# Chapter 6

# The CGI and bioperl Modules

## 6.1 Introduction to chapter 6

This chapter will mostly concern modules!. One specific module (`CGI.pm`) and one project (BioPerl project) containing a lot of modules will be studied in more detail. The use of these modules requires some knowledge about references, objects and how to invoke a method, so section 5.3 is important here. The following list summarizes this chapter.

- Numerical Perl
- CGI.pm
- BioPerl project (part 1)
- BioPerl project (part 2)

There will not be enough time to learn everything about the different modules. In fact we will only be able to very briefly look into the capabilities that they offer. The lectures will consist of a variety of examples that show how to use the modules. More detailed descriptions of the modules can be found in various reference documents that will be handed out or that can be accessed via your computer. Before we start with CGI and BioPerl we will look at Perl from a numerical point of view.

## 6.2 Numerical Perl

The purpose of this section is:

- Demonstration that Perl can be used as a programming tool in applications beyond retrieving/manipulating text data.

We will look into matrices, random numbers, a simple statistics module and PDL.

### 6.2.1 Matrices

A $NxM$ matrix $\mathcal{A}$ is a set of numbers arranged in a rectangular scheme.

$$\mathcal{A} = \begin{pmatrix} a_{11} & a_{12} & ... & a_{1M} \\ a_{21} & a_{22} & ... & a_{2M} \\ . & & & \\ . & & & \\ . & & & \\ a_{N1} & a_{N2} & ... & a_{NM} \end{pmatrix}$$

The numbers $a_{jk}$ are matrix elements and the first index $j$ gives the row number while the second index $k$ gives the column number. A square matrix has $N = M$. How do we store a matrix i Perl? The simplest approach is to create a two-dimensional array or an array of arrays to be more precise. Remember the section about references and then you will not be surprised that an array of arrays is nothing but an array of references to other arrays. Below is an example of a 4x5 matrix of integers stored in the array `@A`.

```
@A = (
      [2, 5, 6, -7, 9],
      [1, 4, -6, 7, 10],
      [0, 2, 3, 3, 0],
      [1, 1, 4, 2, 8],
      );
```

The overall list is enclosed by parentheses, not square brackets, because we are assigning a list and not a reference to a list. The square brackets within the list are anonymous arrays of integers. Now that we have created a matrix, how do we access the elements? One can use the arrow operator,

```
print "$A[2]->[1] "; print "$A[3]->[4]\n";
```

which would print 2 8 (make sure you understand this). But the simplest way is to write,

```
print "$A[2][1] ";
print "$A[3][4]\n";
```

Remember that each element in the array `@A` is a reference to another array, this means that you can easily access each row of the matrix A. The example below print the rows of the matrix A.

matrix1.pl†

```
#! /usr/bin/perl -w
use strict;

my @A = (
```

```
5           [2, 5, 6, -7, 9],
6           [1, 4, -6, 7, 10],
7           [0, 2, 3, 3, 0],
8           [1, 1, 4, 2, 8],
9           );
10  foreach my $rowref (@A) {
11      print "@{$rowref}\n";
12  }
```

matrix1.pl†

In most of the cases, however, you have data (represented as matrices) stored as files somewhere on you computer. The task is then to read such a matrix into your Perl program. Assume that you have a file called "matrix.dat" and that the numbers in "matrix.dat" are arranged in space-separated columns and one row per line. A straight-forward way to store "matrix.dat" in the Perl array `@mat` is given by the below code.

matrix2.pl

```
1   #! /usr/bin/perl -w
2   use strict;
3
4   # Open the external file
5   open my $MAT, '<', './matrix.dat';
6
7   my @A;
8   while ( <$MAT> ) {
9
10      my @row = split ' ' ;
11      chomp @row;
12      push @A, \@row;
13  }
14  close $MAT;
15
16  foreach my $rowref ( @A ) {
17      print "@{$rowref}\n";
18  }
```

matrix2.pl

Note that this example is not robust in the sense that it does not check that all rows contain the same number of elements and possible blank lines are not ignored. A somewhat more robust version of "matrix2.pl" is given below.

matrix2r.pl

```
1   #! /usr/bin/perl -w
2   use strict;
3
4   # Open the external file
5   open my $MAT, '<', './matrix.dat';
6   my @mat;
7   my $nrow = 0;
8   my $ncol0;
9   while ( <$MAT> ) {
10
11      next if ( /^\s*$/ );
12      my @row = split ' ';
```

```
13      chomp @row;
14      push @mat, \@row;
15
16      if ( $nrow == 0 ) {
17          $ncol0 = scalar @row;
18      } else {
19          die "Not a matrix\n" unless ( scalar @row == $ncol0 );
20      }
21      $nrow++;
22  }
23  close $MAT;
24  print "Found a $nrow X $ncol0 matrix.\n";
```

matrix2r.pl

Once we have the matrix stored in an array we can start to use it in calculations. Below is a simple example of computing the *trace* of a square matrix `@A`. Can you figure out what the trace is?

matrix3.pl

```
1   #! /usr/bin/perl -w
2   use strict;
3
4   # Define a small square matrix
5   my @A = (
6            [2, 5, 6, -7, 9],
7            [1, 4, -6, 7, 10],
8            [0, 2, 3, 3, 0],
9            [1, 1, 4, 2, 8],
10           [4, 1, -5, 2, 22],
11           );
12  my $N = scalar @A;
13  my $trace = 0;
14  for ( my $j = 0; $j < $N; $j++ ) {
15      $trace += $A[$j][$j];
16  }
17  print "The trace is $trace\n";
```

matrix3.pl

### 6.2.2 Random Numbers

There is a built-in function in Perl called `rand` that produces (a flat) distribution of random numbers between 0 and 1 if called without an argument. Here follow some examples:

- `my $rnd = rand;` will produce a random number in the interval `0 < $rnd < 1`
- `my $rnd = rand 10;` will produce a random number in the interval `0 < $rnd < 10`
- If we want to produce random integers we can use the `int` function to get the integer part. `my $rnd = int rand 10;` will produce a random integer between 0 and 9.
- `my $rnd = int( rand $a ) + $b;` will produce a random integer in the interval `$b < $rnd < ($a+$b-1)`

If we want more "fancy" random numbers we can of course write a program that produces such random numbers or we can use an existing module. The `Math::Random` module[1] is such a module. The following script will print 10 normally distributed random numbers (mean 1 and standard deviation 2).

rnd1.pl

```
#!/usr/bin/perl -w
use strict;
use Math::Random;

my @rndN = random_normal(10, 1, 2);

print "@rndN\n";
```

rnd1.pl

It is sometimes useful to randomly reshuffle all items of an array. The `Math::Random` module also contains such functions.

rnd2.pl

```
#!/usr/bin/perl -w
use strict;
use Math::Random;

my @lett = 'a'..'z';
my @lettR = random_permutation(@lett);
print "@lett\n";
print "@lettR\n";

my @idxR = random_permuted_index(10);
print "@idxR\n";
```

rnd2.pl

- `random_permutation(@array)` returns `@array`, randomly permuted.
- The above code produced the following output
  `g z o d q l k w h a e b p m u n y j t v x c i s r f`
  which of course will change next time it is run!
- Read more about the `Math::Random` module using perldoc (e.g. `perldoc Math::Random`

### 6.2.3 Simple Statistics Module

It is often useful to be able to compute simple statistical properties of a series of numbers, e.g. average, variance etc. There are a few modules available and we are going to look at the `Statistics::Descriptive` module. Here is an example:

stat1.pl

```
#!/usr/bin/perl -w
use strict;
```

[1] http://search.cpan.org/~grommel/Math-Random-0.71/Random.pm

```
3  use Math::Random;
4  use Statistics::Descriptive;
5
6  my $N = 1000000;
7  my $mean = 0;
8  my $std = 2;
9  my @rndN = random_normal($N, $mean, $std);
10
11 my $stat = Statistics::Descriptive::Full->new();
12 $stat->add_data(@rndN);
13
14 print
15     "Some statistical properties of $N normally distributed random numbers\n",
16     "with mean $mean and standard deviation $std:\n\n",
17     "           Mean: ", $stat->mean(), "\n",
18     "       Variance: ", $stat->variance(), "\n",
19     "        Std dev: ", $stat->standard_deviation(), "\n",
20     "            Min: ", $stat->min(), "\n",
21     "            Max: ", $stat->max(), "\n",
22     "25th Percentile: ", $stat->percentile(25), "\n",
23     "95th Percentile: ", $stat->percentile(95), "\n";
```

stat1.pl

This module has an object oriented design, where the object `$stat` is used throughout the example. Running the above script can produce the following output:

```
1  Some statistical properties of 1000000 normally distributed random numbers
2  with mean 0 and standard deviation 2:
3
4             Mean: 0.000703418313303351
5         Variance: 4.00088577006997
6          Std dev: 2.00022143025965
7              Min: -9.54742409658454
8              Max: 9.58716934834071
9  25th Percentile: -1.34753794542408249999
10 95th Percentile: 3.28915144183336949999
```

