'` => évalué comme étant vrai !
il faut écrire :
`'b' eq 'a'` => évalué faux bien-sûr

3. de booléens

Même si le type booléen n'existe pas en tant que tel, des opérateurs existent :
`||` (ou inclusif), `&&` (et), `!` (négation)

```
(! 2 < 1)                  => vrai
(1 < 2) && (2 < 3)         => vrai
($a < 2) || ($a == 2)      équivaut à ($a <= 2)

(!$a && !$b)               équivaut à !($a || $b) (règle de Morgan !)
```

Remarque : depuis Perl5 une notation plus agréable existe :
`or` (au lieu de ||), `and` (au lieu de &&), `not` (au lieu de !)

```
if (not ($trop_cher or $trop_mur)) {print "J'achete !";}
```

cf. Perldoc.com *perlop* pour une description détaillée de tous les opérateurs.

V. Syntaxe générale

Chaque instruction doit être terminée par un point-virgule. Un passage à la ligne ne signifie pas une fin d'instruction (ce qui est souvent source d'erreurs au début).

```
my $a = 'a'
print $a;
```

il manque ";"

Ce programme est erroné !

```
my $a = 'Une très longue chaîne de caractère qui ne peut s'écrire
sur une seule ligne';
```

Ok !

Les commentaires commencent par un #
Tout le reste de la ligne est considéré comme un commentaire.

```
# voici un commentaire
my $a = 3;  # Et en voici un autre
```

Un bloc est un ensemble de commandes entourées par des crochets ({ }), chaque commande étant suivie d'un point-virgule.
La syntaxe générale de Perl est décrite dans Perldoc.com *perlsyn*

A. Soyons rigoureux

Il est fortement recommandé d'utiliser les modules « strict » et « warnings » : le fait de les inclure au début de programme vous oblige à déclarer toutes les variables que vous utilisez (à l'aide de « my »). C'est assez contraignant mais cela vous économisera des heures de « débogage ». En effet, Perl est parfois trop souple et vous laisse écrire des choses erronées, comme :

```
$mavariable=5;
print $Mavariable;
```

Le « m » est en majuscule dans la seconde ligne, et n'affichera donc pas « 5 ».
Par contre, si vous utilisez les deux modules

```
use strict; use warnings;
my $mavariable=5; # L'usage de "my" devient obligatoire
print $Mavariable;
```

vous aurez le message :
Global symbol "$Mavariable" requires explicit package name
qui, même s'il n'est pas très explicite, vous indique que « $Mavariable » pose problème

Chaque variable utilisée (scalaire, tableau…) doit maintenant être déclarée avant d'être utilisée. Cette déclaration doit être faite dans le même bloc. Voici quelques exemples :

```
use strict; use warnings;
my $mavariable='oui';
print $mavariable;
```

⇨ Ok

```
use strict; use warnings;
my $mavariable;
if (3>1) {$mavariable='oui';}
print $mavariable;
```

⇨ Ok

```
use strict; use warnings;
if (3>1) {my $mavariable='oui';}
print $mavariable;
```

⇨ Non : déclenche une erreur disant que « $mavariable » est utilisée sans être déclarée
⇨ On a déclaré « $mavariable » dans le bloc « if », elle n'est « visible » que dans ce bloc, pas à l'extérieur

B. Expressions conditionnelles

Syntaxe	Exemple

if (*expression*) {
bloc;
}
	`if ($prix{'datte'} > 20) {`
	`  print 'Les dattes sont un peu chères...';`
	`  print 'Achetons plutôt des cerises';`
	`}`

if (*expression*) {
bloc;
} else {
bloc2;
}
	`if ($fruit eq 'fraise') {`
	`  print 'parfait !';`
	`} else {`
	`  print 'Bof !';`
	`  print 'Je préfère sans pépin';`
	`}`

if (*exp1*) {
bloc1;
} elsif (*exp2*) {
bloc2;
} elsif (*exp3*) {
bloc3;
}
	`if (($fruit eq 'cerise')or($fruit eq 'fraise'))`
	`{`
	`  print 'rouge';`
	`} elsif ($fruit eq 'banane') {`
	`  print 'jaune';`
	`} elsif ($fruit eq 'kiwi') {`
	`  print 'vert';`
	`} else {`
	`  print 'je ne sais pas...';`
	`}`

Remarque : il existe une autre notation :
commande if (*condition*);

```
print 'Quand nous chanterons...' if ($fruit eq 'cerise');
```

Condition inversée :
unless (*condition) {*
bloc
};

Ou, selon la notation alternative:
commande unless (*condition*);

```
print 'Avec du sucre SVP...' if ($fruit eq 'fraise');
```

C. Boucles

1. « tant que »

Syntaxe	Exemple

while (*expression*) {
bloc;
}
	`my $mon argent = $porte monnaie;`
	`while ($mon argent > $prix{'cerise'}) {`
	`$mon argent -= $prix{'cerise'};`
	`# ou $mon_argent = $mon_argent - $prix{'cerise'}`
	`print "Et un kilo de cerises !";`
	`}`

Remarque:
Là aussi il existe une autre notation
commande while (*condition*)

```
print "Je compte : $i" while ($i++ < 10);
```

2. « répéter »

Syntaxe	Exemple

do {
bloc;
} while (*expression*);
	`# Recherche du premier fruit <= 10 F`
	`my $i = 0; my $f;`
	`do {`
	`$f = $fruits[$i];`
	`$i++;`
	`} while ($prix{$f} > 10);`
	`print "Je prend : $f";`

do {
bloc;
} until (expression);
	`my $i = 10;`
	`do {`
	`print $i;`
	`$i--;`
	`} until ($i == 0);`

Autre exemple : recherche du premier fruit par ordre alphabétique :

```
my $min = $fruits[0]; my $i=0; # $min contient le premier fruit
do {
     # Si on trouve un fruit moins grand (par ordre alphabétique)
     if ($fruits[$i] lt  $min) {
          $min = $fruits[$i]; # il devient le minimum
     }
     $i++;      # On passe au suivant
} until ($i > $#fruits);
```

3. « pour »

Syntaxe	Exemple

for (*init*;*condition*;*cmd*) {
bloc;
}
	`for (my $mon argent = $porte monnaie;`
	`  $mon argent > $prix{'cerise'};`
	`  $mon argent -= $prix{'cerise'}) {`
	`  print "Et un kilo de cerises !";`
	`}`

Plus souvent utilisé pour le parcours de tableaux :	

for ($i = 0;$i < *taille*; $i++) {
bloc;
}
	`for (my $i=0; $i <= $#fruit; $i++) {`
	`  print $fruit[$i];`
	`}`

Attention à bien mettre un point-virgule entre les différents éléments d'un « for »

4. « pour tout »

Syntaxe	Exemple

foreach *élément* (*tableau*) {
bloc;
}
	`foreach my $f (@fruits) {`
	`  print $f;`
	`}`

Très utilisé pour le parcours de tableaux

D. Procédures

1. déclaration

```
sub ma procedure {
     bloc;
}
```

Appel : `&ma_procedure();` ou, plus simplement: `ma_procedure;`

2. avec paramètre(s)

```
sub pepin {
     my($fruit) = @ ;

     if (($fruit ne 'amande') and ($fruit ne 'fraise')) {
          print "ce fruit a des pépins !";
     }
}
```

Appel: `&pepin('cerise');` ou: `pepin('cerise');`

3. fonctions

Une fonction est en fait une procédure qui retourne quelque chose :

```
sub pluriel {
     my($mot) = @ ;

     $m = $mot.'s';
     return($m);
}
```

Appel: `$mot_au_pluriel = &pluriel('cerise');` => 'cerises'

Remarque: Le passage de paramètres se fait donc à l'aide du tableau spécial @_ (géré par le système Perl). L'instruction « my » réalise une affectation dans des variables locales à la procédure avec les éléments du tableau. Ce type de passage de paramètre est très pratique car le nombre de paramètres n'est pas forcément fixe.

Variables locales :
Attention au petit piège habituel :

```
$m = 'Dupont';
$f = &pluriel('cerise');
print "M. $m vend des $f\n";
```

affichera : M. cerises vend des cerises !
A cause du fait qu'on utilise la variable $m qui est modifiée dans le programme pluriel ...
La solution consiste à déclarer toutes les variables utilisées dans les procédures en variables locales à l'aide de « my ». D'ou l'avantage d'utiliser les modules "strict" et "warning" (cf. paragraphe "Soyons rigoureux" p. 11).
A la troisième ligne de la procédure « pluriel » on rajoutera : `my $m;`

perlsub

VI. Fonctions prédéfinies

Quelques fonctions offertes par Perl pour manipuler les données. L'inventaire est loin d'être exhaustif (cf. Perldoc.com *perlfunc*)

A. Système

- print : permet d'afficher un message, ou le contenu de variables.

```
print 'bonjour';
print 'J\'ai acheté ', $nombre,' kilos de ', $fruit;
print;     => affiche le contenu de la variable spéciale $_
```

ou encore :

```
print "J'ai acheté $nombre kilos de ", &pluriel($fruit);
```

Problèmes courants : les caractères spéciaux (',",$,...), que l'on peut afficher en les précédant d'un « \ »

ex: `print "Il dit alors : \"Non, je n'ai pas 50 \$ !\"...";`

Autre solution pour les longs textes :

```
print <<"FIN"; #Tout ce qui suit jusqu'à FIN en début de ligne
Ceci est un long texte
qui n'est pas perturbé par la présence de "guillemets" ou
d'apostrophes
FIN
```

Quelques caractères spéciaux affichables avec « print » :
\n => « retour-chariot », \t => tabulation, \b => « bip »

(une fonction utile: `quotemeta($chaine)` permet de préfixer chaque caractère spécial par un "\")

- exit : permet d'arrêter le programme en cours

```
if ($erreur) {exit;}
```

- die : permet d'arrêter le programme en cours en affichant un message d'erreur.

```
if ($fruit eq 'orange') {die 'Je déteste les oranges !'}
```

- system : permet de lancer une commande système

```
system 'mkdir mon_repertoire';
```

- sleep *n* : le programme « dort » pendant *n* secondes

ex: programme « bip horaire »

```
while (1) {sleep 3600; print "\b";}
```

le fait d'écrire « while (1) » permet de faire une boucle infinie (on aurait pu écrire : « for (;;) »)

B. Mathématique

Les fonctions mathématiques habituelles existent aussi en Perl :
`sin`, `cos`, `tan`, `int` (partie entière d'un nombre), `sqrt`, `rand` (nombre aléatoire entre 0 et n), exp (exponentielle de n), `log`, `abs` (valeur absolue).

```
$s = cos(0);                => 1
$s = log(exp(1));           => 1
$i = int(sqrt(8));          => 2
$tirage_loto = int(rand(42)) + 1;
$i = abs(-5.6)              => 5.6
```

