Modern Perl for Biologists I | The Basics

M-BIM / INBTBI

Denis BAURAIN / ULiège

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 1

Chapter 1

Introduction

1.1 What is Perl?

From the Perl documentation:
`http://perldoc.perl.org/`

Perl is a high-level programming language with an eclectic heritage written by Larry Wall and a cast of thousands. It derives from the ubiquitous C programming language and to a lesser extent from sed, awk, the Unix shell, and at least a dozen other tools and languages. Perl's process, file, and text manipulation facilities make it particularly well-suited for tasks involving quick prototyping, system utilities, software tools, system management tasks, database access, graphical programming, networking, and web programming. These strengths make it especially popular with system administrators and web developers, but mathematicians, geneticists, journalists, and even managers also use Perl. Maybe you should, too.

Interestingly, geneticists are mentioned in this very small excerpt. This looks promising.

1.2 What is Modern Perl?

From the Preface of *Modern Perl* (2014) by chromatic:
`http://modernperlbooks.com/books/modern_perl_2014/`

Modern Perl is one way to describe the way the world's most effective Perl 5 programmers work. They use language idioms. They take advantage of the CPAN. They show good taste and craft to write powerful, maintainable, scalable, concise, and effective code. You can learn these skills too!

Perl first appeared in 1987 as a simple tool for system administration. Though it began by declaring and occupying a comfortable niche between shell scripting and C programming, it has become a powerful, general-purpose language family. Perl 5 has a solid history of pragmatism and a bright future of polish and enhancement.

Over Perl's long history—especially the 19 years of Perl 5 [now 26 years]—our understanding of what makes great Perl programs has changed. While you can write productive programs which never take advantage of all the language has to offer, the global Perl community has invented, borrowed, enhanced, and polished ideas and made them available to anyone willing to learn them.

1.3 Why for biologists?

A number of bioinformaticians are Perl programmers, especially older ones. Even if Perl is less popular than it was a decade ago, I argue that it is still very relevant for biologists. My goal here is to teach you how to get the most of its modern incarnation using examples from the biological field.

In that respect, this course can be seen as a hybrid between *Learning Perl* by Randal L. Schwartz et al. (O'Reilly Media), *Beginning…* and *Mastering Perl for Bioinformatics* by James D. Tisdall (O'Reilly Media), and *Modern Perl* by chromatic (Onyx Neon Inc). However, all its content is original and peppered with words of wisdom and good programming practices taken from my own 30-year long experience in computer programming.

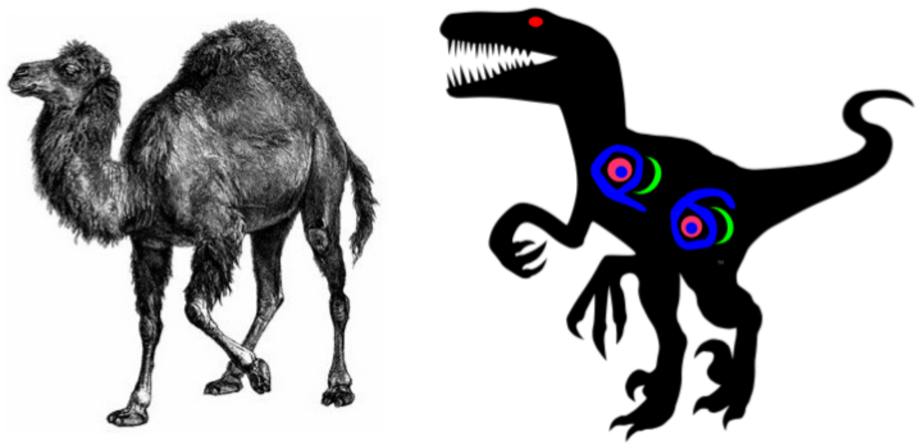

Figure 1.1: Perl 5 *vs.* Modern Perl

1.4 Covered topics

This course was originally meant to be introductory but ended up being quite advanced. The only important area of *Modern Perl* that will not be discussed here is **Object-Oriented Programming**. Below is a rough and incomplete overview (see "Contents", p.viii, for details).

1. The Basics
    1. Variables (Scalars, Arrays, Hashes)
    2. Operators, Boolean expressions and Control flow
    3. Input/output
    4. Regular expressions
    5. One-liners
2. Deeper Concepts
    6. Functions
    7. References and Nested data structures
    8. Modules and Unit tests
    9. Best of CPAN
    10. Idiomatic Perl – **TIMTOWTDI**

1.5 Homework guidelines

The aim of homework assignments in this course is to get you started in Perl as fast as possible. Instead of small code snippets (as in other textbooks), we study complete programs that you have to improve more or less radically.

You'd better see these assignments as *oral expression* exercises in a foreign language. You cannot learn English just by attending classes about vocabulary and grammar. To make any progress, you must try to speak in a real setting, e.g., to ask your way in New York. Therefore, I don't expect you to use words and constructs not yet introduced in class.

Of course, you are free to use new concepts if you want to, but this is not strictly required. If you need help, look at the websites below. Most are excellent resources for learning Perl, and especially its modern incarnation.

- `http://perldoc.perl.org/`
- `http://www.cpan.org/`
- `http://perlmonks.org/`
- `http://stackoverflow.com/questions/tagged/perl`
- `http://perl.com/`
- `http://blogs.perl.org/`

Don't forget the *Modern Perl* website by chromatic, which, written as a reference book, is the perfect complement to the present tutorial. The last (fourth) edition can be found here:

- `http://modernperlbooks.com/books/modern_perl_2016/`

1.6 Perl Cheat Sheet

If you feel you need a very compact overview of Perl's syntax, keep a copy of the following **cheat sheet**. This is the official one but other sheets, some much more detailed, are freely available on the web. You can generate this one at will using the command below:

```
$ perldoc perlcheat > perlcheat.txt
```

```
CONTEXTS  SIGILS  ref        ARRAYS        HASHES
void      $scalar SCALAR     @array        %hash
scalar    @array  ARRAY      @array[0, 2]  @hash{'a', 'b'}
list      %hash   HASH       $array[0]     $hash{'a'}
          &sub    CODE
          *glob   GLOB       SCALAR VALUES
                  FORMAT     number, string, ref, glob, undef
REFERENCES
\      reference       $$foo[1]       aka $foo->[1]
$@%&*  dereference     $$foo{bar}     aka $foo->{bar}
[]     anon. arrayref  ${$$foo[1]}[2] aka $foo->[1]->[2]
{}     anon. hashref   ${$$foo[1]}[2] aka $foo->[1][2]
\()    list of refs
                       SYNTAX
OPERATOR PRECEDENCE    foreach (LIST) { }     for (a;b;c) { }
->                     while   (e) { }        until (e)   { }
++ --                  if      (e) { } elsif (e) { } else { }
**                     unless  (e) { } elsif (e) { } else { }
! ~ \ u+ u-            given   (e) { when (e) {} default {} }
=~ !~
* / % x                 NUMBERS vs STRINGS  FALSE vs TRUE
+ - .                   =          =        undef, "", 0, "0"
<< >>                   +          .        anything else
named uops              == !=      eq ne
< > <= >= lt gt le ge   < > <= >=  lt gt le ge
== != <=> eq ne cmp ~~  <=>        cmp
&
| ^             REGEX MODIFIERS       REGEX METACHARS
&&              /i case insensitive   ^      string begin
|| //           /m line based ^$      $      str end (bfr \n)
.. ...          /s . includes \n      +      one or more
?:              /x /xx ign. wh.space  *      zero or more
= += last goto  /p preserve           ?      zero or one
, =>            /a ASCII    /aa safe  {3,7}  repeat in range
list ops        /l locale   /d  dual  |      alternation
not             /u Unicode            []     character class
and             /e evaluate /ee rpts  \b     boundary
or xor          /g global             \z     string end
                /o compile pat once   ()     capture
```

```
DEBUG                                   (?:p)  no capture
-MO=Deparse     REGEX CHARCLASSES       (?#t)  comment
-MO=Terse       .   [^\n]               (?=p)  ZW pos ahead
-D##            \s  whitespace          (?!p)  ZW neg ahead
-d:Trace        \w  word chars          (?<=p) ZW pos behind \K
                \d  digits              (?<!p) ZW neg behind
CONFIGURATION   \pP named property      (?>p)  no backtrack
perl -V:ivsize  \h  horiz.wh.space      (?|p|p)branch reset
                \R  linebreak           (?<n>p)named capture
                \S \W \D \H negate      \g{n}  ref to named cap
                                        \K     keep left part
FUNCTION RETURN LISTS
stat      localtime    caller           SPECIAL VARIABLES
 0 dev    0 second      0 package       $_    default variable
 1 ino    1 minute      1 filename      $0    program name
 2 mode   2 hour        2 line          $/    input separator
 3 nlink  3 day         3 subroutine    $\    output separator
 4 uid    4 month-1     4 hasargs       $|    autoflush
 5 gid    5 year-1900   5 wantarray     $!    sys/libcall error
 6 rdev   6 weekday     6 evaltext      $@    eval error
 7 size   7 yearday     7 is_require    $$    process ID
 8 atime  8 is_dst      8 hints         $.    line number
 9 mtime                9 bitmask       @ARGV command line args
10 ctime               10 hinthash      @INC  include paths
11 blksz               3..10 only       @_    subroutine args
12 blcks               with EXPR        %ENV  environment
```

Chapter 2

Before beginning

2.1 The need for a sandbox

Perl is part of all UNIX-like operating systems (including Linux and macOS), in which it fulfills important functions. This has two consequences:

1. The installed version of Perl is often out of date.
2. The number of directly available Perl modules is modest.

This can be changed (using `sudo`), but it is better not to mess with the system's Perl.

To explore *Modern Perl* at no risk, we setup a specialized infrastructure allowing us to install multiple versions of Perl without ever touching the system's Perl.

This infrastructure is known as **Perlbrew** and is available at: `http://perlbrew.pl/`

2.2 How to install our sandbox?

Open a fresh terminal and type in the following commands. Note that the lines to enter always begin with the **command prompt character** (`$`). However, you *must not* type in `$` itself; it symbolizes the **shell** waiting for user commands. Moreover, the lines beginning with the **comment character** (`#`) are only there for your information. Thus, you don't need to (but you can) enter them at the shell.

```
# install developer tools (if needed)
$ sudo apt install build-essential            # Linux
$ xcode-select --install                      # macOS

# download the Perlbrew installer
$ wget -O - http://install.perlbrew.pl | bash   # Linux
$ curl -L http://install.perlbrew.pl | bash     # macOS

# initialize Perlbrew
$ source ~/perl5/perlbrew/etc/bashrc
$ perlbrew init
```

```
# install a recent stable version of the perl interpreter
# this will take a while...
$ perlbrew available
$ perlbrew install perl-5.32.0 --thread
$ perlbrew install-cpanm

# enable the installed version
$ perlbrew list
$ perlbrew switch perl-5.32.0
```

To make the `perlbrew` command (and your new `perl` **interpreter**) always accessible, you need to amend your shell configuration file. First, check the name of your current shell. Enter your password then simultaneously press the keys `Ctrl` and `C` (abbreviated as `^C` and known as **Break**) to abort.

```
$ chsh
```

Using a **text editor** (e.g., `gedit` on Linux), add the following line either to your `~/.bashrc` file (if using the `/bin/bash` shell) or to your `~/.zshrc` file (if using the `/bin/zsh` shell). On macOS, add it instead to your `~/.profile` file. The line itself is the same in all three cases.

```
source ~/perl5/perlbrew/etc/bashrc
```

To check that everything is working as expected, close your terminal, open a fresh one and ask for the current `perl` version.

```
$ perl -v
```

You should get a **version number** matching the `perl` install to which you have switched above. If so, you are ready! Otherwise, log out from your session, reopen a terminal and retry only this very last step. If it still does not work, something has gone wrong and you need to reinstall Perlbrew.

Chapter 3

First steps in Perl

3.1 Motivation

- Learning a new technology (here, Perl) can be tedious.
- We learn better when we are motivated.
- One way to get motivated is to be shown a **killer app**.

Figure 3.1: The *Killer App* [Rosa Gago, 2016]

3.2 Our first killer app

Molecular biologists often manipulate *DNA sequences*. How many times did you try to *reverse complement* a DNA sequence in your head? For example, when designing PCR primers or analyzing sequence chromatograms... Of course, computers can help.

Let's pretend we want to reverse-complement the following DNA sequence:
`CATGAACTTCTTTGGCGTCTTGAT`.

We can submit it to a web application:
`http://www.bioinformatics.org/sms/rev_comp.html`

Yes, it works, but it is *tedious*.

3.3 How to make our own `rev_comp`?

To write programs, we need a developer-oriented text editor. By this, I mean a graphical application allowing us to easily type in and modify computer code. If you use Linux, `gedit` is available by default. However, I advise you to install `geany`, which is much more powerful while not too bloated.

```
$ sudo apt install geany
```

If you are on macOS, a convenient solution is `BBEdit`, which is not free software but can still be downloaded at no cost from the website of the Bare Bones company:
http://www.barebones.com/products/bbedit/

Once you are done with installing a suitable text editor, follow the steps below.

1. Open the text editor and type in the short Perl program shown below. Do your best to respect the **code indentation** (i.e., the vertical alignment of the code). Line numbers are only there to help you. Consequently, they must not be entered.

```
use strict;
use warnings;

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

my $dna_string = shift;
my @bases = split //, $dna_string;
my $len = @bases;

my @comp_bases;
for my $base (@bases) {
    my $comp_base = $comp_for{$base};
    unshift @comp_bases, $comp_base;
}
my $rev_comp_dna_string = join q{}, @comp_bases;

print 'fwd: ' . $dna_string . "\n";
print 'rev: ' . $rev_comp_dna_string . "\n";
```

2. Save it as `rev_comp.pl` (in a new `mod_perl` subdirectory). You should do this right after beginning typing in order to enable your editor's **syntax highlighting** feature (based on the **file suffix**, here `.pl`).

3. Open a fresh terminal.

4. Go to the `mod_perl` subdirectory containing your program.

```
$ cd mod_perl
```

5. Try it with the following command.

```
$ perl rev_comp.pl CATGAACTTCTTTGGCGTCTTGAT
```

(Drum roll…) *Tada!*

```
fwd: CATGAACTTCTTTGGCGTCTTGAT
rev: ATCAAGACGCCAAAGAAGTTCATG
```

BOX 1: What if it does not work?

If `rev_comp.pl` is your very first program, chances are that it will fail to run on the first attempt. Don't become discouraged and keep the following check-list in mind to **debug** your code.

1. Always look for the very first **syntax error**. If there is one, all subsequent errors are likely to be mere consequences of this original sin. To determine where the first syntax error is, consider the line numbers given in the error messages produced by the Perl interpreter.

2. If there is no syntax error, look for other kinds of errors and fix them each one in turn, starting from the first one. Frequently retry to run your program as later errors may disappear due to the debugging of earlier ones.

3. If you really don't see where the error comes from, look at the line preceding the first error. Did you forget the semicolon character (`;`)? Perl statements must always end with such a character. Other weird errors can come from a missing `use` directive. This is unlikely in `rev_compl.pl` but will become relevant once we deal with Perl modules.

4. Finally, pay attention to spelling (e.g., `$dna_string` is not `$dan_string` nor `$dnastring`) and punctuation characters (e.g., `;` is not `,`). In that respect, the **autocompletion** feature of your text editor can be helpful (i.e., `Ctrl-Space` in `geany`). Moreover, don't forget that most computer languages (including Perl) are **case-sensitive**, which means that lower and upper case letters are not interchangeable (e.g., `use` is not `USE`). In short, the Perl interpreter is very fussy and you will quickly learn to become as persnickety as it!

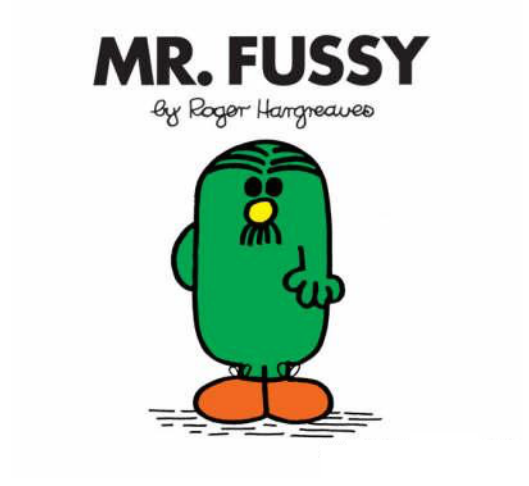

Figure 3.2: Mr. Fussy [Roger Hangreaves, 1976]

3.4 How to document our program?

Let's add a few comments to explain our code… As with the shell, comments begin with the **comment character** (#). You should not paraphrase your code, but instead explain the purpose of each logical group of **statements**. A statement is like a sentence for the computer. It does not begin with a capital letter, but it must end with a **semicolon character** (;).

```perl
# enforce good programming practices
use strict;
use warnings;

# define a hash mapping the base complements
my %comp_for = (
    A => 'T',   T => 'A',   G => 'C',   C => 'G',
    a => 't',   t => 'a',   g => 'c',   c => 'g',
);

# read a DNA string, split it into bases and compute its length
my $dna_string = shift;
my @bases = split //, $dna_string;
my $len = @bases;

# take the complement of each base in turn
# elongate reverse string by inserting complemented bases at its beginning
my @comp_bases;
for my $base (@bases) {
    my $comp_base = $comp_for{$base};
    unshift @comp_bases, $comp_base;
}
my $rev_comp_dna_string = join q{}, @comp_bases;

# print forward and reverse complemented DNA strings
print 'fwd: ' . $dna_string . "\n";
print 'rev: ' . $rev_comp_dna_string . "\n";
```

Within a statement, carefully choosing variable names and other identifiers can also help other people to understand your code. This practice is called **autodocumentation**. Observe how I used descriptive identifiers in our first program (e.g., `$dna_string`, `@bases`, `$len`).

This program is pretty straightforward. Here's a basic transcript:

1. It reads a DNA sequence from the command line.
2. It defines a data structure for determining base complements.
3. It splits the DNA sequence into its constituting bases.
4. It then **loops** over bases (from 5′ to 3′-end)…
    1. to take the complement of each base using the data structure and…
    2. to insert the complemented base at the beginning of the complemented DNA sequence, which has the effect of simultaneously reversing it.

5. It concatenates the reverse complemented bases into a DNA sequence.
6. Finally, it prints both the original and the reverse-complemented DNA sequences.

3.5 Perl variables

A **variable** is a storage location in computer memory that can generally be referred to by a specific name, the **identifier** of the variable.

Figure 3.3: A variable is a named storage location.

One can reserve such a location without putting any value into it. This is called **declaring** a variable. In contrast, storing a value into a variable (already declared or not) is called **defining** the variable.

```
# variable declaration
my $box;

# variable definition
$box = 'some stuff';

# variable declaration followed by its definition
my $trunk = 'secret stash';
```

Perl has three main types of variables: **scalars**, **arrays** and **hashes**.

- **Scalars** hold a single value, whether a **number**, a **string** or something else.

  ```
  my $dna_string = 'CATGAACTTCTTTGGCGTCTTGAT';
  ```

- **Arrays** hold a collection of ordered **elements**, which are scalars. The elements can be manipulated as a **list** or individually. The specific elements to be manipulated are selected by using **indices** (singular: **index**) between **square bracket characters** (`[` and `]`). In Perl, indices go from `0` to the number of elements minus one.

  ```
  my @bases = ('A', 'C', 'G', 'T');
  ```

Table 3.1: Elements of the array `@bases` and their indices

0	1	2	3
A	C	G	T

- **Hashes** hold a collection of unordered **values**, which again are scalars and can be accessed by a collection of unique **keys**. They are useful for associating pairs of information, hence their formal names of **associative arrays** or **dictionaries**. Specific values are selected by using keys between **curly brace characters** ({ and }).

```
my %comp_for = (
    A => 'T',
    T => 'A',
    G => 'C',
    C => 'G',
);
```

Table 3.2: Key/value pairs of the hash %comp_for

key	value
A	T
T	A
G	C
C	G

Even if keys have to be *unique*, values can be repeated. Consider the following hash associating a cellular process to each gene of the cluster *dcw* (for *division and cell wall*) in *Escherichia coli*.

```
my %process_for = (
    mraZ => 'other',
    mraW => 'other',
    ftsL => 'FtsI/FtsW recruitment on Z ring',
    ftsI => 'peptidoglycan subunit translocation and assembly',
    murE => 'peptidoglycan biosynthesis',
    murF => 'peptidoglycan biosynthesis',
    mraY => 'peptidoglycan biosynthesis',
    murD => 'peptidoglycan biosynthesis',
    ftsW => 'peptidoglycan subunit translocation and assembly',
    murG => 'peptidoglycan biosynthesis',
    murC => 'peptidoglycan biosynthesis',
    ddlB => 'peptidoglycan biosynthesis',
    ftsQ => 'FtsI/FtsW recruitment on Z ring',
    ftsA => 'cellular division',
    ftsZ => 'cellular division',
);
```

Be careful that smart comments always display hashes as if their keys were sorted in alphabetical order (also known as **lexical order**). This is certainly prettier than the default randomized order (see "The hash random order in gory detail", p.79). However, this can also mask potential bugs (or simply make them more difficult to find).

3.6 How to see what's going on?

Let's add a debugging facility to our program. Insert the following statement after the two first lines.

```
use Smart::Comments;
```

The `use` **keyword** (or **directive** but technically a **builtin function**) has the effect of loading the module `Smart::Comments`. A Perl **module** is basically a collection of high-level features already programmed for you. Perl modules cover a vast array of application domains, including bioinformatics, and the most famous of them are very fine pieces of software. Powerful and robust modules are one of the top reasons why Perl is still widely used today in spite of fierce competition (e.g., **Python**).

The standard Perl **distribution** comes with many `core` modules that you don't need to install before loading them. Much more modules are available on **CPAN** repositories (196,000 as of September 2020) and these can be installed in a breeze using the `cpanm` command-line tool. This is the case of `Smart::Comments`.

```
$ cpanm Smart::Comments
```

So-called *smart comments* begin with three comment characters (`###`). They can do a lot of useful things. To learn more about this module, try using `perldoc` in your terminal.

```
$ perldoc Smart::Comments
```

All installed Perl modules come with a similar **documentation**. This habit is one the strongest assets of the *Perl ecosystem*. For uninstalled modules, go to the main CPAN website instead:
`http://search.cpan.org/`

In their simplest use, smart comments simply display the content of the specified variable to the **standard error stream** (`STDERR`).

```
### $dna_string

# gives:

### $dna_string: 'CATGAACTTCTTTGGCGTCTTGAT'
```

In the remaining of this course, I will always show you the expected output of the computer (i.e., its answer) after a comment similar to the `# gives:` above. Thus, in this case, the *program* is:

```
### $dna_string
```

… while the computer's *output* is:

```
### $dna_string: 'CATGAACTTCTTTGGCGTCTTGAT'
```

When the variable contains multiple elements (e.g., it is an array or a hash), smart comments print its content in a very readable way. This is achieved by formatting the output with the `Data::Dumper` module, which is part of the standard Perl distribution.

```
my @bases = ('A', 'C', 'G', 'T');
### @bases

### %comp_for
```

```
# gives:

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

### %comp_for: {
###              A => 'T',
###              C => 'G',
###              G => 'C',
###              T => 'A',
###              a => 't',
###              c => 'g',
###              g => 'c',
###              t => 'a'
###            }
```

If we want to look at a specific array or hash element using its index, the very simple syntax presented above does not work anymore. Instead, we must use a slightly more sophisticated form.

```
### base: $bases[2]

# gives:

### base: 'G'
```

The expression after the **colon character** (`:`) can be arbitrarily complex, but both the colon and the text before it (any text but some text) are required.

By definition, an **expression** is any combination of **literals**, **constants**, **variables**, **operators**, and **functions** that gets evaluated and returns a **value**. In Perl, nearly everything is an expression. This is what allows Perl programmers to write very concise and natural code.

3.7 A closer look to our killer app

Let's riddle our code with smart comments…

```
# enforce good programming practices
use strict;
use warnings;

use Smart::Comments;

# define a hash mapping the base complements
my %comp_for = (
    A => 'T',   T => 'A',   G => 'C',   C => 'G',
    a => 't',   t => 'a',   g => 'c',   c => 'g',
);
### %comp_for

# read a DNA string, split it into bases and compute its length
my $dna_string = shift;
### $dna_string
my @bases = split //, $dna_string;
### @bases
my $len = @bases;
### $len

# take the complement of each base in turn
# elongate reverse string by inserting complemented bases at its beginning
my @comp_bases;
### @comp_bases
for my $base (@bases) {
    my $comp_base = $comp_for{$base};
    ### base / comp_base : $base . '/' . $comp_base
    unshift @comp_bases, $comp_base;
    ### @comp_bases
}
my $rev_comp_dna_string = join q{}, @comp_bases;
### $rev_comp_dna_string

# print forward and reverse complemented DNA strings
print 'fwd: ' . $dna_string . "\n";
print 'rev: ' . $rev_comp_dna_string . "\n";
```

To capture the output sent to `STDERR`, use the following **shell redirection** in `bash`. This will send all smart comments to the file `rev_comp.log`.

```
$ perl rev_comp.pl CATGAACTTCTTTGGCGTCTTGAT 2> rev_comp.log
```

Then, look at the file and try to understand what is going on.

```
$ less rev_comp.log
```

As you can see, the output sent to `STDOUT`, the **standard output stream**, still gets printed to the screen. This is why separating the two standard streams in your code can be very useful.

