Modern Perl for Biologists II | Deeper Concepts

M-BIM / INBTBI

Denis BAURAIN / ULiège

LIÈGE université
Sciences

Edition 2020–2021

Acknowledgment

This course has been developed over eight years (2013–2020) primarily as teaching materials for the *Modern Perl* module of the (INBTBI) Bioinformatics Training organized by the Biotechnology Training Centre in the GIGA Tower of the University of Liège (Belgium). Until 2016, the Biotechnology Training Centre was directed by Laurent Corbesier, who is warmly thanked for his support.

The training itself was funded through the *Bioinformatics Training Program*, 6th call 2010–2014, BioWin (the Health Cluster of Wallonia), Le Forem (the Walloon Office for Employment and Training), and the Biotechnology Training Centre (Forem-GIGA).

This two-part document benefitted from the feed-back of about 150 students, trainees and colleagues, among which Arnaud Di Franco, Agnieszka Misztak, Loïc Meunier and Sinaeda Anderssen are especially acknowledged.

If you spot typos, grammatical or technical errors, or have any suggestion on how to improve the next edition of this course, let me know (`denis.baurain@uliege.be`). I will answer all messages.

Finally, I thank Pierre (Pit) Tocquin for his help with the mighty triad Markdown / Pandoc / LaTeX.

How to read this course?

This document is written in American English.

The first time they are introduced, programming terms and biological terms are typeset in **bold** and *italic*, respectively, whereas computer keywords are always typeset in a `monospaced` font (Consolas). All these terms are indexed at most once per section in which they appear (see Index).

While the main text is typeset in Palatino, user statements (either at the shell or as small code excerpts) and computer answers are typeset in the same `monospaced` font as computer keywords.

```
$ echo "Hello world!"          # user statement (note the shell prompt)
Hello world!                   # computer answer
```

Example BOX

Advanced material that can be skipped on first reading is enclosed in boxes with a light blue background. A List of Boxes is available for convenience.

General programming advices and good practices that apply beyond the Perl language are typeset in a smaller size and introduced by a yellow bar surmounted by an icon.

```
say <<'EOT';
Complete program listings are typeset
in the same monospaced font as keywords.
Their lines are numbered.
EOT
```

Contents

List of Boxes

List of Tables

List of Figures

Part I

Lesson 6

Chapter 1

Thinking in list context

1.1 Old-school plots in the terminal

In "Operator precedence and associativity" (see the first part of this course), I gave you three approaches to determine whether two DNA fragments overlap or not. Below is a solution for the corresponding homework that is more advanced than what I asked you to do at that time.

1. Type in the program shown in the following pages and save it as `check_overlap.pl`.
2. Install the new required CPAN module.

   ```
   $ cpanm Term::Size::Any
   ```

3. Build a test file such as the one displayed here.

   ```
   # inclusions
   10  60  10  60
   10  60  30  60
   30  60  10  60
   10  40  10  60
   10  60  10  40
   10  60  30  50
   30  50  10  60

   # left overhangs
   10  50  30  60
   10  50  50  60

   # right overhangs
   30  60  10  50
   50  60  10  50

   # no overlap
   10  30  31  60
   31  60  10  30
   10  30  40  60
   ```

```
40  60  10  30

# invalid coordinates
30  10  31  60
10  30  60  31
0   30  31  60
30  0   31  60
10  30  0   60
10  30  31  0
```

4. Play with the program. Resize your terminal and run it again. Observe how it adapts. You can also edit the use `Smart::Comments '###'` line to fine tune the **verbosity** of the debugging.

```
#!/usr/bin/env perl

use Modern::Perl '2011';
use autodie;

use Smart::Comments '###';

use List::AllUtils qw(min max uniq);
use POSIX;
use Term::Size::Any 'chars';


unless (@ARGV == 2) {
    die <<"EOT";
Usage: $0 <coords.txt> <tic-width>
This tool plots two fragments and determines whether they overlap. It requires
an input file where each line contains four whitespace-delimited strictly
positive coordinates for the two fragments to check (x1, x2, y1, y2).
Example: $0 coords.txt 8
EOT
}

my $infile = shift;
my $tic_width = shift;

# setup plotting area
my ($cols, $rows) = chars();
my $tic_n = floor( $cols / $tic_width );
#### area: $cols . 'x' . $rows
#### $tic_n

open my $in, '<', $infile;

LINE:
while (my $line = <$in>) {

    next LINE if $line =~ m/^ \s* $/xms;        # skip empty lines
    next LINE if $line =~ m/^ \#/xms;           # skip comment lines

    # extract coordinates
    chomp $line;
    my @coords = split /\s+/xms, $line;
    my ($x1, $x2, $y1, $y2) = @coords;
    #### @coords

    # setup plot scale
    my $min = min @coords;
    my $max = max @coords;
    my $range = $max - $min;
    my $step = ceil( $range / ($tic_n-1) );
    #### $min
    #### $max
    #### $range
    #### $step

    # skip invalid coordinates
    if ($min < 1) {
        warn "WARNING! At least one non-natural bound: @coords";
        next LINE;
    }
    if ($x2 < $x1 || $y2 < $y1) {
        warn "WARNING! Incoherent coordinates: @coords";
        next LINE;
    }

    # draw plot
    plot_scale($min, $step);
    plot_fragment($x1, $x2, 0, $min, $step);
    plot_fragment($y1, $y2, 1, $min, $step);

    #     x1             x2                       x1             x2
    #     |==============|                        |==============|
    #             |==============|        |==============|
    #             y1             y2       y1             y2
    #
    #     x1             x2                 x1         x2
    #     |==============|                  |==========|
    #       |==========|                  |==============|
    #       y1         y2                 y1             y2

    # check overlap
    my @overlaps = (
    # using high-precedence logical operators
        (   ($y1 >= $x1 && $y1 <= $x2)          #  left cases
         || ($x1 >= $y1 && $x1 <= $y2) ),       # right cases

    # mixing high- and low-precedence logical operators
        (    $y1 >= $x1 && $y1 <= $x2           #  left cases
         or  $x1 >= $y1 && $x1 <= $y2  ),       # right cases

    # ... or more intelligently...
        (not $y1 >  $x2 || $y2 <  $x1  ),       # test disjunction
    );

    # output overlap status
    @overlaps = uniq(@overlaps);
    ### assert: @overlaps == 1
    say "---> $x1-$x2 and $y1-$y2 "
        . (shift @overlaps ? 'DO' : 'DO NOT')
        . ' overlap!'
        . "\n"
    ;
}


sub plot_scale {
    say step_line(0, @_);
    say step_line(1, @_);
    say q{-} x $cols;
    return;
}

sub plot_fragment {
    my $x1 = shift;
    my $x2 = shift;
    my $second = shift;

    my $pad_n = xloc($x1, @_);
    my $chr_n = xloc($x2, @_) - $pad_n + 1;
    my $fragm_str = q{ } x $pad_n . q{#} x $chr_n;

    my $spc_n = $chr_n - length($x1) - length($x2);
       $spc_n = 0 if $spc_n < 0;
    my $coord_str = q{ } x $pad_n . $x1 . q{ } x $spc_n . $x2;
    #### $x1
    #### $x2
    #### $pad_n
    #### $chr_n
    #### $spc_n
    #### $fragm_str
    #### $coord_str

    say $coord_str unless $second;
    say $fragm_str;
    say $coord_str     if $second;
    return;
}

sub step_line {
    my ($bars, $min, $step) = @_;

    my $str;
    for (my ($x, $tic) = ($min, 0); $tic < $tic_n; $x += $step, $tic++) {
        $str .= sprintf("%-*d", $tic_width, $x) unless $bars;
        $str .= '|' . q{ } x ($tic_width-1)        if $bars;
    }
    return $str;
}

sub xloc {
    my ($x, $min, $step) = @_;
    return sprintf "%.0f", ($x-$min) / $step * $tic_width;
}
```

1.2 Perl values: Lists

List manipulation is at the heart of *Modern Perl*. We have already seen how to iterate over a list one value at a time (using the "*foreach-style* `for` loop", see the first part of this course), but experienced programmers often use lists in a more direct way. In this section, we will summarize a series of idioms related to **list-oriented programming**. Let's first remind you what a list is and how to define it.

A list is a group of one or more expressions separated by **comma characters** (`,`). Even if lists are often surrounded by a pair of parenthesis characters (`(` and `)`), these are not required for creating a list, whereas the infix **comma operator** is. Parentheses instead act on precedence (here, which expressions have to be grouped in a given list). To see that, consider the examples in the box below.

When you need to define a list corresponding to a range of values, use the range operator (`..`), as explained in the *foreach-style* `for` loop. For example, we could have defined the list above as follows.

```
my @numbers = 5..8;
### @numbers

# gives:
### @numbers: [
###             5,
###             6,
###             7,
###             8
###           ]
```

The range operator is very common in `for` loops with a numeric iterator, such as the genetic-code building loop encountered in several of our programs (e.g., `translate.pl`).

```
my $codon_n = length $aa;
for my $i (0..$codon_n-1) {
    # loop body
}
```

This operator is not smart enough to count in **descending order**, but you can achieve the same result by reversing the corresponding list in **ascending order**.

```
my @countdown = 3..0;
### @countdown

my @countdown_2nd = reverse 0..3;
### @countdown_2nd

# gives:

### @countdown: []

### @countdown_2nd: [
###                   3,
###                   2,
###                   1,
###                   0
###                 ]
```

In contrast, the range operator can operate on non-numeric characters. This is occasionally useful and reminiscent of the character ranges defined in regular expressions with the hyphen character (`-`).

```
my @letters = 'a'..'z';
### @letters

# gives:

### @letters: [
###             'a',
###             'b',
###             'c',
...
###             'y',
###             'z'
###           ]
```

Finally, remember that lists of literal words can be defined using the quoted word operator (`qw(...)`), which splits the enclosed string on whitespace to produce a list of substrings. Observe how we can even include single quotes (`'`) in some words.

```
my @bases = qw(A C G T);
### @bases

my @gene_parts = qw(promoter 5'-UTR exon intron 3'-UTR);
### @gene_parts

# gives:

### @bases: [
###           'A',
###           'C',
###           'G',
###           'T'
###         ]

### @gene_parts: [
###                'promoter',
###                '5\'-UTR',
###                'exon',
###                'intron',
###                '3\'-UTR'
###              ]
```

We have already seen `qw(...)` in `codon_usage.pl` and explained that it was convenient to produce header lines for tables when combined with `join` (see "Writing files", in the first part of this course).

```
say '# ' . join "\t", qw(codon count aa total usage);

# gives:
```

```
# codon count   aa  total   usage
```

The `qw(...)` operator does not like comment characters and outputs a warning when it sees one.

```
say join "\t", qw(#codon count aa total usage);

# gives:
```

```
Possible attempt to put comments in qw() list at qw.pl line 6.
#codon  count   aa  total   usage
```

Lists also occur as the results of expressions evaluated in so-called list context. Assigning to an array or to a hash forces list context, as well as assigning to a list of one or more scalars surrounded by a pair of parentheses. Below, I give you a few examples from our previous programs.

```
# scalars
my ($id) = $code =~ m/ id \s* (\d+) /xms;                  # xxl_xlate.pl
my ($basename, $dir) = fileparse($infile, qr{\.[^.]*}xms);  # xxl_xlate.pl
while (my ($id, $dna_string) = each %seq_for) {            # xxl_xlate.pl
    # loop body
}
```

```
# array
my @bases = split //, $dna_string;                        # rev_comp.pl
my @questions = shuffle keys %aa_for;                     # codon_quizz.pl
my @codes = $gc_content =~ m/ \{ ( [^{}]+ ) \} /xmsg;      # xxl_xlate.pl

# hash
my %comp_for = (                                          # rev_comp.pl
    A => 'T',   T => 'A',   G => 'C',   C => 'G',
    a => 't',   t => 'a',   g => 'c',   c => 'g',
);
```

List assignment is both predictable and greedy. If you want to discard the last value(s) of a list, simply provide less variables than the number of values in the list. If you want to discard some values in the middle of a list, assign these to `undef`.

```
my ($adenine, $cytosine, $guanine, $thymine) = qw(A C G T);
### $adenine
### $cytosine
### $guanine
### $thymine

my ($one) = qw(A C G T);
### $one
my ($first, @others) = qw(A C G T);
### $first
### @others
my (undef, $second, undef, $fourth) = qw(A C G T);
### $second
### $fourth

# gives:

### $adenine: 'A'
### $cytosine: 'C'
### $guanine: 'G'
### $thymine: 'T'

### $one: 'A'

### $first: 'A'
### @others: [
###             'C',
###             'G',
###             'T'
###           ]

### $second: 'C'
### $fourth: 'T'
```

BOX 1: Operator precedence in list definitions

```
my @numbers = (5, 6, 7, 8);
### @numbers

my @number = 5, 6, 7, 8;    # ouch! '=' has bigger precedence than ','
### @number

my $answer = 42;
my $count = ( my @more_numbers = (@numbers, $answer) );
### @numbers
### @more_numbers
### $count

# gives:

### @numbers: [
###             5,
###             6,
###             7,
###             8
###           ]
```

```
Useless use of a constant (6) in void context at lists.pl line 15.
Useless use of a constant (7) in void context at lists.pl line 15.
Useless use of a constant (8) in void context at lists.pl line 15.
```

```
### @number: [
###            5
###          ]

### @numbers: [
###             5,
###             6,
###             7,
###             8
###           ]
### @more_numbers: [
###                  5,
###                  6,
###                  7,
###                  8,
###                  42
###                ]
### $count: 5
```

1.3 Storing lists in arrays

It might look subtle but you should not conflate **lists** and **arrays**. Lists are (collections of) *values*, whereas arrays are *containers*. Think of it as follows: lists are book collections, whereas arrays are bookcases. You may store a list in an array, just as you may store a collection of books in a bookcase. Inversely, you may produce a list from an array, just as you may take a collection of books out of a bookcase. However, book collections and bookcases are intrinsically separate entities.

Arrays store only scalar values. These are called **array elements**. You already know how to *iterate* over the elements of an array using the *foreach-style* `for` loop. You also know how to *add* or *remove* elements to or from an array using list-oriented builtin functions (`push`, `unshift`, `pop` and `shift`). Now, let's examine how to *access* some specific array elements.

Figure 1.1: Lists are values, whereas arrays are containers.

Array elements are referred to using numeric **indices** between **square bracket characters** (`[` and `]`). The first element of the array has index zero (`0`), while the last element has index *length-of-the-array-minus-one*. This last index value can always be given as `$#array_name`. You may also count backwards from the end of the array using negative indices starting at minus one (`-1`) for the last element. Note that this is different from, e.g., the **R** language, in which negative indices denote exclusion.

```
my @bases = qw(A C G T);

### $bases[0]: $bases[0]
### $bases[1]: $bases[1]
### $bases[2]: $bases[2]
### $bases[3]: $bases[3]

### $bases[$#bases]: $bases[$#bases]

### $bases[-1]: $bases[-1]
### $bases[-2]: $bases[-2]
### $bases[-3]: $bases[-3]
### $bases[-4]: $bases[-4]

# gives:

### $bases[0]: 'A'
### $bases[1]: 'C'
### $bases[2]: 'G'
### $bases[3]: 'T'

### $bases[$#bases]: 'T'

### $bases[-1]: 'T'
### $bases[-2]: 'G'
### $bases[-3]: 'C'
### $bases[-4]: 'A'
```

Elements can be manipulated as individual entities (scalar context), as in the examples above, or as collective, though sometimes disjoined, entities (list context). The latter are called **array slices** and are particularly powerful. Consider the following examples.

```
my @bases = qw(A C G T);

### @bases[0..1]: @bases[0..1]
### @bases[0,1] : @bases[0,1]
### @bases[1,0] : @bases[1,0]
### @bases[1,3] : @bases[1,3]
### @bases[-1,3]: @bases[-1,3]

# gives:

### @bases[0..1]: 'A',
###               'C'
### @bases[0,1] : 'A',
###               'C'
### @bases[1,0] : 'C',
###               'A'
### @bases[1,3] : 'C',
###               'T'
### @bases[-1,3]: 'T',
###               'T'
```

Since we retrieve more than one element in a single operation, we are in list context. That is why the sigil of the `@bases` array switches from `$` to `@`. As you can see, array slices can either be specified as enumerated lists of indices or using the range operator. They can mix and match different indexing schemes, select non-contiguous elements and even select the same element more than once (examine the last example). Slices are especially useful when dynamically computed, i.e., stored in variables.

BOX 2: Array slices and the amount context

Observe how the `perl` interpreter warns you of the inadequate sigil when you *should know* that you are in scalar context, but not when you cannot determine the amount context in advance.

```
my @bases = qw(A C G T);
my @gc_slice = 1..2;
### @bases[@gc_slice]: @bases[@gc_slice]

my $lone_slice = 3;
### @bases[$lone_slice]: @bases[$lone_slice]

my @dyn_slice = (0);
### @bases[@dyn_slice]: @bases[@dyn_slice]

# gives:

### @bases[@gc_slice]: 'C',
###                    'G'
```

```
Scalar value @bases[$lone_slice] better written as $bases[$lone_slice] \
    at slice.pl line 11.
```

```
### @bases[$lone_slice]: 'T'

### @bases[@dyn_slice]: 'A'
```

Individual array elements and array slices can be the object of assignment operations. An existing array can be reset to zero elements by assigning the **empty list** (`()`) to it. This is useful for persisting arrays, but lexically-scoped arrays rarely need to be explicitly emptied.

```
my @bases = qw(A C G T);
### @bases
$bases[-1] = 't';
### @bases
$bases[1] = qw(c);
### @bases
@bases[0,2] = qw(a g);
### @bases
@bases = ();
### @bases
```

```
# gives:

### @bases: [
###           'A',
###           'C',
###           'G',
###           'T'
###         ]

### @bases: [
###           'A',
###           'C',
###           'G',
###           't'
###         ]
```

BOX 3: Accessing individual values in lists

Values in a list can also be accessed individually using indices between square brackets, just as you would do with an array. For this to work, enforce the list context using a pair of parentheses.

```
my $dna_string = 'CATGAACTTCTTTGGCGTCTTGAT';
my $first = (split //, $dna_string)[0];
### $first

gives:

### $first: 'C'
```

Of course, this works with slices too.

```
my ($second, $fourth) = (split //, $dna_string)[1,3];
### $second
### $fourth

gives:

### $second: 'A'
### $fourth: 'G'
```

However, you can only *read* the values of a list, not *assign* them new values (because they are already values and not storage slots like the elements of an array).

```
(split //, $dna_string)[0] = 'A';   # will cause a compilation error

gives:

Can't modify list slice in scalar assignment at list.pl line 8, near "'A';"
Execution of list.pl aborted due to compilation errors.
```

```
### @bases: [
###           'A',
###           'c',
###           'G',
###           't'
###         ]

### @bases: [
###           'a',
###           'c',
###           'g',
###           't'
###         ]

### @bases: []
```

1.4 Storing lists in hashes

Lists can be stored in hashes provided that they have an *even* number of values. If you try to assign an *odd* number of values to a hash, you will receive a warning. This is so because, in hash assignment, list values have to be taken two at a time, the first one becoming the **key** and the second one becoming the associated **value**.

```
my %comp_for = (
    A => 'T',   T => 'A',   G => 'C',   C => 'G',
    a => 't',   t => 'a',   g => 'c',   c => 'g',
);
```

In principle, you may use the regular comma operator (`,`) when defining a hash. However, the **fat comma operator** (=>) makes the pairing more visible. It also automatically quotes the keys. Thus, the previous hash definition is equivalent to the (less readable) following one.

```
my %comp_for = (
    'A', 'T',   'T', 'A',   'G', 'C',   'C', 'G',
    'a', 't',   't', 'a',   'g', 'c',   'c', 'g',
);
```

Finally, if you have key/value pairs in two different lists of the same size, you can build a hash from them using the `mesh` function exported by `List::MoreUtils`. It is also known as the `zip` function. Both are equally available in the more convenient `List::AllUtils` module.

```
use List::AllUtils 'mesh';

my @genera = qw(human yeast arabidopsis amoeba);
my @groups = qw(Opisthokonta Opisthokonta Plantae Amoebozoa);

my %taxon_for = mesh @genera, @groups;
### %taxon_for
```

```
# gives:

### %taxon_for: {
###                amoeba => 'Amoebozoa',
###                arabidopsis => 'Plantae',
###                human => 'Opisthokonta',
###                yeast => 'Opisthokonta'
###              }
```

Everything we have just explained about manipulating array elements also applies to hashes, except that hash values are referred to using string keys between **curly brace characters** ({ and }) instead of numeric indices between square brackets. So, there also exist **hash slices**, which are extremely powerful when dynamically computed and/or used in hash assignment.

```
### $taxon_for{human}: $taxon_for{human}
### @taxon_for{ qw(human yeast) }: @taxon_for{ qw(human yeast) }

@taxon_for{ qw(human yeast) } = qw(Metazoa Fungi);
### %taxon_for

# gives:

### $taxon_for{human}: 'Opisthokonta'
### @taxon_for{ qw(human yeast) }: 'Opisthokonta',
###                                'Opisthokonta'

### %taxon_for: {
###                amoeba => 'Amoebozoa',
###                arabidopsis => 'Plantae',
###                human => 'Metazoa',
###                yeast => 'Fungi'
###              }
```

Single hash keys do not need to be quoted between curly braces and can be specified as **barewords**. In hash slices, individual keys have to be single-quoted, e.g., `@taxon_for{ 'human', 'yeast' }`. When composed of single words, use the `qw(...)` operator for better legibility (as in the example above).

1.5 Working with lists

Idiomatic Perl makes heavy use of lists. In subsequent lessons, we will discuss a few builtin functions that are specially devoted to their manipulation (see for example `grep` and `map` in "Implicit loops", p.140). For now, let's just review the list-oriented functions used in `check_overlap.pl`. All of them are available in `List::AllUtils`. Since we want to import several functions at once, we need to specify a list, hence the use of the `qw(...)` operator.

```
use List::AllUtils qw(min max uniq);
```

Consider the format of the input file below.

```
# inclusions
10  60  10  60
10  60  30  60
30  60  10  60
10  40  10  60
10  60  10  40
10  60  30  50
30  50  10  60

# left overhangs
10  50  30  60
10  50  50  60
...
```

Empty lines and comment lines are allowed. We thus need to skip them.

```
next LINE if $line =~ m/^ \s* $/xms;        # skip empty lines
next LINE if $line =~ m/^ \#/xms;           # skip comment lines
```

Each data line specifies two *genomic intervals*, each one corresponding to one fragment. We first split the line on whitespace characters to extract the four individual coordinates. Because of the regular expression, they may be separated by one or more space(s) or tab character(s). Then, we copy them into four discrete variables using list assignment.

```
# extract coordinates
chomp $line;
my @coords = split /\s+/xms, $line;
my ($x1, $x2, $y1, $y2) = @coords;
```

This allows us to refer to individual coordinates while still being able to work in list context if more convenient. Hence, to determine the lower and higher bounds of our scale, list context is ideal.

```
# setup plot scale
my $min = min @coords;
my $max = max @coords;
```

In contrast, fragment overlap tests are easier to spell out using individual coordinates.

```
my @overlaps = (
# using high-precedence logical operators
    (   ($y1 >= $x1 && $y1 <= $x2)          #  left cases
     || ($x1 >= $y1 && $x1 <= $y2) ),       # right cases

# mixing high- and low-precedence logical operators
    (     $y1 >= $x1 && $y1 <= $x2          #  left cases
     or   $x1 >= $y1 && $x1 <= $y2  ),      # right cases

# ... or more intelligently...
    (not $y1 >  $x2 || $y2 <  $x1  ),       # test disjunction
);
```

In the previous chunk of code, we actually perform three variants of the same test. The boolean results of each test are stored in the `@overlaps` array, which is thus a three-element array containing boolean values. To ensure that all tests return the same result, we ask for the *unique* values in the array.

```
# output overlap status
@overlaps = uniq(@overlaps);
### assert: @overlaps == 1
```

If all three tests succeed or fail, the `uniq` function returns a list containing a single value (equivalent to *true* or *false*). Otherwise, it returns a list of two values (one *true* and one *false*). We will come back to the `assert` smart comment in “More on `Smart::Comments`”, p.25.

Chapter 2

Formatting output

2.1 `check_overlap.pl` output sample

This section is heavily based on `check_overlap.pl`. To help you to follow the upcoming explanations, I show you below an excerpt of a typical run of the program.

```
$ check_overlap.pl coords.txt 8

10      16      22      28      34      40      46      52      58      64
|       |       |       |       |       |       |       |       |       |
--------------------------------------------------
10                                                                60
####################################################################
####################################################################
10                                                                60
---> 10-60 and 10-60 DO overlap!

10      16      22      28      34      40      46      52      58      64
|       |       |       |       |       |       |       |       |       |
--------------------------------------------------
10                                                                60
####################################################################
                           #########################################
                           30                                     60
---> 10-60 and 30-60 DO overlap!
...
```

2.2 How to round numbers?

In `check_overlap.pl`, we need to round numbers in different ways depending on our objectives:

1. drawing the plotting axis,
2. computing the drawing scale,
3. positioning the fragments.

2.2.1 `floor` and `ceil`

First, given the width of the screen (in characters) and a user-specified tic width (also in characters), we determine how many tics we can plot on our axis. This requires rounding to *the largest integer value less than or equal to* the ratio between terminal width and tic width, which is achieved with the `floor` function provided by the `POSIX` module.

```
use POSIX;

# and later...
my $tic_width = shift;

my ($cols, $rows) = chars();
my $tic_n = floor( $cols / $tic_width );
```

Second, for a given input line, once we know the lower and upper bounds of our drawing scale based on fragment coordinates, we compute the drawing scale. This corresponds to the ratio between the coordinate range and the number of tic intervals (i.e., number of tics minus one). To avoid falling too short for the last tic value, we round this ratio to *the smallest integer value greater than or equal to* this number (`ceil` function).

```
my $range = $max - $min;
my $step = ceil( $range / ($tic_n-1) );
```

2.2.2 `sprintf`

Third, when computing the position of the beginning or the end of a fragment based on the current drawing scale, we need to round our transformed fragment coordinate to *the closest integer*. We do that using another Perl workhorse, the `sprintf` builtin function. Don't worry about `return`: we will cover it in the section about functions (see "`return`", p.29).

```
return sprintf "%.0f", ($x-$min) / $step * $tic_width;
```

`sprintf` is incredibly powerful and complex. It also exists as `printf`. The only difference is that the output of `printf` goes to the screen (or to a file), whereas `sprintf` returns a string that can be further processed and/or stored in a variable. Both builtin functions expect at least two arguments: one *format* string and a list of one or more values to format.