C. Chaînes de caractères

- chop(*ch*) Enlève le dernier caractère de la chaîne
  `$ch='cerises'; chop($ch);` => ch contient 'cerise'

- chomp(*ch*) Même chose que « chop » mais enlève uniquement un « retour-chariot » éventuel en fin de chaîne. Utilisé dans le parcours de fichiers (cf p. 22)

- length(*ch*) Retourne la longueur de la chaîne (nombre de caractères)
  `$l = length('cerise')` => 6

- uc(*ch*) Retourne la chaîne en majuscules (Perl 5)
  `$ch = uc('poire')` => *'POIRE'*

- lc(*ch*) Retourne la chaîne en minuscules (Perl 5)
  `$ch = lc('POIRE')` => 'poire'

- lcfirst(*ch*), ucfirst(*ch*) Retourne la chaîne avec simplement lepremier caractère en minuscule/majuscule (Perl 5)
  `$ch = ucfirst('la poire')` => 'La poire'

- split('*motif*', *ch*) Sépare la chaîne en plusieurs éléments (le séparateur étant *motif*). Le résultat est un tableau. (Fonction très utilisée pour l'analyse de fichiers)
  ```
  @t = split(' / ', 'amande / fraise / cerise');
  ```
  => ('amande','fraise', 'cerise')

- substr(*ch, indicedébut, longueur*)
  Retourne la chaîne de caractère contenue dans *ch*, du caractère *indicedébut* et de longueur *longueur*.
  ```
  $ch=substr('dupond', 0, 3)        => 'dup'
  $ch=substr('Les fruits', 4)       => 'fruits'
  ```

- index(*ch, recherche*) Retourne la position de *recherche* dans la chaîne *ch*
  `$i=index('Le temps des cerises','cerise');` => 13

Remarques:

Par défaut la plupart de ces fonctions travaillent sur la variable spéciale $_

```
$_ = 'amandes'; chop; print;
```
=> Affichera 'amande'

Si votre environnement est bien configuré, (cf. Perldoc.com *perllocale*) tout ces fonctions prennent en compte les caractères internationaux :

```
print uc(substr('euro ou ●●●●',8));
```
=> Affichera ' EYPΩ'

D. tableaux, listes

- grep(*/expression/*, *tableau*) Recherche d'une expression dans un tableau

```
if (grep(/poivron/, @fruits));
```
=> faux

```
if (grep(/$f/, @fruits) {print 'fruit connu';}
```

grep retourne un tableau des éléments trouvés :

```
@f = grep(/ise$/, @fruits);
```
=> fraise;cerise

- join(*ch*, *tableau*) Regroupe tous les éléments d'un tableau dans une chaîne de caractères (en spécifiant le séparateur)

```
print join(', ', @fruits);
```
=> affiche 'amande, fraise, cerise'

- pop (*tableau*) Retourne le dernier élément du tableau (et l'enlève)

```
print pop(@fruits);
```
=> affiche 'cerise', @fruits devient ('amande','fraise')

- push (*tableau*, *element*) Ajoute un élément en fin de tableau (contraire de pop)

```
push(@fruits, 'abricot');
```
=> @fruits devient ('amande','fraise','abricot')

- shift(*tableau*) Retourne le premier élément du tableau (et l'enlève)

```
print shift(@fruits)
```
=> Affiche 'amande', @fruits devient ('fraise','abricot')

- unshift (*tableau*, *element*) Ajoute un élément en début de tableau

```
unshift ('coing', @fruits);
```
=> @fruits devient ('coing', 'fraise','abricot')

- sort (*tableau*) Tri le tableau par ordre croissant

```
@fruits = sort (@fruits);
```
=> @fruits devient ('abricot', 'coing', 'fraise')

- reverse (*tableau*) Inverse le tableau

```
@fruits = reverse(@fruits);
```
=> @fruits devient ('fraise', 'coing', 'abricot')

- splice (*tableau, début, nb*) Enlève *nb* éléments du tableau à partir de l'indice *début*

```
@derniers = splice(@fruits, 1,2);
```

=> @derniers devient ('coing', 'abricot') @fruits devient ('fraise')

On peut éventuellement remplacer les éléments supprimés :

```
@fruits=('fraise','pomme');
splice(@fruits, 1,1, ('elstar','golden'));
```

=> @fruits contient ('fraise', 'elstar','golden')

Exemple : Tirage des 7 chiffres du loto

```
my @c = (1..42); my $i=0;
print splice(@c, int(rand($#c+1)),1),"\n" while ($i++ < 7);
```

On enlève 7 fois un élément (pris au hasard) du tableau des 42 chiffres.

Pour information, l'exemple précédent aurait pu s'écrire :

```
my @c = (1..42); my $i=0;
while ($i < 7) {
     my $nb = int(rand($#c + 1));
     print "$nb\n";
     splice (@c, $nb, 1);
     $i++;
}
```

E. tableaux indicés

- each(*tabi*) *Les couples clé/valeurs d'un tableau indicé*

```
while (my ($fruit, $valeur) = each(%prix)) {
     print "kilo de $fruit : $valeur F";
}
```

L'utilisation habituelle est d'afficher le contenu d'un tableau
Attention : Les clés d'un tableau associatif ne sont pas triées !
ex:

```
my %t=("bernard"=>45,"albert"=>32, "raymond"=>2);
while (my ($n, $s) = each(%t)) {print "$n,$s\n";}
```

affichera :
raymond,2
albert,32
bernard,45

- values(*tabi*) *Toutes les valeurs d'un tableau indicé (sous la forme d'un tableau)*

```
print 'les prix:', join(', ', values(%prix));
```

- keys(*tabi*) *Toutes les "clés" d'un tableau indicé*

```
print 'les fruits:', join(', ', keys(%prix));
```

- exists(*élément*) *Indique si un élément a été défini*

```
if (exists $prix{'kiwi'}) {
     print $prix{'kiwi'};
} else {
     print 'Je ne connais pas le prix du kiwi !';
}
```

- delete(*élément*) *Supprimer un élément*

```
delete $prix{'cerise'};
```

Remarque:
Il n'existe pas de fonction permettant d'ajouter un élément dans un tableau indicé (comme le *push* des tableaux normaux) car il suffit d'écrire :
$tab{*nouvel-élément*} = *nouvelle-valeur*;

VII. Gestion de fichiers

A. Ouverture

L'ouverture consiste (le plus souvent) à associer un descripteur de fichier (filehandle) à un fichier physique.

1. en lecture

```
open (FENT, '< fichier');
```
ouverture d'un fichier, référencé ensuite par `FENT`.

Le caractère « < » est facultatif mais recommandé

```
open(FENT, 'fruits.txt');
```

Depuis perl 5.8, on peut ouvrir un fichier en spécifiant l'encodage utilisé :

```
open(FENT, '<:encoding(iso-8859-1)', 'fruits.txt');
open(FENT, '<:encoding(iso-8859-7)', 'texteGrec.txt');
open(FENT, '<:utf8', 'texteUnicode.txt');
```

```
open (COM, 'commande|');
```
ouverture d'une commande dont le résultat sera dans `COM`

```
open (FDESS, 'ls /users/dess |'); (Unix)
open (Ftmp, 'dir c:\temp |'); (DOS)
```

Un fichier spécial : STDIN, le clavier (entrée standard).

2. en écriture

```
open (FSOR, '> fichier');
```
Ecriture du fichier, si ce fichier existait auparavant : l'ancien contenu est écrasé.

```
open(FSOR, '> etat.txt');
```

```
open (FSOR,'>>fichier');
```
Ecriture à la fin du fichier, Le fichier est créé si besoin

```
open (FSOR, '>> liste.txt');
open (FSOR, '| commande');
```
Le fichier de sortie sera en fait l'entrée standard de la commande

```
open(FTRIE, '| lpr');
```
=> La sortie sera imprimée

```
open (FMAIL, '| mail -s "Bonjour" lim\@univ-rennes1.fr');
```

Deux fichiers spéciaux : STDOUT, STDERR (respectivement: sortie standard, et sortie erreur), par défaut l'écran.

3. Gestion des erreurs (| |)

Lorsque l'on ouvre un fichier il se peut qu'il y ait une erreur.
En lecture : le fichier n'existe pas, ou ne peut pas être lu (droits d'accès)...
En écriture : Le fichier existe mais on n'a pas le droit d'écrire dessus, pour une commande Unix : la commande est inconnue...

Il faut prendre l'habitude, quand on ouvre un fichier, de détecter l'erreur éventuelle.
On peut le faire sous la forme suivante : (dérivée du C)

```
if (! open (F, ...)) {
```

```
    die "Problème à l'ouverture du fichier";
}
```

Ou sous la forme plus simple et plus usitée en Perl :

```
open (F, ...) || die "Pb d'ouverture";
```

On peut, et c'est même conseillé, récupérer le texte de l'erreur contenu dans la variable $!

```
open (F, ...) || die "Pb d'ouverture : $!";
```

Cf. Perldoc.com *perlopentut* (tutoriel sur l'ouverture de fichiers)

B. Fermeture

```
Commande close
close FENT; close FSOR;
```

C. Lecture

```
$ligne = <FENT>;
```

ex: `$reponse = <STDIN>;` => lecture d'une ligne à l'écran

Remarque :

- La fin de ligne (retour-chariot) est lue également. Pour enlever cette fin de ligne il suffit d'utiliser la commande `chop`, ou son équivalent : `chomp` (enlève le dernier caractère uniquement si c'est un retour-chariot)
- On peut lire toutes les lignes d'un fichier dans un tableau (en une seule instruction)

```
    @lignes = <FENT>;
```

D. Ecriture

print Fsor *ce-que-je-veux*;

```
print FSOR 'DUPONT Jean';
print FMAIL 'Comment fait-on pour se connecter SVP ?';
print STDOUT "Bonjour\n";
print STDERR 'Je déteste les oranges !';
```

E. Parcours

Se fait d'une manière intuitive :

```
open (F, $fichier) || die "Problème pour ouvrir $fichier: $!";
while (my $ligne = <F>) {
    print $ligne;
}
close F;
```

L'instruction « $ligne=<F> » retourne toujours quelque chose sauf à la fin du fichier, la condition devient alors fausse.

F. Fichier spécial : <>

Perl offre une fonctionnalité bien pratique : l'utilisation d'un fichier spécial en lecture qui contiendra ce qui est lu en entrée standard.

Exemple : le petit programme du début peut s'écrire

```
#!/bin/perl
my $i=0;
while (my $ligne = <>) {
     $i++;
}
print "Nombre de lignes : $i";
```

Remarque: Lorsque, dans une boucle « while », on ne spécifie pas dans quelle variable on lit le fichier : la ligne lue se trouvera dans la variale spéciale « `$_` ».

