,

Le langage Perl

D. Puthier[1]

[1] Inserm U928/Technologies Avancées pour le Génome et la Clinique,
http://tagc.univ-mrs.fr/staff/Puthier,
puthier@tagc.univ-mrs.fr

ESIL, 2009

- **Perl** est l'abréviation de "**Practical Extraction and Report Language**" (un langage adapté à l'extraction et la génération de rapports).
- Il a été développé durant les années 80 par Larry Wall.
- C'est un langage interprété dont la syntaxe est proche des scripts **shell**, **awk**, **sed** ou encore de celle du langage **C**.
- Il est particuliérement adapté à:
  - Manipulation du texte (analyse syntaxique / "Parsing"). +++++
  - Manipulation des séquences (Bioperl).
  - Accés aux bases de données (DBI). +++
  - Programmation web (CGI Perl).+
- Pour ces raisons, il est particulièrement utilisé dans le domaine de la bio-informatique.

- Tapez le texte ci-dessous dans un éditeur (gedit, emacs, kate,vi,...).
- La première ligne ("Shebang") indique que le code doit être interprété par l'executable "perl" (situé dans "/usr/bin/").

```
#!/usr/bin/perl
print "Hello";
```

- Enregistrez le document sous le nom 'hello.pl'. Rendez le fichier executable.

```
[userx@mamachine] ls hello.pl
-rw-rw-r--  1 puthier user  31 nov 23 14:12 hello.pl
[userx@mamachine] chmod u+x hello.pl
[userx@mamachine] ls hello.pl
-rwxrw-r--  1 puthier user  31 nov 23 14:12 hello.pl
```

- Lancez le mini programme avec la ligne de commande suivante:

```
[userx@mamachine] ./hello.pl
Hello
```

- Le type **scalaire** est l'équivalent du singulier (variable 'atomique').

```
$pi = 3.14159265;
$char = "yes";
print $pi, "\t", $char, "\n";
```

3.14159265 yes

- les variables peuvent contenir des entiers des décimales ou des caractères.
- Au contraire des simple guillemets (') Les doubles guillemets (") permettent l'interpolation des variables et des caractères particuliers comme: '\n', '\t', ...

```
$text1 = 'Ce panier contient $a poires\n';
$text2 ="\nCe panier contient $a poires\n";

print $text1;
print $text2;
```

Ce panier contient $a poires\n
Ce panier contient 2 poires

- Un vecteur de scalaires est appelé **tableau**.
- La variable de type **tableau** est préfixée par le caractère '@'.
- les éléments à inclure sont entre parenthèses. +++
- Ce tableau peut contenir des chaînes de caractères, des numériques ou des pointeurs.

```
@tab1 = ('Gly','Ala','Arg');
@tab2 = ('A'..'Z');
@tab3 = (4,5,6);
@tab4 = (1..50);

print join(" ", @tab1) , "\n;
print join(" ", @tab2) , "\n;
print join(" ", @tab3) , "\n;
print join(" ", @tab4) , "\n;
```

```
Gly Ala Arg
A B C D E F
4 5 6
1 2 3 4 5
```

- Pour l'indexation des tableaux on utilisera l'opérateur '['.
- Le premier élément d'un tableau se trouve à l'indice 0.
- On peut récupérer plusieurs éléments d'un tableau en séparant les indices par des virgules ou en utilisant l'opérateur **flip-flop** (on parle de 'slices').
- Attention, si on veut extraire un seul élément, on doit préfixer le tableau avec '$'.
- Par contre, si on veut récupérer plusieurs éléments, il faut préfixer avec l'opérateur '@'.

```
print $tab1[0],    "\n";
print join(" ",@tab1[0,2]),  "\n";
print join(" ",@tab1[0..2]), "\n";
```

```
Gly
Gly Arg
Gly Ala Arg
```

- La variable scalaire '$#LeNomDeMonTableau' contient la valeur d'indice de la dernière case du tableau.
- Pour compter les éléments d'un tableau, on utilisera la fonction **scalar**.
- Notez l'utilisation facultative des parenthèses en Perl.

```
print join(" ", @tab1) , "\n";
print $#tab1 , "\n";
print $tab1[$#tab1], "\n";
$nb1 = scalar(@tab1);
$nb2 = scalar @tab1;
print $nb1, " ", $nb2, "\n";
```

```
Gly Ala Arg
2
Arg
3 3
```

- On peut utiliser la fonction **splice** pour modifier les éléments d'un tableau.
- Syntaxe: *splice (array, start, length, replacement-values)*

```
print @tab1 , "\n";
splice (@tab1, 1, 2, ('Arg','Thr'));
print @tab1 , "\n";
```

Gly Ala Arg
Gly Arg Thr

- La fonction **pop** supprime le dernier élément d'un tableau et le renvoie.
- La fonction **push** ajoute un élément à la fin d'un tableau.

```
print join(" ",@tab1), "\n";
$x = pop @tab1;
print $x,"\n";
@tmp = ('His','Asn','His');
push  @tab1, @tmp;
print join(" ",@tab1), "\n";
```

Gly Arg Thr
Thr
Gly Arg His Asn His

- De même on utilisera les fonctions **shift** et **unshift** qui suppriment et ajoutent un ou plusieurs éléments au début d'un tableau.

```
$x= shift @tab1;
print "x= ", $x, "\n";
unshift @tab1, "Lys" ;
print join(" ",@tab1), "\n";
```

```
Gly Arg His Asn His
x= Gly
Lys Arg His Asn His
```

