Inf7214 - Langage de programmation Perl (partie 3)

Vladimir Makarenkov et Alix Boc

UQAM

Hiver 2010

Plan

- Introduction au langage Perl
  - Les fonctions (*subroutines*)
  - Les expressions régulières
  - La recherche des motifs dans des chaînes de caractères
  - Un exemple bioinformatique
  - La librairie BioPerl
- Exercices

Les fonctions

Le mot clef **sub** permet de définir une fonction (subroutine). **sub** doit être suivi du nom de la fonction.

Exemple de fonction :

```
 5 # fonction sans valeur de retour
 6 sub Perimetre{
 7     my $Longueur = $_[0];
 8     my $Largeur = $_[1];
 9
10     print "le perimetre est : " . ($Longueur*$Largeur) . "\n" ;
11 }
```

Une fonction peut retourner une valeur en utilisant l'opérateur **return**.

```
13 # fonction avec valeur de retour
14 sub Perimetre2{
15     my $Longueur = $_[0];
16     my $Largeur = $_[1];
17
18     return  ($Longueur*$Largeur) ;
19 }
```

Les fonctions – valeur retournée

L'opérateur **return** est utilisé pour spécifier la valeur à retourner. Si l'opérateur **return** est absent, la dernière valeur calculée sera la valeur retournée !

Exemple d'une fonction :

```
sub sum_of_fred_and_barney {
        print "Hey, you called the sum_of_fred_and_barney subroutine!\n";
        $fred + $barney; # La valeur retournée !
 }
$fred = 3; $barney = 4;
$wilma = sum_of_fred_and_barney;          # $wilma est égal à 7
print "\$wilma is $wilma.\n";
$betty = 3 * sum_of_fred_and_barney;      # $betty est égal à 21
print "\$betty is $betty.\n";
```

Les fonctions – valeur retournée (2)

Exemple d'une fonction :

La valeur retournée par cette fonction sera 1 ou 0 (dépendamment du succès de la commande print).

```
sub sum_of_fred_and_barney {
        print "Hey, you called the sum_of_fred_and_barney subroutine!\n";
        $fred + $barney; # Cette valeur ne sera pas retournée !
        print "Hey, I'm returning a value now!\n"; # Oops!
}
```

La valeur retournée correspond à la dernière instruction évaluée !

(pas toujours la dernière ligne de la fonction):

```
sub larger_of_fred_or_barney {
        if ($fred > $barney) { $fred; } # on compare les variables globales ici
        else { $barney; }
 }
```

Les paramètres passés avec $_ et @_

Appel:

$n = max(10, 15); # Cette fonction a 2 paramètres

La variable globale par défaut $_ peut être utilisée pour récupérer les valeurs des paramètres:

```
 sub max {
        # À comparer avec la fonction larger_of_fred_or_barney
         if ($_[0] > $_[1]) { $_[0]; }
        else { $_[1]; }
}
```

Le tableau @_ privé à la fonction peut aussi être utilisé pour récupérer la liste des paramètres:

```
sub max {
        my ($m, $n) = @_;           # Définition des variables locales
        if ($m > $n) { $m; } else { $n; }
}
```

Le nombre de paramètres passés à une fonction

Exemple d'une fonction vérifiant le nombre d'arguments:

```
sub max {
        if (@_ != 2) {
                print "WARNING! &max should get exactly two arguments!\n";
                }
        # continuez comme avant ...
}
```

La fonction max permettant n'importe quel nombre de paramètres:

```
$maximum = &max(3, 5, 10, 4, 6);    # Appel de la fonction
sub max {
        my ($max_so_far) = shift @_; # la première valeur est la plus grande
          foreach (@_) {                    # on parcourt toutes les autres valeurs
                if ($_ > $max_so_far) { # on effectue la comparaison
                        $max_so_far = $_; } } }
        $max_so_far;
}
```

Utilisation d’un ampersand (&) à l’appel d’une fonction

Exemple d’une fonction vérifiant le nombre d’arguments:

```
my @cards = shuffle(@deck_of_cards); # & est optionnel: les arguments sont indiqués


&division;          # & n’est pas optionnel – la définition de la fonction est après son appel
sub division {
        print "C’est la division ! \n ";
}
division;           # & est optionnel – la définition de la fonction est avant son appel


sub chomp {
        print "C’est ma fonction chomp ! \n";
}
&chomp;             # Cet ampersand n’est pas optionnel !
```

Retourner une liste d'arguments

Une fonction Perl peut retourner non seulement une variable scalaire, mais une liste de paramètres. Voici un exemple d'une telle fonction:

```
sub list_from_fred_to_barney {
        if ($fred < $barney) {        # On compte en avant $fred à $barney
                $fred..$barney;
        }
        else {                        # On compte en arrière de $fred à $barney
                reverse $barney..$fred;
        }
}
```

Un exemple d'appel de cette fonction:

```
$fred = 11;
$barney = 6;
@c = &list_from_fred_to_barney;     # @c reçois les valeurs (11, 10, 9, 8, 7, 6)
```

