# Chapter 3

# Basics in Perl

## 3.1 Variables and operations

### 3.1.1 Scalars

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

- `$hello` is a *scalar variable*. It represents an area in the memory where you can store data.
- `hello` is an *identifier* (name). It represents the address of the memory area. The `$`-sign means that we want to access the actual memory area.
- `$hello = "Hello World!"` means: *Store the string of characters `"Hello World!"` in the memory area represented by `hello`.*
- A scalar variable can be in three different states: If nothing has been assigned to it, it is *undefined*. Otherwise it may contain a string or a number.
- Unlike most other programming languages you do not have to declare what a variable should contain.
- A variable name can be any letter from the English alphabet or an underscore followed by other letters, numbers or underscores.

### 3.1.2 Strings

- There are two ways of specifying string literals: Any sequence of characters surrounded by single or double quotation marks:

```
"This is a string"
'This is another string'
```

- The difference between the two forms is the way *special* characters are interpreted.
- If you start a string with a single (double) quote, another single (double) quote is a special character which ends the string.
- To actually have a single (double) quote character *in* a string you have to have to have it in a string surrounded by double (single) quotes: `"'"` is a single quote and `'"'` is a double quote.
- To prevent a special character to be interpreted as special, they can be *escaped* with a *backslash*: `'\''` is a single quote and `"\""` is a double quote.
- A double quoted string has a lot of special characters, a single quote only have two (the single quote and the backslash).
- A backslash can also be used introduce special characters in a double quoted string. E.g. `"\n"` means the *new-line* character and `"\t"` means the *tab* character. We call these *escape sequences*.

### 3.1.3 Numbers

- Number literals are specified in the standard computer way:

```
1
42
28.2
-109
6.04E23
```

- Note the decimal *point* is used rather than the Swedish decimal *comma*.
- The `E` should be interpreted as *times ten to the power of*. Hence `6.04E23` means $6.04 \cdot 10^{23}$.
- Note that decimal numbers are not exact in Perl (or in any other language).

```
1 print 0.3+0.6-0.9;
```

will print out

```
-1.11022302462516e-16
```

or something similar (depending on the hardware).

### 3.1.4 Operators

- The standard arithmetic operators are available and can be used for variables and number literals.
- We have addition `+`, subtraction `-`, multiplication `*`, division `/`, modulus `%` and exponentiation `**`.
- `3 + 4 * 2` is processed as `3 + (4 * 2)`, i.e. the operators have different *precedence*. To change this we can group expressions with parenthesis: `(3 + 4) * 2`.
- If arithmetic operators are used with variables, the variables are assumed to contain numbers.

```
1 $a = 4;
2 $b = 2;
3 $c = 3 + $a * $b;
4 print $c, "\n";
```

  will print out `11`.

- If a variable does not contain a number when one is expected, Perl will try to convert it to a number. An undefined variable is interpreted as `0`. A variable containing a string is parsed to try to read a number - if that fails it gives `0`:

```
1 $a = "four";
2 $b = "two";
3 $c = 3 + $a * $b;
4 print $c, "\n";
```

  will print out `3`. Mixing numbers and strings when doing arithmetics is not a good idea, even though Perl can be forgiving. The following code is an example of this. Again, relying on this behaviour will most certainly lead to bugs.

```
1 $a = " \n4";
2 $b = "2 blah";
3 $c = 3 + $a * $b;
4 print $c, "\n";
```

  will print out `11`.

- This is different from other computer languages where operators have different meaning depending on the type of variables. E.g. `+` typically means concatenation if the arguments are strings.
- To concatenate two strings in Perl one uses the `.` operator (or *variable interpolation* below).

```
$a = "four";
$b = "two";
print $a . $b, "\n";
```

will print out `fourtwo`. and

```
$a = 4;
$b = 2;
print $a . $b, "\n";
```

will print out `42`.

- `=` is the assignment operator. Note that all expressions in Perl has a value associated to them. Also an expression involving an assignment has a value (the value is the variable which you have assigned to) and can itself be assigned to a variable.

```
$c = $b = $a = 4;
print "$c$b$a\n";
```

will print out `444`, and

```
print $c = $b = 5, "\n";
```

will print out `5`.

### 3.1.5 Shortcut operators

- Some combinations of operators have shortcuts.
- As an example `+=` is the add and assign operator. It is defined so that `$a += $b` is equivalent to `$a = $a + $b`.
- Similarly `$a -= $b` is equivalent to `$a = $a - $b` and `$a *= $b` is equivalent to `$a = $a * $b`, etc.
- `++` is the increment and `--` is the decrement operator. They are defined so that `$a++` is equivalent to `$a = $a + 1` and `$a--` is equivalent to `$a = $a - 1`

```
$a = 10;
$a = $a - 1;
$a--;
$a -= 4;
print $a, "\n";
$a++;
```

```
7 $a += 10;
8 print $a, "\n";
```

will print 4 and 15.

### 3.1.6 Variable interpolation

- For double quoted strings we have seen that the double quote is a special character.
- Another special character is `$`. When followed by a variable identifier, the contents of the variable will be inserted:

```
1 $a = 4;
2 print "The value of \$a is $a\n";
```

will print out `The value of $a is 4`. Note that the number will be converted to a string.

- This is called *variable interpolation*. It only affects the variable itself so

```
1 $a = 4;
2 print "The value of \$a+1 is $a+1\n";
```

will print out `The value of $a+1 is 4+1`. To perform the addition we can do

```
1 $a = 4;
2 print "The value of \$a+1 is", $a+1, "\n";
```

which will print out `The value of $a+1 is 5`.