3.8 Basic Perl syntax

3.8.1 Names

Perl names (or **identifiers**) all begin with a letter or an **underscore character** (_) and may include any combination of letters, numbers and underscores.

```
# examples from our program

# only letters
@bases
$len
$base

# letters and underscores
%comp_for
$dna_string
@comp_bases
$rev_comp_dna_string

# additional examples with numbers
$seq1
package MyApp2
%id4seq
@blast2_results
```

3.8.2 Sigils

Variable names always have a leading **sigil** (or symbol) that indicates the type of the variable's value.

- Scalars use the **dollar sign character** (`$`).
- Arrays use the **at sign character** (`@`).
- Hashes use the **percent sign character** (`%`).

```
# examples from our program

# scalars
my $dna_string;
my $len;
my $base;

# arrays
my @bases;
my @comp_bases;

# hashes
my %comp_for;
```

3.8.3 Context

Perl **context** sometimes causes issues to beginners. It directly comes from the fact that Larry Wall (born in 1954), the creator of Perl, studied linguistics and human languages.

Basically, it boils down to the idea that the `perl` interpreter always analyzes your code in the light of two different contexts, which can be compared to the *number* (singular/plural) and the *gender* (masculine/feminine) in spoken languages.

In Perl, these two kinds of contexts are the **amount context** (*void*/*scalar*/*list*) and the **value context** (*numeric*/*string*/*boolean*). These are complex and pervasive concepts in Perl and we will come back to them on several occasions. For now, two things are important to remember:

- Variable **assignment** imposes a specific amount context depending on the type of variable receiving the value. Scalars impose **scalar context**, while arrays and hashes impose **list context**. Such impositions are called **coercions**. We will see them again in "Transliteration: `tr//`", p.129.
- Sigils can help us to determine the amount context.

```
# examples from our program

# the hash %comp_for imposes list context
my %comp_for = (
    A => 'T',   T => 'A',   G => 'C',   C => 'G',
    a => 't',   t => 'a',   g => 'c',   c => 'g',
);

# the array @bases imposes list context as well
my @bases = split //, $dna_string;

# the scalar $len imposes scalar context
# in scalar context, an array evaluates to its number of elements
my $len = @bases;

# the scalar $comp_base imposes scalar context
# we get a single value from the hash %comp_for
# thus, the % sigil changes to $ to reflect this fact
my $comp_base = $comp_for{$base};
```

3.8.4 Scope

Perl variables only exist (and are visible) within a certain **scope**, corresponding to where and how they were declared (e.g., with `my`). There exist different scopes (e.g., **our scope** or **dynamic scope**), but the most important to understand for now is the **lexical scope**.

Lexical scope is governed by our (and `perl`'s) reading of a given program. A new lexical scope is created when we enter in a new **block**. A block is defined by a pair of curly brace characters (`{` and `}`) and should be indented (i.e., shifted to the right) with respect to the surrounding code for clarity.

```
for my $base (@bases) { # block start
    my $comp_base = $comp_for{$base};       # these two lines
    unshift @comp_bases, $comp_base;        # are indented with 4 spaces
} # block end
```

Blocks can be put within others like Russian dolls (**nested blocks**). Variables declared inside an **outer lexical scope** exist and are visible from **inner lexical scopes** (delimited by inner blocks). When we exit from a given lexical scope (by crossing over a closing `}`), variables declared inside that scope are destroyed (and the corresponding memory location is freed).

The `for` loop itself will be explained in "The `for` loop", p.43.

```
my $len = @bases;

# $comp_base only exists within the for loop
for my $base (@bases) {
    my $comp_base = $comp_for{$base};
    ### base / comp_base : $base . '/' . $comp_base
    unshift @comp_bases, $comp_base;
    ### @comp_bases
}
### $len         # will work because declared in the same lexical scope
### $comp_base   # will cause a compilation error

# due to the loop construction, $base also only exists within it
### $base        # will cause a compilation error
```

If a variable exists in an outer lexical scope, declaring a new variable of the same name in an inner scope does not overwrite the first one. It only *masks* it (or *shadows* it) until the end of the inner scope.

```
# toy example
my $i = 'Hello world!';

for my $i (1, 2, 3, 4, 5, 6, 7, 8, 9, 10) {
    ### $i
}

### $i

# gives:

### $i: 1
### $i: 2
### $i: 3
...
### $i: 9
### $i: 10

### $i: 'Hello world!'
```

For this reason (and others), a good practice is to declare a variable as late as possible, which in Perl often translates to the innermost scope possible. This will also reduce the total lifetime of the variable since it will get disposed of sooner, i.e., at the end of the narrower scope. However, in the interest of clarity, it is better not to reuse the same identifier for different variables, except if they are obviously designed to hold conceptually related values. We will come back to that when covering functions.

Finally, there is also a lexical scope for the current program file. This **file scope** begins at the first statement. All variables declared outside any block thus exist at the file scope and are available from anywhere in the file. They are sometimes called **global variables**.

3.8.5 `strict` and `warnings` pragmas

At the very beginning of our script are the two following lines.

```
use strict;
use warnings;
```

`strict` and `warnings` are two **pragmas** introduced by the `use` builtin function. While most Perl modules provide new functionality (such as `Smart::Comments`), pragmas influence the behavior of the language itself. By convention, they have lowercase names to differentiate them from other modules.

The `strict` pragma enables compiler checking of symbolic references, bareword use, and variable declaration, whereas the `warnings` pragma enables optional warnings for deprecated, unintended, and awkward behaviors.

For us, this means that *undeclared* variables (e.g., misspelled variables) will cause a **compilation error** and abort execution (`strict`), whereas manipulating *uninitialized* variables will cause a **warning** during execution (`warnings`). Since, enabling these behaviors is incredibly helpful to write better code (i.e., with less bugs), every single Perl script should begin with these two lines.

3.9 List builtin functions

Perl programs often deal with lists of values stored in arrays. Values can be added or removed from arrays using a series of dedicated keywords corresponding to builtin functions.

```
# let's define an array...
my @bases = ('A', 'C', 'G', 'T');
```

Table 3.3: The fresh array `@bases`

0	1	2	3
A	C	G	T

3.9.1 push & pop

`push` and `pop` handle the array as a **stack**, a classical computer data structure governed by the *last-in-first-out* (*LIFO*) motto, in which elements are added and removed from the *end*.

```
# append a new element at the end of the array
push @bases, 'U';
```

The array is automatically *resized* to accomodate the new element (or elements because `push` can add several elements at once). The indices of the new elements start counting at the number following the index of the *last* element.

Table 3.4: `@bases` after `push`

0	1	2	3	4
A	C	G	T	U

```
# now, remove and return the last element of the array
my $base = pop @bases;
### $base

# gives:

### $base: 'U'
```

Contrarily to `push`, `pop` can only deal with one element at a time.

Table 3.5: `@bases` after `pop`

0	1	2	3
A	C	G	T

3.9.2 `unshift` & `shift`

`unshift` and `shift` are similar to `push` and `pop` except that they add and remove values from the *beginning* of the array (instead of the end).

```
# insert a new element at the beginning of the array
unshift @bases, 'N';
```

Since the new element (or elements) is inserted at the beginning of the array, all pre-existing indices are shifted upwards, i.e., towards higher values.

Table 3.6: `@bases` after `unshift`

0	1	2	3	4
N	A	C	G	T

```
# now, remove and return the first element of the array
my $base = shift @bases;
### $base

# gives:

### $base: 'N'
```

Of course, `shift` has the opposite effect and, like `pop`, can only deal with one element at a time.

Table 3.7: `@bases` after `shift`

0	1	2	3
A	C	G	T

If the array is omitted, `shift` acts on the **default array** `@ARGV`, which always contains the **command-line arguments**.

```
my $dna_string = shift;
### $dna_string

### $dna_string: 'CATGAACTTCTTTGGCGTCTTGAT'
```

Here is a toy example to help you to understand how to read arguments from the command line in Perl with `shift`. You can save it as `shift.pl` and try it for yourself using various sets of arguments.

```
$ perl shift.pl ape bear cow dolphin
```

```
use strict;
use warnings;

use Smart::Comments;

### @ARGV
my $one = shift;
### $one
### @ARGV
my $two = shift;
### $two
### @ARGV
my $three = shift;
### $three
### @ARGV
my $four = shift;
### $four
### @ARGV
```

3.9.3 `split` & `join`

`split` and `join` build a list of **substrings** from a string and vice versa.

These builtin functions require a **separator** (also known as a **delimiter**) to *separate* the substrings to split or to join. While separators are often single characters (e.g., `,` or `-`), one can also use multi-character separators (e.g., `::`) and even empty separators, both when splitting and when joining. Consider the examples below to learn of `split` and `join` work.

```
# split a string on each character (using an empty separator)
my $dna_string = 'CATGAACTTCTT';
my @bases = split //, $dna_string;
### base: $bases[3]

# gives:

### base: 'G'
```

Table 3.8: The array `@bases` resulting from the `split`

0	1	2	3	4	5	6	7	8	9	10	11
C	A	T	G	A	A	C	T	T	C	T	T

```
# split a string on a given character (here, the space)
my $sentence = 'Crick proposed the central dogma.';
my @words = split ' ', $sentence;
# the ' ' pattern is actually a special case; for details see
# <http://perldoc.perl.org/functions/split.html>
### word: $words[4]

# gives:

### word: 'dogma.'
```

Table 3.9: The array `@words` resulting from the `split`

0	1	2	3	4
Crick	proposed	the	central	dogma.

```
# join the elements of an array using a given separator
my @bases = ('A', 'C', 'G', 'T');
my $gapped_seq = join '-', @bases;
### $gapped_seq

# gives:

### $gapped_seq: 'A-C-G-T'

# one can use an empty separator
my $plain_seq = join q{}, @bases;
# q{} is equal to '' but easier to read
### $plain_seq

# gives:

### $plain_seq: 'ACGT'

# split and join can use multi-character separators
my $cds = 'CAT::GAA::CTT::CTT';
my @codons = split /::/, $cds;
my $box_cds = '<' . join('> <', @codons) . '>';
### $box_cds

### $box_cds: '<CAT> <GAA> <CTT> <CTT>'
```

Table 3.10: The array `@codons` resulting from the `split`

0	1	2	3
CAT	GAA	CTT	CTT

BOX 2: Perl as an interactive shell

Contrary to Python, which is often used in an interactive way, the Perl language is primarily designed for writing scripts and **one-liners** (see "Perl *one-liners*", p.145 to learn more). Yet, if you want to experiment with Perl in an interactive mode, install one of the so-called **REPL** (for *Read, Evaluate, Print, Loop*) modules. The two most complete ones are `Devel::REPL` and `Reply`.

```
$ cpanm Reply
# ensure that Perl shell supports command history etc
$ cpanm Term::ReadLine::Gnu
```

The transcript below was produced with `Reply`. Note how the Perl shell automatically prints the result of each expression and stores it in the global array `@res`. To quit the Perl shell and get back to the real shell, press the keys `Ctrl` and `D` (abbreviated as `^D`).

```
$ reply

0> my @nums = 1..4
$res[0] = [
  1,
  2,
  3,
  4
]

1> use List::AllUtils qw(sum)
2> sum(@nums)
$res[1] = 10

3> say 'Hello world!'
Hello world!
$res[2] = 1

4> ^D
```

Chapter 4

Digging deeper into Perl

4.1 Another killer app

Reverse-complement is only one of the usual manipulations that molecular biologists impose to their DNA sequences. Another important one is *conceptual translation*, i.e., computer-assisted generation of a protein sequence from a DNA sequence.

Again, there exist web applications for that, such as:
http://www.bioinformatics.org/sms/translate.html

However, since we are *bioinformatician wannabes*, we leave these unsatisfactory solutions to *little league* players! Instead, let's try to make our own `translate` program…

1. Download the *NCBI* file describing all the *genetic codes* (using the shell).

   ```
   $ wget ftp://ftp.ncbi.nih.gov/entrez/misc/data/gc.prt       # Linux
   $ curl -O ftp://ftp.ncbi.nih.gov/entrez/misc/data/gc.prt    # macOS
   ```

2. Open it in a text editor.
3. Locate the definition of the standard genetic code.

   ```
   {
    name "Standard" ,
    name "SGC0" ,
    id 1 ,
    ncbieaa  "FFLLSSSSYY**CC*WLLLLPPPPHHQQRRRRIIIMTTTTNNKKSSRRVVVVAAAADDEEGGGG",
    sncbieaa "---M---------------M---------------M----------------------------"
    -- Base1  TTTTTTTTTTTTTTTTCCCCCCCCCCCCCCCCAAAAAAAAAAAAAAAAGGGGGGGGGGGGGGGG
    -- Base2  TTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGG
    -- Base3  TCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAG
   },
   ```

4. Copy-paste it into a new blank text file.
5. Edit the pasted snippet as follows.

   ```
   my $aa = 'FFLLSSSSYY**CC*WLLLLPPPPHHQQRRRRIIIMTTTTNNKKSSRRVVVVAAAADDEEGGGG';
   my $b1 = 'TTTTTTTTTTTTTTTTCCCCCCCCCCCCCCCCAAAAAAAAAAAAAAAAGGGGGGGGGGGGGGGG';
   ```

```
    my $b2 = 'TTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGG';
    my $b3 = 'TCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAG';
```

6. Save it as `translate.pl` (in the `mod_perl` subdirectory).

4.2 The code for our own `translate`

Complete this code snippet with the following lines.

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

# avoid boilerplate
use Modern::Perl '2011';
use Smart::Comments;

# standard genetic code definition from NCBI gc.prt file
my $aa = 'FFLLSSSSYY**CC*WLLLLPPPPHHQQRRRRIIIMTTTTNNKKSSRRVVVVAAAADDEEGGGG';
my $b1 = 'TTTTTTTTTTTTTTTTCCCCCCCCCCCCCCCCAAAAAAAAAAAAAAAAGGGGGGGGGGGGGGGG';
my $b2 = 'TTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGG';
my $b3 = 'TCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAG';

# build hash for standard code
my %aa_for;
my $codon_n = length $aa;
for my $i (0..$codon_n-1) {
    my $codon = substr($b1, $i, 1) . substr($b2, $i, 1) . substr($b3, $i, 1);
    $aa_for{$codon} = substr($aa, $i, 1);
}
### %aa_for

# read a DNA string and compute its length
my $dna_string = shift;
my $len = length $dna_string;

# split it into uppercased codons
my @codons;
for (my $i = 0; $i < $len; $i += 3) {
    my $codon = substr($dna_string, $i, 3);
    push @codons, uc($codon);
}
### @codons

# translate codons into amino acids
my @aminoacids;
for my $codon (@codons) {
    my $aa = $aa_for{$codon} // 'X';
    push @aminoacids, $aa;
}
```

```
### @aminoacids

# output protein sequence
my $protein = join q{}, @aminoacids;
say $protein;
```

Running this program requires installing a new CPAN module. Do it using the shell in the terminal.

```
$ cpanm Modern::Perl
```

Then, make your program **executable** and try it with the following commands.

```
$ chmod a+x translate.pl
$ ./translate.pl CATGAACTTCTTTGGCGTCTTGAT 2> translate.log
```

(More drum roll…)

```
HELLWRLD
```

Wow, this looks familiar, doesn't it?

Homework

1. Try to understand the Perl novelties appearing in this program.
2. Modify it to accept a second argument that will give the frame to translate (1, 2 or 3).

   ```
   $ ./hw1_translate_rf.pl CATGAACTTCTTTGGCGTCTTGAT 1
   $ ./hw1_translate_rf.pl CATGAACTTCTTTGGCGTCTTGAT 2
   $ ./hw1_translate_rf.pl CATGAACTTCTTTGGCGTCTTGAT 3
   ```

   … which should give the output below.

   ```
   HELLWRLD
   MNFFGVLX
   *TSLAS*X
   ```

Part II

Lesson 2

Chapter 5

Looking at the novelties in `translate.pl`

5.1 Shebang line

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

The first line of our **script** begins by the **shebang character sequence** (`#!`). It is recognized by the shell as an **interpreter directive**, which launches the specified interpreter and passes it the **path** of our script. This allows us to run the script without invoking the `perl` interpreter on the command line. As explained in "Another killer app", p.29, the script must be first made executable with `chmod`.

```
$ chmod a+x translate.pl
```

Older (non-Modern) Perl scripts often begin with an alternative line.

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

You should not use this form because it forces the invokation of the system's Perl, thus bypassing Perlbrew and your own Perl install(s). In contrast, system utilities programmed in Perl always run on the system's Perl due to this very line.

5.2 `Modern::Perl`

```
use Modern::Perl '2011';
```

This line loads the `Modern::Perl` module, which is a shortcut for enabling a series of modern features of Perl. This helps reducing **boilerplate code** (lines of standard code you would have to put in every program). In our case, it replaces the two following lines.

```
use strict;
use warnings;
```

Among the features enabled by `Modern::Perl`, another useful one is `say` (see "The `say` builtin function", p.49). Otherwise, we should have added yet another line.

```
use feature 'say';
```

The `'2011'` means that we want the version of Perl that was available that year, i.e., `perl 5.12`. Each year increments the version number by two units. Odd numbers correspond to **development versions**, whereas even numbers are indicative of **stable versions**.

- `2010-perl 5.10`
- `2011-perl 5.12`
- `2012-perl 5.14`
- `2013-perl 5.16`
- `2014-perl 5.18`
- `2015-perl 5.20`
- `2016-perl 5.22`
- `2017-perl 5.24`
- `2018-perl 5.26`

This module is not part of the standard Perl distribution. It has to be installed, thus. For more information, see the module's documentation.

```
$ cpanm Modern::Perl
$ perldoc Modern::Perl
```

5.3 Perl values: Strings

Perl was invented to process textual information. It is very efficient at this task. That is why Perl is often expanded as *Practical Extraction and Reporting Language* (even if this is only a *backronym*). Later on, we will cover the almighty "Regular expressions", p.125, which were first really developed in Perl. For now, let's just have a look at some more basic ways of handling strings.

5.3.1 Defining and concatenating strings

To represent a **literal string** in a program, surround it with a pair of **quoting characters**. The most common **string delimiters** are **single quotes** (`'` and `'`) and **double quotes** (`"` and `"`). Obviously, strings are printed with the (aptly named) `print` builtin function.

```
my $aa = 'FFLLSSSSYY**CC*WLLLLPPPPHHQQRRRRIIIMTTTTNNKKSSRRVVVVAAAADDEEGGGG';
print 'fwd: ' . $dna_string . "\n";
```

When using single quotes, all characters are treated *literally*, except the single quote itself, which must be *escaped* by preceding it with a **backslash character** (`\`).

```
my $sentence = 'Watson didn\'t propose the central dogma. Crick did.';
```

In contrast, double-quoted strings are pre-processed by Perl in at least two ways:

- **Special characters** are translated into their invisible counterparts. The most common special characters are the **newline character** (`\n`) and the **tab character** (for *tabulation*) (`\t`), which respectively separate lines and columns of data in UNIX-like text files.
- Embedded scalar and array variables are *interpolated*, i.e., their content is substituted for their name. Variable interpolation for an array is like performing a `join` on its elements using the **special variable** `$"` as the **list separator**. By default, the value of `$"` is the single space (`' '`).

  ```
  my @bases = ('A', 'C', 'G', 'T');
  say "@bases";
  ```

```
    # gives

    A C G T
```

In rev_comp.pl, we used something like the following.

```
my $dna_string = 'CATGAACTTCTTTGGCGTCTTGAT';
print 'fwd: ' . $dna_string . "\n";
```

This gives the output below (including the newline).

```
fwd: CATGAACTTCTTTGGCGTCTTGAT
```

We combined multiple strings (here 3) into a larger string with two dots standing for the **concatenation operator** (.). If instead we had wanted to print 5'-CATGAACTTCTTTGGCGTCTTGAT-3', we could have used either one of the two following lines.

```
print '1. 5\'-' . $dna_string . '-3\'' . "\n";
print "2. 5'-$dna_string-3'\n";
```

However, using double quotes in the second line dispenses us from (1) escaping single quotes appearing within the string and (2) using dots for concatenation.

```
1. 5'-CATGAACTTCTTTGGCGTCTTGAT-3'
2. 5'-CATGAACTTCTTTGGCGTCTTGAT-3'
```

BOX 3: Avoiding unwanted interpolation

In *Modern Perl*, we often prefer explicitly concatenating **single-quoted** strings with the dot operator rather than putting all of them within a single **double-quoted** string. This helps avoiding undesired variable interpolation.

```
my $device = 'Illumina HiSeq 2000';
my $cost = '500,000';

print "1. The $device is worth at least $cost $!\n";
print '2. The ' . $device . ' is worth at least ' . $cost . '$!' . "\n";
```

The issue here is that $! is a special variable (**ERRNO**) containing a description of the last error related to the *C library* (e.g., failed file opening). Since it is empty at this moment, it gets interpolated to nothing by the double quotes, hence the lack of literal $! at the end of our string in the first case.

```
1. The Illumina HiSeq 2000 is worth at least 500,000
2. The Illumina HiSeq 2000 is worth at least 500,000$!
```

Alternatively, we could have escaped $! in the double-quoted string.

```
print "3. The $device is worth at least $cost\$!\n";
```

This would have resulted in the same output as in the second case.

```
3. The Illumina HiSeq 2000 is worth at least 500,000$!
```

5.3.2 Alternate quoting operators

If we have a string with too many quoting characters to escape, we can switch to alternate **quoting operators** (`q{}` and `qq{}`), which respectively emulate single quotes and double quotes.

```
print '1. Watson didn\'t propose the central dogma. Crick did.';
print q{2. Watson didn't propose the central dogma. Crick did.};
```

Both give the same output (without a newline).

```
1. Watson didn't propose the central dogma. Crick did.
2. Watson didn't propose the central dogma. Crick did.
```

Actually, you can use any *balanced pair* of characters (or even a single character) in place of the curly brace characters, e.g., `q||` or q<>.

```
print qq|The q{} and qq{} operators can be used in place of '' and "".\n|;
```

This greatly helps defining complicated strings (including a newline here).

```
The q{} and qq{} operators can be used in place of '' and "".
```

BOX 4: Unicode strings

If you read any good recent text about Perl (e.g., the reference book used here), you will find an in-depth discussion of **Unicode strings**. Unicode and the associated **utf8 encoding** are important topics when processing international text (containing non-English characters, such as accented or even non-roman letters).

However, it is quite complex and clearly dispensable in a Perl course that is oriented to bioinformatics. Just remember to consult the adequate documentation if you ever want to process French text. Otherwise, you will rapidly run into trouble!

5.4 String builtin functions

Our `translate.pl` begins by the following unwieldy piece of code. Let's analyze it!

```
# standard genetic code definition from NCBI gc.prt file
my $aa = 'FFLLSSSSYY**CC*WLLLLPPPPHHQQRRRRIIIMTTTTNNKKSSRRVVVVAAAADDEEGGGG';
my $b1 = 'TTTTTTTTTTTTTTTTCCCCCCCCCCCCCCCCAAAAAAAAAAAAAAAAGGGGGGGGGGGGGGGG';
my $b2 = 'TTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGG';
my $b3 = 'TCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAG';

# build hash for standard code
my %aa_for;
my $codon_n = length $aa;
for my $i (0..$codon_n-1) {
    my $codon = substr($b1, $i, 1) . substr($b2, $i, 1) . substr($b3, $i, 1);
    $aa_for{$codon} = substr($aa, $i, 1);
}
```

5.4.1 length

The first four lines simply *declare* and immediately *define* four string variables. Then, we declare en empty hash, `%aa_for`. Its name suggests that it will hold values corresponding to *amino acids*. Since it is a hash, each value will be associated to some specific key, presumably a codon. The remaining of the code serves the purpose of building this hash.