### 6.2.4 The Perl Data Language

If there is a need for a lot of numerical calculations in your Perl script, perhaps the *Perl Data Language*[2] (PDL) is the way to go. From the man page of PDL one can read:

> PDL is the Perl Data Language, a Perl extension that is designed for scientific and bulk numeric data processing and display. It extends Perl's syntax and includes fully vectorized, multidimensional array handling, plus several paths for device-independent graphics output.

[2] http://pdl.perl.org/

PDL is fast, comparable and often outperforming IDL and MATLAB in real world applications. PDL allows large N-dimensional data sets such as large images, spectra, etc to be stored efficiently and manipulated quickly.

Here is an example with simple matrix operations:

pdl1.pl

```
#!/usr/bin/perl -w
use strict;
use PDL;

# A 3 by 3 matrix
my $A = pdl [[1,2,3],[-1,0,-3], [5,6,7]];

# An identity matrix of size 3
my $B = identity 3;
print $B;

# Simple addition
my $C = $A + $B;

# Invert the $C matrix
my $D = inv $C;

print $C, $D;
```

pdl1.pl

with the following output,

```

[
 [1 0 0]
 [0 1 0]
 [0 0 1]
]

[
 [ 2  2  3]
 [-1  1 -3]
 [ 5  6  8]
]

[
 [       5.2        0.4       -1.8]
 [      -1.4        0.2        0.6]
 [      -2.2       -0.4        0.8]
]
```

- See the PDL home page `http://pdl.perl.org/` for more information.

### 6.2.5 Follow-up tasks 6-1

For the follow-up task you will need the `matrix.dat` file.

1. Write a Perl program that reads the matrix stored in the file `matrix.dat` and does the following with this matrix (called $\mathcal{A}$) (in this exercise you should not use the `Statistics::Descriptive` or the `PDL` module):
   a. Print the mean value for all columns in $\mathcal{A}$.
   b. Print the mean value for all rows in $\mathcal{A}$.
   c. Find the maximum value in $\mathcal{A}$.
   d. (If you have time) Compute the transpose of $\mathcal{A}$. The transpose matrix $\mathcal{B}$ of $\mathcal{A}$ is constructed by changing rows and columns in $\mathcal{A}$ (i.e. $b_{jk} = a_{kj} \quad \forall j, k$).
2. Use appropriate modules to perform the following (numerical) tasks;
   a. Generate 1000 uniform random numbers between 0 and 1 and calculate the variance of the numbers.
   b. Generate 1000 normal random numbers (mean 0 and std. dev. 1) and calculate the skewness of this distribution.
   c. Generate 1000 random numbers from the exponential distribution (mean 1) and calculate the median of this distribution.
3. Write a small Perl script the generates the first 50 Fibonacci numbers.

## 6.3 `CGI.pm` (Common Gateway Interface)

From the documentation of CGI you can read:

```
CGI.pm is a stable, complete and mature solution for processing and preparing
HTTP requests and responses. Major features including processing form
submissions, file uploads, reading and writing cookies, query string generation
and manipulation, and processing and preparing HTTP headers. Some HTML
generation utilities are included as well.

CGI.pm performs very well in in a vanilla CGI.pm environment and also comes with
built-in support for mod_perl and mod_perl2 as well as FastCGI.

It has the benefit of having developed and refined over 10 years with input from
dozens of contributors and being deployed on thousands of websites. CGI.pm has
been included in the Perl distribution since Perl 5.4, and has become a de-facto
standard.
```

To learn about CGI.pm in more detail there is a reference document available. At the computer the best way is to use the perldoc utility to read about CGI module. Just type `>> perldoc CGI`. Here we will study this module by looking at 5 different examples.

### 6.3.1 Example 1

cgi1.pl

```
#! /usr/bin/perl -w

##### Program description #######
#
# Title:     cgi1.pl
# Author(s): Mattias Ohlsson
# Description:
#   Small example of how to use the CGI module.
#
###################################

use strict;
use CGI;

# Get a new CGI object
my $cgi = new CGI;

print $cgi->header(),
    $cgi->start_html(-title=>'Perl is fun'),
    $cgi->h1('The CGI module in action'),
    'Just a simple sentence',
    $cgi->end_html();
```

cgi1.pl

This Perl program will produce the following output if you run it directly from the command line (e.g. `>> perl cgi1.pl`)

```
Content-Type: text/html; charset=ISO-8859-1

<!DOCTYPE html
        PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
         "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="en-US" xml:lang="en-US">
<head>
<title>Perl is fun</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
</head>
<body>
<h1>The CGI module in action</h1>Just a simple sentence
</body>
</html>
```

However if we point a WWW-browser to this file and access it as an cgi script[3], the result will be different (see figure 6.1).

[3]How to do this will be explained in more detail in the "Follow-up tasks" section


Figure 6.1: The browser looking at cgi1.pl.

### 6.3.2 Example 2

Here is another example of how the CGI module can be used. This script uses four different *form* elements:

1. Text entry fields (`textfield(...)`)
2. Checkbox groups (`checkbox_group(...)`)
3. Popup menus (`popup_menu(...)`)
4. Submit buttons (`submit`)

The form is started and ended with the `start_form` and `end_form` *methods* of the CGI object `cgi`. The `if ($cgi->param())` statement will evaluate to true when the submit button has been pressed.

cgi2.pl

```
1  #! /usr/bin/perl -w
2
3  ##### Program description #######
4  #
5  # Title:     cgi2.pl
6  # Author(s): Mattias Ohlsson
7  # Description:
8  #   Small example of how to use the CGI module.
9  #
10 ###################################
11
12 use CGI;
13 use strict;
14
15 my $cgi = new CGI;
16
17 print $cgi->header,
18     $cgi->start_html(-title=>'A simple form example'),
```

```
19      $cgi->h1('A simple form example'),
20      $cgi->start_form,
21      "What is your name? ", $cgi->textfield('name'), $cgi->p,
22      "What is the combination", $cgi->p,
23      $cgi->checkbox_group(-name=>'words',
24                           -values=>['eenie','meenie','minie','moe'],
25                           -defaults=>['eenie']), $cgi->p,
26      "What is your favorite color? ",
27      $cgi->popup_menu(-name=>'color',
28                       -values=>['red','green','blue','yellow']),$cgi->p,
29      $cgi->submit,
30      $cgi->end_form,
31      $cgi->hr;
32
33  if ( $cgi->param() ) {
34      print "Your name is ",$cgi->em($cgi->param('name')),$cgi->p,
35      "The keywords are: ",$cgi->em(join(", ",$cgi->param('words'))),$cgi->p,
36      "Your favorite color is ",$cgi->em($cgi->param('color')),
37      $cgi->hr;
38  }
39  print $cgi->end_html();
```

cgi2.pl

The result of running this Perl (in a WWW-browser) is shown in figure 6.2. The right figure shows the result after the submit button has been pressed.