Ainsi ce programme demandera d'écrire quelques lignes et affichera le nombre de lignes qu'il a lues. Mais il permet également de prendre un ou plusieurs fichiers en paramètre.
(voir chapitre Passage de paramètres au programme page 30)

G. Autre notation

Une autre notation est possible pour désigner les « filehandle »
Exemple : Lecture d'un fichier en iso-latin et écriture dans un fichier Unicode (utf8)

```
#!/bin/perl
open (my $Fin, "<:encoding(iso-
8859-1)", "texte latin.txt")
  || die "problème lecture: $!";
open (my $Fout, ">:utf8",
"texte unicode.txt")
  || die "problème écriture: $!";
 while (my $ligne = <$Fin>) {
     print {$Fout} $ligne;
}
close $Fin; close $Fout;
```

VIII. Expressions régulières

Comme la commande "egrep" d'Unix, Perl offre la même puissance de manipulation d'expressions régulières.
(pour une introduction : cf. Perldoc.com *perlrequick*)
On utilise l'opérateur conditionnel =~ qui signifie "ressemble à".

Syntaxe: *chaîne* =~/*expression*/

Exemple: `if ($nom =~ /^[Dd]upon/) {print "OK !";}`
=> *Ok si nom est 'dupont', 'dupond', 'Dupont-Lassoeur'*
^ *signifie « commence par »*
On peut rajouter « i » derrière l'expression pour signifier qu'on ne différencie pas les majuscules des minuscules.

Le contraire de l'opérateur `=~` est `!~` (ne ressemble pas à ...)
`if ($nom !~ /^dupon/i) {print "Non...";}`

A. Rappel

Une expression régulière (appelée aussi « regexp ») est un *motif* de recherche, constitué de :

- Un caractère
- Un ensemble de caractères :
  - [a-z] tout caractère alphabétique[2]
  - [aeiouy] toute voyelle
  - `.` tout caractère sauf fin de ligne
  - [a-zA-Z0-9] tout caractère alphanumérique
  - le caractère ^ au début d'un ensemble signifie « tout sauf »
    ([^0-9] : tout caractère non numérique)
  - Un caractère spécial: \n, « retour-chariot », \t : tabulation, ^ : début de ligne, $: fin de ligne
  - \w signifie « un mot », \s signifie « un espace », \W « tout sauf un mot », \S « tout sauf un espace »
- Quelques opérateurs : « ? » 0 ou 1 fois, « * » : 0 ou n fois, « + » : 1 ou n fois, « | » : ou (inclusif)
  Il y a même l'opérateur {*n*,*m*} qui signifie de *n* à *m* fois.

Remarque : Si l'on veut qu'un caractère spécial apparaisse tel quel, il faut le précéder d'un « anti-slash » (\), les caractères spéciaux sont :
« ^ | ( ) [ ] { } \ / $ + * ? . »
ex: `if ($chaine =~ /\(.*\)/) {print $chaine;}`
qui affiche la chaîne de caractères si elle contient des parenthèses

[2] sauf accents, depuis Perl version 5.8 utiliser plutôt `[[:alpha:]]`

Attention :
le caractère spécial « ^ » a deux significations différentes :
1) dans un ensemble il signifie « tout sauf »
2) en dehors il signifie « commence par »

1. Exemples d'utilisation :

Le traitement d'une réponse de type (Oui/Non) : en fait on teste que la réponse commence par "O"

```
print 'Etes-vous d\'accord ? ';
my $reponse = <STDIN>;
if ($reponse =~ /^O/i) { # commence par O (minus. ou majus.)
     print "Alors on continue";
}
```

Remarque: Le petit piège dans ce programme serait de mettre `$reponse =~ /O/` (sans le « ^ ») qui serait reconnu dans 'NON' (qui contient un « O »)

Recherche des numéros de téléphone dans un fichier :

```
while (my $ligne = <>) { # Tant qu'on lit une ligne
     if ($ligne =~ /([0-9][0-9]\.)+[0-9][0-9]/) {
          # si la ligne est sous la forme 02.99.45...
          print $ligne;  # Alors on l'affiche
     }
}
```

Remarque:
Par défaut l'expression régulière est appliquée à la variable $_.
Donc écrire `if ($_ =~ /exp/)` est équivalent à écrire : `if (/exp/)`

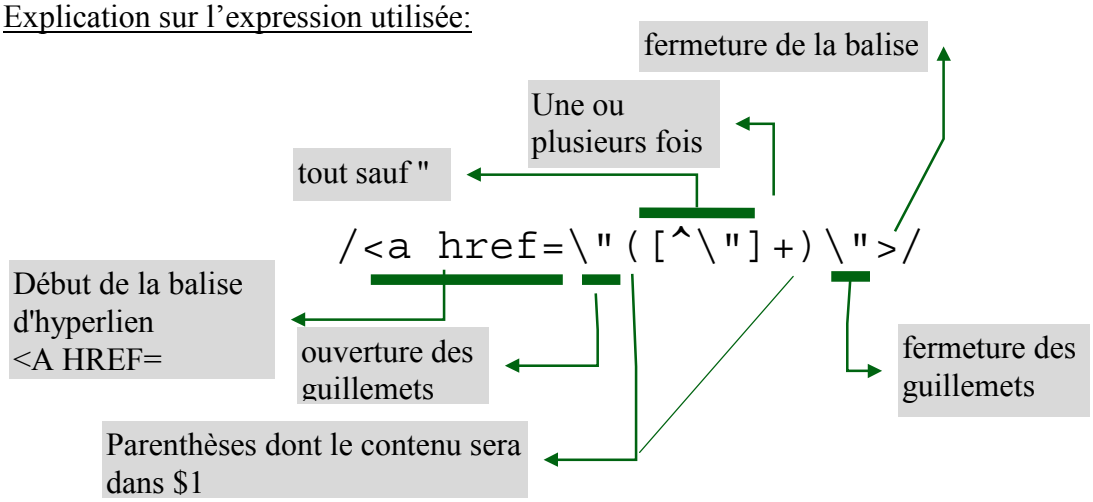
2. Utilisation avancée :

Le contenu des parenthèses d'une expression régulière est retourné sous la forme d'un tableau.

Exemple: Lister tous les hyperliens d'un document HTML :
un hyperlien est noté sous la forme <a href="*quelquechose*"> (avec *quelquechose* ne pouvant pas contenir le caractère ". Donc *quelquechose* sera reconnu par `[^\"]+` qui signifie tout caractère autre que ".

```
while (my $ligne = <>) {
     if ( my ($h) = ($ligne =~ /<a href=\"([^\"]+)\">/)) {
          print "Hyperlien: $h\n";
     } # ce programme ne detecte qu'un hyperlien par ligne
}
```

Explication sur l'expression utilisée:



On peut utiliser cette fonctionnalité pour faire des affectations :

exemple :
On a un fichier annuaire de la forme :

M. Dupont tel: 02.99.27.82.67
M. Durant tel: 02.99.34.54.56
...
Mme Larivoisière 02.99.23.43.21

Et on souhaiterait avoir un programme automatique qui soit capable de retrouver un numéro de téléphone d'après un nom (saisi à l'écran).

```
print " entrez un nom : ";
my $nom = <STDIN>; # on lit ce que « tape » l'utilisateur
chomp($nom); # Enlève le « retour-chariot » en fin de chaîne
open (ANNUAIRE, '<mon-annuaire.txt') ||die "Problème ouverture ";
while (my $l = <ANNUAIRE>) {
   if (my ($tel) = ($l =~ /$nom.*(([0-9][0-9]\.)+[0-9][0-9])/)) {
     print "Le numéro de téléphone de $nom est : $tel\n";
   }
}
close ANNUAIRE;
```

B. Remplacement

Comme le fait la commande « sed » en Unix, Perl permet de faire des remplacements sur une chaîne de caractère, en utilisant la syntaxe :
$chaîne =~ s/*motif*/*remplacement*/;
où *motif* est une expression régulière et *remplacement* ce qui remplace.

Exemples:
`$fruit =~ s/e$/es/;` remplace un « e » final par « es »
`$tel =~ s/^02\.99\./00\.33\.2\.99\./;` remplace les numéros de téléphone par leur équivalent international (numéros d'Ille et Vilaine uniquement)

On peux référencer une partie du motif dans le remplacement avec $1 ($1 est une variable spéciale : le contenu de la première parenthèse).
exemple :

Remplacement de TOUS les numéros de téléphone français
```
$tel =~ s/^0([1-4])\.([0-9][0-9])\./00\.33\.$1\.$2\./;
```

Transformer automatiquement les noms d'arbre par « arbre à *fruit* »
```
$texte =~ s/([a-z]+)ier /arbre à $1es /;
```
'cerisier' sera traduit par 'arbre à cerises' (contenu de $1 => 'ceris')
'manguier' => 'arbre à mangues' ...

Les options :
`s/exp/remp/i;` => Indifférenciation minuscules/majuscules
`s/exp/remp/g;` => Remplace toute occurrence (pas seulement la première)

Pour remplacer un texte par le même en majuscule (\U):
```
s/([a-z])/\U$1/g;
```

Exemple: mettre toutes les balises d'un fichier HTML en majuscule (en fait on met le premier mot de la balise en majuscule). Ne pas oublier les fins de balises (commençant par « / »)
```
while (my $ligne = <>) {
     $ligne =~ s/\<(\/?[a-z]+)/\<\U$1/g;
     # \U$1 signifie $1 en majuscules
     print $ligne;
}
```

Si on appelle ce programme avec le fichier HTML suivant :
```
<html><body bgcolor="ffffff">
<a href="http://www.cnam.fr/">bonjour</a>
</body></html>
```

Il affichera :
```
<HTML><BODY bgcolor="ffffff">
<A href="http://www.cnam.fr/">bonjour</A>
</BODY></HTML>
```

IX. Variables et tableaux spéciaux

Petit récapitulatif des variables spéciales, ce sont les variables sous la forme $*c* (avec *c* un caractère non alphabétique):

$_ La dernière ligne lue (au sein d'une boucle « while »)

$! La dernière erreur, utilisée dans les détections d'erreurs

```
open(F,'fichier') || die "erreur $!"
```

$$ Le numéro système du programme en cours: parfois utile car il change à chaque fois

$1, $2, ... le contenu de la parenthèse numéro *n* dans la dernière expression régulière

$0 Le nom du programme (à utiliser, cela vous évitera de modifier le programme s'il change de nom)

Tableaux spéciaux

@_ qui contient les paramètres passés à une procédure.