```
my $codon_n = length $aa;
```

First, we infer the number of codons from the length of the `$aa` string using the string builtin function `length`. Note that `length` also counts special characters, such as newline characters. However, there are no special characters here.

You might be surprised that we don't directly assign `64` to `$codon_n`. The reason we do this is to make our code the most *general* possible. (Granted, in this case, we would probably need to visit another planet to deal with anything other than 64 codons!) An additional advantage of this approach is that if our programming logic is buggy or if our input get corrupted, we notice it right away because `$codon_n` will not hold `64`.

Moreover, you should avoid using any **magic number** when programming, because such *naked values* hinder the autodocumentation of your code. Hence, in the code snippet below, we use the expression `$codon_n-1` rather than a mere `63`. This makes our code much clearer.

5.4.2 substr

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

Then, we setup a **loop**, in which the **iterator variable** `$i` will go from the value `0` to the number of codons minus one, i.e., `63`. This loop will help processing each codon in turn. We will come back to the `for` loop and to the `..` construct in the "*foreach-style* `for` loop", p.43.

```
my $codon = substr($b1, $i, 1) . substr($b2, $i, 1) . substr($b3, $i, 1);
```

Within the loop, we build each codon by concatenating three individual bases (hence the `1`), each taken from the specific position `$i` in the strings `$b1`, `$b2` and `$b3`. To this end, we use the string builtin function `substr`, which extracts a *substr*ing out of larger string, as its name suggests. This is a powerful function, but we use it here in one of its most basic forms.

```
my $string = 'ABCDEFGHIJ';
### $string

my $start_position = 4;
my $number_of_chars = 3;
my $substr = substr($string, $start_position, $number_of_chars);
### $substr

# gives (back to using Smart::Comments):
```

```
### $string: 'ABCDEFGHIJ'

### $substr: 'EFG'
```

Beware, as for arrays, character indexing begins at `0`, not `1`, hence `'EFG'` and not `'DEF'` when starting at position 4. For more information and examples, see:
http://perldoc.perl.org/functions/substr.html

BOX 5: The fourth argument of `substr` and `splice`

If you look up for `substr` documentation, you will come across its optional fourth argument, the *replacement* string. Even if rarely used, this mode is intriguing for the biologist because it literally *splices* a string to surgically replace a substring by another substring.

```
#                          1         2         3         4         5
#                0123456789012345678901234567890123456789012345678901234567890
my $sentence = 'Introns are meant to be removed by the spliceosome.';
my $substr = substr $sentence, 24, 7, 'spliced out';
### $substr
say $sentence;

# gives:

### $substr: 'removed'
Introns are meant to be spliced out by the spliceosome.
```

Similarly, most list builtin functions (`push`, `pop`, `shift` and `unshift`) are special cases of a more general function called `splice`, which changes the elements of an array. As you can see, `substr` and `splice` behave consistently and return what was spliced out from the string or the array.

```
my @words = split q{ }, $sentence;
my @array = splice @words, 5, 2, ('discarded');
### @array
say "@words";

# gives:

### @array: [
###           'spliced',
###           'out'
###         ]
Introns are meant to be discarded by the spliceosome.
```

5.4.3 uc & lc

When splitting the user's string into codons in `translate.pl`, we use the following statement.

```
push @codons, uc($codon);
```

The builtin function uc (for *uppercase*) transforms each codon to capital letters, which ensures that both lowercase and uppercase versions of the codons are recognized during conceptual translation. There is also a lc (for *lowercase*) function that performs the opposite transformation.

5.5 The for loop

The versatility of the for loop is what makes it one of the workhorses of Perl. It comes in two main flavors: the *foreach-style* and the *C-style*. Those are illustrated below.

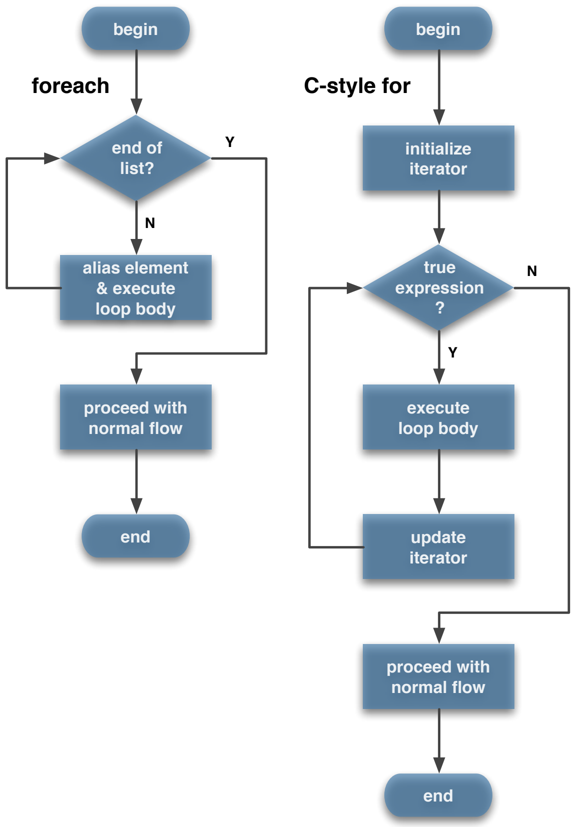

Figure 5.1: The two styles of the for loop

In principle, the equivalent keyword foreach can be used anywhere the for keyword is acceptable. For the sake of clarity, however, foreach should be reserved for the first style, i.e., going through a list, as we will see right now.

5.5.1 *foreach-style* for loop

In rev_comp.pl, we used a *foreach-style* for loop to iterate over the elements of an array.

```
for my $base (@bases) {      # equal to: foreach my $base (@bases) {
    my $comp_base = $comp_for{$base};
    unshift @comp_bases, $comp_base;
}
```

BOX 6: Iterator variables as aliases

The iterator variable is actually some kind of *alias* for the current element of the array `@bases`. This means that if you modify `$base` within the loop, your changes will be reflected in `@bases`. This is incredibly powerful but this can be extremely surprising too!

```
my @bases = ('A', 'C', 'G', 'T');
### @bases

for my $base (@bases) {
    ### $base
    $base = '?';
}
### @bases

# gives:

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

### $base: 'A'
### $base: 'C'
### $base: 'G'
### $base: 'T'

### @bases: [
###            '?',
###            '?',
###            '?',
###            '?'
###          ]
```

In `translate.pl`, the first `for` loop is similar, except that we don't iterate over the elements of an array, but over the values of a list. This list is a collection of numbers generated on the fly with the useful **range operator** (`..`).

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

### 0..63

# gives:
```

```
### 0..63: 0,
###        1,
...
###        62,
###        63
```

Table 5.1: The list resulting from the expression `0..63`

0	1	2	3	4	5	6	7	...	58	59	60	61	62	63

If you have ever programmed in another language than Perl, you might wonder whether building a list of 1,000,000,000 elements just to count from one to one billion is the way to go. Even if it looks counter-intuitive, the `perl` interpreter is so optimized that a *foreach-style* `for` loop will actually be faster for this specific task than the (apparently less wasteful) *C-style* `for` loop described just below. Therefore, never sacrifice code clarity for execution speed, except when you really need it and are sure that the awkward version is indeed faster (thanks to **code profiling**).

5.5.2 *C-style* `for` loop

The second `for` loop is slightly more complicated to understand. It is a *C-style* for loop, the purpose of which is to split our input DNA sequence into codons.

```
my @codons;
for (my $i = 0; $i < $len; $i += 3) {
    my $codon = substr($dna_string, $i, 3);
    push @codons, uc($codon);
}
```

Splitting a string into chunks of size greater than 1 cannot be easily achieved with `split`, contrary to what we did for reverse-complementing DNA. Instead, we must use `substr`, but the difficulty is to move our *sliding window* by steps of 3. That is where the *C-style* `for` loop becomes useful.

The canonical looping construct has three subexpressions:

- The first subexpression (**initialization**) executes only once, before the **loop body** (enclosed in a block delimited by `{` and `}`) executes. This is often where the iterator variable is both declared and defined, which will limit its lexical scope to the loop body.
- Perl evaluates the second subexpression (**conditional comparison**) before each iteration of the loop body. When this yields a Boolean **true value** (see "Boolean expressions", p.47), iteration proceeds, whereas iteration stops as soon as the conditional comparison returns a **false value**.
- The final subexpression executes after each iteration of the loop body. This is generally where to put the instructions for modifying the iterator variable.

To better understand how it works, isolate a minimal (yet functional) chunk of code containing the *C-style* `for` loop and riddle it with smart comments.

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

use Modern::Perl '2011';
use Smart::Comments;

my $dna_string = 'CATGAACTTCTTTGGCGTCTTGAT';
my $len = length $dna_string;

my @codons;
### @codons
for (my $i = 0; $i < $len; $i += 3) {
    ### $len
    ### $i
    ### $i < $len: $i < $len
    my $codon = substr($dna_string, $i, 3);
    ### $codon
    push @codons, uc($codon);
    ### @codons
}
```

Save this piece of code as `c_for_test.pl`, make it executable and try it with the following commands. Look at the log file. Can you explain the *C-style* `for` loop behavior now?

```
$ chmod a+x c_for_test.pl
$ ./c_for_test.pl 2> c_for_test.log
$ less c_for_test.log
```

BOX 7: Iterators and scope

Unfortunately, we don't see the conditional comparison evaluating to false. Can you see why? How could we modify the code for the purpose of illustrating that?

```
...

### $len: 24
### $i: 21
### $i < $len: 1

### $codon: 'GAT'

### @codons: [
###             'CAT',
###             'GAA',
###             'CTT',
###             'CTT',
###             'TGG',
###             'CGT',
###             'CTT',
###             'GAT'
###           ]
```

Yes, we simply have to print the conditional comparison once again after the loop. However, this requires the iterator variable `$i` to have survived past the loop, which is not the case since its lexical scope is limited to the loop body. This can be changed by declaring (and optionally defining) it outside the loop.

```
my $i = 0;
for ( ; $i < $len; $i += 3) {
    ### $len
    ### $i
    ### $i < $len: $i < $len
    # ...
}
### $i
### $i < $len: $i < $len
```

Here, we leave out the first subexpression of the *C-style* `for` loop because we initialize the iterator variable `$i` to `0` when we declare it. Actually, all three subexpressions are *optional*, yet omitting more than one is either high-level or badly-written Perl, depending on the circumstances!

```
# infinite C-style for loop
for ( ; ; ) {
    # please let me outta here!
}
```

5.6 Boolean expressions

We have just seen that the *C-style* `for` loop uses a conditional comparison to control its iterating behavior. Conditional comparisons and conditional statements enable a specific value context known as the **boolean context**.

5.6.1 Perl's vision of truth

In boolean context, any Perl expression evaluates to either **true** or **false**. If you know some other programming language, the way Perl handles *truth* can be disorienting. Indeed, it has no single true value, nor a single false value. Instead, scalars (e.g., numbers and strings), arrays and hashes evaluate to true or false depending on their content.

The main rules are as follows:

- Evaluate to false…
  - the value `undef` and any undefined (uninitialized) variable,
  - any number that means zero: `0`, `0.0`, `0e0`,
  - the empty string (`''`, better written as `q{}`) and the string `'0'`,
  - the empty list (`()`), the empty array and the empty hash.
- Evaluate to true…
  - any number that does not mean zero,
  - all non-empty and non-zero strings, including the surprising `'0.0'`, `'0e0'`,
  - an array that contains at least one element, even if undefined,
  - a hash that contains at least one key/value pair, even if undefined.

5.6.2 The undef value

Perl undef value represents an unassigned, undefined and unknown value. Declared but undefined scalar variables contain undef. In *Modern Perl*, there is no such thing as an undefined array or an undefined hash, only empty arrays and hashes (with zero element or key/value pair).

```
my $dna_string;
### $dna_string

# beware, empty and zero are not undefined!
my $empty = q{};
my $zero = 0;
### $empty
### $zero

# gives:

### $dna_string: undef

### $empty: ''
### $zero: 0
```

Whenever you try to print or manipulate a variable containing undef, the perl interpreter warns you. This is due to the use warnings or use Modern::Perl directives at the beginning of your programs (see "strict and warnings pragmas", p.23).

```
use warnings;

my $dna_string;
print $dna_string;
my @bases = split //, $dna_string;

# gives:

Use of uninitialized value $dna_string in print at undef.pl line 4.
Use of uninitialized value $dna_string in split at undef.pl line 5.
```

Pay attention to warnings because they often indicate a bug in your code!

5.6.3 Logical defined-or operator

In the last loop of translate.pl, in which the conceptual translation actually takes place, there is a *Modern Perl* construct taking advantage of the logical **defined-or operator** (//).

```
my @aminoacids;
for my $codon (@codons) {
    my $aa = $aa_for{$codon} // 'X';
    push @aminoacids, $aa;
}
```

This reads as follows: *"Put into variable $aa the amino acid for the codon used as the hash key. If there is no amino acid for that codon, put* `'X'` *instead."* The purpose of this construct is thus to provide a **default value** for non-existing (i.e., misspelled or ambiguous) codons.

To further elaborate on this, providing a *non-existing* key when accessing a hash always returns the `undef` value. Carelessly using an `undef` value in subsequent code then leads to the *uninitialized value* warning. To avoid this issue, one has to test for the `undef` value with the `defined` builtin function.

```
# define non-existing hash key
my $codon = 'CTN';

# use returned value carelessly
my $aa = $aa_for{$codon};
say qq{aa for codon "$codon" is "$aa".};

# fix 1a: check for undef value
say qq{There is no defined AA for codon "$codon".}
    unless defined $aa;

# fix 1b: check for false value
say qq{There is no true AA for codon "$codon".}
    unless $aa;

# fix 2: explictly check for non-existing key
say qq{There exists no codon "$codon" in the genetic code.}
    unless exists $aa_for{$codon};

# gives:
```

```
Use of uninitialized value $aa in concatenation (.) or string at exists.pl line 28.
aa for codon "CTN" is "".
There is no defined AA for codon "CTN".
There is no true AA for codon "CTN".
There exists no codon "CTN" in the genetic code.
```

Alternatively, one can explicitly test for the existence of a given key in the hash using the `exists` builtin function. This approach is more verbose but useful to avoid triggering **autovivification**, an advanced topic covered in "Defining and using nested data structures", in the second part of this course.

5.7 The say builtin function

The very last line of `translate.pl` uses the `say` builtin function to print the protein sequence. This useful extension of *Modern Perl* acts exactly like a regular `print`, except that it automatically appends a newline character (`\n`) to the end of the string, thus dispensing us from explicitly adding it.

```
my $protein = join q{}, @aminoacids;
say $protein;
```

We will see in "Writing files", p.98, that `say` helps lightening the syntax of code chunks that have to display complex data structures to the user. Just remember to enable it with the required `use` directive.

```
use Modern::Perl '2011';    # allows us to use 'say'

my $device = 'Illumina HiSeq 2000';
my $cost = '500,000';

print '1. The ' . $device . ' is worth at least ' . $cost . '$!' . "\n";
say '2. The ' . $device . ' is worth at least ' . $cost . '$!';
```

Both lines output the same text (including a newline).

```
1. The Illumina HiSeq 2000 is worth at least 500,000$!
2. The Illumina HiSeq 2000 is worth at least 500,000$!
```

Chapter 6

Batch *vs.* interactive programs

6.1 Acquiring user input

In contrast to *GUI*-based applications, software designed for *console* use often requires command-line arguments to control its behavior. However, this is not mandatory. Instead, some programs directly ask the user for input in an interactive fashion, such as text-based games.

In this section, we will present two programs (`translate_rf.pl` and `codon_quizz.pl`), each one following a different approach with respect to user input.

6.2 How to improve our killer app?

1. Make a copy of `translate.pl` and save it as `translate_rf.pl`.
2. Edit the new document to match the revised version below. Can you spot the differences?
3. Look at your new code. What do you think it will do?
4. Execute `translate_rf.pl` and try to trigger all its behaviors.

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

# avoid boilerplate
use Modern::Perl '2011';

# use Smart::Comments;      # disabled by default; when debugging use
                            # perl -MSmart::Comments <script.pl>

unless (@ARGV == 2) {
    die <<"EOT";
Usage: $0 <dna-string> <reading-frame>
This tool translates DNA sequences to proteins using the standard genetic code.
It requires a DNA string and a reading frame (1, 2, 3, -1, -2, -3).
Example: $0 CATGAACTTCTTTGGCGTCTTGAT 1
EOT
}

```

```perl
# standard genetic code definition from NCBI gc.prt file
my $aa = 'FFLLSSSSYY**CC*WLLLLPPPPHHQQRRRRIIIMTTTTNNKKSSRRVVVVAAAADDEEGGGG';
my $b1 = 'TTTTTTTTTTTTTTTTCCCCCCCCCCCCCCCCAAAAAAAAAAAAAAAAGGGGGGGGGGGGGGGG';
my $b2 = 'TTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGG';
my $b3 = 'TCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAG';

# build hash for standard code
my %aa_for;
my $codon_n = length $aa;
for my $i (0..$codon_n-1) {
    my $codon = substr($b1, $i, 1) . substr($b2, $i, 1) . substr($b3, $i, 1);
    $aa_for{$codon} = substr($aa, $i, 1);
}
### %aa_for

# read a DNA string and reading frame from command line
my $dna_string = shift;
my $reading_frame = shift;

die 'ABORT! Negative reading frames not yet implemented!'
    if $reading_frame < 0;

die 'ABORT! Reading frame must be one of (1, 2, 3)!'
    unless $reading_frame >= 1 && $reading_frame <= 3;

# compute length of DNA string and split it into uppercased codons
my @codons;
my $len = length $dna_string;
my $offset = $reading_frame - 1;
for (my $i = $offset; $i < $len; $i += 3) {
    my $codon = substr($dna_string, $i, 3);
    push @codons, uc($codon);
}
### @codons

# translate codons into amino acids
my @aminoacids;
for my $codon (@codons) {
    my $aa = $aa_for{$codon} // 'X';
    push @aminoacids, $aa;
}
### @aminoacids

# output protein sequence
my $protein = join q{}, @aminoacids;
say $protein;
```

6.3 Let's make our first game!

When you are done with our translation tool, open a blank document, type in the program below and save it as `codon_quizz.pl`. Note that the beginning is very similar to our last work (think *copy-paste*).

If you want to try it, you'd better having a look at the cheat sheet at the end of this chapter!

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

# avoid boilerplate
use Modern::Perl '2011';
use List::Util 'shuffle';

# use Smart::Comments;      # disabled by default; when debugging use
                            # perl -MSmart::Comments <script.pl>

# standard genetic code definition from NCBI gc.prt file
my $aa = 'FFLLSSSSYY**CC*WLLLLPPPPHHQQRRRRIIIMTTTTNNKKSSRRVVVVAAAADDEEGGGG';
my $b1 = 'TTTTTTTTTTTTTTTTCCCCCCCCCCCCCCCCAAAAAAAAAAAAAAAAGGGGGGGGGGGGGGGG';
my $b2 = 'TTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGG';
my $b3 = 'TCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAG';

# build hash for standard code
my %aa_for;
my $codon_n = length $aa;
for my $i (0..$codon_n-1) {
    my $codon = substr($b1, $i, 1) . substr($b2, $i, 1) . substr($b3, $i, 1);
    $aa_for{$codon} = substr($aa, $i, 1);
}
### %aa_for

print <<'EOT';
Try to guess the correct amino acid for each codon.
STOPs are symbolized by an asterisk (*).
Type EXIT for leaving the quizz.
Good luck!
EOT

# assemble pool of questions from hash keys
my @questions = shuffle keys %aa_for;
my $score = 0;
my $total = 0;

# loop through questions asking for correct answer

QUESTION:
while (my $codon = shift @questions) {

    # ask question and wait for answer
    print $total+1 . '. ' . $codon . ': ';
    my $answer = <>;
    chomp $answer;
    $answer = uc $answer;       # make our quizz robust

    # leave early if asked to do so
    last QUESTION if $answer eq 'EXIT';

    # check answer
    my $aa = $aa_for{$codon};
    if ($answer eq $aa) {
        say 'Correct!';
        $score++;               # increment score
    }
    else {
        say "Wrong! Answer was $aa!";
    }

    # track number of questions
    $total++;
}

say "Your final score is $score on $total";
```

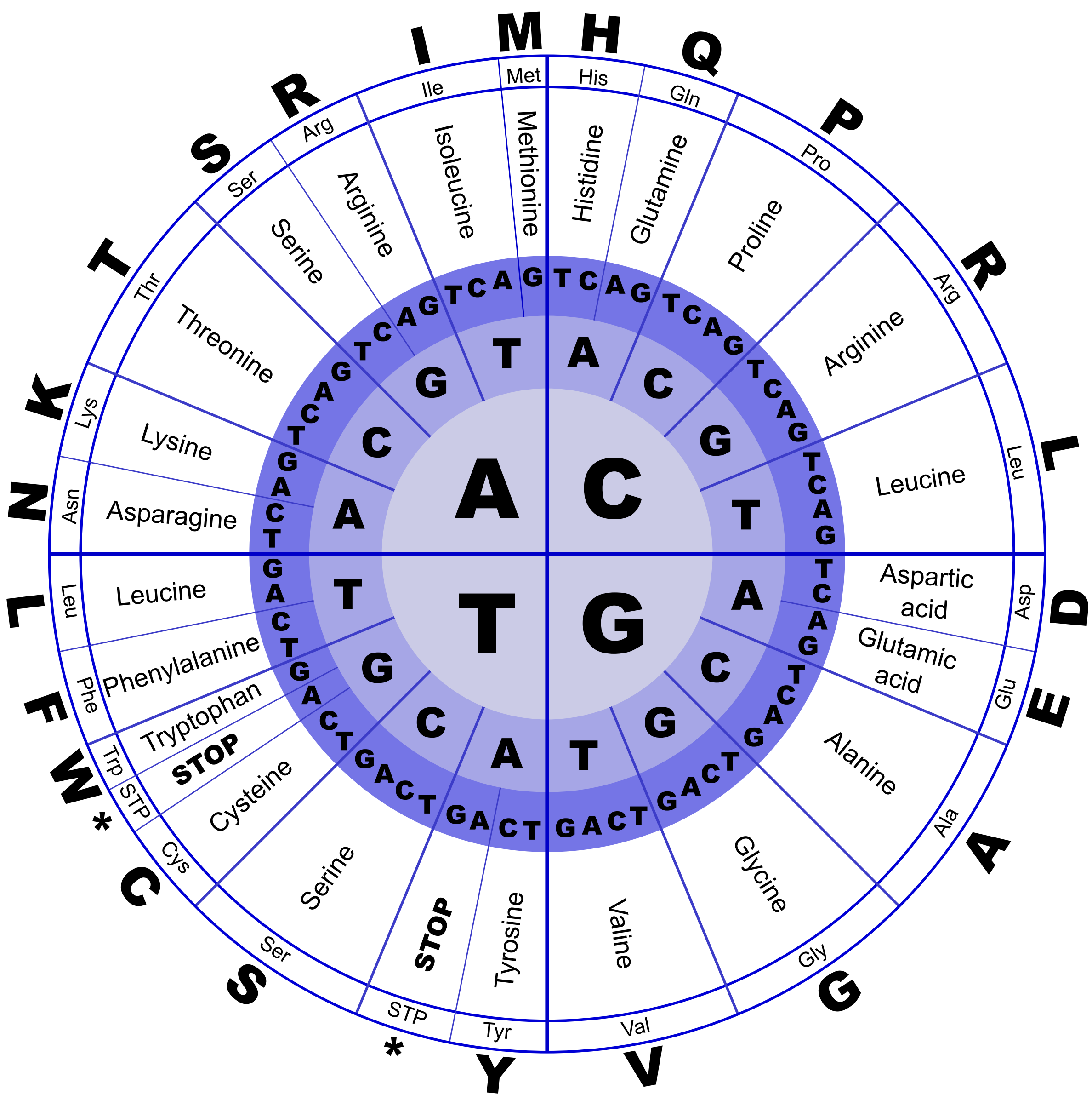

Figure 6.1: The standard genetic code

Homework

1. Try to understand the Perl novelties appearing in our two new programs.
2. As you have probably noticed by now, our conceptual translation tool (`translate_rf.pl`) is not yet complete. Recycle the reverse-complementing code in `rev_comp.pl` to implement the negative reading frames. You might be inspired by the **branching directives** used in `codon_quizz.pl`. Don't worry, we will cover them in detail in "Branching directives", p.61.

   ```
   $ ./hw2_translate_rf_minus.pl CATGAACTTCTTTGGCGTCTTGAT  1
   $ ./hw2_translate_rf_minus.pl CATGAACTTCTTTGGCGTCTTGAT -1
   $ ./hw2_translate_rf_minus.pl CATGAACTTCTTTGGCGTCTTGAT -2
   ```

   … which should give the output below.

   ```
   HELLWRLD
   IKTPKKFM
   SRRQRSSX
   ```

Part III

Lesson 3

Chapter 7

Becoming a control freak

7.1 Control flow in Perl

Perl's basic **control flow** is straightforward. Program execution starts at the *beginning* (the first line of the file executed) and continues to the *end*. Our last two programs used several **branching directives** and **looping directives** beyond the `for` loop. Together, such Perl constructions allow us to alter the flow of execution of our code.

This means that instead of executing our statements each one in turn, from the first one to the last one, these directives can force the `perl` interpreter to skip (*branching*) or repeat (*looping*) one or more statements based on the results of conditional comparisons. Let's recapitulate the control flow directives we encountered in `translate_rf.pl` and `codon_quizz.pl`.

7.2 Branching directives

The new version of our conceptual translation tool is definitely more user-friendly than our first attempt. It can achieve that by adapting its behavior depending on the user's actions. Hence, when the user invokes `translate_rf.pl` without specifying any command-line arguments, the program reacts by printing a brief set of instructions (known as the **usage message**) to the standard error stream.