- Perl permet de réaliser certaines opérations en mode scalaire ou en mode liste.

```
#Exemple:  on crée un tableau;

@tab = ('one','two','three');

# On effectue une opération en contexte scalaire:
$var =  @tab;
print "En contexte scalaire var contient: ", $var, "\n";

# On effectue une opération en contexte de liste:
($var) =  @tab;
print "En contexte de liste var contient: ", $var, "\n";

# Une utilisation fréquente du mode liste:
($var1, $var2) =  @tab;
print "var1 et var 2 contiennent: ", $var1, " ",$var2,  "\n";
```

```
En contexte scalaire var contient: 3
En contexte de liste var contient: one
var1 et var 2 contiennent: one two
```

- Le **tableau associatif** est une structure de données dans laquelle chaque indice est identifié par un nom (on parle de **clef**).
- La variable de type **hash** est préfixée par le caractère '%'.
- les éléments à inclure sont entre parenthèses. +++
- Elle pourra contenir des chaînes de caractères, des numériques ou des adresses mémoire.
- l'accès à un élément du hash se fait en préfixant le hash avec le caractère "$".

```
%codon2AA = (
  TTT  => F,
  TTC  => F,
  TTA  => L,
  TTG  => L
);

print $codon2AA{'TTT'},$codon2AA{'TTA'},$codon2AA{'TTG'};
```

FLL

- On pourra aussi générer un hash vide et créer des clefs à la volé.
- Les valeurs associées aux clefs peuvent être incrémentées via les opérateurs classiques d'incrémentation et de décrémentation.
- On pourra avoir accès aux noms des clefs avec la fonction **keys**.

```
%countAA = ();
$countAA{'Gly'} = 10;
$countAA{'Ala'}++;
$countAA{'Gly'}--;
@AA = keys(%countAA);
print $AA[0]," ",  $countAA{$AA[0]}, " ", $AA[1]," ",  $countAA{$AA[1]};
```

Ala 1 Gly 9

D. Puthier

- On pourra créer des tableaux à 2 dimensions avec la syntaxe suivante:

```
@matrice = ([0,2,3],[4,5,6],[6,7,8]);
print $matrice[0][2],  "\n";
```

3

D. Puthier

- Une référence, en Perl, peut être considérée comme le pointeur du langage C.
- C'est une variable qui contient une adresse mémoire.
- On peut créer un référence à un variable existante en utilisant l'opérateur '\.'

```
#Référence vers un scalaire
$GeneSymbol = 'Bcl2';
$pGeneSymbol = \$GeneSymbol;
print "La variable GeneSymbol contient ", $GeneSymbol, "\n";
print "La variable pGeneSymbol contient ", $pGeneSymbol, "\n";

#Référence vers un tableau
@Domain = ('BH1','BH2','BH3');
$pDomain = \@Domain;
print "La variable Domain contient ", @Domain, "\n";
print "La variable pDomain contient ", $pDomain, "\n";

#Référence vers un hash
%Motif = (AAATTCCT => 2,AATTGGC => 3, AATTGC => 4);
$pMotif = \%Motif;
print "La variable Motif contient ", %Motif, "\n";
print "La variable pMotif contient ", $pMotif, "\n";
```

```
La variable GeneSymbol contient Bcl2
La variable pGeneSymbol contient SCALAR(0x817b994)
La variable Domain contient BH1BH2BH3
La variable pDomain contient ARRAY(0x817ba0c)
La variable Motif contient AATTGC4AATTGGC3AAATTCCT2
La variable pMotif contient HASH(0x817bde4)
```

D. Puthier

- Quand les variables sont des références on accède au contenu des adresses mêmoire en 'déréférençant' ces variables.
  - En préfixant la variable de $, @ ou % (pour un scalaire, un tableau et un hash respectivement)
  - En utilisant des accolades et un préfixe ($, @ ou %).
  - En utilisant l'opérateur '->'.

```
#Pour un scalaire
print $$pGeneSymbol,"\n";
print ${$GeneSymbol},"\n";

#Pour un tableau
print @$pDomain,"\n";
print @{$pDomain},"\n"
print $pDomain->[0],"\n";

#Pour un hash
print %$pMotif,"\n";
print ${$pMotif}{"AATTGC"}, "\n";
print $pMotif->{"AATTGC"},"\n";
```

```
Bcl2
Bcl2
BH1BH2BH3
BH1BH2BH3
BH1
AATTGC4AATTGGC3AAATTCCT2
4
4
```

- Souvent on créera directement une référence à un hash ou à un tableau
- Dans ce cas on utilisera '' et '[]'.
- On dira que ce tableau ou ce hash sont **anonymes** car on ne peut y accéder que par la référence.

```
$Organism = [];
$Organism->[1]="Mus musculus";
print $Organism->[1],"\n";

$goTerm = {};
$goTerm->{'GO:0008283'} = "cell proliferation";
print $goTerm->{'GO:0008283'},"\n";
```

```
Mus musculus
cell proliferation
```

- Dans le cas d'une matrice "creuse" (comportant beaucoup de valeurs nulles), on pourra utiliser à la place d'une matrice un **hash de hash**.

```
$interActome = {};
$interActome->{"CD3E"}{"TCRA"}++;
$interActome->{"CD3E"}{"ZAP70"}++;
$interActome->{"CD3E"}{"SHC1"}++;
if(exists($interActome->{"CD3E"}{"SHC1"})){
print "CD3E peut se lier à SHC1\n";
}
```

CD3E peut se lier à SHC1

- Il n'existe pas de variables de type booléen (Vrai/Faux).
- En perl, les valeurs numériques 0 (ou 0.0), le caractère '0', la chaine vide "" ou une variable de type **undef** sont faux.
- Le reste est vrai.
- Notez que **undef** correspond à une variable déclarée (par exemple avec la fonction **my**) mais n'ayant pas reçu d'affectation.