### 3.1.7 Arrays and lists

- An *array* can be used to store several strings or numbers referenced by a single identifier.
- An array is identified with the `@` character.

```
1 @numbers = ( 1, 2, 3, 4, 5 );
```

is an array which contains five numbers which are indexed with a number $(0 \ldots 4)$. To access one of the numbers one gives the identifier followed by an index inside square brackets.

```
1 $numbers[3] = "four";
2 print "$numbers[0]\n";
```

- Note that we use a `$` to get the contents. `@numbers` is the whole array, while the indexation selects one of the scalars in the array, hence the `$` sign is used to show that we are dealing with a scalar.

- Inside the square brackets we can put any expression which evaluates to a number:

```
1 $a = 2;
2 print "$numbers[$a+$a]\n";
```

  will print out `5`, i.e. the expression inside the square brackets is evaluated even if it is inside a double-quoted string.

- A *list* can be assigned to an array: `(1, 2, 3, 4, 5)` is a list. Also `(1, $a, 3, "four", 5)` is a list.

- If a list only contains variables, they can be assigned to:

```
1 @numbers = ( 1, 2, 3, 4, 5 );
2 ( $one, $two, $three ) = @numbers;
3 print "$two\n";
```

  will print out `2`.

- There are a number of *functions* which acts on arrays, e.g. `split`, `join`, `push` and `pop`[1]

- `split` will take a string and split it into an array of string.

```
1 $a = "This is a, rather silly, sentence";
2 @words = split ' ', $a;
3 print "$words[0] $words[2]";
```

  will print `This a,` while

```
1 $a = "This is a, rather silly, sentence";
2 @words = split ',', $a;
3 print "$words[0] $words[2]";
```

  will print `This is a  sentence`

- `join` will take a list or array and concatenate all items into one string.

```
1   @numbers = ( 1, 2, 3, 4, 5 );
2   $a = join ':', @numbers;
3   print "$a\n";
```

[1] We will discuss these function in more depth in section 4.1

will print out `1:2:3:4:5`. Note that when an array is interpolated inside a double-quoted string, it will be implicitly converted into a string with `join ' ', @anarray`

- `push` will add a scalar to the end of an array.

```
1 @numbers = ( 1, 2, 3, 4, 5 );
2 push @numbers, 6;
3 push @numbers, 7;
4 print $numbers[5] + $numbers[6], "\n";
```

will print out `13`.

- `pop` will return the last element of an array and then remove that element from the array.

```
1 @numbers = ( 1, 2, 3, 4, 5 );
2 $a = pop @numbers;
3 $b = pop @numbers;
4 print $a + $b, "\n";
```

will print out `9`.

- Note that simply assigning to a non-existing element in an array will expand the array, adding undefined elements as necessary.

```
1 @numbers = ( 1, 2, 3, 4, 5 );
2 $numbers[9] = 10;
3 print @numbers + 0, " (@numbers)\n";
```

- In the previous snippet, the `numbers` array initially only contains five elements (0-4). When assigning to element indexed 9, the array is expanded with undefined elements indexed 5-8, whereafter the last element is assigned the number 10.

- Note that when an array is used where a number is expected as in `@numbers + 0`, the number of elements in the array is used. As mentioned above, when an array is interpolated in a double-quoted string, the elements are concatenated with `join ' ', @numbers` and since undefined elements will be written out as empty strings, the code above will print out `10 (1 2 3 4 5 10)`.

### 3.1.8 Hashes

- A *hash* can also be used to store several numbers and strings in the memory. But they are not ordered with indices as arrays are. In a hash, the elements are accessed with *user-defined* indices, or *keys*, represented by strings inside curly brackets.

- A hash is identified with the `%` sign.

```
%numbers;
$numbers{'one'} = 1;
$numbers{'two'} = 2;
$numbers{'three'} = 3;
$numbers{'four'} = 4;
print $numbers{'four'} + $numbers{'one'}, "\n"
```

will print out 5.

- An array or a list can be assigned to a hash. In that case the first element will be interpreted as the index of the second element, the third as the index to the fourth and so on.

```
%numbers = ( "one", 1, "two", 2, "three", 3, "four", 4);
$a = "four";
print $numbers{$a} + $numbers{'one'}, "\n";
```

will print out 5.

- Not that we can have any string as key, and any expression given inside the curly brackets will be converted to a string and used as key.

- Another way to define a hash is to use the => notation

```
%aminos = ('A' => 'Ala', 'C' => 'Cys', 'D' => 'Asp');
print "The three letter abbreviation for Alanine is $aminos{'A'}\n";
```

- The elements in a hash can be extracted into an array, and after `@vals = values %numbers;` the `vals` array will contain the values `1`, `2`, `4` and `3`, but in *no predictable order*.

- Also the keys of a hash can be extracted into an array and after `@kks = keys %numbers;` the `kks` array will contain the strings `"four"`, `"one"`, `"three"` and `"two"`, again in no predictable order (although the same order as the corresponding values obtained from `values`).