@ARGV qui contient les paramètres passés au programme

%ENV tableau indicé contenant toutes les variables d'environnement

@INC tous les répertoires Perl contenant les « librairies » (rarement utilisé)

Cf. Perldoc.com *perlvar*

X. Structures complexes

En Perl (version 5), on utilise la syntaxe des tableaux indicés pour désigner une structure complexe, exemple:

```
$patient->{nom} = 'Dupont';
$patient->{prenom} = 'Albert';
$patient->{age} = 24;
```

On peut initialiser la structure comme lorsqu'on initialise un tableau indicé :

```
$patient = {
nom => 'Dupont',
prenom => 'Albert',
age => 24};
```

On utilise les accolades plutôt que les parenthèses (référence vers un tableau indicé)

Pour afficher le contenu d'une structure complexe :

```
print "Nom:$patient->{nom} $patient->{prenom},
  âge:$patient->{age}";
```

ou:

```
foreach my $champ ('nom','prenom','age') {
     print "$champ : $patient->{$champ}\n";
}
```

qui affiche:
nom : Dupont
prenom : Albert
age : 24

ou encore :

Pour référencer $patient comme un tableau indicé on utilise cette syntaxe

```
foreach my $champ (keys %$patient) {
     print "$champ : $patient->{$champ} ";
}
```

qui affiche:
prenom : Albert
nom : Dupont
age : 24

Les champs apparaissent dans le "désordre" comme dans l'affichage d'un tableau indicé

XI. Exemples

A. Passage de paramètres au programme

Sur Unix (ou Dos) on peut appeler un programme Perl en lui donnant des paramètres, comme on le fait pour les commandes Unix

Les paramètres sont stockés dans un tableau spécial : @ARGV
Le premier paramètre est donc $ARGV[0]

exemple: bienvenue.pl

```
#!/bin/perl
use strict; use warnings;
my $email = $ARGV[0];
open (MAIL, "| mail -s 'Bonjour' $email") # commande unix mail
     || die "Problème : $!";
print MAIL "Bonjour très cher\n";
print MAIL "Nous sommes très heureux de vous envoyer un mail\n"
print MAIL " A bientôt\n"
close MAIL;
```

L'appel (sous Unix) se fera :
`bienvenue.pl durand@univ-rennes1.fr`
sur DOS : `perl bienvenue.pl durand@univ-rennes1.fr`

Si on veut traiter plusieurs paramètres il suffit de consulter le tableau @ARGV :
ex: noel.pl

```
#!/bin/perl
use strict; use warnings;
foreach my $email (@ARGV) {
     open (MAIL, "| mail -s '25 decembre' $email")
          || die "Problème : $!";
     print MAIL "Joyeux noël\n";
     close MAIL;
}
```

Programme de recherche d'un motif dans plusieurs fichiers, avec affichage du nom du fichier
ex: grep.pl

```
#!/bin/perl
# recherche un motif dans un ou plusieurs fichiers.
# Le motif est demandé à l'utilisateur
use strict; use warnings;
print "Entrez votre motif de recherche\n";
my $motif = <STDIN>;
chomp($motif); # Enlever le retour-chariot
foreach my $f (@ARGV) {
     open(F, "<$f") || die "Impossible de lire le fichier $f : $!";
     while (my $ligne = <F>) {
        if ($ligne =~ /$motif/) { # Si la ligne y ressemble
          print "On l'a trouvé dans le fichier $f : $ligne";
        }
     }
     close F;
}
```

Remarque : On pourra utiliser ce dernier exemple sur Macintosh, on fera alors glisser les fichiers sur le programme Perl

B. Parcours de fichier

Une des fonctionnalités de Perl est la manipulation de fichiers. Le parcours le plus simple, on l'a vu, est le suivant :

```
#!/bin/perl
while (my $ligne = <>) {
     print $ligne;
}
```

Qui se contente de lire le contenu du (ou des) fichier(s) passé(s) en paramètre pour l'afficher à l'écran.

On peut avoir envie d'écrire le résultat d'une modification dans un fichier, par exemple, enlever toutes les balises HTML et les écrire dans un fichier "sansbalises.txt" (ouvert donc en écriture).

```
#!/bin/perl
use strict; use warnings;
open (ST, '> sansbalises.txt') ||die "Impossible d'écrire: $!";
while (my $ligne = <>) {
     $ligne =~ s/<[^>]+>//g;# On remplace <...> par rien du tout
     print ST $ligne;    # Le résultat est écrit dans le fichier
}
close ST;
```

Par contre le traitement serait relativement complexe pour traiter tous les fichiers un par un, et, à chaque fois, de lui enlever les balises, Perl a une option (-i) qui permet de renommer le fichier d'origine et le résultat de la transformation sera écrit dans le fichier lui-même.
Cela parait compliqué mais c'est bien utile pour faire des transformations automatique sur plusieurs fichiers en même temps.
Cette option (-i*extension*) se met dans la première ligne :

```
#!/bin/perl -i.avecbalises
while (my $ligne = <>) {
     $ligne =~ s/<[^>]+>//g;
     print $ligne;  # Affichage dans un fichier, pas à l'écran
}
```

Ainsi on peut lui donner en paramètre tous les fichiers HTML d'un répertoire, ils seront tous renommés sous la forme *fichier.HTML*.avecbalises, et le fichier HTML de départ sera le résultat du programme.

Remarque :
Sous Unix, ou DOS, pour tester une petite commande Perl, on peut appeler Perl sans créer de programme avec la syntaxe :
noemed% `perl -e 'commande'`

Ou même on peut l'utiliser pour appliquer une commande à chaque ligne :
`ls | perl -n -e 'print if (! /\.tmp$/);'` (Unix, sous DOS utiliser "dir")
=> affiche tous les fichiers se trouvant sous le répertoire courant sauf les temporaires
=> Quand on utilise les option « n » et « e », chaque ligne lue se trouve dans la variable spéciale $_

l'option -e permet de prendre une commande Perl en paramètre
l'option -n permet d'appliquer la commande à chaque ligne lue
l'option –i*ext* permet de renommer un fichier avec l'extension *ext* avant d'appliquer des traitements

Exemple : Supprimer en une seule commande toutes les balises HTML de plusieurs fichiers :

```
perl -n -i.sansbalises -e 's/<[^>]+>//g; print;'
```

Car l'option -i permet de renommer chaque fichier (avec l'extension donnée), ensuite, chaque fichier est lu l'un à la suite de l'autre, tout ce qui est affiché est écrit dans chacun des fichiers. Notons que l'option « -p » remplace avantageusement l'option précédente, elle a le même effet que « -n » sauf qu'elle fait automatiquement le « print ».

Autre exemple : Enlever toutes les lignes des fichiers qui comporte la chaîne de caractère « mot de passe »

```
perl -n -i.sanspw -e 'print unless (/mot de passe/);'
```

Cf. Perldoc.com *perlrun*

C. Programmation objet

Remarque : Ce paragraphe a été écrit uniquement pour avoir un exemple de programmation objet avec Perl : il ne sera pas développé dans le cadre de ce cours.

Pour ceux qui connaissent la programmation objet, Perl (version 5) permet de définir des objets.

En résumé on associe à un « objet » des données et des « méthodes ». Ce qui simplifie l'utilisation ensuite de ces objets.

En Perl objet : un « objet » est une référence, une « méthode » est une procédure, une « classe » d'objet est un package.

En pratique :
On crée un fichier Perl (avec le suffixe .pm) qui contient la classe d'objet que l'on veut définir

Deux méthodes particulières :
new, constructeur, appelée automatiquement à la création de l'objet
et **DESTROY**, destructeur, appelée automatiquement à la destruction de l'objet

Notes:
⇒ Il est, encore une fois, préférable d'utiliser « strict »
⇒ La fonction `bless` permet de rendre un objet « visible » de l'extérieur.

ex: maclasse.pm

```
#!/bin/perl
use strict;   # Pour etre bien sur de ne pas faire d'erreurs
package maclasse;   # Le nom que l'on donnera à l'objet

sub new {      # constructeur, Méthode appelée à la création
  my ($classe, ...) = @ ;# La classe est toujours le 1er paramètre
  ...
  return bless référence, $class;
  # on ne retourne pas une variable, mais sa référence
}

sub DESTROY {  # destructeur, appelée à la destruction
     my ($objet) = @ ; # L'objet est toujours le 1er paramètre
     ...
}

sub methode1 { # On définit ainsi toutes les méthodes
     my ($objet, ...) = @ ; # tjs l'objet en 1er paramètre
     ...
}
```

La classe de l'objet étant définie, on peut l'utiliser maintenant dans un programme Perl de la manière suivante :

```
use maclasse;
my $mon_objet = new maclasse(paramètres); # Appel du constructeur
```

```
$mon_objet->methode1(paramètres);  # Appel d'une méthode

exit;      # fin du programme, appel du destructeur de l'objet
```

cf. Perldoc.com *perlobj, perlboot*

Comme exemple de programmation Objet nous allons prendre un objet « patient » qui sera défini par son nom et l'unité médicale dans laquelle il se trouve.

Deux méthodes seront attribuées au patient :

`transfert` -> Transfert vers une nouvelle unité médicale

`affiche` -> Affiche le nom du patient ainsi que l'unité médicale dans laquelle il se trouve

ex : patient.pm

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

package patient;                # Déclaration d'un package

# Un nouveau patient consiste à lui donner un nom
# et éventuellement une unité
sub new {      # Constructeur
     my ($class, $nom, $unite) = @ ;
     my $patient = {}; # Structure complexe

     $patient->{nom} = $nom;
     if (defined $unite) { # Si unité définie
          $patient->{unite} = $unite;
     } else {
          $patient->{unite} = 'Non définie';
     }
     return bless $patient, $class;
}

sub transfert { # Transfert du patient vers nouvelle unité
     my ($patient, $nvunite) = @ ;

     $patient->{unite} = $nvunite;
}

sub affiche { # Affichage de la situation du patient
     my ($patient) = @ ;

     print "Le patient $patient->{nom} est dans l'unité
$patient->{unite}\n";
}

sub DESTROY {
     my ($patient) = @ ;

     print "Le patient $patient->{nom} est parti !\n";
}
1; # Une classe d'objet se termine toujours par 1;
```

On peut maintenant utiliser cette classe d'objet dans un programme Perl :

```
#!/bin/perl
use strict; use warnings;
use patient; # On utilise le module patient

# Déclaration de deux nouveaux patients
# (deux objets de la classe patient)

my $patient1 = new patient('Dupont Pierre');
my $patient2 = new patient('Durand Albert', 'urgences');

$patient1->affiche;     # Appel d'une méthode pour patient
$patient2->affiche;     # (affichage de sa situation)

$patient1->transfert('cardio');    # transfert vers nelle unite
$patient2->transfert('pneumo');

$patient1->affiche;# Affichage de la situation des 2 patients
$patient2->affiche;
# fin du programme, appel des destructeurs
```

Le résultat de ce programme sera :

Le patient Dupont Pierre est dans l'unité Non définie
Le patient Durand Albert est dans l'unité urgences
Le patient Dupont Pierre est dans l'unité cardio
Le patient Durand Albert est dans l'unité pneumo
Le patient Durand Albert est parti !
Le patient Dupont Pierre est parti !

D. Perl et CGI

Rappel: Les CGI (common gateway interface), sont les programmes exécutés sur un serveur Web. Ce sont des programmes qui s'exécutent sur le serveur Web et qui produisent (le plus souvent) de l'HTML.

Remarque: Un CGI est un programme (ici écrit en Perl) qui doit afficher un « header » (ici « Content-type: text/html » suivit d'une ligne vide), et qui ensuite affiche du HTML.

Une alternative aux CGI : Les ASP du serveur Apache, http://www.apache-asp.org/
C'est une nouvelle technologie (encore en cours de développement) qui permet d' « embarquer » des instructions Perl dans un code HTML
Exemple : Affiche 10 fois « Salut »

```
<html><head><title>Bonjour</title>
</head><body>
<% foreach my $i (1..6) { %>
<h2>Salut</h2>
<%} %>
```

1. Premier CGI: sans paramètres

Exemple de petit programme CGI :
Il s'agit d'un programme qui affiche les personnes du DESS (commande UNIX ls /users/DESS).