Here, I will not describe all the possibilities of `sprintf`. Instead, I will explain how we used it so far and refer you to the documentation for the gory detail:
http://perldoc.perl.org/functions/sprintf.html

In the statement above, we format our screen coordinate as a floating-point number (`%f`) with zero decimal digits (`.0`), which has the effect of rounding it in a mathematically-correct way. See:
http://en.wikipedia.org/wiki/Rounding#Round_half_to_even

In `codon_usage.pl`, we used something similar to limit the precision of our usage percentages to one digit after the decimal point. We also asked for a total width of five characters (including the decimal point) to align the resulting numbers to the right.

```
say {$out} join "\t", $codon, $count, $aa, $total,
    sprintf "%5.1f", $usage;
```

```
# gives:

# codon count   aa      total   usage
AAA     46021   K       60126    76.5
AAC     29496   N       53737    54.9
...
ATA     5959    I       81870     7.3
ATC     34384   I       81870    42.0
ATG     37917   M       37917   100.0
...
GAC     26136   D       70121    37.3
GAG     24399   E       78482    31.1
```

Such *width specifications* can be quite complex. For example, in `check_overlap.pl`, we print the actual tic values using a *constant* width so that they nicely line up above the tics symbols. This is done by formatting these values as integer numbers (`%d`), aligned to the left (`-`) but consuming a total of exactly `$tic_width` characters (`*` followed by a number at the corresponding position in the value list).

```
$str .= sprintf("%-*d", $tic_width, $x) unless $bars;
```

This statement is part of a pretty hairy *C-style* `for` loop drawing the plotting scale, the decoding of which is left as an exercise to the reader!

```
my $str;
for (my ($x, $tic) = ($min, 0); $tic < $tic_n; $x += $step, $tic++) {
    $str .= sprintf("%-*d", $tic_width, $x) unless $bars;
    $str .= '|' . q{ } x ($tic_width-1)        if $bars;
}
```

2.3 The repetition operator

In `check_overlap.pl`, we position things using a *padding* strategy. This means that we print as many whitespace characters as needed to reach the right place before printing what we want to print. This is clearly *old-school* but useful to illustrate the use of the **repetition operator** (`x`).

The behavior of this infix operator is relatively complex to master. Thus, you'd better to limit yourself to two use cases:

1. repeating a scalar in scalar context,
2. repeating a list in a list context.

This is the first case that we use for padding. Evaluating the following expressions results in strings *concatenating* the first operand as many times as specified by the second operand.

```
my $fragm_str = q{ } x $pad_n . q{#} x $chr_n;
...
my $coord_str = q{ } x $pad_n . $x1 . q{ } x $spc_n . $x2;
```

Here's a detailed execution for a better understanding.

```
my $pad_n = xloc($x1, @_);
my $chr_n = xloc($x2, @_) - $pad_n + 1;
my $fragm_str = q{ } x $pad_n . q{#} x $chr_n;

my $spc_n = $chr_n - length($x1) - length($x2);
   $spc_n = 0 if $spc_n < 0;
my $coord_str = q{ } x $pad_n . $x1 . q{ } x $spc_n . $x2;

# gives:

### $x1: 10
### $x2: 25
### $pad_n: 13
### $chr_n: 41
### $spc_n: 37
### $fragm_str: '             #########################################'
### $coord_str: '             10                                     25'
```

Maybe easier, the statements to plot the axis line.

```
my ($cols, $rows) = chars();

# and later...
say q{-} x $cols;
```

As an aside, the `chars` function returns a list of two values corresponding to the dimensions of the terminal window (*width* x *height*). It was imported from the `Term::Size::Any` module.

```
use Term::Size::Any 'chars';
```

The second use case is very similar to the first one, except that there is no concatenation. Instead, the expression below returns a large list composed of multiple identical sublists. The returned list is said to be **flattened** because sublists are not **nested**: there is only a single level of values.

```
my @triplets = (1, 2, 3) x 5;
### @triplets;

# gives:

### @triplets: [
###              1,
###              2,
###              3,
###              1,
###              2,
###              3,
###              1,
###              2,
###              3,
###              1,
###              2,
###              3,
###             1,
###             2,
###             3
###           ]
```

2.4 More on `Smart::Comments`

In our last programs, we have begun to use smart comments for more than just debugging our code. For example, in `codon_usage.pl`, we use them to report progress to the user.

```
### Reading input file: $infile

### Building hash for standard code...

### Processing: keys(%seq_for) . ' sequences...'

for my $dna_string (values %seq_for) {      ### Elapsed time |===[%]
    # loop body
}

### Computing codon usage statistics...

### Writing output file: $outfile
```

Observe how we embed variables (`$infile`, `$outfile`) and Perl expressions (`keys(%seq_for)`) in our comments for better reporting. We also include the special construct `### ... |===[%]` to get an animated **progress bar** during the loop.

In `xxl_xlate.pl`, we use the same approach with one difference, though: *progress comments* are preceded by three comment characters (`###`), whereas *debugging comments* are preceded by four of them (`####`). Further, we load `Smart::Comments` with an explicit *level of smartness*, here three.

```
use Smart::Comments '###';

### Reading input file: $infile

### Building hash for code: $gc_id . '...'

#### $gc_content
#### @codes
#### %aa_for

### Translating: keys(%seq_for) . ' sequences...'

while (my ($id, $dna_string) = each %seq_for) {     ### Elapsed time |===[%]
    # loop body
}

### Writing output file: $outfile->stringify
```

As a result, normal execution of this program only prints progress comments. If we want to debug our code (and see the content of `@gc_content`, `@codes` and `%aa_for`), we can invoke it with a finer level of smartness, either by changing the `use` statement…

```
use Smart::Comments ('###', '####');
```

… or by directly specifying it on the command line!

```
$ perl -M"Smart::Comments ('###','####')" ./xxl_xlate.pl ...
```

Finally, in `check_overlap.pl`, we use smart comments **assertions** to ensure correct execution of error-prone statements (**defensive programming**). Consider the following chunk of code.

```
# output overlap status
@overlaps = uniq(@overlaps);
### assert: @overlaps == 1
say "---> $x1-$x2 and $y1-$y2 "
    . (shift @overlaps ? 'DO' : 'DO NOT')
    . ' overlap!'
    . "\n"
;
```

If the three tests do not agree, we get a list of two values. Thus, the conditional expression of the smart comment evaluates to a false value, which causes the program to halt with an error message due to the `assert` function. Beware that assertions are only evaluated at their smartness level.

```
### assert: @overlaps == 1

### @overlaps == 1 was not true at ./check_overlap.pl line 97, <$_[...]> line 19.
###     @overlaps was: [
###                      '',
###                      1
###                    ]
```

If the three tests agree, the `say` statement gets executed. It uses several idioms that make it very compact but not so easy to understand.

```
say "---> $x1-$x2 and $y1-$y2 "
    . (shift @overlaps ? 'DO' : 'DO NOT')
    . ' overlap!'
    . "\n"
;
```

The parenthesis characters enclose an expression involving the ternary conditional operator. Its purpose is to select between two strings: `'DO'` and `'DO NOT'`. The expression evaluates as follows…

1. The first (actually *unique*) element of `@overlaps` is taken out of the array using `shift`.
2. If it is *true*, the conditional expression evaluates to *true* and the whole ternary conditional expression takes the string value `'DO'`. Otherwise, the conditional expression evaluates to *false* and the whole expression takes the string value `'DO NOT'`.
3. The selected string is then *concatenated* with the preceding and following strings given in the `say` statement before getting printed to the screen.

Chapter 3

Functions

3.1 What are functions?

From *Modern Perl* by chromatic:
http://modernperlbooks.com/books/modern_perl_2014/

A function (or subroutine) in Perl is a discrete, encapsulated unit of behavior. A program is a collection of little black boxes where the interaction of these functions governs the control flow of the program. A function may have a name. It may consume incoming information. It may produce outgoing information.

Functions are a prime mechanism for abstraction, encapsulation, and re-use in Perl 5.

Figure 3.1: The famous black box

3.2 Defining and using functions

`check_overlap.pl` uses several functions. Some of them are defined for *abstraction* and *encapsulation* (`plot_scale`), whereas others are moreover defined for *re-use* (`plot_fragment`, `step_line` and `xloc`). All are defined with the `sub` keyword, followed by the **function name**, and then by the **function body** specified as a block surrounded by curly brace characters.

```
sub plot_scale {
    # function body
}

sub plot_fragment {
    # function body
}

sub step_line {
    # function body
}

sub xloc {
    # function body
}
```

To **call** a function, simply use its name immediately followed by a pair of parenthesis characters. For maximum robustness, it is safer not to put whitespace characters before the opening parenthesis.

```
plot_scale()        # OK
plot_scale ()       # could fail
```

If the function expects **parameters**, put them between the trailing parentheses.

```
plot_scale($min, $step);
plot_fragment($x1, $x2, 0, $min, $step);
plot_fragment($y1, $y2, 1, $min, $step);
```

Otherwise, leave the parentheses empty, as in the example above. Empty parentheses can be omitted completely if the function has been declared (or imported from a module) before the call.

```
plot_scale          # will fail unless declared beforehand
chars               # OK if imported from Term::Size::Any
```

From the point of view of the *calling code*, function parameters are called **arguments**. These can be any Perl expression, including literal values and variables. The semantic distinction between parameters and arguments is quite subtle and often overlooked but nevertheless important in computer theory.

Function names should be chosen after *action verbs* and be as explicit as possible to contribute to the autodocumentation of your code (e.g., `plot_scale`, `plot_fragment`).

As a rule of thumb, place all your functions at the end of your program file and order them by decreasing importance (from the most abstract to the most practical). This is known as the *top-down* approach.

3.3 Function parameters and return values

3.3.1 The default array `@_`

A Perl function receives its parameters in the default array `@_`. You access them using numeric indices.

```
print_dna('CATGAACTTCTTTGGCGTCTTGAT');

sub print_dna {
    say 'Your DNA sequence is: ' . $_[0];
}

# gives:
```

```
Your DNA sequence is: CATGAACTTCTTTGGCGTCTTGAT
```

3.3.2 `return`

By default, a function call evaluates to the value of the last expression evaluated in the function. However, relying on that behavior is a bad idea. Instead, prefer the more explicit `return` keyword.

```
my $dna_string = 'CATGAACTTCTTTGGCGTCTTGAT';
my $def_string = def_rev_comp($dna_string);
my $ret_string = ret_rev_comp($dna_string);

### $dna_string
### $def_string
### $ret_string

sub def_rev_comp {
    scalar reverse $_[0] =~ tr/ACGTacgt/TGCAtgca/r;             # avoid
}

sub ret_rev_comp {
    return scalar reverse $_[0] =~ tr/ACGTacgt/TGCAtgca/r;      # OK
}

# gives:

### $dna_string: 'CATGAACTTCTTTGGCGTCTTGAT'
### $def_string: 'ATCAAGACGCCAAAGAAGTTCATG'
### $ret_string: 'ATCAAGACGCCAAAGAAGTTCATG'
```

3.3.3 Argument aliasing

Be careful that the default array `@_` is akin to iterators in *foreach-style* `for` loops. This means that it actually *aliases* the arguments passed to the function, which may thus be damaged by accident.

```
my $dna_string = 'CATGAACTTCTTTGGCGTCTTGAT';
my $rev_string = sloppy_rev_comp($dna_string);
```

```
### $dna_string
### $rev_string

sub sloppy_rev_comp {
    $_[0] =~ tr/ACGTacgt/TGCAtgca/;         # forgot /r
    return scalar reverse $_[0];
}

# gives:

### $dna_string: 'GTACTTGAAGAAACCGCAGAACTA'
### $rev_string: 'ATCAAGACGCCAAAGAAGTTCATG'

# ouch! $dna_string has been complemented by accident...
```

To avoid such issues, the first thing to do is to copy the function arguments in lexically-scoped variables. This can be done in two ways:

1. through one or more `shift` statements,
2. through a single list assignment.

Here's an example using the first approach.

```
my $dna_string = 'CATGAACTTCTTTGGCGTCTTGAT';
my $rev_string = robust_rev_comp($dna_string);

### $dna_string
### $rev_string

sub robust_rev_comp {
    my $dna = shift;                        # lexical copy of $dna_string

    $dna =~ tr/ACGTacgt/TGCAtgca/;          # forgot /r
    return scalar reverse $dna;
}

# gives:

### $dna_string: 'CATGAACTTCTTTGGCGTCTTGAT'
### $rev_string: 'ATCAAGACGCCAAAGAAGTTCATG'
```

As you can see, it is not necessary to mention the default array `@_` when using `shift`. This is so because it is automatically implied in `shift` calls from within function definitions. Outside functions, however, remember that it is the special array `@ARGV` that is implied, as explained in "`unshift` & `shift`" (see the first part of this course). Many other list-oriented Perl builtin functions display the same behavior.

3.3.4 Lexical and file variables

If a function defines variables with the `my` keyword, even unrelated to the arguments, these variables are also lexically-scoped to the function body. This is very useful for encapsulating your code and is thus an absolute *must-do* practice.

```
sub plot_fragment {
    my $x1 = shift;

    # ...

    # here's a lexical variable scoped to the sub
    my $fragm_str = q{ } x $pad_n . q{#} x $chr_n;

    # ...
}
```

Moreover, this does not prevent you from using file-scoped variables defined outside of any block when it is more convenient instead of passing them as parameters (e.g., `$tic_width` in `xloc`).

```
my $tic_width = shift;

# and later...
sub xloc {
    my ($x, $min, $step) = @_;
    return sprintf "%.0f", ($x-$min) / $step * $tic_width;
}
```

3.3.5 Argument slurping

The second way of copying function arguments uses list assignment, as shown in functions `step_line` and `xloc`. This style is useful when a function either needs all the elements of `@_` or can discard those that it does not require. This works because of the greediness of list assignment.

```
sub step_line {
    my ($bars, $min, $step) = @_;

    # function body
}

sub xloc {
    my ($x, $min, $step) = @_;

    # function body
}
```

3.3.6 Argument currying

In some cases, a function actually needs only some of its arguments, while the others are passed unmodified to another function. For example, `plot_fragment.pl` could have been written like this.

```
sub plot_fragment {
    my ($x1, $x2, $second, $min, $step) = @_;

    my $pad_n = xloc($x1, $min, $step);
    my $chr_n = xloc($x2, $min, $step) - $pad_n + 1;

    # ...
}

sub xloc {
    my ($x, $min, $step) = @_;
    return sprintf "%.0f", ($x-$min) / $step * $tic_width;
}
```

However, this is a bit silly to extract `$min` and `$step` just to pass them *as-is* to the `xloc` function. That is why I used the alternative version shown below. Passing some of (or all) incoming arguments to another function directly from `@_` is known as **argument currying**. Even if it is a quite advanced concept, it is not that difficult to implement in Perl. Of course, the order of parameters has to be well thought out for it to work.

```
sub plot_fragment {
    my $x1 = shift;
    my $x2 = shift;
    my $second = shift;

    my $pad_n = xloc($x1, @_);
    my $chr_n = xloc($x2, @_) - $pad_n + 1;

    # ...
}
```

The `plot_scale` function also uses currying, but to an even greater extent since it does not use any of its arguments and passes them all to the `step_line` function. Observe how `plot_scale` triggers two different behaviors of the latter function by inserting a boolean flag before the arguments in `@_`.

```
sub plot_scale {
    say step_line(0, @_);
    say step_line(1, @_);
    say q{-} x $cols;
    return;
}

sub step_line {
    my ($bars, $min, $step) = @_;

    # function body
    # $bars is a boolean flag selecting between the two behaviors
}
```

3.3.7 Bare `return` statements

At the end of `plot_scale`, there is bare `return` statement. This will return the `undef` value to the caller. Though it is not strictly required, I advise you to include at least one `return` in all your functions, even those that do not return any value. This improves readability and avoids returning automatically the value of the last expression, which could lead to undesired effects in some cases.

```
my ($cols, $rows) = chars();

my $def = plot_def();
my $ret = plot_ret();

### $def
### $ret

sub plot_def {
    say q{-} x $cols;
}

sub plot_ret {
    say q{-} x $cols;
    return;
}

# gives:

----------------------------------------------
----------------------------------------------

### $def: 1
### $ret: undef
```

3.3.8 Argument flattening

Perl *flattens* the function arguments in a single list before passing them to the function. This is often an issue for novices, but one can work around that (see "Interesting bits in `Forem::FastaFile`", p.152).

```
my $type = 'DNA';
my @bases = qw(A C G T);
my $strand = '+';

my $result = analyze_this($type, @bases, $strand);

sub analyze_this {
    my ($type, @bases, $strand) = @_;
    ### @_
    ### $type
    ### @bases
    ### $strand
}
```

```
# gives:

### @_: [
###       'DNA',
###       'A',
###       'C',
###       'G',
###       'T',
###       '+'
###     ]
### $type: 'DNA'
### @bases: [
###           'A',
###           'C',
###           'G',
###           'T',
###           '+'
###         ]
### $strand: undef
```

In this particular case, one straightforward solution is simply to put the array in last position, but this would not work had we to pass more than one container to the function.

```
my $type = 'DNA';
my @bases = qw(A C G T);
my $strand = '+';

my $result = analyze_this($type, $strand, @bases);

sub analyze_this {
    my ($type, $strand, @bases) = @_;
    ### @_
    ### $type
    ### $strand
    ### @bases
}

# gives:

### @_: [
###       'DNA',
###       '+',
###       'A',
###       'C',
###       'G',
###       'T'
###     ]
### $type: 'DNA'
### $strand: '+'
### @bases: [
###           'A',
###           'C',
###           'G',
###           'T'
###         ]
```

3.4 Bonus—Computing anagrams using recursive function calls

Even though not enclosed in a blue box, this section deals with more advanced concepts. If you do not want to dig into these right now, you can skip it and proceed directly to "Homework", p.39.

Some problems are better solved using a programming approach known as **recursion**. In short, a *recursive algorithm* uses a function that calls itself until the job is done. The textbook example is the recursive computation of the *factorial* function (*n!*).

```
#!/usr/bin/env perl

use Modern::Perl '2011';

die "Usage: $0 <n>" unless @ARGV == 1;

my $n = shift;
say "$n! is " . fact($n);

sub fact {
    my $n = shift;
    return 1 if $n == 0;        # termination condition
    return fact($n-1) * $n ;    # recursive call
}
```

```
$ perl fact.pl 5
```

```
5! is 120
```

This algorithm works because each call to `sub fact` has its own private version (i.e., lexically scoped) of the variable `$n`. To convince yourself, just drop the `my` keyword from the function definition.

```
$ perl fact_no_my.pl 5
```

```
5! is 0
```

Albeit classic, I do not like the recursive algorithm for computing the factorial due to it being both overkill and inefficient. *Overkill* because a mere `while` loop would be enough and conceptually much simpler. *Inefficient* because such a simple loop requires less resources than the recursive version. Indeed, proper recursion requires a private environment for each function call (**reentrancy**) and deeply nested recursive function calls are generally considered suspicious by the `perl` interpreter.

```
sub fact_for {
    my $n = shift;
    my $f = 1;
    while ($n > 0) { $f *= $n-- }
    return $f;
}
```

```
$ perl fact_for.pl 100
```

```
100! is 9.33262154439442e+157
```

```
$ perl fact.pl 100
```

```
Deep recursion on subroutine "main::fact" at fact.pl line 13.
100! is 9.33262154439441e+157
```

In combinatorics (an area of mathematics), the factorial function is used to compute the number of *permutations* of the elements of a list. Therefore, a better example of a recursive algorithm would be the one allowing us to enumerate all these permutations.

If we consider a word as a simple list of letters, then such an algorithm would allow us to enumerate all the *anagrams* for this word (whether biological or linguistic).

```
$ perl anagrams.pl ATG
```

```
AGT
ATG
GAT
GTA
TAG
TGA
```

As expected by the value of *3!* (for a list of size 3), there are 5 anagrams (i.e., a total of 3 x 2 x 1 = 6 permutations) for the word corresponding to the `ATG` codon.

However, when the word contains repeated letters, some permutations become redundant and there are less anagrams than predicted by the factorial function. You can see this with the `TATA` word, where 24 (*4!*) permutations reduce to 6 due to both `A` and `T` being used twice.

```
$ perl anagrams.pl TATA
```

```
AATT
ATAT
ATTA
TAAT
TATA
TTAA
```

Our program properly deals with such cases because it loops on a list of unique letters and keeps track of how many times each of these letters has been used so far in the word being assembled.

Here is the code for `anagrams.pl`. Most of its logic is explained in plain English in dumb comments. To get a better idea of how it works, run in with `Smart::Comments` enabled.

```
#!/usr/bin/env perl

use Modern::Perl '2011';
use Smart::Comments '###';          # remove '###' to enable all comments

die "Usage $0 <word>" unless @ARGV == 1;

### Reading input...
# get input word and count occurrences of each of letter
my $word = shift;
my $len  = length $word;
#### $len

my %is_available;
for my $letter (split //, $word) {
    $is_available{$letter}++;
}
#### %is_available
my @letters = sort keys %is_available;
#### @letters

### Computing anagrams...
# main call to recursive subroutine
# we start with an empty word at depth (= length) 0 and all letters available
my @anagrams;
add_letter( q{}, 0, %is_available );

### Printing results...
# print all computed anagrams
say join "\n", @anagrams;

# recursive subroutine in charge of adding a letter to an anagram
sub add_letter {
    my $anagram = shift;
    my $depth   = shift;
    my %is_available = @_;
    #### entering depth: $depth
    #### $anagram

    # if current anagram is complete store it and leave current extension
    # we could also compare length $anagram to $len but understanding
    # the depth concept is important in a recursive algorithm
    if ($depth == $len) {                    # termination condition
        push @anagrams, $anagram;
        #### finished anagram at depth: $depth
        return;
    }
    #### %is_available

    # otherwise try each letter in turn for current position in anagram
    for my $letter (@letters) {

        # use the letter only if it still is available
        # removing this condition makes every letter available as many times
        # as there are positions in the anagram; you can try with or without
        # letters in multiple occurrences
        if ( $is_available{$letter} ) {
            #### adding: "$letter at depth $depth"

            # extend current anagram with available letter
            add_letter(                         # recursive call
                $anagram . $letter,
                $depth + 1,
                (%is_available, $letter => $is_available{$letter} - 1)
            );  # here we decrement the availability of the used letter
        }       # but in a private copy of the hash (not the global one)
    }

    #### finished looping at depth: $depth
    return;
}
```

Homework

Write a new version of our translation tool (`hw6_xxl_xlate_subs.pl`) that uses distinct functions for:

- the FASTA file reader,
- the genetic code builder,
- the conceptual translation of a single DNA sequence.

Here are the specifications of each function.

`read_fasta` Expects a file path to a FASTA file to read. Returns the hash `%seq_for` (see `xxl_xlate.pl`). Be sure to recover the ordered hash into another ordered (tied) hash; otherwise, you will loose the order of the sequences implied by the FASTA file.

`get_genetic_code` Expects a file path to the NCBI `gc.prt` file (or the `--remote` option) and an integer number giving the id of the requested code. Returns the hash `%aa_for` (see `xxl_xlate.pl`).

`translate` Expects a string containing a DNA sequence to translate, an integer giving the reading frame (`1`, `2`, `3`, `-1`, `-2`, `-3`) and the hash of the code (`%aa_for`). Returns a string with the protein.

Part II

Lesson 7

Chapter 4

Sorted codon usage

4.1 A gentle introduction to references

4.1.1 Motivation

In Homework of Lesson 3 (see "The code for our own `codon_usage`", in the first part of this course), I asked you to sort the table of `codon_usage.pl` on the amino acids rather than on the codons.

Below is an *alternative ending* (director's cut?) for this program. Save a copy as `codon_usage_sort.pl` and replace the last part by the following lines of code.

```
### Computing codon usage statistics...

my @lines;
for my $codon (sort keys %count_for) {
    my $aa    =    $aa_for{$codon};
    my $count = $count_for{$codon};
    my $total = $total_for{$aa};
    my $usage = 100.0 * $count / $total;     # usage in percents
    my @fields = ($aa, $codon, $count, $total, sprintf "%5.1f", $usage);
    push @lines, \@fields;
}

# sort lines lexically by aa then numerically descending by usage
my @sorted_lines = sort {
    $a->[0] cmp $b->[0] || $b->[4] <=> $a->[4]
} @lines;


### Writing output file: $outfile

open my $out, '>', $outfile;

say {$out} '# ' . join "\t", qw(aa codon count total usage);
```

```
for my $line (@sorted_lines) {
    say {$out} join "\t", @$line;
}
```

If you try this new program, you will see that the table is now sorted by amino acid (first column) and then (within each amino acid group) by decreasing usage (fifth column). This is achieved by the `sort` statement in the middle of the code.

```
my @sorted_lines = sort {
    $a->[0] cmp $b->[0] || $b->[4] <=> $a->[4]
} @lines;
```

We will come back to this statement in a few moments because it is pretty hairy. For now, let's just have a look at the result and explain the general strategy.

```
# aa    codon   count   total   usage
*       TAA     2896    4653     62.2
*       TGA     1378    4653     29.6
*       TAG     379     4653      8.1
A       GCG     46056   129593   35.5
A       GCC     34946   129593   27.0
A       GCA     27685   129593   21.4
A       GCT     20906   129593   16.1
C       TGC     8785    15846    55.4
C       TGT     7061    15846    44.6
...
L       CTG     72134   145490   49.6
L       TTA     19016   145490   13.1
L       TTG     18673   145490   12.8
L       CTC     15208   145490   10.5
L       CTT     15121   145490   10.4
L       CTA     5338    145490    3.7
M       ATG     37917   37917   100.0
N       AAC     29496   53737    54.9
N       AAT     24241   53737    45.1
...
V       GTG     35812   96777    37.0
V       GTT     25064   96777    25.9
V       GTC     20966   96777    21.7
V       GTA     14935   96777    15.4
W       TGG     20885   20885   100.0
Y       TAT     22058   38750    56.9
Y       TAC     16692   38750    43.1
```

The idea is to first build an array called `@lines`, in which each of the 64 elements corresponds to a usage line. The trick is that these usage elements are themselves (anonymous) arrays. Since an array can only hold scalar values, we cannot store these 64 elemental arrays directly in `@lines`. Instead, we store a **reference** to each of them. Perl can then follow these references to access the elemental arrays at the corresponding **memory locations**. `@lines` is thus a **nested data structure**. It is shown in an abridged version in the left part of the figure below.

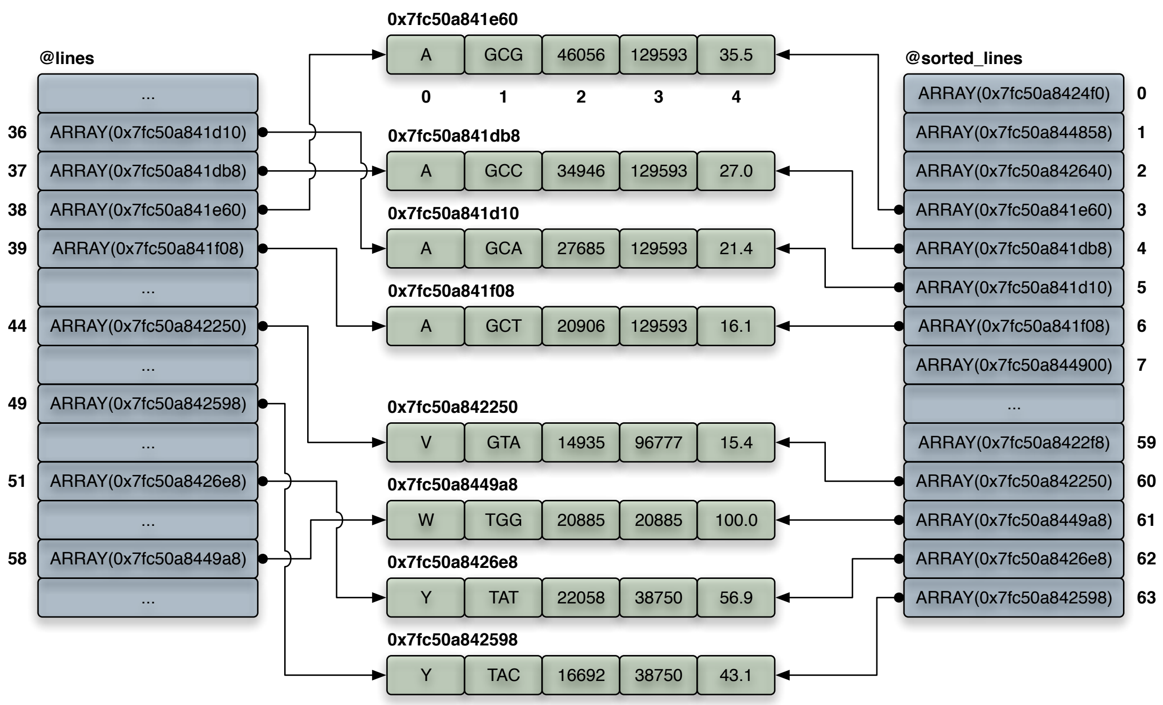

Figure 4.1: Using references to elemental arrays for sorting lines

The objective of this approach is to keep the five fields of each usage line *separate*, so that we can sort the lines on the fields of interest considered individually. After the `sort` statement, we store the returned sorted list in a new array aptly named `@sorted_lines`. As you can see in the right part of the figure, we have not modified the elemental arrays themselves, but only the order in which their references appear in our new array.