Les fonctions : exemple

Exemple complet illustrant l'utilisation des fonctions :

```
1 #!/bin/perl
2
3 use strict; use warnings;
4
5 # fonction sans valeur de retour
6 sub Perimetre{
7     my $Longueur = $_[0];
8     my $Largeur = $_[1];
9
10    print "le perimetre est : " . ($Longueur*$Largeur) . "\n" ;
11 }
12
13 # fonction avec valeur de retour
14 sub Perimetre2{
15    my $Longueur = $_[0];
16    my $Largeur = $_[1];
17
18    return  ($Longueur*$Largeur) ;
19 }
20
21 Perimetre($ARGV[0],$ARGV[1]);
22 print "le perimetre est : " . Perimetre2($ARGV[0],$ARGV[1]);
```

Expressions régulières

Pour retrouver un motif donné (une expression régulière) dans une chaîne de caractères contenue dans la variable $_, on met ce motif entre une paire de slashs (/) comme dans cet exemple:

```
$_ = "yabba dabba doo";
if (/abba/) {
        print "Ça correspond !\n";
}
```

L'expression /abba/ recherche cette sous-séquence de 4 lettres dans la chaine $_. Si la sous-séquence est trouvée, la valeur VRAI est retournée. Le fait que l'expression est trouvée plus qu'une fois ne fait pas de différence. Sinon, ça retourne FAUX.

Parce que la valeur retournée par les expressions régulières est VRAI ou FAUX, on les retrouve généralement dans les expressions conditionnelles **if** et boucles **while**.

/coke\tsprite/ - correspond à 'coke', une tabulation et 'sprite'.

Les métacaractères

Un certain nombre de métacaractères ont une signification spéciale dans les expressions régulières.

Par exemple, le point (.) correspond à tout caractère simple à l'exception d'un saut de ligne (représenté par "\n").

L'expression régulière /bet.y/ va correspondre à betty. Elle va également correspondre à betsy, bet=y ou à bet.y, ou tout autre chaîne commençant par bet suivi d'un caractère quelconque (sauf "\n"), suivi d'un y.

Ça ne correspond pas à bety ou betsey par contre.

Un backslash en avant d'un métacaractère enlève ses propriétés. Donc, l'expression régulière /3\.14159/ n'a pas de métacaractères.

Les quantifieurs simples

L’ astérisque (*) signifie la correspondance de 0 ou plus de fois au caractère précédent.

Donc, /fred\t*barney/ correspond à n’importe quel nombre de tabulations entre fred et barney. Ça correspond à "fred\tbarney" avec une tabulation, "fred\t\tbarney" avec deux tabulations ou à "fredbarney" avec 0 tabulations.

Le point correspond à un caractère et .* va donc correspondre à tout caractère apparaissant n’importe quel nombre de fois. Ceci signifie que l’expression /fred.*barney/ correspond à fred, suivi de n’importe quel texte, suivi de barney.

Le signe plus (+) est un autre quantificateur en Perl. Le plus signifie la correspondance de 1 ou plus de fois au caractère précédent.

/fred +barney/ correspond au cas quand fred et barney sont séparés par des espaces (et seulement par des espaces).

Cette expression ne correspond pas à fredbarney !

Le point d’interrogation (?) est le troisième quantificateur de Perl. Il signifie que l’item précédent est optionnel (il peut apparaitre 0 ou 1 fois seulement).

Ainsi, /bamm-?bamm/ correspond à bamm-bamm ou à bammbamm.

Regroupement des motifs

Les parenthèses sont aussi des métacaractères. Comme en mathématiques, les parenthèses (( )) peuvent être utilisées pour regrouper les éléments.

Par exemple, l'expression régulière /fred+/ correspond à des chaînes comme freddddddddd.
Mais l'expression /(fred)+/ correspond à fredfredfred.
De l'autre coté l'expression /(fred)*/ correspond à des chaînes comme "hello, world".

Les références \1, \2, etc. sont utilisées pour indiquer le contenu se trouvant dans la première, la deuxième, etc. paires de parenthèses.

```
$_ = "yabba dabba doo";
if (/y(….) d\1/) {
        print "Ça correspond à l'élément entre y et  d !\n",
}

 if (/y(.)(.)\2\1/) {
        print "Ça correspond à abba !\n",
}
```

Les alternatives (1)

La barre verticale (|), ou l'opérateur « ou », indique que soit la partie gauche, soit la partie droite est recherchée.

Donc, l'expression:

/fred|barney|betty/

acceptera des chaînes de caractères mentionnant fred, ou barney, ou betty.

Maintenant nous pouvons créer les expressions régulières comme:

/fred(|\t)+barney/

qui correspond à fred et barney séparés pas les espaces, les tabulations ou par une combinaison des deux. Il doit y avoir au moins un espace ou une tabulation entre les 2 noms.

Aussi les expressions suivantes sont possibles:

```
$some_input = "C'est une \t chaine de \n        caracteres";

my @args = split (/\s+/, $some_input);

# @args = ("C'est", "une", "chaine", "de", "caracteres");
```

Les alternatives (2)

Si on veut que les caractères entre fred et barney soient tous les mêmes, on écrit l'expression régulière comme suit:

/fred(+|\t+)barney/.

Dans ce cas, les séparateurs seront seulement des espaces ou seulement des tabulations.

L'expression :

/fred (and|or) barney/

acceptera toute chaîne de caractères contenant les chaînes:

"fred and barney" ou "fred or barney".