- Pour des valeurs numériques, on utilisera les opérateurs de comparaison classiques: '==', '<', '<=', '>', '>=', '!=' [1].
- Lors d'une comparaison si le test est vérifié, perl renvoie 1 sinon il renvoie une chaîne vide (qui est fausse par défaut).

```
$a = 1.2;
$b = 1.2;
print $a==$b, "\n"; # renvoie 1 (VRAI)
print $a!=$b, "\n"; # une chaine vide (FAUX)
```

- Pour les chaînes de caractères on utilisera les opérateurs: 'eq', 'lt', 'gt', 'le', 'ge', 'ne' (**eq**ual, **l**ower **t**han, **g**reater **t**han, **l**ower or **e**qual, **g**reater or **e**qual, **n**ot **e**qual ...)
- Lorsqu'on compare des chaînes, on compare la valeur ASCII de chaque caractère.

```
$a = 'ATG';
$b = 'ATG';
print $a eq $b, "\n"; # renvoie 1 (VRAI)
print $a ne $b, "\n"; # une chaine vide (FAUX)
```

- Comme dans de nombreux langages de programmation on pourra utiliser la structure conditionnelle **if**.
- on pourra aussi utiliser **unless**.

```
# Structure if
if(condition){
  ...
}

# Structure if/else
if(condition){
  ...
}else{
  ...
}

# Structure if synthéthique
... if(condition);

# Structure unless
unless(condition){
}

# Structure unless synthéthique
... unless(condition);
```

- Le langage Perl permet d'effectuer des traitements itératifs comme de nombreux langages.
- On retrouvera les structures: **while**, **for**, **until**, **do while**,...
- Par ailleurs, Perl dispose de la structure **foreach** qui permet, de parcourir facilement les éléments d'un tableau.

```
# Exemple de boucle for
$ADN= "ATTTCTTCTTCT";
for($i=0; $i < (length($ADN) -2); $i+=3){
print substr($ADN,$i,3),"\n";
}

# Exemple de structure while
while(<FILE>){
  s/[\r\n]+//g'
  split "\t";
  print $_[0];
}

# Exemple de structure foreach

foreach $var (@tableau){
  print $var,"\n";
}
```

- La syntaxe des expressions régulières est très proche de celle que nous avons rencontrée avec les commandes **Unix**.

.	Un caractère quelconque (.sauf \n)
x*	0 ou n fois le caractères *x*.
x+	1 ou n fois le caractère *x*.
x{n,m}	Le caractère *x* répété entre n et m fois.
[*a* — *z*]	Une lettre minuscule (interval, ex: [*u* — *w*]).
[*A* — *Z*]	Une lettre majuscule (interval, ex: [*A* — *E*]).
[*ABc*]	A ou B ou c.
[^*ABab*]	Toute lettre différente de a et b.
a|b	Un a ou un b.
(toto)|(yaya)	Une chaîne de caractères ou une autre.
^	Début de ligne.
$	Fin de ligne.
\n	Retour à la ligne.
\t	Une tabulation.
\d	Un chiffre (**d**igit, [0 — 9]) .
\w	Un caractère rencontré dans un mot (un alphanumérique ou le caractère '_', [0-9a-zA-Z_]).
\s	Un blanc ([\t\r\n\f]).
\D	Tout sauf un chiffre.
\W	Tout sauf un caractère rencontré dans un mot.
\S	Tout sauf un blanc.
\	Caractère d'échappement (dé-spécialisation).

- $& Contient, lors d'une opération de recherche avec **match** la valeur correspondant au modèle de recherche.
- $1, $3,... Contient, lors d'une opération de recherche avec **match**, la valeur d'une expression régulière partielle repérée entre parenthèses.

```
$mail= 'mon adresse mail est la suivante: toto@cgfb.univ-mrs.fr';
$mail=~/(\w*)@(\w*)\.(\w*)-(\w*)\.(\w*)/;
print $1 ,"\n";
print $2, "\n";
print $3, "\n";
print $4, "\n";
print $&, "\n";
```

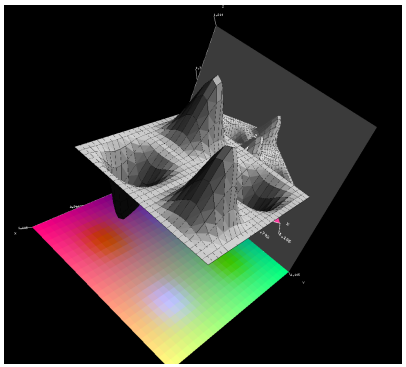
toto
cgfb
univ
mrs
toto@cgfb.univ-mrs.fr