### 3.1.9 Follow-up tasks 3-1

1. Execute the code snippets in this section and **make sure** you understand them.

2. You are strongly encouraged to play around with the examples and test anything that comes to your mind.

3. Also play around with the following code examples:

```
$a = 9;
$b = 3;
$b = $a + $b;
print $b, "\n";

$a = 4;
$b = 3;
$a = $a + $b;
$b = $a + $b;
print $a, "\t", $b, "\n";

$a = 5;
$b = 6;
$c = 5 + $a * $b;
print $c, "\n";

$a = 3;
$b = 2;
$c = $a**$b;
print $c, "\n";

$a = 5;
$b = 2;
$c = $a%$b;
print $c, "\n";

$a = "high";
$b = "low";
$c = $a . " or " . $b;
print $c, "\n";
```

4. and practice a bit on variable interpolation:

```
$a = ' \t2' + 2;
$b = '  t2' + 2;
print $a, "\t", $b, "\n";

$a = "oj";
$b = $a;
$c = "$a";
$d = '$a';
print $a, "\t", $b, "\t", $c, "\t", $d, "\n";

$a = "aj";
$b = $a;
$c = "$a";
$d = '$a';
```

```
15 print "$a \t $b \t $c \t $d \n";
```

5. Do you understand all the shorthand notations?

```
1  $a = 4;
2  $a += 4;
3  $b = 10;
4  $b -= 1;
5  print "$a\t$b\n";
6
7  $a = 4;
8  $a += 2 + 2;
9  $b = 10;
10 $b -= $a;
11 print "$a\t$b\n";
12
13 $a = 3;
14 $a++;
15 $b = $a;
16 $b--;
17 print "$a\t$b\n";
```

6. If you have spare time, write some small Perl programs and ask another student to predict their output. Make them as tricky as you want, as long as you understand them yourself.

## 3.2 Basic input/output

- Input and output to and from a Perl program is handled by *files*.
- Files are identified in the program by *filehandles*.
- There are three standard filehandles which are always automatically defined: `STDIN` (*standard input*), `STDOUT` (*standard output*) and `STDERR` (*standard error*). They are all associated with the terminal window from which the Perl script is called.
- The full form of the print function is
  `print` *file-handle expression* `[,` *expression* `...]`
  If no filehandle is specified, Perl assumes you mean `STDOUT` (an example of Perl *slang*).
- `STDOUT` is the special file associated with the output to the shell, or terminal window from which the script is called.
- In a Unix *shell* (the program which reads your keystrokes in a terminal window) it is possible to *redirect* the output from a program to a file or to another program.

```
prompt> ./myscript.pl > afile.dat
```

will take the output and write it to a file called `afile.dat`. **Note:** the command prompt in your terminal window is represented by '`prompt>`'.

- `STDERR` is also connected to the terminal window. If the Perl script in `myscript.pl` above contained a statement

```
1  print STDERR "Goodbye World!";
```

that message would not be redirected to `afile.dat`, but would be written to the window. Note that on a Mac or a Windoze machine there is no difference between `STDOUT` and `STDERR`.

- If you continue typing after you have started a Perl script, what you type may be read into the program from the `STDIN` filehandle. The following program will read one line from `STDIN` and print it back to `STDOUT`.

```
1  #!/usr/bin/perl
2  $line = <STDIN>;
3  print $line;
```

- The `<STDIN>` expression (or `<>`, as Perl assumes you mean `STDIN` if you do not specify a file handle) causes the Perl interpreter to wait for input from the terminal, and when it reads a *new line* character (or an *end-of-file* character) it will return a string of the characters given, *including* the new-line character.

- Any standard file ends with an end-of-file character. From the shell, you can indicate the end of the input by pressing *ctrl-d*. If only an end-of-file character is read by `<>`, the value returned is *undefined*.

- Also `STDIN` may be redirected:

```
prompt> ./myscript.pl < afile.dat
```

read from the file `afile.dat` instead from the keyboard.

- There is one important difference between `<STDIN>` and `<>`. In the latter case the following

```
prompt> ./myscript.pl file1.dat file2.dat file3.dat
```

cause the script to first read from `file1.dat` and then, when the end of that file is reached, continue to read from `file2.dat` etc.

- It is possible to redirect the output of one script to the input of another through a *pipe*:

```
prompt> ./onescript.pl | ./anotherscript.pl
```

  will take the output from `onescript.pl` and send it as input to the `anotherscript.pl` script.

- The following program will ask you to write your name and will then write a greeting:

```
1 #!/usr/bin/perl
2 print "Hi there! What's your name?\n";
3 $name = <STDIN>;
4 print "Howdy $name, nice to meet you!\n";
```

  If your name is Frodo the following will happen in your window

```
prompt> ./hello.pl
Hi there! What's your name?
Frodo
Howdy Frodo
, nice to meet you!
```

  This is because also the new-line character is included in the string that was read in. To get rid of the new-line character we can use a function `chomp`:

```
1 #!/usr/bin/perl
2 print "Hi there! What's your name?\n";
3 $name = <STDIN>;
4 chomp $name;
5 print "Howdy $name, nice to meet you!\n";
```

  `chomp` will remove a trailing new-line character of the scalar given as argument and the output of the script will look nicer.

- We could have written `chomp ($name = <>)` which is Perl slang for *read a line from standard input, assign the resulting string to the scalar $name and then send the scalar as argument to the chomp function.*

### 3.2.1 Follow-up tasks 3-2

1. Execute the code snippets in this section and **make sure** you understand them.

2. Type in the following Perl script which simply copies lines from the standard input to the standard output prefixing each line with the line number:

```
#!/usr/bin/perl
$i = 0;
while ( $line = <> ) {
  $i++;
  print "$i: $line";
}
```

Note that this script make use of a *while-loop* and we have not talked about loops yet. Just see it as a way of reading lines from '<>' and print it on the terminal. Name the script `cat.pl` and run it in a terminal window.

```
prompt> ./cat.pl
```

Type in a couple of lines and end with *ctrl-d*. Also try the following examples:

```
prompt> ./cat.pl cat.pl
prompt> ./cat.pl cat.pl cat.pl
prompt> ./cat.pl cat.pl > test.dat
prompt> ./cat.pl < test.dat
prompt> ./cat.pl cat.pl cat.pl | ./cat.pl
```

Make sure you understand what happens!