On pourrait retrouver les mêmes 2 chaînes de caractères avec l'expression:

/fred and barney|fred or barney/,

mais c'est moins efficace.

Les classes des caractères (1)

Une classe des caractères est une liste des caractères contenus entre les crochets [ ]. Une telle expression recherche *n'importe quel caractère* de cette liste.

Il est important qu'un seul caractère, parmi ceux indiqués, est recherché.

Par exemple, la classe des caractères [abcwxyz] correspond à une des lettres a, b, c, w, x, y ou z de cette classe.

Il est pratique de spécifier un intervalle des caractères en utilisant un trait d'union (-), par exemple:

[a-cw-z]

[a-zA-Z] – définit la classe des 52 lettres de l'alphabet.

La classe [\000-\177] – définit n'importe quel caractère ASCII (sur 7 bits).

Les classes des caractères (2)

Par exemple nous pouvons écrire un code comme suit:

```
$_ = "The HAL-9000 requires authorization to continue.";

if (/HAL-[0-9]+/) {

        print "The string mentions some model of HAL computer.\n";

 }
```

Parfois, il est plus facile de spécifier un caractère à omettre qu'un groupe de caractères à rechercher à l'intérieur d'une classe de caractères. Le chapeau ("^") au début de la classe des caractères inverse le critère de recherche : au lieu de chercher les caractères de la classe, on recherche tout caractère qui n'appartient pas à cette classe. Par exemple:

[^def] – on recherche tout caractère sauf d, e et f.

[^n\-z] – on recherche tout caractère sauf n, un trait d'union et z.

Mais le premier trait d'union dans /HAL-[0-9]+/ n'a pas besoin d'un backslash parce que le trait d'union n'est pas un caractère spécial ici.

Les raccourcis

Certaines classes de caractères en Perl apparaissent tellement fréquemment qu'elles ont des raccourcis. Par exemple, la classe définissant tous les chiffres, [0-9], peut être spécifiée comme \d. Ainsi, l'expression régulière concernant HAL pourrait être réécrit comme suit:

/HAL-\d+/.

Le raccourci \w (appelé caractère de type « word ») est équivalent à [A-Za-z0-9_].

Ce raccourci suppose cependant que les mots sont composés des lettres, chiffres et soulignés (des apostrophes, traits d'union et lettres accentuées ne sont pas incluses dans cette classe).

Bien sur, \w ne recherche pas un mot au complet, mais seulement un seul caractère de la classe « word ». Pour rechercher un mot au complet on utilise le quantificateur (+). Une expression comme /fred \w+ barney/ recherchera fred, un espace, un seul mot, un espace et puis barney.

Il est aussi pratique d'avoir un seul caractère pour tous les caractères d'espacement.

Le raccourci \s s'en occupe. C'est le même que de définir la classe [\f\t\n\r] - incluant les 5 caractères: saut de page, tabulation, saut de ligne, retour de chariot et espace.

Les négation des raccourcis

Parfois, on a besoin des opposés des 3 raccourcis, [\d], [\w] et [\s],

soient : [^\d], [^\w] et [^\s],

On les obtient en utilisant les majuscules : \D, \W et \S, qui signifient respectivement un caractère qui n'est pas un chiffre, n'est pas de type « word » et n'est pas un caractère d'espacement.

Nous pouvons par exemple utiliser le code:

/[\dA-Fa-f]+/

pour rechercher des nombres hexadécimaux (à la base de 16), qui utilisent les lettres ABCDEF (ou les mêmes lettres minuscules) dans leur écriture.

Une autre classe de caractères composée est [\d\D], ce qui signifie :

« tout chiffre ou tout non-chiffre »

voulant dire : n'importe quel caractère.

Expressions régulières (suite 1)

Jusqu'à maintenant nous avons toujours écrit des expressions régulières en utilisant une paires de slashs, comme /fred/. En fait, c'est un raccourci de l'opérateur m// (pattern match) de Perl. Ce raccourci fonctionne de la même façon que l'opérateur qw// vu précédemment.

Ainsi, nous pourrions définir la même expression régulière en utilisant:

/fred/, m(fred), m<fred>, m{fred} ou m[fred].

Pour faire une expression régulière insensible aux lettres minuscules/majuscules, de façon qu'elle correspondra aux trois:

fred, Fred et FRED,

on utilise le modificateur /i:

```
print "Would you like to play a game? ";
chomp($_ = <STDIN>);
if (/yes/i) { # insensible aux minuscules/majuscules
        print "In that case, I recommend that you go bowling.\n";
}
```

Expressions régulières (suite 2)

L'opérateur de recherche des motifs dans une chaîne de caractères (=~), *binding operator* en anglais, est souvent utilisé en Perl.