```
$ ./translate_rf.pl
```

```
Usage: ./translate_rf.pl <dna-string> <reading-frame>
This tool translates DNA sequences to proteins using the standard genetic code.
It requires a DNA string and a reading frame (1, 2, 3, -1, -2, -3).
Example: ./translate_rf.pl CATGAACTTCTTTGGCGTCTTGAT 1
```

Every program you will ever write should act like this. Trust me, even *you* will eventually forget how your own programs work after a few months of non-use!

- A minimal usage message should print the name of the program and the *mandatory* command-line arguments it expects. The words between **angle bracket characters** (< and >) are called **placeholders** and stand for the actual arguments. The angle brackets themselves must not be typed on the command line.

- A better usage message will further include a sentence about the purpose of the program, possibly supplemented by examples of use that the user can copy-paste to her terminal.

Moreover, if the user specifies a reading frame that `translate_rf.pl` cannot handle, the program reacts by printing an **error message** to the standard error stream and then stopping its execution.

```
$ ./translate_rf.pl CATGAACTTCTTTGGCGTCTTGAT -1

ABORT! Negative reading frames not yet implemented! at ./translate_rf.pl line 37.

$ ./translate_rf.pl CATGAACTTCTTTGGCGTCTTGAT 4

ABORT! Reading frame must be one of (1, 2, 3)! at ./translate_rf.pl line 40.
```

7.2.1 `if` & `unless`

The display of the usage message is controlled by the following block of code. It makes use of several Perl **idioms** to reach an excellent functionality / verbosity ratio without sacrificing legibility.

```
unless (@ARGV == 2) {
    die <<"EOT";
Usage: $0 <dna-string> <reading-frame>
This tool translates DNA sequences to proteins using the standard genetic code.
It requires a DNA string and a reading frame (1, 2, 3, -1, -2, -3).
Example: $0 CATGAACTTCTTTGGCGTCTTGAT 1
EOT
}
```

`unless` is the keyword for introducing a branching directive that exactly means what it is supposed to mean: *"Execute the following block of code* ***unless*** *the associated conditional expression evaluates to a true value."* In such a **block form**, the conditional expression has to be enclosed between a pair of regular **parenthesis characters** (`(` and `)`) and followed by the code block to be executed, hence delimited by curly brace characters (`{` and `}`).

The **numeric equality** (`==`) is a **comparison operator** that means *is numerically equal to* and imposes scalar context to the evaluation. In scalar context, remember that an array evaluates to its number of elements. Since `@ARGV` is the default array that contains the command-line arguments (and given the content of the code block that follows), this line thus reads: *"Print the usage message and stop execution* ***unless*** *the number of command-line arguments* ***is equal to*** *two."*

Note that the exact same result could have been achieved with the code below. It is based on the `if` branching directive, of which `unless` is the negated form, and on the **numeric inequality** (`!=`) comparison operator. It reads: *"Print the usage message and stop execution* ***if*** *the number of command-line arguments* ***is not equal to*** *two."*

```
if (@ARGV != 2) {
    # print usage message and stop execution
}
```

`if` and `unless` can be used interchangeably, provided that the associated conditional expression is properly negated. You can use the one you find the most appropriate for each particular situation. This is an illustration of the famous Perl motto: *"TIMTOWTDI—There is more than one way to do it."*

7.2.2 `else` & `elsif`

Overall, branching directives are needed in three cases (illustrated below):

1. when we need to **fix something before proceeding** with the normal flow;
2. when we **face an alternative** with two or more options that each entails a different set of actions;
3. when we encounter an error and have to **abort the execution** of the program.

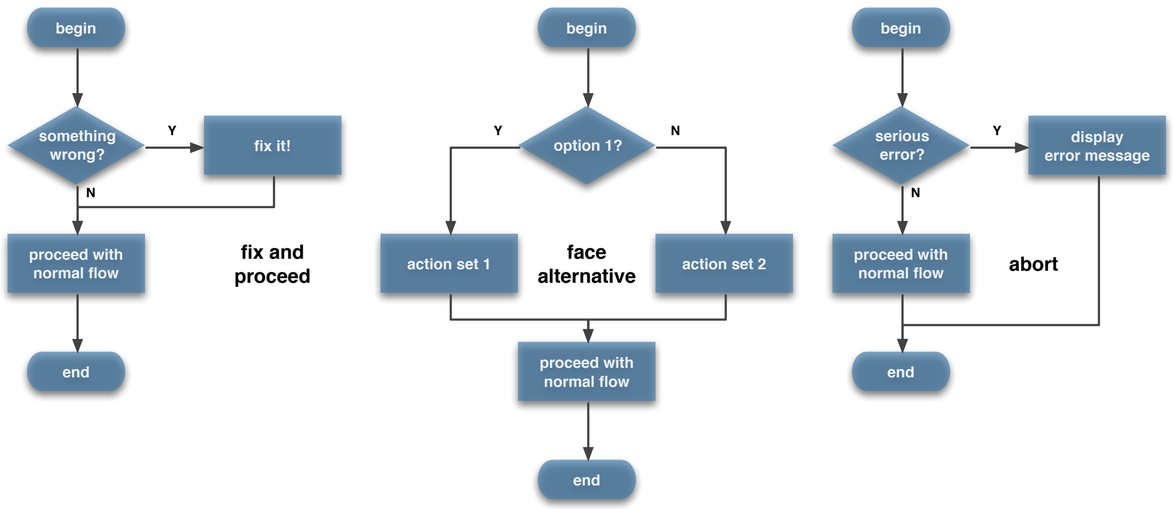

Figure 7.1: The three uses of branching directives

In my experience, cases 1 (*fix-and-proceed*) and 3 (*abort*) are more common than case 2 (*face-alternative*). For example, we have just discussed three error handling situations that correspond to case 3, whereas the last homework assignment can be solved with a case 1 (partial listing only).

```
# ... beginning of the program

# read a DNA string and reading frame from command line
my $dna_string = shift;
my $reading_frame = shift;

die 'ABORT! Reading frame must be one of (1, 2, 3, -1, -2, -3)!'
    unless $reading_frame >= -3 && $reading_frame <= 3 && $reading_frame != 0;

# takes reverse complement if negative frame
if ($reading_frame < 0) {

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

    my @bases = split //, $dna_string;
    my @comp_bases;
    for my $base (@bases) {
        my $comp_base = $comp_for{$base};
        unshift @comp_bases, $comp_base;
    }
    $dna_string = join q{}, @comp_bases;

    $reading_frame = -$reading_frame;    # negate frame
}

# compute length of DNA string and split it into uppercased codons
my @codons;
my $len = length $dna_string;
my $offset = $reading_frame - 1;
for (my $i = $offset; $i < $len; $i += 3) {
    my $codon = substr($dna_string, $i, 3);
    push @codons, uc($codon);
}

# ... remaining of the program
```

BOX 8: Calling external programs with the `qx(...)` operator

As an alternative to cutting-and-pasting the guts of `rev_comp.pl` into `translate_rf.pl`, one could run the former directly from the latter. This is very easy to do using the **executing quoting operator** (`qx(...)`), especially designed to execute **system calls** from a Perl program.

```
# in translate_rf.pl
if ($reading_frame < 0) {
    $dna_string = qx(perl rev_comp.pl $dna_string);
    $reading_frame = -$reading_frame;
}
```

The output sent to `STDOUT` by `rev_comp.pl` is captured by `qx(...)` to overwrite the content of the variable $dna_string. For this to work, however, one must also tweak the original script, so as to only print the reverse-complemented DNA string.

```
# in rev_comp.pl
# print 'fwd: ' . $dna_string . "\n";
# print 'rev: ' . $rev_comp_dna_string . "\n";
print $rev_comp_dna_string;
```

With case 2, additional keywords are necessary for introducing the sets of actions beyond the first one. When the alternative has only two options, a single `else` directive is enough. For example, in `codon_quizz.pl`, we need to do only two different things depending on the user's answer to our question, hence the straightforward `if/else` construct.

```
# check answer
my $aa = $aa_for{$codon};
if ($answer eq $aa) {
    say 'Correct!';
    $score++;                # increment score
}
else {
    say "Wrong! Answer was $aa!";
}
```

Alternatives with three options or more are even less common, but can nevertheless be handled with one or more `elsif` keywords before the final `else`.

```
if ($unit eq 'bp') {
    # do nothing
}
elsif ($unit eq 'kb') {
    $size /= 1000;           # divide $size by 1000
}
elsif ($unit eq 'Mb') {
    $size /= 1e6;            # ... by one million
}
elsif ($unit eq 'Gb') {
    $size /= 1e9;            # ... by one billion
}
else {
    die 'Unknown unit for reporting sequence size!';
}
```

BOX 9: Hashes instead of cascades of `elsif`

However, most of these situations can be elegantly solved using a **switch table** strategy.

```
my %div_for = (
    bp => 1,
    kb => 1000,
    Mb => 1e6,               # one million
    Gb => 1e9,               # one billion
);
my $div = $div_for{$unit};
die 'Unknown unit for reporting sequence size!' unless $div;
$size /= $div;               # divide $size by suitable divisor
```

If there is no key for the unit in the hash `%div_for`, the associated value is `undef`, which evaluates to a false value in a boolean context and triggers the error message. Otherwise, the returned value is used as the divisor. Consider how it is easy to expand our code to handle more units.

```
my %div_for = (
    ...
    Gb => 1e9,               # one billion
    Tb => 1e12,              # one trillion
    Pb => 1e15,              # one quadrillion
);
```

When the multiple sets of actions are very similar, yet another possibility is to write a *parameterized version of the code*. We used this approach in `translate_rf.pl` to handle the three different frames. Instead of always extracting codons from position `0`, we extract them from an `$offset` variable that itself directly depends on the user-defined `$reading_frame`.

```
my $reading_frame = shift;

# ...

my $offset = $reading_frame - 1;
for (my $i = $offset; $i < $len; $i += 3) {
    my $codon = substr($dna_string, $i, 3);
    push @codons, uc($codon);
}
```

7.2.3 `die` and the postfix form

As its name implies, the `die` builtin function stops the execution of the program. If it is passed a string as an argument, it first prints the string to the standard error stream before stopping execution. It is very useful for *abort*ing a program whenever user input does not satisfy the expectations (case 3).

```
die 'ABORT! Negative reading frames not yet implemented!'
    if $reading_frame < 0;

die 'ABORT! Reading frame must be one of (1, 2, 3)!'
    unless $reading_frame >= 1 && $reading_frame <= 3;
```

Here, both `if` and `unless` are placed after the statement they control and the conditional expression is not surrounded by parentheses. This **postfix form** reduces visual clutter and sounds more natural when reading the code aloud. However, it should be reserved for situations where the conditional code is brief.

By the way, if you carefully study the last chunk of code, you will notice that the two tests are partially overlapping. For example, a value of `-1` for `$reading_frame` would trigger the two `die` statements. The order in which they appear is thus important to provide an informative feedback to the user.

7.2.4 Interlude—defining multiline strings with the *heredoc* syntax

If the message to be printed includes multiple lines, you can use the **heredoc syntax** (`<<'EOT'`). More generally, this approach is handy whenever you need to define a multiline string in your code.

```
die <<"EOT";
Usage: $0 <dna-string> <reading-frame>
This tool translates DNA sequences to proteins using the standard genetic code.
It requires a DNA string and a reading frame (1, 2, 3, -1, -2, -3).
Example: $0 CATGAACTTCTTTGGCGTCTTGAT 1
EOT
```

The heredoc syntax is introduced by **double angle bracket characters** (<<). The quotes determine whether the heredoc has to be considered as a single-quoted (<<`'EOT'`) or double-quoted (<<`"EOT"`) multiline string, the latter being the default when omitting the quotes (<<`EOT`). Beware that recent versions of the `perl` interpreter (starting with v5.28.0) raise a compilation error when encountering a bare <<`EOT` (without quotes).

`EOT` is an arbitrary identifier (chosen by *you*) that `perl` interprets as the *ending delimiter* of the string. Here, we use `EOT` for *end-of-text*. Everything from the line following the <<`'EOT'` to the one before the line starting with `EOT` is part of the defined string (including the last newline character). Be careful, the ending delimiter must absolutely lie at the very *beginning* of the line to be taken into account.

Finally, the special variable `$0` (also known as **program name**) contains the path (directories and file name) to the current script, in our case `./translate_rf.pl`. Our heredoc has double quotes around `EOT`, so that the `$0` variable is correctly interpolated.

```
# single-quoted heredoc, no interpolation
print <<'EOT';
Usage: $0
EOT

# double-quoted heredoc, $0 is interpolated
print <<"EOT";
Usage: $0
EOT

# unquoted heredoc (defaults to double-quoted), $0 is interpolated
print <<EOT;
Usage: $0
EOT

# gives:

Usage: $0
Usage: ./translate_rf.pl
Usage: ./translate_rf.pl
```

It is good practice to print the special variable `$0` in usage messages to ensure that the computer uses the exact name by which the user has launched the program.

7.3 Looping directives

Looping directives allow us to repeat the execution of any chunk of code. They come in different flavors for maximum expressivity and legibility, but for most applications, using one or another form is often a matter of personal taste. We have already discussed the `for` loop before, p.43, so we will not come back to it now.

7.3.1 The while loop and its variants

The `while` loop is very simple. As long as the conditional expression evaluates to a true value, it repeats the block of code that follows. As in the `for` loop, the conditional expression must be surrounded by parenthesis characters and the code block delimited by curly brace characters. In `codon_quizz.pl`, we used a `while` loop to progressively consume the array `@questions`.

```
my @questions = shuffle keys %aa_for;
while (my $codon = shift @questions) {
    # loop body
}
```

This structure is a very common Perl idiom. The way it works is subtle:

- As long as the array `@questions` contains at least one element, `shift` puts a string into `$codon`. Since `$codon` evaluates to a true value in a boolean context, the loop keeps iterating.
- Eventually, `@questions` gets exhausted. At the next iteration, `shift` then returns `undef`, which evaluates to a false value and stops the loop. (Can you imagine a situation where this idiom would fail? Here's a hint: an array of natural numbers or of strings where some can be empty.)

BOX 10: Arrays and `while`/`shift` *vs. foreach-style* `for` loops

This use of the `while` loop is close to a *foreach-style* `for`... but with a twist. At the end of the `while` loop, the array is empty due to the repeated calls to `shift`, whereas in the `for` loop, it is left untouched, except if you alter the iterator variable (see "Iterator variables as aliases", p.43).

```
my @questions = shuffle keys %aa_for;
### array size: scalar @questions

for my $codon (@questions) {
    # loop body
}
### array size: scalar @questions

while (my $codon = shift @questions) {
    # loop body
}
### array size: scalar @questions

# gives:

### array size: 64

### array size: 64

### array size: 0
```

Note that here the `scalar` builtin function has the purpose of imposing scalar context to the smart comment lines, which leads to the display of the array size instead of its content.

Similar to the for loop, the conditional expression is evaluated before each iteration. This means that if the condition is false from the very beginning, the loop body *never* gets executed. In most cases, this behavior is perfectly reasonable.

```
# imagine we have built @questions from an empty file...

while (my $codon = shift @questions) {
    # this will never gets executed...
    # ... and that's perfect!
}
```

However, there exist situations where we would like to execute the loop body *at least once*. For these, the do {...} while loop is the way to go. Note that in this postfix form, the conditional expression does not need to be bracketed by parenthesis characters anymore.

```
# imagine a program for extracting hits from a BLAST report
# it lets us choose an E-value threshold and then displays the hit ids

my $answer;
do {
    # select E-value threshold
    say q{Here's your current selection...};
    # print the corresponding hit ids

    say 'Are you satisfied with your selection? (Y/N)';
    $answer = <>;
    chomp $answer;
    $answer = uc $answer;

} while $answer eq 'N';
```

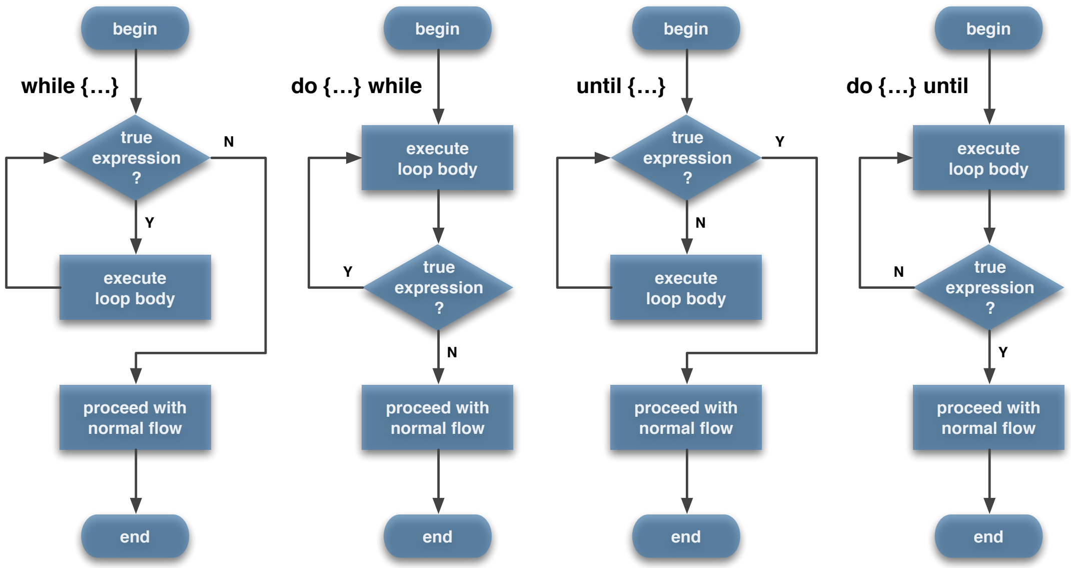

Figure 7.2: The four variants of the while loop

BOX 11: A personal account on programming styles

At the beginning of computing (in the sixties), there was the `goto` statement that allowed **BASIC** programmers to **jump** from places to places in their programs. This rapidly led to the so-called *spaghetti* programming style, in which control flow is impossible to follow by anyone but the programmer who writes the code (at least in the best cases; otherwise nobody can).

In 1968, Edsger W. Dijkstra (1930–2002), a famous Dutch computer scientist teaching at the Eindhoven University of Technology, published a landmark paper entitled *Go To Statement Considered Harmful*. Instead, Dijkstra and Niklaus Wirth (born in 1934), the Swiss computer scientist who invented the **PASCAL** language in 1970 at the ETH Zürich, advocated *structured programming*. In the most extreme forms of this ascetic style of programming, control flow directives are limited to `if/else` and `while`.

In the mid-eighties, I learnt to program in BASIC (using `goto`). During my studies in software engineering in the early nineties, I was trained to design programs in PASCAL, **COBOL** and **C** (among others) using only `if/else` and `while` for control flow. Even my `for` loops had to be written using the `while` directive!

In parallel, I learnt **x86 assembly** programming, which was a barely readable translation of binary code. For example, assembly language had no `while`, no `else` and made heavy use of `JMP`-like statements (the low-level equivalents of `goto`). You can imagine that I nearly became schizophrenic at that time…

Here's an excerpt of my good ol' `VGA-MAC` (https://orbi.uliege.be/handle/2268/80586). Observe the two jump statements (`JNC` and `JMP`, the second one implementing an *abort* pattern).

```
ParmsOk :   ADD     SI, 2                   ;|
            DEC     CX                      ;|
            MOV     DI, OFFSET FileName     ;+ Préparer le transfert
            MOV     NameLength, CX          ;+ NameLength <- Longueur du nom
            REP     MOVSB                   ;+ Copier le nom de fichier
            MOV     BYTE PTR [DI], 0        ;+ Ajouter un 0 à la fin du nom
            MOV     AX, 3D00h               ;|
            MOV     DX, OFFSET FileName     ;|
            INT     21h                     ;+ Ouvrir le fichier
            JNC     FileOk                  ;+ Fichier ouvert -> On continue
            MOV     Msg, OFFSET NoFileMsg   ;+ Fixer Msg sur NoFileMsg
            JMP     EndVGAMAC               ;+ Terminer prématurément
```

Then, I met (Modern) Perl and it all fell into place. My own programming style still evolves at the margin, but its main characteristics do not. It could be described as an *object-oriented structured programming* style that *funnels the execution flow through the default path using just-in-time adaptation* and *handles exceptions via multiple exit points*. This might look pedantic, but it is not. Instead, it is a powerful and scalable yet legible and maintainable programming style, and this the one I teach you in this course.

Just like `unless` is the negated form of the `if` branching directive, `until` is the negated form of the `while` looping directive. Consequently, an `until` loop keeps iterating as long as its conditional expression evaluates to false and stops when it becomes true. Using it or not is a matter of taste.

```
do {
    # ...
    say 'Are you satisfied with your selection? (Y/N)';
    # ...
} until $answer eq 'Y';
```

As you can see, the `do {...} until` form also exists. To sum up, there are thus four variants of the basic `while` loop. For easy comparison, they are all illustrated above.

7.3.2 Loop control directives

We don't have enough time to cover the *object-oriented* aspect of my programming style. However, I have already begun to present you the *structured* part and the concept of a *default path* enforced by *just-in-time adaptation*. Hence, when discussing the three uses for branching directives, p.61, the *fix-and-proceed* strategy (case 1) was an application of this philosophy, whereas the *abort* pattern (case 3, using the `die` builtin function) was an illustration of *exception handling via multiple exit points*.

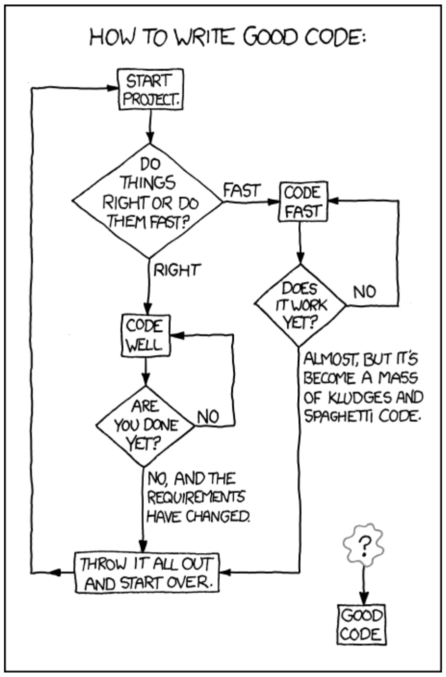
HOW TO WRITE GOOD CODE:
START PROJECT.
DO THINGS RIGHT OR DO THEM FAST?
FAST
CODE FAST
DOES IT WORK YET?
NO
ALMOST, BUT IT'S BECOME A MASS OF KLUDGES AND SPAGHETTI CODE.
RIGHT
CODE WELL
ARE YOU DONE YET?
NO
NO, AND THE REQUIREMENTS HAVE CHANGED.
THROW IT ALL OUT AND START OVER.
?
GOOD CODE

Figure 7.3: How to write good code? [xkcd.com]

An example of the latter is found in the `while` loop of `codon_quizz.pl`.

```
QUESTION:
while (my $codon = shift @questions) {

    # ask question and wait for answer
    # ...

    # leave early if asked to do so
    last QUESTION if $answer eq 'EXIT';
    # check answer
    my $aa = $aa_for{$codon};
    if ($answer eq $aa) {
        say 'Correct!';
        $score++;               # increment score
    }
    else {
        say "Wrong! Answer was $aa!";
    }

    # track number of questions
    $total++;
}
```

As you can see, there are two **exit points** in this loop:

1. Iteration stops normally when the array `@questions` is exhausted.
2. Iteration stops early when `$answer` is equal to `EXIT`.

The second exit point is enabled by the `last` keyword. It has the effect of immediately leaving the loop labelled by its argument (here, `QUESTION`). The **loop label** must be an identifier in uppercase, followed by a single colon character (`:`).

Though the loop label is optional, I highly recommend using it to make your intention as clear as possible. Moreover, leaving a blank line above the label improves legibility.

Without `last`, the code above should have been written using an `$exit` **boolean flag**. I don't know for you, but I find this ugly… Look especially at the conditional expression governing the `while` loop and at the indentation depth of the code checking the answer.