4.1.2 Defining references

A reference is a scalar variable that *refers* to another variable. To take a reference to something, use the **reference operator** (\). For example, the last line of the loop below takes a reference to the elemental array `@fields` and pushes it to the long-lasting array `@lines`.

```
# original version
my @lines;
for my $codon (sort keys %count_for) {
    # compute field values

    my @fields = ($aa, $codon, $count, $total, sprintf "%5.1f", $usage);
    push @lines, \@fields;
}
```

As `@fields` is lexically-scoped to its enclosing loop, it is freshly created at each loop iteration. In other words, whenever a new `@fields` spawns into existence, it is a completely distinct array from the previous ones that does not overwrite any of them. This allows us to append each `@fields` array in turn to `@lines` in order to progressively build our nested data structure.

4.1.3 Using references

To print the final table, we iterate over the references in `@sorted_lines`, follow them to access the corresponding elemental arrays and `join` their five individual fields with a tab character. Following a reference is termed **dereferencing**. The syntax for this operation can be quite noisy and is admittedly one of the weaknesses of Perl, though it is not the case here.

```
for my $line (@sorted_lines) {
    say {$out} join "\t", @$line;
}
```

Since a reference is a scalar, it always comes with its `$` sigil. If you want to dereference it, you need to prepend it with a second sigil, the nature of which depending on the reference type and on the amount context. In our example, the entity referenced by `$line` is an array that we want to use in a `join` statement. We are thus in list context, hence the use of the `@` sigil.

If you forget to dereference a reference, for example before printing it, it will stringify into a string providing the type of the variable referenced by the reference, followed by a long hexadecimal number (between parentheses) associated to its location in computer memory.

```
for my $line (@sorted_lines) {
    say {$out} join "\t", $line;        # forgot to dereference
}

# gives:

ARRAY(0x7fc50a8424f0)
ARRAY(0x7fc50a844858)
ARRAY(0x7fc50a842640)
...
ARRAY(0x7fc50a8449a8)
ARRAY(0x7fc50a8426e8)
ARRAY(0x7fc50a842598)
```

Stringified references are produced by concatenating the corresponding memory location (in hexadecimal notation) to the return value of the `ref` function. This builtin function takes a scalar variable and returns a description of the kind of reference it contains (e.g., `SCALAR`, `ARRAY`, `HASH`). When used on a variable that is not a reference, `ref` returns the `undef` value.

4.1.4 Anonymous arrays

For our strategy to work, the most straightforward approach is to declare `@fields` within the loop. To see why, move the declaration of `@fields` outside the loop and run the program again.

```
# alternative version 1 (buggy)
my @lines;
my @fields;
for my $codon (sort keys %count_for) {
    @fields = ($aa, $codon, $count, $total, sprintf "%5.1f", $usage);
    push @lines, \@fields;
}
```

Do you understand what is happening? Yes, since there is now a single `@fields` array, all our references actually point to the same memory location, the content of which is last updated during the last loop iteration. To fix the issue, we need to be sure that we take and store references to distinct arrays.

```
# alternative version 2 (with unnecessary copying)
my @lines;
my @fields;
for my $codon (sort keys %count_for) {
    @fields = ($aa, $codon, $count, $total, sprintf "%5.1f", $usage);
    push @lines, [ @fields ];
}
```

The square bracket characters (`[` and `]`) create a new **anonymous array** and return its reference. Hence, what we `push` onto `@lines` is a reference to a fresh array containing the list of values resulting from the evaluation of the `@fields` array. This anonymous array is thus a copy of the `@fields` array. To avoid unnecessary copying, get back to the original code or use the idiomatic version below.

```
# alternative version 3 (idiomatic)
my @lines;
for my $codon (sort keys %count_for) {
    push @lines, [ $aa, $codon, $count, $total, sprintf "%5.1f", $usage ];
}
```

BOX 4: Reference count and the garbage collector

A reference is akin to a **pointer** in languages such as **C**, except that you cannot use it directly to manipulate the computer memory. It it thus more like an *identifier* than the *address* of a memory location. This is so because Perl has a mechanism known as the **garbage collector** that periodically reclaims unused memory locations. Had we direct access to the computer memory, Perl could not manage it for us.

Given that `@fields` is a lexical array, why is it not destroyed after the loop then?

Well, whenever one takes and stores a reference to a variable (here, the `@fields` array), Perl increments an internal counter known as the **reference count** of the variable. After the loop, the reference count of each `@fields` array is thus equal to one, while they all rise to two after the creation of `@sorted_lines`. In the figure above, this value is reflected by the two arrows pointing to each green array (the former `@fields`).

Before reclaiming the memory used by any variable, the garbage collector first checks that its reference count is equal to zero. If not, it does not free the corresponding memory location, even though the original variable identifier (here `@fields`) has been lost.

Conversely, whenever a reference is destroyed, Perl decrements the reference count of the referenced variable. In our case, it happens when both `@lines` and `@sorted_lines` arrays get out of scope. At that moment, the `push`ed array references are all destroyed at once and their reference counts get back to zero. Think of it like deleting all the arrows in the figure. The next time the garbage collector enters into action, the corresponding memory locations (green arrays) will be freed and used for storing new variables.

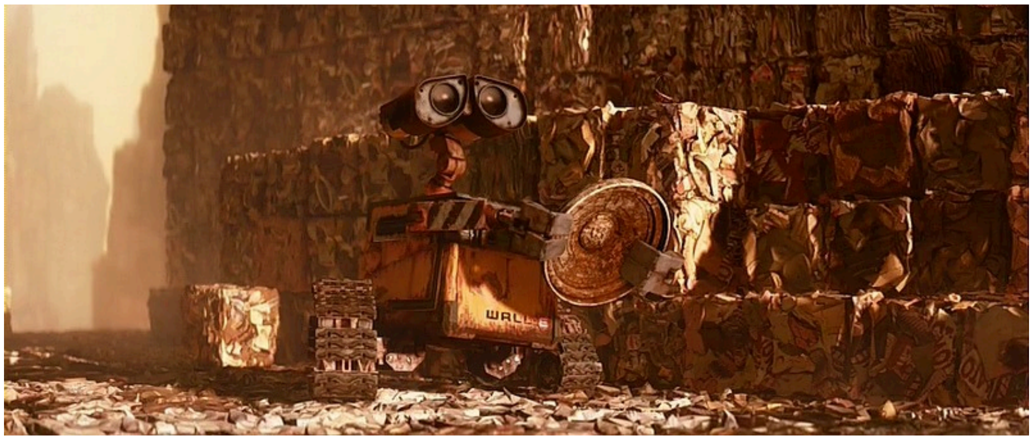

Figure 4.2: Garbage collection with WALL-E [Disney / Pixar, 2008]

4.1.5 Peeking into nested data structures

`Smart::Comments` makes looking at complex nested data structures very easy. If you want to try it, simply add the following line somewhere in your code.

```
### @lines

# gives:

### @lines: [
###           [
###             'K',
###             'AAA',
###             46021,
###             60126,
###             ' 76.5'
###           ],
###           [
###             'N',
###             'AAC',
###             29496,
###             53737,
###             ' 54.9'
###           ],
...
###           [
###             'F',
###             'TTT',
###             30444,
###             53065,
###             ' 57.4'
###           ]
###         ]
```

4.2 Sorting tables

Perl has very complete sorting facilities. It features several algorithms (**quicksort**, **mergesort**) and, given one of these algorithms, you can decide whether the sort should be in **lexical order** (default) or in **numeric order**, and whether it should be in **ascending order** or in **descending order**. Moreover, you can specify the *type* and *direction* of the sort for an arbitrary large number of pieces of data.

4.2.1 `sort`

The workhorse behind sorting is the builtin function `sort`. In its most basic use, it takes a list and sorts its values in ascending lexical order. For example, in the original `codon_usage.pl`, we sorted the usage table on the codon (used as key in the hash `%count_for`).

```
for my $codon (sort keys %count_for) {
    # loop body
}
```

If this default sort direction and type is not what you need, you may specify a block describing how a single comparison between two values of your list should be carried out. In this block, you will be using two special variables (`$a` and `$b`), one or more **sort comparison operators** (`cmp` or `<=>`) and zero or more logical or operators (`||`) taking advantage of **short-circuiting** for breaking ties.

4.2.2 Sort blocks and sort comparison operators

In `codon_usage_sort.pl`, a single sorting statement does all the work. As already explained, it extracts the elements of the array `@lines`, sorts them and returns a sorted list that is then stored in the new array `@sorted_lines`. Let's have a look at the **sort block**.

```
my @sorted_lines = sort {
    $a->[0] cmp $b->[0] || $b->[4] <=> $a->[4]
} @lines;
```

Such a block is actually an **anonymous function** that is called whenever the sorting algorithm has to compare two values of our list. Within this function, the values under comparison are *aliased* to the special variables `$a` and `$b` (in their current order in the list). The output of the sort block should be a number either less than or greater than zero or exactly equal to zero.

To generate this number given `$a` and `$b`, use the specially designed sort comparison operators `cmp` and `<=>`. These infix operators evaluate their operands in string context or numeric context, respectively, and return a number (`-1`, `0` or `1`) describing the order in which they are currently arranged.

Table 4.1: Return values of sort comparison operators

value	meaning
`-1`	*left operand is less than right operand*
`0`	*left operand is equal to right operand*
`1`	*left operand is greater than right operand*

Below is an example using numeric context.

```
### $a: 5
### $b: 10
### $a <=> $b: -1
### $b <=> $a: 1

### $a: 12
### $b: 6
### $a <=> $b: 1
### $b <=> $a: -1

### $a: 8
### $b: 8
### $a <=> $b: 0
### $b <=> $a: 0
```

When designing sort blocks, `$a` always represents the smallest of the two elements under comparison and `$b` the largest. Technically, the sorting algorithm will exchange the current positions of the values *aliased* to `$a` and `$b` whenever the sort block returns a positive number. Thus, if you want to sort your list in ascending order, put `$a` on the left and `$b` on the right. Conversely, put `$b` on the left and `$a` on the right if you want descending order. Here's an example using string context.

```
my @bases = qw(A T C G);
my  @asc_bases = sort { $a cmp $b } @bases;
my @desc_bases = sort { $b cmp $a } @bases;

### @asc_bases
### @desc_bases

# gives:

### @asc_bases: [
###                'A',
###                'C',
###                'G',
###                'T'
###              ]
### @desc_bases: [
###                 'T',
###                 'G',
###                 'C',
###                 'A'
###               ]
```

4.2.3 Sorting contexts

In many scientific applications, one needs to sort numbers whereas, as mentioned above, Perl `sort` defaults to ascending lexical order, i.e., string context. Forgetting to tell Perl to sort in numeric context can lead to very nasty bugs that are easily overlooked. Consider the following example.

```
my @numbers = (7, 12, 2, 4, 11, 1);
### default: sort @numbers
### lexical: sort { $a cmp $b } @numbers
### numeric: sort { $a <=> $b } @numbers

# gives:

### default: 1,
###          11,
###          12,
###          2,
###          4,
###          7
### lexical: 1,
###          11,
###          12,
###          2,
###          4,
###          7
### numeric: 1,
###          2,
###          4,
###          7,
###          11,
###          12
```

As you can see, only the third `sort` statement yields the expected order!

BOX 5: Sorting in natural order

If you need to sort *strings with a numeric component*, neither `cmp` nor `<=>` are suitable to the job.

```
my @ids = qw(seq7 seq12 seq2 seq4 seq11 seq1);
### lexical: sort { $a cmp $b } @ids
### numeric: sort { $a <=> $b } @ids

# gives:

### lexical: 'seq1',
###          'seq11',
###          'seq12',
###          'seq2',
###          'seq4',
###          'seq7'
```

```
Argument "seq7" isn't numeric in sort at sort_context.pl line 8.
Argument "seq12" isn't numeric in sort at sort_context.pl line 8.
Argument "seq2" isn't numeric in sort at sort_context.pl line 8.
Argument "seq4" isn't numeric in sort at sort_context.pl line 8.
Argument "seq11" isn't numeric in sort at sort_context.pl line 8.
Argument "seq1" isn't numeric in sort at sort_context.pl line 8.

### numeric: 'seq7',
###          'seq12',
###          'seq2',
###          'seq4',
###          'seq11',
###          'seq1'
```

For these special cases, there exists the convenient **natural order** sorting function `ncmp`, which is provided by the module `Sort::Naturally` (install it with `cpanm`). Strictly speaking, it is not a sort comparison operator, so look at the example below to see how to invoke it.

```
use Sort::Naturally;

my @ids = qw(seq7 seq12 seq2 seq4 seq11 seq1);
### natural: sort { ncmp($a, $b) } @ids

# gives:

### natural: 'seq1',
###          'seq2',
###          'seq4',
###          'seq7',
###          'seq11',
###          'seq12'
```

4.2.4 The dereferencing arrow

The actual sort block used in `codon_usage_sort.pl` is way more complex than the examples above with respect to two things:

- it sorts a nested data structure,
- it uses two sorting criteria for breaking ties.

When sorting lines, we compare only some specific fields of the two elemental arrays under comparison. This requires a special form of dereferencing using the **dereferencing arrow** (->), which allows us to access the individual elements of an array for which we only have a reference.

```
{
    $a->[0] cmp $b->[0]         # sort in  ascending lexical order on first field
 || $b->[4] <=> $a->[4]         # then in descending numeric order on fifth field
}
```

Here, it means that `$a` and `$b` are two array references, in which we want to compare the elements first at index zero (`aa`) and then at index four (`usage`). The first comparison should be done in *ascending string context* and the second in *descending numeric context*.

Table 4.2: Structure of the anonymous arrays

0	1	2	3	4
aa	codon	count	total	usage

Consider the example below.

```
### $a: [
###       'A',
###       'GCG',
###       '46056',
###       '129593',
###       '35.5'
###     ]
### $b: [
###       'A',
###       'GCC',
###       '34946',
###       '129593',
###       '27.0'
###     ]

### $a->[0]: 'A'
### $b->[0]: 'A'
### $a->[0] cmp $b->[0]: 0

### $a->[4]: '35.5'
### $b->[4]: '27.0'
### $b->[4] <=> $a->[4]: -1
```

4.2.5 Short-circuiting and multiple sorting criteria

When Perl encounters complex conditional expressions linked by logical operators (`||`, `&&`, `and` and `or`), it exhibits a short-circuiting behavior. This means that it stops evaluating an expression as soon as it can determine whether the whole expression would succeed or fail (i.e., evaluate to a true value or a false value, respectively).

```
### $a->[0] cmp $b->[0]: 0
### $b->[4] <=> $a->[4]: -1

### $a->[0] cmp $b->[0] || $b->[4] <=> $a->[4]: -1
```

In the example above, the two comparisons are linked by a logical or (`||`) and thus potentially subject to short-circuiting. Given the current values under comparison (`'A' cmp 'A'`), the first comparison yields `0`. Since `perl` cannot decide at this point whether the whole expression evaluates to true (`-1` or `1`) or false (`0`), it performs the second comparison (`27.0 <=> 35.5`), which yields `-1`. It therefore leaves the two values at their current positions, as expected.

When designing a sort block, add the sorting criteria one by one and link them by logical or's. The `perl` interpreter will evaluate the comparisons from left to right as long as it cannot decide between `-1` and `1`. There is thus a direct mapping between the position of a given criterion and its importance in the hierarchy for breaking ties between the values under comparison.

BOX 6: Short-circuiting and assignments

Generally speaking, short-circuiting can have an effect if the non-evaluated expressions involve assignments or in-place operators such as `+-` or `++`. In most cases, such constructs will lead to nasty bugs and thus have to be avoided.

```
my $x1 = 5;
my $x2 = 10;
my $x3 = 15;

if ($x1 < $x2 || $x2++ > $x3) {
    say q{No need to evaluate the cryptic second part!};
}
### $x1
### $x2
### $x3

# gives:

No need to evaluate the cryptic second part!

### $x1: 5
### $x2: 10
### $x3: 15
```

Within sort blocks, expressions should never try to modify `$a` and `$b`. However, thinking in short-circuit mode can help understanding what is going on, as shown above.

Chapter 5

In silico restriction mapping

5.1 Your very personal cutter

It is time for a new killer app. This one is serious stuff since it combines everything we have seen so far, including regular expressions, functions and references. It also features a new batch of novelties, among which a production-grade autodocumenting command-line interface allowing for **optional arguments**, **argument type checking** and **default values**. This marvel is accomplished through the use of a powerful new module known as `Getopt::Euclid`. You will need to install it to try the program.

```
$ cpanm Getopt::Euclid
```

Every molecular biologist has at least once made a construction involving the restriction of a plasmid and the subsequent ligation of some DNA fragment into the linearized vector. Now, most of them use nifty graphical (commercial) tools, such as **Vector NTI**. In the old times, however, we had to resort to cruder solutions. In this section, we will develop a program that does basic *in-silico restriction mapping*.

Our program is called `cutter.pl` and is quite sophisticated. Here are its main features:

- It can take a large database of restriction enzymes.
- It can then only consider a specified set of enzymes.
- It can process a FASTA file containing more than one DNA sequence.
- It can report the different cleavage sites.
- It can compute the expected restriction fragments and their lengths.
- It can plot them on a restriction map.

5.1.1 How to build an enzyme database?

Before testing our new tool, we need a database of *restriction enzymes*. To build it, borrow and reformat the list of cleavage site expressions available at our usual source of bioinformatics web applications.

1. Go to `http://www.bioinformatics.org/sms/rest_sum.html`.
2. Copy the content of the second input field, paste it into a new file and save it as `patterns.txt`.
3. Using the terminal, go to the directory containing the file and type in the following Perl one-liner. Be very careful when copying it! Remember that the trailing backslash \ (used as the line continuation character) is unnecessary if you type in the whole one-liner on a single line.

```
$ perl -nle 'next if m/MboII/; m{/([^\/]+)/\s+\((\w+)\s+[^\)]+\)(\d+),?}; \
    print join "\t", $2, $3, $1' patterns.txt > enzyme-db.txt
```

4. Check that your enzyme database (`enzyme-db.txt`) looks fine. Here's an excerpt from mine for comparison. If your file is different, double-check your one-liner and give it a second shot. Note that we exclude *Mbo*II because its non-symmetrical cut is not supported by our program.

```
AatII   1   gacgtc
AccIII  5   tccgga
AluI    2   agct
ApaI    1   gggccc
AvaI    5   c[ct]cg[ag]g
...
VspI    4   attaat
XbaI    5   tctaga
XhoI    5   ctcgag
XmaI    5   cccggg
```

5.1.2 How to fetch sequences from the NCBI website?

Once we have an enzyme database, we need a few sequences to cut. A logical choice would be plasmid vectors with a multiple cloning site, such as the *pBluescript II* family of phagemids that allow for blue/white screening of recombinant colonies. The sequences of four variants can be found in the nucleotide section of *GenBank* under the *accessions* `X52327` to `X52330`.

To download them as a single FASTA file, use the **NCBI E-utilities**. These tools are a vast topic that is well worth exploring for yourself. You can find more about them here:
http://www.ncbi.nlm.nih.gov/books/NBK25500/

For our purposes, I simply give you the magical command that does the job on Linux. If you instead use macOS, just replace the call to `wget -O` by `curl -o` (note the upper- *vs.* lowercase `O`).

```
$ wget -O phagemids.fasta \
    "https://eutils.ncbi.nlm.nih.gov/entrez/eutils/efetch.fcgi?\
    db=nuccore&retmode=text&rettype=fasta&id=X52327,X52328,X52329,X52330"
```

5.1.3 How to try the program?

Since the command-line interface of `cutter.pl` is quite flexible, there are many ways of trying it. You will find below a few suggestions, but feel free to play with it.

```
$ perldoc cutter.pl
$ ./cutter.pl
$ ./cutter.pl --man

$ ./cutter.pl phagemids.fasta --enzyme-db=enzyme-db.txt \
    --enzymes=MspI

$ ./cutter.pl phagemids.fasta --enzyme-db=enzyme-db.txt \
    --enzymes=EcoRI KpnI XhoI
```

```
$ ./cutter.pl phagemids.fasta --enzyme-db=enzyme-db.txt \
    --enzymes=MspI --plot-map

$ ./cutter.pl phagemids.fasta --enzyme-db=enzyme-db.txt \
    --enzymes=MspI --plot-map --tic-width=10

$ ./cutter.pl phagemids.fasta --enzyme-db=enzyme-db.txt \
    --enzymes=MspI --plot-map --tic-width=10 --min-frag-len=100
```

5.2 The code for `cutter.pl`

```
#!/usr/bin/env perl

use Modern::Perl '2011';
use autodie;

use Getopt::Euclid;
use Smart::Comments '###';

use List::AllUtils qw(min max);
use POSIX;
use Term::Size::Any 'chars';
use Tie::IxHash;


#### %ARGV

warn 'WARNING! --circular option not yet implemented!'
    if $ARGV{'--circular'};

tie my %seq_for, 'Tie::IxHash';
%seq_for = read_fasta( $ARGV{'<infile>'} );
#### %seq_for

my %pattern_for = read_enzymes( $ARGV{'--enzyme-db'} );
#### %pattern_for

SEQ:
while (my ($id, $dna_string) = each %seq_for) {

    say "\n# $id";

    my @cuts = compute_cuts($dna_string);
    next SEQ unless @cuts;

    #### @cuts

    my %sites_for = collect_sites(@cuts);
    #### %sites_for

    my @fragments = infer_fragments($dna_string, @cuts);
    #### @fragments

    say "\n# \UList of cleavage sites";
    say '# ' . join "\t", qw(enz sites);
    for my $enzyme (sort keys %sites_for) {
        say join "\t", $enzyme, join ', ', @{ $sites_for{$enzyme} };
    }

    say "\n# \UList of fragments\E (>= $ARGV{'--min-frag-len'} bp)";
    say '# ' . join "\t", qw(len 5'-pos 3'-pos 5'-enz 3'-enz);

    FRAGMENT:
    for my $fragment (@fragments) {
        last FRAGMENT if $fragment->[0] < $ARGV{'--min-frag-len'};
        say join "\t", @$fragment;
    }

    plot_map($dna_string, @fragments) if $ARGV{'--plot-map'};
}


sub read_fasta {            # recycled "as is" from xxl_xlate.pl
    my $infile = shift;

    open my $in, '<', $infile;

    my $seq_id;
    my $seq;
    tie my %seq_for, 'Tie::IxHash';         # preserve original seq order

    LINE:
    while (my $line = <$in>) {
        chomp $line;

        # at each '>' char...
        if (substr($line, 0, 1) eq '>') {

            # add current seq to hash (if any)
            if ($seq) {
                $seq_for{$seq_id} = $seq;
                $seq = q{};
            }

            # extract new seq_id
            $seq_id = substr($line, 1);
            next LINE;
        }

        # elongate current seq (seqs can be broken on several lines)
        $seq .= $line;
    }

    # add last seq to hash (if any)
    $seq_for{$seq_id} = $seq if $seq;

    return %seq_for;
}


sub read_enzymes {
    my $infile = shift;

    open my $in, '<', $infile;

    my %pattern_for;

    LINE:
    while (my $line = <$in>) {
        chomp $line;

        next LINE if $line =~ m/^ \s* $/xms;        # skip empty lines
        next LINE if $line =~ m/^ \#/xms;           # skip comment lines

        my ($enzyme, $dist3p, $pattern) = split /\t/xms, $line;
        $pattern_for{ lc $enzyme } = {              # case-insensitive name
            enzyme => $enzyme,
            dist3p => $dist3p,
            regexp => qr{$pattern}xmsi,             # case-insensitive pattern
        };
    }

    return %pattern_for;
}


sub compute_cuts {
    my $dna_string = shift;

    my @cuts;

    ENZYME:
    for my $name ( @{ $ARGV{'--enzymes'} } ) {      # only specified enzymes

        my $key = lc $name;
        my $enzyme = $pattern_for{$key}{enzyme};
        my $dist3p = $pattern_for{$key}{dist3p};
        my $regexp = $pattern_for{$key}{regexp};

        unless ($enzyme) {
            warn "WARNING! Unknown enzyme: $name; skipping it.";
            next ENZYME;
        }

        while ($dna_string =~ m/$regexp/g) {
            my $site = pos($dna_string) - $dist3p;  # $site is 1-based!
            push @cuts, {
                enzyme => $enzyme,
                site   => $site,
            };
        }
    }

    @cuts = sort { $a->{site} <=> $b->{site} } @cuts;

    return @cuts;
}


sub collect_sites {
    my @cuts = @_;

    my %sites_for;
    for my $cut (@cuts) {
        my ($site, $enzyme) = @{ $cut }{ qw(site enzyme) };
        push @{ $sites_for{ $enzyme } }, $site;
    }

    return %sites_for;
}


sub infer_fragments {
    my ($dna_string, @cuts) = @_;

    unshift @cuts, { enzyme => q{5'-end}, site => 1                  };
    push    @cuts, { enzyme => q{3'-end}, site => length $dna_string };

    my @fragments;
    my ($enz1, $x1) = @{ shift @cuts }{ qw(enzyme site) };
    while (@cuts) {

        my ($enz2, $x2) = @{ shift @cuts }{ qw(enzyme site) };
        my $len = $x2 - $x1 + 1;
        warn 'WARNING! Some cleavage sites overlap; inaccurate results!'
            if $len < 1;

        push @fragments, [ $len, $x1, $x2, $enz1, $enz2 ];
        ($x1, $enz1) = ($x2+1, $enz2);
    }

    @fragments = sort { $b->[0] <=> $a->[0] } @fragments;

    return @fragments;
}


BEGIN{      # this block will be executed just after compilation
            # its lexical variables will be available for the included subs

    #### Processing BEGIN block...

    # setup plotting area
    my $tic_width = $ARGV{'--tic-width'};
    my ($cols, $rows) = chars();
    my $tic_n = floor( $cols / $tic_width );


    sub plot_map {
        my ($dna_string, @fragments) = @_;

        # setup scale
        my $min = 1;
        my $max = length $dna_string;
        my $range = $max - $min;
        my $step = ceil( $range / ($tic_n-1) );

        # plot scale
        say "\n# \URestriction map";
        say step_line(0, $min, $step);
        say step_line(1, $min, $step);
        say q{-} x $cols;

        FRAGMENT:
        for my $fragment (@fragments) {
            my ($len, $x1, $x2, $enz1, $enz2) = @$fragment;

            last FRAGMENT if $len < $ARGV{'--min-frag-len'};

            my $pad_n = xloc($x1, $min, $step);
            my $chr_n = xloc($x2, $min, $step) - $pad_n + 1;

            say pair_line($pad_n, $chr_n, $x1, $x2);        # positions
            say q{ } x $pad_n . q{#} x $chr_n;              # fragment
            say pair_line($pad_n, $chr_n, $enz1, $enz2);    # enzymes
        }

        return;
    }


    sub pair_line {
        my ($pad_n, $chr_n, $id1, $id2) = @_;

        my $spc_n = $chr_n - length($id1) - length($id2);
           $spc_n = 0 if $spc_n < 0;        # avoid warning when $spc_n < 0
        return q{ } x $pad_n . $id1 . q{ } x $spc_n . $id2;
    }


    sub step_line {         # recycled "as is" from check_overlap.pl
        my ($bars, $min, $step) = @_;

        my $str;
        for (my ($x, $tic) = ($min, 0); $tic < $tic_n; $x += $step, $tic++) {
            $str .= sprintf("%-*d", $tic_width, $x) unless $bars;
            $str .= '|' . q{ } x ($tic_width-1)        if $bars;
        }
        return $str;
    }


    sub xloc {              # recycled "as is" from check_overlap.pl
        my ($x, $min, $step) = @_;
        return sprintf "%.0f", ($x-$min) / $step * $tic_width;
    }

}


=head1 NAME

```

```
cutter - Compute a restriction map for one or more DNA sequences

=head1 VERSION

This documentation refers to cutter version 0.0.1

=head1 USAGE

    cutter.pl <infile> --enzymes <name>... --enzyme-db <infile> [options]

=head1 REQUIRED ARGUMENTS

=over

=item <infile>

Path to input FASTA file.

=for Euclid:
    infile.type: readable

=item --enzyme-db [=] <infile>

Path to restriction enzyme database.

=for Euclid:
    infile.type: readable

The database is a tab-delimited 3-column flat file structured as follows:

    1. enzyme name
    2. distance of the cleavage site from the 3'-end of the pattern
    3. recognition pattern (regular expression)

Empty lines and comment lines beginning with '#' are allowed.

Example:

    # EcoRI generates sticky ends: G|AATTC
    EcoRI   5   gaattc
    HinfI   4   ga[gatc]tc

=item --enzymes [=] <name>...

List of whitespace-separated enzyme names to consider in the analysis.
For convenience, enzyme names are case insensitive.