Par défaut la recherche des correspondances définies dans des expressions régulières se fait dans la variable $_. L'opérateur (=~) dit à Perl de rechercher dans une chaîne à gauche le motif défini par l'expression régulière à droite. Par exemple:

```
my $some_other = "I dream of betty rubble.";
if ($some_other =~ /rub/)
        { print "Aye, there's the rub.\n"; }
```

Une expression régulière peut inclure des variables incorporées. Par exemple:

```
my $what = "Larry";
while (<>) {
                if (/$what/) {
                        print "We saw $what in $_";
                }
}
```

Analyse d’un rapport de BLAST – un exemple

Une exemple d’un rapport BLAST :

```
. . .
gb|AC005288.1|AC005288  Homo sapiens chromosome 17, clone hC...   268  2e-68
gb|AC008812.7|AC008812  Homo sapiens chromosome 19 clone CTD...   264  3e-67
gb|AC009123.6|AC009123  Homo sapiens chromosome 16 clone RP1...   262  1e-66
emb|AL137073.13|AL137073  Human DNA sequence from clone RP11...   260  5e-66
gb|AC020904.6|AC020904  Homo sapiens chromosome 19 clone CTB...   248  2e-62
>gb|AC007421.12|AC007421 Homo sapiens chromosome 17, clone hRPC.1030_O_14,
complete sequence
Query: 3407 accgtcataaagtcaaacaattgtaacttgaaccatcttttaactcaggtactgtgtata 3466
             ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Sbjct: 1366 accgtcataaagtcaaacaattgtaacttgaaccatcttttaactcaggtactgtgtata 1425
Query: 3467 tacttacttctccccctcctctgttgctgcagatccgtgggcgtgagcgcttcgagatgt 3526
             ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Sbjct: 1426 tacttacttctccccctcctctgttgctgcagatccgtgggcgtgagcgcttcgagatgt 1485
Query: 3527 tccgagagctgaatgaggccttggaactcaaggatgcccaggctgggaaggagccagggg 3586
             ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Sbjct: 1486 tccgagagctgaatgaggccttggaactcaaggatgcccaggctgggaaggagccagggg 1545


Query: 3587 ggagcagggctcactccaggtgagtgacctcagcccttcctggccctactcccctgcct 3646
             ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Sbjct: 1546 ggagcagggctcactccaggtgagtgacctcagcccttcctggccctactcccctgcct 1605
Query: 3647 tcctaggttggaaagccataggattccattctcatcctgccttcatggtcaaaggcagct 3706
. . .
```

Analyse d’un rapport de BLAST – un exemple (2)

On a besoin d’écrire un programme qui parcourt le fichier de rapport de BLAST et qui retourne le nombre d’occurrences des sous-séquences suivantes (dans les lignes "Query" et "Sbjct") :

- 'gtccca'
- 'gcaatg'
- 'cagct'
- 'tcggga'
- plus des données manquantes (représentées par des ‘-’ dans des séquences).

Les informations générales contenues au début d’un rapport BLAST (les 7 premières lignes) doivent être ignorées.

Le programme d’analyse pourrait être divisé en 3 parties, qui sont comme suit:

- L’entrée des données et leur préparation pour l’analyse;
- Le parcourt des données et la recherche des fragments spécifiés;
- La compilation des résultats et leur stockage dans le fichier *report.txt.*

Analyse d'un rapport de BLAST – un exemple (3)

Partie 1 du programme

La déclaration des variables:

```
#!/usr/bin/perl
# Search through a large datafile, looking for particular sequences

use strict;

my $REPORT_FILE = "report.txt";
my $blast_file  = $ARGV[0] || 'blast.dat';

unless ( -e $blast_file ) {
  die "$0: ERROR: missing file: $blast_file";
}
```

La lecture des séquences dans les variables:

```
# First, slurp all the Query sequences into one scalar. Same for the
# Sbjct sequences.
my ($query_src, $sbjct_src);

# Open the blast datafile and end program (die) if we can't find it
open (IN, $blast_file) or die "$0: ERROR: $blast_file: $!";

# Go through the blast file line by line, concatenating all the Query and
# Sbjct sequences.
while (my $line = <IN>) {
  chomp $line;
  print "Processing line $.\n";
```

Analyse d'un rapport de BLAST – un exemple (4)

Partie 2 du programme

La séparation des données en tableau @words à 4 colonnes:

```
    my @words = split /\s+/, $line;
    if ($line =~ /^Query/) {
        $query_src .= $words[2];
    } elsif ($line =~ /^Sbjct/) {
        $sbjct_src .= $words[2];
    }
}

# We've now read the blast file, so we can close it.
close IN;
```

Le parcourt des séquences et l'analyse des données:

```
# Now, look for these given sequences...
my @patterns = ('gtccca', 'gcaatg', 'cagct', 'tcggga', '-');

# ...and when we find them, store them in these hashes
my (%query_counts, %sbjct_counts);

# Search and store the sequences
foreach my $pattern (@patterns) {
   while ( $query_src =~ /$pattern/g ) {
       $query_counts{ $pattern }++;
   }
   while ( $sbjct_src =~ /$pattern/g ) {
       $sbjct_counts{ $pattern }++;
   }
}
```

Analyse d'un rapport de BLAST – un exemple (6)

Partie 5 du programme

La création d'un fichier de sortie (fichier report.txt):

```
# Create an empty report file
open (OUT, ">$REPORT_FILE") or die "$0: ERROR: Can't write $REPORT_FILE";

# Print the header of the report file, including
# the current date and time
print OUT "Sequence Report\n",


    "Total length of 'Query' sequence: ",
    length $query_src, " characters\n", "Results for 'Query':\n";

# Print the Query matches
foreach my $key ( sort @patterns ) {
    print OUT "\t'$key' seen $query_counts{$key}\n";
}

print OUT "\nTotal length of 'Sbjct' sequence: ",
    length $sbjct_src, " characters\n", "Results for 'Sbjct':\n";

# Print the Sbjct matches
foreach my $key ( sort @patterns ) {
    print OUT "\t'$key' seen $sbjct_counts{$key}\n";
}

close OUT;
```