```
#!/bin/perl
print "Content-type: text/html\n\n"; # Header indispensable

# Corps du programme
open(LISTE, 'ls /users/DESS |')
    || die "Impossible d'executer la commande ls: $!";
print "Liste du DESS: <UL>";
while (my $nom = <LISTE>) {
    print "<LI>$nom</LI>";
}
print "</UL>";
```

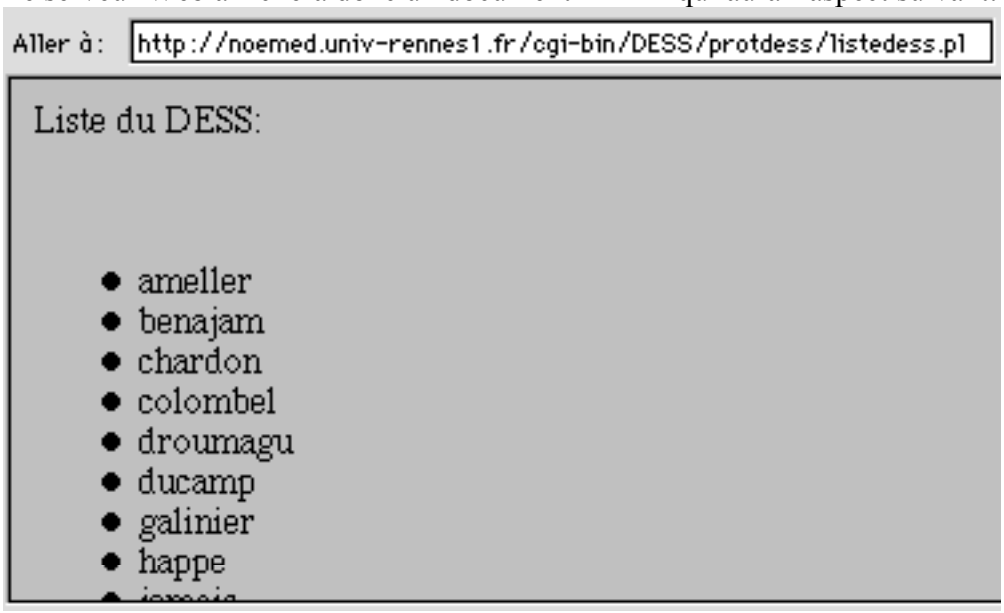
Ce programme affiche un fichier HTML qui ressemble à :

```
Content-type: text/html

Liste du DESS: <UL>
<LI>ameller< /LI>
<LI>benajam</LI >
...
<LI>vannier</LI>
</UL>
```

Il s'agit d'un « Header » qui ne sert que pour le serveur Web

Le reste est du HTML

Le serveur Web affichera donc un document HTML qui aura l'aspect suivant:



2. Deuxième CGI: associé à un formulaire

Le CGI précédent est un cas particulier relativement rare... En général un programme CGI est **paramétrable**. C'est à dire que l'utilisateur peut entrer des données qui seront analysées par le CGI. Ce qui, sous Web, sous-entend l'utilisation de formulaires.

Le cas général est donc : Un formulaire pour entrer les données, et un CGI pour les exploiter. 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/*programme*.pl) et sera accessible avec l'URL correspondante (en général http://*serveur*/cgi-bin/*programme*.pl).

Exemple: Un petit CGI poli, bonjour.pl

```
#!/bin/perl
use CGI Lite;                    # Utilisation d'un module CGI
my $cgi=new CGI Lite;               # Nouveau CGI
my %in = $cgi->parse form data;     # Lecture des parametres
print "Content-type: text/html\n\n"; # Header indispensable
# Corps du programme
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 le tableau associatif %in. (cf. paragraphe XIII.B « Installation de modules »).
Il reste ensuite à écrire un formulaire qui remplit les données dont on a besoin (pour l'exemple seul le paramètre « nom » est utilisé).

bonjour.html

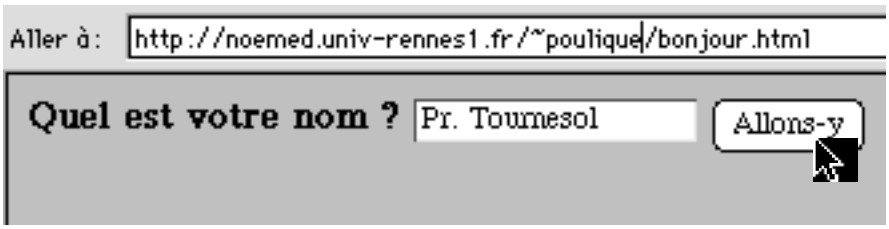
```
<form action="http://noemed.univ-rennes1.fr/cgi-bin/DESS/protdess/bonjour.pl"
METHOD=POST>
<b>Quel est votre nom ?</b>
<input type=text name=nom>
<input type=submit value="Allons-y">
</form>
```

C'est ici que l'on nomme le CGI que l'on appelle

Il faut que le nom de la variable corresponde à celui utilisé dans le CGI ($in{'nom'})

Le bouton « submit » est indispensable

Ce formulaire HTML sous un navigateur Web :



qui appellera le CGI bonjour.pl avec le paramètre « nom » ayant pour valeur « Pr. Tournesol »

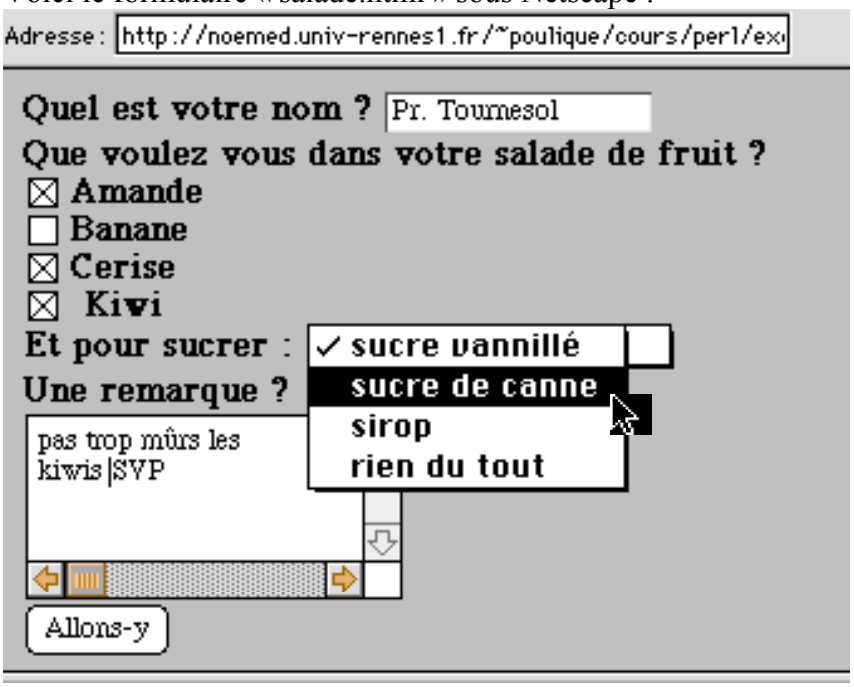
Le CGI répondra alors:

cf. Perldoc.com Module CGI_Lite

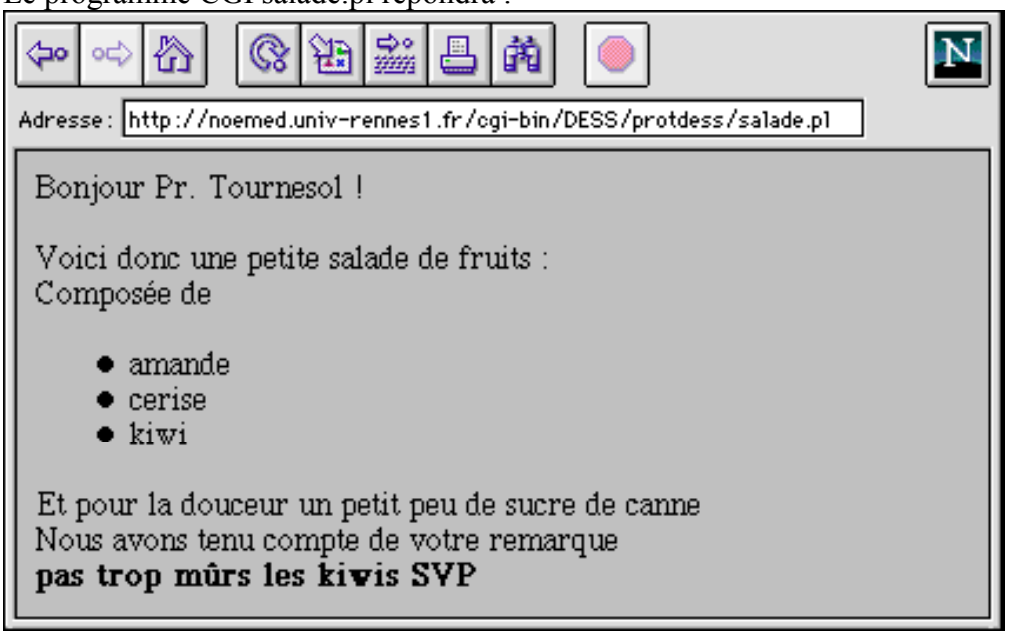
3. Un exemple un peu plus complexe

Utilisation de différentes balises de formulaire pour saisir des informations.
Un problème se pose avec les balises 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 valeur du champ.
On va créer un CGI qui « crée » une salade de fruits avec les paramètres de l'utilisateur.

