Lecture notes for
Programming in Perl – BINP13

Mattias Ohlsson[1]
Department of Astronomy and Theoretical Physics
Lund University

Fall 2015

[1]The following persons have also contributed significantly to the notes: Leif Lönnblad and Jari Häkkinen

Contents

1 Preface **9**

1.1 Introduction to this course 9

1.2 How to obtain files and documents for the exercises 10

1.3 How to find (local) Perl information 10

1.4 Disclaimer 10

1.5 Contact Information 10

2 Introduction to programming **11**

2.1 What is a computer 11

2.2 What is data 11

2.3 What is a program 12

2.4 Operating system 12

2.5 Languages 14

2.6 Perl 15

2.7 Hello World 15

2.8 Procedural vs. Object Oriented programming 15

2.9 Comments 16

2.10 Solving problems with Perl 16

2.11 Hand-in exercises 17

2.11.1 Rules for the hand-in exercises 18

3 Basics in Perl **21**

3.1 Variables and operations 21

3.1.1 Scalars 21

3.1.2 Strings 21

3.1.3 Numbers 22

3.1.4 Operators 23

3.1.5 Shortcut operators 24
3.1.6 Variable interpolation 25
3.1.7 Arrays and lists 25
3.1.8 Hashes 27
3.1.9 Follow-up tasks 3-1 28
3.2 Basic input/output 30
3.2.1 Follow-up tasks 3-2 32
3.3 Loops and conditions 34
3.3.1 Condition for one statement 34
3.3.2 Condition for compound statements 35
3.3.3 What is truth? 36
3.3.4 Logical operators 37
3.3.5 Loops 37
3.3.6 More loops 38
3.3.7 The magic of `$_` 40
3.3.8 A complete program 40
3.3.9 New features: the switch statement 41
3.3.10 Follow-up tasks 3-3 42
3.4 Pattern matching 42
3.4.1 Approximately equal to 43
3.4.2 Regular expressions 43
3.4.3 Meta characters 44
3.4.4 What has been matched? 46
3.4.5 Match again 46
3.4.6 Greedy versus tight matching 47
3.4.7 Match the matched 47
3.4.8 Variable interpolation in regular expressions 48
3.4.9 Substitutions 48
3.4.10 New features: smart matching 50
3.4.11 Follow-up tasks 3-4 51
3.5 Filehandles 52
3.5.1 User defined filehandles 52
3.5.2 Pipes 54
3.5.3 Follow-up tasks 3-5 55

3.6 Hand-in Exercise 1 55

4 Programming in Perl 57

4.1 More on built-in functions in Perl 57
4.1.1 split and join (+ qw, x operator, "here docs", .=) 57
4.1.2 push, pop, shift, unshift and splice 60
4.1.3 The map function 61
4.1.4 Sorting 62
4.1.5 Standard mathematical functions 64
4.1.6 Other built-in functions 64
4.1.7 Follow-up tasks 4-1 65
4.2 User-defined functions 65
4.2.1 Defining and calling simple functions 66
4.2.2 Functions with arguments 67
4.2.3 Functions returning a value 68
4.2.4 Namespaces and variable scopes 69
4.2.5 References as function arguments 71
4.2.6 Follow-up tasks 4-2 73
4.3 Accessing things outside a script 74
4.3.1 Command-line arguments 74
4.3.2 Environment variables 76
4.3.3 Running programs from within a script 77
4.3.4 Follow-up tasks 4-3 79
4.4 Modules 80
4.4.1 A simple user-provided module 80
4.4.2 Using the `Getopt::Std` module 82
4.4.3 Follow-up tasks 4-4 84
4.5 Rules of thumb and recommendations 84
4.5.1 Before starting coding 84
4.5.2 Disciplined programming 84
4.5.3 User-friendly programming 85
4.6 Hand-in Exercise 2 85

5 Applying Perl 87

5.1 Introduction to chapter 5 87

5.2 Change of file formats . . . 87
5.2.1 clustalw to phylip and phylip to clustalw . . . 87
5.2.2 Follow-up tasks 5-1 . . . 93
5.3 References, objects and methods [1] . . . 94
5.3.1 Creating references . . . 94
5.3.2 Using references . . . 95
5.3.3 Examples of using references . . . 97
5.3.4 Objects and methods . . . 99
5.3.5 Follow-up tasks 5-2 . . . 99
5.4 Searching in large text files . . . 100
5.4.1 The Swiss-Prot flat file . . . 100
5.4.2 The pdb2sprot.txt file . . . 102
5.4.3 search.pl . . . 102
5.4.4 Follow-up tasks 5-3 . . . 104
5.5 Blast parsing . . . 105
5.5.1 Follow-up tasks 5-4 . . . 111
5.6 Hand-in exercise 3 . . . 111