- On rencontre les opérateurs de calcul classiques ('+', '-', '/', '*').
- Par ailleurs, on rencontrera les opérateurs 'préfixés' ('*=', '+=', '-=', '\=').
- De même les opérations d'incrementation/décrémentation ('++', '–') peuvent être utilisées en Perl.

```
$num1  = 4;
$num2  = 3;

$num1 += $num2;
print $num1, "\n";

$num1  = 4;
$num1 *= $num2;
print $num1, "\n";

print ++$num1, "\n";
```

7 12 13

- Perl dispose de fonctions pour le calcul plus évolué (**abs**, **sqrt**, **cos**, **exp**,...).
- La fonction **rand** peut être utilisée pour générer des valeurs aléatoires.
- On pourra eventuellement utiliser le module **PDL** pour faciliter les calculs sur des matrices.
- Cependant dans le cas de traitement essentiellement numériques (ex: statistiques), il peut être intéressant d'utiliser un autre langage (ex: R).



- La concaténation est effectuée avec l'opérateur '.'.
- On pourra utiliser la version préfixée '.='.
- On peut répéter une chaine avec l'opérateur 'x'.

```
$str1 = 'ATTT';
$str2 = 'GGA';
$str3 = $str1.$str2."NNN";
print $str3, "\n";
print "GCC" x 4 ,"\n";
```

```
ATTTGGANNN
GCCGCCGCCGCC
```

- La fonction **match**.
- Elle est appelée implicitement via la structure '=~/.../'.
- Permet de tester la présence d'une chaîne de caractères désignée par une expression régulière.
- On la retrouvera souvent à l'intérieure d'une boucle conditionnelle.
- Pour compter le nombre d'occurences d'un motif on pourra faire une recherche globale ('g') et stocker le résultat dans un tableau.

```
$str = "AUUCCAUU";
if($str =~/AUU/){
  print "La séquence contient AUU\n";
}

@tab = $str =~/AUU/g;
print join(';',@tab), "\n";
print scalar(@tab), "\n";
```

```
La séquence contient AUU
AUU;AUU 2
```

- La fonction **substitute**.
- Elle est appelée implicitement via la structure '=∼/.../.../'.
- Elle permet de substituer une chaîne de caractères par une autre.
- Le caractère 'g' placé eventuellement à la fin de cette structure, signifie 'global'.

```
$str =~ s/U/T/;
print $str, "\n";

$str =~ s/C/G/g;
print $str, "\n";
```

- **chomp**: Elle permet d'éliminer les caractères correspondant à la variable '$/' qui correspond au séparateur d'enregistrement (gènéralement \n').
- **split**: Etant donné un séparateur, la fonction découpe une chaîne de caractères et renvoie un tableau.

```
# Chomp
chomp $line;      ##  ou eventuellement $line =~ s/[\r\n]+//g;

# Split
$str = "AAU\tATT\tCCG";
@tab = split("\t", $str);
print join("\n", @tab);
```

D. Puthier

- On utilisera la fonction **open** souvent associée à **die**.
- Perl permet de gérer STDIN, STDOUT et de capturer les erreurs (STDERR).
- Pour lire dans un fichier, écrire ou ajouter des lignes, on utilisera les opérateurs '<', '>' et '>>', respectivement.
- Pour fermer un fichier, on utilisera **close**.
- Le fichier sera généralement parcouru à l'aide d'une boucle **while** et de l'**opérateur diamant** ('<>').
- Si on souhaite lire un flux (tube) on utilisera: while(<STDIN>).

```
# le script checkPasswd.pl
# Vérifie que chaque ligne contient un caractère ':'.

#!/usr/bin/perl
use warnings;
open(IN,"</etc/passwd") || die("File could not be read\n");

while($ligne = <IN>){
    chomp $ligne;
    print STDERR $ligne if($ligne !~/:/);
    $ligne =~ s/:.*//g;
    print $ligne,"\n";
}

close IN;
# Utiliser ce script de la façon suivante:

[user@machine] ./checkPasswd.pl > passwdNew.txt 2> error.txt
```

- $_ La ligne courante (dans une boucle while).
- $. Le numéro de la ligne.
- $& *Cf* paragraphe "expressions régulières".
- $1, $3,... *Cf* paragraphe "expression régulières".
- $0 Le nom du programme.
- @_ Un tableau contenant les arguments passés à une fonction ou le résultat de **split** sur $_.
- @ARGV Les arguments du programme.

```
# le script testARGS.pl
# Renvoie les arguments du programme
#!/usr/bin/perl
use warnings;

foreach $arg (@ARGV){
print "arg = ",$arg,"\n";
}

# Utiliser ce script de la façon suivante:
[user@machine]  ./test.pl "Mus musculus" DNA GENOMIC
```

```
arg = Mus musculus
arg = DNA
arg = GENOMIC
```