3. Change the input statement in the script to `$line = <STDIN>` and run the above examples again.

4. Change the original `cat.pl` to look like this:

```
#!/usr/bin/perl
$i = 0;
while ( $line = <> ) {
  $i++;
  print STDOUT "(stdout)-$i: $line";
  print STDERR "(stderr)-$i: $line";
}
```

Now test the following two examples:

```
prompt> ./cat.pl cat.pl
prompt> ./cat.pl cat.pl > test.dat
```

and check the content of `test.dat`. Draw your own conclusions about the difference between `STDOUT` and `STDERR`.

5. With the `wget` program you can retrieve any document on the web and *pipe* it into your program (the first version `cat.pl`). Try

```
prompt> wget -q -O - http://www.thep.lu.se/ | ./cat.pl
```

6. Here is another script which copies the standard input to the standard output.

```
1 #!/usr/bin/perl
2 @line = ( <> );
3 print @line;
```

   All lines are read into the array, one line per element, and then `print` prints each of the elements in the array. Run the script in different ways.

7. Change the print line to `print "@line";` and run it in different ways. The array is interpolated by concatenating all elements separated by a space character.

8. Change the print line to `print @line + 0, "\n";` In this case the length of the array is printed. Why? When `@line` is used in a *scalar context* (eg. when a number is expected) the length of the array is used. It is perfectly valid to write an expression like, `$len = @line;`, meaning that `$len` is now the number of elements in `@line`. We will see this behaviour in Perl many times.

## 3.3 Loops and conditions

### 3.3.1 Condition for one statement

- A statement can be made conditional by ending it with an *if*-statement.
- In the following, the print expression will only be executed if your name is not the one of a well-known dark lord:

```
1 #!/usr/bin/perl
2 print "Hi there! What's your name?\n";
3 $name = <STDIN>;
4 chomp $name;
5 print "Howdy $name, nice to meet you\n" if ( $name ne "Sauron" );
```

- `if` must be followed by a *boolean* value which can be *true* or *false*. In this case the expression uses the *comparison operator* `ne` which returns *true* if the strings on each side are not identical. The expression before `if` will only be executed if the boolean expression is `true`.
- There are several other comparison operators for strings: `eq` returns true if the strings on each side are identical, `lt` (`le`) if the left side string is lexically less than (or equal to) the right side string, `gt` (`ge`) if the left side string is lexically greater than (or equal to) the right side string.

- There are also comparison operators for numbers. `==` tests for equality (note the double equal signs), `!=` test for inequality. `>`, `>=`, `<`, `<=`, should be self explanatory.
- The `if` statement can be replaced by an `unless` statement for which the preceding expression will only be executed if the following expression is *false*:

```
1 #!/usr/bin/perl
2 print "Hi there! What's your name?\n";
3 $name = <STDIN>;
4 chomp $name;
5 print "Howdy $name, nice to meet you!\n" unless ( $name eq "Sauron" );
```

- `die` is a useful function which writes the string given as argument to `STDERR` and terminate the program with a message:

```
1 #!/usr/bin/perl
2 print "Hi there! What's your name?\n";
3 $name = <STDIN>;
4 chomp $name;
5 die "Be gone you evil dictator!" if ( $name eq "Sauron" );
6 print "Howdy $name, nice to meet you!\n";
```

### 3.3.2 Condition for compound statements

- It is possible to make whole *blocks* of statements conditional.

```
1 #!/usr/bin/perl
2 print "Hi there! What's your name?\n";
3 $name = <STDIN>;
4 chomp $name;
5 if ( $name eq "Sauron" ) {
6    print "Be gone you evil dictator.";
7 }
```

- A block of statements is any number of statements surrounded by curly brackets.
- The boolean expression must be enclosed in parenthesis.
- The block after the `if` will only be executed if the condition is true.
- If the `if`-block is directly followed by `else` and another block the latter will be only be executed if the `if`-condition is false.

```
1 #!/usr/bin/perl
2 print "Hi there! What's your name?\n";
3 $name = <STDIN>;
4 chomp $name;
```

```
5  if ( $name eq "Sauron" ) {
6     print "Be gone you evil dictator.";
7  } else {
8     print "Howdy $name, nice to meet you!\n";
9  }
```

- It is possible to have more than two alternatives:

```
1  #!/usr/bin/perl
2  print "Hi there! What's your name?\n";
3  $name = <STDIN>;
4  chomp $name;
5  if ( $name eq "Sauron" ) {
6     print "Be gone you evil dictator.";
7  } elsif ( $name eq "Gandalf" ) {
8     print "Hello there wizard!\n";
9  } else {
10    print "Howdy $name, nice to meet you!\n";
11 }
```

- The `elsif` block will only be executed if the `if`-condition was false (and possible previous `elsif`-conditions were false) and the `elsif`-condition is true. The optional final `else`-block will only be executed if the `if`-condition and all following `elsif`-conditions were false.

### 3.3.3 What is truth?

- In the real world the truth is a difficult concept, but in Perl the truth is well defined.
- Several things may be used as a conditional (boolean) expression.
- We have already seen comparison expressions: `$a == $b` or `$a ne $b`
- Any numerical expression which evaluates to zero is *false*, any other numerical expression is *true*.
- An empty string (`""`), or a string with a single character zero (`"0"`) is *false*. Any other string is *true*.
- An undefined variable is *false*.
- Note that it is easy to make a mistake with `=` and `==`. Assignment gives a value which is may be interpreted as true or false: `if ( $a = 10 )` will always execute the following block since the value is `10` which is true.