6 The CGI and bioperl Modules 113

6.1 Introduction to chapter 6 . . . 113
6.2 Numerical Perl . . . 113
6.2.1 Matrices . . . 114
6.2.2 Random Numbers . . . 116
6.2.3 Simple Statistics Module . . . 117
6.2.4 The Perl Data Language . . . 118
6.2.5 Follow-up tasks 6-1 . . . 120
6.3 `CGI.pm` (Common Gateway Interface) . . . 120
6.3.1 Example 1 . . . 121
6.3.2 Example 2 . . . 122
6.3.3 Example 3 . . . 123
6.3.4 Example 4 . . . 126
6.3.5 Example 5 . . . 127
6.3.6 Follow-up tasks 6-2 . . . 128

[1] Inspiration of how to write this section is coming from "Programming Perl, third edition"

6.4 BioPerl project (Part 1) 129
6.4.1 `Bio::Seq` 129
6.4.2 `Bio::SeqIO` 131
6.4.3 `Bio:DB::GenBank` 135
6.4.4 Follow-up tasks 6-3 137
6.5 BioPerl project (Part 2) 138
6.5.1 `Bio::SearchIO` 138
6.5.2 `Bio::Tools::Run::RemoteBlast` 140
6.5.3 `Bio::AlignIO` 142
6.5.4 `Bio::Tools::OddCodes` 142
6.5.5 Follow-up tasks 6-4 144
6.6 Hand-in exercise 4 144

A Example code output **147**

B Common Perl mistakes **155**

Chapter 1

Preface

1.1 Introduction to this course

In short, this course will teach you the basics in Perl programming, were the focus will be on solving common problems that may appear in everyday bioinformatics.

The course is divided into lectures in the morning (10.15 - 12.00) and follow-up exercises in the afternoon. The will be exercise leaders between 13.15 - 15.00 (16.00).

During the course you should complete and hand in four "hand-in exercises"[1] that will be examined by the exercise leaders. These four exercises are part of the examination of this course together with a written exam at the end of the course. Your final grade is depends on **both** the hand-in exercises and the written exam.

This document contains the lecture notes, follow-up exercises and hand-in exercises. The notes are meant to be a complement for the lectures and not as a standalone "textbook". Usually you will need additional files in order to complete the exercises. These can be accessed on the course web page (see below). The lecture notes contains a lot of example code, e.g.

```
1  #! /usr/bin/perl -w
2  print "Hello World\n";
```

Usually one can find the output of the example code either directly in the notes or in the Appendix A. When the output of the example code is in the Appendix A, it is marked with a † character. E.g.

hello.pl†

```
1  #! /usr/bin/perl -w
2
3  print "Hello World\n";
4
```

hello.pl†

[1]Look at section 2.11 for more information about the hand-in exercises.

1.2 How to obtain files and documents for the exercises

There is a web page for this course where Perl files and other documents for the exercises can be accessed. The homepage is

`http://home.thep.lu.se/∼mattias/teaching/binp13`

1.3 How to find (local) Perl information

You will most likely have to look up information about various Perl related questions. One very good start is to use the `perldoc` command available in the Linux console. `perldoc` is like the unix `man` command but only for "Perl questions". Try

```
>> perldoc perl
```

to start exploring all the (local) information available on Perl. You can also look at the course homepage under the "Misc" page to find other online Perl information sources.

1.4 Disclaimer

The idea of this course is to introduce everyone to Perl and show how to solve some common problems in bioinformatics. Solving a problem using Perl can be done in more than one way, easier or more complicated, writing compact code or writing code that is easy to follow, writing code that is fast or writing code that takes longer time to execute, etc. These notes are supposed to cover straightforward Perl usage and the examples are not meant to be robust.

1.5 Contact Information

Lecturer & exercise leader
Mattias Ohlsson
mattias@thep.lu.se
046-222 77 82

Exercise leader
André Larsson
andre@thep.lu.se
046-222 34 94

Chapter 2

Introduction to programming

2.1 What is a computer

- A computer takes data from one or more input *device*s, processes it and sends it to one or more output devices. The computer is a *data machine*.
- An input device can be the keyboard, the mouse, the network, a disk, a microphone, a camera, a scanner, ...
- An output device can be a disk, the network, a screen, a braille display, a loudspeaker, a printer, ...
- Every time you move the mouse, you generate lots of data which is processed by the computer.
- A disk and the network are both input and output devices.
- The processing is performed by the central processing unit (CPU) which uses a primary random access memory (RAM) to temporarily store the data it is processing.