- Perl est souvent utilisé en mode 'one-liner'.
  - L'argument -e signifie 'execute'.
  - L'argument '-n' que le traitement est vrai pour chaque ligne.
  - L'argument '-a' indique à Perl qu'il doit effectuer une autosegmentation (split).
  - Le résultat de la segmentation est stocké dans une variable spéciale nommée '@F'.
  - '-F' : le type de séparateur utilisé pour la segmentation. Par défaut un espace.
  - L'argument '-l' delete \n en entrée et l'ajoute en sortie.

```
#EXEMPLES DE ONE-LINERS

# imprime la ligne si entre 1 et 10
[user@machine] perl -e 'print if(1..10)'

# avec l'opérateur flip-flop
perl -ne 'if(/table_begin/../table_end/){split "\t"; print $_[0]}' GSE1024_SeriesMatrix.t

# Substituer une thymine par un uracyle dans un fichier fasta (en évitant les en-têtes).
perl -ne 's/T/U/gi unless(/^>/); print uc' a14180.fasta

# Remplacer la deuxième ligne du fichier
perl -ne 'if ($. == 2){print "nouvelle ligne 2\n"}else{ print}' leFichierAModifier.txt

#Imprime la ligne sauf si elle commence par #
perl -F: -lane 'print $F[0] if !/^#/' /etc/passwd
```

- Lorsqu'on écrit des one-liners il est courant d'utiliser les structures BEGIN et END.
- Les instructions dans une structure BEGIN sont effectuées avant tout traitement.
- Les instructions dans un structure END sont effectuées en fin de traitement.

```
#EXEMPLES: Un fichier 1 contient des identifiants en colonne 1.
# Le fichier 2 contient des identifiants en colonne 1.
# Le séparateur est un espace dans les deux cas.
# Imprimer les lignes du fichiers 2 si l'identifiant est connu dans le fichier 1.

perl -ne 'BEGIN{open(FILE,"<file1.txt"); while(<FILE>){split; $id{$_[0]}++}}; split; pri
```

- La définition d'une nouvelle fonction est précédée du mot clef **sub**.
- les arguments passés à la fonction sont présents dans le tableau '@_'.
- Pour passer un tableau ou un hash on utilisera généralement un pointeur.

```
sub concatDNA {
    ($head,$tail) = @_;
    return($head.$tail);
}

print concatDNA("ATAGATACTTAG","AAAAAAAAA");
```

```
ATAGATACTTAGAAAAAAAAA
```

- Un espace de noms est un ensemble nommé contenant des éléments uniques.
- Définir un espace de noms permet d'empêcher les incohérences liées à des redondances dans les noms des variables.
- Le mot clef **package** permet de passer d'un espace de noms à l'autre.

```
package english;

$bonjour = "Hello";

package main;

$bonjour = "bonjour";

print $bonjour,"\n";

print $english::bonjour,"\n";
```

```
bonjour
Hello
```

- Un module Perl est une librairie ("package") réutilisable et stockée dans un fichier ".pm".

```
[user@machine]cat myPack.pm
package myPack;
sub hello {
   print "Hello $_[0]\n"
}
1;

[user@machine] cat callMyPack.pl
#!/usr/bin/perl

use lib "/home/puthier/Desktop/";
use myPack;

print "Starting...\n";
myPack::hello('Denis');


[user@machine]./callMyPack.pl
Starting...
Hello Denis
```

bonjour
Hello

- Un module Perl est une librairie ("package") réutilisable et stockée dans un fichier ".pm".

```
[user@machine]cat myPack.pm
package myPack;
sub hello {
   print "Hello $_[0]\n"
}
1;

[user@machine] cat callMyPack.pl
#!/usr/bin/perl

use lib "/home/puthier/Desktop/";
use myPack;

print "Starting...\n";
myPack::hello('Denis');


[user@machine]./callMyPack.pl
Starting...
Hello Denis
```

bonjour
Hello

- La programmation objet nécéssite la définition de classes.
- Une classe correspond à la modélisation informatique d'un objet réel.
- Si on omet la notion d'héritage (ce que nous ferons ici), un objet peut être vu simplement comme une structure contenant elle-même d'autres structures.
- Une classe possède des méthodes notamment, un constructeur.

```
[user@machine]  cat Gene.pm
package Gene;
use strict;
use warnings;

sub new {
    my ($class,%args) = @_;  # La fonction reçoit en paramètre le nom de la classe et un
    my $self= {              # On crée une référence vers un hach
        name => $args{name},
        org  => $args{org},
        chr  => $args{chr}
    };
    bless $self, $class;     # La référence contiendra un objet de type $class
    return $self;            # On renvoie la référence.
}

sub getName {$_[0]->{name}}; # on définie une méthode qui renvoie le nom.
sub getOrg {$_[0]->{org}};   # on définie une méthode qui renvoie l'organisme.

1;
```

```
[user@machine]   cat createGene.pl
#!/usr/bin/perl
use Gene;
use warnings;
use strict;

my $gene = Gene->new(name => 'bcl2', org => 'Homo sapiens',chr => '18q21');

print $gene->getName,"\n";
print $gene->getOrg,"\n";
```

```
bcl2
Homo sapiens
```

- **Bioperl** est une suite de librairies Perl pour la bioinformatique.
- Ecrit en Perl objet.
- Orienté vers l'analyse de séquences et leur représentation.
-
- Modules pour l'analyse de données de microarrays.

The Bioperl Toolkit: Perl Modules for the Life Sciences

Jason E. Stajich, David Block, Kris Boulez, et al.