Analyse d'un rapport de BLAST – un exemple (6)

Un exemple de sortie du programme ci-dessus (fichier report.txt):

```
Sequence Report

Total length of 'Query' sequence: 1115 characters
Results for 'Query':
        '-' seen 7
        'cagct' seen 11
        'gcaatg' seen 1
        'gtccca' seen 6
        'tcggga' seen 1

Total length of 'Sbjct' sequence: 5845 characters
Results for 'Sbjct':
        '-' seen 12
        'cagct' seen 2
        'gcaatg' seen 6
        'gtccca' seen 1
        'tcggga' seen 6
```

Cet exemple est tiré du livre: *Developing Bioinformatics Computer Skills*
Cynthia Gibas et Per Jambeck, O'Reilly, 2001

BioPerl : Contenu

Fonctionnalités générales

BioPerl est une bibliothèque comportant un ensemble de modules Perl facilement utilisables dans des programmes ou scripts pour la manipulation et le traitement de données biologiques, plus particulièrement de celles issues de la génomique, la génétique ou de l'analyse du transcriptome. Bioperl fournit des outils pour l'indexation, l'interrogation et l'extraction de banques de données publiques ou locales, des filtres pour un très grand nombre de sorties de programmes usuels en bioinformatique (blast, clustalw, hmmer, est2genome, ...) permettant ainsi de réaliser simplement des enchaînements de programmes (pipelines).

Contexte d'utilisation

Le contexte d'utilisation de BioPerl est très large. On peut utiliser Bioperl dans des simples scripts pour extraire de l'information des banques de données ou analyser des résultats d'un programme, d'enchaînement de programmes (pipeline) ou pour des projets plus complexes nécessitant de la modélisation de données biologiques.

Un bon tutoriel pour les débutants

Disponible sur: http://www.bioperl.org/wiki/HOWTO:Beginners

BioPerl : Installation

Seulement ActivePerl versions **>= 5.8.8.819** sont supportées par BioPerl.
On installe ActivePerl d'abord donc. Puis, les instructions à suivre sont comme suit (voir aussi le site: http://www.bioperl.org):

1) Démarrez Perl Package Manager GUI (installé par ActivePerl) depuis le menu Démarrer.

2) Allez dans Edit >> Preferences et cliquez sur Repositories tab. Ajoutez une nouvelle "repository" pour chacun des paquets ci-dessous (pour ActivePerl 5.10 et >):

Nom	**Perl 5.10**
BioPerl-Regular Releases	**http://bioperl.org/DIST**
BioPerl-Release Candidates	**http://bioperl.org/DIST/RC**
Kobes	**http://cpan.uwinnipeg.ca/PPMPackages/10xx/**
Bribes	**http://www.Bribes.org/perl/ppm**
Trouchelle	**http://trouchelle.com/ppm10**
Tcool	NA

3) Sélectionnez View >> All Packages.

4) Dans la boite de recherche, tapez bioperl.

5) Cliquez avec le bouton droit sur la version de bioperl à installer.

6) Cliquez sur la flèche verte (Run marked actions) pour compléter l'installation.

BioPerl : quelques exemples rapides (1)

Bioperl est une collection d'objets et de méthodes définis et correspondants à des éléments courants utilisés en bioinfo. Par exemple, l'objet séquence :

```
use Bio::Seq;
my $seqobj = Bio::Seq->new( -seq => 'atcgatcg', -id => 'GeneFragment-12', -
                                    accession_number => 'X78121', -alphabet => 'dna' );
```

Pour afficher, à tout moment, les attributs d'un objet :

```
use Data::Dumper;                    #/usr/lib/perl5/5.6.0/i386-linux/Data/Dumper.pm
print Dumper($seqobj);
```

Ces objets pourront être manipulés par des méthodes spécifiques :

```
# la méthode seq() permet d'accéder à la valeur séquence de l'objet $seqobj
print "Sequence : ",$seqobj->seq(),"\n";
if (! $seqobj->validate_seq($seq_str) ) {
        print "la séquence $seq_str n est pas valable pour : ", ref($seq), "\n";
 }
print "Nouvelle séquence : ",$seqobj->seq($seq_str),"\n";
```

BioPerl : quelques exemples rapides (2)

Écriture de l'objet séquence dans un fichier en spécifiant le format souhaité, le fichier généré s'appellera GeneFragment-12.genbank :

```
use Bio::SeqIO;

$outfileformat="genbank";
$outfile = $seqobj->id().".".$outfileformat;
my $seq_out = Bio::SeqIO->new('-file' => ">$outfile", '-format' => $outfileformat);
$seq_out->write_seq($seqobj);
```

Bioperl possède des méthodes pouvant se connecter directement aux banques de données, par exemple :

```
# get a sequence from a database (assumes internet connection)
use Bio::Perl qw( get_sequence );

$seq_object = get_sequence('swissprot',"ROA1_HUMAN");

print "Description : ",$seq_object->desc(),"\n";
print "Sequence ID : ",$seq_object->id() ,"\n";
print "Sequence : ",$seq_object->seq() ,"\n";
```