2.2 What is data

- Data consists of binary digits – bits. Each can be 0 or 1.
- Eight bits make up a *byte*.
- Four (or eight, depending on you computer) bytes make up a *word*.
- The data does not have any inherent meaning. It is just ones and zeros.
- The CPU typically have four inherent ways of interpreting data. One or two words can be interpreted as an *integer* number, a *floating point* number, an *address* of another word in the RAM or an *instruction*.
- Any other interpretation of the data is defined by the program which is running.

- We shall mostly be concerned with *characters*, but the CPU does not know what a character is. Instead characters are internally represented by integer numbers corresponding to the *code* of a character.
- Characters can be encoded in many ways. The standard encoding is called ASCII which can describe 256 different characters, i.e. a character fits in a byte. In the future one may hope that UNICODE encoding will take over. This uses two bytes and can handle almost all characters, even Chinese ones. Perl uses ASCII, but can handle unicode aswell.

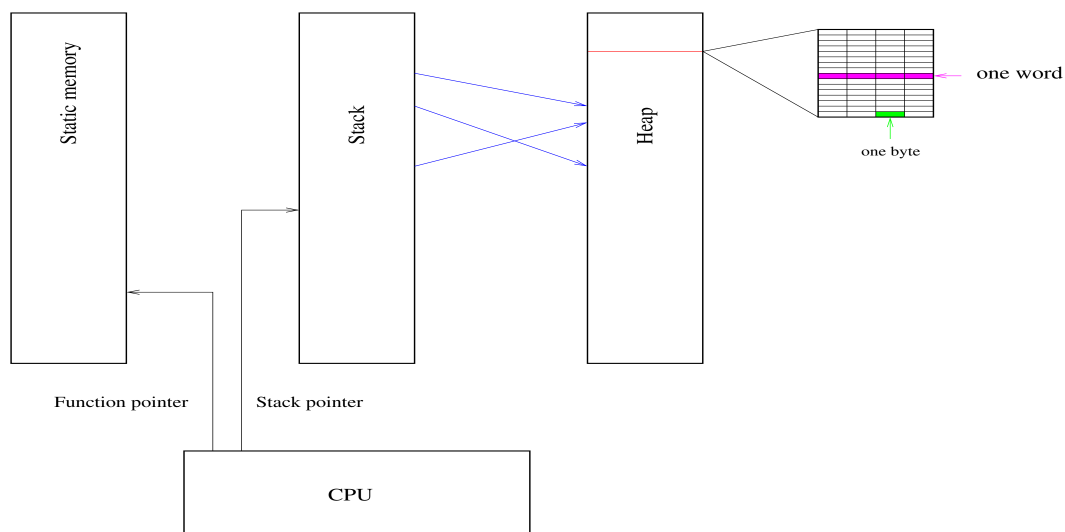
2.3 What is a program

- A program tells the CPU where to get the input data, what to do with it and where to put the result.
- A program is a sequence of very simple instructions. The CPU only knows how to do very simple things such as
  - Take two words at given addresses in the RAM and, treating them as integers, add them together and put the result at a third address.
  - Take a word from a device and put it at a given address in the RAM.
  - Jump to a given address in the RAM and continue reading instructions from there.
  - Check the value of a bit and if it is zero, jump to a given address in the RAM and continue reading instructions from there, otherwise continue reading the next instruction.
  - Take two words at two addresses in the RAM and, treating them as floating point numbers, multiply them together and put the result at a third address.
- It would be impractical if we had to write each of these instructions by hand. To be able to tell the computer what to do we need to have a high level *language* which can be translated into the basic instructions understood by the CPU.
- The instructions are grouped into *functions* / *subroutines* / *procedures* / *subprograms* which can do more complicated tasks.
- Functions are grouped into programs.
- A program is also data. It can be read in from a device and executed.

2.4 Operating system

- The operating system (OS) is *The Mother of all programs* in a computer. It runs all the time and allows other programs to execute.