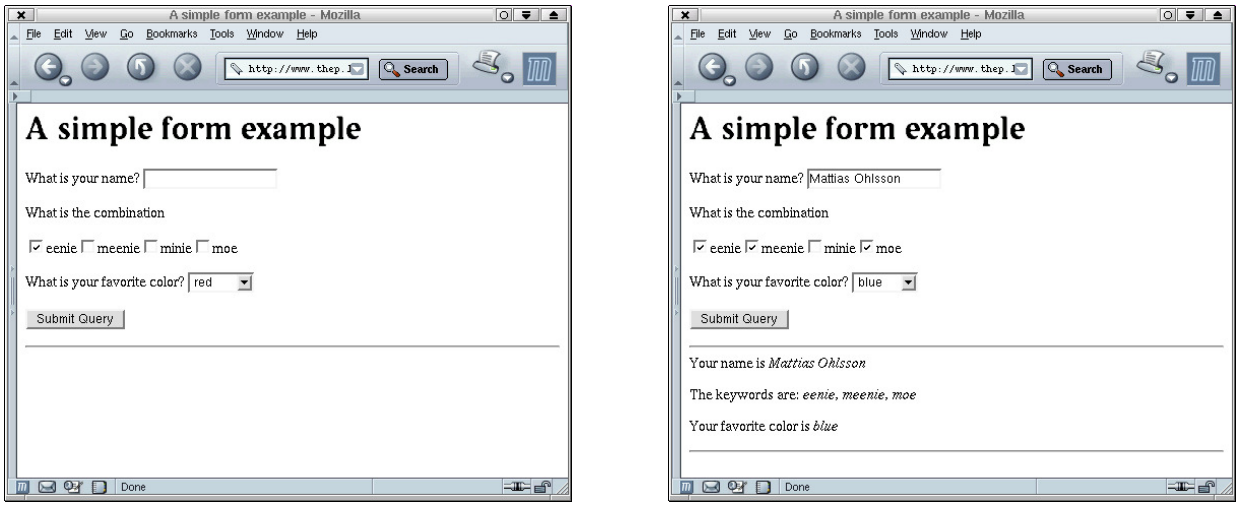


Figure 6.2: The browser looking at cgi2.pl (left figure) and the result after pressing the `submit` button.

### 6.3.3 Example 3

It is of course still possible to use all the utilities in Perl to process the information that is submitted via the forms. The following Perl script takes an amino acid sequence as input and computes the frequencies of the different amino acids in the chain.

cgi3.pl

```
#! /usr/bin/perl -w

##### Program description #######
#
# Title:     cgi3.pl
# Author(s): Mattias Ohlsson
# Description:
#   Small example of how to use the CGI module.
#
###################################

use CGI;
use strict;

my $cgi = new CGI;

print $cgi->header,
    $cgi->start_html(-title=>'A Simple Example'),
    $cgi->h1('Amino Acid Frequencies'),
    $cgi->start_form,
    "Enter or paste an amino-acid sequence? ",
    $cgi->p,
    $cgi->textarea(-name=>'name', rows=>10, columns=>30),
    $cgi->p,
    $cgi->submit,
    $cgi->end_form,
    $cgi->hr;

if ( $cgi->param() ) {
    my $seq = $cgi->param('name');
    Count($seq);
}
print $cgi->end_html();


sub Count {
    my ($seq) = @_;

    my %amino = (
        'Ala','A',  'Cys','C',  'Asp','D',  'Glu','E',
        'Phe','F',  'Gly','G',  'His','H',  'Ile','I',
        'Lys','K',  'Leu','L',  'Met','M',  'Asn','N',
        'Pro','P',  'Gln','Q',  'Arg','R',  'Ser','S',
        'Thr','T',  'Val','V',  'Trp','W',  'Tyr','Y');
    my @amino_keys = keys %amino;

    # Make it into an array
    my @seq2 = split //, $seq;
    my $len = scalar @seq2;

    my %count;
    foreach my $aa ( @seq2 ) {
        chomp $aa;
        $count{uc($aa)} += 1;
```

```
55      }
56      print "Frequencies for the different amino acids", $cgi->br;
57      foreach my $aa ( @amino_keys ) {
58          my $oneletter = $amino{$aa};
59          if ( exists $count{$oneletter} ) {
60              my $frac =  100.0 * $count{$oneletter} / $len;
61              print $cgi->em($aa);
62              printf ": %5.2f%", $frac;
63              print $cgi->br;
64          }
65      }
66
67  } # End of Count
```

cgi3.pl

The result of this Perl script is shown in figure 6.3.

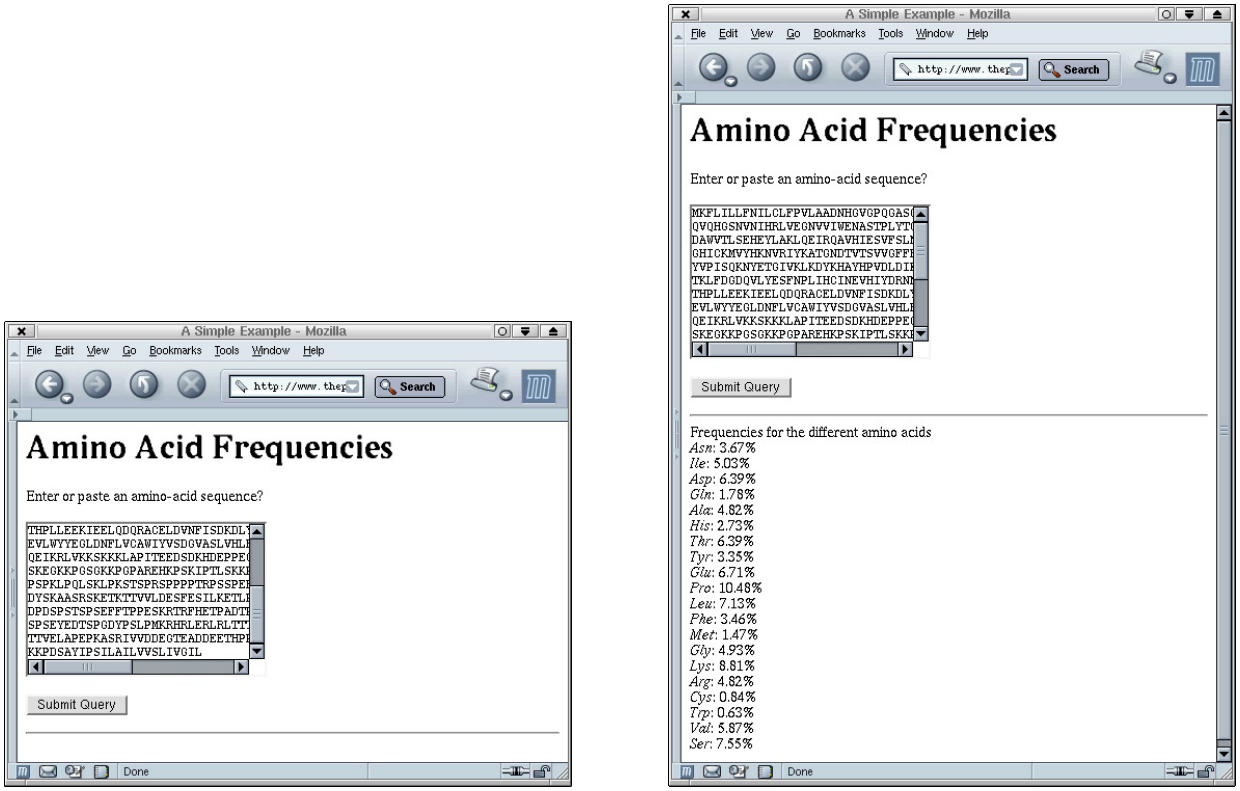


Figure 6.3: The browser looking at cgi3.pl (left figure) and the result after pressing the `submit` button (right figure).

The sequence that you type/paste into the textfield is returned in the variable `$seq`. This variable is then passed to the sub-routine `Count()` that computes the frequencies of the amino acids in the sequence.