```

```
=back

=head1 OPTIONS

=over

=item --circular

Treat DNA sequences as circular molecules [default: no].

=item --min-frag-len [=] <length>

Minimum length of fragments to be reported (in bp) [default: length.default].

=for Euclid:
    length.type: +integer
    length.default: 20

=item --plot-map

Plot a graphical restriction map of each sequence [default: no].

=item --tic-width [=] <length>

Distance between two tics on the fragment map (in char)
[default: length.default].

=for Euclid:
    length.type: +integer
    length.default: 8

=item --version

=item --usage

=item --help

=item --man

Print the usual program information

=back

=head1 AUTHOR

Your Name (your.email@host.com)

```

```
=head1 BUGS

There are undoubtedly serious bugs lurking somewhere in this code.
Bug reports and other feedback are most welcome.

=head1 COPYRIGHT

Copyright (c) 2013, Your Name. All Rights Reserved.
This program is free software. It may be used, redistributed
and/or modified under the terms of the Perl Artistic License
(see http://www.perl.com/perl/misc/Artistic.html)
```

Homework

Even though `check_overlap.pl` computes the best possible scale for the plot, it falls short when it comes to choosing a scale that would always appear natural to a human. Hence, some combinations of fragment coordinates, terminal width and tic width yield scales that are very unwieldy, such as 104, 6910, 13720, 20530…

As your next assignment (`hw7_check_overlap_scale.pl`), try to tweak the computation of the scale, so as to obtain tic values that look *rounded* independently of the order(s) of magnitude of the scale (e.g., 50, 125, 500, 2000). Tip: One way to achieve that would be to use *log*-based rounding.

Part III

Lesson 8

Chapter 6

The innards of `cutter.pl`

6.1 Plain Old Documentation

Many Perl developers have an excellent programming culture. This means that the Perl community produces software of high-quality. In the remaining lessons, I will try to demonstrate this thesis by presenting you some robust and powerful Perl modules that can really help you on a daily basis.

High-quality software implies at least two things: comprehensive testing and good documentation. This lesson explores both aspects.

Perl's documentation system is called **POD** (for *Plain Old Documentation*). It is a simple **markup language** used for writing documentation for `perl` itself, Perl programs and Perl modules. Translators are available for converting POD to various formats like plain text, HTML, man pages, and more.

POD blocks can be *interspersed* in Perl code. This makes documenting programs very handy because you can include user-oriented explanations physically close to the corresponding code. Hence, a function can be documented by a POD block directly preceding or following its definition in Perl.

POD markup consists of three basic kinds of paragraphs:

- **Ordinary paragraphs** are simple blocks of text that must be preceded and followed by a blank line. They will be rewrapped to screen width by the formatters. Different **formatting codes** are allowed in these paragraphs.
- **Verbatim paragraphs** are used to display blocks of code or any other text that should not be rewrapped. Such paragraphs must be indented (with spaces or tab characters) to be recognized by **POD parsers**. No formatting codes are allowed within verbatim paragraphs. Everything appears *as is*.
- **Command paragraphs** indicate that the following chunk of text has to get a special treatment. For example, it has to be formatted as a title or as a list. These paragraphs typically consist in a single line beginning with an **equal character** (=) followed by an identifier then arbitrary text.

The next page features a table summarizing the most common POD commands.

Table 6.1: Main POD commands

command	meaning
`=head1`	level-1 header
`=head2`	level-2 header
`=head3`	level-3 header
`=head4`	level-4 header
`=over`	list opening or beginning of one or more indented paragraphs
`=back`	list closing or end of one or more indented paragraphs
`=item`	list item (maybe be made of multiple paragraphs)
`=pod`	explicit opening of a POD block (often unnecessary)
`=cut`	end of a POD block (needed for following it by regular code)
`=for`	command for a specific POD parser

Below is an excerpt from my module `Bio::MUST::Core::Ali`, in which the POD block is just above the definition of the corresponding function (here, the function is called a **method** because it is an object-oriented module). The POD block uses all three kinds of paragraphs. Regarding the method `load` itself, you should recognize our good old FASTA file reader.

```
=head1 I/O METHODS

=head2 load

Class method (constructor) returning a new Ali read from disk. This method
will transparently import plain FASTA files in addition to the MUST
pseudo-FASTA format (ALI files).

    use Test::Deeply;
    use aliased 'Bio::MUST::Core::Ali';
    my $ali1 = Ali->load('example.ali');
    my $ali2 = Ali->load('example.fasta');
    my @seqs1 = $ali1->all_seqs;
    my @seqs2 = $ali2->all_seqs;
    is_deeply, \@seqs1, \@seqs2, 'should be true';

This method requires one argument.

=cut

sub load {
    my $class  = shift;
    my $infile = shift;

    open my $in, '<', $infile;

    my $ali = $class->new();
    my $seq_id;
    my $seq;

    LINE:
    while (my $line = <$in>) {
        chomp $line;

        # skip empty lines and process comments
        next LINE if $line =~ $EMPTY_LINE
                  || $ali->is_comment($line);

        # at each '>' char...
        my ($defline) = $line =~ $DEF_LINE;
        if ($defline) {

            # add current seq to ali (if any)
            if ($seq) {
                my $new_seq = Seq->new( seq_id => $seq_id, seq => $seq );
                $ali->add_seq($new_seq);
                $seq = q{};
            }

            $seq_id = $defline;
            next LINE;
        }

        # elongate current seq (seqs can be broken on several lines)
        $seq .= $line;
    }

    # add last seq to ali (if any)
    if ($seq) {
        my $new_seq = Seq->new( seq_id => $seq_id, seq => $seq );
        $ali->add_seq($new_seq);
    }

    return $ali;
}
```

There exist a dozen formatting codes. The following table lists the common ones.

Table 6.2: Main POD formatting codes

code	meaning
`I<text>`	typeset `text` in italics
`B<text>`	typeset `text` in boldface
`C<text>`	typeset `text` using a typewriter font
`L<text>`	define a hyperlink for `text`

Here's another POD example that demonstrates the use of some formatting codes. Observe how the documentation is designed. It first explains the purpose of the method then gives some details, followed by illustrative use cases. Finally, it lists the parameters expected by the method (here termed arguments) and specify their meaning. If the method returns something, it should be mentioned in its purpose, as with the `load` method above.

```
=head2 store_fasta

Writes the Ali to disk in the plain FASTA format.

For compatibility purposes, this method automatically fetches sequence ids
using the C<foreign_id> method instead of the native C<full_id> method, both
described in L<Bio::MUST::Core::SeqId>.

    $ali->store_fasta('output.fasta');
    $ali->store_fasta('output.fasta', {chunk => -1, degap => 1});

This method requires one argument and accepts a second optional argument
controlling the output format. It is a hash reference that may contain one
or more of the following keys:

    - clean: replace '?' by 'X' for compatibility
    - chunk: line width (default is 60 chars; negative values means no wrap)
    - degap: boolean value controlling degapping (default: false)
```

We will stop our description here. To read more about POD, see:
http://perldoc.perl.org/perlpod.html

… or use your terminal. A sizable part of this section comes from there.

```
$ perldoc perlpod
```

6.2 Getopt::Euclid

`cutter.pl` uses only one new module, but this module changes a lot! It belongs to the `Getopt` **namespace**, which hints at the fact that it is designed to parse command-line arguments.

`Getopt::Euclid` is pretty unique in the way that it analyzes your program's POD blocks to create a powerful command-line argument parser. Beyond a terrific economy of effort, this approach has two additional benefits:

- it forces you to document your program in the Perl's standard way;
- it ensures that your documentation always matches your actual user interface.

The created parser includes many features, such as argument type checking, required arguments, exclusive arguments, optional arguments with default values, automatic usage message, etc.

To take advantage of this module, it is enough to `use` it at the beginning of your program.

```
use Getopt::Euclid;
```

This line causes the following things to happen upon program launch:

1. The POD blocks are analyzed and the user interface of your program is deduced from its description in the `=head1 REQUIRED ARGUMENTS` and `=head1 OPTIONS` sections.
2. A customized parser is built that parses the command-line arguments and options specified by your POD. If a required argument is missing or if any argument does not match its expected type, it automatically stops the program with an informative error message.
3. If all arguments pass the validation step, the parser removes them from the default array `@ARGV` and puts them in the global hash `%ARGV`.

Argument type checking is controlled by command paragraphs of the form `=for Euclid:`. For example, the POD block below documents the optional argument `--min-frag-len` and specifies that it should be a strictly positive integer number with a default value of `20`.

```
=head1 OPTIONS

...

=item --min-frag-len [=] <length>

Minimum length of fragments to be reported (in bp) [default: length.default].

=for Euclid:
    length.type: +integer
    length.default: 20

...
```

`Getopt::Euclid` is a powerful module that you should investigate seriously if you plan to develop production-grade Perl software. For now, simply use the POD block in `cutter.pl` as a *template*.

```
$ perldoc Getopt::Euclid
```

6.3 Overview of `cutter.pl`

`cutter.pl` is our most complex program so far. Before diving into its technical novelties, it might be useful to describe its general organization and functioning.

6.3.1 Command-line interface

First, let's have a look at the `%ARGV` hash because it will drive the behavior of the whole program.

```
$ ./cutter.pl data/phagemids.fasta --enzyme-db=data/enzyme-db.txt \
    --enzymes=EcoRI KpnI XhoI --plot-map
```

Assuming that `Smart::Comments`' *fourth level of smartness* is enabled, launching `cutter.pl` with this command will yield the following hash.

```
### %ARGV: {
###          '--enzyme-db' => 'data/enzyme-db.txt',
###          '--enzymes' => [
###                           'EcoRI',
###                           'KpnI',
###                           'XhoI'
###                         ],
###           '--min-frag-len' => 20,
###           '--plot-map' => '1',
###           '--tic-width' => 8,
###           '<infile>' => 'data/phagemids.fasta'
###         }
```

BOX 7: Named arguments

The idea of the %ARGV hash with its keys corresponding to argument names can be recycled to autodocument function arguments and simplify function calls. This only requires pretending that the function takes a hash as a single argument.

```
sub check_overlap {
    my %args = @_;

    # test disjunction
    return not ($args{y1} > $args{x2} || $args{y2} < $args{x1});
}

say 'Fragments DO overlap!'
    if check_overlap( x1 => 10, x2 => 20, y1 => 15, y2 => 25 );
```

Beyond autodocumentation, this approach has two more advantages: (1) arguments can be provided in any order and (2) some arguments can be missing (and assigned default values).

```
sub format_seq {
    my %args = @_;

    my $seq    = $args{seq};
    return unless $seq;                         # nothing to do if empty

    my $seq_id = $args{seq_id}     // 'seq';    # default id
    my $width  = $args{line_width} // 60;       # default line width

    # ...
}

# all these calls are equivalent
format_seq( seq => 'GAATTC...', seq_id => 'seq', width => 60 );
format_seq( seq_id => 'seq', seq => 'GAATTC...' );
format_seq( seq => 'GAATTC...' );
```

As a rule of thumb, you should consider using named arguments as soon as a function requires more than two or three parameters.

There is a perfect match between the keys of `%ARGV` and the names of the arguments in the documentation. This hash is a nested data structure since the value corresponding to the key `--enzymes` is a reference to an anonymous array of three strings. Moreover, the parser has inserted key/value pairs for the optional arguments with default values (here `--min-frag-len` and `--tic-width`).

6.3.2 Architecture

`cutter.pl` is made of a main block of code that is defined at the file scope and of a series of functions. In Figure 6.1 below, the three columns correspond to the three levels of indentation of the main block (shown here in abridged form). Functions are represented by boxes with a green background.

```
tie my %seq_for, 'Tie::IxHash';                            # level 1
%seq_for = read_fasta( $ARGV{'<infile>'} );                # level 1
my %pattern_for = read_enzymes( $ARGV{'--enzyme-db'} );    # level 1

# ...

SEQ:
while (my ($id, $dna_string) = each %seq_for) {            # level 1

    # ...

    for my $enzyme (sort keys %sites_for) {                # level 2
        # inner loop body                                  # level 3
    }

    # ...

    FRAGMENT:
    for my $fragment (@fragments) {                        # level 2
        # inner loop body                                  # level 3
    }

    # ...
}
```

Most functions here are defined for *abstraction*. Those pertaining to the plotting of the restriction map are also there for *re-use* and are thus called from more than one point in the program.

```
say "\n# \URestriction map";
say step_line(0, $min, $step);                     # call to step_line
say step_line(1, $min, $step);                     # call to step_line
say q{-} x $cols;

# ...

my $pad_n = xloc($x1, $min, $step);                # call to xloc
my $chr_n = xloc($x2, $min, $step) - $pad_n + 1;   # call to xloc
```

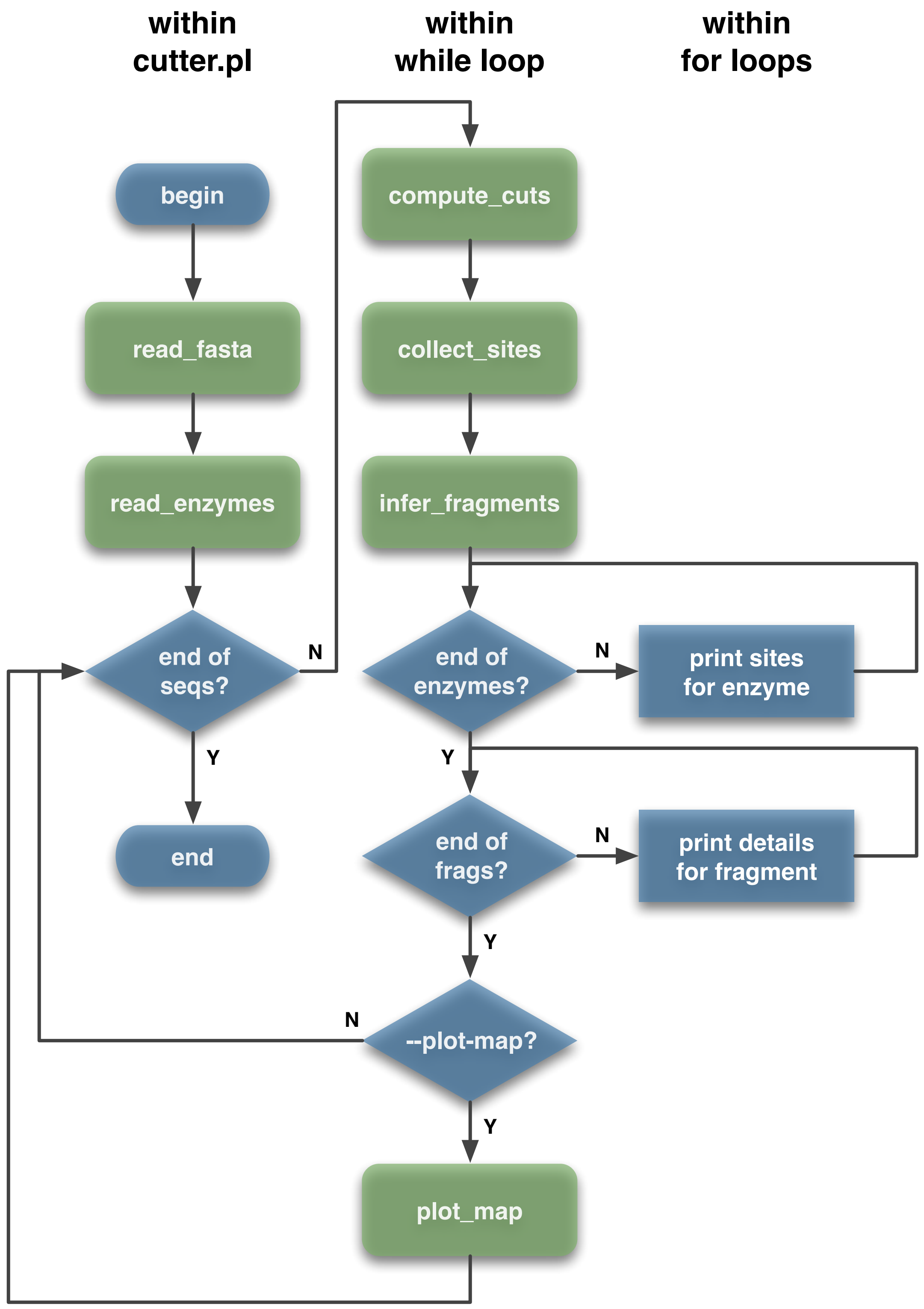

Figure 6.1: Flowchart of `cutter.pl`

```
say pair_line($pad_n, $chr_n, $x1, $x2);            # call to pair_line
say q{ } x $pad_n . q{#} x $chr_n;
say pair_line($pad_n, $chr_n, $enz1, $enz2);        # call to pair_line
```

However, some functions are not completely *encapsulated*, which means that they use global variables in addition to function parameters. This is acceptable in the context of a standalone program like `cutter.pl`. If we were to develop a Perl module, this would be an issue because module interfaces based on global variables are not fashionable anymore.

For example, the function `compute_cuts` receives the DNA sequence to cut as a string argument. This is required because it works on a single sequence at a time, whereas our sequences are stored in a container, the ordered hash `%seq_for`. The loop that iterates on the key/value pairs of this hash repeatedly calls this function for each sequence in the hash.

In contrast, the hash `%pattern_for`, which contains the details for each restriction enzyme in our database, is directly used as a global variable. And so is `$ARGV{'--enzymes'}`, which is a reference to the anonymous array storing the specific enzymes to consider.

```
sub compute_cuts {
    my $dna_string = shift;                         # function parameter

    my @cuts;

    ENZYME:
    for my $name ( @{ $ARGV{'--enzymes'} } ) {      # global variable

        my $key = lc $name;
        my $enzyme = $pattern_for{$key}{enzyme};    # global variable
        my $dist3p = $pattern_for{$key}{dist3p};    # global variable
        my $regexp = $pattern_for{$key}{regexp};    # global variable

        # ...
    }

    # ...
}
```

BOX 8: Closures and `BEGIN` code blocks

The function `plot_map` has three utility functions of a lower hierarchical level. Several of these functions have to share some variables that can be safely ignored by the other functions in `cutter.pl`, for example `$tic_width` and `$tic_n`.

To satisfy these requirements, we could do two things:

- pass them along as additional function parameters, which would increase the complexity of our function calls;
- define them in a lexical scope that is visible by all functions that need them, but not by the others. A function using lexical variables declared in an outer lexical scope is a **closure**.

Here, we choose the second solution and put the four utility functions in a *private* block including the definition of the two shared variables `$tic_width` and `$tic_n`. However, since this block physically comes below the main `while` loop, from which the `plot_map` function is called, `$tic_width` and `$tic_n` will not be defined in time. This is because in Perl code execution flows from the first to the last statement (see "Control flow in Perl", in the first part of this course).

Thus, to be sure that `$tic_width` and `$tic_n` are properly initialized at the moment `plot_map` tries to use them, we setup the block as a `BEGIN` code block. Such a block is executed as soon as it is completely defined, even before the rest of the containing file is parsed and compiled. Multiple `BEGIN` blocks are allowed; they will execute in order of definition. Because a `BEGIN` code block executes immediately, it can pull in definitions of functions and variables from other files in time to be visible to the rest of the compile and run time.

Amazingly, Perl is so dynamic that we could even decide to setup these shared variables and subs only when really required. All we need is to put the corresponding declarations and definitions in a conditional block as demonstrated below.

```
BEGIN{      # this block will be executed just after compilation
            # its lexical variables will be available for the included subs

    #### Processing BEGIN block...

    if ($ARGV{'--plot-map'}) {

        #### Declaring and defining plotting variables and subs...

        # setup plotting area
        my $tic_width = $ARGV{'--tic-width'};
        my ($cols, $rows) = chars();
        my $tic_n = floor( $cols / $tic_width );

        sub plot_map { ... }

        sub pair_line { ... }

        sub step_line { ... }

        sub xloc { ... }
    }
}
```

To finish with the architecture of `cutter.pl`, Figure 6.2 below shows the main lexical scopes existing in our program and summarizes the information flow between our functions.

6.4 More on references

`cutter.pl` rests on several nested data structures, the elements of which need dereferencing. Let's examine how these are built and the syntax for using them.

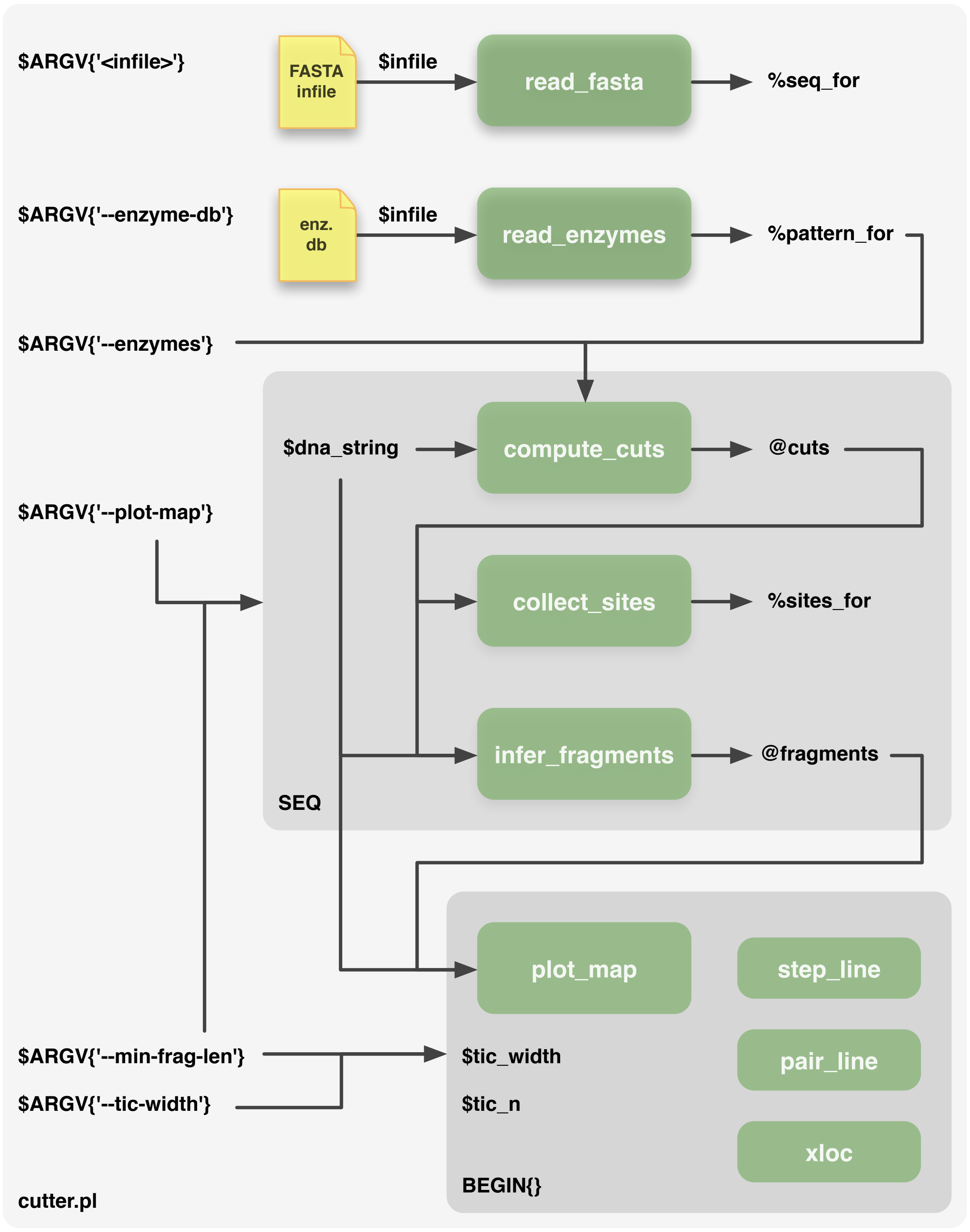

Figure 6.2: Lexical scopes and argument passing in `cutter.pl`

6.4.1 Tabular file parsing

Reading and storing a multi-column text file in a data structure, possibly transforming its content in the process, is a common task in bioinformatics, yet easily tackled with Perl. The function definition below is thus very useful because it can serve as a template for futures ones you would have to write.