- When you *bootstrap* a computer it reads in the OS from the disk, and starts executing it.
- In this course we will be using *Linux*.
- Just as most modern OS's, Linux allows for having several programs running seemingly simultaneously. This is achieved by continuously switching between reading instructions from different programs (*processes*).
- Typically there are a lot of programs running which we are not aware of. There is one program which is handling the drawing on the screen. Another listening for input from the keyboard, a third taking care of network connections etc.
- When we run a Perl program, we will be using a program giving a *terminal window*. In the terminal there is a program running which interprets our commands – a *shell*. The shell listens for keyboard input from another program, and the screen output is sent to the terminal program which sends it on to the *X-server* which draws on the screen.
- The OS logically divides the RAM available for a program into three different parts.



- The program is stored in the *static* memory. The *local* data for the currently executing function is stored in the *stack* and all other data is stored on the heap.
- When a function is executed its local data is stored at the top of the stack. When the functions is finished, this memory is released and will be used by the next function called. To create data which should survive after a function call, it must be stored on the heap.
- The OS also defines a *file system*.
- In Linux there are different kinds of files, and they do not always represent data stored on a disk. A directory is a special file which represents a set of other files. Executable programs are also represented by special files.

- The file system is organized in a hierarchical structure which may be spread out on several disks. All files can be accessed from the *root* directory (`/`). The same physical file can be represented at several places in the hierarchy using (*soft* and *hard*) *links*.
- Devices are also represented by special files. E.g. `/dev/mouse` looks like a file, but is actually directly connected to the mouse device.
- When we talk about input and output we need only concern ourselves with *files*.

2.5 Languages

- We use a programming language to tell the computer how to interpret the data and what to do with it.
- The first languages concentrated on the *what to do with the data* part, while modern languages are more concerned with how to represent the data – allowing the programmer to define new types of data suitable for the problem to be solved. The former are referred to as *procedural* languages, while the latter are *object oriented*. (If you are a really cool computer scientist, even object orientation is now passé – the new funky thing is *functional* programming).
- The term programming *languages* is not a misnomer. They really are languages in the same sense as English or Swedish.
- If you are good at learning foreign languages you are usually good at learning programming languages.
- Normal languages contains words, expressions and sentences. Computer languages contains tokens, expressions and statements.
- The main difference is that the computer is extremely picky when it comes to grammar. If you make the slightest mistake, the computer pretends it doesn't understands anything.
- Similarly to the case of spoken languages, the only way to learn a programming language fluently is to sit by the computer and write programs. Programming is 50% in the brain and 50% in the fingers.
- A file containing a text (*code*) written in a program language needs to be translated into the basic instructions which the CPU understand. This can be done in two ways.
- The code can be *compiled*, producing an executable file containing the basic instructions. Typically the code is compiled into an object file which is then combined together with other object files to produce an executable. One object file contains the *main* function and it needs to be *linked* together with other object files containing functions which need to be called.
- Alternatively the code is read by an *interpreter* program which translates the code on the fly and executes the corresponding instructions.

- Interpreted code is always slower than compiled code.
- A file with code for an interpreted language can be made to look as an executable file if it starts with the characters `#!` followed by the filename representing a program which can interpret the code.

2.6 Perl

- Perl is an interpreted language.
- Perl works with text, sequences of characters arranged in *strings*. All input and output is sequences of characters.
- Perl was first mainly used by system administrators who needed to make *quick-and-dirty* scripts (*hacks*) for boring repetitive tasks. It is full of *handy* little features.
- A file with Perl code which starts with the line `#!/usr/bin/perl` can be made into an executable.

2.7 Hello World

```
1  #!/usr/bin/perl
2
3  print "Hello World!";
```

- This is a Perl program with one *statement*.
- A statement is an expression, or a several expressions connected by *operators* (which may span several lines), ended by a `;`
- This expression consists of two words. One identifier (`print`) refers to a *function*. The other is a *string literal*.
- The statement means: Execute the function `print` with an *argument* which is a string.
- When run, this program will write out 'Hello World' on the *standard output*. Normally the standard output is the window where you started the program.

2.8 Procedural vs. Object Oriented programming

- Perl is a procedural language with some support for object oriented programming.
- The data can be strings or numbers and we call functions (procedures) to operate on the data.

- The first week we will write simple programs which call builtin standard functions to perform simple tasks.
- The second week we will explore more builtin functions and build our own functions which we can call from a main program and from other functions.
- It is important to understand the difference between global variables (on the heap) and variables which are local to a function (located on the stack) and cannot be accessed outside of the function.
- It is also important to understand how *arguments* are passed to the functions (*by value* or *by reference*).
- We will use Perl *objects* in the end of the course.
- We will, however, not learn how to construct *classes* of objects ourselves.
- Classes are combinations of variables and *methods* defined for them.
- In this way we can have variables which represents almost any concept, rather than just strings and numbers.
- Functions are verbs, Objects are nouns.