### 6.3.4 Example 4

We are now going to create a *File Upload Field* by using the `filefield` method. Later we can access the contents of the uploaded file and process it as we wish. The following Perl script is an example of this.

cgi4.pl

```
#! /usr/bin/perl -w

##### Program description #######
#
# Title:     cgi4.pl
# Author(s): Mattias Ohlsson
# Description:
#   Small example of how to use the CGI module.
#
###################################

use strict;
use CGI;

# The error can be redirected to the browser
use CGI::Carp qw(fatalsToBrowser);

my $cgi = new CGI;

print $cgi->header,
    $cgi->start_html(-title=>'File Upload Example'),
    $cgi->h1('File Upload Example'),
    $cgi->start_multipart_form,
    "Give the path to a fasta file ",
    $cgi->p,
    $cgi->filefield(-name=>'upfile',-size=>50),
    $cgi->p,
    $cgi->submit,
    $cgi->end_form,
    $cgi->hr;

if ( $cgi->param() ) {
    my $filename = $cgi->upload('upfile');
    print "The fasta file $filename contains the following:",
    $cgi->br,$cgi->br;
    print "<pre>";
    while ( my $line = <$filename> ) {
        print "$line";
    }
    print "</pre>";
}
print $cgi->end_html();
```

cgi4.pl

The result of this Perl script is shown in figure 6.4.

Note that the variable `$filename` returned by the method `upload()` is both a filename and a file handle!

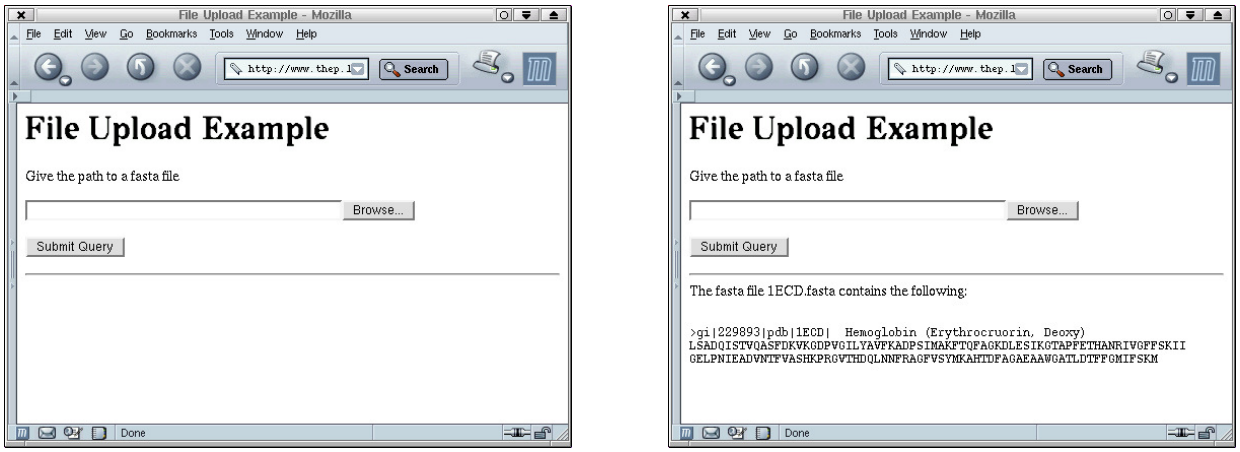


Figure 6.4: The browser looking at cgi4.pl (left figure) and the result after pressing the `submit` button (right figure).

### 6.3.5 Example 5

Another small example using a `password_field` form element. When you type text in such a field the characters are printed as "bullets". In this example your password is validated against a regular expression. Can you figure out the requirement for a safe password?

cgi5.pl

```
1  #! /usr/bin/perl -w
2
3  ##### Program description #######
4  #
5  # Title:     cgi5.pl
6  # Author(s): Mattias Ohlsson
7  # Description:
8  #   Small example of how to use the CGI module.
9  #
10 ###################################
11
12 use strict;
13 use CGI;
14 use CGI::Carp qw(fatalsToBrowser);
15
16 my $cgi = new CGI;
17
18 print $cgi->header,
19     $cgi->start_html(-title=>'Password validation form'),
20     $cgi->h1('A small utility to validate passwords');
21
22 if ( $cgi->param() ) {
23
24    # Get the password
25    my $passwd = $cgi->param("passwd");
26
27    if ($passwd =~ /^(?=.*\d)(?=.*[a-z])(?=.*[A-Z])(?=.*[%\.@#\$]).{8,}$/) {
28       print $cgi->p, "PASS: Your password is safe!";
```

```
29      } else {
30         print $cgi->p, "WARNING: Your password is unsafe!";
31      }
32
33  } else {
34
35      print $cgi->start_form,
36      $cgi->p, "Type password here ", $cgi->password_field("passwd"),
37      $cgi->p, $cgi->submit("Validate"), $cgi->reset("Clear"),
38      $cgi->end_form();
39
40  }
41  print $cgi->end_html();
```

cgi5.pl

### 6.3.6 Follow-up tasks 6-2

There are some preparations for these follow up tasks. In order to test the Perl scripts for this exercise you need to run them as CGI-scripts. This means that the WWW-server knows how to handle the Perl scripts so that when you point a WWW-browser to that specific file you will see the result of the execution of the Perl script and not just the Perl code itself. For this to happen all Perl scripts must be placed in a special directory in your (computer) account. This directory is called `public_html` and is placed in the top-most level in your directory tree. If you do not have such a directory you must create it. The make sure that it has the right set of permissions by running a `chmod 755` command. The following example shows how to do it:

```
bim0> cd
bim0> mkdir public_html
bim0> chmod 755 public_html
```

The example files used in this lecture are available as `cgi1.perl, ..., cgi5.perl`. Note that you have to rename these scripts to `cgi1.pl, cgi2.pl, ...` because the WWW-server only accepts .pl (or .cgi) as extensions of cgi-scripts.

Complete the following tasks:

1. At pages 33-34 in the CGI.pm reference document the Radio Button Group is described. Add such a button group in the `cgi2.pl` Perl script and print out the selected button when the `submit` button is pressed.
2. Create a Perl script that uploads a fasta file, shows the contents in the WWW-browser and count the number of Glysine in the sequence.
3. Create a small Perl script that uploads a clustalw alignment file and then converts that file to a phylip format. The result should be shown in the WWW-browser.

**HINT 1:** To get a well-formatted output in the www-browser one can use the `<pre> ... </pre>` html tag (see `cgi4.pl`). There is also a module called `HTML::FromText` that one can use.

**HINT 2:** In order to debug your cgi programs you may find the `CGI::Carp` module useful. See `cgi4.pl` for an example of how it can be used.

## 6.4 BioPerl project (Part 1)

What is BioPerl? The following can be read on their FAQ:

> What is BioPerl?
>
> BioPerl is a toolkit of perl modules useful in building bioinformatics solutions in Perl. It is built in an object-oriented manner so that many modules depend on each other to achieve a task. The collection of modules in the bioperl-live repository consist of the core of the functionality of bioperl. Additionally auxiliary modules for creating graphical interfaces (bioperl-gui), persistent storage in RDMBS (bioperl-db), running and parsing the results from hundreds of bioinformatics applications (Run package), software to automate bioinformatic analyses (bioperl-pipeline) are all available as Git modules in our repository.

The current version of BioPerl is 1.6.9 (and this version is available on your computers). The homepage of BioPerl is `http://www.bioperl.org` and the documentation for the modules can be found at `http://doc.bioperl.org/releases/bioperl-1.6.1`

We will start to look at two modules:

1. `Bio::Seq` - Sequence object with features
2. `Bio::SeqIO` - Handler for SeqIO formats

### 6.4.1 `Bio::Seq`

`Seq`, which is the central sequence object in BioPerl, is a sequence with sequence features placed on it. The `Seq` object also contains a `PrimarySeq` object. In summary

1. `PrimarySeq` = just the sequence and its names, nothing else.
2. `Seq` = A sequence and a collection of sequence features.