6.4.1.1 read_enzymes

Our restriction enzyme database is stored in a tab-separated (**TSV**) tabular file. Here's the reading function for processing this file. Note that the function automatically skips empty and comment lines. You should always handle these to allow maximum flexibility when writing input files.

```
sub read_enzymes {
    my $infile = shift;

    open my $in, '<', $infile;

    my %pattern_for;

    LINE:
    while (my $line = <$in>) {
        chomp $line;

        next LINE if $line =~ m/\A \s* \z/xms;      # skip empty lines
        next LINE if $line =~ m/\A \#/xms;          # skip comment lines

        my ($enzyme, $dist3p, $pattern) = split /\t/xms, $line;
        $pattern_for{ lc $enzyme } = {              # case-insensitive name
            enzyme => $enzyme,
            dist3p => $dist3p,
            regexp => qr{$pattern}xmsi,             # case-insensitive pattern
        };
    }

    return %pattern_for;
}
```

The execution scheme can be described as follows:

1. The input file passed as an argument is opened.
2. An empty main hash is declared.
3. The input file is read line by line until its end…
   - to split each line on the separator (here, the tab character),
   - to store the resulting list of values in an anonymous hash in which the various fields are each referred to by a specific key,
   - to store in the main hash a reference to this smaller (elemental) hash as the value for a key derived from one of the columns.
4. The main hash, which is now a *hash of hashes*, is returned to the caller.

```
AatII   1   gacgtc
AccIII  5   tccgga
...
XhoI    5   ctcgag
XmaI    5   cccggg
```

Given the input file shown above, the hash returned by this function would look like this:

```
### %pattern_for: {
###                 aatii => {
###                            dist3p => '1',
###                            enzyme => 'AatII',
###                            regexp => qr/(?umsix:gacgtc)/
###                          },
###                 acciii => {
###                             dist3p => '5',
###                             enzyme => 'AccIII',
###                             regexp => qr/(?umsix:tccgga)/
###                           },
...

###                 xhoi => {
###                           dist3p => '5',
###                           enzyme => 'XhoI',
###                           regexp => qr/(?umsix:ctcgag)/
###                         },
###                 xmai => {
###                           dist3p => '5',
###                           enzyme => 'XmaI',
###                           regexp => qr/(?umsix:cccggg)/
###                         }
###               }
```

Since the return value evaluates to a list (remember that a hash is a container for a list of key/value pairs), we store it in another hash (of the same name, for clarity).

```
my %pattern_for = read_enzymes( $ARGV{'--enzyme-db'} );
```

The only other syntactic novelty in the function `read_enzymes` is the in-place definition of an **anonymous hash**, to which a reference is directly taken. This is achieved with the curly brace characters.

```
$pattern_for{ lc $enzyme } = {
    enzyme => $enzyme,
    dist3p => $dist3p,
    regexp => qr{$pattern}xmsi,
};
```

This code is a shorthand for the following longer equivalent chunk. Beside its brevity, it also avoids us to make up a temporary name for the elemental hash.

```
my %data_for = (
    enzyme => $enzyme,
    dist3p => $dist3p,
    regexp => qr{$pattern}xmsi,
);
$pattern_for{ lc $enzyme } = \%data_for;
```

6.4.2 Defining and using nested data structures

Within the main `while` loop, three functions each produce a different nested data structure. They will help us to illustrate the multiple syntaxes available for dereferencing such structures. The table below summarizes them, including the function reading the enzyme database for comparison.

Table 6.3: Examples of nested data structures

function name	structure name	structure type
`compute_cuts`	`@cuts`	*array of hashes*
`collect_sites`	`%sites_for`	*hash of arrays*
`infer_fragments`	`@fragments`	*array of arrays*
`read_enzymes`	`%pattern_for`	*hash of hashes*

6.4.2.1 `compute_cuts`

This is the function that cuts a given DNA sequence with the specified restriction enzymes. Basically, it tries each enzyme in turn and adds the identified cuts to an array in which each element is a smaller hash giving the name of the cutting enzyme and the location of the cleavage site.

```
sub compute_cuts {
    my $dna_string = shift;

    my @cuts;

    ENZYME:
    for my $name ( @{ $ARGV{'--enzymes'} } ) {       # only specified enzymes

        my $key = lc $name;
        my $enzyme = $pattern_for{$key}{enzyme};
        my $dist3p = $pattern_for{$key}{dist3p};
        my $regexp = $pattern_for{$key}{regexp};

        unless ($enzyme) {
            warn "WARNING! Unknown enzyme: $name; skipping it.";
            next ENZYME;
        }

        while ($dna_string =~ m/$regexp/g) {
            my $site = pos($dna_string) - $dist3p;  # $site is 1-based!
            push @cuts, {
                enzyme => $enzyme,
                site   => $site,
            };
        }
    }

    @cuts = sort { $a->{site} <=> $b->{site} } @cuts;

    return @cuts;
}
```

To loop over the list of specified enzymes, the function uses the following construct, which allows it to dereference the array nested in %ARGV. Note the simultaneous use of two different sigils, one ($) for the reference itself and the other (@) for enabling list context on the array referenced by the reference.

```
for my $name ( @{ $ARGV{'--enzymes'} } ) {
    # loop body
}
```

Had we used the hash value directly, we would have got the (useless) stringified version of the array reference, as explained in "Using references", p.46. Overlooking the second sigil is a common source of bugs when working with nested data structures. Pay attention to that!

```
say    "$ARGV{'--enzymes'}";
say "@{ $ARGV{'--enzymes'} }";

# gives:
```

```
ARRAY(0x7f98229c6b40)
EcoRI KpnI XhoI
```

Within the enzyme for loop, each piece of data for the enzyme under consideration is copied from the hash %pattern_for using a simple syntax, which consists in listing the keys for traversing the different levels of nesting in their hierarchical order. We do not need any dereferencing arrow (->) because we start with a plain hash and not with a hash reference. Moreover, each arrow between a closing square bracket (or curly brace) and an opening square bracket (or curly brace) is optional.

```
my $enzyme = $pattern_for{$key}{enzyme};
my $dist3p = $pattern_for{$key}{dist3p};
my $regexp = $pattern_for{$key}{regexp};
```

Before proceeding, we check that the specified enzyme is indeed found in our database file. If not, $enzyme is undef and the unless block gets executed. In this block, we warn the user of the issue (using a milder form of the die builtin function that does not halt our program but still write to the standard error stream). Then we leave the current loop iteration to process the next enzyme. Such checks may appear tedious to write but are part of what makes good-quality software.

```
unless ($enzyme) {
    warn "WARNING! Unknown enzyme: $name; skipping it.";
    next ENZYME;
}
```

There is a caveat here, due to the fact that `%pattern_for` is a nested data structure. When we try to access the value for the `enzyme` key in the elemental hash corresponding to `$key`, the `perl` interpreter automatically creates this elemental hash for us. This behavior is termed **autovivification**. Our test works anyway because the freshly created hash is empty and thus accessing the key `enzyme` returns `undef`. However, after that, an entry `$key => {}` is present in `%pattern_for`, which is now in a somewhat dirty state. If you want to keep the nested data structure pristine, you may either test for the existence of `$key` beforehand or forbid autovivification using the pragma `no autovivification`.

Then comes a `while` loop that uses the regular expression built from the restriction site pattern to compute cuts. It uses the global-search mode enabled with the corresponding regex modifier (`/g`) in a stepwise fashion to identify cleavage sites one at a time.

```
while ($dna_string =~ m/$regexp/g) {
    my $site = pos($dna_string) - $dist3p;   # $site is 1-based!
    # ...
}
```

This is needed because we want to capture the *position* of each match and not the matches themselves. To do that, we use the builtin function `pos`, which returns the position in the analyzed string where the regex engine last left off. Therefore, this is the character immediately following the match. Since we want to report 1-based cleavage sites, we do not need to subtract one from this 0-based value. However, we subtract the number of bases needed to generate the expected *sticky* (or *blunt*) ends.

Cuts are stored as nested anonymous hashes in a way very similar to what we have described in `read_enzymes`, except that elemental hashes are `push`ed onto the main array `@cuts` instead of being each associated to a specific key. It thus results in an *array of hashes*.

```
push @cuts, {
    enzyme => $enzyme,
    site   => $site,
};
```

Once cuts are computed, a `sort` statement orders the array by cleavage site. The sorting function is simple but still requires the dereferencing arrow because `$a` and `$b` are references to the two anonymous hashes under comparison.

```
@cuts = sort { $a->{site} <=> $b->{site} } @cuts;
```

Finally, the function returns the main array `@cuts` as a list to the caller, which stores the returned list in a lexical array of the same name.

```
### @cuts: [
###         {
###           enzyme => 'EcoRI',
###           site => 707
###         },
###         {
###           enzyme => 'XhoI',
###           site => 740
###         },
###           {
###             enzyme => 'KpnI',
###             site => 759
###           }
###         ]
```

6.4.2.2 collect_sites

The two remaining functions use `@cuts` for performing their duties. The first one collects the list of cleavage sites for each restriction enzyme.

```
sub collect_sites {
    my @cuts = @_;

    my %sites_for;
    for my $cut (@cuts) {
        my ($site, $enzyme) = @{ $cut }{ qw(site enzyme) };
        push @{ $sites_for{ $enzyme } }, $site;
    }

    return %sites_for;
}
```

Within the `for` loop, the first line uses a heavy dereferencing syntax that allows us to fetch and store the two values at once (hash slice). It is not pretty but not particularly complicated either: `$cut` is the hash reference that is dereferenced in a list context using `@{...}`, so that we can pass it several keys.

```
my ($site, $enzyme) = @{ $cut }{ qw(site enzyme) };
```

The second line `push`es the site position onto the anonymous array corresponding to this enzyme with the dereferencing syntax aimed at using an array reference as the referenced array (list context).

```
push @{ $sites_for{ $enzyme } }, $site;
```

This idiom works because the `perl` interpreter creates a fresh anonymous array the first time you try to `push` a value for a key that is not yet in the hash. This behavior is similar to the one we exploited for using hashes as counters (see `codon_usage.pl` and "Hash uses", in the first part of this course). The effect of this single statement is to progressively build a *hash of arrays*, such as the one shown below.

```
### %sites_for: {
###               EcoRI => [
###                          707
###                        ],
###               KpnI => [
###                         759
###                       ],
###               XhoI => [
###                         740
###                       ]
###             }
```

Here, each nested array has only one element, but using *4-cutter* enzymes yields a different picture.

```
### %sites_for: {
###              AluI => [
###                        57,
###                        314,
###                        529,
...
###                        2398
###                      ],
###              MspI => [
###                        329,
###                        696,
###                        753,
...
###                        2580
###                      ]
###            }
```

The collected sites are printed in the main `while` loop using two interesting constructs. First, the title of the output is set to capital letters using an uppercasing **escape sequence** beginning with `\U`.

```
say "\n# \UList of cleavage sites";
```

Second, each line of the table is assembled using a pretty hairy double `join` involving two different lists, each one `join`ed by its own separator:

- the tab character between the enzyme name and its list of cleavage sites,
- the comma character (`,`) between each site of the list.

```
say '# ' . join "\t", qw(enz sites);
for my $enzyme (sort keys %sites_for) {
    say join "\t", $enzyme, join ', ', @{ $sites_for{$enzyme} };
}
```

The dereferencing syntax for `%sites_for` is the same as used in the `push` explained a few lines above (elemental array used in list context).

6.4.2.3 `infer_fragments`

The second remaining function also uses `@cuts` to build another nested data structure holding the restriction fragments. Its logic is slightly more complicated.

```
sub infer_fragments {
    my ($dna_string, @cuts) = @_;

    unshift @cuts, { enzyme => q{5'-end}, site => 1                  };
    push    @cuts, { enzyme => q{3'-end}, site => length $dna_string };

    my @fragments;
    my ($enz1, $x1) = @{ shift @cuts }{ qw(enzyme site) };
    while (@cuts) {
        my ($enz2, $x2) = @{ shift @cuts }{ qw(enzyme site) };
        my $len = $x2 - $x1 + 1;
        warn 'WARNING! Some cleavage sites overlap; inaccurate results!'
            if $len < 1;

        push @fragments, [ $len, $x1, $x2, $enz1, $enz2 ];
        ($x1, $enz1) = ($x2+1, $enz2);
    }

    @fragments = sort { $b->[0] <=> $a->[0] } @fragments;

    return @fragments;
}
```

First, we add two new elemental hashes to `@cuts`, corresponding to the *5′-end* and *3′-end* of the DNA sequence viewed as linear (we do not handle circular sequences here). These hashes are added before the first cut and after the last cut, respectively, using `unshift` and `push`.

```
unshift @cuts, { enzyme => q{5'-end}, site => 1                 };
push    @cuts, { enzyme => q{3'-end}, site => length $dna_string };
```

Then, we setup a `while` loop for the purpose of iterating over `@cuts` using a *sliding window* of one (1) cut. This is needed since each cleavage site defines the boundary of its two flanking fragments.

```
my ($enz1, $x1) = @{ shift @cuts }{ qw(enzyme site) };
while (@cuts) {

    my ($enz2, $x2) = @{ shift @cuts }{ qw(enzyme site) };
    # ...
}
```

The enzyme name and site position for the current cut are copied from the nested hash reference as in the function `collect_sites`, except that we do not store the hash reference in a variable. Instead, we directly use the value `shift`ed from a lexical copy of `@cuts`.

```
my ($enz1, $x1) = @{ shift @cuts }{ qw(enzyme site) };
```

Whenever a 3′-cut is *recycled* as the 5′-cut of the next fragment, its site position is incremented by one.

```
($x1, $enz1) = ($x2+1, $enz2);
```

This could fail if two enzymes cut at the same site, hence the warning.

```
my $len = $x2 - $x1 + 1;
warn 'WARNING! Some cleavage sites overlap; inaccurate results!'
    if $len < 1;
```

The details for each fragment are stored in an anonymous array itself nested in the larger array `@fragments` using `push`. Remember that such arrays are created using square bracket characters.

```
push @fragments, [ $len, $x1, $x2, $enz1, $enz2 ];
```

I insist on this idiom (already covered in "Anonymous arrays", p.46) because it is very common. Yet, the line above could have been written more verbosely as follows.

```
my @data = ($len, $x1, $x2, $enz1, $enz2);
push @fragments, \@data;
```

Finally, the *array of arrays* `@fragments` is sorted numerically by fragment length in descending order (first element at index `0`) using the now familiar `sort` statement…

```
@fragments = sort { $b->[0] <=> $a->[0] } @fragments;
```

… and then returned as a list to the caller, which again stores it in a lexical array of the same name.

```
### @fragments: [
###              [
###                2202,
###                760,
###                2961,
###                'KpnI',
###                '3\'-end'
###              ],
###              [
###                707,
###                1,
###                707,
###                '5\'-end',
###                'EcoRI'
###              ],
###              [
###                33,
###                708,
###                740,
###                'EcoRI',
###                'XhoI'
###              ],
###              [
###                19,
###                741,
###                759,
###                'XhoI',
###                'KpnI'
###              ]
###            ]
```

The fragments for each input sequence are printed in the main `while` loop using a simple construct. There are two subtleties, though. First, we stop the uppercasing before the end using the `\E` sequence.

```
say "\n# \UList of fragments\E (>= $ARGV{'--min-frag-len'} bp)";
```

Second, we halt the printing of the fragments as soon as their length falls below the command-line threshold (`--min-frag-len`). This works because `@fragments` is sorted on descending length.

```
say '# ' . join "\t", qw(len 5'-pos 3'-pos 5'-enz 3'-enz);
```

```
FRAGMENT:
for my $fragment (@fragments) {
    last FRAGMENT if $fragment->[0] < $ARGV{'--min-frag-len'};
    say join "\t", @$fragment;
}
```

Note that the dereferencing syntax for accessing the anonymous array in list context (`@$fragment`) is lighter than above because the array reference is stored in a scalar variable (`$fragment`). However, it would have been perfectly valid to use the noisier syntax `@{$fragment}` instead.

Finally, in function `plot_map`, the construct for looping over `@fragments` and for fetching the details of each fragment in a list of scalar variables is a useful idiom for arrays (or hashes) of arrays.

```
FRAGMENT:
for my $fragment (@fragments) {
    my ($len, $x1, $x2, $enz1, $enz2) = @$fragment;

    last FRAGMENT if $len < $ARGV{'--min-frag-len'};

    # ...
}
```

BOX 9: Scalar references and alternate dereferencing syntax

As aforementioned, Perl often offers several ways to achieve the same meaning (TIMTOWTDI; see "if & unless", in the first part of this course), which is not always a good thing. This is especially true with the dereferencing syntax for nested data structures, for which there exist alternative ways of accessing element(s) that you may stumble on when perusing Perl scripts.

The general idea of this alternate syntax is to *double the sigil*. It comes from the way of dereferencing a scalar reference, a kind of beast that we have not encountered yet. References to scalar variables are useful for dealing with very large pieces of data because they are much cheaper (in terms of both speed and memory) to handle than the referenced data itself.

```
my $large_string = file('infile.txt')->slurp;
my $string_ref = \$large_string;     # take scalar ref on string

# and later (e.g., in a sub)...
say {$out} $$string_ref;             # dereference scalar ref for printing
```

For nested data structures, one has to distinguish between accessing a single element (scalar context) and several elements at once (list context). Below are specific examples that illustrate the syntactic differences between Modern Perl (MP) and *shorter-but-uglier* (SU) approaches. Note that failing to use `@` for fetching more than one element is always a bug (BG). This makes sense as the first sigil dictates the amount context (either scalar or list).

```
my $sites = $sites_for{'AluI'};      # $sites is an array ref
### $sites
### [MP] third site: $sites->[2]
### [SU] third site: $$sites[2]
### [MP] two first sites: @{ $sites }[0..1]
### [SU] two first sites: @$sites[0..1]
### [BG] two first sites: $$sites[0..1]
### [BG] two first sites: $sites->[0..1]

# gives:

### $sites: [
###           57,
###           314,
###           529,
...
###           2398
###         ]
### [MP] third site: 529
### [SU] third site: 529
### [MP] two first sites: 57,
###                       314
### [SU] two first sites: 57,
###                       314
### [BG] two first sites: 314
### [BG] two first sites: 314

my $cut = shift @cuts;              # $cut is a hash ref
### $cut
### [MP] enzyme: $cut->{enzyme}
### [SU] enzyme: $$cut{enzyme}
### [MP] enzyme and site: @{ $cut }{ qw(enzyme site) }
### [SU] enzyme and site: @$cut{ qw(enzyme site) }
### [BG] enzyme and site: $$cut{ qw(enzyme site) }
### [BG] enzyme and site: $cut->{ qw(enzyme site) }

# gives:

### $cut: {
###         enzyme => 'AluI',
###         site => 57
###       }
### [MP] enzyme: 'AluI'
### [SU] enzyme: 'AluI'
### [MP] enzyme and site: 'AluI',
###                       57
### [SU] enzyme and site: 'AluI',
###                       57
### [BG] enzyme and site: undef
### [BG] enzyme and site: undef
```

Chapter 7

Parsing BLAST reports

7.1 BLAST tabular format

To introduce Perl modules, we will write a reader for BLAST reports. BLAST reports come in many different flavors, but in most cases, the tabular file (TSV) format is the way to go because it is both compact and easy to read by a computer.

This format can be generated using the command-line BLAST executables by specifying the optional argument `-outfmt 6` (or `7`). The difference between `6` and `7` is that the latter format inserts a few comment lines before each result set (one set per *query*). This can be useful when processing BLAST reports directly at the command line.

Default BLAST tables have 12 columns that each contain one specific piece of data:

1. query id
2. subject (also known as hit) id
3. percent identity
4. alignment length
5. # mismatches
6. # gap opens
7. query start
8. query end
9. subject (or hit) start
10. subject (or hit) end
11. E-value
12. bit score

As already mentioned on several occasions, reading a structured text format, such as tabular BLAST reports, is called **parsing**. Thus, this section presents a **BLAST parser**. We will first generate a tabular report for the purpose of testing our future parser. To this end, follow the instructions below.

1. Download two sequences from *GenBank* (use `curl -o` on macOS).

```
$ wget -O queries.fasta \
    "https://eutils.ncbi.nlm.nih.gov/entrez/eutils/efetch.fcgi\
    ?db=protein&retmode=text&rettype=fasta&id=BAE95412,AAG33633"
```

2. Perform a *remote* BLASTP search against *GenBank*. (If you get one or more odd warnings, do not be afraid as long as the program continues to run and yields a report.)

   ```
   $ blastp -query queries.fasta -db nr -remote -evalue 1e-75 \
       -outfmt 7 -out report.blastp.fmt7
   ```

3. Have a look at the report. Do you understand its content?

   ```
   $ less report.blastp.fmt7
   ```

4. Produce a reduced version of the report for testing purposes. The new file should contain hit (subject) id (column 2), alignment length (column 4) and E-value (column 11) for the ten top results. Since the report begins with six comment lines (`-outfmt 7`), we ask for the 16 first lines (`head -n16`), of which we keep only the ten last ones (`tail -n10`), before selecting the three columns of interest (`cut -f2,4,11`).

   ```
   $ head -n16 report.blastp.fmt7 | tail -n10 | cut -f2,4,11 > expected.txt
   $ cat expected.txt
   ```

   ```
   XP_822944.1 329 0.0
   BAE16576.1  305 0.0
   BAE16577.1  323 0.0
   CCC93836.1  305 0.0
   BAE16575.1  305 0.0
   ...
   ```

7.2 How to write a Perl module?

The Perl community has settled on a series of guidelines for writing modules. These are quite demanding. We will not cover them in detail in this introductory course. However, some other modules exist to help us to write our own modules. Let's use one of them: `Module::Starter`! (In the following, I am assuming that you work in the `mod_perl` directory.)

```
$ cpanm Module::Starter
$ cd mod_perl/
$ module-starter
$ module-starter --module=Forem::BlastTable --author='Your Name' \
    --email='your.email@host.com' --minperl=5.12 --verbose
```

The helper module has built an empty module **distribution** that we only have to fill in. Using an incredible technology known as **templating**, it has written all the boilerplate code for us! When following the instructions below, be sure to stick to the specified file paths or the program will not run.

1. Enter the module directory. It already contains a number of files and directories, among which sub-directories `lib`, `t` and `xt`. Create two additional sub-directories `bin` and `test`.

   ```
   $ cd Forem-BlastTable/
   $ mkdir bin
   $ mkdir test
   ```

2. Move your data files for testing in the `test` sub-directory.

   ```
   $ mv ../queries.fasta ../report.blastp.fmt7 ../expected.txt test/
   ```

3. Create an additional automated test file for our Perl module in the `t` sub-directory. Its content is shown at the next section (see "The code for `blast_table.t`", p.95). You might want to add the `.t` suffix to geany configuration, so that it can highlight your Perl code even if the `.t` suffix is unknown to it. This can be done using the menu option `Tools > Configuration files > filetype_extensions.conf`.

   ```
   $ geany t/blast_table.t &
   ```

4. Similarly, create a standalone program making use of our module in the `bin` sub-directory. Again, its content is given below (see "The code for `parser.pl`", p.96).

   ```
   $ geany bin/parser.pl &
   ```

5. Edit the autogenerated `lib/Forem/BlastTable.pm` and replace its beginning with the content of the real `BlastTable.pm` shown below (see "The code for `BlastTable.pm`", p.97).

   ```
   $ geany lib/Forem/BlastTable.pm &
   ```

6. When you are done, install the missing modules and launch the automated tests.

   ```
   $ cpanm Test::Most
   $ cpanm Exporter::Easy
   $ prove -lv t/blast_table.t
   ```

7. Finally, test your program with the following command.

   ```
   $ perl -Ilib bin/parser.pl test/report.blastp.fmt7
   ```

7.2.1 The code for `blast_table.t`

```
#!/usr/bin/env perl

use Modern::Perl '2011';
use autodie;

use Path::Class 'file';
use Test::Most;

use Forem::BlastTable 'get_parser';

my $report = file('test', 'report.blastp.fmt7');
my $parser = get_parser($report);

my $exp_file = file('test', 'expected.txt');
my $exp_ref = read_exp_file($exp_file);

my @got;

HSP:
while ( my $hsp_ref = $parser->() ) {

    explain $hsp_ref;
```

```

    push @got, [ @{ $hsp_ref }{ qw(hit_id length evalue) } ];
    last HSP if @got == 10;
}

is_deeply \@got, $exp_ref,
    'correctly parsed the 10 first lines of the report';

done_testing;


sub read_exp_file {
    my $infile = shift;

    open my $in, '<', $infile;

    my @expected;
    while (my $line = <$in>) {
        chomp $line;
        push @expected, [ split /\t/xms, $line ];
    }

    return \@expected;
}
```

7.2.2 The code for `parser.pl`

```
#!/usr/bin/env perl

use Modern::Perl '2011';
use Forem::BlastTable 'get_parser';

die "Usage: $0 <report.blast.fmt7>" unless @ARGV == 1;

my $report = shift;
my $parser = get_parser($report);

my $query_id = q{};
while ( my $hsp_ref = $parser->() ) {

    if ( $query_id ne $hsp_ref->{query_id} ) {
        $query_id = $hsp_ref->{query_id};
        say "=== $query_id";
    }

    say join "\t", @{ $hsp_ref }{ qw(evalue length hit_id) };
}
```

7.2.3 The code for `BlastTable.pm`

```
package Forem::BlastTable;

use Modern::Perl '2011';
use autodie;

use Exporter::Easy (
    OK => [ qw(get_parser) ],
);

use List::AllUtils 'mesh';

=head1 NAME

Forem::BlastTable - A parser for NCBI BLAST tabular output

=head1 VERSION

Version 0.01

=cut

our $VERSION = '0.01';


=head1 SYNOPSIS

    use Forem::BlastTable 'get_parser';

    my $report = 'report.blast.fmt7';
    my $parser = get_parser($report);

    while ( my $hsp_ref = $parser->() ) {
        say join "\t", @{ $hsp_ref }{ qw(query_id hit_id length evalue) };
    }

=head1 EXPORT

This module exports nothing by default. Only one function is available for
import, C<get_parser>, which returns an HSP iterator over the BLAST report.

=head1 SUBROUTINES/METHODS

=head2 get_parser

=cut

```

```
sub get_parser {
    my $infile = shift;

    open my $fh, '<', $infile;

    # define HSP hash keys
    # slightly different from column names
    my @attrs = qw(
        query_id hit_id
        identity length mismatches gaps
        query_start query_end
          hit_start   hit_end
        evalue bits
    );

    # setup parser
    my $closure = sub {

        LINE:
        while (my $line = <$fh>) {
            chomp $line;

            next LINE if $line =~ m/\A \s* \z/xms;  # skip empty lines
            next LINE if $line =~ m/\A \#/xms;      # skip comment lines

            # process HSP line
            my @fields = split /\t/xms, $line;

            # Fields for BLAST -outfmt 6 or 7
            #    0. query id
            #    1. subject id
            #    2. % identity
            #    3. alignment length
            #    4. mismatches
            #    5. gap opens
            #    6. q. start
            #    7. q. end
            #    8. s. start
            #    9. s. end
            #   10. evalue
            #   11. bit score

            # here, we may add or modify some fields if needed

            # build and return anonymous HSP hash ref
            return { mesh @attrs, @fields };
        }

        return;         # no more line to read
    };

    return $closure;
}

=head1 AUTHOR

...
```

Homework

For our next lesson, I propose you to write an improved version of `parser.pl` (`hw8_parser_euclid.pl`). It should have the following features (do your best):

- a command-line interface based on `Getopt::Euclid`,
- an optional argument `--evalue` (similar to `--min-frag-len` in `cutter.pl`) allowing the user to limit the display of the BLAST results to the specified threshold,
- an optional argument `--fields` (similar to `--enzymes` in `cutter.pl`) allowing the user to specify the list of fields to be displayed by their names. (It might be useful to list the available fields in the documentation.) The argument `--fields` should have a default value that emulates the behavior of the current version of `parser.pl`.