```
my $exit;
while ( (my $codon = shift @questions) && !$exit ) {

    # ask question and wait for answer
    # ...

    if ($answer ne 'EXIT') {
        # check answer
        my $aa = $aa_for{$codon};
        if ($answer eq $aa) {
            say 'Correct!';
            $score++;           # increment score
        }
        else {
            say "Wrong! Answer was $aa!";
        }
        # track number of questions
        $total++;
    }
    else {
        # leave early if asked to do so
        $exit = 1;
    }
}
```

Basically, with this older programming style, whenever we have an additional **exception** to handle, we need to add one boolean flag and to increase the depth of our interesting code by burying it under a pile of nested `if/else` branching directives.

A very *Perlesque* way to avoid the use of a boolean flag here would be to empty the array `@questions` in the outer `else` block using `@questions = ();`. Hence, no need to change the loop condition.

`last` is only one of the three **loop control** keywords (technically builtin functions). The two other ones are `next` and `redo`. While `next` is extremely useful to immediately begin the next loop iteration, you should not use `redo` because it *goes back to* the beginning of the current iteration without re-evaluating the conditional expression (which is potentially as harmful as a regular `goto`).

Finally, `while` and `for` loops can be completed by a `continue` block that is executed after each iteration (akin to the third subexpression of the *C-style* `for` loop).

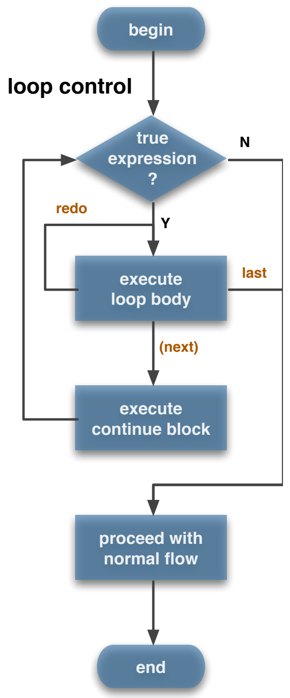

Figure 7.4: Loop control directives

To see working examples of `next` and `continue`, consider `codon_quizz_jokers.pl`, an improved program in which the player is offered three jokers that allow her to skip the codons she doesn't know without losing points! If you want to try it, you only need to add (or edit) a few lines of code in the current version of the program. (For the laziest among you, the required changes are highlighted and discussed just after the listing.)

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

# avoid boilerplate
use Modern::Perl '2011';
use List::Util 'shuffle';

# use Smart::Comments;      # disabled by default; when debugging use
                            # perl -MSmart::Comments <script.pl>

# standard genetic code definition from NCBI gc.prt file
my $aa = 'FFLLSSSSYY**CC*WLLLLPPPPHHQQRRRRIIIMTTTTNNKKSSRRVVVVAAAADDEEGGGG';
my $b1 = 'TTTTTTTTTTTTTTTTCCCCCCCCCCCCCCCCAAAAAAAAAAAAAAAAGGGGGGGGGGGGGGGG';
my $b2 = 'TTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGG';
my $b3 = 'TCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAG';

# build hash for standard code
my %aa_for;
my $codon_n = length $aa;
for my $i (0..$codon_n-1) {
    my $codon = substr($b1, $i, 1) . substr($b2, $i, 1) . substr($b3, $i, 1);
    $aa_for{$codon} = substr($aa, $i, 1);
}
### %aa_for

print <<'EOT';
Try to guess the correct amino acid for each codon.
STOPs are symbolized by an asterisk (*).
You start with 3 jokers [***] that you can use to skip questions.
Type SKIP for skipping a question and EXIT for leaving the quizz.
Good luck!
EOT

# assemble pool of questions from hash keys
my @questions = shuffle keys %aa_for;
my $score = 0;
my $total = 0;
my $quest_n = 1;
my $joker_n = 3;

# loop through questions asking for correct answer

QUESTION:
while (my $codon = shift @questions) {

    # ask question and wait for answer
    print "$quest_n. $codon [" . ('*' x $joker_n) . '] : ';
    my $answer = <>;
    chomp $answer;
    $answer = uc $answer;       # make our quizz robust

    # skip current question if asked to use a joker
    if ($answer eq 'SKIP') {
        if ($joker_n > 0) {
            $joker_n--;
            next QUESTION;
        }
        say q{You don't have any joker left!};
        $answer = 'EXIT';
    }

    # leave early if asked to do so
    last QUESTION if $answer eq 'EXIT';

    # check answer
    my $aa = $aa_for{$codon};
    if ($answer eq $aa) {
        say 'Correct!';
        $score++;               # increment score
    }
    else {
        say "Wrong! Answer was $aa!";
    }

    # track number of (answered) questions
    $total++;
}

# track (total) number of questions
continue {
    $quest_n++;
}

say "Your final score is $score on $total";
```

First, we need to update the rules of the game.

```
print <<'EOT';
...
You start with 3 jokers [***] that you can use to skip questions.
Type SKIP for skipping a question and EXIT for leaving the quizz.
...
```

Second, we need two new **counters** in addition to `$total`: `$quest_n` will store the number of the current question and `$joker_n` will store the number of jokers not yet used.

```
my $total = 0;
my $quest_n = 1;
my $joker_n = 3;
```

Third, we need a new block in the `while` loop to implement the jokers.

```
# ask question and wait for answer
print "$quest_n. $codon [" . ('*' x $joker_n) . '] : ';
# ...

# skip current question if asked to use a joker
if ($answer eq 'SKIP') {
    if ($joker_n > 0) {
        $joker_n--;
        next QUESTION;
    }
    say q{You don't have any joker left!};
    $answer = 'EXIT';
}

# leave early if asked to do so
last QUESTION if $answer eq 'EXIT';
```

If the user types in `SKIP`, she can leave the current question unanswered, but at the expense of one joker. If she tries to skip a question after having exhausted all her jokers, the game ends as if she had typed in `EXIT`. Observe how we implement this complex behavior in a very clean and concise way using a fix-and-proceed approach based on the update of the `$answer` variable.

If you wonder how we print the number of jokers remaining, examine the first statement above, which takes advantage of the **repetition operator** (`x`). This handy trick is fully explained in "The repetition operator", in the second part of this course.

Fourth, we need a `continue` block to keep track of the number of questions asked so far, whether answered or skipped by the user. This allows us to decouple this value (`$quest_n`) from the potential maximal score (`$total`).

```
# track (total) number of questions
continue {
    $quest_n++;
}
```

This works because the `continue` block is executed even when the iteration is interrupted by the `next` keyword, but not by `last` or `redo`. For this reason, I advise you to avoid `continue` blocks, as they make your code more difficult to understand.

BOX 12: redo (for those who insist on using it)

Above, we said that trying to skip a question without any joker left was equivalent to typing in `EXIT`. An alternative possibility is to simply refuse the user request and *go back to* the current question. Such a behavior is a typical use case for the `redo` keyword.

```
# skip current question if asked to use a joker
if ($answer eq 'SKIP') {
    if ($joker_n > 0) {
        $joker_n--;
        next QUESTION;
    }
    say q{You don't have any joker left!};
    # $answer = 'EXIT';         # old behavior
    redo QUESTION;              # new behavior
}
```

Interestingly, one can achieve the same behavior without using `redo`: simply wrap this chunk in a `do {...} until` loop that enforces the user to answer. The code is a bit heavier because it requires to declare `$answer` before the loop.

```
# ask question and wait for answer
my $answer;
do {
    print "$quest_n. $codon [" . ('*' x $joker_n) . '] : ';
    $answer = <>;
    chomp $answer;
    $answer = uc $answer;       # make our quizz robust

    # skip current question if asked to use a joker
    if ($answer eq 'SKIP') {
        if ($joker_n > 0) {
            $joker_n--;
            next QUESTION;
        }
        say q{You don't have any joker left!};
    }
} until $answer ne 'SKIP';
```

Chapter 8

Other novelties in `codon_quizz.pl`

8.1 `keys` & `shuffle`

```
my @questions = shuffle keys %aa_for;
```

Perl can be a very concise language. For example, a rough English translation of the line above would be: *"Build a list with the keys (i.e., codons) of the hash `%aa_for`, randomize their order and put the shuffled codon list into a new array named `@questions`."*

As it names suggests, `keys` is a builtin function that returns a list consisting of the keys of the specified hash. In a scalar context, `keys` returns the number of key / value pairs in the hash.

```
my $codon_n = keys %aa_for;
### $codon_n

# gives:

### $codon_n: 64
```

Concerning `shuffle`, it is a function that we **imported** from the **module** `List::Util`.

```
use List::Util 'shuffle';
```

While this module is part of the standard Perl distribution, its evil twin, `List::MoreUtils`, is not. Both modules are replete with functions for working with lists that are really worth exploring (e.g., `max`, `all`, `mesh`, `uniq`). However, remembering which function is in which module can be quite difficult. That is why a Perl guru decided to merge these two modules into an unified module aptly named `List::AllUtils`, which, *sigh*, is not part of the standard Perl distribution. Let's remedy it!

```
$ cpanm List::AllUtils
```

And if you get bored during the week, have a look at the corresponding documentation…

```
$ perldoc List::AllUtils
```

BOX 13: The hash random order in gory detail

In list context, hash keys are returned in an *apparently random* order. The actual random order is specific to a given hash and (since `perl` 5.18.0) to a given a run of your script. This is so to prevent an ill-intended hacker to devise a way to bring Perl scripts to their knees using so-called *algorithmic complexity attacks*. For more information about this, see the corresponding topic in: http://perldoc.perl.org/perlsec.html

The exact same series of operations on two hashes may result in a different order for each hash. However, any *insertion* into the hash may change the order (as will any *deletion*, with the exception that the most recent key returned by `each` or `keys` may be deleted without changing the order). So long as a given hash is unmodified you may rely on `keys`, `values` (see "`values` & `sort` and reuse of variable names", p.110) and `each` (see "`each` and list assignment", p.121) to repeatedly return the same order as each other.

Therefore, this means that we don't actually need `shuffle` to get a differently ordered list of codons each time we launch `codon_quizz.pl`. However, this behavior is quite new (with `perl` 5.16, one always gets the same list of codons). Further, explicitly using `shuffle` makes our intention clear to anyone reading our code. That is why we used it in the first place.

Finally, if you want to sample codons with replacement, you can use the aptly named `rand` builtin function, which is meant to return random numbers.

```
# ask 10 random questions (possibly twice the same)
for my $i (1..10) {
    my $codon = $questions[ int(rand(@questions)) ];
    ...
}
```

For more information and examples, see: http://perldoc.perl.org/functions/rand.html

8.2 Reading from the standard input stream

8.2.1 The `readline` operator

```
my $answer = <>;
chomp $answer;
$answer = uc $answer;        # make our quizz robust
```

This code chunk uses the **readline operator** (<>), which reads an **input file** (**infile** for short) line by line. If you prefer, you can use the `readline` builtin function instead of <>. Personally, I don't.

```
my $answer = readline;
```

When used alone as here, the readline operator reads from the **standard input stream**, which is the keyboard, except if a shell redirection is enabled (with <). Thus, our program expects user input from the keyboard, but could also work had the user pre-recorded all its answers in an input file.

8.2.2 Line endings

Lines are delimited by the character contained in the special variable `$/`, also known as the **input record separator**. By default, `$/` is set to the newline character (`\n`), which traditionally separates consecutive lines on UNIX systems, such as Linux and macOS. In contrast, Classic Mac OS uses the **carriage return character** (`\r`), whereas Windows uses the sequence of these two characters (`\r\n`).

BOX 14: How to fix incorrect line endings?

Though in principle the `perl` interpreter should transparently take care of the various platform-dependent **line endings**, Windows-formatted files processed on Linux are a very common source of deeply perplexing bugs.

Therefore, if you find yourself scratching your head because a Perl script behaves weirdly on an input file having transited by a Windows computer, try fixing its line endings. This can be done with the following Perl **one-liner**, which generates a new file using the original name and appends the `.bak` suffix to the original file (see "Perl *one-liners*", p.145 to learn how they work).

```
# convert Windows line endings to Linux line endings
$ perl -i.bak -ne 's/\r\n/\n/g; print;' infile.txt
```

If the file was indeed badly (i.e., Windows) formatted, the size of the new file should be slightly smaller than the size of the original file.

```
$ ll infile.txt*

-rw-r--r--  1 denis  admin     42504 Oct 11 22:40 infile.txt
-rw-r--r--  1 denis  admin     43326 Oct 10 11:25 infile.txt.bak
```

Below are similar one-liners for performing the other possible line-ending conversions. Keep them handy as they might prove useful one day.

```
# Linux to Windows
$ perl -i.bak -ne 's/\n/\r\n/g; print;' infile.txt

# Classic Mac OS to Linux
$ perl -i.bak -ne 's/\r/\n/g; print;' infile.txt

# Linux to Classic Mac OS
$ perl -i.bak -ne 's/\n/\r/g; print;' infile.txt
```

The next line calls the `chomp` builtin function. Its purpose is to remove any trailing string that corresponds to the current value of `$/`. Removing the newline character is the first thing to do on any user input, even when reading from the keyboard. Failing to do so is bound to misery. Here, our `$answer` would end by a newline character and, for that, would never be considered as a valid amino acid.

Note that older Perl programs often use the `chop` builtin function instead of `chomp`. *That not a good idea™*. Reason is that `chop` always removes the last character of the string, even if it is not equal to the `$/` variable. The risk thus exists to remove something that should not be removed.

BOX 15: Soft *vs.* hard-wrapping

It seems that is a good place to explain the difference between **hard wrapping** and **soft wrapping** when formatting text. A paragraph is said to be *hard wrapped* if it contains *hard* line breaks (encoded by newline characters) at the end of each line. In contrast, it is *soft wrapped* when it contains no line break other than the one introduced by the newline character occurring at the very end of the paragraph.

Soft-wrapped text adapts to the current width of the window. One says that it dynamically *reflows*. By default, some text editors introduce hard line breaks upon editing, which can unexpectedly breaks otherwise perfectly valid code. This is the case of `nano`. To avoid this undesirable behavior, you should always invoke this editor with the `-w` option that disables hard wrapping.

```
$ nano -w script.sh
```

Chapter 9

A first look at operators

9.1 What are operators?

An **operator** is a series of one or more **symbol characters** used as part of the syntax of a language. Perl operators are of different *types* (*numeric*, *string*, *logical*, *bitwise* and *special*).

Operators are further characterized by four properties:

1. The **precedence** of an operator governs when Perl should evaluate it in an expression. Evaluation *order* proceeds from highest to lowest precedence. This corresponds to the common knowledge that multiplications have priority over additions in mathematical formulas. The evaluation order can be changed by bracketing specific parts of the expression with parenthesis characters.

   ```
   ### a: 3 * 2 + 5
   ### b: (3 * 2) + 5
   ### c: 3 * (2 + 5)

   # gives:

   ### a: 11
   ### b: 11
   ### c: 21
   ```

2. The **associativity** of an operator governs whether it evaluates from left to right or right to left. For example, addition is left associative, whereas exponentiation is right associative. Again, parentheses help making your intention clear.

   ```
   ### a: 2 ** 2 ** 3
   ### b: 2 ** (2 ** 3)
   ### c: (2 ** 2) ** 3

   # gives:

   ### a: '256'
   ### b: '256'
   ### c: '64'
   ```

3. The **arity** of an operator is the number of **operands** on which it operates. A *nullary* operator operates on zero operands, a *unary* operator on one operand, a *binary* operator on two operands, a *ternary* operator on three operands, and a *listary* operator on a list of operands.

4. The **fixity** of an operator is its position relative to its operands. Operators are *infix* operators when appearing between their operands, *prefix* operators when preceding their operands, and *postfix* operators when following their operands. Moreover, *circumfix* operators surround their operands, whereas *postcircumfix* operators follow certain operands and surround others.

   ```
   ### infix: $codon_n - 1
   ### prefix: -$reading_frame
   ### postfix: $total++
   ### circumfix: q{Watson didn't propose the central dogma.}
   ### postcircumfix: $aa_for{$codon}
   ```

9.2 Comparison operators

Let's sort out the different **comparison operators** we have met so far. All are infix operators and thus appear between their operands.

```
# example of numeric equality '=='
unless (@ARGV == 2) {
    # ...
}

# examples of numeric less than '<'
for (my $i = 0; $i < $len; $i += 3) {
    # ...
}
die 'ABORT! Negative reading frames not yet implemented!'
    if $reading_frame < 0;

# example of numeric greater/less than or equal to '>=' and '<='
die 'ABORT! Reading frame must be one of (1, 2, 3)!'
    unless $reading_frame >= 1 && $reading_frame <= 3;

# examples of string equality
last QUESTION if $answer eq 'EXIT';
if ($answer eq $aa) {
    # ...
}
```

Comparison operators automatically impose boolean context on the result of the expression. This fits well with control flow directives that all impose boolean context as well. However, you must specify the value context for the comparison itself: numeric or string comparison.

When comparing strings, *less than* means *before in lexical order* and *greater than* means *after in lexical order*. Be careful that the actual sorting order can depend on the human language you use on your system (known as the `locale`).

```
'abc' lt 'def'            # true
'abc' lt 'abd'            # true
'abc' lt 'abcde'          # true
'abc' lt 'abb'            # false
'abc' gt 'abb'            # true
```

Table 9.1: Comparison operators for numeric and string contexts

operator name	fixity	numeric cont.	string cont.
equality	infix	==	eq
inequality	infix	!=	ne
greater than	infix	>	gt
less than	infix	<	lt
greater than or equal to	infix	>=	ge
less than or equal to	infix	<=	le

9.3 Logical operators

Logical operators impose boolean context on the result of the expression but also on their operand(s). They combine (*and*, *or*) or negate (*not*) other boolean expressions, which is useful but error-prone. The two first ones are infix operators, while logical *not* is a prefix operator appearing before its operand.

Table 9.2: Logical operators of high and low precedence

operator name	fixity	high prec.	low prec.
and	infix	&&	and
or	infix	||	or
not	prefix	!	not

Mastering the interplay between branching directives on the one hand and comparison and logical operators on the other hand requires a good understanding of **boolean algebra**.

```
# these four code chunks are all equivalent

die 'ABORT! Reading frame must be one of (1, 2, 3)!'
    unless $reading_frame >= 1 && $reading_frame <= 3;

die 'ABORT! Reading frame must be one of (1, 2, 3)!'
    unless $reading_frame >  0 && $reading_frame <  4;

die 'ABORT! Reading frame must be one of (1, 2, 3)!'
        if $reading_frame <  1 || $reading_frame >  3;

die 'ABORT! Reading frame must be one of (1, 2, 3)!'
        if $reading_frame <= 0 || $reading_frame >= 4;
```

9.4 Operator precedence and associativity

All three logical operators exist in both **word form** and **punctuation form**, the former ones having lower precedence than the latter ones. Operator precedence and associativity are quite complex but nearly dispensable topics that boil down to the order in which evaluation proceeds. The reason why they are dispensable is that you can force your own order of evaluation by using parenthesis characters, as demonstrated in the code below.

```
# let's say we have the coordinates of two DNA fragments
# how to determine whether these fragments overlap?

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

# using high-precedence logical operators
if (   ($y1 >= $x1 && $y1 <= $x2)           #  left cases
    || ($x1 >= $y1 && $x1 <= $y2) ) {       # right cases
    say 'fragments x and y overlap';
}

# mixing high- and low-precedence logical operators
if (    $y1 >= $x1 && $y1 <= $x2            #  left cases
    or  $x1 >= $y1 && $x1 <= $y2  ) {       # right cases
    say 'fragments x and y overlap';
}

# ... or more intelligently...
say 'fragments x and y overlap'
    unless $y1 > $x2 || $y2 < $x1;          # test disjunction
```

Chapter 10

Using Perl to compute some stats

10.1 A killer app with a fast bite

To introduce input file processing and to explore Perl's handling of numbers, we continue our rewriting *spree* of bioinformatics applications. Our third program again uses the standard genetic code, but this time to compute *codon usage* statistics over a series of input sequences. The equivalent web app can be found here: `http://www.bioinformatics.org/sms/codon_usage.html`

To test our new killer app, follow the instructions below.

1. Open a blank text document, copy the listing in the next section and save it as `codon_usage.pl`.
2. Download the complete set of cDNA sequences for the model organism *Escherichia coli* K-12 MG1655 from the *Ensembl Bacteria* web portal. Go to: `http://bacteria.ensembl.org/` and follow the links there or enter the following line in your terminal. Note that the **line continuation character** (\) is required if you do not type the whole URL on a single line (without spaces).

```
$ wget ftp://ftp.ensemblgenomes.org/pub/bacteria/release-20/fasta/\
    bacteria_22_collection/escherichia_coli_str_k_12_substr_mg1655/cdna/\
    Escherichia_coli_str_k_12_substr_mg1655.GCA_000005845.1.20.cdna.all.fa.gz
$ curl -O ftp://ftp.ensemblgenomes.org/...      # macOS
```

3. Unpack the downloaded file using `gunzip`.

```
$ gunzip Escherichia_coli_str_k_12_substr_mg1655.GCA_000005845.1.20.cdna.all.fa.gz
```

4. For convenience, create a **symbolic link** to the unpacked file.

```
$ ln -s Escherichia_coli_str_k_12_substr_mg1655.GCA_000005845.1.20.cdna.all.fa \
    Ecoli_cds.fasta
```

5. Install a new module from CPAN.

```
$ cpanm autodie
```

6. Make your program executable, launch it and look at its output.

```
$ chmod a+x codon_usage.pl
$ ./codon_usage.pl Ecoli_cds.fasta Ecoli_cds.usage
$ less Ecoli_cds.usage
```

10.2 The code for our own codon_usage

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

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

use Smart::Comments;

unless (@ARGV == 2) {
    die <<"EOT";
Usage: $0 <infile.fasta> <outfile.txt>
This tool computes codon usage over a series of DNA sequences.
It requires a FASTA file as input and the name of the output file to be created.
Example: $0 Ecoli_cds.fasta Ecoli.usage
EOT
}

my $infile  = shift;
my $outfile = shift;

### Reading input file: $infile

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

my $seq_id;
my $seq;
my %seq_for;

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;

close $in;

### Building hash for standard code...

# standard genetic code definition from NCBI gc.prt file
my $aa = 'FFLLSSSSYY**CC*WLLLLPPPPHHQQRRRRIIIMTTTTNNKKSSRRVVVVAAAADDEEGGGG';
my $b1 = 'TTTTTTTTTTTTTTTTCCCCCCCCCCCCCCCCAAAAAAAAAAAAAAAAGGGGGGGGGGGGGGGG';
my $b2 = 'TTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGG';
my $b3 = 'TCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAG';

my %aa_for;
my $codon_n = length $aa;
for my $i (0..$codon_n-1) {
    my $codon = substr($b1, $i, 1) . substr($b2, $i, 1) . substr($b3, $i, 1);
    $aa_for{$codon} = substr($aa, $i, 1);
}

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

my %count_for;
my %total_for;

for my $dna_string (values %seq_for) {      ### Elapsed time |===[%]
    my $len = length $dna_string;

    # split seq into codons...
    # ... and count their occurrences...
    # ... and count total occurrences of each amino acid

    CODON:
    for (my $i = 0; $i < $len; $i += 3) {
        my $codon = uc substr($dna_string, $i, 3);

        last CODON if length $codon < 3;    # skip truncated codons

        $count_for{         $codon  }++;
        $total_for{ $aa_for{$codon} }++;
    }
}

```

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

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

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

close $out;          # useless because files are automatically closed
                     # when the filehandle variable gets out of scope

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

Homework

1. Try to understand the Perl novelties appearing in this new program.
2. Write a program (`hw3_check_overlap.pl`) to verify that the three code chunks in the section *Operator precedence and associativity* actually do give the same result. The basic strategy for testing its behavior would be to repeatedly invoke your program with *N* different sets of coordinates.

   To do this, ask for four values in turn (`x1`, `x2`, `y1`, `y2`) using the readline operator (<>).

   ```
   $ ./hw3_check_overlap.pl
   ```

   ```
   10
   40
   20
   60
   10-40 and 20-60 DO overlap
   ```

   If you want to process more than one set of four values, you will need a `while` loop checking that new values are indeed typed in by the user. To quit your program smoothly, press the keys `Ctrl` and `D` (abbreviated as `^D` and known as **end-of-file**). This key combo makes the <> operator to return an `undef` value that can stop the loop.

   It is also possible to take advantage of shell redirection to avoid typing in your test values again and again. For this, create a suitable input file (with 4*N* values) and pass it to your program. You can even simultaneously redirect its output into an output file.

   ```
   $ cat input.txt
   ```

   ```
   10
   40
   20
   60
   100
   120
   80
   90
   ```

   ```
   $ ./hw3_check_overlap.pl < input.txt > output.txt
   $ cat output.txt
   ```

   ```
   10-40 and 20-60 DO overlap
   100-120 and 80-90 DO NOT overlap
   ```

Part IV

Lesson 4

Chapter 11

Input/output in Perl

11.1 Reading files

11.1.1 FASTA format

`codon_usage.pl` is our first program with the ability to read a file. And not just any file but a FASTA file! In spite of its boring appearance, the *FASTA format* is not easy to process, precisely because of its loose specification. Hence, a given sequence *may* span several lines and ends *either* with the *definition line* of the next one (the > followed by a unique identifier) *or* with the end of the file.

Thanks to loop control, however, our code for reading a FASTA file remains relatively straightforward. Its logic is illustrated below. Observe how we need to store the last sequence as a special case due to the two ways of ending a sequence. Moreover, note that empty sequences are automatically excluded.

11.1.2 `open` & `close`

Apart from the file opening/closing statements, there are few new Perl concepts in the corresponding block of code. In *Modern Perl*, the `open` builtin function takes three arguments:

1. a lexically-scoped variable that acts as the **filehandle reference** (we will start covering references themselves in "A gentle introduction to references", in the second part of this course),
2. an **opening mode**, of which the most common ones are `'<'` for *reading*, `'>'` for *writing* and `'>>'` for *appending* to an existing file,
3. a string expression giving the **file name** (optionally including its path).

The `close` builtin function closes the file associated to the filehandle reference. It is rarely needed in well-designed programs using functions (see the second part of this course) and filehandles that have lexical scope because files are automatically closed when the filehandle variable gets out of scope.

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

my $seq_id;
my $seq;
my %seq_for;
```

```
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;

close $in;
```

11.1.3 autodie and $!