BioPerl : comment créer une séquence (1)

Tous les objets de BioPerl sont créés par des modules spécifiques. Donc, pour créer un nouvel objet, il faudra dire à Perl quel module doit être utilisé.

```
#!/bin/perl –w
use Bio::Seq;
```

La ligne ci-dessus dit à Perl d'utiliser le module appelé "Bio/Seq.pm". Nous allons utiliser le module Bio::Seq pour créer des objets Bio::Seq. Le module Bio::Seq est l'un des modules clef en BioPerl. Un objet de type séquence - Bio::Seq, ou "Sequence object", ou "Seq object", contient une séquence unique avec les noms, identificateurs et propriétés associés.

```
#!/bin/perl -w
use Bio::Seq;
$seq_obj = Bio::Seq->new(  -seq => "aaaatgggggggggggccccgtt",
                           -alphabet => 'dna' );
```

La variable $seq_obj contient l'objet en question. Un objet de type séquence Bio::Seq peut être créé manuellement, comme ci-haut. De tels objets sont aussi créés automatiquement dans plusieurs opérations de BioPerl, par exemple lorsqu'on aligne les séquences multiples contenues dans des fichiers ou on scanne les rapports de BLAST.

BioPerl : comment créer une séquence (2)

Quand un nouvel objet de type séquence est créé, on utilise la méthode (i.e., fonction) new(). La syntaxe de cette commande est très typique pour BioPerl:
le nom de l'objet ou de la variable,
le nom du module,
le symbole ***->***
le nom de la méthode (new),
certains arguments comme ***–seq****,*
le symbole ***=>***
et puis la valeur elle-même, comme ***aaaatgggggggggggccccgtt****.*
Par exemple le module Bio::Seq peut accéder à la méthode appelée seq() qui retourne la séquence de l'objet de type Bio::Seq. On peut le faire de cette façon:

```
#!/bin/perl –w
use Bio::Seq;
$seq_obj = Bio::Seq->new(-seq => "aaaatgggggggggggccccgtt", -alphabet => 'dna' );
print $seq_obj->seq;
```

Comme on pourrait y s'attendre, ce code affichera: **aaaatgggggggggggccccgtt**. Le symbole **->** est utilisé quand un objet appelle ses méthodes.

BioPerl : comment créer une séquence (3)

Montrons maintenant un exemple plus réaliste. Ici un objet de type séquence a un identificateur (ID), une description et un type de séquence.

```
#!/bin/perl -w
use Bio::Seq;
$seq_obj = Bio::Seq->new( -seq => "aaaatgggggggggggccccgtt",
                          -display_id => "#12345",
                          -desc => "example 1",
                          -alphabet => "dna" );
print $seq_obj->seq();
```

Ceci est encore un exemple du passage des paramètres à la méthode new().

BioPerl : écriture d'une séquence dans un fichier (1)

Nous avons déjà un objet de type séquence, $seq_obj, et nous allons créer un objet additionnel dont la responsabilité sera de lire et d'écrire dans un fichier. Cet objet sera de type SeqIO (IO signifie - Input-Output). En utilisant le module Bio::Seq, nous sommes capables de lire les fichiers et d'écrire dans les fichiers en utilisant tous les formats de séquence supportés par BioPerl. La façon de créer un objet de type Bio::Seq est très similaire à l'appel de new() utilisé pour créer un objet de type séquence. Notons le symbole > dans le paramètre de -file. Ce caractère indique que nous allons *écrire* dans le fichier nommé "sequence.fasta" (> est également utilisé dans la fonction open() de Perl permettant ouvrir un fichier en écriture). Le paramètre de -format, "fasta", dit à l'objet SeqIO que le fichier créé doit être en format fasta.

```
#!/bin/perl –w
use Bio::Seq;
use Bio::SeqIO;
$seq_obj = Bio::Seq->new(-seq => "aaaatgggggggggggccccgtt",
                                 -display_id => "#12345",
                                 -desc => "example 1",
                                 -alphabet => "dna" );
$seqio_obj = Bio::SeqIO->new(-file => '>sequence.fasta', -format => 'fasta'); );
$seqio_obj->write_seq($seq_obj);
```

BioPerl : écriture d’une séquence dans un fichier (2)

Le fichier "sequence.fasta" comme suit sera créé:

>#12345 example 1
aaaatgggggggggggccccgtt

Montrons maintenant les possibilités de SeqIO: l’exemple ci-dessous présente le fichier créé par le code précédent si on remplace "fasta" par "genbank" comme paramètre de -format.

LOCUS #12345 23 bp dna linear UNK
DEFINITION example 1
ACCESSION unknown
FEATURES Location/Qualifiers
BASE COUNT 4 a 4 c 12 g 3 t
ORIGIN 1 aaaatggggg gggggggcccc gtt
//

BioPerl : lecture des données à partir d'un fichier (1)

Utilisez les méthodes de BioPerl au lieu de la fonction open() de Perl pour ouvrir et scanner les fichiers de séquence !