### 3.3.4 Logical operators

- It is possible to combine boolean expressions with operators.
- `!` or `not` will give true if the following expression is false and vice versa.
- `||` or `or` will give true if either the lefthand side (lhs) or the righthand side (rhs) is true.
- `&&` or `and` will give true if both the lhs and rhs is true.
- `xor` will give true if either the lhs or rhs are true but not if both are.
- `||` and `or` are special in that if the lhs is true the rhs will not be evaluated since it is already clear that the whole expression is true. This can also be used outside of `if` expressions:

```
1 $len = @arr or die "Zero length array!";
```

- Similarly for `&&` and `and`, if the lhs is false, the rhs will not be evaluated.

### 3.3.5 Loops

- You will always run into situations where you will need to do (almost) the same thing over and over again. Rather than writing the same code over and over, you may then make a loop over a block of statements.
- Rather than

```
1 print "Hello Adam, nice to meet you!\n";
2 print "Hello Bert, nice to meet you!\n";
3 print "Hello Carl, nice to meet you!\n";
4 print "Hello Dave, nice to meet you!\n";
5 print "Hello Eric, nice to meet you!\n";
```

- You can make a `foreach` loop:

```
1 foreach $name ( "Adam", "Bert", "Carl", "Dave", "Eric") {
2    print "Hello $name, nice to meet you!\n";
3 }
```

- `foreach` will assign to the given scalar each of the elements of the subsequent list (or array) in turn and execute the following block. After all elements have been used, the execution continues after the block. The most common use of `foreach` is to loop over the elements of an array:

```
@names = ( "Adam", "Bert", "Carl", "Dave", "Eric");
foreach $name (@names) {
   print "Hello $name, nice to meet you!\n";
}
```

### 3.3.6 More loops

- It is possible to control loops with a running index:

```
@names = ( "Adam", "Bert", "Carl", "Dave", "Eric");
for ( $i = 0; $i < 5; $i = $i + 1) {
   print "Hello $names[$i], nice to meet you!\n";
}
```

  This means *set $i to 0, then as long as $i is less than 5 execute the following block. And after each time the block is executed, add 1 to $i.*

- The same thing can be accomplished with a `while`-loop:

```
@names = ( "Adam", "Bert", "Carl", "Dave", "Eric");
$i = 0;
while ( $i < 5 ) {
   print "Hello $names[$i], nice to meet you!\n";
   $i = $i + 1;
}
```

- Just as in spoken languages there are usually many different ways of doing the same thing in a programming language. Perl is almost extreme in this respect.

- `until` is for `while` what `unless` is for `if`:

```
$i = 0;
until ( $i == 5 ) {
   print "Hello $names[$i], nice to meet you!\n";
   $i = $i + 1;
}
```

- The `++` operator adds 1 to its argument, `$#`*array* gives the index number of the last element in an array:

```
for ( $i = 0; $i <= $#names; $i++ ) {
   print "Hello $names[$i], nice to meet you!\n";
}
```

There is also a `--` operator and a `+=` and `-=` operator:

```
for ( $i = 0; $i < @names; $i += 2 ) {
   print "Hello $names[$i], nice to meet you!\n";
}
```

- If an array is given where a number is expected, the length of the array is used.
- `..` is the range operator and `($a .. $b)` will give the list `($a, $a+1, $a+2, ...$a+n)` where `n` is the largest number for which `$a+n <= $b`.

```
  foreach $i ( 0 .. $#names ) {
     print "Hello $names[$i], nice to meet you!\n";
  }
```

Note that also `( "a" .. "z" )` makes sense!

- We can use `while` to loop over all lines from standard input:

```
while ($name = <>) {
   chomp $name;
   last if ( $name eq "quit" );
   print "Hello $name, nice to meet you!\n";
}
```

In this while loop the scalar `$name` is assigned, in turn, one line of the standard input or lines from a file given at the command line (remember the `<>` function). This loop condition will end the loop when the *end-of-file* is reached. In addition, the loop will terminate if we write *quit* alone on a line, because of the `last` instruction. This instruction will also terminate other loops (`foreach`, `until` and `for`).

- Here we have another instruction to control the loop behavior - `next`. When this statement is executed all lines below the `next` statement are skipped for this turn in the loop. Hence the below code will sum all even integers from 2 - 10.

```
$sum = 0;
for ($i = 1; $i <= 10; $i++) {
   next unless ($i % 2 == 0);
   $sum += $i;
}
print "Sum of even integers [0-10] = $sum\n";
```

- For many loops you can use either `foreach`, `for`, `while` or `until`, but the default choice is often `foreach`. The rule is to use the construction which is logical for the problem at hand. `for` should be used when a more complex iteration scheme is needed. `foreach` should be used when you need to do something for each element of a list and simple iterations. Otherwise use `while` or `until`.

### 3.3.7 The magic of $_

- In many cases if you do not give a variable when and where Perl would expect one, the *default* variable $_ will be used instead.
- By using $_ you can write very sloppy code – don't do that.
- Here are some examples:

```
1 while (<>) {
2    chomp;
3    print "Hello $_, nice to meet you!\n";
4 }
```

- Each line from <> will be assigned to $_ since no variable is given. chomp will use $_ as argument since nothing else is specified. In the print-statement $_ is used as any other scalar variable.

```
1 foreach ( "Adam", "Bert", "Carl", "Dave", "Eric" ) {
2    print;
3 }
```

- foreach without a scalar puts each element in the list in $_. print without an argument will print $_.
- The use of $_ can result in the compact and effective coding, but is often errorprone and make the code difficult to read. So use it with care!