Tied to the `Seq` object are methods that can be used in order to extract information about the sequence. There is also a method that creates a new `Seq` object. `new`:

```
Title   : new
Usage   : $seq = Bio::Seq->new( -seq => 'ATGGGGGTGGTGGTACCCT',
                                -id  => 'human_id',
                                -accession_number => 'AL000012',
                              );

Function: Returns a new Seq object from
```

```
           basic constructors, being a string for the sequence
           and strings for id and accession_number
 Returns : a new Bio::Seq object
```

If you have sequences stored in a `Seq` object, there are methods available to find information about the sequences. Below is a subset of such methods.

```
The following methods return scalars

  $seqobj->seq();              # string of sequence
  $seqobj->subseq(5,10);       # part of the sequence as a string
  $seqobj->accession_number(); # when there, the accession number
  $seqobj->length()            # length
  $seqobj->desc();             # description
  $seqobj->alphabet();         # string, either 'dna','rna','protein'
  $seqobj->primary_id();       # a unique id for this sequence regardless
                               # of its display_id or accession number
  $seqobj->display_id();       # the human readable id of the sequence
```

The method `display_id()` has the following description:

```
 Title   : display_id
 Usage   : $id = $obj->display_id or $obj->display_id($newid);
 Function: Gets or sets the display id, also known as the common name of
           the Seq object.

           The semantics of this is that it is the most likely string
           to be used as an identifier of the sequence, and likely to
           have "human" readability.  The id is equivalent to the LOCUS
           field of the GenBank/EMBL databanks and the ID field of the
           Swissprot/sptrembl database. In fasta format, the >(\S+) is
           presumed to be the id, though some people overload the id
           to embed other information. Bioperl does not use any
           embedded information in the ID field, and people are
           encouraged to use other mechanisms (accession field for
           example, or extending the sequence object) to solve this.

           Notice that $seq->id() maps to this function, mainly for
           legacy/convenience issues.
 Returns : A string
 Args    : None or a new id
```

Let us show an example of how to use this method. We will use the `SeqIO` module (see below for more details) to read a set of sequences stored in fasta format and the print the ID for these sequences. Here is the code for doing this:

bp_ex1.pl

```
1 #! /usr/bin/perl -w
2
```

```
##### Program description #######
#
# Title:     bp_ex1.pl
# Author(s): Mattias Ohlsson
# Description:
#   A small script that reads a sequences stored in fasta format and
#   then prints the ID for these sequences. BioPerl is used.
#
###################################

use strict;
use Bio::SeqIO;

# Create an object that points to the file containing fasta sequences
my $in  = Bio::SeqIO->new(-file   => "seqs.fasta",
                          -format => 'Fasta');

# Make a loop over all the sequences
while ( my $seq = $in->next_seq() ) {
    my $Id = $seq->display_id();
    my $desc = $seq->description();
    print "$Id\n";
    print "$desc\n";
}
```

bp_ex1.pl

The result from running this script is shown below. Note, the efficency in reading and parsing the fasta file. This leads us to the `Bio::SeqIO` module.

```
gi|239758|bbs|68379
glucocorticoid receptor,  GR [human, Peptide Partial, 394 aa]
gi|239752|bbs|68871
PML-3=putative zinc finger protein [human, Peptide, 802 aa]
gi|238775|bbs|65126
putative tyrosine kinase receptor=UFO [human, NIH3T3, Peptide, 894 aa]
gi|239006|bbs|65162
alpha(1,3)-fucosyltransferase, ELFT [human, Peptide, 400 aa]
gi|237597|bbs|60089
putative adhesion molecule=ADMLX [human, Peptide, 679 aa]
gi|237995|bbs|62046
NK-1 receptor [human, lung, Peptide, 407 aa]
```

### 6.4.2 `Bio::SeqIO`

The `SeqIO` object is the "interface" that should be used in order to "load" external files into `Seq` objects. The `SeqIO` module also handles the complexity of parsing sequences of many standard formats that have emerged over the years. We start by looking at the method `Bio::SeqIO->new` in more detail. Here are two common ways of using this method[4]:

[4]See also the Bio::SeqIO Howto

```
$seqIO = Bio::SeqIO->new(-file => 'filename',   -format => $format);
$seqIO = Bio::SeqIO->new(-fh   => \*FILEHANDLE, -format => $format);
```

The returned object is a `Bio::SeqIO`, that you can think of as a collection of sequences. In the first example the sequences are read from the file `filename` with the format `$format`. In the second example we are instead reading from a supplied file-handle. The `new()` method accepts the following parameters:

```
-file
    A file path to be opened for reading or writing.  The usual Perl
    conventions apply:

   'file'       # open file for reading
   '>file'      # open file for writing
   '>>file'     # open file for appending
   '+<file'     # open file read/write
   'command |'  # open a pipe from the command
   '| command'  # open a pipe to the command

-fh
    You may provide new() with a previously-opened filehandle.  For
    example, to read from STDIN:

       $seqIO = Bio::SeqIO->new(-fh => \*STDIN);

    Note that you must pass filehandles as references to globs.

    If neither a filehandle nor a filename is specified, then the
    module will read from the @ARGV array or STDIN, using the familiar
    <> semantics.

    A string filehandle is handy if you want to modify the output in
    the memory, before printing it out. The following program reads in
    EMBL formatted entries from a file and prints them out in fasta
    format with some HTML tags:

      use Bio::SeqIO;
      use IO::String;
      my $in  = Bio::SeqIO->new('-file' => "emblfile" ,
                                    '-format' => 'EMBL');
      while ( my $seq = $in->next_seq() ) {
         # the output handle is reset for every file
         my $stringio = IO::String->new($string);
         my $out = Bio::SeqIO->new('-fh' => $stringio,
                                   '-format' => 'fasta');
         # output goes into $string
         $out->write_seq($seq);
         # modify $string
         $string =~ s|(>)(\w+)|$1<font color="Red">$2</font>|g;
         # print into STDOUT
         print $string;
      }
```

```
-format
    Specify the format of the file.  Supported formats include:

      AB1         ABI tracefile format
      ABI         ABI tracefile format
      ALF         ALF tracefile format
      CTF         CTF tracefile format
      EMBL        EMBL format
      EXP         Staden tagged experiment tracefile format
      Fasta       FASTA format
      Fastq       Fastq format
      GCG         GCG format
      GenBank     GenBank format
      PIR         Protein Information Resource format
      PLN         Staden plain tracefile format
      SCF         SCF tracefile format
      ZTR         ZTR tracefile format
      ace         ACeDB sequence format
      game        GAME XML format
      locuslink   LocusLink annotation (LL_tmpl format only)
      phd         phred output
      qual        Quality values (get a sequence of quality scores)
      raw         Raw format (one sequence per line, no ID)
      swiss       Swissprot format

    If no format is specified and a filename is given then the module
    will attempt to deduce the format from the filename suffix.  If
    this is unsuccessful then Fasta format is assumed. The format name
    is case insensitive.  'FASTA', 'Fasta' and 'fasta' are all valid
    suffixes.
```

The Perl script `bp_ex1.pl` used the `new` method to read the file `seqs.fasta`. We now understand how that works!

Two other useful methods in `SeqIO` are `next_seq` and `write_seq`. With these we can read and write sequences. The description is as follows:

```
 Title   : next_seq
 Usage   : $seq = stream->next_seq
 Function: Reads the next sequence object from the stream and returns it.

           Certain driver modules may encounter entries in the stream
           that are either misformatted or that use syntax not yet
           understood by the driver. If such an incident is
           recoverable, e.g., by dismissing a feature of a feature
           table or some other non-mandatory part of an entry, the
           driver will issue a warning. In the case of a
           non-recoverable situation an exception will be thrown.  Do
           not assume that you can resume parsing the same stream
           after catching the exception. Note that you can always turn
           recoverable errors into exceptions by calling
           $stream->verbose(2).
```

```
 Returns : a Bio::Seq sequence object
 Args    : none

 Title   : write_seq
 Usage   : $stream->write_seq($seq)
 Function: writes the $seq object into the stream
 Returns : 1 for success and 0 for error
 Args    : Bio::Seq object
```

The following Perl program reads the file `seqs.fasta` and creates a new file called `seqs.gbank` that stores the sequences in the GenBank format. Only 4 lines of code!