Many Perl courses explain that you should ensure that the input file exists and can be read by testing the return value of the open builtin function, and then printing the special variable `$!` if it has a false value. We don't because we use the `autodie` pragma, which takes care of all of that (and many other errors) for us. `autodie` has just to be loaded once with a `use` directive at the beginning of our program.

```
# without autodie
# but using short-circuiting (or...) and the special Perl variable $!
open my $in, '<', $infile
    or die "Can't open '$infile' for reading: '$!'";

# with autodie
use autodie;
open my $in, '<', $infile;
```

Both code chunks give the same output.

```
$ ./codon_usage.pl missing.fasta missing.usage
```

```
Can't open 'missing.fasta' for reading: 'No such file or directory'
    at ./codon_usage.pl line 22
```

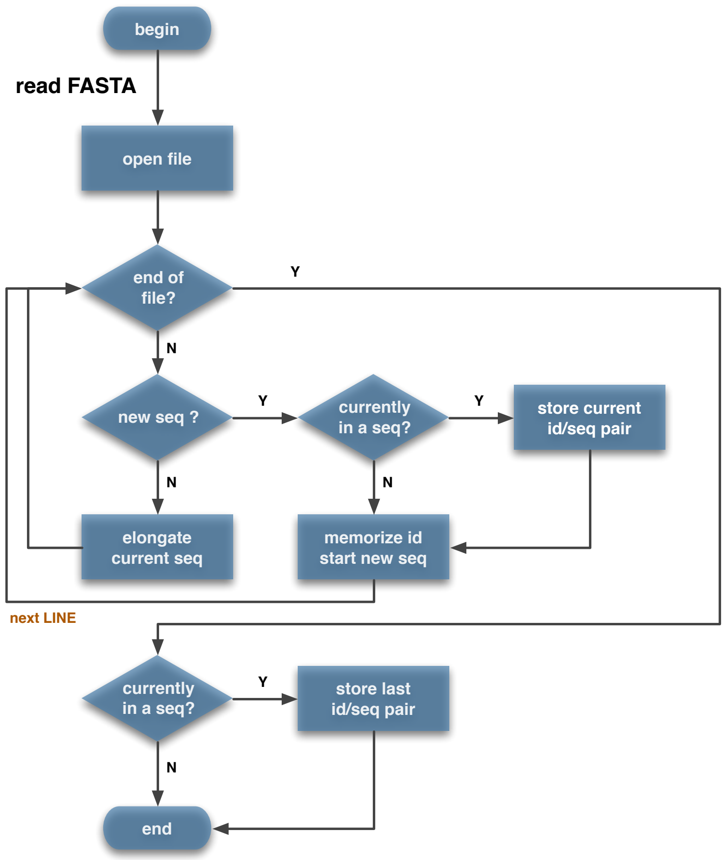

Figure 11.1: Flowchart of our FASTA file reader

11.2 Writing files

At the end of the analysis, `codon_usage.pl` produces a table reporting the frequency of each codon relatively to the total number of codons specifying each amino acid. This table is directly written into an **output file** (**outfile** for short), the name of which is specified by the user.

```
my $outfile = shift;

# later in the file...
open my $out, '>', $outfile;
```

Observe the opening mode of the outfile (`'>'`), which tells `perl` that we want to write to the file. If the file does not exist, it is *created*, otherwise, it is *overwritten*. Of course, we need to have **write permission** on the corresponding directory (and on the file if it already exists).

The next line of code implements a common pattern when producing **tabular files** in Perl. It begins by writing a **header line** that starts with a comment character (#).

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

Using such a header is a weak convention but it improves the autodocumentation of output files. Since specialized text-processing programs often ignore comment lines when processing their input files, this practice generally does not harm. That is why I suggest you to adopt it.

To write to a file, insert the variable containing the filehandle reference between the `print` or `say` builtin functions and the list of things to be written.

In *Modern Perl*, it is recommended to enclose the filehandle between curly brace characters for clarity.

The remaining of the line is also idiomatic. It first defines a list of column headers using the **quoted word operator** (`qw(...)`), which splits a literal string on **whitespace characters** to produce a list of strings. Then, it immediately `join`s the values of the list using the tab character (`\t`) as the separator.

This is a case where `say` is extremely convenient. With `print`, the line must include an extra pair of parenthesis characters. Otherwise, the newline character (`\n`) ends up part of the column headers.

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

Then comes a *foreach-style* `for` loop that iterates over the codons, computes their usage and prints it to the outfile. Again, we use a `say` combined to a call to `join` to generate our multi-column tabular file. The mysterious `sprintf` syntax will be covered in “`sprintf`”, in the second part of this course.

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

```
close $out;         # useless because files are automatically closed
                    # when the filehandle variable gets out of scope
```

Again, the `close` builtin function is not required here. Remember: at the end of the program file, variables with a file scope get out of scope, and the corresponding files are automatically closed.

Chapter 12

A first look at mathematics in Perl

12.1 Perl values: Numbers

Beyond strings, Perl handles other kinds of values, including various kinds of numbers. Internally, there exist two types of numbers, each one associated to a specific way of doing arithmetic operations:

- **integer numbers** (i.e., round numbers),
- **floating-point numbers** (i.e., decimal numbers).

When you need to input numbers in a program, you can use different notations, some of them corresponding to less familiar **base systems**. Indeed, along the **decimal system** (based on 10 digits because we have ten fingers), computer languages support three other systems:

- the **binary system** (2 digits, `0` and `1`),
- the **octal system** (8 digits, `0` to `7`),
- the **hexadecimal system** (16 digits, `0` to `9` and `A` to `F`).

The latter three systems are only used for inputting integers, even if, internally, floating-point numbers are also represented in binary form.

```
# examples
my $integer   = 46;           # number of human chromosomes
my $float     = 3.1416;       # pi
my $sci_float = 3.08e9;       # human genome size: 3,080,000,000 bp
my $binary    = 0b101110;     # number of human chromosomes: 46
my $octal     = 056;          # number of human chromosomes: 46
my $hex       = 0x2E;         # number of human chromosomes: 46
```

Hexadecimal numbers are quite common in computing, for example to specify **RBG colors** in a compact way (try searching for `#6a5acd` in Google). In contrast, octal numbers are not very much used these days. A notable exception are **file permissions** in UNIX-like operating systems. For example, `chmod 644` is the octal representation of `110100100`, which reads as *read/write for user, read for group and read for others*. Note that the latter are the default permissions for new files.

BOX 16: Integer and floating-point arithmetics

In `codon_usage.pl`, when computing codon usage frequencies, we want them to be expressed as percentages. That is why we multiply the usage ratio by `100.0`. In older scripting languages, such as the `bash` shell, mathematical operations default to **integer arithmetic**, which can lead to unexpected results, for example due to aggressive rounding by the integer division.

```
$ echo 9/4 = $((9/4))
```

```
9/4 = 2
```

It is not the case in Perl, but for clarity, I always add a trailing `.0` to the integer numbers that I want to consider as decimal. However, you can force integer arithmetic if you need it by using the `integer` pragma.

```
### 9/4: 9/4

use integer;
### 9/4: 9/4

# gives:

### 9/4: '2.25'
### 9/4: 2
```

For more information about the binary representation of numbers in Perl (and other languages), see: `https://floating-point-gui.de/languages/perl/`

12.2 Numeric and in-place operators

Another interesting bit in the FASTA file reader is the use of the **in-place concatenation operator** (`.=`), a variant of the concatenation operator (`.`) that *appends* its second operand to its first operand.

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

All **in-place operators** display the same behavior: they replace their first operand by the result of the corresponding operation. We have already encountered **in-place addition** (`+=`) in the *C-style* `for` loop and **in-place division** (`/=`) in our switch table handling alternatives with more than two options.

```
# the three following couples of lines are equivalent

$seq .= $line;
$seq = $seq . $line;

for (my $i = 0; $i < $len; $i += 3) { ... }
for (my $i = 0; $i < $len; $i = $i + 3) { ... }

$size /= $div;
$size = $size / $div;
```

The table below lists the numeric operators and their in-place variants.

Table 12.1: Regular and in-place numeric operators

operator name	fixity	regular	in-place
addition	infix	`+`	`+=`
subtraction	infix	`-`	`-=`
multiplication	infix	`*`	`*=`
division	infix	`/`	`/=`
exponentiation	infix	`**`	`**=`
modulo	infix	`%`	`%=`

You should always use these operators when you *update* a variable because they clarify your intent.

Two other interesting in-place numeric operators are the **auto-increment operator** (`++`) and the **auto-decrement operator** (`--`), which respectively *adds one to* or *subtracts one from* the affected variable. We have already met both of them in `codon_quizz.pl` and `codon_quizz_jokers.pl`.

```
# in both codon_quizz.pl and codon_quizz_jokers.pl
if ($answer eq $aa) {
    say 'Correct!';
    $score++;                # increment score
}                            # equivalent to: $score = $score + 1

# in codon_quizz_jokers.pl
if ($joker_n > 0) {
    $joker_n--;              # decrement number of jokers
    next QUESTION;           # equivalent to: $joker_n = $joker_n - 1
}
```

BOX 17: Prefix *vs.* postfix forms of the auto-... operators

There exist both prefix and postfix forms of the auto-... operators that differ as to *when* the expression is evaluated relatively to the decrement or increment operation:

- In prefix form, the variable is *first updated* and then the expression is evaluated.
- In postfix form, the expression is *evaluated before* the variable is updated.

```
my $x = 2;
### $x

my $y = $x++;
### $x
### $y

my $z = ++$x;
### $x
### $z

# gives:

### $x: 2

### $x: 3
### $y: 2

### $x: 4
### $z: 4
```

Such subtleties in the order of operations can be used to write ultra-concise Perl code. Consider how much this excerpt from `codon_quizz_jokers.pl` is simplified by taking advantage of the postfix form of the auto-decrement operator, which first evaluates then decrements its operand.

```
# skip current question if asked to use a joker
if ($answer eq 'SKIP') {

    # if ($joker_n > 0) {           # original version
    #    $joker_n--;
    #    next QUESTION;
    # }

    next QUESTION if $joker_n--;    # new (concise) version

    say q{You don't have any joker left!};
    $answer = 'EXIT';
}
```

Chapter 13

More on hashes

13.1 Hash uses

`codon_usage.pl` uses hashes (or associative arrays) in two novel ways. Let's summarize the different uses that we have encountered so far for hashes.

1. **Indexed data storage** — The result of the code reading a FASTA file is the hash `%seq_for` (see "`open` & `close`", p.95). This hash stores the *identifiers* (ids for short) of the input sequences, along with the *sequences* themselves. Associating pieces of data to their ids is a common use for hashes.

Table 13.1: Contents of hash `%seq_for` (reading `Ecoli_cds.fasta`)

identifier	sequence
AAC73112 cdna: …	ATGAAACGCATTAGCACCACCATTACCACCACCATCACCA…
AAC73113 cdna: …	ATGCGAGTGTTGAAGTTCGGCGGTACATCAGTGGCAAATG…
AAC73114 cdna: …	ATGGTTAAAGTTTATGCCCCGGCTTCCAGTGCCAATATGA…
…	(4464 other sequences)
b4701 cdna:pseudo …	TTTGGGTTCGAACGCTGGCCTCAGGTTGATAGAAATATCG…
b4704 cdna:pseudo …	GTAATCCGATTTAAATATCGAGTCTCCTTGTTTCGACTTA…

2. **Dictionaries** — We also used hashes as dictionaries in translating tasks, either when reverse-complementing bases (`%comp_for`) in `rev_comp.pl` or when conceptually translating codons to amino acid (`%aa_for`), for example in `translate.pl`. Don't forget that hashes are unordered.

Table 13.2: Contents of hash `%comp_for`

base	complement
A	T
T	A
G	C
C	G

Table 13.3: Contents of hash `%aa_for`

codon	amino acid
AAA	K
AAC	N
AAG	K
AAT	N
ACA	T
…	…
TTG	L
TTT	F

3. *Counters* — In `codon_usage.pl`, we declare two hashes to be used as counters. The first one (`%count_for`) holds the number of times (or **occurrences**) we see each codon, whereas the second one (`%total_for`) keeps track of the number of occurrences of each amino acid. The heart of the Perl idiom for using hashes as counters lies in the `$hash{$key}++` construct below.

```
my %count_for;
my %total_for;

# later in the loop...

$count_for{$codon}++;
```

Remember that the auto-increment operator (`++`) adds one to its operand. When a hash has just been declared, it is empty and contains no key/value pair. Thus, the first time `++` is applied to a given key, the corresponding value is first created and set to `undef`, then evaluated to zero (`0`) in the numeric context imposed by the operator, and finally incremented to one (`1`).

Observe how the codons of the DNA string below are discovered and counted each one in turn.

```
$dna_string: 'CATGAACTTCTTTGGCGTCTTGAT'

                $count_for: ||
                            ||

$codon: CAT     $count_for: | CAT |
                            |  1  |

$codon: GAA     $count_for: | CAT | GAA |
                            |  1  |  1  |

$codon: CTT     $count_for: | CAT | GAA | CTT |
                            |  1  |  1  |  1  |

$codon: CTT     $count_for: | CAT | GAA | CTT |
                            |  1  |  1  |  2  |
$codon: TGG     $count_for: | CAT | GAA | CTT | TGG |
                            |  1  |  1  |  2  |  1  |

$codon: CGT     $count_for: | CAT | GAA | CTT | TGG | CGT |
                            |  1  |  1  |  2  |  1  |  1  |

$codon: CTT     $count_for: | CAT | GAA | CTT | TGG | CGT |
                            |  1  |  1  |  3  |  1  |  1  |

$codon: GAT     $count_for: | CAT | GAA | CTT | TGG | CGT | GAT |
                            |  1  |  1  |  3  |  1  |  1  |  1  |
```

The next time we use the same key, the existing value is incremented in place, which allows us to update the occurrence count of the key with exactly the same code as for its initialization.

Table 13.4: Final content of hash `%count_for`

codon	count
AAA	46021
AAC	29496
AAG	14105
AAT	24241
ACA	9657
ACC	31986
…	…
TTG	18673
TTT	30444

To keep track of amino acid counts, we need amino acids as keys while we only have codons in our DNA sequences. Consider how we obtain these keys *on the fly* by translating incoming codons to amino acids using our dictionary hash `%aa_for`.

```
$total_for{ $aa_for{$codon} }++;
```

Table 13.5: Final content of hash `%total_for`

amino acid	count
*	4653
A	129593
C	15846
D	70121
E	78482
F	53065
…	…
W	20885
Y	38750

4. **Boolean filters** — Instead of counting the number of occurrences of each key, we can simply create the key/value pair the first time we see the key. These hashes allow testing if some key has been encountered *at least once*, which is useful when filtering data based on some given list. We have not yet used hashes in this way; so let's give an illustration.

   You can save the program below as `filter_fasta.pl`. To try it, build a list of a few ids from your `Ecoli_cds.fasta` file. Beware that **ids in the list must be truncated on the first space character** for this program to work. Do you recover the expected sequences?

   If so, can you modify this program to output the list of ids that did not match any sequence?

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

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

unless (@ARGV == 2) {
    die <<"EOT";
Usage: $0 <infile.fasta> <wanted.ids>
This tool filters and displays sequences based on their identifier.
It requires a FASTA file as input and a second file containing the ids of the
sequences to be displayed on screen (one id per line).
Example: $0 Ecoli_cds.fasta Ecoli.ids
EOT
}

my $fasfile = shift;
my $idsfile = shift;

open my $ids, '<', $idsfile;

my %wanted;
while (my $line = <$ids>) {
    chomp $line;
    $wanted{$line} = 1;
}

open my $fas, '<', $fasfile;

my $seq_id;
my $seq;

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

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

        # output seq (if any) if wanted
        if ($seq && $wanted{$seq_id}) {
            say '>' . $seq_id;
            say $seq;
        }

        # extract new seq_id
        # use only first word (until first whitespace) for simplicity
        my @words = split ' ', substr($line, 1);
        $seq_id = shift @words;

        # prepare for new seq
        $seq = q{};
        next LINE;
    }

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

# output last seq (if any) if wanted
if ($seq && $wanted{$seq_id}) {
    say '>' . $seq_id;
    say $seq;
}
```

Table 13.6: Exemplative content of hash `%wanted`

seq-id	wanted?
AAC77347	1
AAC77348	1
AAC77349	1
…	…

5. **Switch tables** — Finally, hashes can be used in more exotic ways, such as when they replace cascades of `elsif` blocks. We have discussed this peculiar use in "Hashes instead of cascades of `elsif`", p.63. Here's the code again for reference.

```
my %div_for = (
    bp => 1,
    kb => 1000,
    Mb => 1e6,              # one million
    Gb => 1e9,              # one billion
);
my $div = $div_for{$unit};
die 'Unknown unit for reporting sequence size!' unless $div;
$size /= $div;              # divide $size by suitable divisor
```

Table 13.7: Contents of hash `%div_for`

unit	divisor
bp	1
kb	1000
Mb	1,000,000
Gb	1,000,000,000

13.2 `values` & `sort` and reuse of variable names

When counting codon occurrences, we don't need the sequence identifiers. That is why we only iterate over a list of the sequences themselves. This list is obtained from the hash `%seq_for` using the `values` builtin function, which is the mirror of the `keys` function that we used in `codon_quizz.pl`.

```
for my $dna_string (values %seq_for) {
    # loop body
}
```

To improve the autodocumentation of my programs, I often strive to use the same variable names for the keys and for the values when building and using the hash. Here, I don't. This is to illustrate the fact that iterator variables can bear any name. Thus, the `$dna_string` iterator actually corresponds to the various `$seq` strings that have been stored in the hash `%seq_for` when reading the FASTA file.

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

In contrast, I use the same variable name (`$codon`) as the key for accessing the two hashes `%aa_for` and `%count_for` to help conveying what is going on in the program. Note that the four distinct uses of `$codon` do not refer to the same variable but to different variables existing in different lexical scopes.

```
# use 1: when building %aa_for
my %aa_for;
my $codon_n = length $aa;
for my $i (0..$codon_n-1) {
    my $codon = substr($b1, $i, 1) . substr($b2, $i, 1) . substr($b3, $i, 1);
    $aa_for{$codon} = substr($aa, $i, 1);
}

# uses 2 and 3: when building %count_for and using %aa_for
CODON:
for (my $i = 0; $i < $len; $i += 3) {
    my $codon = uc substr($dna_string, $i, 3);
    last CODON if length $codon < 3;        # skip truncated codons
    $count_for{         $codon  }++;
    $total_for{ $aa_for{$codon} }++;
}
```

```
# use 4: when using %count_for
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 / $total;     # usage in percents
    say {$out} join "\t", $codon, $count, $aa, $total,
        sprintf "%5.1f", $usage;
}
```

In the last case, we build a list of codons in lexical order from the hash `%count_for` by chaining the `sort` and `keys` builtin functions. This is reminiscent of what we did in `codon_quizz.pl` when building the list of questions, except that in the case of `%count_for`, we directly iterate over the list instead of storing it in an array.

```
my @questions = shuffle keys %aa_for;
```

Chapter 14

Towards more complex programs

14.1 A production-grade translation tool

Below is a revised and enhanced version of our conceptual translation tool.

1. Type in and save it as `xxl_xlate.pl`. Note that some parts can be copied-pasted from our previous programs to reduce typing.
2. Before proceeding, you will probably need to install a few new CPAN modules.

   ```
   $ cpanm LWP::Simple
   $ cpanm Path::Class
   $ cpanm Tie::IxHash
   ```

3. Try to use the program on your *E. coli* CDS. I let you figure out how to run it.
4. Study the listing and do your best to understand what do the various constructs that we have not covered yet. Don't worry about "Regular expressions", p.125, yet!

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

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

use Smart::Comments '###';

use File::Basename;
use LWP::Simple 'get';
use Path::Class 'file';
use Tie::IxHash;


unless (@ARGV == 3) {
    die <<"EOT";
Usage: $0 <infile.fasta> <path-to-gc.prt|--remote> <gc-id>
This tool translates DNA sequences to proteins using any genetic code.
```

```
It requires a FASTA file as input, the path to a local copy of NCBI 'gc.prt'
file (or the special --remote option) and the identifer of the desired genetic
code (1-6, 9-16, 21-25). There is no option for the name of the output file
because it is automatically derived from the name of the input file.
Example: $0 Ecoli_cds.fasta ~/Downloads/gc.prt 1
EOT
}

my $infile = shift;
my $gcfile = shift;
my $gc_id  = shift;


### Reading input file: $infile

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;

close $in;
```

```


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

# read gc.prt file
my $gc_content
    = $gcfile eq '--remote'
    ? get('ftp://ftp.ncbi.nih.gov/entrez/misc/data/gc.prt')
    : file($gcfile)->slurp
;
#### $gc_content

# split gc.prt file content into code blocks
my @codes = $gc_content =~ m/ \{ ( [^{}]+ ) \} /xmsg;
die qq{ABORT! Cannot process '$gcfile' file!} unless @codes;
#### @codes

my %aa_for;

CODE:
for my $code (@codes) {

    # extract code id
    my ($id) = $code =~ m/ id \s* (\d+) /xms;

    # process only specified code block
    if ($id == $gc_id) {

        # extract amino acid line
        my ($aa) = $code =~ m/ ncbieaa \s* \"(.*?)\" /xms;

        # extract three codon lines
        my ($b1) = $code =~ m/ Base1   \s* ([TACG]+) /xms;
        my ($b2) = $code =~ m/ Base2   \s* ([TACG]+) /xms;
        my ($b3) = $code =~ m/ Base3   \s* ([TACG]+) /xms;

        # build translation table
        my $codon_n = length $aa;
        for my $i (0..$codon_n-1) {
            my $codon
                = substr($b1, $i, 1)
                . substr($b2, $i, 1)
                . substr($b3, $i, 1)
            ;
            $aa_for{$codon} = substr($aa, $i, 1);
        }

```

```
            last CODE;
        }
    }
    #### %aa_for

    die qq{ABORT! No code found for id '$gc_id'!} unless %aa_for;


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

    # derive outfile name from infile name using cross-platform methods
    my ($basename, $dir, $suffix) = fileparse($infile, qr{\.[^.]*}xms);
    my $outfile = file($dir, $basename . '_pep' . $suffix);

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

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

        my @aminoacids;
        my $len = length $dna_string;

        for (my $i = 0; $i < $len; $i += 3) {
            my $codon = uc substr($dna_string, $i, 3);
            push @aminoacids, $aa_for{$codon} // 'X';
        }

        say {$out} '>' . $id;
        say {$out} join q{}, @aminoacids;
    }

    close $out;         # useless because files are automatically closed
                        # when the filehandle variable gets out of scope

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

Homework

1. Try to understand the Perl novelties appearing in this new program.
2. Modify `codon_usage.pl` (rename it `hw4_codon_usage_sort.pl`) to get the codon usage table sorted in lexical order on the amino acids (rather than codons).

Part V

Lesson 5

Chapter 15

Looking at the novelties in `xxl_xlate.pl`

15.1 Would you like some syntactic sugar?

15.1.1 Ordered hashes: `Tie::IxHash`

The FASTA reader of `xxl_xlate.pl` has been borrowed from `codon_usage.pl` except for one detail.