### 3.3.8 A complete program

- The following program will count the number of lines and number of characters read from standard input:

```
1 #!/usr/bin/perl
2 while ( $line = <> ) {
3    $len += length $line ;
4    $words += split ' ', $line ;
5    $lines++;
6 }
7 print "$len characters and $words words in $lines lines\n";
```

- with the use of $_ is can look like this

```
1 #!/usr/bin/perl
2 while ( <> ) {
3    $len += length;
4    $words += split ' ' ;
```

```
5     $lines++;
6  }
7  print "$len characters and $words words in $lines lines\n";
```

Note that the first time `$lines`, `$words` and `$len` are used they are undefined. This means that when they are used in a numeric expression they are interpreted as zero.

- `length` is a function which returns the number of characters in a string.
- `split` will return an array of the words in each line and will be converted to the number of elements in the array in the addition. More on this function in chapter 4.

### 3.3.9 New features: the switch statement

Perl 5.10 introduced a native "switch" statement using the given/when mechanism. Below is an example showing this:

```
1  use feature ":5.10";
2
3  $seq = <STDIN>;
4  chomp $seq;
5  given($seq) {
6
7     when ( 'DAP' ) {
8        say "Sequence is equal to 'DAP'";
9        continue;
10    }
11
12    when ( length($_) <= 3 ) {
13       say "Sequence contains less than or equal to 3 AA";
14    }
15
16    when ( @arr ) {
17       say "Sequence is found in the array \@arr";
18    }
19
20    default {
21       say "The default!";
22    }
23 }
```

- The `use feature ":5.10"` statement is needed to enable this new feature!
- The `when ()` conditions are checked one by one.
- If a `when ()` condition is true the corresponding block is run and the whole `given` block is ended.

- If a `continue` statement is part of one `when` block it will jump to check the next `when` block even if the former was true.
- The `default` is just what it is - the default behavior if no `when` block is true.

You can find more details about given/when here:
`http://perldoc.perl.org/perlsyn.html#Switch-Statements`

### 3.3.10 Follow-up tasks 3-3

1. Execute the code snippets in this section and **make sure** you understand them. You are, as always, strongly encouraged to play around with the examples and test anything that comes to your mind.

2. Write a Perl program that reads a number from the `standard input` and assign it to a variable, lets call the variable `$a`. (The number must be a positive integer.) The program should then repeat the following until `$a` is equal to 1:

```
1 IF ( $a IS EVEN )
2     DIVIDE $a BY 2.
3 ELSE
4     MULTIPLY $a BY 3 AND ADD 1.
5
6 PRINT $a
```

   Run the program with a number of different input values. Notice how this simple program gives an almost chaotic output. Hint: if `$a` is an even number the following is true: `$a % 2 == 0`.

3. The file `numbers.txt` contains a list of numbers. There is one number on each line. Write a Perl program that reads the file from the `standard input` and puts the numbers into an array. Then display the contents of the array. Hint: use the `while` loop for the flow control and the `push` function to push the numbers onto the array.

4. Loop through the array in the previous task and compute the sum of all the numbers. Divide the sum by the length of the array. You have now computed mean value of the numbers in file `numbers.txt`.

5. Find the biggest value and the smallest value of the numbers in the array.

## 3.4 Pattern matching

- Conditional expressions are very useful but sometimes `==` or `>` is not appropriate for what you want to do.

- How do you check if a given sequence of letters is included in a string? Well, we can `split` the string and then loop over the characters one at the time, and each time we find a character equal to the first one in our sought after sequence, we check if the following characters also matches up. But there is a simpler way.

### 3.4.1 Approximately equal to

- The `=~` operator compares the string on the lhs with the *regular expression* (regexp) on the rhs and checks if the former *matches* the latter.

- A regular expression is a sequence of characters enclosed between a pair of slash characters.

- To check if a given string contains the character sequence `GAATTC` we can do:

```
1 if ( $a =~ /GAATTC/ ) {
2   print "$a contains the sequence GAATTC\n";
3 }
```

- This is the simplest example of a regular expression. But they can become arbitrarily complicated.

### 3.4.2 Regular expressions

- There are a number of *meta* characters which have special meaning when used in a regular expression: `[]|\$^.+?*{}()...`

- A sequence of character *matches* exactly that sequence of characters. A sequence of character inside square brackets matches any *one* of those characters:

```
1 if ( $a =~ /B[aei]ll/ ) {
2    print "'$a' contains Ball, Bell or Bill\n";
3 }
```

- `|` is the *or* sign and we can use it to match one of several regular expressions:

```
1 if ( $a =~ /Ball|Bell|Bill/ ) {
2    print "'$a' contains Ball, Bell or Bill\n";
3 }
```

- You can already guess why regular expressions can be extremely helpful when looking through databases with DNA or protein sequences.