Voici le formulaire « salade.html » sous Netscape :



Le programme CGI salade.pl répondra :



Source du formulaire: salade.html

```
<form action="http:/noemed.univ-rennes1.fr/cgi-bin/DESS/protdess/salade.pl"
METHOD=POST>
<b>Quel est votre nom ?</b>
<input type=text name=nom><br>
<b>Que voulez vous dans votre salade de fruit ?<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 vannill&eacute;
<option>sucre de canne
<option>sirop
<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>
```

Le source de salade.pl est un petit peu plus complexe :

```
#!/bin/perl
use CGI Lite;                # Module CGI Lite...
my $cgi=new CGI Lite;           # On cree un objet de type CGI
my %in = $cgi->parse form data;# On lit les donnees
print "Content-type: text/html\n\n"; # Indispensable: Header

# Les champs du formulaire sont maintenant dans le tableau
# associatif %in. Le contenu d'un champ sera donc $in{'champ'}
# ATTENTION: Certains champs ont plusieurs valeurs
# (SELECT multiple, boites-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>Compos&eacute;e de <ul>\n";

# Pour toute valeur
foreach my $f ($cgi->get multiple values ($in{'fruit'})) {
    print "<li>$f</li>\n";
}

print "</ul>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";
}
```

E. Accès aux bases de données : DBI

Un des aspects les plus intéressants de Perl. Il permet d'intégrer des requêtes SQL dans un programme.
Depuis Perl version 5, on accède de la même manière à une base de données quelque soit le système choisi. Auparavant l'accès était différent si on utilisait Oraperl (Oracle), ou Syperl (Sybase)... Maintenant on utilise un module DBI (database interface), ce qui permet de spécifier, à la connexion, que l'on travaille sur une base Oracle, MySQL, Ingres, ou Sybase...

Quand on installe Perl, il faut installer un « module » DBI, et un « module » DBD : :oracle (ou *DBD : :Sybase*, ou *DBD :mysql* …). Pour l'installation de modules cf XIII.BInstallation de modules p. 53).

Au LIM nous utilisons indifféremment MySQL et Oracle (sur Unix uniquement)
Voici quelques commandes à utiliser pour accéder aux bases de données.

- `use DBI`; En début de programme spécifie que l'on va faire de l'accès aux bases de données

- `$dbh = DBI->connect("dbi:Oracle:cours", 'nom', 'mot-de-passe');`
  => Connexion à Oracle, sur la base de données qui s'appelle « cours », avec le l'utilisateur *nom* et son mot de passe.
  Ou `$dbh = DBI->connect("dbi:mysql:test", 'nom', 'mot-de-passe');` => Connexion à MySQL, sur la base de données qui s'appelle « test»

- `$dbh->disconnect`; => Pour se déconnecter de la base de données en fin de programme

- `$dbh->do("requête");` => Pour exécuter une requête SQL.
  A n'utiliser qu'avec des requêtes du genre **create table**, **update** *nom-table*, **insert into** *nom-table*...
  Il est préférable de traiter une erreur éventuelle dans ce genre d'opération (requête SQL mal formulée, connexion interdite, ...). On le fait de la même manière que pour la détection des erreurs dans l'ouverture de fichier :
  `$dbh->do("requête") || die "Pb de requête : $DBI::errstr";`
  (l'erreur vient du système de base de données, c'est pourquoi on utilise la variable $DBI::errstr)

- Pour une requête de type **select**, on va procéder en quatre temps :

1) préparation de la requête (`prepare`),
2. exécution (`execute`),
3. parcours de chaque ligne retournée par la requête (`fetchrow_array`), dans une boucle,
4. fin de la requête (`finish`)

cf. Perldoc.com Module DBI

Exemple: Lister toutes les lignes d'une table

```
my $sel = $dbh->prepare("select * from table where condition");
$sel->execute || die "Pb de sélection : $DBI::errstr";
while (my ($champ1, $champ2, ...) = $sel->fetchrow array) {
    print "Contenu: $champ1, $champ2, ... \n";
}
$sel->finish;                    # On ferme la requête select
```

Remarque:
La lecture d'une ligne se fait donc avec `$sel->fetchrow_array`, qui retourne en fait un tableau (d'où son nom). On pourrait donc écrire `@tab = $sel->fetchrow_array`;

Il est possible de paramétrer une requête, en mettant des points d'interrogation dans la requête au niveau du `prepare`, les paramètres seront spécifiés dans le `execute`.

`Exemple:` Afficher le nom d'un patient dont le numéro est demandé à l'utilisateur

```
my $sel = $dbh->prepare("select nom from patient where no = ?");
print "Veuillez entrer un numéro de patient";
my $nopatient = <STDIN>; # lecture au clavier (entrée standard)
chomp($nopatient); # ne pas oublier d'enlever le retour-chariot
$sel->execute($nopatient)
     || die "Pb de sélection : $DBI::errstr";
my ($nom) = $sel->fetchrow array;
print "Nom du patient $nopatient : $nom \n";
$sel->finish;
```

Voici maintenant un exemple de programme Perl qui crée une table "patient" avec trois champs (numéro, nom, prénom), et qui demande de saisir une liste de noms - prénoms, et qui, pour chaque ligne, insère les données dans la table (le programme se charge d'incrémenter à chaque fois le numéro de patient) :

```
#!/bin/perl
use strict;
use DBI;                  # On va utiliser la base de données

# Connexion à Oracle sous l'utilisateur scott
my $dbh = DBI->connect("dbi:Oracle:cours", 'scott','tiger');

# Création de la table patient :
$dbh->do("create table mpatient (
numero number(10) not null primary key,
nom    varchar2(40),
prenom varchar2(20))") || die "Pb création table: $DBI::errstr";

# Préparation d'une requête d'insertion des valeurs dans la BDD
my $ins = $dbh->prepare("insert into mpatient values (?, ?, ?)");

my $nopatient=0;                    # Initialisation du compteur

print "Entrez une série de nom-prenom, finir par CONTROL-D\n";

while (my $ligne = <>) {            # Pour chaque ligne lue...
     # Séparation en nom,prenom
     # caractère de séparation: tabulation
     my ($nom, $prenom) = split(/\t/, $ligne);

     # Exécution de la requete (insert), détection de l'erreur
     $ins->execute($nopatient, $nom, $prenom)
        || die "Pb à l'insertion : $DBI::errstr";
     $nopatient++;               # Incrémentation du compteur
}
$ins->finish;                    # On ferme la requête insert
$dbh->disconnect;                # Déconnexion de la BDD
```

En fait, on lit les données de l'entrée standard (<>), ce même programme pourrait fonctionner aussi bien avec un fichier venu d'un tableur (Excel ou autre).

Voici maintenant un programme qui affiche le contenu de la table mpatient (que l'on vient de créer).

```
#!/bin/perl
use strict;
use DBI;                    # On va utiliser la base de données

my $dbh = DBI->connect('dbi:Oracle:cours', 'scott','tiger');

# Préparation d'une requête de sélection des valeurs
my $sel = $dbh->prepare('select * from mpatient');
$sel->execute || die "Pb à la sélection : $DBI::errstr";

print "Voici la liste des patients enregistrés";

while (my ($nopatient, $nom, $prenom) = $sel->fetchrow array) {
# Pour chaque ligne lue...
    print "Patient no $nopatient : $nom, $prenom\n";
}
$sel->finish;               # On ferme la requête insert
$dbh->disconnect;           # Déconnexion de la BDD
```

Il existe d'autres méthodes pour accéder aux bases de données. Voir la documentation Perl à ce propos,

cf. Perldoc.com Module DBI

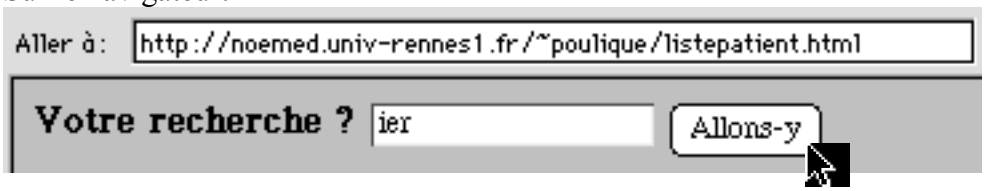
F. Accès à une base de données depuis Web

Pour accéder à une base de données depuis Web, cela sous-entend que l'on va utiliser des programmes CGI qui accèdent à une base de données (CGI_Lite et DBI)
Voici un petit exemple simple qui liste tous les patients de notre base de données sous la forme d'un tableau (on demandera au préalable de saisir un motif de recherche des patients).

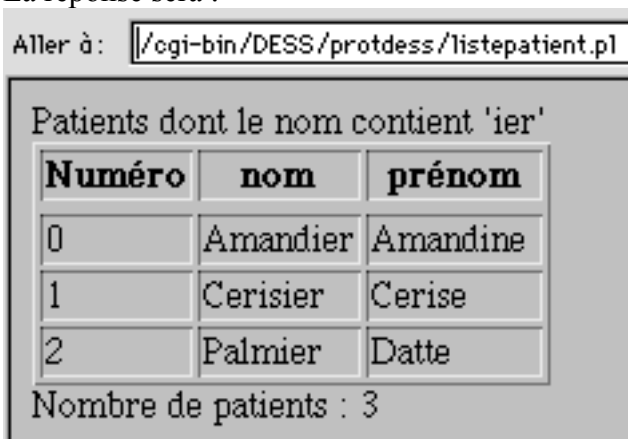
Formulaire: listepatient.html

```
<form action="/cgi-bin/DESS/protdess/listepatient.pl" METHOD=POST>
<b>Votre recherche ? </b>
<input type=text name=recherche>
<input type=submit value="Allons-y">
</form>
```

Sur le navigateur:



La réponse sera :



le programme listepatient.pl

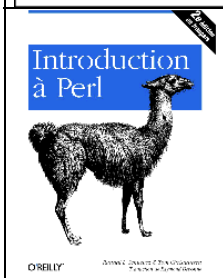
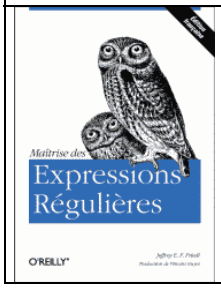
```
#!/bin/perl
use strict;
use DBI;                    # On va utiliser la base de donnees
use CGI Lite;               # Module CGI Lite...
my $cgi=new CGI Lite;           # On cree un objet de type CGI
my %in = $cgi->parse form data; # On lit les donnees

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

my $dbh = DBI->connect('dbi:Oracle:cours','scott','tiger');

# Preparation d'une requete de selection des valeurs
my $sel=$dbh->prepare('select * from mpatient where nom like ?');
my $recherche = "%$in{'recherche'}%";
$sel->execute($recherche)
              || die "Pb à la sélection : $DBI::errstr";

print "Patients dont le nom contient $in{'recherche'}";
print "<table border><tr><th>Num&eacute;ro</th><th>nom</th>
<th> pr&eacute;nom</th></tr>";
my $nblignes=0;
while (my ($nopat, $nom, $prenom) = $sel->fetchrow array) {
# Pour chaque ligne lue...
   print "<tr><td>$nop</td><td>$nom</td><td>$prenom</td></tr>\n";
   $nblignes++;
}
print "</table>Nombre de patients : $nblignes";
$sel->finish;                   # On ferme la requête insert
$dbh->disconnect;               # Déconnexion de la BDD
```