```
use Tie::IxHash;

# and later...
tie my %seq_for, 'Tie::IxHash';
```

Tied objects are beyond the scope of this course. However, the `Tie::IxHash` CPAN module is so useful that we need to mention it. In short, this module offers peculiar hashes in which the order of key/value pairs is preserved.

For us, this means that DNA sequences will be stored in the order by which they appear in the input FASTA file. This new behavior benefits to all **hash iterators**, whether `keys`, `values` or the powerful `each` that allows traversing the hash by iterating over key/value pairs.

15.1.2 `each` and list assignment

The `each` builtin function returns a list of two values corresponding to the next key/value pair. These were respectively called `$seq_id` and `$seq` in our FASTA file reader. Here, we retrieve and store them in two different variables called `$id` and `$dna_string` in a single operation.

```
while (my ($id, $dna_string) = each %seq_for) {
    # translate sequence
}
```

This very concise syntax is called a **list assignment**. Basically, the first value of the list goes into the first variable, while the second value of the list goes into the second variable. This is enough for now. We will fully describe list assignment in “Perl values: Lists”, in the second part of this course.

As for any hash iterator, the `while` loop lasts as long as there are key / value pairs remaining to be returned in the hash. When the whole hash has been traversed, `each` returns an empty list that evaluates to a false value in a boolean context and terminates the loop.

15.1.3 The ternary conditional operator

`xxl_xlate.pl` assembles its dictionary hash for translating sequences directly from the NCBI `gc.prt` file (gc for *genetic code*). This is achieved using regexes, which are described in "Regular expressions", p.125. Before that, the file has to be read either from the user's local disk or from the NCBI **FTP server** (if the `--remote` command-line argument has been specified).

```
my $gc_content
    = $gcfile eq '--remote'
    ? get('ftp://ftp.ncbi.nih.gov/entrez/misc/data/gc.prt')
    : file($gcfile)->slurp
;
```

We will explain the functional novelties in a minute, but for now, let's consider the general syntax of this statement. This is actually both the declaration and the definition of a new variable called `$gc_content`, which stores the full content of the `gc.prt` file.

The strategy we use for handling both options of the alternative in a single statement rests on the **ternary conditional operator** (`?:`). This logical operator takes three operands. It evaluates the first in boolean context and then evaluates to the second if the first yields a true value and to the third otherwise. Yes, it is a shortcut for an `if/else` construct. Here's a simpler example.

```
say $codon eq 'ATG' ? 'starting...' : 'continuing...';
```

There exist many different uses for the ternary conditional operator. Later, in "Character classes", p.135, we will see it in action to replace a cascade of `elsif`.

15.2 Writing portable code

15.2.1 LWP::Simple

When downloading files in the shell, we use `wget` (or `curl`). We could do the same in Perl with the handy executing quoting operator (`qx(...)`) discussed in a box above.

```
my $gc_content = qx(wget ftp://ftp.ncbi.nih.gov/entrez/misc/data/gc.prt);
```

However, this would not be **portable**, which means that the system call could easily fail, for example if run on an operating system where `wget` is not available. The recommended workaround for such cases is to look on CPAN for a Perl module offering the same functionality.

This module exists and is called `LWP::Simple`. Not surprisingly, it exports the `get` function that does exactly what we need. Moreover, it takes care of the numerous details that might cause trouble when using the command-line `wget`.

```
use LWP::Simple 'get';

# and later...
my $gc_content = get('ftp://ftp.ncbi.nih.gov/entrez/misc/data/gc.prt');
```

15.2.2 Path::Class

When gc.prt is locally available, we have to read it. Of course, we could open it and read it line by line in a while loop, as we have explained in "open & close", p.95. For our purposes, however, we prefer to read it at once and to store all its content in a string variable. This form of file reading is known as **slurping** the file and can be carried out using the following construct.

```
use Path::Class 'file';

my $gcfile = shift;

# and later...
my $gc_content = file($gcfile)->slurp;
```

To completely understand this statement, you would need to know *object-oriented* Perl. For now, simply think of it as turning the $gcfile string into some kind of File **object** (using the exported file function) that is then immediately slurped.

Slurping files is not the primary goal of the Path::Class CPAN module. Instead, it has been written to manipulate directory and file paths in a portable (i.e., system-independent) way. For example, under Windows, this module builds paths using the **backslash character** (\) as the directory separator, whereas it uses the **forward slash character** (/) on Linux and macOS.

```
my $outfile = file($dir, $basename . '_pep.fasta');

# and later...
### Writing output file: $outfile->stringify
```

Because of the object-orientation of the module and of the way Smart::Comments functions, we need to use the stringify **method** to print the output file name built by Path::Class. In normal string context, such as when opening a file, this **stringification** happens automatically.

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

15.2.3 File::Basename

The last CPAN module we discuss here automatically exports a function for parsing file paths into their directory, file name and file suffix (also known as **file extension**). Again, it does that portably.

```
my ($basename, $dir) = fileparse($infile, qr{\.[^.]*}xms);
```

The fileparse function expects a file path and a (list of) **pattern(s)** describing the allowed suffix (or suffices). Here, the pattern is a regular expression allowing for any suffix that starts with a dot (.) (see "Regular expressions", p.125 for details).

The function returns a list of three values corresponding to the file name, the directory and the suffix. Since we only provide two variables for storing them, the last one (the suffix) is discarded. We then use the other two values for building the output file name (as shown above).

```
my $outfile = file($dir, $basename . '_pep.fasta');
```

The code above assumes that the suffix of the output file name should be `.fasta`. By slightly modifying the two previous lines, we can preserve the original suffix, whatever it is (e.g., `.fasta`, `.fst`).

```
my ($basename, $dir, $suffix) = fileparse($infile, qr{\.[^.]*}xms);
my $outfile = file($dir, $basename . '_pep' . $suffix);
```

Chapter 16

Regular expressions

16.1 What are regular expressions?

Perl **regular expressions** are a sort of black magic. Actually, they are more like a language within the language. This is why mastering their use is the quest of a lifetime. Here, I will only scratch the surface and give some useful advices using examples from real-world bioinformatics. For a more complete treatment, see the official Perl documentation:

- tutorial [`http://perldoc.perl.org/perlretut.html`]
- full documentation [`http://perldoc.perl.org/perlre.html`]
- quick reference [`http://perldoc.perl.org/perlreref.html`]

You can also consult the following books:

- *Modern Perl* by chromatic [`http://modernperlbooks.com/books/modern_perl_2014/`]
- *Mastering Regular Expressions* by Jeffrey Friedl (published by O'Reilly Media).

If you look at the latter book, you will realize that Perl-like regular expressions have spread to many other languages (such as **R** or **PHP**) thanks to the **PCRE library** written in **C**. Knowing them is thus a serious asset that will prove useful in many situations. Even though, Perl still has the most powerful regular expression engine of the known universe…

16.2 Defining regular expressions

Regular expressions are usually surrounded by a pair of **forward slash characters** (`//`). If you want to store a regular expression (**regexp** or **regex** for short) in a variable (or pass it to a function; see "Functions", in the second part of this course), use the **regex quoting operator** (`qr//`). In the example below, the `ATG` is called a **pattern**. It is even a **literal pattern** because it consists in a simple substring.

```
my $start_regex = qr/ATG/;
```

As for the quoting operators (`q{}` and `qq{}`), you can use any *balanced pair* of characters (or even a single character) in place of the slashes (e.g., `qr{}`). This helps building regexes containing slashes. Otherwise, you would need escaping them using the backslash character (`\`).

```
my $web_regex = qr/http:\/\//;      # hard way...
my $web_regex = qr{http://};        # better way!
```

This is very important. In the following, we will see that many characters have special meanings within regexes. Remember to always **escape** such special characters when you intent to use them in their **literal meaning**.

Stored regexes can be combined into larger regexes, which helps improving legibility. This can be useful when using the **alternation metacharacter** (`|`) that allows a string to match either one pattern or another pattern (or other patterns).

```
my $web_regex = qr{http://};
my $ftp_regex = qr{ftp://};
my $web_or_ftp_regex = qr{$web_regex|$ftp_regex};
```

16.3 Using regexes

To apply a regex to a string, use one of the two **binding operators** (`=~` and `!~`). These are infix operators requiring the string to be analyzed as their left operand and the regex to be applied as their right operand. In boolean context, the **positive binding operator** (`=~`) evaluates to a true value if the match succeeds, whereas its negated form (`!~`) evaluates to a true value if the match does not succeed.

```
say 'web'     if 'http://www.ncbi.nlm.nih.gov/' =~ $web_regex;
say 'not ftp' if 'http://www.ncbi.nlm.nih.gov/' !~ $ftp_regex;

# gives:

web
not ftp
```

Of course, the string can be stored in a variable.

```
my $url = 'http://www.ncbi.nlm.nih.gov/';
say 'web'     if $url =~ $web_regex;
say 'not ftp' if $url !~ $ftp_regex;
```

The regex can be used either in a simple *search* or in a *search and replace*. Each operation is in principle introduced by one of two **circumfix operators** (`m//` and `s///`, respectively). In their absence, a simple search is performed. Thus, the examples above could have been written more explicitly as follows.

```
say 'web'     if $url =~ m/$web_regex/;
say 'not ftp' if $url !~ m/$ftp_regex/;
```

When searching and replacing, the **substitution operator** (`s///` or `s{}{}`) requires itself two operands: a regex for the search part and a regular string (not a regex) for the replacement part.

```
my $url = 'http://www.ncbi.nlm.nih.gov/';
$url =~ s{$web_regex}{ftp://};
say $url;

# gives:

ftp://www.ncbi.nlm.nih.gov/
```

The table below lists the operators related to regexes. Note that the **negative binding operator** (`!~`) should only be used with the **search operator** (`m//`). Don't worry if you don't know the last operator; we will describe it in a few minutes.

Table 16.1: Operators for regular expressions

operation	fixity	operator
match	infix	`=~`
do not match	infix	`!~`
search	circumfix	`m//`
search and replace	circumfix	`s///`
transliterate	circumfix	`tr///` or `y///`

Here's a complete example. Save it as `url_regexes.pl` to play with it.

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

use Modern::Perl '2011';

my $web_regex = qr{http://};
my $ftp_regex = qr{ftp://};

my $web_or_ftp_regex = qr{$web_regex|$ftp_regex};

my $url = 'http://www.ncbi.nlm.nih.gov/';
say $url;

say 'web'             if $url =~ m/$web_regex/;
say 'ftp'             if $url =~ m/$ftp_regex/;
say 'web or ftp'      if $url =~ m/$web_or_ftp_regex/;

say 'not web'         if $url !~ m/$web_regex/;
say 'not ftp'         if $url !~ m/$ftp_regex/;
say 'nor web nor ftp' if $url !~ m/$web_or_ftp_regex/;

$url =~ s{$web_or_ftp_regex}{https://};     # change protocol
say $url;
```

16.4 When *not* to use regexes?

Regexes are extremely powerful yet expensive monsters. This means that they can slow down your code if not carefully crafted. Moreover, there exist cases where they should not be used at all: searches for literal patterns and single-character search and replace (known as **transliteration**).

16.4.1 Literal searches: `index`

The searches shown in "Using regexes", p.126, can be rewritten using the `index` builtin function. This efficient function returns either the position of the first occurrence of the searched substring within the string or `-1` if the string does not contain the substring. Optionally, you can even specify a position from where to begin the search.

```
my $url = 'http://www.ncbi.nlm.nih.gov/';
say 'web'     unless index($url, 'http://') < 0;
say 'not ftp'     if index($url,  'ftp://') < 0;

#           1         2
# 0123456789012345678901234567
# http://www.ncbi.nlm.nih.gov/

say index($url, 'http://');
say index($url, 'ncbi');
say index($url, 'nih');
say index($url, 'n');
say index($url, 'n', 17);

# gives:
```

```
web
not ftp
0
11
20
11
20
```

In the programming literature, the string to be searched through is called the *haystack* and the substring to be searched for is called the *needle*. To find all occurrences of a given string (as in `index.pl` below), one can use a `while` loop in which the position to start the search from is updated at every match.

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

use Modern::Perl '2011';

die "Usage: $0 <haystack> <needle>" unless @ARGV == 2;

my $haystack = shift;
my $needle   = shift;

say qq{Searching for "$needle" in "$haystack"...};

my $pos = 0;
while ( ($pos = index($haystack, $needle, $pos)) >= 0 ) {
    say $pos++;
}
```

BOX 18: When speed matters

To illustrate the efficiency of the `index` builtin function with respect to regexes, consider the following two one-liners (see "Perl *one-liners*", p.145 to learn more). The goal is simply to reorder the input file (`names.dmp`), so as to place all the lines that contain the string `scientific name` at the end of the file (without messing with the other lines).

```
$ time perl -nle 'unless (index($_, "scientific name") == -1) { push @sns, $_ }
    else { print } END{ print for @sns }' names.dmp > names.dmp.sort.index

real    0m2.552s
user    0m2.086s
sys     0m0.523s

$ time perl -nle 'if (m/scientific name/) { push @sns, $_ }
    else { print } END{ print for @sns }' names.dmp > names.dmp.sort.regex

real    0m17.801s
user    0m5.942s
sys     0m8.998s

$ diff names.dmp.sort.index names.dmp.sort.regex
```

While the two output files are identical, the `index`-based solution has run about 7 times faster than the approach using regexes. Moreover, observe how one can split a long one-liner over multiple lines without having to introduce a backslash character (\).

16.4.2 Transliteration: `tr///`

You might be tempted to use regexes for searching and replacing single characters. Consider the following example where we manipulate gap symbols in a DNA sequence.

```
# let's have a gapped sequence
my $seq = '-ACGT--GATG-AGT-AGCTGC-';
say $seq;

# replace all hyphens by stars
$seq =~ s/\-/*/;
say $seq;

# oups... replace all hyphens!
$seq =~ s/\-/*/g;
say $seq;

# remove all stars
$seq =~ s/\*//g;
say $seq;
```

```
# gives:
```

```
-ACGT--GATG-AGT-AGCTGC-
*ACGT--GATG-AGT-AGCTGC-
*ACGT**GATG*AGT*AGCTGC*
ACGTGATGAGTAGCTGC
```

Since the first operand of the `s///` operator is a regex, we need to escape the special characters that appear within it. Furthermore, we want to convert all hyphens, which requires enabling the **global-search mode** with the corresponding **regex modifier** (`/g`).

However, these operations would benefit from using the **transliteration operator** (`tr///` or `y///` or `tr{}{}` or `y{}{}`). This operator does not use regexes but is much faster while still powerful.

```
# let's have a gapped sequence
my $seq = '-ACGT--GATG-AGT-AGCTGC-';
say $seq;

# replace all hyphens by stars
$seq =~ tr/-/*/;
say $seq;

# degap sequence
$seq =~ tr/*//d;
say $seq;

# gives:
```

```
-ACGT--GATG-AGT-AGCTGC-
*ACGT**GATG*AGT*AGCTGC*
ACGTGATGAGTAGCTGC
```

Have you noticed that we did escape special characters nor enable the global-search mode? Instead, we used the `/d` **transliteration option** to *delete* all matching chars in the second transformation.

This is certainly nice but transliteration can do much more… Remember our tool for reverse complementing a DNA sequence? Type in this new and much shorter version, save it as `rev_comp_tr.pl` and try it in your terminal.

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

use Modern::Perl '2011';

die "Usage: $0 <dna-string>" unless @ARGV == 1;

my $dna_string = shift;

say $dna_string;
say scalar reverse $dna_string =~ tr/ACGTacgt/TGCAtgca/r;
```

As you can see, transliteration actually uses a **search list** and a corresponding **replacement list**, which advantageously replaces a dictionary hash for single-character mappings.

If the characters follow each other in the lexical order, simply list the first and last ones separated by a **hyphen character** (`-`) to indicate that you want a **character range**.

```
my $number = 987.123;
$number =~ tr/0-9/A-J/;
say $number;

# stupidly gives:
```

```
JIH.BCD
```

Note that `$number` was first *coerced* to a string prior to transliteration. Perl performs automatic **coercion** based on the value context. This means that a given variable can change its type (e.g., from number to string or the other way around) depending on the last operation that was applied to it.

Finally, the `/r` option tells Perl that we do not want to alter the original string. This has the effect of returning the transliterated string. Otherwise, `tr///` returns the number of characters that have been effectively replaced in the string. Combined with the automatic *replication* of the search list when the replacement list is omitted, this behavior is useful for counting specific characters.

To see it in action, type in and save the short program below as `gc_content.pl` (gc for *GC content*). The latter file should not to be confused with the variable `$gc_content` (for *genetic code file content*) used in `xxl_xlate.pl`! Sorry for the deceptively similar names.

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

use Modern::Perl '2011';

die "Usage: $0 <dna-string>" unless @ARGV == 1;

my $dna_string = shift;
my $len = length $dna_string;
my $gc_count = $dna_string =~ tr/GCgc//;
my $gc_content = 100.0 * $gc_count / $len;

say "DNA: $dna_string";
say "len: $len";
say "#GC: $gc_count";
say "%GC: $gc_content";
```

The table below lists all the options that are available when transliterating strings. You can explore the uses of `/c` and `/s` for yourself.

Table 16.2: Options for the transliteration operator

option	meaning
`/c`	*complement* the search list (find anything except these characters)
`/d`	*delete* found but unreplaced characters
`/s`	*squash* duplicate replaced characters (reduce them to one)
`/r`	*return* the modified string and leave the original string untouched

BOX 19: `scalar reverse`

Before printing the complemented string in `rev_comp_tr.pl`, we need to reverse it. We achieve that without using a temporary array thanks to the construct `scalar reverse`, which reverses the characters of a string.

Note that we force scalar context because, by default, the `reverse` builtin function reverses the values of a list. In our case, the list has only one value: the transliterated string. Reversing it in a list context would thus not change anything!

```
my $protein = 'HELLWRLD';
say reverse $protein;
say scalar reverse $protein;

my @proteins = qw(HELL WRLD);
say reverse @proteins;

# gives:
```

```
HELLWRLD
DLRWLLEH
WRLDHELL
```

You might be surprised by the fact that the last line does not insert a space between the two protein sequences. This is so because we don't use a `join`. Instead, we directly pass a list of strings to `say`, which reacts by printing each of them in turn followed by a single newline character (`\n`). There is no variable interpolation here and thus no use of the list separator (`$"`). Note that `print` would act the same except for the newline.

16.5 Anchors

When searching for literal patterns, longer words that include the searched word (known as **superwords**) are a common issue. They are particularly dangerous in the case of numeric or semi-numeric (sequence) identifiers for which smaller numbers are substrings of larger numbers (`seq1`, `seq2`, `seq3`, … `seq10`, `seq11`, … `seq20`, `seq21`…). This problem crops up very easily when using the `grep` command directly in the shell.

```
# very dangerous
$ grep seq2 seqs.fasta
```

Here's the same example in Perl.

```
# build a series of fake seq ids
my @ids;
for my $i (1..25) {
    push @ids, "seq$i unknown protein";
}
### @ids
```

```
# look for seq2
for my $id (@ids) {
    say "found seq2: $id" unless index($id, 'seq2') < 0;
}

# gives:

### @ids: [
###         'seq1 unknown protein',
###         'seq2 unknown protein',
###         'seq3 unknown protein',
            ...
###         'seq23 unknown protein',
###         'seq24 unknown protein',
###         'seq25 unknown protein',
```

```
found seq2: seq2 unknown protein
found seq2: seq20 unknown protein
found seq2: seq21 unknown protein
found seq2: seq22 unknown protein
found seq2: seq23 unknown protein
found seq2: seq24 unknown protein
found seq2: seq25 unknown protein
```

Ouch! Simply using a regex instead of the `index` function does not help.

```
# look for seq2
for my $id (@ids) {
    say "found seq2: $id" if $id =~ m/seq2/;
}

# gives:
```

```
found seq2: seq2 unknown protein
found seq2: seq20 unknown protein
...
```

To avoid matching superwords, we need to use the **word boundary anchor** (`\b`) that forces the match to occur at the *beginning* or at the *end* of a *word* (i.e., either between `\w` and `\W` or between `\W` and `\w`; see "Metacharacters", p.136, for details). A negated anchor also exists (`\B`), which matches anywhere except at a word boundary (i.e., either between `\w` and `\w` or `\W` and `\W`).

```
for my $id (@ids) {
    say "found seq2: $id" if $id =~ m/\bseq2\b/;
}

# gives:
```

```
found seq2: seq2 unknown protein
```

When processing text files, two useful anchors (`^` and `$`) allow us to match at the *beginning* or at the *end* of the current line.

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

    if ($line =~ m/^>/) {
        # process FASTA id
        # ...
    }

    # remaining of our FASTA file reader
    # ...
}
```

The following table summarizes the anchors described so far.

Table 16.3: Common anchors in regular expressions

meaning	with `/m`	without `/m`
word boundary	`\b`	`\b`
except word boundary	`\B`	`\B`
line start	`^`	
line end	`$`	
string start	`\A`	`\A` or `^`
string end	`\z`	`\z` or `$`

BOX 20: Line anchors *vs.* string anchors

Strictly speaking, `^` and `$` respectively mean the beginning and the end of the *string*, which does not make a difference since we deal with the file line by line. However, this behavior can become very confusing when working with strings containing newline characters (`\n`).

To force these anchors to match at the beginning and at the end of each *line*, use the `/m` regex modifier enabling the **multi-line mode**.

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

use Modern::Perl '2011';

my $exons = <<'EOT';
CTTCTTTGGCGTCTTGATCAT
ATGCATGAACTTCTTTGGCGTCTTGAT
CATGAACTTCTTTGGCGT
CTTTGGCGTCTTGATCATGAA
EOT

say '[^]   found one starting exon' if $exons =~ m/^ATG/;
say '[^/m] found one starting exon' if $exons =~ m/^ATG/m;
```

```
# gives:
```

```
[^/m] found one starting exon
```

If you want to match the absolute beginning and end of the string, independently of the `/m` regex modifier, use `\A` and `\z` anchors, respectively.

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

use Modern::Perl '2011';

my $protein = <<'EOT';
mlqtapmlpglgphlvpqlgalasasrllgsiasvppqhggagfqavrgf
atgavstpaasspghkpaathapptrldlkpgagsfaagavaphpginpa
rmaadsasaaagasgdaalaesymahpaysdeyvesvrpthvtpqklhqh
vglrtiqvfrylfdkatgytptgsmteaqwlrrmifletvagcpgmvagm
lrhlkslrsmsrdrgwihtlleeaenermhlitflqlrqpgpaframvil
aqgvffnayfiayllsprtchafvgfleeeavktythalveidagrlwkd
tpappvavqywglkpganmrdlilavradeachahvnhtlsqlnpstdan
pfatgasqlp
EOT

say '[\A]   full-length protein'  if $protein  =~ m/\Am/i;
say '[\A/m] full-length protein'  if $protein  =~ m/\Am/mi;

my $fragment = <<'EOT';
asrllgsiasvppqhggagfqavrgfatgavstpaasspghkpaathapp
trldlkpgagsfaagavaphpginparmaadsasaaagasgdaalaesym
ahpaysdeyvesvrpthvtpqklhqhvglrtiqvfrylfdkatgytptgs
mteaqwlrrmifletvagcpgmvagmlrhlkslrsmsrdrgwihtlleea
enermhlitflqlrqpgpaframvilaqgvffnayfiayllsprtchafv
EOT

say '[\A]   full-length fragment' if $fragment =~ m/\Am/i;
say '[\A/m] full-length fragment' if $fragment =~ m/\Am/mi;
```

```
# gives:
```

```
[\A]   full-length protein
[\A/m] full-length protein
```

In the last code chunk, we used the `/i` regex modifier to enable **case-insensitive mode**. This means that the pattern `/m/` matches both `"M"` and `"m"`. (All these M's are confusing!)

16.6 Character classes

When several different characters are allowed at a position, you can define a **character class** by listing them between a pair of square bracket characters (`[` and `]`). If it is more convenient to list the disallowed characters, use the **caret character** (`^`) at the beginning of the class to negate its content.

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

use Modern::Perl '2011';

my @seqs = qw(
    HELLWRLDHELLWRLDHELLWRLD
    caugaacuucuuuggcgucuugau
    catraacttntttggcgtsttgat
    catgaacttctttggcgtcttgat
);

my    $prot_like = qr/[EFILPQ]/i;
my     $rna_like = qr/[U]/i;
my $bad_dna_like = qr/[^ACGT]/i;

for my $seq (@seqs) {
    say "$seq looks like "
        . ($seq =~ $prot_like    ? 'a protein'
        :  $seq =~  $rna_like    ? 'RNA'
        :  $seq =~ $bad_dna_like ? 'bad DNA'
        :                          'DNA')
    ;
}
```

```
# gives:

HELLWRLDHELLWRLDHELLWRLD looks like a protein
caugaacuucuuuggcgucuugau looks like RNA
catraacttntttggcgtsttgat looks like bad DNA
catgaacttctttggcgtcttgat looks like DNA
```

Note how we used the ternary conditional operator (`?:`) as a concise way to handle an alternative with more than two options. If you use it, pay attention to code formatting for maximum legibility.

You can also use character ranges for defining classes.