bp_ex2.pl

```
1  #! /usr/bin/perl -w
2
3  ##### Program description #######
4  #
5  # Title:     bp_ex2.pl
6  # Author(s): Mattias Ohlsson
7  # Description:
8  #   A small script that converts a fasta file to a swissprot file
9  #   using the Bio::SeqIO module in BioPerl.
10 #
11 ###################################
12
13 use strict;
14 use Bio::SeqIO;
15
16 my $in  = Bio::SeqIO->new(-file => 'seqs.fasta',  -format => 'Fasta');
17 my $out = Bio::SeqIO->new(-file => '>seqs.gbank', -format => 'GenBank');
18 my $out2 = Bio::SeqIO->new(-file => '>seqs.swiss', -format => 'Swiss');
19
20 while ( my $seq = $in->next_seq() ) {
21     $out->write_seq($seq);
22     $out2->write_seq($seq);
23 }
```

bp_ex2.pl

The first entry in `seqs.fasta` looks like,

```
>gi|239758|bbs|68379 glucocorticoid receptor,  GR [human, Peptide Partial, 394 aa]
MDSKESLTPGREENPSSVLAQERGDVMDFYKTLRGGATVKVSASSPSLAVASQSDSKQRRLLVDFPKGSV
SNAQQPDLSKAVSLSMGLYMGETETKVMGNDLGFPQQGQISLSSGETDLKLLEESIANLNRSTSVPENPK
SSASTAVSAAPTEKEFPKTHSDVSSEQQHLKGQTGTNGGNVKLYTTDQSTFDILQDLEFSSGSPGKETNE
SPWRSDLLIDENCLLSPLAGEDDSFLLEGNSNEDCKPLILPDTKPKIKDNGDLVLSSPSNVTLPQVKTEK
EDFIELCTPGVIKQEKLGTVYCQASFPGANIIGNKMSAISVHGVSTSGGQMYHYDMNTASLSQQQDQKPI
FNVIPPIPVGSENWNRCQGSGDDNLTSLGTLNFPGRTVFSNGYS
```

and the corresponding entry in the new file `seqs.gbank` is,

```
LOCUS       gi|239758|bbs|68379          394 aa            linear   UNK
DEFINITION  glucocorticoid receptor,  GR [human, Peptide Partial, 394 aa]
ACCESSION   unknown
FEATURES             Location/Qualifiers
ORIGIN
        1 mdskesltpg reenpssvla qergdvmdfy ktlrggatvk vsasspslav asqsdskqrr
       61 llvdfpkgsv snaqqpdlsk avslsmglym getetkvmgn dlgfpqqgqi slssgetdlk
      121 lleesianln rstsvpenpk ssastavsaa ptekefpkth sdvsseqqhl kgqtgtnggn
      181 vklyttdqst fdilqdlefs sgspgketne spwrsdllid encllsplag eddsfllegn
      241 snedckplil pdtkpkikdn gdlvlsspsn vtlpqvktek edfielctpg vikqeklgtv
      301 ycqasfpgan iignkmsais vhgvstsggq myhydmntas lsqqqdqkpi fnvippipvg
      361 senwnrcqgs gddnltslgt lnfpgrtvfs ngys
//
```

Compare this program to the one that you (possible) wrote in one of the hand-in exercises of the second week.

### 6.4.3 `Bio:DB::GenBank`

There are modules in the BioPerl project that makes it easy to get sequences from many different databases (e.g. GenBank, GenPept, Swissprot etc). We will now look at the module for retrieving sequences from GenBank. Here are some useful methods for a `Bio::DB:GenBank` object:

```
get_Seq_by_id

 Title   : get_Seq_by_id
 Usage   : $seq = $db->get_Seq_by_id('ROA1_HUMAN')
 Function: Gets a Bio::Seq object by its name
 Returns : a Bio::Seq object
 Args    : the id (as a string) of a sequence
 Throws  : "id does not exist" exception

get_Seq_by_gi

 Title   : get_Seq_by_gi
 Usage   : $seq = $db->get_Seq_by_gi('405830');
 Function: Gets a Bio::Seq object by gi number
 Returns : A Bio::Seq object
 Args    : gi number (as a string)
 Throws  : "gi does not exist" exception

get_Stream_by_id

  Title   : get_Stream_by_id
  Usage   : $stream = $db->get_Stream_by_id( [$uid1, $uid2] );
  Function: Gets a series of Seq objects by unique identifiers
  Returns : a Bio::SeqIO stream object
  Args    : $ref : a reference to an array of unique identifiers for
                   the desired sequence entries
```

```
get_Stream_by_gi

  Title   : get_Stream_by_gi
  Usage   : $seq = $db->get_Seq_by_gi([$gi1, $gi2]);
  Function: Gets a series of Seq objects by gi numbers
  Returns : a Bio::SeqIO stream object
  Args    : $ref : a reference to an array of gi numbers for
                   the desired sequence entries
  Note    : For GenBank, this just calls the same code for get_Stream_by_id()
```

The following Perl script downloads two GenBank sequences and displays them on the screen in the GenBank format.

bp_ex3.pl

```
#! /usr/bin/perl -w

##### Program description #######
#
# Title:     bp_ex3.pl
# Author(s): Mattias Ohlsson
# Description:
#  A small perl program that downloads two GenBank records from NCBI.
#  The records are displayed on the screen. Bio::DB:GenBank is used here.
#
###################################

use strict;
use Bio::SeqIO;
use Bio::DB::GenBank;

my $gb = Bio::DB::GenBank->new();
my $out = Bio::SeqIO->new(-fh => \*STDOUT, -format => 'GenBank');

my $seqio = $gb->get_Stream_by_gi(['AA484435', 'AA484440']);

while ( my $seq = $seqio->next_seq() ) {
    $out->write_seq($seq);
    print "---------------------------------\n";
}
```

bp_ex3.pl

The result of running this program (only the first GenBank entry):

```
LOCUS       AA484435                461 bp    mRNA    linear   EST 08-JAN-2011
DEFINITION  nf07c12.s1 NCI_CGAP_Li1 Homo sapiens cDNA clone IMAGE:913078
            similar to SW:A1BG_HUMAN P04217 ALPHA-1B-GLYCOPROTEIN, mRNA
            sequence.
ACCESSION   AA484435
```

```
VERSION     AA484435.1  GI:2213248
DBSOURCE    BioSample accession LIBEST_000884
KEYWORDS    EST.
SOURCE      Homo sapiens (human)
  ORGANISM  Homo sapiens
            Eukaryota; Metazoa; Chordata; Craniata; Vertebrata; Euteleostomi;
            Mammalia; Eutheria; Euarchontoglires; Primates; Haplorrhini;
            Catarrhini; Hominidae; Homo.
REFERENCE   1  (bases 1 to 461)
  CONSRTM   NCI-CGAP http://www.ncbi.nlm.nih.gov/ncicgap
  TITLE     National Cancer Institute, Cancer Genome Anatomy Project (CGAP),
            Tumor Gene Index
  JOURNAL   Unpublished
COMMENT     Contact: Robert Strausberg, Ph.D. Email: cgapbs-r@mail.nih.gov
            Tissue Procurement: David E. Kleiner, M.D., Ph.D., Rodrigo F.
            Chuaqui, M.D., Michael R. Emmert-Buck, M.D., Ph.D. cDNA Library
            Preparation: David B. Krizman, Ph.D. cDNA Library Arrayed by: Greg
            Lennon, Ph.D. DNA Sequencing by: Washington University Genome
            Sequencing Center Clone distribution: NCI-CGAP clone distribution
            information can be found through the I.M.A.G.E. Consortium/LLNL
            at: www-bio.llnl.gov/bbrp/image/image.html Insert Length: 621 Std
            Error: 0.00 Seq primer: -41m13 fwd. ET from Amersham High quality
            sequence stop: 417.
FEATURES             Location/Qualifiers
     source          1..461
                     /organism="Homo sapiens"
                     /lab_host="DH10B"
                     /db_xref="taxon:9606"
                     /clone_lib="LIBEST_000884 NCI_CGAP_Li1"
                     /mol_type="mRNA"
                     /clone="IMAGE:913078"
                     /note="Vector: pAMP10; mRNA made from normal liver
                     hepatocytes, cDNA made by oligo-dT priming.
                     Non-directionally cloned. Size-selected on agarose gel,
                     average insert size 600 bp. Reference: Krizman et al.
                     (1996) Cancer Research 56:5380-5383."
                     /tissue_type="liver"
ORIGIN
        1 ggtcgacctc gatggcgcca gtgtcctgga tcaccgccgg cctgaaaaca acagcagtgt
       61 gccgaggtgt gctgcggggt gtgacttttc tgctgaggcg ggagggcgac catgagtttc
      121 tggaggtgcc tgaggcccag gaggatgtgg aggccacctt tccagtccat cagcctggca
      181 actacagctg cagctaccgg accgatgggg aaggcgccct ctctgagccc agcgctactg
      241 tgaccattga ggagctcgct gcaccaccac cgcctgtgct gatgcaccat ggagagtcct
      301 cccaggtcct gcaccctggc aacaaggtga ccctcacctg cgtggctccc ctgagtggag
      361 tggacttcca gctacggcgc ggggagaaag agctgctggt acccaggagc agcaccagcc
      421 cagatcgcat cttctttcac ctgaacgcgg tggccctggg g
//
---------------------------------
```

### 6.4.4 Follow-up tasks 6-3

The example files used in this lecture are available as `bp_ex1.pl, bp_ex2.pl` and `bp_ex3.pl` In addition to that you also need the set of fasta sequences, available as `seqs.fasta`.