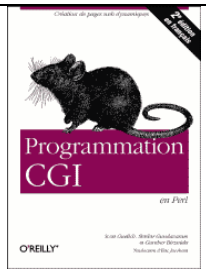
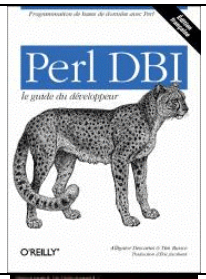
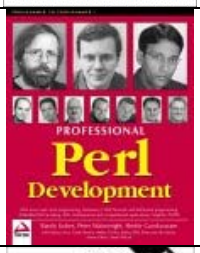
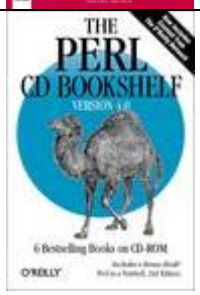
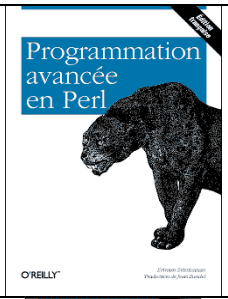
XII. Bibliographie

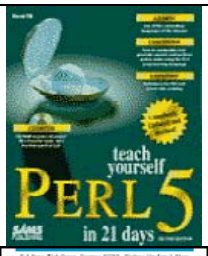
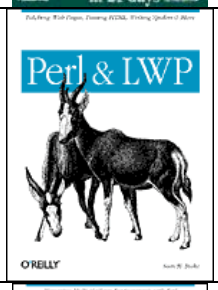
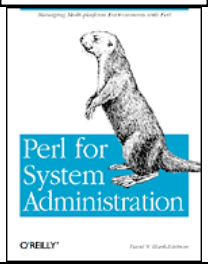
La bibliographie présente quelques références de livres en français ou en anglais. Accessoirement une URL permet d'avoir des informations sur le livre.
La plupart des livres ont étés lus (aussi je me permet de faire un jugement). L'éditeur O'Reilly possède un site web spécial contenant tous ses livres sur Perl : http://perl.oreilly.com/ (http://www.oreilly.fr/perl.html si vous êtes fâchés avec l'anglais)

Voici, pour commencer une petite « webographie » :

- **http://www.perl.com/** contient le programme Perl lui-même (téléchargeable gratuitement bien sûr) ainsi qu'une mine d'information sur les différents modules, sur les adresses à consulter…
- **http://www.enstimac.fr/Perl/** La documentation de Perl traduite en français
- **http://perldoc.com/** La documentation officielle de Perl (en anglais)
- http://books.perl.org/ Bibliographie en ligne
- http://www.med.univ-rennes1.fr/~poulique/cours/perl/ Il s'agit de ce cours, notamment les exemples sous : http://www.med.univ-rennes1.fr/~poulique/cours/perl/exemples/

LIVRE	Titre, auteurs, prix, éditeur (éventuellement URL)	Commentaires
	Programmation Perl (3éme édition) Larry Wall, Tom Christiansen & Jon Orwant Décembre 2001, 1074 pages, 54€ ed. O'Reilly & Associates ISBN: 2-84177-140-7 http://www.oreilly.fr/catalogue/pperl3.html	Le livre de référence sur Perl. Larry Wall (créateur du langage Perl) en est le co-auteur. Parfois trop technique et pas assez pédagogique
	Introduction à Perl (3ème édition) Randal L. Schwartz & Tom Phoenix Janvier 2002, 308 pages, 34€ ed. O'Reilly & Associates ISBN: 2-84177-201-2 http://www.oreilly.fr/catalogue/intro-perl-3ed.html	Un livre plus pédagogique que « programmation Perl », mais moins fournit.
	Maîtrise des expressions régulières Jeffrey E. F. Friedl 2e édition, juin 2003, 486 pages, 40€ ed. O'Reilly & Associates ISBN : 2-84177-236-5 http://www.oreilly.fr/catalogue/2841772365.html	Tout ce que vous avez toujours voulu savoir sur les expressions régulières.

LIVRE	Titre, auteurs, prix, éditeur, (éventuellement URL)	Commentaires
	**Programmer des CGI en Perl** (2ème édition) Scott Guelich, Shishir Gundavaram et G. Birznieks Juin 2001, 477 pages, 38€ ed. O'Reilly & Associates ISBN: 2-84177-098-2 http://www.oreilly.fr/catalogue/cgi_perl2.html	Livre sur les techniques CGI avec les exemples en Perl. Les différents types de CGI sont abordés, mais pas forcément en profondeur
	**Perl DBI, le guide du développeur** Alligator Descartes & Tim Bunce Décembre 2000, 390 pages, 34€ ed. O'Reilly & Associates ISBN : 2-84177-131-8	Notamment dans le domaine médical, on utilise très souvent Perl avec les bases de données.
	**Professional Perl Development** Randy Kobes, Peter Wainwright, S. Gundavaram Avril 2001, 725 pages, 64€ ed. Wrox Press ISBN : 1-86100-438-9	Non lu, mais l'éditeur a bonne réputation
	**Perl resource kit** (V4.0 prévue pour Janvier 2004) **6 livres sur un CD-ROM**: *"Perl in a Nutshell"*, "*Mastering Regular Expressions*"; "*Learning Perl*", "*Programming Perl*", "*Learning Perl Objects, References, and Modules*", et "*Perl Cookbook*" Janvier 2004 (actuelle version: 3.0), 100$ ISBN 0-596-00622-5 http://www.oreilly.com/catalog/perlcdbs4/	Un CDROM contenant 6 livres (et une version papier de "*Perl in a nutshell*").Certaine ment le meilleur achat (si vous lisez l'anglais)
	**Programmation avancée en Perl** Sriram Srinivasan Juin 1998, 448 pages, 43€ ed. O'Reilly & Associates ISBN : 2-84177-039-7	Pour ceux qui veulent mieux comprendre le fonctionnement de Perl lui-même. Comprend aussi une initiation à PerlTk
	**Perl en action** Tom Christiansen & Nathan Torkington Septembre 1999, 972 pages, 54€ ed. O'Reilly & Associates ISBN : 2-84177-077-X	Recettes et solutions. Une vraie mine d'or.

	**Perl 5 en 21 jours** David Till Mai-96, 840 pages,30$ ed. SAMS ISBN: 2-7440-1202-5	Un gros livre qui permet d'apprendre pas-à-pas toutes les fonctionnalités de Perl. **Mais il faut avoir 21 jours devant vous !**
	**Perl & LWP** Sean M. Burke Juin 2002, 264 pages, 35€ ed. O'Reilly & Associates ISBN: 0-596-00178-9	Si vous avez l'intention d'utiliser Perl pour la consultation automatique de sites web, voilà la bible.
	**Perl for System Administration** David N. Blank-Edelman Juillet 2000, 35$ ed. O'Reilly & Associates ISBN: 1-56592-609-9 http://www.oreilly.com/catalog/perlsysadm/	Utile pour l'administration sur Windows / NT (livre décevant pour Unix, encore plus pour Mac OS…)

A découvrir chez O'reilly: "Learning Perl Objects, References & Modules", "Introduction à Perl pour la bioinformatique", "Perl & XML"

INDEX

XIII. ANNEXES : Installation et utilisation de Perl

A. Installation du programme principal

Tout commence à l'adresse http://www.perl.org/.
Pour installer Perl, vous avez le choix entre télécharger les « sources » du programme et les compiler vous-même (c'est intéressant, faisable, mais prévoyez 2-3 jours !).
Bien heureux les utilisateurs de Linux qui ont Perl d'entrée de jeux !
Sur les autres systèmes, je vous conseille de télécharger une distribution « binaire » que l'on trouvera sur les serveurs CPAN (Comprehensive Perl Archive Netword) dont la page de garde est : http://www.cpan.org/.
Heureusement pour nous ils existe de nombreux sites miroirs, notamment en France :
Exemple : ftp://ftp.pasteur.fr/pub/computing/CPAN/README.html

Suivez le lien « binary distributions ("ports") », et choisissez votre système d'exploitation.

1. Sur Unix

Tout dépend de l'Unix, en général vous téléchargerez un fichier compacté (utilisez « gunzip »). Parfois le résultat sera un « installeur » (programme faisant toute l'installation), parfois une archive (fichier finissant par « .tar », que vous décompacterez en utilisant la commande « tar xvf *fichier.tar* »), lire les instruction d'installation dans un fichier « README ».

2. Sur Windows ou NT

Je vous propose de le télécharger directement sur le Web une version ActiveState :
http://www.activestate.com/Products/ActivePerl/Download.html

Il suffit de suivre les instructions derrière "dowload", préférez la version « **ActivePerl 5.8.1 build 807** » (ou ultérieure)...

3. Sur Mac OS

ftp://ftp.pasteur.fr/pub/computing/CPAN/ports/index.html#mac
Vous téléchargez un fichier compacté (ex : Mac_Perl_520r4_appl.bin), que vous traiterez avec l'application « stuffit expander », qui créera un fichier exécutable (ex : Mac_Perl_520r4_appl).

B. Installation de modules

Les « modules » sont des librairies qui ajoutent des possibilités au langage Perl. Dans ce cours nous avons vu les deux exemples de modules : « CGI_Lite » et « DBI », ces modules ne sont pas livrés avec le langage, c'est à vous de les installer.
Ces modules sont tous répertoriés dans les archives CPAN. Un utilitaire vous permet de télécharger, compiler et installer un module depuis Internet. Cela signifie qu'il faut que vous soyez connectés à Internet, et que vous connaissez les paramètres réseau (il faudra spécifier, si besoin, votre proxy…).

1. Sur Unix

Utilisez la commande « perl –MCPAN –e shell »
Pour installer un module, tapez :

```
install CGI_Lite
```

Pour recherche un module (par exemple : les modules parlant de CGI) tapez :

```
i /CGI/
```

2. Sur Windows

Vous trouverez dans votre distribution un programme « ppm.bat », il suffit de le lancer, et vous accéderez à la même interface que sous Unix.