Genome Res. 2002 12: 1611-1618
Access the most recent version at doi:10.1101/gr.361602

- Installation
- En suivant les directives sur **bioperl.org**.
- http://www.bioperl.org/wiki/Installing$_B$*ioperl*$_f$*or*$_U$*nix*
- En utilisant **Synaptic**.
- En utilisant **CPAN**

```
# Avec cpan
[user@machine] perl -MCPAN -e shell;
cpan> o conf prerequisites_policy follow
cpan> help
    Display Information
     command argument            description
     a,b,d,m WORD or /REGEXP/    about authors, bundles, distributions, modules
     i         WORD or /REGEXP/  about anything of above
     r         NONE              reinstall recommendations
     ls        AUTHOR            about files in the author's directory

    ....


cpan> o conf prerequisites_policy follow
cpan> i /bioperl/
cpan> install CJFIELDS/BioPerl-1.6.0.tar.gz
```

- **Seq**: Objet central de Bioperl.
- Permet de stocker une séquence et les annotations correspondantes.
- **Seq** peut être créé à partir d'un fichier ou d'un BD distante.

```
#Création à partir d'un ficher
use Bio::SeqIO;
$seqio  = Bio::SeqIO->new( '-format' => 'embl' , -file => 'mySeq.embl');
$seqobj = $seqio->next_seq();

#Création à partir de GenBank (remote)

$db = Bio::DB::GenBank->new();
$seqobj = $db->get_Seq_by_acc('X78121');

#Création à partir de EMBL (remote)
use Bio::DB::EMBL;
$embl = Bio::DB::EMBL->new();
$seqobj = $embl->get_Seq_by_id('ak297217');
print $seqobj->seq;
```

- Un fichier de séquence contient généralement de nombreuses informations.

```
ID   AK297217; SV 1; linear; mRNA; STD; HUM; 998 BP.
XX
AC   AK297217;
XX
DT   24-JUL-2008
DT   31-JUL-2008
XX
DE   Homo sapiens cDNA FLJ53764 complete cds, highly similar to Induced myeloid
DE   leukemia cell differentiation protein Mcl-1.
XX
KW   FLI_CDNA; oligo capping.
XX
OS   Homo sapiens (human)
OC   Eukaryota; Metazoa; Chordata; Craniata; Vertebrata; Euteleostomi; Mammalia;
OC   Eutheria; Euarchontoglires; Primates; Haplorrhini; Catarrhini; Hominidae;
OC   Homo.
XX
RN   [1]
RP   1-998
RA   Isogai T., Yamamoto J.;
RT   ;
RL   Submitted (09-OCT-2007) to the EMBL/GenBank/DDBJ databases. Contact:Takao
RL   Isogai Reverse Proteomics Research Institute; 1-9-11 Kaji-cho, Chiyoda-ku,
RL   Tokyo 101-0044, Japan E-mail :flj-cdna@nifty.com
XX
RN   [2]
RA   Wakamatsu A., Yamamoto J., Kimura K., Ishii S., Watanabe K., Sugiyama A.,
RA   Murakawa K., Kaida T., Tsuchiya K., Fukuzumi Y., Kumagai A., Oishi Y.,
RA   Yamamoto S., Ono Y., Komori Y., Yamazaki M., Kisu Y., Nishikawa T., Sugano
RA   S., Nomura N., Isogai T.;
RT   "NEDO human cDNA sequencing project focused on splicing variants";
RL   Unpublished.
XX
CC   Human cDNA sequencing project focused on splicing variants of mRNA in NEDO
CC   functional analysis of protein and research application project supported
CC   by Ministry of Economy, Trade and Industry, Japan; cDNA selection for
```

- l'objet seq contient l'ensemble des informations liées à une séquence.

```
use Bio::SeqIO;
use Bio::DB::EMBL;

$embl = Bio::DB::EMBL->new();
$seqobj = $embl->get_Seq_by_id('X64011');


print "id       = ",   $seqobj->display_id()       ,"\n"; # the database identifier
print "sequence = ",   $seqobj->seq()              ,"\n"; # the sequence
print "AccNum   = ",   $seqobj->accession_number() ,"\n"; # Sequence acc. number
print "Alphabet = ",   $seqobj->alphabet()         ,"\n"; # one of 'dna','rna','protein'

my $species = $seqobj->species();
print "Species  = ",  $species->binomial()        ,"\n";  # "Genus species"
```

- Les méthodes de l'objet Seq permettent d'effectuer les traitement de base sur une séquence:

```
print "subSeq      = ",       $seqobj->subseq(5,10)       ,"\n"; # a slice
print "Translation = ",       $seqobj->translate()        ,"\n";
print "Reverse complement", = $seqobj->revcom()           ,"\n";
```

- L'objet Seq contient des élements ('features').
- On peut ajouter ses propres éléments.

```
# Exemple: Features d'un fichier GenBank
FEATURES             Location/Qualifiers
     source          1..1846
                     /organism="Homo sapiens"
                     /db_xref="taxon:9606"
                     /chromosome="X"
                     /map="Xp11.4"
     gene            1..1846
                     /gene="NDP"
                     /note="ND"
                     /db_xref="LocusID:4693"
                     /db_xref="MIM:310600"
     CDS             409..810
                     /gene="NDP"
                     /note="Norrie disease (norrin)"
                     /codon_start=1
                     /product="Norrie disease protein"
                     /protein_id="NP_000257.1"
                     /db_xref="GI:4557789"
                     /db_xref="LocusID:4693"
                     /db_xref="MIM:310600"
                     /translation="MRKHVLAASFSMLSLLVIMGDTDSKTDSSFIMDSDPRRCMRHHY
                     VDSISHPLYKCSSKMVLLARCEGHCSQASRSEPLVSFSTVLKQPFRSSCHCCRPQTSK
                     LKALRLRCSGGMRLTATYRYILSCHCEECNS"
```