2.9 Comments

- In most programming languages you can add comments to the code which are ignored by the compiler/interpreter. This is used to increase the readability of the code.
- In Perl, everything between a `#` and the end of a line is treated as a comment.
- It is important to add comments to your code. But remember that Perl is itself a language and people reading the code is supposed to understand that language, so do not comment *obvious code*:

```
1  #!/usr/bin/perl
2  print "Hello World!"; # prints Hello World! to the screen.
```

2.10 Solving problems with Perl

- Even if we are only writing a single main program it is essential that we thoroughly think through what we are going to do before we start writing the code.
- For a procedural language the standard method is called *stepwise refinement.*
- First think about the general outline of the problem. Divide it into subproblems, and then consider each subproblem and try to divide them, in turn, into subproblems.

- It is like writing a recipe for cooking a meal. You do not start with *boil some rice, chop the onion* and then *oh, don't forget to add salt to the water and maybe you want to rince the rise first*. Rather you would first specify the ingredients, then you would specify the main steps, such as *boiling the rice*, *frying the veggies* and *making the sauce*. For each of these you would then list the actual procedure.

2.11 Hand-in exercises

- All the hand-in exercises in this course will require you to write Perl programs.
- We do not require you to write any reports describing the programs, but we do require you to comment your code in a special way so that we can follow how you have been thinking when you wrote them.
- The program must start with a comment section which describes the general outline of the code, followed by a description of the procedure used.
- We strongly recommend that you start by writing this comment section **before** you begin to write the code.
- The comment block should contain the following headings:
  - Title
  - Author
  - Description
  - List of subroutines
  - Procedure
  - Usage

  Here is an example:

```
1  ##### Program description
2  #
3  # Title: lotto.pl
4  #
5  # Author(s): Mattias Ohlsson
6  #
7  # Description:
8  #    This program will estimate the probability of winning a game of
9  #    Lotto. The method used here is a Monte Carlo simulation of the
10 #    lottery. This means that 7 numbers are randomly generated from
11 #    the interval [1,35] without doublets and then 7+4 winner numbers
12 #    are randomly generated from the same interval. According to the
13 #    rules of Lotto the following situations will result in a winnings:
14 #
15 #    4   correct numbers
```

```
16 #    5   correct numbers
17 #    6   correct numbers
18 #    6+1 correct numbers
19 #    7   correct numbers
20 #
21 #
22 # List of subroutines:
23 #
24 #    iRnd: This function takes two integers, i1 and i2, as
25 #       arguments and returns random integer in the interval [i1, i2].
26 #
27 #    GetNumbers: This function returns a vector of random numbers
28 #       in the interval [1,35], with all numbers different from
29 #       each other. The length of this vector is given as an
30 #       argument and must be <= 35.
31 #
32 # Procedure:
33 #    1. Get the number of Monte Carlo runs to perform from STDIN.
34 #    2. Start the main Monte Carlo loop
35 #       2a. Get an array of 7 numbers (my Lotto row) and one with 11
36 #           numbers (the "winner numbers").
37 #       2b. Make a loop over all the 7 numbers and check if each of
38 #           them are present in the array of 7+4 winner numbers. Add
39 #           hits to the two variables $Corr7 and $Corr4, representing
40 #           the real hits or "tilläggsnummer" hits.
41 #       2c. Check the possible combinations for a winnings and record
42 #           such an event. Also record the different combinations.
43 #    3. Print a summary of the results found.
44 #
45 # Usage:
46 #       ./lotto.pl [number of MC runs]
47 #
48 #
49 ##################################
```

2.11.1 Rules for the hand-in exercises

There a few rules for the hand-in exercises:

- All 4 hand-in exercises are mandatory, you need to get an OK on each of them in order to pass the course.
- They are used together with the written exam to make up the final grade.
- For most of the hand-in exercises there are sample/result output files. When running your Perl program you should get “similar” results. For some exercises you need to comply exactly with the sample files.

- Before you hand in the exercises make sure that:
  - the program contains a documented program header.
  - the program runs without errors and warnings.
- You should all hand in individual solutions, but it is of course allowed (and even recommended) to ask questions, to discuss etc with other students.
- Have fun!