### 3.4.3 Meta characters

- Here are some metacharacters:
  - `.` matches any character except the newline character.
  - `^` matches the beginning of a line.
  - `$` matches the end of a line (except if it is directly followed by an identifier, in which case the variable is interpolated).
  - `\w` matches any *word* character (non-white-space, non-punctuation character).
  - `\W` matches any non-word character.
  - `\s` matches any white space character (tab, space, newline).
  - `\S` matches any non-white-space character.
  - `\d` matches any digit.
  - `\D` matches any non-digit.
  - `\b` matches the beginning of a word or the end of a word.
- We can now make a regexp for a Swedish personal ID number:

```
1 $a = 'This is a ID number : 123456-2345';
2 if ( $a =~ /\b\d\d\d\d\d\d-\d\d\d\d\b/ ) {
3    print "'$a' seems to contain an ID number\n";
4 }
```

- A matching expression can be repeated:
  - `\d` matches one digit.
  - `\d+` matches one or more consecutive digits.
  - `\d*` matches zero or more consecutive digits.
  - `\d?` matches zero or one digit.
  - `\d{24}` matches exactly 24 consecutive digits.
  - `\d{24,}` matches at least 24 consecutive digits.
  - `\d{24,42}` matches at least 24 but at most 42 consecutive digits.
  - `\d{,42}` matches at most 42 consecutive digits.
- The ID number match can be written

```
1 $a = 'This is a ID number : 123456-2345';
2 if ( $a =~ /\b\d{6}-\d{4}\b/ ) {
3    print "'$a' seems to contain an ID number\n";
4 }
```

- The regexp `/[AG]C[GATC]{40,80}[AG]C/` will match any sequence starting with `AC` or `GC` followed by at least forty but at most 80 arbitrary nucleotides and ending with `AC` or `GC`

- A regexp is grouped together with parenthesis. To match a time stamp on the form *hh:mm* or *hh:mm.ss* we can use the regexp `/\b\d\d:\d\d(\.\d\d)?\b/`. Here the `?` means zero or one matches of the whole expression inside the preceding parenthesis. Note that to match a point we have to escape the point character, otherwise it is interpreted as a metacharacter.

- The following will check if the file fed into the standard input is a Perl script:

```
1 #!/usr/bin/perl
2 if ( <> =~ /^#!\s*\/usr\/bin\/perl\b/ ) {
3    print "This is a Perl script!\n";
4 }
```

  The `^` metacharacter ensures that there is nothing before on the line. Note that the `/` character needs to be escaped otherwise it terminates the regexp.

- If we want to match several `/` characters, the regexp tend to look rather messy. It is, however possible to pick another delimiter using the the `m` modifier prefix:

```
1 #!/usr/bin/perl
2 if ( <> =~ m|^#!\s*/usr/bin/perl| ) {
3    print "This is a Perl script!\n";
4 }
```

- We can check how many lines in a Perl script contains only comments:

```
1 #!/usr/bin/perl
2 <> =~ /^#!\s*\/usr\/bin\/perl/ or die "This is not a Perl script!";
3 while ( <> ) {
4    $comments++ if (/^\s*#.*$/);
5 }
6 print "There were $comments comment lines in this script\n";
```

  A regexp by itself is slang for `$_ =~ /regexp/`.

- We can match any typical Swedish surname with like this[2]:

```
1 #!/usr/bin/perl
2 while ( <> ) {
3    $swedes++ if (/\b[A-Z][a-z]*son\b/);
4 }
5 print "There were $swedes Swedes in this file\n";
```

[2]For simplicity we will not try to match åääöÅÄÖ characters

The dash inside square brackets indicates a range of characters. (But what if there are more than one Swedish name on a line?)

- For the rest of this chapter we will try to avoid using the `$_` slang as much as possible.

### 3.4.4 What has been matched?

- Let´s print all Swedish names in a file:

```
1 #!/usr/bin/perl
2 while ( $line = <> ) {
3    if ( $line =~ /(\b[A-Z][a-z]*son\b)/ ) {
4       print "$1\n";
5    }
6 }
```

After a match, the special `$1` will contain the sequence matched by a regexp part enclosed in parenthesis. If there are more than one parenthesis `$2` will contain the sequence matched by the second parenthesis, and so on. (Still if there are several Swedes on a line, only the first one will be matched.)

### 3.4.5 Match again

- If the same regexp matching expression is repeated several times, and the regexp is followed by the character 'g', the next time it is executed it will try to find the *next* match. This is how you catch *all* the Swedes:

```
1 #!/usr/bin/perl
2 while ( $line = <> ) {
3    while ( $line =~ /(\b[A-Z][a-z]*son\b)/g ) {
4       print "$1\n";
5    }
6 }
```

- Let's try to extract web addresses from a file:

```
1 #!/usr/bin/perl
2 while ( $line = <> ) {
3    while ( $line =~ /\b(www(\.\w+)+)\b/g ) {
4       print "$1\n";
5    }
6 }
```

Here `$2` would match the inner parenthesis, but as it is quantified by a `+`, only the last of the matched sequences are put in `$2`. Hence for a well known address, `print "$1 $2";` would result in `www.thep.lu.se .se`

### 3.4.6 Greedy versus tight matching

- The following code will extract the codons in a DNA sequence between any start and stop codons,

codons.pl

```
#!/usr/bin/perl
$seq ="CGGTTAATGGGAATACGCGGATAAGGCGAAATGCACCACGACGCCATAATTTGAAGGATG";
#            ^^^.............^^^      ^^^...................^^^   ^^^

while ( $seq =~ /ATG(([ATGC]{3})+)(TAG|TAA|TGA)/g ) {
    #print "$1\n";
    push @arr, $1;
}
print "@arr\n";

```

codons.pl

We start with matching the start codon `ATG`. Then the thing we want to match is inside a parenthesis and it is at least one group of three arbitrary nucleotides (`[ATGC]{3}`). Finally we want to end with either of the three stop codons. Now if we look by eye, we can see that there are two such matches in the string: `GGAATACGCGGA` and `CACCACGACGCCATAATT`, but if we run the script it will only find one sequence starting after the first start codon and ending with the last stop codon: `GGAATACGCGGATAAGGCGAAATGCACCACGACGCCATAATT`. This is because the `+` character matches as many as possible. It is greedy. To stop it from being greedy we need to add a `?` sign after it:

```
while ( $seq =~ /ATG(([ATGC]{3})+?)(TAG|TAA|TGA)/g ) { print "$1\n"; }
```

will print both sequences.