- L'objet Seq contient des élements ('features').
- On peut ajouter ses propres éléments.
- http://www.bioperl.org/wiki/HOWTO:Feature-Annotation

```
# Exemple dans un fihier GenBank
FEATURES             Location/Qualifiers
     source          1..12337571
                     /db_xref="taxon:9606"
                     /mol_type="genomic DNA"
                     /chromosome="10"
                     /organism="Homo sapiens"
     gene            complement(43727..71437)
                     /db_xref="GeneID:79754"
                     /db_xref="HGNC:19765"
                     /gene="ASB13"
                     /note="Derived by automated computational analysis using
                     gene prediction method: BestRefseq. Supporting evidence
                     includes similarity to: 1 mRNA"
     mRNA            complement(join(43727..45689,46629..46820,53829..53963,
                     56072..56222,57722..57909,71368..71437))
                     /db_xref="GI:22208956"
                     /db_xref="GeneID:79754"
                     /db_xref="HGNC:19765"
                     /exception="unclassified transcription discrepancy"
                     /gene="ASB13"
                     /product="ankyrin repeat and SOCS box-containing 13"
                     /transcript_id="NM_024701.2"
                     /note="Derived by automated computational analysis using
```

- L'objet **SeqIO** est utilisé pour lire un flux de séquence, le stocker et le re-formater.
- On utilisera la méthode **next_seq** pour traiter la séquence suivante.

```
#!/usr/bin/perl
use strict;
use warnings;
use Bio::SeqIO;

use Bio::SeqIO;
my $in = Bio::SeqIO->new(-file => "mySeq.embl", -format => 'EMBL');

while ( my $seq = $in->next_seq() ) {
    print $seq->seq,"\n";
}
```

- Conversion de format avec SeqIO
- Formats: EMBL, GenBank, Fasta, SWISS...
- http://bioperl.open-bio.org/wiki/HOWTO:SeqIO

```
my $in = Bio::SeqIO->new(-file => "mySeq.embl", -format => 'EMBL');
my $out = Bio::SeqIO->new(-file => ">mySeq.fasta", -format => 'FASTA');
while (my $seq = $in->next_seq() ) {
   $out->write_seq($seq);
}
```

- Permet de calculer des statistiques sur une séquence:
  - PM, fréquence en nt/AA, fréquence en codons...
  - Indice d'hydrophobie (GRAVY score)

```
my $seq_stats   =  Bio::Tools::SeqStats->new(-seq => $seqobj);  # on crée une instance de

my $hash_ref = $seq_stats->count_monomers();                    # nt ou AA

foreach my $base (sort keys %$hash_ref) {
    print "Number of bases of type ", $base, "= ", %$hash_ref->{$base},"\n";
}

$hash_ref = $seq_stats-> count_codons();  # for nucleic acid sequence

foreach my $base (sort keys %$hash_ref) {
             print "Number of codons of type ", $base, "= ",
             %$hash_ref->{$base},"\n";
}
```

D. Puthier


```
#!/usr/bin/perl
use strict;
use warnings;
use Bio::SeqIO;
use Bio::Tools::Run::RemoteBlast;

my $prog = 'blastn';
my $db   = 'nr';
my $e_val= '1e-10';

my @params = (
    '-prog' => $prog,
    '-data' => $db,
    '-expect' => $e_val,
    '-readmethod' => 'SearchIO');

my $factory = Bio::Tools::Run::RemoteBlast->new(@params);
my $str = Bio::SeqIO->new(-file=>'2Seqs.embl' , -format => 'EMBL' );

while (my $input = $str->next_seq()){                    # Pour chaque séquence

   my $r = $factory->submit_blast($input);               # On soumet la requête
   print STDERR "waiting...\n";

   while ( my @rids = $factory->each_rid ) {             # Tant que le traitement n'est pa
      foreach my $rid ( @rids ) {                        # Pour chaque requête
      my $rc = $factory->retrieve_blast($rid);           # On interroge le serveur

      if( !ref($rc) ) {                                  # Si rc n'est plus une référence
         if( $rc < 0 ) {                                 # Si rc == 0, le traitement est f
            $factory->remove_rid($rid);                  # Le traitement est fini on met r
         }
      } else {
         my $result = $rc->next_result();                # On recupère le résultat
         my $filename = $result->query_name()."\.out"; # On crée un fichier
         $factory->save_output($filename);               # On écrit dans ce fichier
         $factory->remove_rid($rid);
      }
      }
   }
}
```