Complete the following tasks:

1. Modify the perl script `bp_ex1.pl` so that the alphabet and the length of the sequence are printed together with the ID.

2. Make a Perl script that can convert a fasta sequence (or sequences) to one of the following formats:

1. EMBL
2. Swissprot
3. GCG

The format to use should be selected using command line options. Read the fasta file from standard input and display the result on standard output.

3. Make a Perl script that creates a text field using `CGI.pm` where the user can enter a GenBank accession number. Use Bio:DB:GenBank to download this sequence and display it in EMBL format in the WWW browser.

**HINT 1:** To get a well-formatted output in the www-browser one can use the `<pre> ... </pre>` html tag (see `cgi4.pl`). There is also a module called `HTML::FromText` that one can use.

**HINT 2:** In order to debug your cgi programs you may find the `CGI::Carp` module useful. See `cgi4.pl` for an example of how it can be used.

**NOTE:** You can find more information about the different modules and their methods if you look at `http://doc.bioperl.org/releases/bioperl-1.6.1` or using the perldoc command (e.g. `>> perldoc Bio::DB::GenBank`)

## 6.5 BioPerl project (Part 2)

We will now continue to examine some other modules in the BioPerl project, where the focus will be on sequence alignment and its subsequent analysis. Specifically we will look at the following modules:

1. `Bio::SearchIO`: This module is used for parsing blast and fasta reports.
2. `Bio::Tools::Run::RemoteBlast` Remote execution of blasts at NCBI.
3. `Bio::Tools::AlignIO` Modules for handling multiple alignments (similar to SeqIO).
4. `Bio::Tools::OddCodes` Producing an alternative alphabet coding.

### 6.5.1 `Bio::SearchIO`

The module `Bio::SearchIO` can be used to parse BLAST reports. For more information read the SearchIO Howto. The following Perl program does similar things as your `parse.pl` that you worked on last week.

bp_ex4.pl†

```
1 #! /usr/bin/perl -w
2
3 ##### Program description #######
4 #
```

```
# Title:     bp_ex4.pl
# Author(s): Mattias Ohlsson
# Description:
#   A small Perl program that parses the result of a blast run. The
#   Bio::Search and Bio::SearchIO is used.
#
###################################

use strict;
use Bio::SearchIO;

my $in = new Bio::SearchIO(-fh => \*STDIN, -format => 'blast');

# We only have one report!
my $result = $in->next_result();

while ( my $hit = $result->next_hit() ) {
    while ( my $hsp = $hit->next_hsp ) {
        if ( $hsp->percent_identity >= 80 && $hsp->percent_identity <= 85 ) {

            print "======= ID: ", $hit->name, "=======\n";
            printf "=> Identities: %5.2f%%\n", $hsp->percent_identity;

            my $q = $hsp->query_string;
            my @qarr = ($q =~ /(.{1,60})/g);
            my $sQ = $hsp->start('query');

            my $s = $hsp->hit_string;
            my @sarr = ($s =~ /(.{1,60})/g);
            my $sS = $hsp->start('subject');

            my $hom = $hsp->homology_string;
            my @homarr = ($hom =~ /(.{1,60})/g);

            for ( my $i = 0; $i < scalar @qarr; $i++ ) {
                printf "Query: %4d $qarr[$i] %4d\n", $sQ, $sQ+length($qarr[$i])-1;
                printf "       %4s $homarr[$i]\n", '';
                printf "Subj:  %4d $sarr[$i] %4d\n\n", $sS, $sS+length($sarr[$i])-1;
            }
        }
    }
}
```

bp_ex4.pl†

(See appendix A for the result when running this program on the 1TEN_blastp.res file.) The high scoring pair (`$hsp`) obtained by the `next_hsp` method has many methods. Here are some:

```
$hsp->evalue
$hsp->length
$hsp->gaps
$hsp->frac_identical
$hsp->frac_conserved
$hsp->query_string
```

```
$hsp->hit_string
$hsp->homology_string
$hsp->start('arg')
$hsp->end('arg')
```

You can find more information looking at the documentation for the `Bio::Search::HSP::HSPI` module (e.g `>> perldoc Bio::Search::HSP::HSPI`).

FASTA reports can also be analyzed using the `Search/SearchIO` modules.

### 6.5.2 `Bio::Tools::Run::RemoteBlast`

This module makes it possible to run a remote blast at NCBI. The documentation for this module is not that good. However one can find an example similar to the one below.

bp_ex5.pl.pl†

```
#! /usr/bin/perl -w

##### Program description #######
#
# Title:     bp_ex5.pl
# Author(s): Mattias Ohlsson
# Description:
#   A Perl program that can run a remote blast at NCBI. The
#   module Tools::Run::RemoteBlast is used.
#
###################################

use strict;
use Bio::SeqIO;
use Bio::Tools::Run::RemoteBlast;

# Get the query sequence
my $Seq_in = Bio::SeqIO->new (-file => '1TEN.fasta',
                              -format => 'fasta');
my $query = $Seq_in->next_seq();


# Remote-blast "factory object" creation
my $factory = Bio::Tools::Run::RemoteBlast->new(
                                                -prog       => 'blastp',
                                                -data       => 'nr',
                                                -expect     => '1e-6',
                                                -readmethod => 'SearchIO');

# Here we submit the query sequence to the Blast server
$factory->submit_blast($query);

print STDERR "waiting...";

# Loop over all possible remote id's
while ( my @rids = $factory->each_rid ) {
    # printf "Debug: No rids %d\n", scalar @rids;
```

```
38
39    foreach my $rid ( @rids ) {
40       my $rc = $factory->retrieve_blast($rid);
41
42       unless ( ref $rc ) {
43
44          # retrieve_blast returns -1 on error
45          if ( $rc < 0 ) {
46             $factory->remove_rid($rid);
47          }
48
49          # retrieve_blast returns 0 on 'job not finished'
50          sleep 5;
51          print STDERR ".";
52
53       } else {
54
55          # Blast done
56          $factory->remove_rid($rid);
57          my $result = $rc->next_result();
58
59          # Save the output
60          my $filename = 'blast.out';
61          $factory->save_output($filename);
62
63          # Analyse the blast result
64          print "Database: ", $result->database_name(), "\n";
65          print "\nQuery Name: ", $result->query_name(), "\n\n";
66          while ( my $hit = $result->next_hit ) {
67             while ( my $hsp = $hit->next_hsp ) {
68                if ( $hsp->percent_identity >= 95 && $hsp->percent_identity <= 99 ) {
69                   print "======= ID: ", $hit->name, "=======\n";
70                   printf "=> Identities: %5.2f%%\n\n", $hsp->percent_identity;
71
72                   my $q = $hsp->query_string;
73                   my @q = ($q =~ /(.{1,60})/g);
74                   my $sQ = $hsp->start('query');
75
76                   my $s = $hsp->hit_string;
77                   my @s = ($s =~ /(.{1,60})/g);
78                   my $sS = $hsp->start('subject');
79
80                   my $hom = $hsp->homology_string;
81                   my @hom = ($hom =~ /(.{1,60})/g);
82
83                   for ( my $i = 0; $i < scalar @q; $i++ ) {
84                      printf "Query: %4d $q[$i] %4d\n", $sQ, $sQ+length($q[$i])-1;
85                      printf "       %4s $hom[$i]\n", '';
86                      printf "Subj:  %4d $s[$i] %4d\n\n", $sS, $sS+length($s[$i])-1;
87                   }
88                }
89             }
90          }
91       }
92    }
```

```
93 }
```
bp_ex5.pl.pl†

This program reads the fasta file '1TEN.fasta' using `SeqIO` and this sequence object is then used to create a `RemoteBlast` object called `$factory`. Using this `$factory` object we then call the method `submit_blast()`. A given alignment job is given it's unique "remote id" (rid) and a collection of rids can be retrieved using the `each_rid()` method. Given a rid one can call the method `retrieve_blast()` to actually obtain the blast result, if ready. Try to understand how this program works! See appendix A for the result when running program and compare this to the result of `bp_ex4.pl`.