Lisons le fichier "sequence.fasta" que nous avons créé précédemment en utilisant le module SeqIO. La syntaxe des commandes sera familière:

```
#!/bin/perl –w
use Bio::SeqIO;
$seqio_obj = Bio::SeqIO->new(-file => "sequence.fasta", -format => "fasta" );
$seq_obj = $seqio_obj->next_seq;   # la récupération de la séquence courante
```

La dernière ligne permet de récupérer un objet de type séquence en utilisant la méthode next_seq.
Dans une boucle, on utilise la méthode next_seq comme suit:

```
while ($seq_obj = $seqio_obj->next_seq) {
        # imprimer la séquence
        print $seq_obj->seq,"\n";
}
```

Cette boucle permet de parcourir toutes les séquences du fichier d'entrée.

BioPerl : extraction des données à partir d'une base de données

Une des force de BioPerl est qu'il vous permet d'extraire des données séquentielles de plusieurs sources d'information: fichiers, bases de données éloignées, bases de données locales, peu importe leur format. Illustrons cette capacité de BioPerl pour extraire des informations de **Genbank**. Nous allons extraire tout d'abord un objet de type séquence en utilisant le module Bio::DB::GenBank.

```
use Bio::DB::GenBank;
```

Nous pourrions aussi interroger **SwissProt** (Bio::DB::SwissProt), **GenPept** (Bio::DB::GenPept), **EMBL** (Bio::DB::EMBL), **SeqHound** (Bio::DB::SeqHound), **Entrez Gene** (Bio::DB::EntrezGene), ou **RefSeq** (Bio::DB::RefSeq) de façon analogique (e.g., "use Bio::DB::SwissProt"). Nous créons maintenant l'objet:

```
use Bio::DB::GenBank;
$db_obj = Bio::DB::GenBank->new;
$seq_obj = $db_obj->get_Seq_by_acc("A12345");
```

Le paramètre passé à la méthode get_Seq_by_acc est un numéro d'accession de **Genbank** ("A12345"). Nous pourrions aussi utiliser les méthodes: get_Seq_by_version en utilisant un numéro d'accession avec version (e.g. "A12345.2") ou la méthode get_Seq_by_id en utilisant un identificateur (e.g., 2).

BioPerl : extraction de séquences multiples à partir des bases de données (1)

L'exemple suivant montre une autre façon (plus concrète) d'extraire des données de **Genbank**. Le module **Bio::DB::Query::GenBank** sera utilisé. Nous voulons extraire toutes les entrées de *Genbank Nucleotide* mentionnant l'organisme *Arabidopsis topoisomerases* :

```
use Bio::DB::Query::GenBank;

$query = "Arabidopsis[ORGN] AND topoisomerase[TITL] and 0:3000[SLEN]";
$query_obj = Bio::DB::Query::GenBank->new(-db => 'nucleotide', -query => $query );
```

Notons que ce type de requêtes est seulement possible avec GenBank et la version de BioPerl >= 1.5. Les requêtes vers d'autre banques de données (e.g., **SwissProt** ou **EMBL**) sont limitées à des numéros d'accession et à des identificateurs.

Ceci est un autre exemple de requête pour extraire les EST de *Trypanosoma brucei* :

```
$query_obj = Bio::DB::Query::GenBank->new(
                    -query =>'gbdiv est[prop] AND Trypanosoma brucei [organism]',
                    -db => 'nucleotide' );
```

BioPerl : extraction de séquences multiples à partir des bases de données (2)

Cet exemple montre comment construire un objet de type requête (sans extraire la séquence encore). Pour ce faire, on crée d'abord un *objet de type base de données*, qui est un objet permettant d'extraire plusieurs *objets de type séquence* :

```
use Bio::DB::GenBank;
use Bio::DB::Query::GenBank;

$query = "Arabidopsis[ORGN] AND topoisomerase[TITL] and 0:3000[SLEN]";
$query_obj = Bio::DB::Query::GenBank->new(-db => 'nucleotide', -query => $query );

$gb_obj = Bio::DB::GenBank->new;
$stream_obj = $gb_obj->get_Stream_by_query($query_obj);

while ($seq_obj = $stream_obj->next_seq) {
        # impression des informations sur l'objet courant
        print $seq_obj->display_id, "\t", $seq_obj->length, "\n";
}
```

L'idée ici est d'utiliser le flux de données (*stream*) quand on essaye d'extraire une série d'objets de type séquence.

BioPerl: Objet de type Séquence : Méthodes disponibles (1)

Nom	Retourne	Exemple	Note
new	sequence object	$so = Bio::Seq->new(-seq => "MPQRAS")	create a new one, see Bio::Seq for more
seq	sequence string	$seq = $so->seq	get or set the sequence
display_id	identifier	$so->display_id("NP_123456")	get or set an identifier
primary_id	identifier	$so->primary_id(12345)	get or set an identifier
desc	description	$so->desc("Example 1")	get or set a description
accession_number	identifier	$acc = $so->accession_number	get or set an identifier
length	length, a number	$len = $so->length	get the length
alphabet	alphabet	$so->alphabet('dna')	get or set the alphabet ('dna','rna','protein')
subseq	sequence string	$string = $seq_obj->subseq(10,40)	Arguments are start and end
trunc	sequence object	$so2 = $so1->trunc(10,40)	Arguments are start and end
is_circular	Boolean	if $so->is_circular { # }	get or set this value
revcom	sequence object	$so2 = $so1->revcom	Reverse complement

BioPerl: Objet de type Séquence : Méthodes disponibles (2)