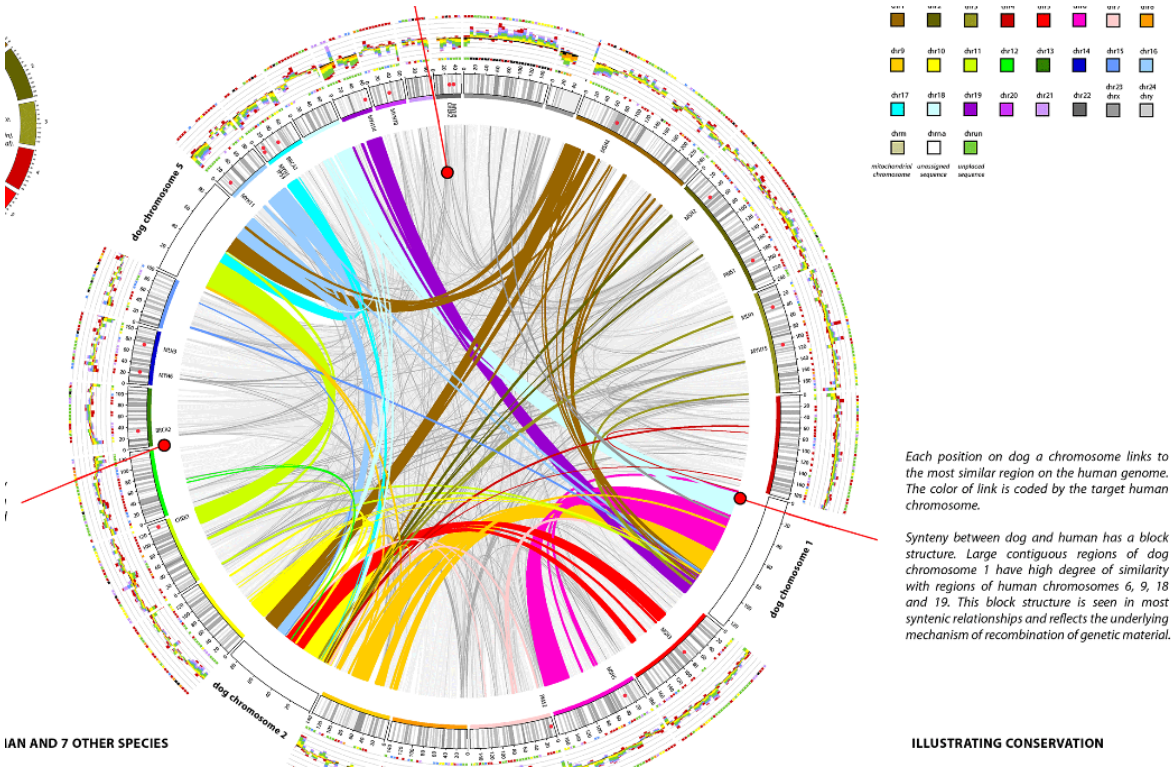
D. Puthier

- Un objet permettant de stocker et de 'parser' les résultats d'un algorithme d'alignement (ex: Blast).
- Extraction 'facile' des HSP's (High Scoring Sequence Pairs) via l'objet Bio::Search::HSP::HSPI

```
use strict;
use Bio::SearchIO;

my $in = new Bio::SearchIO(-format => 'blast',
                           -file   => 'NM_152710.bls');

while( my $result = $in->next_result ) {    # On se positionne sur un résultat
  while( my $hit = $result->next_hit ) {    # On récupère le nom des séquences à forte si
   while( my $hsp = $hit->next_hsp ) {      # On récupère les HSPs.
    if( $hsp->length('total') > 50 ) {      # Si la taille du HSP est > 50
     if ( $hsp->percent_identity >= 75 ) { # Si la similarité est forte
      print "Hit= ",        $hit->name,     # On imprime
            ",Length=",     $hsp->length('total'),
            ",Percent_id=", $hsp->percent_identity, "\n";
     }
    }
   }
  }
}
```

- Bio::DB::EUtilities
  - Recherche dans les bases de données du NCBI.
  - http://www.bioperl.org/wiki/HOWTO:EUtilities_Cookbook
  - Please do NOT spam the Entrez web server with multiple requests !!
- Bio::Restriction::Analysis;
  - Collection d'enzymes de restriction.
  - Analyse de restriction.
- Prédiction de gènes
  - interface vers Grail, Genescan,.Genemark, MZEF, Sim4, GlimmerM, GlimmerHMM
- Bio::Tools::SeqWords
  - Recherche de motifs
- Bio::Tools::SeqWords
- ...

- Perl can be beautiful...Isn't it



- http://mkweb.bcgsc.ca/circos/poster/circos-poster-8-large.png

- La commande **perldoc**
  - nécessite d'installer **perl-doc** via Synaptic.

```
# Aide sur une fonction
[useràmachine]   perldoc -f sort
# Aide sur un module
[useràmachine] perldoc Bio::Seq
```

- Sites web
  - http://www.cpan.org/
  - http://www.bioperl.org/wiki/Main*P**agehttp* : */ / www . bioperl . org / wiki / HOWTOs*

- http://www.bioperl.org/wiki/Deobfuscator
- http://paris.mongueurs.net/

NB: Nous avons introduit les notions de base dans ce cours. Pour des informations plus exhaustives, vous pouvez consulter les ouvrages suivants:

- Introduction à Perl (Rendre ce qui est facile facile et difficile possible). Randal L. Schwartz , Tom Phoenix , Brian D. Foy. Editeur: O'Reilly
- Introduction à Perl pour la bioinformatique. James Tisdall. Editeur: O'Reilly
- Mastering Perl for bioinformatics. James Tisdall. Editeur: O'Reilly O'Reilly