### 6.5.3 Bio::AlignIO

The `Bio::AlignIO` is similar to the `SeqIO` but it works on alignment files rather then sequences files as for the latter. An alignment file can be produced by e.g. the clustalw program for multiple sequence alignments. Below is a very short converter between phylip and clustalw formats. Compare this to the converter you wrote in chapter 5! NOTE: The clustalw format that `Bio::AlignIO` produces is somewhat different from the one we looked at in chapter 5.

bp_ex7.pl†

```
1  #! /usr/bin/perl -w
2
3  ##### Program description #######
4  #
5  # Title:     bp_ex7.pl
6  # Author(s): Mattias Ohlsson
7  # Description:
8  #   The shortest phylip to clustalw converter ever!!
9  #
10 ###################################
11
12 use strict;
13 use Bio::AlignIO;
14
15 my $in  = Bio::AlignIO->new(-fh => \*STDIN,  -format => 'phylip');
16 my $out = Bio::AlignIO->new(-fh => \*STDOUT, -format => 'clustalw');
17
18 $out->write_aln( $in->next_aln() );
```
bp_ex7.pl†

### 6.5.4 Bio::Tools::OddCodes

This module can be used to produce an alternative alphabet coding for one protein sequence. One can for instance turn an protein sequence, displayed in the usual 20-letter alphabet, into a 2-letter hydrophobicity alphabet sequence. Here are more specific information about the module:

```
Creating the OddCodes object, eg:
```

```
        my $inputstream = Bio::SeqIO->new( '-file' => "seqfile",
                                           '-format' => 'Fasta');
        my $seqobj = $inputstream->next_seq();
        my $oddcode_obj = Bio::Tools::Oddcodes->new(-seq => $seqobj);
```

```
hydrophobic

 Title   : hydrophobic
 Usage   : $output = $oddcode_obj->hydrophobic();
 Function: turns amino acid sequence into 2-letter hydrophobicity alphabet
         : O (hydrophobic), I (hydrophilic)
 Example : a sequence ACDEFGH will become OIIIOII
 Returns : Reference to the new sequence string
 Args    : none
```

Here is an example that uses `Bio::Tools::Oddcodes`.

bp_ex8.pl

```
1  #! /usr/bin/perl -w
2
3  ##### Program description #######
4  #
5  # Title:     bp_ex8.pl
6  # Author(s): Mattias Ohlsson
7  # Description:
8  #   An example of the OddCodes module
9  #
10 ###################################
11
12 use strict;
13 use Bio::SeqIO;
14 use Bio::Tools::OddCodes;
15
16 # Read the file containing fasta sequences
17 my $in  = Bio::SeqIO->new(-fh => \*STDIN, -format => 'Fasta');
18
19 # Make a loop over all the sequences
20 while ( my $seq = $in->next_seq() ) {
21
22     # Create a an OddCodes object
23     my $odd = Bio::Tools::OddCodes->new(-seq => $seq);
24
25     # Make a hydrophobic sequence
26     my $hseq_ref = $odd->hydrophobic();
27
28     # Get the aa sequence
29     my $aaseq = $seq->seq();
30
31     # Make a nice printout
32     my $Id = $seq->display_id();
33     print ">>>>ID: $Id\n";
34
35     my @tmp1 = ($aaseq =~ /(.{1,50})/g);
```

```
36      my @tmp2 = (${$hseq_ref} =~ /(.{1,50})/g);
37
38      my $len = @tmp1;
39      for ( my $i = 0; $i < $len; $i++ ) {
40          print "$tmp1[$i]\n";
41          print "$tmp2[$i]\n\n";
42      }
43  }
```

bp_ex8.pl

The result from running this program i.e. (`>>/bp_ex8.pl < 1TEN.fasta`)

```
1  >>>>ID: 1TEN:_
2  RLDAPSQIEVKDVTDTTALITWFKPLAEIDGIELTYGIKDVPGDRTTIDL
3  OIOIIOOIOIOOIOOOOIIIOIIOIIIOIOOIOIOOOIOOIIOOOOOIOI
4
5  TEDENQYSIGNLKPDTEYEVSLISRRGDMSSNPAKETFTT
6  OOOOOOOOIOOIOIOOOOOIOIIOOOOOIOOOIIOOOIOO
7
```

### 6.5.5 Follow-up tasks 6-4

The perl programs used in this lecture are available as `bp_ex4.pl` `bp_ex5.pl`, `bp_ex7.pl` and `bp_ex8.pl`. To run these programs you will also need: `1TEN_blastp.res`, `1TEN.fasta` and `phylip1.align`. In addition, for the follow-up tasks, you also need `1TEN_fasta.res`

Complete the following tasks:

1. Make a parser for the `1TEN_fasta.res` result file using the SearchIO module.
2. Use the `Bio::Tools::SeqStats` module to make a frequency table of all amino acids in the `1TEN.fasta` file.

## 6.6 Hand-in exercise 4

Select one of the following problems as the hand-in exercise for this week.

1. **Parsing and simple analysis of large GenBank files**. In this hand-in you will work with a relatively large GenBank file. The task is to write a parser for a GenBank file that can accomplish the following tasks:
    1. Extract all entries in the GenBank file that belongs to the organism "Borrelia" and save them in an external, FASTA formatted, file. Report the number of entries found.
    2. Report the number of unique organisms in the GenBank file.

3. Report the average (nucleotide) sequence length.
4. Report the number of entries with at least one Coding sequence.
5. Show the protein sequence for the last translated protein found in the Genbank file. Also, download, using the `Bio::DB::GenBank` module, this protein and display the downloaded sequence.

There are two Genbank files to choose from, `ldb1.gb` or `ldb2.gb` containing 5000 and 10000 entries respectively. The files are available from the course homepage. Sample outputs can also be found there.

**Hints:** Most of the parsing can be accomplished using modules/methods from the Bioperl project. Detailed information about the GenBank format can be found here `http://www.ncbi.nlm.nih.gov/Sitemap/samplerecord.html`.

2. **Sequence statistics**. In this hand-in you will use perl and perl modules to compute some nucleotide sequence statistics. For the task you will use the complete sequence of the "Dengue virus 4". Your program should perform the following tasks:

   1. Find the number of occurrences for the individual nucleotides ("A", "G", "T", "C") in the sequence. Present as both numbers a percentages.
   2. Count the number of occurrences of all DNA words of length 2. Present in a decreasing order.
   3. Some words are more frequent than others. Is this a simple consequence of the fact the nucleotides occur with different frequencies? One can compare the numbers with the corresponding numbers one would get by permuting the sequence (randomly) many times and count the occurrences of the different DNA words. If the "real" number of some specific word significantly lies outside the distribution of numbers obtained from the permuted sequences one would take this as a indication of non-random behavior. The task can be broken down into the following steps:
      a. permute the sequence.
      b. count the number of all DNA words of length 2 (step 2 above) and store all numbers.
      c. repeat a-b "many" times ($> 1000$).
      d. find significant differences between the distribution and the real number. You can use the criteria of $\pm$ four standard deviations.
      e. present your findings.

   The "Dengue virus 4" can be found in the file `dengue_virus_4.gb`, available from the course homepage. A sample output can also be found there.

   **Hints:** `Bio::SeqIO`, `Math::Random` and `Statistics::Descriptive` are useful modules for this exercise.