Part IV

Lesson 9

Chapter 8

More on Perl modules

8.1 Structure of a Perl distribution

When we used `module-starter` for creating the distribution directory for `Forem::BlastTable`, the command automatically created a series of files and sub-directories for us. Let's briefly describe them.

`Changes` The file giving the list of bug fixes and new features of each release of the distribution.

`MANIFEST` The list of files included in the distribution.

`Makefile.PL` The file governing the building of the distribution. In principle, it should contain the dependencies (other Perl modules) required by the distribution.

`README` The classic starting point for the documentation of the distribution. It also provides a guide for installing the distribution.

`ignore.txt` The list of (temporary) files to be excluded from the packaged distribution.

`lib/` The sub-directory containing the actual code hierarchy for the modules included in the distribution. Each word in a module name begins with an uppercase letter, the remaining appearing in lowercase letters (a practice called **CamelCase**), and filenames further end with the `.pm` suffix (for *Perl Modules*), e.g., `BlastTable.pm`. Modules are written in standard Perl.

`t/` The sub-directory containing the code for the automated testing of the modules of the distribution. Test filenames use underscore characters between words and end with the `.t` suffix (for *Test*), e.g., `blast_table.t`. Again, they are written in standard Perl. Ideally, there should be at least one `.t` file per `.pm` file. There is also an `xt/` sub-directory supposed to contain so-called "author tests". You should not worry about them. Such tests are meant for Perl developers who care very much about the quality of their code— even more than you do!

With `Module::Starter`, we only need to edit (or add) files in `lib` and `t` sub-directories, the rest being handled for us. However, we created two other sub-directories in the `Forem-BlastTable` directory.

`test` The sub-directory containing the data files for testing the modules of the distribution.

`bin` (or `scripts`) The sub-directory containing the really useful programs taking advantage of the module(s) and distributed as an integral part of the module distribution. Once installed as executable files in the user's `$PATH`, they will be available from any directory (even after deleting the distribution files).

8.2 Installing Perl modules

So far, we have encountered two kinds of modules:

- modules included in the standard distribution of Perl, which do not need any installation,
- optional CPAN modules that we install using the `cpanm` command.

Except if we polish them for uploading to the CPAN website, our own modules belong to a third species. They also need to be installed, but manually. If we glance at the `README` file created for us, we see the following section.

```
INSTALLATION

To install this module, run the following commands:

        perl Makefile.PL
        make
        make test
        make install
```

This is the standard manual procedure for installing Perl modules. It is not that hard. Try to install our module `Forem::BlastTable` using the few commands above. Note that they should always be issued from the main directory of the distribution.

```
$ cd Forem-BlastTable/
$ perl Makefile.PL
...
```

However, we get an error message after the first installation command.

```
Perl v5.120.0 required (did you mean v5.12.0?)--this is only v5.32.0, \
    stopped at Makefile.PL line 1.
BEGIN failed--compilation aborted at Makefile.PL line 1.
```

Something went wrong… Apparently, we ask for a futuristic version of the `perl` interpreter (`5.120.0`), which will not be available before half a century, since there are two releases a year! How is it possible?

Well, we made a typo in the `module-starter` command. We specified `--minperl=5.12` instead of `--minperl=5.012`. How to fix that without restarting from scratch? This is a job for another one-liner (see "Perl *one-liners*", in the first part of this course).

First, look for the files that have to be corrected.

```
$ grep -r "\b5\.12\b" *
```

This recursive `grep` follows your directory structure and gives the following output.

```
Makefile.PL:use 5.12;
Makefile.PL:    MIN_PERL_VERSION => 5.12,
t/00-load.t:use 5.12;
t/manifest.t:use 5.12;
t/pod-coverage.t:use 5.12;
t/pod.t:use 5.12;
xt/boilerplate.t:use 5.12;
```

We do not see `lib/Forem/BlastTable.pm` nor `t/blast_table.t` because we replaced `use 5.12` by `use Modern::Perl '2011'` when we edited these files. To fix the remaining files, use the one-liner below. Then retry to install the module.

```
$ perl -i.bak -nle 's/5\.12/5.012/g; print' Makefile.PL t/*.t xt/*.t
$ perl Makefile.PL
$ make
$ make test
$ make install
```

If you are lucky, installation should complete successfully. Otherwise, you probably messed up the one-liner: replace each modified file found with the `grep` command by its original version (ending with the `.bak` suffix, as in "The `-i` switch", see the first part of this course), double-check the one-liner and retry the commands above.

In some rare cases, `make` doesn't work because the building process was left in a somewhat *unstable* state. If that happens, try `make clean` before `perl Makefile.PL` etc.

8.2.1 Fine-tuning the installation process

After some fiddling, `Forem::BlastTable` should be (hopefully) installed. The net result of the installation process is that our module is now available for `use` without having to explicitly specify the location of our **libraries**, which is another word for *modules*. Remember: the last time we tried our program, we launched it like this.

```
$ perl -Ilib bin/parser.pl test/report.blastp.fmt7
```

Now, we can drop the `perl -Ilib` part and simply use the following command.

```
$ parser.pl test/report.blastp.fmt7
```

```
-bash: parser.pl: command not found
```

WTF? Well, we forgot to tell `make` to install our `parser.pl` script. Therefore, it installed all our files except that one! To fix this issue, edit `Makefile.PL` and add the following configuration snippet as an additional key/value pair.

```
EXE_FILES => [
    'bin/parser.pl',
],
```

Then reinstall our module (from `perl Makefile.PL`) and retry our parser. It should work now…

In any case, installing a module is copying it to a specific location of your hard drive. This means that if you later modify something in the `.pm` library files in the `lib` sub-directory or in the `.pl` program files in the `bin` sub-directory, e.g., `parser.pl`, you will need to reinstall the module for the changes to be effective in a system-wide fashion.

8.2.2 Specifying module dependencies

While we are at editing `Makefile.PL`, we should also specify our dependencies, i.e., Perl modules on which our own module depends. These will help future users (and `cpanm`) to determine which modules are missing from the system and have to be installed prior to installing `Forem::BlastTable` itself. To find all our dependencies, use the funny command below.

```
$ grep -r '^use' * | cut -d' ' -f2 | cut -d';' -f1 | sort | uniq
```

It will recursively search for use lines in our files and extract the names of the used modules in a non-redundant way. The returned list should be something like this.

```
5.012
Exporter::Easy
ExtUtils::MakeMaker
Forem::BlastTable
List::AllUtils
Modern::Perl
Path::Class
Test::More
Test::Most
autodie
strict
warnings
```

Some of these modules (or pragmas) are part of the standard Perl distribution (e.g., `strict` and `warnings`); these will not need to be specified as dependencies in our `Makefile.PL`. In contrast, we have manually installed most of the other modules. We should thus list them in one (or more) of the hash keys shown in the table below (depending on the installation step requiring a given module).

Table 8.1: Keys to specify module dependencies in `Makefile.PL`

key	meaning
`CONFIGURE_REQUIRES`	needed to run `Makefile.PL` but not to run our module
`BUILD_REQUIRES`	needed to build our module but not to run it
`TEST_REQUIRES`	needed to test our module but not to run or build it
`PREREQ_PM`	needed to run our module

In our case, I dispatched the required modules in the keys below. It does not harm to list some modules that are actually part of the standard Perl distribution (e.g., `ExtUtils::MakeMaker`, `Test::More`).

```
CONFIGURE_REQUIRES => {
    'ExtUtils::MakeMaker' => 0,
},
TEST_REQUIRES => {
    'Test::More' => 0,
    'Test::Most' => 0,
},
PREREQ_PM => {
    'Exporter::Easy' => 0,
    'List::AllUtils' => 0,
    'Modern::Perl' => 0,
    'Path::Class' => 0,
    'autodie' => 0,
},
```

The values `0` mean that we do not ask for a minimum version of each module. Our `Makefile.PL` will be happy as long as these modules are installed on the system, even if their versions are old and buggy. Therefore, we should ideally specify the minimum version number of each module that our own module requires to run flawlessly. Given that modules are often updated by CPAN authors, this can reveal tricky: ask for a too old version and some bugs may crop up when running your code; ask for a too recent version and the user will need to update many of its modules before installing your own… Feel free to experiment with `PREREQ_PM` key/value pairs to understand how they work (see my `Makefile.PL` below for some ideas).

For more information about `Makefile.PL`, peruse the documentation of `ExtUtils::MakeMaker`.

If you want to learn about an almighty (yet intimidating) module designed to help programmers to write, package, manage and release their Perl module distributions, have a look at `Dist::Zilla`. Among many other things, it can automate the tedious process of gathering and updating module dependencies.

Finally, as we use Perlbrew, we can list all the modules we have installed in our sandbox (either directly or indirectly as a dependency of another module) with a simple command.

```
$ perlbrew list-modules
```

For reference, here's a listing of my (nearly) complete `Makefile.PL`.

```
use 5.012;
use strict;
use warnings;
use ExtUtils::MakeMaker;

my %WriteMakefileArgs = (
    NAME             => 'Forem::BlastTable',
    AUTHOR           => q{Denis BAURAIN <denis.baurain@ulg.ac.be>},
    VERSION_FROM     => 'lib/Forem/BlastTable.pm',
    ABSTRACT_FROM    => 'lib/Forem/BlastTable.pm',
    LICENSE          => 'artistic_2',
    MIN_PERL_VERSION => 5.012,
    CONFIGURE_REQUIRES => {
        'ExtUtils::MakeMaker' => 0,
    },
    TEST_REQUIRES => {
        'Test::More' => 0,
        'Test::Most' => 0,
    },
    PREREQ_PM => {
        'Exporter::Easy' => 0,
        'List::AllUtils' => 0,
        'Modern::Perl' => 0,
        'Path::Class' => 0,
        'autodie' => 0,
#       uncomment the following lines to see the effect of missing
#       or outdated modules on the command: perl Makefile.PL
#       'Missing::Module' => 0,
#       'Modern::Perl' => 3.4,
    },
    dist  => { COMPRESS => 'gzip -9f', SUFFIX => 'gz', },
    clean => { FILES => 'Forem-BlastTable-*' },
    EXE_FILES => [
        'bin/parser.pl',
    ],
);
...
```

8.3 Anatomy of a Perl module

The actual code for our module is in the file `BlastTable.pm`. The beginning and end of this file are what makes it a Perl module, whereas the remaining that lies in between is mostly standard Perl code.

```
package Forem::BlastTable;

use Modern::Perl '2011';
use autodie;

use Exporter::Easy (
    OK => [ qw(get_parser) ],
);

# ...

1; # End of Forem::BlastTable
```

Let's begin by the last statement: `1;`. All Perl modules must return a true value and the convention is to use this expression to this end. Helpers such as `Module::Starter` ensure that you do not forget it.

8.3.1 packages and namespaces

The very first line of the file simultaneously declares the **package** `Forem::BlastTable` and its associated **namespace**. The package corresponds to the source code, while the namespace is the entity that encapsulates this code once analyzed by the `perl` interpreter.

A namespace is a named collection of **symbols**, i.e., global variables and functions. Namespaces can be multi-level; in this case, each level is separated from the next one using a **double colon sequence** (`::`). Here are some namespaces that we have seen so far: `Smart::Comments`, `List::AllUtils`, `Getopt::Euclid`, `Term::Size::Any` and of course `Forem::BlastTable`.

Perl does not enforce any relationship between namespaces sharing one or more level names. These are only semantic conventions that hint the user at what the package might do. However, both in the `lib` sub-directory of the distribution and once installed in the right place on your hard drive, Perl modules are stored in a hierarchy of sub-directories perfectly mapping the nesting levels of the namespace (see table below).

Table 8.2: Examples of namespaces and directory hierarchies

namespace	directory hierarchy
`Forem::BlastTable`	`lib/Forem/BlastTable.pm`
`Forem::FastaFile`	`lib/Forem/FastaFile.pm`
`Term::Size::Any`	`lib/.../Term/Size/Any.pm`

User-defined namespaces should always begin with an uppercase letter. If a single naming level needs several words, these are concatenated using CamelCase, as already mentioned. Perl reserves lowercase package names for pragmas (e.g., `strict`, `warnings`, `autodie`). As often in Perl, these are only conventions… but stick to them!

Table 8.3: Examples of namespace spelling conventions

spelling	correctness
`BlastTable`	OK
`blastTable`	not OK: namespace should begin with an uppercase letter
`Blasttable`	not OK: each word should begin with an uppercase letter
`blasttable`	obviously not OK
`blast_table`	not OK for a namespace but perfect for a `.t` file
`Blast_Table`	good try but not OK!

After a `package` directive, we are in the specified namespace. The consequence is that all the symbols defined in this namespace can be referred to by their short name. In contrast, to access symbols from another namespace, you can either **import** them or use their **fully-qualified names**.

```
# using importing
use Term::Size::Any 'chars';
my ($cols, $rows) = chars();

# using fully-qualified names
use Term::Size::Any;
my ($cols, $rows) = Term::Size::Any::chars();
```

The scope of the package continues until the next `package` directive. In the absence of any `package` directive, the package defaults to `main`.

Each package has a **version number**, which is a series of integer numbers separated by dots, as in `1.23` or `1.1.10`. The package version is stored in the global variable `$VERSION` and can be obtained with the `VERSION()` method.

```
say Forem::BlastTable->VERSION;
```

Until Perl 5.10 included, `$VERSION` was defined as follows (using the our scope, a subtle variant of the lexical scope controlled by the `our` keyword). This is the way `Module::Starter` still works.

```
package Forem::BlastTable;
```

```
our $VERSION = '0.01';
```

Starting with Perl 5.12, you may use the more direct syntax below. Version numbers specified in this way must begin with a v and contain at least three series of integer numbers separated by dots.

```
package Forem::BlastTable v0.0.1;
```

If a use statement includes a version number, the `VERSION()` method of the corresponding module is called so as to check that its `$VERSION` indeed is recent enough. Otherwise, the script dies with an appropriate error message.

```
use Forem::BlastTable 1.2;

gives:

Forem::BlastTable version 1.2 required--this is only version 0.01 at blast.pl line 6.
BEGIN failed--compilation aborted at blast.pl line 6.
```

BOX 10: Multiple namespaces in source files

It is perfectly legal to declare several packages in a single source file, but don't do that. You can even add a symbol to another package than the current one by fully-qualifying the symbol.

```
package Forem::FastaFile;

sub read_fasta {
    # function body
}

package Forem::BlastTable;

sub get_parser {
    # function body
}

sub Forem::FastaFile::write_fasta {         # please avoid!
    # function body
}
```

8.3.2 Exporting symbols

Some packages export some (or all) of their symbols by default. Hence, simply using the module `POSIX` gives you access to the functions `floor` and `ceil` without needing to explicitly import them. Newer modules often do not export anything by default to avoid polluting your namespace.

The details about how symbol importing exactly works are beyond the scope of this course. However, we can easily mark symbols for importation in our own modules using the module `Exporter::Easy` and its simple configuration hash. Below is the relevant statement for `Forem::BlastTable`.

```
use Exporter::Easy (
    OK => [ qw(get_parser) ],
);
```

It means that the symbols listed in the nested anonymous array corresponding to the hash key `OK` *can* be imported by the file using our module. To import them, the latter file should do the following.

```
use Forem::BlastTable 'get_parser';
```

Symbols corresponding to the hash key `EXPORT` are exported by default, but again, this is now considered bad practice. Thus, I do not provide an example!

If we want to let the user import several symbols at once, we can group them using **tags** beginning with a single colon character (`:`). Here's an example from our next module, `Forem::FastaFile`.

```
use Exporter::Easy (
    OK   => [ qw(:io prefix_ids) ],
    TAGS => [
        io => [ qw(read_fasta write_fasta write_tmp_fasta) ],
    ],
);
```

The purpose of this is explained in the documentation of the module:

```
=head1 EXPORT

This module exports nothing by default. If you want to import all readers and
writers at once, use the tag C<:io> with the C<use> directive. Other functions
are available on an individual basis.
```

8.4 Automated tests

The Perl community has a very strong tradition for automatic testing. The consequence of this inclination is that Perl modules deposited on CPAN are generally of a high quality.

When developing a new module, you should always at least include so-called **unit tests**. These tests check that individual functions (or methods) of your Perl modules work as expected (and documented), but not the behavior of whole programs.

Unit tests are stored in the `t` sub-directory and are run when installing the modules of the distribution (with `make test`). They are also run when installing a module with `cpanm`. In principle, if any of these automated test fails, the distribution cannot be installed. This is why Perl modules are so robust in comparison to libraries from sloppier languages.

There are many ways of writing unit tests in Perl. Our approach is a classic one. We have one `.t` file for each `.pm` file, and within these `.t` files, we test each function marked as exportable at least once (in testing the more the better).

The next page starts with a template for our test files.

```
#!/usr/bin/env perl

use Modern::Perl '2011';
use autodie;

use Path::Class 'file';
use Test::Most;

# ...

done_testing;
```

The module `Test::Most` exports a number of specialized functions to help you write automated tests. We will cover them in "More on automated tests", p.151. The basic idea is to compare the return value of the tested function (what we *got*) with a *prediction* of the return value (what we *expected*). If both values match, the test passes; otherwise, the test fails with an informative error message.

To run our tests, we can simply enter the following command.

```
$ prove -lv t/*.t
```

The option `-l` is a shorthand for the `-Ilib` option of `perl`, while the option `-v` means *verbose*. Try without the latter option to see the difference. If everything works as expected, we get the heartwarming message below.

```
All tests successful.
```

When developing a module, you can also run only some specific test files.

```
$ prove -lv t/blast_table.t
```

Chapter 9

Our ultimate killer app

9.1 Annotating sequences with reference sequences

A common bioinformatics application is to annotate a set of poorly characterized DNA or protein sequences with a set of reference sequences. Generally, the *annotation* process is based on *sequence similarity* between the former and the latter.

If a sequence for which the function is unknown is very (or modestly) similar to another sequence for which the function is known, evolutionary genomics predicts that the first one might fullfil a function highly (or more distantly) related to the second one. This is the main strategy by which newly sequenced genomes are analyzed. Since annotation requires a lot of pairwise sequence comparisons, one often uses heuristic search tools, such as BLAST (for *Basic Local Alignment Search Tool*).

Here, we will develop an application for annotating a FASTA input file based on another FASTA file containing the reference sequences. This application is based on a real program used in a paper I published with my pal Marc Hanikenne (`http://hdl.handle.net/2268/160735`).

9.2 Packaging `read_fasta` and friends

First, we need a module for manipulating FASTA files. It will feature a slightly modified version of our function `read_fasta` that returns a reference to the hash `%seq_for` (instead of its content). Along with the reader, we will include functions for writing FASTA files and for modifying sequence identifiers.

The procedure to program the module `Forem::FastaFile` is detailed below.

1. Create the directory for the distribution. Watch the `--minperl` option!

```
$ cd mod_perl/
$ module-starter --module=Forem::FastaFile --author='Your Name' \
    --email='your.email@host.com' --minperl=5.012 --verbose
```

2. Enter the module directory and create a sub-directory `test`. We have no need for a `bin` sub-directory in this module.

```
$ cd Forem-FastaFile/
$ mkdir test
```

3. Copy your `phagemids.fasta` file in the `test` sub-directory under a new name: `infile.fasta`. Here's a possible command but your mileage may vary depending on your directory structure.

```
$ cp ../phagemids.fasta test/infile.fasta
```

4. Make a copy of this file under the name `expfile.fasta` and replace the four lengthy sequence identifiers by the following shorter identifiers: `seq1`, `seq2`, `seq3`, `seq4`. Do it with the one-liner explained above (see "The `/e` regex modifier", in the first part of this course).

```
$ cp test/infile.fasta test/expfile.fasta
# shorten sequence ids
$ perl -i.bak -nle 's/>.*/">seq" . ++$i /ge; print' test/expfile.fasta
# check that sequence ids are now indeed shorter
$ grep -A1 \> test/expfile.fasta
```

```
>seq1
CTAAATTGTAAGCGTTAATATTTTGTTAAAATTCGCGTTAAATTTTTGTTAAATCAGCTCATTTTTTAAC
--
>seq2
CTAAATTGTAAGCGTTAATATTTTGTTAAAATTCGCGTTAAATTTTTGTTAAATCAGCTCATTTTTTAAC
--
>seq3
CTGACGCGCCCTGTAGCGGCGCATTAAGCGCGGCGGGTGTGGTGGTTACGCGCAGCGTGACCGCTACACT
--
>seq4
CTGACGCGCCCTGTAGCGGCGCATTAAGCGCGGCGGGTGTGGTGGTTACGCGCAGCGTGACCGCTACACT
```

5. Create an automated test file for our Perl module in the `t` sub-directory (see "The code for `fasta_file.t`", p.117, for content).

```
$ geany t/fasta_file.t &
```

6. Edit the default `lib/Forem/FastaFile.pm` and alter its beginning to match my version (shown in "The code for `FastaFile.pm`", p.118).

```
$ geany lib/Forem/FastaFile.pm &
```

7. When you are done, install missing modules and launch the automated tests.

```
$ cpanm Const::Fast
$ cpanm Test::Files
$ prove -lv t/fasta_file.t
```

8. If everything runs smoothly, edit `Makefile.PL` to add our dependencies. These are exactly the same as `Forem::BlastTable`, supplemented by four new modules: `Test::Files` (under `TEST_REQUIRES`), `Const::Fast`, `File::Temp` and `Tie::IxHash` (under `PREREQ_PM`).

```
$ geany Makefile.PL &
```

9. Finally, proceed with installing the distribution.

```
$ perl Makefile.PL
$ make
$ make test
$ make install
```

9.2.1 The code for `fasta_file.t`

```perl
#!/usr/bin/env perl

use Modern::Perl '2011';
use autodie;

use List::AllUtils 'shuffle';
use Path::Class 'file';
use Test::Most;
use Test::Files;

use Forem::FastaFile qw(:io prefix_ids);

my $infile = file('test', 'infile.fasta');
my $seq_for = read_fasta($infile);
explain $seq_for;
my $exp_seq_n = 4;

# test 1
cmp_ok keys %$seq_for, '==', $exp_seq_n,
    "correctly read $exp_seq_n seqs from $infile";

# test 2
my $exp_ids = qx{grep \\> test/infile.fasta | cut -c2-};
explain $exp_ids;
my $got_ids = join("\n", keys %$seq_for) . "\n";
cmp_ok $got_ids, 'eq', $exp_ids,
    "correctly parsed the ids in $infile";

# test 3
my @exp_lens = (2961) x 4;
my @got_lens = map { length } values %$seq_for;
explain \@got_lens;
is_deeply \@got_lens, \@exp_lens,
    "correctly read the seqs in $infile";

# test 4
my $outfile = file('test', 'outfile.fasta');
$outfile->remove if -e $outfile;
write_fasta($outfile, $seq_for);
compare_ok $outfile, $infile,
    "correctly written $outfile";

# test 5
my $expfile = file('test', 'expfile.fasta');
my ($tmpfile, $id_for) = write_tmp_fasta($seq_for);
compare_ok $tmpfile, $expfile,
    "correctly written temporary $tmpfile";
explain $id_for;

# test 6
my @exp_ids = map { "seq$_" } 1..$exp_seq_n;
explain \@exp_ids;
my @got_ids = keys %$id_for;
cmp_bag \@got_ids, \@exp_ids,
    "correctly remapped ids in temporary $tmpfile";

# test 7
my $tmp_seq_for = read_fasta($tmpfile);
my @tags = map { 'type' . chr(64 + $_) } shuffle 1..$exp_seq_n;
explain \@tags;
my $i = 0;
my $ann_for = { map { $_ => $tags[$i++] } keys %$tmp_seq_for };
explain $ann_for;
my $ann_seq_for = prefix_ids($tmp_seq_for, $ann_for);
explain $ann_seq_for;
my @exp_ann_ids = map { $ann_for->{$_} . '-' . $_ } keys %$tmp_seq_for;
explain \@exp_ann_ids;
my @got_ann_ids = keys %$ann_seq_for;
is_deeply \@got_ann_ids, \@exp_ann_ids,
    "correctly prefixed ids using annotation hash";

done_testing;
```

9.2.2 The code for `FastaFile.pm`

```
package Forem::FastaFile;

use Modern::Perl '2011';
use autodie;

use Exporter::Easy (
    OK   => [ qw(:io prefix_ids) ],
    TAGS => [
        io => [ qw(read_fasta write_fasta write_tmp_fasta) ],
    ],
);

use Const::Fast;
use File::Temp;
use Tie::IxHash;

const my $CHUNK_LEN => 70;

```

```

=head1 NAME

Forem::FastaFile - Reader and Writers for FASTA files

=head1 VERSION

Version 0.01

=cut

our $VERSION = '0.01';


=head1 SYNOPSIS

    use Forem::FastaFile ':io';

    my $infile = 'infile.fasta';
    my $seq_for = read_fasta($infile);

    while ( my ($seq_id, $seq) = each %$seq_for ) {
        say "id: $seq_id";
        say "seq: $seq";
    }

    my $outfile = 'outfile.fasta';
    write_fasta($outfile, $seq_for);

    my ($tmp_file, $id_for) = write_tmp_fasta($seq_for);

    # given an annotation hash %ann_for
    my $annfile = 'annotated.fasta';
    write_fasta( $annfile, prefix_ids($seq_for, \%ann_for) );

=head1 EXPORT

This module exports nothing by default. If you want to import all readers and
writers at once, use the tag C<:io> with the C<use> directive. Other functions
are only available on an individual basis.

=head1 SUBROUTINES/METHODS

=head2 INPUT/OUTPUT

=head3 read_fasta

```

```
Reads a FASTA file and returns a reference to an ordered hash containing the
id/seq pairs. This function takes only one argument: the FASTA filename.

=cut

sub read_fasta {
    my $infile = shift;

    open my $in, '<', $infile;

    my $seq_id;
    my $seq;
    tie my %seq_for, 'Tie::IxHash';         # preserve original seq order

    LINE:
    while (my $line = <$in>) {
        chomp $line;

        # at each '>' char...
        if (substr($line, 0, 1) eq '>') {

            # add current seq to hash (if any)
            if ($seq) {
                $seq_for{$seq_id} = $seq;
                $seq = q{};
            }

            # extract new seq_id
            $seq_id = substr($line, 1);
            next LINE;
        }

        # elongate current seq (seqs can be broken on several lines)
        $seq .= $line;
    }

    # add last seq to hash (if any)
    $seq_for{$seq_id} = $seq if $seq;

    return \%seq_for;
}

=head3 write_fasta

Writes a FASTA file. Sequences are wrapped at 70 chars. This function takes two
arguments: the FASTA filename and a reference to an ordered hash containing the
id/seq pairs
```

```

=cut

sub write_fasta {
    my $outfile = shift;
    my $seq_for = shift;

    open my $out, '>', $outfile;

    while (my ($seq_id, $seq) = each %$seq_for) {
        say {$out} ">$seq_id\n" . _wrap_seq($seq);
    }

    return;
}

=head3 write_tmp_fasta

Writes a temporary FASTA file for helper programs. Ids are abbreviated and
sequences are wrapped at 70 chars. This function takes only one argument: a
reference to an ordered hash containing the id/seq pairs. It returns a list of
two values: the temporary FASTA filename and a reference to an unordered hash
containing the abbr-id/full-id pairs.

=cut

sub write_tmp_fasta {
    my $seq_for = shift;

    # open temporary file
    my $out = File::Temp->new(UNLINK => 0, EXLOCK => 0, SUFFIX => '.fasta');

    # remap ids on the fly when writing temporary FASTA file
    # seq ids in temp file are seq1, seq2 etc.
    my %id_for;
    for (my $i = 1; my ($seq_id, $seq) = each %$seq_for; $i++) {
        my $tmp_id = "seq$i";
        say {$out} ">$tmp_id\n" . _wrap_seq($seq);
        $id_for{$tmp_id} = $seq_id;
    }

    # return temp file name and id mapping hash ref
    return ($out->filename, \%id_for);
}

# private sub ; undocumented and unexportable

```

```perl
sub _wrap_seq {
    my $seq = shift;

    my $seq_len = length $seq;

    # wrap seq in chunks of length $CHUNK_LEN
    my $wrapped_seq;
    for (my $i = 0; $i < $seq_len; $i += $CHUNK_LEN) {
        $wrapped_seq .= substr($seq, $i, $CHUNK_LEN) . "\n";
    }

    return $wrapped_seq;
}

=head2 ANNOTATION

=head3 prefix_ids

Prefix each seq id with a tag giving its annotation and returns the new hash
in the same order as the original hash. This function takes two arguments: a
reference to an ordered hash containing the id/seq pairs and a reference to a
(unordered) hash containing the id/tag pairs.

=cut

sub prefix_ids {
    my $seq_for = shift;
    my $ann_for = shift;

    tie my %new_hash, 'Tie::IxHash';

    while (my ($seq_id, $seq) = each %$seq_for) {
        my $prefix = $ann_for->{$seq_id};
        $new_hash{ $prefix ? $prefix . '-' . $seq_id : $seq_id } = $seq;
    }

    return \%new_hash;
}


=head1 AUTHOR

...
```

9.3 Building our application

It is now time to reap the benefits of all our efforts…

1. In the directory above `Forem` (e.g., directly in `mod_perl`), create a new program file and save it as `annotate.pl`. Its content is shown in "The code for `annotate.pl`", p.123.