Nom	Retourne	Exemple	Note
translate	protein sequence object	$prot_obj = $dna_obj->translate	See the Bioperl Tutorial
species	species object	$species_obj = $so->species	See the Bio::Species
seq_version	version, if available	$so->seq_version("1")	get or set a version
keywords	keywords, if available	@array = $so->keywords	get or set keywords
namespace	namespace, if available	$so->namespace("Private")	get or set the name space
authority	authority, if available	$so->authority("FlyBase")	get or set the organization
get_secondary_accessions	array of secondary accessions, if available	@accs = $so->get_secondary_accessions	get other identifiers
division	division, if available	$div = $so->division	get division (e.g. "PRI")
molecule	molecule type, if available	$type = $so->molecule	get molecule (e.g. "RNA", "DNA")
get_dates	array of dates, if available	@dates = $so->get_dates	get dates
pid	pid, if available	$pid = $so->pid	get pid

BioPerl : BLAST (1)

BLAST est certainement le programme de comparaison et d'alignement de séquence le plus populaire. Premièrement, il faut installer un algorithme de **BLAST**, connu aussi comme *blastall*, sur votre machine (les versions de *blastall* pour toutes les plateformes sont disponibles sur le site de NCBI). Mentionnons que l'exemple ci-dessous suppose que le programme *formatdb* a été utilisé pour indexer le fichier de base de données "db.fa".

Comme d'habitude, nous commençons par choisir un module approprié. Dans ce cas, c'est **Bio::Tools::Run::StandAloneBlast**. On spécifie les paramètres à utiliser par *blastall* en les affectant, dans cet exemple, aux éléments du tableau @params (tout autre nom de tableau pourrait aussi être utilisé).

```
use Bio::Tools::Run::StandAloneBlast;

@params = (program => 'blastn', database => 'db.fa');
```

BioPerl : BLAST (2)

Nous voulons créer un *objet de type BLAST* et puis passer un *objet de type séquence* à cet objet de type BLAST. Cet objet de type séquence sera utilisé dans la requête:

```
use Bio::Seq;
use Bio::Tools::Run::StandAloneBlast;

@params = (program => 'blastn', database => 'db.fa');

$blast_obj = Bio::Tools::Run::StandAloneBlast->new(@params);

$seq_obj = Bio::Seq->new(-id =>"test query", -seq=>
            "TTTAAATATATTTTGAAGTATAGATTATATGTT");

$report_obj = $blast_obj->blastall($seq_obj);
$result_obj = $report_obj->next_result;
print $result_obj->num_hits;
```

En appelant la méthode *blastall* nous exécutons BLAST, créons le fichier rapport et scannons le contenu de ce fichier. Toutes les données d'un rapport BLAST sont représentées en tant que des *objets de type rapport*. Ces objets peuvent ensuite être accédés ou imprimés de façons différentes.

BioPerl : BLAST (3)

Ici est un exemple d'utilisation du module **SearchIO** pour extraire les données d'un rapport **BLAST** :

```
use Bio::SearchIO;

$report_obj = new Bio::SearchIO(-format => 'blast', -file => 'report.bls');

while( $result = $report_obj->next_result ) {
        while( $hit = $result->next_hit ) {
                while( $hsp = $hit->next_hsp ) {
                        if ( $hsp->percent_identity > 75 ) {
                           print "Hit\t", $hit->name, "\n", "Length\t",
                                $hsp->length('total'), "\n", "Percent_id\t",
                                $hsp->percent_identity, "\n";
                         }
                }
        }
}
```

Ce code imprime les détails sur les séquences pour lesquelles le pourcentage de correspondance (*match*) est supérieur à 75%.

Exercices (1)

1. Écrivez un programme Perl permettant de lire un fichier texte contenant des nombres entiers situés sur les lignes séparées et d'afficher seulement les nombres pairs.

2. Écrivez une fonction appelée *total* qui retourne le total de la liste des nombres passée en paramètre. Voici un exemple d'appel d'une telle fonction:

```
my @fred = qw { 1 3 5 7 9 };

my $fred_total = total(@fred);

print "Le total de \@fred est: $fred_total.\n";

print "Entrez les nombres sur les lignes séparées: ";

my $user_total = total(<STDIN>);

print "Le total de ces nombres est: $user_total.\n";
```

3. Écrivez une fonction appelée *above_average* qui prend en argument la liste des nombres et retourne ceux qui sont supérieurs à la valeur moyenne.

Exercices (2)

1) Faites un programme qui imprime chaque ligne de son entrée mentionnant fred. Est-ce que ça marche pour les chaînes de caractères contant Fred, frederick ou Alfred ? Modifiez le programme pour que ça accepte Fred également.

2) Écrivez un programme qui imprime toute la ligne de l'entrée qui mentionne wilma et fred (les deux).

3) En utilisant la librairie Bio::Perl :

   a) Écrivez un programme récupérant un *objet de type séquence* de **Genbank** avec le numéro d'accession *J01673*.

   b) Affichez à l'écran les valeurs de l'objet récupéré retournées par les méthodes suivantes: *display_id, desc, display_name, primary_id, seq_version, keywords, length, seq* et *molecule.*

   c) Enregistrez les informations sur l'objet dans un fichier texte au format *Fasta*.