```
my $decimal_digit = qr/[0-9]/;
my $hexadec_digit = qr/[0-9A-F]/i;
```

There exist placement rules for specifying *true* hyphens (`-`) and carets (`^`) in character classes, but it is both easier and more robust to escape them with a backslash character (`\`), as for any special character you need to use literally.

```
my $sci_num_chunk = qr/[\+\-\.0-9E]/i;
```

16.7 Metacharacters

Common character classes are specified using shortcuts known as **metacharacters**. Most of them also exist in a negated form (uppercase metacharacter) that implies a caret character (`^`) before the corresponding characters.

Be careful that these classes are `locale`-aware and Unicode-aware. This means that matching characters are likely more numerous than those of the character classes shown in the following table. However, this is not very important for bioinformatics applications.

Table 16.4: Common character classes in regular expressions

meaning	char class	metachar	negated
word	`[A-Za-z0-9_]`	`\w`	`\W`
digit	`[0-9]`	`\d`	`\D`
whitespace	`[\ \t\n]`	`\s`	`\S`
*anything**	`[^\n]`	`.`	

(*) As you can see from the character class above, the **dot metacharacter** (`.`) does not really match *anything* because it does not match the newline character (`\n`). If you want it to really match anything (including the newline), use the `/s` regex modifier enabling the **single-line mode**.

16.8 Quantifiers

So far, we have not used any **regex quantifier**, which implied that the components of our patterns had to appear *exactly once*, in the order dictated by the regex. In most cases, however, quantifiers are needed for specifying how often a regex component may appear in the matching string. For example, to check whether some strings are numbers, we might do the following.

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

use Modern::Perl '2011';

my $sci_num_chunk = qr/[\+\-\.0-9E]/i;
my $num_like = qr/\A$sci_num_chunk+\z/;

for my $value ( qw(0 13 +6 -70 5.8 5e-2 -2.45e8 FF 527ex) ) {
    say "$value\tlooks like a number" if $value =~ $num_like;
}
```

```
# gives:

0       looks like a number
13      looks like a number
+6      looks like a number
-70     looks like a number
5.8     looks like a number
5e-2    looks like a number
-2.45e8 looks like a number
```

This works by requiring our pattern to match the whole string (using `\A` and `\z` anchors) and the string to be exclusively composed of **one or more** characters (using the `+` quantifier) that are allowed in a scientific number (using the character class `[\+\-\.0-9E]`). Finally, we enable case-insensitive mode (using the `/i` regex modifier) for allowing both lower- and uppercase exponents (`e` and `E`).

Even if it looks great it is not… because we should never reinvent the wheel! In this particular case, Perl has an internal function for testing whether a string looks like a number. It is exported from the `Scalar::Util` module (found in the standard Perl distribution) and is aptly called `looks_like_number`. Thus, we should have used it instead.

```
use Scalar::Util qw(looks_like_number);

for my $value ( ... ) {
    say "$value\tlooks like a number" if looks_like_number $value;
}
```

Other common quantifiers are **zero or one** (`?`) and **zero or more** (`*`). The *zero* means that the component is *optional* but can appear *once* or *more* than once, respectively. Hence, `check_overlap.pl` uses the following expression to skip empty lines, i.e., consisting of zero or more whitespace characters.

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

The **extended-legibility mode** enabled by the (`/x`) regex modifier allows us to embed whitespace characters and comment characters in our regexes to make them more readable. Here's an example from one of my own modules.

```
my $ncbi_pkey = qr/
    \A
    [1-9]               # a number
    \d*                 # ... but not beginning by zero
    \z
/xms;
```

Of course, *true* spaces and comment characters have to be escaped with a backslash character (`\`) when using the `/x` regex modifier. This is often overlooked.

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

The table below summarizes the available quantifiers. The four last ones allow you to be more specific with respect to the number of times a given component should match. Use them wisely… Don't worry about the *greedy* stuff: I explain it in the box below.

Table 16.5: Quantifiers in regular expressions

meaning	greedy	non-greedy
zero or one	`?`	`??`
one or more	`+`	`+?`
zero or more	`*`	`*?`
exactly n	`{n}`	`{n}?`
at least n	`{n,}`	`{n,}?`
at most m	`{,m}`	`{,m}?`
between n *and* m	`{n,m}`	`{n,m}?`

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

use Modern::Perl '2011';

my $ensembl1 = qr/\A [A-Z]{4}   \d{11} \z/xms;
my $ensembl2 = qr/\A [A-Z]{4,7} \d{11} \z/xms;
my $ensembl3 = qr/\A [A-Z]+     \d{11} \z/xms;
my $ensembl4 = qr/\A [A-Z]+     \d+    \z/xms;
my $ensembl5 = qr/\A ENS[A-Z]*G \d+    \z/xms;

my $human_rps2_ens  = 'ENSG00000140988';
my $mouse_rps2_ens  = 'ENSMUSG00000095406';
my $human_rps2_ncbi = 'P15880';

for my $id ($human_rps2_ens, $mouse_rps2_ens, $human_rps2_ncbi) {
    say "{4}   {11} found ENSEMBL ID: $id" if $id =~ $ensembl1;
    say "{4,7} {11} found ENSEMBL ID: $id" if $id =~ $ensembl2;
    say " +    {11} found ENSEMBL ID: $id" if $id =~ $ensembl3;
    say " +     +   found ENSEMBL ID: $id" if $id =~ $ensembl4;
    say " careful   found ENSEMBL ID: $id" if $id =~ $ensembl5;
}
```

```
# gives:

{4}   {11} found ENSEMBL ID: ENSG00000140988
{4,7} {11} found ENSEMBL ID: ENSG00000140988
 +    {11} found ENSEMBL ID: ENSG00000140988
 +     +   found ENSEMBL ID: ENSG00000140988
 careful   found ENSEMBL ID: ENSG00000140988
{4,7} {11} found ENSEMBL ID: ENSMUSG00000095406
 +    {11} found ENSEMBL ID: ENSMUSG00000095406
 +     +   found ENSEMBL ID: ENSMUSG00000095406
 careful   found ENSEMBL ID: ENSMUSG00000095406
 +     +   found ENSEMBL ID: P15880
```

16.9 Capturing groups

When processing the content of the `gc.prt` file in `xxl_xlate.pl`, we used a number of **capturing groups**. To understand how they work, let's first remind you the NCBI definition of a genetic code.

```
name "Standard" ,
name "SGC0" ,
id 1 ,
ncbieaa  "FFLLSSSSYY**CC*WLLLLPPPPHHQQRRRRIIIMTTTTNNKKSSRRVVVVAAAADDEEGGGG",
sncbieaa "---M---------------M---------------M----------------------------"
-- Base1  TTTTTTTTTTTTTTTTCCCCCCCCCCCCCCCCAAAAAAAAAAAAAAAAGGGGGGGGGGGGGGGG
-- Base2  TTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGG
-- Base3  TCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAG
```

Capturing groups are specified by enclosing some components of the regex between one or more pairs of parenthesis characters (`(` and `)`). If the regex matches, the corresponding part(s) of the string is (are) returned as a list. Otherwise, the regex engine returns `undef`.

```
my ($id) = $code =~ m/ id \s* (\d+) /xms;

# and later...

my ($aa) = $code =~ m/ ncbieaa \s* \"(.*?)\" /xms;
my ($b1) = $code =~ m/ Base1   \s* ([TACG]+) /xms;
my ($b2) = $code =~ m/ Base2   \s* ([TACG]+) /xms;
my ($b3) = $code =~ m/ Base3   \s* ([TACG]+) /xms;
```

Since it is a list, we need to use list context for storing the returned text chunks. Indeed, in scalar context, the list would evaluate to its number of values. That is why we use parentheses around the variables that will store the various pieces of data. Forgetting these context-enabling parentheses often results in horrible headaches, especially when the first thing to capture is a plain number.

As an illustration, consider the following code… The captured value for `$id`, which must be a NCBI genetic code identifier, would appear correct at first sight. However, it is not and `1` actually corresponds to the number of captured matches (one). In contrast, `$aa` clearly looks incorrect, since we expect a long string of amino acids. We would thus easily notice that something went wrong.

```
my $id = $code =~ m/ id \s* (\d+) /xms;             # buggy...
### $id
my $aa = $code =~ m/ ncbieaa \s* \"(.*?)\" /xms;    # buggy...
### $aa

# gives:

### $id: 1
### $aa: 1
```

By using a single capturing group in conjunction with the `/g` regex modifier enabling global-search mode, we can ask the regex engine to return all matches at once. That is what we do to extract all codes (each one delimited by a pair of curly braces) at once from the `gc.prt` file.

```
my @codes = $gc_content =~ m/ \{ ( [^{}]+ ) \} /xmsg;
```

The above statement for sure packs a lot of power. You may think of it like a `split` on steroids that yields a list of multi-line strings. Didn't I tell you that regexes were almighty!

By the way, `split` itself actually uses regexes. Hence, in `check_overlap.pl`, we split the lines of the input file on any amount (one or more) of whitespace characters.

```
my @coords = split /\s+/xms, $line;
```

Note that there exist two other ways of accessing the captured substrings. These are the **numbered captures** and the more recent **named captures**. The first ones are quite handy for use in branching directives, even if slightly less legible and sometimes ambiguous.

BOX 21: Greedy *vs.* non-greedy quantifiers

By default, quantifiers are *greedy*, meaning that they try to match as many characters as they can. This can be an issue with non-specific patterns like `/.*/` that will match anything in the string.

A quick-and-dirty fix that works most of the time is to append the **non-greedy quantifier** (`?`) to the greedy quantifier to reduce its greediness! In the presence of this non-greedy quantifier, the regex engine instead tries to match as few characters as possible.

As an example, the line capturing the amino acids uses a non-greedy quantifier (`.*?`). Observe how removing it affects the way the regex matches.

```
my ($aa) = $code =~ m/ ncbieaa \s* \"(.*?)\" /xms;  # non-greedy
### $aa

my ($aa) = $code =~ m/ ncbieaa \s* \"(.*)\" /xms;   # greedy and buggy
### $aa

# gives:

### $aa: 'FFLLSSSSYY**CC*WLLLLPPPPHHQQRRRRIIIMTTTTNNKKSSRRVVVVAAAADDEEGGGG'

### $aa: 'FFLLSSSSYY**CC*WLLLLPPPPHHQQRRRRIIIMTTTTNNKKSSRRVVVVAAAADDEEGGGG",
sncbieaa "---M--------------M--------------M----------------------------'
```

Here's a variant of the loop in our FASTA file reader. We extract the sequence identifier from the definition line using a numbered capture. If the regex does not match, the variable `$1` is set to `undef`.

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

    # at each '>' char...
    if ($line =~ m/^ >(.*)/xms) {

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

        # extract new seq_id
        $seq_id = $1;
        next LINE;
    }

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

Multiple chunks can be captured in the order dictated by the successive pairs of parentheses through the use of higher-rank numbered captures: `$2`, `$3` etc. Imagine that we want to use only the first word of the definition line as the identifier but also want to print the (optional) remainder of the line.

```
if ($line =~ m/^ >(\w+) \s* (.*)/xms) {
    $seq_id = $1;
    say "description for $seq_id is: $2" if $2;
}
```

16.10 Non-capturing groups

It sometimes happens that one needs to group several components of a regex but without wanting to capture the match. For example, to allow for either of the components with the alternation metacharacter (`|`) or to make them optional as a whole using the `?` quantifier. In these cases, you can use the non-capturing sequence (`(?:)`). Here are two examples from my own modules to illustrate its use.

```
my $genus_only = qr/
    \A                   # nothing before!
    (?: ssp | spp | sp)  # either ssp, spp or sp
    \.?                  # optionally followed by a dot
    \z                   # and nothing after!
/xms;

my $full_id = qr/
    \A(?:                # begins with...
    ($family)            # optional family
    -)?                  #   followed by a literal dash
    ($genus)             # genus
    \s+                  # whitespace
    ($species)           # species
    (?:_                 # optional underscore
    ($strain))?          #   followed by strain
    @                    # literal '@'
    ($accession)         # accession
    \z                   # ... at the end
/xms;
```

Interestingly, when displaying regexes with `Smart::Comments`, these appear surrounded by one level of non-capturing parentheses. This is how they are stored internally by Perl.

```
my $start_regex = qr/ATG/;
### $start_regex

# gives:

### $start_regex: qr/(?^u:ATG)/
```

16.11 Modifiers

We have already covered a number of **regex modifiers** that affect the way the regex engine deals with the specified regex. The table below summarizes them. Others do exist but are less common.

Table 16.6: Common regular expression modifiers

modifier	meaning
`/g`	global-search mode (search-and-replace)
`/i`	case-insensitive mode
`/m`	multi-line mode (affecting the `^` and `$` anchors)
`/s`	single-line mode (affecting the `.` metacharacter)
`/x`	extended-legibility mode (allowing for whitespace and comments)

Modern Perl programmers often enforce the rule that all regexes should bear the `/xms` regex modifiers for maximum clarity and robustness. Do your best to respect it too.

Chapter 17

Perl *one-liners*

17.1 Forget sed and awk

When learning the UNIX command line, one often gets very enthusiastic about the text filtering capabilities of utilities such as `grep`, `sort`, `join` etc. This is perfectly justified since these programs are invaluable tools in the toolbox of the bioinformatician.

However, as soon as you find yourself beginning to experiment with `sed` and `awk`, you should stop and consider `perl` instead. There are two reasons for this:

1. Perl is much faster than chaining calls to these utilities using the **pipe character** (`|`).
2. Perl is also much more flexible.

In such *quick-and-dirty* applications, Perl is often used with **one-liners**, i.e., programs that are so short that they fit on the command line. These single-use *disposable* code snippets make heavy use of idiomatic Perl constructs and default variables (see the second part of this course). They can thus be very difficult to read. One-liners are specified by the `-e` switch of the `perl` interpreter.

```
$ perl -e 'print "There are " . 4**3 . " codons.\n"'
```

```
There are 64 codons.
```

17.2 -n and -l switches

To write one-liners, one has to understand that they make different assumptions based on the presence of different switches when invoking the `perl` interpreter. Two important ones are `-n` and `-l`.

The `-n` switch setups an implicit `while` loop reading either directly from the standard input stream or from the input files specified as command-line arguments. Thus…

```
$ ls * | perl -ne 'print'
```

… is equivalent to running the code snippet below against each file name of the current directory (one file name per line). If your directory contains sub-directories, their content will also be printed.

```
while (<>) {
    print;
}
```

Similarly, the following command…

```
$ perl -ne 'print' infile1 infile2
```

… is semantically (albeit only approximately) equivalent to executing the following program and will print all the lines of the two specified files.

```
while (my $infile = shift) {
    open $in, '<', $infile;
    while (<$in>) {
        print;
    }
}
```

The `-l` switch enables **line-ending processing**. This means that each line is automatically `chomp`ed and that each `print` statement behaves like `say`, thus appending a newline character (`\n`) to the printed string. In most cases, this switch is required for the robust operation of one-liners. That is why you will often see one-liners of the form `perl -nle '...'`.

```
while (<>) {
    chomp;
    say;
}
```

17.3 The `/e` regex modifier

Here's a one-liner that might be used to rename sequence identifiers in a FASTA file: each one of the original identifiers will be replaced by `seq1`, `seq2`, `seq3` and so on.

```
$ perl -nle 's/>.*/">seq" . ++$i /ge; print' infile.fasta > outfile.fasta
```

It uses a regular expression to substitute the original FASTA definition lines by new ones derived from the counter `$i`. There are two tricks in this one-liner.

1. The **evaluation mode** enabled by the `/e` regex modifier forces the replacement string to be evaluated as a snippet of Perl code. Thus, each time the regex matches a definition line, the replacement string is evaluated and its result used as the actual replacement string.

2. `$i` begins undefined. The first time `perl` sees it, it is prefixed by the auto-increment operator (`++`). Since the latter imposes numeric context, `perl` first coerces it from `undef` to `0` and then increments it to `1`. Finally, it evaluates the concatenating expression to which it belongs and uses the string `>seq1` as the replacement part of the regex. Subsequent evaluations will result in `>seq2`, `>seq3` etc. Note that the incrementation only happens when `perl` finds a definition line and not for each line of the infile.

If completely renaming the sequences seems a bit too brutal to you, consider the following two one-liners. Can you predict what they do?

```
$ perl -nle 's/^>(.*)/">$1." . ++$seen{$1}/e; print' infile.fasta > outfile.fasta
$ perl -nle 's/>(.*)/ ++$seen{$1} > 1 ? ">$1." . $seen{$1} : ">$1"/e; print' ...
```

17.4 The -i switch

If you specify multiple files to a Perl one-liner, they retain their independence and can be processed separately. This is extremely powerful when combined with the `-i` switch that enables **in-place modification** of the input files. Used alone, this switch is dangerous because it is easy to make a mistake that will mangle your files in an unrecoverable way.

```
$ perl -i -nle '...'        # dangerous; please avoid
```

To preserve the possibility of discarding undesired changes to a batch of files wrongly modified by a too powerful one-liner, specify a file suffix to the `-i` switch. This suffix will be used to backup the files before proceeding.

```
$ perl -i.bak -nle 's/>.*/">seq" . ++$i /ge; print' *.fasta
```

In case of mistake, simply overwrite the original files with the backup files to restore their previous content. This can be done in two different ways.

```
# from the point-of-view of the original files
$ for f in *.fasta; do mv -f $f.bak $f; done

# from the point-of-view of the backup files
$ for f in *.bak; do mv -f $f `basename $f .bak`; done
```

17.5 The -a and -F switches

The very handy `-a` switch asks the `perl` interpreter to **auto-split** the input lines and to put the resulting substrings in the default array `@F`. Columns are delimited by whitespace characters.

```
while (<>) {
    chomp;
    my @F = split ' ';
    ...
}
```

If you want to emulate the behavior of `cut`, which by default uses the tab character (`\t`) as its column separator, add the `-F` switch. For example, we might filter the content of our `Ecoli_cds.usage` output file to print only those codons that are used in at least 60 percent of the cases.

```
$ perl -F"\t" -anle 'print if $F[4] >= 60.0' Ecoli_cds.usage

AAA 46021   K   60126    76.5
ATG 37917   M   37917   100.0
CAG 39498   Q   60503    65.3
GAA 54083   E   78482    68.9
GAT 43985   D   70121    62.7
TAA 2896    *   4653     62.2
TGG 20885   W   20885   100.0
```

As you may have guessed by now, this one-liner is equivalent to the following piece of code.

```
while (<>) {
    chomp;
    my @F = split "\t";
    say if $F[4] >= 60.0;
}
```

17.6 The END code block

A last trick for one-liners involves using an END code block to perform a final operation after having processed the content of an input file. This is useful for quickly computing a sum or a mean. Here's a one-liner computing the average codon usage frequency.

```
$ perl -F"\t" -anle 'next if m/^#/; $sum += $F[4]; $n++; END{ print $sum/$n }' \
    Ecoli_cds.usage
```

```
32.8109375
```

The first statement skips the lines starting with a comment character (#). For every other line, the frequency, which is stored in the fifth field (`$F[4]`), is extracted and cumulated in the `$sum` variable, while the `$n` variable keeps track of the number of processed lines. Finally, the statement in the END block computes and prints the average frequency.

In a regular script, an END code block is executed as late as possible, after `perl` has finished running the program and just before the interpreter is being exited. It is executed even if the exit results from a call to the `die` builtin function. You may have multiple END blocks within a file; they will execute in reverse order of definition.

17.7 Envoi

Obviously the applications of Perl one-liners are infinite. Nevertheless, as soon as they begin to span more than one or two lines, you should consider writing a real program instead.

As an example of such a conversion, below is the program version of the second one-liner meant to rename duplicate sequences (`rename_seqs.pl`). It is noteworthy that a full program also makes debugging easier thanks to `Smart::Comments`.

```
# one-liner version
$ perl -nle 's/>(.*)/ ++$seen{$1} > 1 ? ">$1." . $seen{$1} : ">$1"/e; print' ...

# program version
```

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

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

use Smart::Comments;

my %seen;
```

```

while (my $infile = shift) {
    open my $in,  '<', $infile;
    open my $out, '>', "$infile.out";
    while (my $line = <$in>) {
        ### %seen
        $line =~ s/^>(.*)/ ++$seen{$1} > 1 ? ">$1." . $seen{$1} : ">$1"/e;
        print {$out} $line;
    }
}
```

BOX 22: How to install cutting-edge or specialized software?

Ordinary Linux package managers such as `apt` tend to be slow to adopt recent versions of rapidly evolving software. This is the case for **NCBI-BLAST+**, for which the Ubuntu 18.04 repository only provides version 2.6.0 (January 2017). In contrast, Ubuntu 20.04 comes with version 2.9.0 (April 2019), whereas the latest is 2.10.1 (September 2020). You can check that by yourself:

```
$ apt search ncbi-blast
```

Similarly, specialized bioinformatics applications (such as CAP3) are often missing from `apt` repositories. To fix this problem once for all, you should install the **Homebrew** package manager. The following instructions are for Linux, but conveniently Homebrew is also available (and was actually first developed) for macOS. For more information, see: `https://brew.sh/`

```
# install dependencies (if needed)
$ sudo apt install build-essential curl git ruby

# install brew
$ URL=https://raw.githubusercontent.com/Linuxbrew/install/master/install.sh
$ sh -c "$(curl -fsSL $URL)"

# setup brew (following the "Next steps" instructions)
$ echo 'eval $(/home/linuxbrew/.linuxbrew/bin/brew shellenv)' >> ~/.profile
$ eval $(/home/linuxbrew/.linuxbrew/bin/brew shellenv)

# add repository for bioinformatics applications
$ brew tap brewsci/bio
```

After Homebrew is properly installed and configured, it becomes very easy to install new software. Just use the `brew` command.

```
$ brew search blast
$ brew search cap3
$ brew install blast cap3
```

Homework

```
Number of segment pairs = 342; number of pairwise comparisons = 8
'+' means given segment; '-' means reverse complement

Overlaps            Containments  No. of Constraints Supporting Overlap

******************* Contig 1 ********************
gi|125991078+
                    gi|109775518+ is in gi|125991078+
******************* Contig 2 ********************
gi|125991053+
gi|109784559+
******************* Contig 3 ********************
gi|125991021-
                    gi|125991020+ is in gi|125991021-
                    gi|125989076+ is in gi|125991020+
******************* Contig 4 ********************
gi|125991014+
gi|109780664+
******************* Contig 5 ********************
gi|125991007+
gi|109778688-
                    gi|109773906+ is in gi|109778688-

DETAILED DISPLAY OF CONTIGS
******************* Contig 1 ********************
                          .    :    .    :    .    :    .    :    .    :    .    :
gi|125991078+         CTGGACGAGCTGCAGGAGGAGGCGCTGGCGCTGGTGGCGCAGGCCCGACGAGAGGGCGAC
                      ------------------------------------------------------------
consensus             CTGGACGAGCTGCAGGAGGAGGCGCTGGCGCTGGTGGCGCAGGCCCGACGAGAGGGCGAC

                          .    :    .    :    .    :    .    :    .    :    .    :
gi|125991078+         ACGCCGGAAAAGACGCCCCGCGGGGAGAAGCACAACATGTTCCAGGACGCGGGCAAGCCC
                      ------------------------------------------------------------
consensus             ACGCCGGAAAAGACGCCCCGCGGGGAGAAGCACAACATGTTCCAGGACGCGGGCAAGCCC

...
```

CAP3 is a famous bioinformatics program for assembling DNA sequences into contigs. Upon running, it writes out several files and displays its assembly to the screen. At the previous page is an excerpt of a typical screen output of CAP3.

Your new assignment is to write a program (`hw5_cap3_parser.pl`) that parses this file and lists the identifiers of the DNA fragments composing each contig (the expected output is shown just below).

```
Contig 1
- gi|125991078
- gi|109775518
Contig 2
- gi|125991053
- gi|109784559
Contig 3
- gi|125991021
- gi|125991020
- gi|125989076
Contig 4
- gi|125991014
- gi|109780664
Contig 5
- gi|125991007
- gi|109778688
- gi|109773906
```

Here, all ids are of the form `gi|NNNNNNNNN`, which can help you to write your regexes. Yet, if you do this, your parser will not work with other id types. Therefore, try not to rely on this observation. In contrast, you can safely assume that contigs will always be named `Contig N` or `Contig NN` etc.

Index