   ```
   $ geany annotate.pl &
   $ chmod a+x annotate.pl
   ```

2. When you are done, install missing modules. (Yes, each new killer app comes with a new batch of great modules: these two ones will be briefly introduced in "Building programs with programs", p.156, and in "Calling external programs", p.158.)

   ```
   $ cpanm Template;
   $ cpanm IPC::System::Simple
   ```

3. Finally, try the program with the following commands (you may need to edit the file paths).

   ```
   $ perldoc annotate.pl
   $ ./annotate.pl --help

   $ ./annotate.pl phagemids.fasta --ref-file Ecoli_cds.fasta \
       --ref-regex '^(\w+)'

   $ ./annotate.pl phagemids.fasta --ref-file Ecoli_cds_pep.fasta \
       --ref-regex '^(\w+)'

   $ ./annotate.pl phagemids.fasta --ref-file Ecoli_cds_pep.fasta \
       --ref-regex '^(\w+)' --evalue=1e-20

   $ ./annotate.pl phagemids.fasta --ref-file Ecoli_cds_pep.fasta \
       --ref-regex '^(\w+)' --evalue=1e-30

   $ ./annotate.pl phagemids.fasta --ref-file Ecoli_cds_pep.fasta \
       --ref-regex '^(\w+)' --evalue=1e-20 --write-ann-file
   ```

9.3.1 The code for `annotate.pl`

```
#!/usr/bin/env perl

use Modern::Perl '2011';
use autodie;

use Getopt::Euclid;
use Smart::Comments '###';
use Template;

use File::Basename;
use Path::Class 'file';

```

```
use IPC::System::Simple qw(system);
use List::AllUtils 'any';

use Forem::FastaFile qw(:io prefix_ids);
use Forem::BlastTable 'get_parser';

#### Arguments: %ARGV

### Reading infile: $ARGV{'<infile>'}
my ($seq_for, $intype, $infile,  $inid_for) = process_file($ARGV{'<infile>'});

### Reading infile: $ARGV{'--ref-file'}
my (  undef, $reftype, $refile, $refid_for) = process_file($ARGV{'--ref-file'});

# determine blast program based on type combination
my %pgm_for = (
    'prot:prot' =>  'blastp',
    'nucl:prot' =>  'blastx',
    'prot:nucl' => 'tblastn',
    'nucl:nucl' => 'tblastx',
);
my $pgm = $pgm_for{ "$intype:$reftype" };
### assert: $pgm

# define command template
my $report = "$infile.blast.fmt7";
my $template = <<'EOT';
makeblastdb -in [% refile %] -dbtype [% reftype %]
[% pgm %] -query [% infile %] -db [% refile %] -evalue [% E %] -outfmt 7 -out [% report %]
EOT

# build command
my %vars = (
    refile   => $refile,
    reftype  => $reftype,
    pgm      => $pgm,
    infile   => $infile,
    E        => $ARGV{'--evalue'},
    report   => $report,
);
my $command;
my $tt = Template->new;
$tt->process(\$template, \%vars, \$command);
#### $command

### Performing BLAST...
system($command);
```

```

### Parsing BLAST report...
my $parser = get_parser($report);
my %ann_for;
my $curr_id = q{};

HSP:
while ( my $hsp_ref = $parser->() ) {
    my ($qid, $hid, $evalue) = @{ $hsp_ref }{ qw(query_id hit_id evalue) };

    next HSP if $evalue > $ARGV{'--evalue'};        # skip weak hits
    next HSP if $qid eq $curr_id;                   # skip non-first hits
    $curr_id = $qid;

    # capture annotation bit in ref seq id using regex
    my ($annotation) = $refid_for->{$hid} =~ $ARGV{'--ref-regex'};
    $ann_for{ $inid_for->{$qid} } = $annotation;
}
#### Annotations: %ann_for

say '# ' . join "\t", qw(tag id);
for my $id (sort keys %ann_for) {
    say join "\t", $ann_for{$id}, $id;
}

if ($ARGV{'--write-ann-file'}) {
    my ($basename, $dir, $ext) = fileparse($ARGV{'<infile>'}, qr{\.[^.]*}xms);
    my $outfile = file($dir, $basename . '_ann' . $ext);
    ### Writing annotated file: $outfile->stringify
    write_fasta( $outfile, prefix_ids($seq_for, \%ann_for) );
}

sub process_file {
    my $infile = shift;

    my $seq_for = read_fasta($infile);
    my $type = ( any { m/[EFILPQ]/i } values %$seq_for ) ? 'prot' : 'nucl';
    my ($tmpfile, $id_for) = write_tmp_fasta( {
        map { $_ => $seq_for->{$_} =~ tr/-//dr } keys %$seq_for
    } );  # degap seqs on the fly

    return ($seq_for, $type, $tmpfile, $id_for);
}

=head1 NAME

annotate - Annotate a sequence file by similarity to reference sequences
```

```

=head1 VERSION

This documentation refers to annotate version 0.0.1

=head1 USAGE

    annotate.pl <infile> --ref-file <infile> --ref-regex <regex> [options]

=head1 REQUIRED ARGUMENTS

=over

=item <infile>

Path to input FASTA file.

=for Euclid:
    infile.type: readable

=item --ref-file [=] <infile>

Path to reference FASTA file.

=for Euclid:
    infile.type: readable

=item --ref-regex [=] <regex>

Regular expression for capturing the reference seq id part that has to be used
for annotating infile seq ids [default: none].

=for Euclid:
    regex.type: string

=back

=head1 OPTIONS

=over

=item --evalue [=] <float>

E-value threshold for annotating a sequence [default: float.default].

=for Euclid:
    float.type: number
    float.default: 1e-10

=item --write-ann-file

Write an annotated version (with prefixed ids) of the infile [default: no];

=item --version

=item --usage

=item --help

=item --man

Print the usual program information

=back

=head1 AUTHOR

Your Name (your.email@host.com)

=head1 BUGS

There are undoubtedly serious bugs lurking somewhere in this code.
Bug reports and other feedback are most welcome.

=head1 COPYRIGHT

Copyright (c) 2013, Your Name. All Rights Reserved.
This program is free software. It may be used, redistributed
and/or modified under the terms of the Perl Artistic License
(see http://www.perl.com/perl/misc/Artistic.html)
```

Homework

For the next lesson, I will ask you something slightly different: search CPAN for suitable modules in order to add a few bells and whistles to your BLAST parser. The exercise lies both in the quest for the best modules (i.e., functionality, programming interface, documentation) and in your efforts to understand how they work and make proper use of them (by perusing the synopsis, reading the examples and possibly studying the test files).

Your improved parser (`hw9_parser_euclid_pro.pl`) should have the additional functionalities described just below with respect to `hw8_parser_euclid.pl`.

1. The optional argument `--excel` replaces the screen output of the BLAST report by the generation of a **Microsoft Excel** (`.xlsx`) output file. This file should be named after the input file (i.e., `report.blastp.fmt7` would transform into `report.xlsx`). Each query should lead to a different spreadsheet in the Excel workbook (i.e., two queries would yield two sheets). Finally, the header of each sheet (i.e., the first line providing the field/column names) should be formatted differently from the regular rows (e.g., in boldface or on background color, as you wish).
2. The optional argument `--tsv` stores the current screen output (tab-separated) into a TSV output file named according to the same scheme (i.e., `report.tsv`). This point does not require a specific module (since it already works with `join "\t"`). However, your program should be able to accept both the options `--excel` and `--tsv` if provided simultaneously by the user.
3. In any case, the parser should generate a screen output that is *pretty to look at* (i.e., the content of the different columns has to be vertically aligned). Optionally, you might want to add a specially-formatted header, as well as horizontal and/or vertical rules.

Part V

Lesson 10

Chapter 10

Idiomatic Perl

10.1 More on closures

In "Closures and BEGIN code blocks", p.79, we introduced the concept of closure while discussing the BEGIN code block. The function get_parser in our module Forem::BlastTable also uses a closure, but in a more complex setting.

Since this whole section is conceptually more difficult than anything else in this course, it should probably be enclosed in a big blue box. If you are not afraid, please proceed and learn more about closures! Otherwise, skip this part and go directly to "The default variables", p.137.

10.1.1 Function references

The goal of Forem::BlastTable is to encapsulate the parsing of a tabular BLAST report. From the user's point-of-view, one call to get_parser opens the report and returns an **iterator function** on it. Then, each call to the iterator returns a line of the report until its end, where it returns undef instead.

```
my $parser = get_parser($report);
while ( my $hsp_ref = $parser->() ) {
    # process HSP
}
```

In the code above, $parser is a **function reference**, i.e., a reference to a sub. If we try to print this variable, it stringifies as any reference, even though Smart::Comments prints it in a generic way.

```
say "$parser";
### $parser

# gives:

CODE(0x7ffe688299b8)
### $parser: sub { "DUMMY" }
```

The interest of a function reference is that we can invoke it nearly as if it was a regular function. It only requires the dereferencing arrow (->) followed by a pair of parenthesis characters. Our BLAST report iterator does not use parameters, but were it the case, the corresponding arguments would have been placed between these parentheses.

```
my $hsp_ref = $parser->();
```

Note that the parentheses are mandatory, even here, so that `perl` understands that we want to *call* the referenced function and not just to deal with its reference. Compare the following examples.

```
my $hsp_ref = $parser->();
### $hsp_ref

# my $error = $parser->;
# does not compile: syntax error at bin/parser.pl line 20, near "->;"

my $oups = $parser;
### $oups

# gives:

### $hsp_ref: {
###             bits => ' 543',
###             evalue => '0.0',
###             gaps => '0',
###             hit_end => '305',
###             hit_id => 'gi|71609884|dbj|BAE16576.1|',
###             hit_start => '1',
###             identity => '84.26',
###             length => '305',
###             mismatches => '48',
###             query_end => '305',
###             query_id => 'gi|108743276|dbj|BAE95412.1|',
###             query_start => '1'
###           }

### $oups: sub { "DUMMY" }
```

As you can see, the first example actually calls the referenced function, which answers by returning a hash reference to the next HSP (for *High-Scoring Pair*) in BLAST parlance. In contrast, the second example does not compile, whereas the third one only copies the reference itself to a new variable.

10.1.2 subs with a memory

Such a straightforward user interface implies that the iterator function somehow remembers its current position in the report. In other words, the iterator has a **state**. To implement this approach, we combine two different tricks: a closure and a function reference to an anonymous function. Since these concepts are quite complex, let's first consider the program below.

```
#!/usr/bin/env perl

use Modern::Perl '2011';
```

```

die "Usage: $0 <start> <times>" unless @ARGV == 2;

my $start = shift;
my $times = shift;

my $next = get_next($start);
say $next->() for 1..$times;

sub get_next {
    my $i = shift;

    my $closure = sub {
        return ++$i;
    };

    return $closure;
}
```

`get_next` declares a lexically scoped counter variable `$i` and defines it to its argument `$start` (provided by the user). In principle, this variable should be destroyed when the function quits, but it will not because of the returned reference to the closure that makes use of it.

The closure itself is created as an anonymous `sub` to which we directly take a reference. This syntax can be compared to the one used for creating anonymous arrays and hashes. It is also possible to take a reference to a named `sub` using the reference operator (`\`), but we do not do that here.

Every time the function reference stored in `$next` is invoked, it increments the counter `$i` and returns its new value. In this toy example, we call it `$times` times (again user-defined). The closure is thus a `sub` with a *memory*. Here is a typical execution of `get_next.pl`.

```
$ ./get_next.pl 7 5
8
9
10
11
12
$
```

Note that Perl provides another way to implement lexical variables that persist across function calls. Such variables have to be declared with the `state` keyword (instead of `my`). However the initialization behavior of `state` variables is rather tricky, and implementing multiple custom counters (as it would be possible with `get_next`) still requires references to anonymous functions. That is why I would advise you to avoid them and use closures instead.

Now, let's turn to `get_parser`!

```
sub get_parser {
    my $infile = shift;
    open my $fh, '<', $infile;

    # define HSP hash keys
    # slightly different from column names
    my @attrs = qw(
        query_id hit_id
        identity length mismatches gaps
        query_start query_end
          hit_start   hit_end
        evalue bits
    );

    # setup parser
    my $closure = sub {
        # anonymous function body
    };

    return $closure;
}
```

`get_parser` opens the BLAST report passed as `$infile` and defines an array of keys for the HSP hashes. This creates two additional lexically scoped variables: `$fh` and `@attrs`. As with `get_next`, these variables are not destroyed when the function quits because of the returned reference to the closure making use of them.

```
my $closure = sub {

    LINE:
    while (my $line = <$fh>) {
        chomp $line;

        next LINE if $line =~ m/\A \s* \z/xms;  # skip empty lines
        next LINE if $line =~ m/\A \#/xms;      # skip comment lines

        # process HSP line
        my @fields = split /\t/xms, $line;

        # build and return anonymous HSP hash ref
        return { mesh @attrs, @fields };
    }

    return;         # no more line to read
};
```

Within the closure, we find a regular-looking input file reading loop quite similar to what we wrote in `cutter.pl` for reading the restriction enzyme database. There are a number of oddities, though.

- The file is read through `$fh`, the filehandle reference created in the enclosing `get_parser` function. Due to the long-lasting nature of the closure, and therefore of the grip it has on the variable

`$fh`, the latter cannot be closed and destroyed at the end of `get_parser`. This distinctive feature also defines how the iterator remembers its state.

- The hash holding the details of the current HSP is built by the handy `mesh` function using the elements of `@attrs` as keys. For this approach to work, the keys must be in the exact same order as the columns of the BLAST report. Similar to `$fh`, the array `@attrs` is declared in the outer lexical scope and survives because of its usage in the closure.
- The current HSP hash is built as an anonymous hash, to which we take a reference that we `return` to the caller of the iterator. Since this interrupts the loop after each HSP, we could have replaced the `while` by a single `if`. However, using a loop allows us to handle empty lines and comments in the BLAST report and helps the reader to understand that it is an iterator.
- The last `return` is executed as soon as the whole report has been read. That is how the iterator returns `undef` on completion. Everything will eventually be closed and/or destroyed when the function reference to the iterator gets out of scope on the caller side.

10.2 The default variables

The linguistic roots of Perl design have led to the incorporation of another important concept into the language: **pronouns**. These materialize as three main default variables:

- the default arrays `@_` and `@ARGV` (*they/them*),
- the default scalar variable (or **topic variable**) `$_` (*it*).

We have already explained that, in the absence of an explicit array to work on, list-oriented builtin functions (such as `shift`) default to operating on `@_` (within `subs`; see "Argument aliasing", p.29) or on `@ARGV` (outside `subs`; see "`unshift` & `shift`", in the first part of this course).

```
# from xxl_xlate.pl: outside subs
my $infile = shift;              # shift @ARGV
my $gcfile = shift;
my $gc_id  = shift;

# from cutter.pl: inside sub
sub compute_cuts {
    my $dna_string = shift;      # shift @_
    # remaining of function body
}
```

Similarly, many Perl builtin functions (e.g., `chomp`, `print`) and regular expressions work on the default topic variable `$_` when no variable is specified. For example, look at this idiomatic rewrite of the reading loop in the function `read_enzymes` of `cutter.pl`.

```
LINE:
while (<$in>) {
    chomp;

    next LINE if m/\A \s* \z/xms;              # skip empty lines
    next LINE if m/\A \#/xms;                  # skip comment lines
    my ($enzyme, $dist3p, $pattern) = split /\t/xms;
    # ...
}
```

Relying on this behavior is useful in some situations, such as in one-liners and implicit loop constructs, which we will describe in "Implicit loops", p.140. Bear in mind, however, that the resulting code can become very hard to follow. As an illustration, compare the rewrite above with the original version of the same chunk of code (shown below). Don't you think that the latter one reads better?

```
LINE:
while (my $line = <$in>) {
    chomp $line;

    next LINE if $line =~ m/\A \s* \z/xms;      # skip empty lines
    next LINE if $line =~ m/\A \#/xms;          # skip comment lines

    my ($enzyme, $dist3p, $pattern) = split /\t/xms, $line;
    # ...
}
```

10.2.1 The topic variable as the default iterator

Consider the two examples below. In the second one, the topic variable `$_` is used as the default iterator variable of the *foreach-style* `for` loop.

```
for my $i (1..3) {
    say $i;
    ### $i
}
for (1..3) {
    say;
    ### $_
}

# gives:

1
### $i: 1
2
### $i: 2
3
### $i: 3

1
### $_: 1
2
### $_: 2
3
### $_: 3
```

Granted, as is, this construct does not look very compelling. However, combined to a postfix `for` loop, it becomes much more natural (see next page after this box).

BOX 11: The topic variable in one-liners

As a worked example of using `$_` in Perl one-liners, let's explain how we built the restriction enzyme database for `cutter.pl` (see "How to build an enzyme database?", p.55).

```
$ perl -nle 'next if m/MboII/; m{/([^\/]+)/\s+\((\w+)\s+[^\)]+\)(\d+),?}; \
    print join "\t", $2, $3, $1' patterns.txt > enzyme-db.txt
```

To understand it, let's examine its expanded version.

```
while (my $infile = shift) {                # patterns.txt
    open $in, '<', $infile;
    while (<$in>) {
        chomp;                                      # remove newline
        next if m/MboII/;                           # skip line matching MboII
        m{/([^\/]+)/\s+\((\w+)\s+[^\)]+\)(\d+),?};  # analyze line using regex
        say join "\t", $2, $3, $1';                 # output numbered captures
    }
}
```

The regex parses lines in `patterns.txt` and captures three interesting bits using parentheses:

1. the recognized DNA pattern,
2. the enzyme name,
3. the distance of the cleavage site from the 3'-end of the pattern.

```
/gacgtc/ (AatII gacgt|c)1,
/tccgga/ (AccIII t|ccgga)5,
/agct/ (AluI ag|ct)2,
/gggccc/ (ApaI gggcc|c)1,
...
/tctaga/ (XbaI t|ctaga)5,
/ctcgag/ (XhoI c|tcgag)5,
/cccggg/ (XmaI c|ccggg)5
```

The captured elements are then reordered, joined with a tab character (`\t`) and printed to the standard output. On the command line, we redirect the standard output stream to a new file `enzyme-db.txt` using > to permanently store the reformatted enzyme database.

```
AatII   1   gacgtc
AccIII  5   tccgga
AluI    2   agct
ApaI    1   gggccc
...
XbaI    5   tctaga
XhoI    5   ctcgag
XmaI    5   cccggg
```

```
say 'human genome size is:';
my $size = 3.08e9;
say '- ' . 1000 * ($size /= 1000) . ' ' . $_ for qw(bp kb Mb Gb);

# gives:

human genome size is:
- 3080000000 bp
- 3080000 kb
- 3080 Mb
- 3.08 Gb
```

I readily admit that the preceding example remains convoluted. Here's another one that makes use of object-oriented programming. Imagine that we have an object `Seq` that can *degap* itself (using a `tr/-//dr` construct) and an object `Ali` that holds a collection of `Seq`s. Then, to degap all `Seq`s at once, we use a postfix `for` loop and the topic variable.

```
# in Seq.pm
sub degap {
    my $self = shift;
    $self->_set_seq( $self->seq =~ tr/-//dr );      # degap one seq
    return;
}

# in Ali.pm
sub degap_seqs {
    my $self = shift;
    $_->degap for $self->all_seqs;                  # degap all seqs
    return;
}
```

10.3 Implicit loops

During all this course, I refrained from using the powerful `grep` and `map` **implicit loop** constructs. Now, it is time to introduce them to you.

Both setup a loop over a list of values (or an array evaluated in a list context) and executes the corresponding block for each value of the list *aliased* in turn to the topic variable `$_`. Then, `grep` only returns the values for which the expression in the block evaluates to a true value, whereas `map` passes all the values, albeit often transformed through the evaluation of the expression.

10.3.1 `grep`

Here's an example of `grep`. Take a moment to ponder the power packed in this concise statement.

```
my %taxon_for = (
    amoeba => 'Amoebozoa',
    arabidopsis => 'Plantae',
    chlamydomonas => 'Plantae',
    human => 'Opisthokonta',
    yeast => 'Opisthokonta',
    choanoflagellate => 'Opisthokonta',
    diatom => 'Stramenopiles',
    trypanosome => 'Euglenozoa',
);

my @genera = qw(arabidopsis diatom human trypanosome yeast);

my @opisthokonts = grep { $taxon_for{$_} eq 'Opisthokonta' } @genera;
### @opisthokonts

# gives:

### @opisthokonts: [
###                  'human',
###                  'yeast'
###                ]
```

Yes, grep is a list filtering tool. The values that successfully pass through the filter remain in their original order. It is the exact equivalent of the following code snippets. I present them from the longest to the shortest, so as to show you where the final terseness comes from.

```
# explicit prefix if version
my @opisthokonts;
for my $genus (@genera) {
    if ($taxon_for{$genus} eq 'Opisthokonta') {
        push @opisthokonts, $genus;
    }
}

# explicit postfix if version
my @opisthokonts;
for my $genus (@genera) {
    push @opisthokonts, $genus
        if $taxon_for{$genus} eq 'Opisthokonta';
}

# implicit postfix if version (using the topic variable $_)
my @opisthokonts;
for (@genera) {
    push @opisthokonts, $_
        if $taxon_for{$_} eq 'Opisthokonta';
}

# grep version (also using the topic variable $_)
my @opisthokonts = grep { $taxon_for{$_} eq 'Opisthokonta' } @genera;
```

The next time you need to scan an array looking for some specific values, think grep (or even better, first transform the array into a Boolean filter hash, as in "Hash uses", in the first part of this course).

Here's another example modified from `cutter.pl`. Note that the filtered values are array references but that the filtering criterion only applies to one element of each of these arrays. This allows us to keep only the fragments reaching our length threshold in a very straightforward way.

```
sub infer_fragments {

    # ...

    while (@cuts) {
        # ...
        push @fragments, [ $len, $x1, $x2, $enz1, $enz2 ];
        # ...
    }

    # filter out short fragments and order remaining ones by desc len
    @fragments = sort { $b->[0] <=> $a->[0] }
                 grep { $_->[0] >= $ARGV{'--min-frag-len'} } @fragments
    ;

    return @fragments;
}
```

10.3.2 map

`map` is even more powerful than `grep` and its applications are very wide. Let's start with a simple example elaborating on our taxonomic experiments (using the same hash `%taxon_for` as in p.140).

```
my @genera = qw(arabidopsis diatom human trypanosome yeast);

my @taxa = map { $taxon_for{$_} } @genera;
### @taxa

# gives:

### @taxa: [
###          'Plantae',
###          'Stramenopiles',
###          'Opisthokonta',
###          'Euglenozoa',
###          'Opisthokonta'
###        ]
```

Here, the `map` construct was meant to replace one of these code snippets.

```
# explicit version
my @taxa;
for my $genus (@genera) {
    push @taxa, $taxon_for{$genus};
}
```

```
# implicit version (using the topic variable $_)
my @taxa;
for (@genera) {
    push @taxa, $taxon_for{$_};
}

# map version (also using the topic variable $_)
my @taxa = map { $taxon_for{$_} } @genera;
```

Below, I give a few more `map` examples from `fasta_file.t`. If you are like me and fall in love with such constructs, do not forget to have a look at the section "Bonus–Higher-Order Perl", p.146!

In the beginning, we read a FASTA input file containing the four sequences of the *pBluescript II* plasmid family. Then, we setup a series of tests to check the behavior of the function `read_fasta`.

```
my $seq_for = read_fasta($infile);
my $exp_seq_n = 4;
```

Hence, in the third test, we check whether the sequences indeed each have the correct length. This requires comparing the four **obtained** lengths (`@got_lens`) with four **expected** lengths (`@exp_lens`). To generate the former array, we use a `map` construct returning the length of every sequence. Within the block, `length` operates on the topic variable, which automatically iterates over sequences.

```
# test 3
my @exp_lens = (2961) x 4;
my @got_lens = map { length } values %$seq_for;
explain \@got_lens;
is_deeply \@got_lens, \@exp_lens,
    "correctly read the seqs in $infile";

# gives:

# [
#   2961,
#   2961,
#   2961,
#   2961
# ]

# The four plasmids all have the same length; this is not a bug!
```

Don't worry about the weird `is_deeply` and `cmp_bag` testing constructs; they are both explained just below (see "More on automated tests", p.151).

In the example above, the topic variable is only implied with `length`, whereas in the sixth test, it is explicitly used in an interpolated double-quoted string. The `map` block there serves the purpose of building the expected array of shortened sequence ids: `seq1`, `seq2`, `seq3`, `seq4`.

```
# test 6
my @exp_ids = map { "seq$_" } 1..$exp_seq_n;
```

```
explain \@exp_ids;
my @got_ids = keys %$id_for;

cmp_bag \@got_ids, \@exp_ids,
    "correctly remapped ids in temporary $tmpfile";

# gives:

# [
#   'seq1',
#   'seq2',
#   'seq3',
#   'seq4'
# ]
```

The seventh test is more comprehensive and uses three different `map` constructs. Its objective is to test the annotating functions of `Forem::FastaFile`, which require an annotation hash. To this end, it first makes a list of dummy annotations by converting each of the numbers `4` to `1` to the letters `D` to `A` using the function `chr` that returns the character represented by the number (ASCII or Unicode).

```
# test 7
my $tmp_seq_for = read_fasta($tmpfile);
my @tags = map { 'type' . chr(64 + $_) } reverse 1..$exp_seq_n;
explain \@tags;

# gives:

# [
#   'typeD',
#   'typeC',
#   'typeB',
#   'typeA'
# ]
```

Then, it builds the annotation hash itself by using a `map` block that returns a key/value pair for each incoming sequence id. Building a hash from a list of values is a very common use of `map`.

```
my $i = 0;
my $ann_for = { map { $_ => $tags[$i++] } keys %$tmp_seq_for };
explain $ann_for;

# gives:

# {
#   'seq1' => 'typeD',
#   'seq2' => 'typeC',
#   'seq3' => 'typeB',
#   'seq4' => 'typeA'
# }
```

Note also the use of a double sigil (`%$`) to dereference the hash references `$seq_for` and `$tmp_seq_for` in several of the code chunks shown in this section.