3. Sur Macintosh

Contrairement aux autres distributions, il n'y a pas (d'emblée) d'utilitaire permettant de charger automatiquement les modules. Néanmoins un utilitaire a été développé : « cpan-mac », à télécharger à l'adresse :
http://pudge.net/macperl

4. Quelques modules utiles

Nom du module	Commentaires
LWP : :UserAgent	Utilitaires pour la consultation automatiques de site Web
CGI_Lite	Tout ce qu'il faut pour « récupérer » les informations d'un formulaire Web dans un CGI
CGI	Offre plus de possibilités que le précédent module, mais moins facile d'accès
DBI	Accès aux bases de données
Date : :Manip	Tout ce qu'il faut pour manipuler des dates (y compris en Français !)
GD	Permet de créer des images GIF
Net::LDAP	Consulter des annuaires LDAP automatiquement
Net::SMTP	Envoyer des courriers électronique
Mail::POP3Client	Consultation automatisée de courrier
GIFGraph	Créer des images GIF pour faire des graphes statistiques (Camemberts ou barres)
Apache	Un ensemble de modules pour la connexion avec Apache (installation complexe)
XML : :Parse	Permet de manipuler des fichiers XML

C. Documentation en ligne

Sur Unix, vous disposez d'un manuel habituel :
« `man perl` »
Vous accéderez aussi à la documentation de chaque module en tapant :

```
Perldoc module
```

Exemple : `perldoc DBD::mysql`

La documentation sur chaque fonction est accessible avec l'option "-f" :
Ex: `perldoc -f print`

Documentation sur Internet :
http://www.perldoc.com/
ou, en français :
http://www.enstimac.fr/Perl/DocFr/perl.html

D. Perl sous Unix

Il faut bien comprendre qu'un script Perl est un **texte** contenant des instructions Perl. Ce texte sera ensuite exécuté par l'interpréteur Perl.

On créera donc un fichier texte à l'aide d'un éditeur de texte (comme « vi » ou « emacs »…)
De préférence, suffixer les scripts perl par l'extension « .pl »
Créer le script :

```
emacs nom-fichier.pl
```

Il faut obligatoirement commencer par la ligne :

```
#!/bin/perl    (ou le chemin d'accès à l'interpréteur Perl)
Syntaxe Unix pour désigner l'interpréteur …
```

Les programmes Perl sont des « exécutables », il faut donc placer le mode « x » sur le programme.

E. Perl sous Windows

Lancer « notepad » (ou tout un autre éditeur de texte)
Et créer un fichier qui s'appellera "bonjour.pl" (".pl" est l'extension désignant perl)

- taper le programme perl
- l'enregistrer (bien penser à l'enregistrer au format texte).
- Pour l'exécuter, deux solutions :
  - dans le navigateur Windows "double-cliquer" dessus (problème : on n'a le temps de rien voir !)
  - Ouvrir une session DOS et taper : perl bonjour.pl
- Pour la mise au point: n'oubliez pas de le sauvegarder avant de l'exécuter

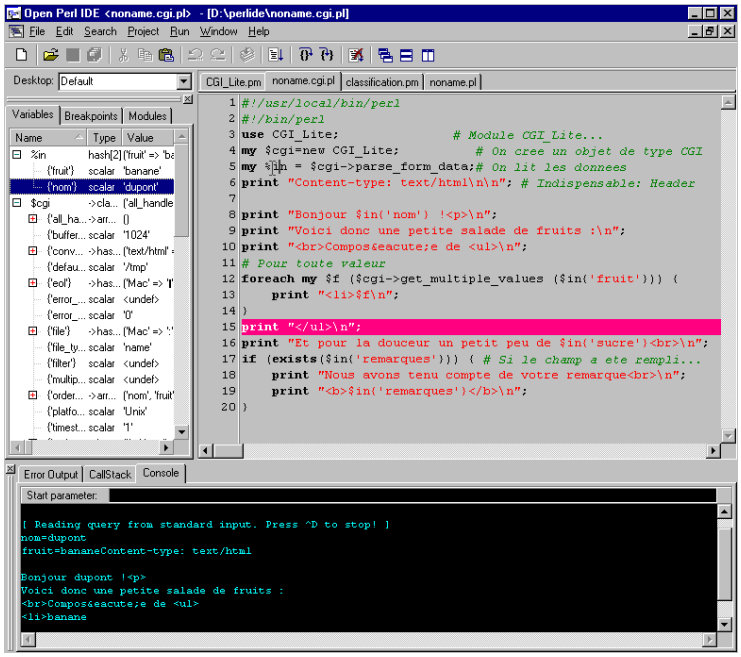
Le principal piège sous Windows : les chemins d'accès aux fichiers sont désignés par des « \ » (séparateur de fichiers) , ce qui amène souvent à écrire :

```
Open (F, "C:\\repertoire\\$fichier") ;
```

F. Environnements de développement

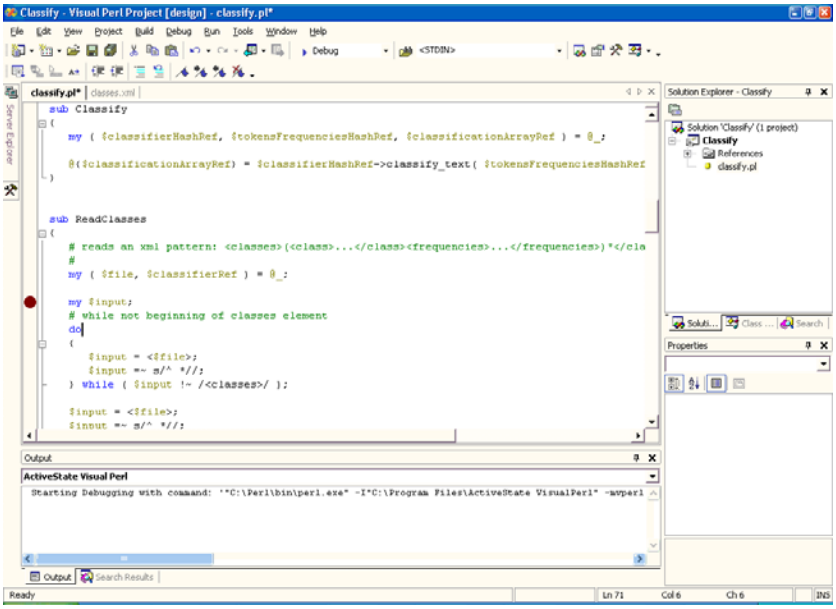
1. Open Perl-Ide

http://open-perl-ide.sourceforge.net/ sous Windows seulement , entièrement gratuit, très utile, interface agréable. Pas de traitement spécial des CGI.



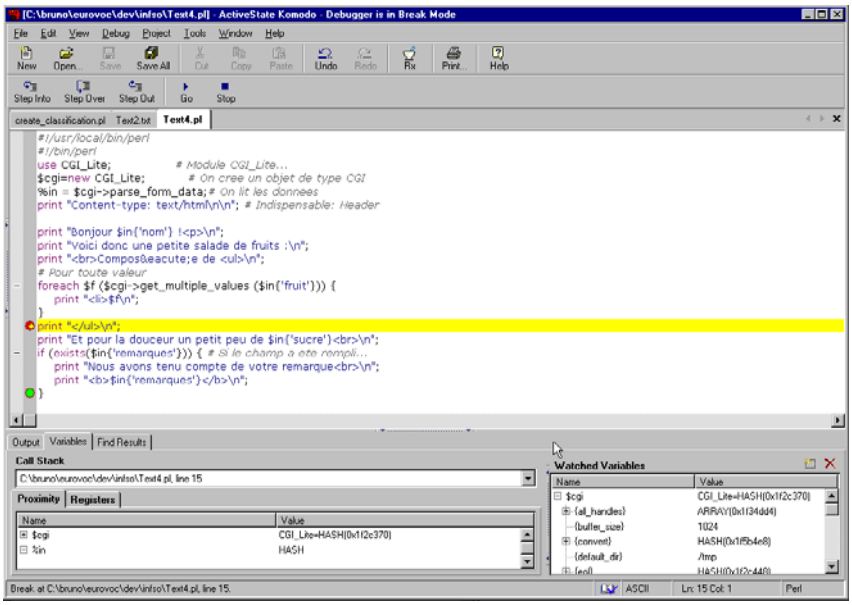
2. Visual perl

http://www.activestate.com/Products/Visual_Perl/
Nécessite d'avoir Visual Studio installé au préalable (coût: 295$)



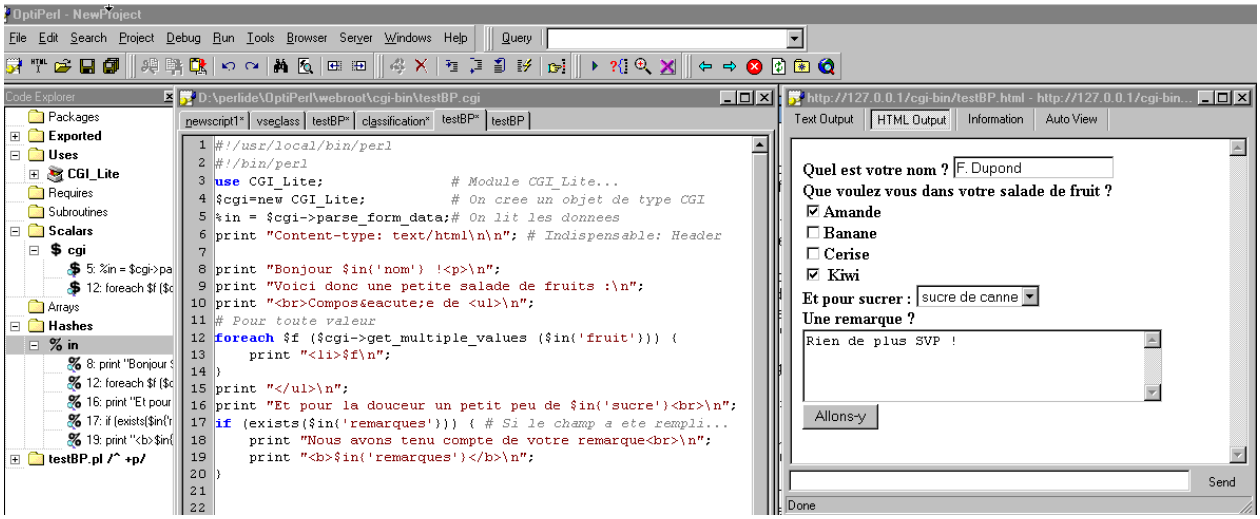
3. Komodo

http://www.activestate.com/Products/Komodo/ Sur Windows ou Linux (coût de 29$ à 299$)



4. OptiPerl

http://www.xarka.com/optiperl sous Windows, pas mal pour les CGI, (39-59-259$)



5. SlickEdit

http://www.slickedit.com/home.php 329$ (ou 109$ prix éducation). Pas vraiment un environnement de développement, mais seulement un éditeur assez sympa pour Perl (si vous êtes fâchés avec *emacs* par exemple, mais un peu cher pour un éditeur de texte à mon avis)

6. Perl Builder

http://www.solutionsoft.com/perl.htm existe sous Windows et Linux (149$) un peu moins cher que les autres. Je dois avouer, sur Windows, que j'ai renoncé après 15 minutes d'utilisation (très, très lent…).