Finally, to check the function `prefix_ids`, it makes a list of the expected annotated ids using the annotation hash and compares them to the ones returned by the function.

```
my $ann_seq_for = prefix_ids($tmp_seq_for, $ann_for);

my @exp_ann_ids = map { $ann_for->{$_} . '-' . $_ } keys %$tmp_seq_for;
explain \@exp_ann_ids;
my @got_ann_ids = keys %$ann_seq_for;

is_deeply \@got_ann_ids, \@exp_ann_ids,
    "correctly prefixed ids using annotation hash";

# gives:

# [
#   'typeD-seq1',
#   'typeC-seq2',
#   'typeB-seq3',
#   'typeA-seq4'
# ]
```

BOX 12: `local` and the dynamic scope

The usage of `@_` in subs is pretty secure because Perl automatically **localizes** it for us when we call a function. This means that even if the default array looks like a global variable that is in principle visible from everywhere, it acts in practice as a lexically-scoped variable that can hold different values in different scopes.

This **dynamic scope** is more difficult to understand. Instead of looking outward in compile-time scopes, lookup traverses backwards through the calling context. While a global variable may be visible within all scopes, its actual value changes depending on localization and assignment.

You should generally not worry about it. Just remember that it is enabled with the `local` keyword (instead of `my`) and that it allows you to temporarily modify a global variable in a certain scope (i.e., code block) without propagating your changes outside (except to the subs called from that scope). This is especially useful with special (or **magic**) variables that govern Perl behavior, such as the list separator (`$"`) (see "Defining and concatenating strings") or the input record separator (`$/`) (see "Line endings"), both covered in the first part of this course.

Hence, before the introduction of the `slurp` method in `Path::Class`, Perl programmers used a weird idiom for slurping files. The idea was to undefine the input record separator, so that a single call to the readline operator (<>) would read the whole file in one shot. However, to avoid side-effects, it was necessary to localize the magic variable to the corresponding scope.

```
# adapted from xxl_xlate.pl
my $gc_content = file($gcfile)->slurp;
```

```
# the same using the old idiom
open my $gc_in, '<', $gcfile;
my $gc_content = do { local $/; <$gc_in> };

# even older idiom using global filehandles (please avoid)
open IN, "<$gcfile";
my $gc_content = do { local $/; <IN> };
```

And here's an example of fiddling with the list separator.

```
my @codons = qw(CAT GAA CTT CTT);
say '1. <' . join('> <', @codons) . '>';

{
    local $" = '> <';
    say "2. <@codons>";
}

# gives:
```

```
1. <CAT> <GAA> <CTT> <CTT>
2. <CAT> <GAA> <CTT> <CTT>
```

As `@_` in subs, the topic variable `$_` is automatically localized in *foreach-style* `for` loops without explicit iterator variables and in `grep`/`map` blocks. Even if it is possible to explicitly localize it using `local $_`, you should probably think twice if you really need it beyond these cases.

10.4 Bonus—Higher-Order Perl

In spite of its intimidating name, *Higher-Order Perl* is not the neo-fascist version of Modern Perl but the title of a famous book by Mark Jason Dominus. This classic work is a goldmine of (functional) programming wisdom. And the good news is that you can legally get it for free here:
https://hop.perl.plover.com/book/

Briefly said, "a higher-order function is a function that operates on other functions instead of on data values." We have already seen the builtin functions `map` and `grep`, whereas `annotate.pl` makes use of a funny `any { ... }` construct from `List::AllUtils` (see below).

All these three use functions as arguments (specified in the block delimited by the curly brace characters). Similarly, sort blocks are also arguments to a higher-order `sort` function. In contrast, some higher-order functions take data values as arguments and return custom built functions, such as `get_parser` from `Forem::BlastTable`, which creates an iterator function for a given BLAST report.

In this bonus section, I would like to scratch the surface of what it is possible to achieve using higher-order constructs. Take this as an appetizer for the true expressiveness of the Perl language. To this end, we will illustrate the purpose of three additional functions (of the `_by` category) taken from `List::AllUtils`: `count_by`, `partition_by` and `zip_by`.

Let's start with our favorite taxonomic example (again using the hash `%taxon_for` from p.140).

```
my @genera = qw(arabidopsis diatom human trypanosome yeast);

# count number of genera for each taxon
my %count_for = count_by { $taxon_for{$_} } @genera;
### %count_for

# collect all genera for each taxon
my %genera_for = partition_by { $taxon_for{$_} } @genera;
### %genera_for

# gives:

### %count_for: {
###               Euglenozoa => 1,
###               Opisthokonta => 2,
###               Plantae => 1,
###               Stramenopiles => 1
###             }

### %genera_for: {
###                Euglenozoa => [
###                                 'trypanosome'
###                               ],
###                Opisthokonta => [
###                                   'human',
###                                   'yeast'
###                                 ],
###                Plantae => [
###                              'arabidopsis'
###                            ],
###                Stramenopiles => [
###                                    'diatom'
###                                  ]
###              }
```

For the record, here are the explicit versions of the equivalent code.

```
# count number of genera for each taxon
my %count_for;
for my $genus (@genera) {
    $count_for{ $taxon_for{$genus} }++;
}

# collect all genera for each taxon
my %genera_for;
for my $genus (@genera) {
    push @{ $genera_for{ $taxon_for{$genus} } }, $genus;
}
```

Now, getting back to our own *Holy Grail* (see the first part of this course), here is a statement to get a reverse hash of the genetic code (see next page for a concise way of building the hash `%aa_for` itself).

```
my %rev_aa_for = partition_by { $aa_for{$_} } keys %aa_for;
### %rev_aa_for

# gives:

### %rev_aa_for: {
###                '*' => [
###                         'TGA',
###                         'TAA',
###                         'TAG'
###                       ],
###                A => [
###                       'GCC',
###                       'GCG',
###                       'GCT',
###                       'GCA'
###                     ],
###                C => [
###                       'TGT',
###                       'TGC'
###                     ],
###                D => [
###                       'GAC',
###                       'GAT'
###                     ],
...
###                V => [
###                       'GTC',
###                       'GTG',
###                       'GTA',
###                       'GTT'
###                     ],
###                W => [
###                       'TGG'
###                     ],
###                Y => [
###                       'TAC',
###                       'TAT'
###                     ]
###              }
```

This is equivalent to the following implicit code snippet.

```
my %rev_aa_for;
push @{ $rev_aa_for{ $aa_for{$_} } }, $_ for keys %aa_for;
```

Finally, I give you below a very terse version of the original `translate.pl` script. I do not pretend that it is more legible, though! Note how the combination of the `split` and `zip_by` functions is used to emulate the original *foreach-style* `for` loop with its explicit `substr` calls.

```
#!/usr/bin/env perl

use Modern::Perl '2011';
use List::AllUtils qw(zip_by);

# build hash for standard code (definition taken from NCBI gc.prt file)
local $" = q{};     # ensure no whitespace when joining lists
my %aa_for =
    zip_by { "@_[1..3]" => $_[0] }      # codon => aminoacid
        map { [ split // ] } qw(
    FFLLSSSSYY**CC*WLLLLPPPPHHQQRRRRIIIMTTTTNNKKSSRRVVVVAAAADDEEGGGG
    TTTTTTTTTTTTTTTTCCCCCCCCCCCCCCCCAAAAAAAAAAAAAAAAGGGGGGGGGGGGGGGG
    TTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGG
    TCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAG
);  # split these lines into anon arrays of letters and mesh them together

# read a DNA string from STDIN (using shift)
# split it into codons (3-letter words) with a regex
# translate uppercased codons into aminoacids (with uc() meaning uc $_)
# output protein sequence (scalar context enforcing implicit join)
say map { $aa_for{uc()} // 'X' } shift =~ m/(\w{3})/xmsg;
```

Did you enjoy the single last statement performing all the work?

Chapter 11

Dissecting our annotation app

11.1 More on automated tests

The files `blast_table.t` and `fasta_file.t` illustrate a number of testing functions offered by the modules `Test::Most` and `Test::Files`.

The basic structure of a testing statement is (nearly) always the same.

```
testing_function <got>, <expected>, 'testing message';
```

In the code above, `got` and `expected` are placeholders, as indicated by the angle bracket characters surrounding them. This structure is for testing containers (also known as **aggregates**), such as arrays, hashes and output files.

The `is_deeply` function expects two references to two different (possibly nested) data structures, the first one corresponding to the output of the tested function and the second one to the expected result. It can be applied to arbitrarily complex data structures, from the simple array (or hash) to deeply nested (multi-level) data structures. If the two structures differ, it marks the test as failed and prints the first difference found.

```
is_deeply \@got, $exp_ref,
    'correctly parsed the 10 first lines of the report';

is_deeply \@got_lens, \@exp_lens,
    "correctly read the seqs in $infile";

is_deeply \@got_ann_ids, \@exp_ann_ids,
    "correctly prefixed ids using annotation hash";
```

The `cmp_bag` function compares two arrays where the elements are supposedly identical, but not necessarily in the same order. This allows for a bit of sloppiness when building arrays of expected values.

```
cmp_bag \@got_ids, \@exp_ids,
    "correctly remapped ids in temporary $tmpfile";
```

The `compare_ok` function fully automatizes the verification of output files. Simply pass it the names of the file produced by the test and of a file containing the expected output and it will do the rest. If the files differ, it shows you their differences in a very legible way.

```
compare_ok $outfile, $infile,
    "correctly written $outfile";

compare_ok $tmpfile, $expfile,
    "correctly written temporary $tmpfile";
```

Finally, we also sometimes compare scalar values using the `cmp_ok` function. Its structure is slightly longer than the previous testing functions in that it further requires a string specifying the exact comparison operator to use for the test.

```
cmp_ok keys %$seq_for, '==', $exp_seq_n,
    "correctly read $exp_seq_n seqs from $infile";

my $exp_ids = qx{grep \\> test/infile.fasta | cut -c2-};
...
cmp_ok $got_ids, 'eq', $exp_ids,
    "correctly parsed the ids in $infile";
```

In this last example, `$exp_ids` is derived from a system call using the executing quoting operator (`qx{}`) introduced in "Writing portable code" (see the first part of this course). Such an approach ensures that the content of `$exp_ids` is always in sync with the actual sequence ids in `infile.fasta`, independently of the changes operated by the NCBI.

There exist many other testing functions. As usual, you can read about them using `perldoc`.

```
$ perldoc Test::Most
```

11.2 Interesting bits in `Forem::FastaFile`

The module `Forem::FastaFile.pm` includes a few novelties that are worth briefly mentioning. These pertain to constants, temporary files, `for` loops, `subs` and references.

First, the module `Const::Fast` allows us to define **constants**. Constants are variables that cannot be modified. By convention, they are written in uppercase letters. They are especially useful to show the reader that some variable has a fixed value that will persist through the whole file, such as magic numbers (see "`length`", in the first part of this course).

```
use Const::Fast;

# and later...
const my $CHUNK_LEN => 70;
```

The constant `$CHUNK_LEN` governs the width of the lines in our output FASTA files. The function writing the temporary FASTA files is interesting for two reasons:

1. It uses the module `File::Temp` to open output files. Using this module spares you the need to devise unique names for your **temporary files**, which can be very tricky when running multiple

instances of a program concurrently. The name can be obtained using the `filename` method. Temporary files can be automatically deleted if asked to do so (`UNLINK => 1`).

```
my $out = File::Temp->new(UNLINK => 0, EXLOCK => 0, SUFFIX => '.fasta');

# and later...
return ($out->filename, \%id_for);
```

2. It uses a funny *C-style* `for` loop to both abbreviate the sequence ids and to build the hash containing the abbr-id / full-id pairs. This loop has an iterator variable that is initialized and incremented as usual, but the conditional comparison does not use it.

```
my %id_for;
for (my $i = 1; my ($seq_id, $seq) = each %$seq_for; $i++) {
    my $tmp_id = "seq$i";
    say {$out} ">$tmp_id\n" . _wrap_seq($seq);
    $id_for{$tmp_id} = $seq_id;
}
```

The code of our module further defines a function with a name beginning with an underscore character (_). This is the naming convention for **private functions** that are not meant to be exported. They are also devoid of POD, which totally makes sense.

```
sub _wrap_seq {
    my $seq = shift;

    my $seq_len = length $seq;

    # wrap seq in chunks of length $CHUNK_LEN
    my $wrapped_seq;
    for (my $i = 0; $i < $seq_len; $i += $CHUNK_LEN) {
        $wrapped_seq .= substr($seq, $i, $CHUNK_LEN) . "\n";
    }

    return $wrapped_seq;
}
```

Finally, all the functions provided by our module expect and return hash references instead of regular hashes. This has three advantages:

1. It encapsulates the use of `Tie::IxHash` within our module. For example, `read_fasta` now returns a reference to the ordered hash `%seq_for`. As long as the caller dereferences it to use the hash without copying the hash into a new one, the order of the sequences is preserved.

```
sub read_fasta {
    my $infile = shift;

    # ...

    tie my %seq_for, 'Tie::IxHash';             # preserve original seq order

    # ...
    return \%seq_for;
}
```

In contrast, in `cutter.pl`, we had to explicitly copy the tied hash to a new tied hash to avoid destroying sequence order. This kind of subtle code dependencies between parts of a program is known as **coupling**, an undesirable property. Functions and modules help reducing coupling.

```
# from cutter.pl
tie my %seq_for, 'Tie::IxHash';                # preserve original seq order
%seq_for = read_fasta( $ARGV{'<infile>'} );     # but at the expense of coupling

# from a buggy cutter.pl
my %seq_for = read_fasta( $ARGV{'<infile>'} );  # loose original seq order
```

2. It is much faster to pass and return references to container variables than the container variables themselves. This is obvious because a reference is only a scalar (number), whereas the corresponding data structure (first level) can be composed of thousands (or millions) of elements.

3. It allows us to invoke functions that require multiple containers without flattening them in a single list. This is the case of the function `prefix_ids` that expects the sequence hash and the annotation hash. Observe how we copy the original hash to an ordered hash to preserve sequence order. This encapsulating strategy could be pushed farther by using **objects**.

```
sub prefix_ids {
    my $seq_for = shift;
    my $ann_for = shift;

    tie my %new_hash, 'Tie::IxHash';

    while (my ($seq_id, $seq) = each %$seq_for) {
        my $prefix = $ann_for->{$seq_id};
        $new_hash{ $prefix ? $prefix . '-' . $seq_id : $seq_id } = $seq;
    }

    return \%new_hash;
}
```

11.3 A mighty modular script: `annotate.pl`

The main annotation script packs a high amount of functionality while staying relatively short thanks to its use of our modules `Blast::Table` and `Fasta::File` and of some other advanced techniques. Let's review them here by examining our listing in detail.

11.3.1 Conciseness, readability and maintainability by code reuse

`annotate.pl` begins with a bunch of `use` statements. This is not uncommon for a script written in *Modern Perl* and is evidence for good programming practice.

```
#!/usr/bin/env perl

use Modern::Perl '2011';
use autodie;

use Getopt::Euclid;
use Smart::Comments '###';
use Template;

use File::Basename;
use Path::Class 'file';

use IPC::System::Simple qw(system);
use List::AllUtils 'any';

use Forem::FastaFile qw(:io prefix_ids);
use Forem::BlastTable 'get_parser';
```

Although you already know most of these modules, two new ones are worth of interest: `Template` and `IPC::System::Simple`. We will cover them in a few moments.

Meanwhile, take a look at the two function calls below. Their role is to process the two FASTA files, i.e., the file with the sequences to be annotated and the file with the reference sequences that will provide the annotation. Using the single same function for the two files requires a bit of thought ahead of time, but eventually results in a more concise, more readable and more maintainable program.

```
#### Arguments: %ARGV

### Reading infile: $ARGV{'<infile>'}
my ($seq_for, $intype, $infile,  $inid_for) = process_file($ARGV{'<infile>'});

### Reading infile: $ARGV{'--ref-file'}
my (  undef, $reftype, $refile, $refid_for) = process_file($ARGV{'--ref-file'});
```

Both function calls invokes the same function `process_file`, defined at the end of our script. This sub is a **wrapper function**. This means that it does its job by calling another function (or a few other functions) while adding little new functionality on its own.

```
sub process_file {
    my $infile = shift;

    my $seq_for = read_fasta($infile);
    my $type = ( any { m/[EFILPQ]/i } values %$seq_for ) ? 'prot' : 'nucl';
    my ($tmpfile, $id_for) = write_tmp_fasta(
        map { $_ => $seq_for->{$_} =~ tr/-//dr } keys %$seq_for
    );  # degap seqs on the fly

    return ($seq_for, $type, $tmpfile, $id_for);
}
```

Here, the purpose of `process_file` is:

1. to read a FASTA file,
2. to determine whether it contains DNA sequences or protein sequences,
3. to rewrite the sequences without gaps and with shorter ids suitable for BLAST use.

Gap removal is performed on the fly through a `map` block building degapped copies of the sequences through transliteration. As this dynamically built anonymous hash has the same structure as the original hash referenced by `$seq_for`, it fits the expectations of the `write_tmp_fasta` function.

The returned list packs the following information:

- `$seq_for` — the ordered hash reference containing the sequences read from the file,
- `$type` — a string giving the type of the sequence, either `'nucl'` or `'prot'`,
- `$tmpfile` — the name of the temporary file (FASTA) to be passed to BLAST,
- `$id_for` — the unordered hash reference associating short and long ids.

Sequence type is needed both to build the BLAST database and to invoke the correct flavor of the BLAST algorithm. The type is determined based on the occurrence of at least one letter that can only be interpreted as an amino acid (`EFILPQ`) in any of the sequences (see "Character classes", in the first part of this course). Remember, other non-`ACGT` letters design valid degenerated nucleotides (e.g., `W` is `A` or `T`). Let's now have a look at how to select the right BLAST flavor.

```
# determine blast program based on type combination
my %pgm_for = (
    'prot:prot' =>  'blastp',
    'nucl:prot' =>  'blastx',
    'prot:nucl' => 'tblastn',
    'nucl:nucl' => 'tblastx',
);
my $pgm = $pgm_for{ "$intype:$reftype" };
```

The trick is to setup a hash as a switch table (see "Hash uses", in the first part of this course). The two sequence types (one from the input file and one from the reference file) are concatenated and the resulting string used as a key to look for the corresponding BLAST flavor. This is extremely concise and less error-prone than a cascade of `elsif` blocks.

11.3.2 Building programs with programs

We now have all the information we need to invoke BLAST, which requires devising a shell command with the required arguments and submitting it to the system. To this end, we use a very famous piece of Perl code, the *Template Toolkit* (*TT2*).

Loaded with the statement `use Template`, the *Template Toolkit* is a (not so) mini-language oriented towards the programmatic completion of generic text files. It is so powerful that a complete book is needed to cover all its features. You can find more about it here:

- http://www.template-toolkit.org/
- http://www.oreilly.com/catalog/perltt/

In this course, we only scratch the surface... The idea is to first define a string (`$template`) containing placeholders in lieu of the variable command-line arguments. Here, placeholders are barewords between a pair of special delimiters `[%` and `%]`. You will recognize the use of the *heredoc* syntax (see "Interlude…", in the first part of this course).

```
# define command template
my $report = "$infile.blast.fmt7";
my $template = <<'EOT';
makeblastdb -in [% refile %] -dbtype [% reftype %]
[% pgm %] -query [% infile %] -db [% refile %] -evalue [% E %] \
    -outfmt 7 -out [% report %]
EOT
```

BOX 13: The `tpage` command-line tool

It is even possible to take advantage of templating outside any Perl program thanks to the `tpage` command-line tool. Here's a real-word example allowing me to easily submit jobs to my grid computer for the assembly of next-generation sequencing data.

First, I need a template file (ending in `.tt` by convention).

```
#$ -S /bin/bash
#$ -V
#$ -cwd
#$ -q [% queue %]
#$ -m beas
#$ -M denis.baurain@uliege.be
Trinity.pl --seqType fq --left [% left %] --right [% right %] \
    --JM [% memory %] --CPU [% threads %]
```

Then, to build the actual script file, I call the `tpage` tool and use its `--define` option to specify the text chunks corresponding to each placeholder.

```
$ tpage --define queue=bignode.q --define memory=200G --define threads=24 \
    --define left="SRR941229_1.fastq SRR941232_1.fastq" \
    --define right="SRR941229_2.fastq SRR941232_2.fastq" \
    ../trinity_pr.tt > trinity.job
```

Here's the content of `trinity.job` after the call to `tpage`. Just imagine how this becomes handy within one or more shell loops to generate a series of related jobs.

```
#$ -S /bin/bash
#$ -V
#$ -cwd
#$ -q bignode.q
#$ -m beas
#$ -M denis.baurain@uliege.be
Trinity.pl --seqType fq --left SRR941229_1.fastq SRR941232_1.fastq \
    --right SRR941229_2.fastq SRR941232_2.fastq --JM 200G --CPU 24
```

Finally, I submit the job using the standard `qsub` command.

```
$ qsub -pe snode 24 trinity.job
```

Then, one has to setup a hash (`%vars`) where each key matches a placeholder while its associated value gives the text chunk to be substituted for the placeholder.

```
# build command
my %vars = (
    refile   => $refile,
    reftype  => $reftype,
    pgm      => $pgm,
    infile   => $infile,
    E        => $ARGV{'--evalue'},
    report   => $report,
);
```

Finally, the string containing the completed command line (`$command`) is built by invoking the `process` method on a fresh object of the `Template` **class**. Again, this is object-oriented programming.

```
my $command;
my $tt = Template->new;
$tt->process(\$template, \%vars, \$command);
#### $command
```

Templating is a very efficient strategy, especially because templates can be stored in external files that users may edit to suit their needs. Therefore, as soon as you find yourself concatenating more than a few variables intermingled with fixed strings, you should consider templating instead.

11.3.3 Calling external programs

Once the command built, we submit it to the system. We have already discussed the executing quoting operator (`qx{}`), which is exactly equivalent to the famous **backtick delimiters** (` and `) that you encounter on websites devoted to Perl. Here, however, we use the `system` builtin function instead.

```
### Performing BLAST...
system($command);
```

`system` is always available in Perl, but the version we use here is actually provided by the CPAN module `IPC::System::Simple`. This module simplifies the process of capturing the standard output and error streams and helps us to deal with potential errors occurring when calling external programs.

These issues are quite advanced topics, so I do not cover them in detail here (use `perldoc` for more information). As an example, here's a `system` call dealing with a missing external program (CAP3).

```
# create CAP3 command
my $cmd = "cap3 $infile > $outfile 2> /dev/null";

# try to robustly execute CAP3
my $ret_code = system( [ 0, 127 ], $cmd);
if ($ret_code == 127) {
    carp 'Cannot execute cap3 command; returning without contigs!';
    return;
}
```

11.3.4 Wrapping it up

At this stage in our course, the remaining of `annotate.pl` should be quite easy to understand. So, I only give you a few indications on what is going on.

First, we setup a parser for our BLAST report.

```
### Parsing BLAST report...
my $parser = get_parser($report);
my %ann_for;
my $curr_id = q{};
```

Then, we loop through the HSPs and extract the annotation bit from each first hit above the user-specified E-value threshold. This allows us to build an annotation hash where our input sequences are associated to reference annotations. Observe how we convert on the fly short ids to long ids using the `$refid_for` and `$inid_for` hash references returned by the `process_file` function.

```
HSP:
while ( my $hsp_ref = $parser->() ) {
    my ($qid, $hid, $evalue) = @{ $hsp_ref }{ qw(query_id hit_id evalue) };

    next HSP if $evalue > $ARGV{'--evalue'};        # skip weak hits
    next HSP if $qid eq $curr_id;                   # skip non-first hits
    $curr_id = $qid;

    # capture annotation bit in ref seq id using regex
    my ($annotation) = $refid_for->{$hid} =~ $ARGV{'--ref-regex'};
    $ann_for{ $inid_for->{$qid} } = $annotation;
}
#### Annotations: %ann_for
```

Finally, we print to the screen a table with the annotations. We also optionally write a new version of the input file in which sequence ids are each prefixed by the annotation bit extracted from the id of the reference sequence matching them the best. This is achieved by calling our `prefix_ids` function.

```
say '# ' . join "\t", qw(tag id);
for my $id (sort keys %ann_for) {
    say join "\t", $ann_for{$id}, $id;
}

if ($ARGV{'--write-ann-file'}) {
    my ($basename, $dir, $ext) = fileparse($ARGV{'<infile>'}, qr{\.[^.]*}xms);
    my $outfile = file($dir, $basename . '_ann' . $ext);
    ### Writing annotated file: $outfile->stringify
    write_fasta( $outfile, prefix_ids($seq_for, \%ann_for) );
}
```

Homework

To finish this course, here's a last assignment. It consists in developing a new module that exports two functions (`Forem::Translate`). Do not forget to program some tests!

`get_genetic_code` Expects a file path to the NCBI `gc.prt` file and an integer number giving the id of the requested genetic code. Returns a **reference** to the hash `%aa_for` (see `xxl_xlate.pl`).

`translate` Expects a string containing a DNA sequence to translate, an integer giving the reading frame (`1`, `2`, `3`, `-1`, `-2`, `-3`) and a **reference** to the genetic code (`%aa_for`) to use. Returns a string with the protein.

Then, re-write `xxl_xlate_subs.pl` (`hw10_xxl_xlate_mod.pl`) to use the modules `Forem::FastaFile` and `Forem::Translate`. Within it, always call the `translate` function with the reading frame `1`. Add a `Getopt::Euclid` interface providing default values for a `--remote` location of the `gc.prt` file and a genetic code id of `1`.

I wish you a lot of fun with Perl!

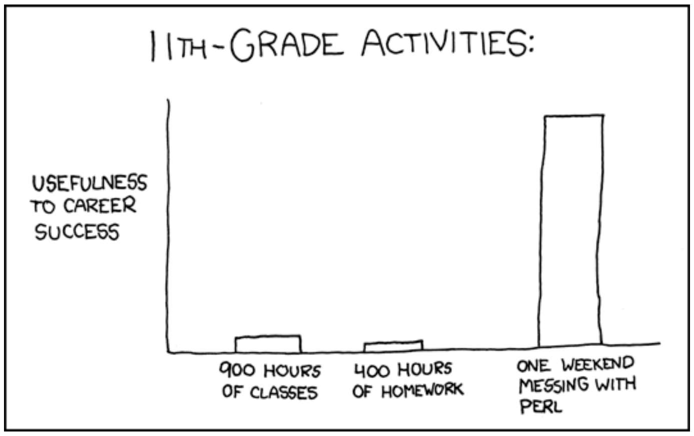

Figure 11.1: 11th-grade activities [xkcd.com]

Index