### 3.4.7 Match the matched

- Sometimes we want to match something which has already been matched in the same regular expression. Say for instance, that we in the previous example only want to match sequences which has the same codon just after the start codon and just before the stop codon. We cannot use the `$1-$9` special variables as these are not set until after the matching. Instead we can use the special regexp *backreferences* `\1-\9` as follows:

codons2.pl

```
#!/usr/bin/perl
$seq ="CGGTTAATGGGAATACGCGGATAAGGCGAAATGCACCACGACGCCATAATTTGAAGGATG";
#            ^^^.............^^^      ^^^...................^^^   ^^^
```

```
4
5  while ( $seq =~ /ATG(([ATGC]{3})([ATGC]{3})*?\2)(TAG|TAA|TGA)/g ) {
6      print "$1\n";
7  }
8
```

codons2.pl

Where we have a parenthesis around the first single codon after the start codon, and since this is the second parenthesis in the we have added \2 to match it again before the stop codon.

### 3.4.8 Variable interpolation in regular expressions

- Although using $2 in the previous expression does not work in the way we wanted, standard variable interpolation does work as inside double-quoted strings. In this way we can avoid having all regular expressions *hard wired* into the code. Variables can also can be used to make complicated expressions slightly more readable:

codons3.pl

```
1  #!/usr/bin/perl
2  $seq ="CGGTTAATGGGAATACGCGGATAAGGCGAAATGCACCACGACGCCATAATTTGAAGGATG";
3  #            ^^^............^^^      ^^^.................^^^   ^^^
4
5  $cod="([ATGC]{3})";
6  $stop = "(TAG|TAA|TGA)";
7  $start = "ATG";
8  while ( $seq =~ /$start(($cod)$cod*?\2)$stop/g ) {
9     print "$1\n";
10 }
```

codons3.pl

### 3.4.9 Substitutions

- Once we have found a match, we can substitute it with something else. Here is how you change all Swedish looking names to Svensson:

```
1  #!/usr/bin/perl
2  while ( $input = <> ) {
3     $input =~ s/\b[A-Z][a-z]*son\b/Svensson/g;
4     print $input;
5  }
```

Without the g qualifier only the first name on each line would be changed.

- We can use the matched word in the substitution. Here is how we change all Swedish names to Danish ones:

```
1  #!/usr/bin/perl
2  while ( $input = <> ) {
```

```
3     $input =~ s/(\b[A-Z][a-z]*)son\b/$1sen/g;
4     print $input;
5  }
```

- The following will turn any text into a DNA sequence:

```
1  #!/usr/bin/perl
2  while ( $input = <> ) {
3     $input =~ s/[^gatcGATC]//g;
4     print $input;
5  }
```

  The ^ normally matches the beginning of a line, but when given as the first character in a range of characters within brackets, it negates the range of characters. `[^gatcGATC]` matches all characters *except* gatcGATC.

- If you want to substitute characters according to a translation table, there is a special construction in Perl. To change a DNA sequence into its complement:

```
1  #!/usr/bin/perl
2  while ( $input = <> ) {
3     $input =~ tr/gatcGATC/ctagCTAG/;
4     print $input;
5  }
```

  The first character in the left expression is replaced by the first character in the second and so on.

- We can change all lower case letters into upper case:

```
1  #!/usr/bin/perl
2  while ( $input = <> ) {
3     $input =~ tr/[a-z]/[A-Z]/;
4     print $input;
5  }
```

- Here is a very primitive encryption algorithm:

```
1  #!/usr/bin/perl
2  while ( $input = <> ) {
3     $input =~ tr/[a-zA-Z]/[b-zaB-ZA]/;
4     print $input;
5  }
```

### 3.4.10 New features: smart matching

Perl 5.10 introduced a new smart match operator called '∼∼'. The easy way to describe this new operator is however to show some examples. Note, for the following examples it is assumed that the line `use feature ":5.10";` is present in the beginning of your program.

- Before we start let's also introduce the `say` function, which was also introduced in Perl 5.10. It is identical to the `print` function except that it adds a new-line character to the print string. The following two expression will produce the same output:

```
#! /usr/bin/perl -w
use feature ':5.10';

print "Hello World!\n";
say "Hello World!";
```

- Now let's look at some smart matching examples!

```
@arr = qw(Perl is way cool);
$x = 'is';

if ('way' ~~ @arr) {
    say 'way exists in @arr';
}
if ($x ~~ @arr) {
    say "$x exists in \@arr";
}
```

  Here the ∼∼ checks whether a certain item exists in an array. The loop way of doing this would look like

```
foreach (@arr) {
  if ( $x eq $_ ) {
    say "$x exists in \@arr";
    last;
  }
}
```

- We can also do `regexp ~~ array`

```
if (/oo/ ~~ @arr) {
    say 'oo can be found in @arr';
}

if (/c\w{2}l/ ~~ @arr) {
```

```
6       say 'regexp matched at least one element in @arr';
7   }
```

Here we are essentially checking each element in `@arr` against the regexp. It will be true if any element matches.

- We can even do `array ~~ array`

```
1   @arr1 = qw(Perl is way cool);
2   @arr2 = qw(Perl is way cool);
3
4   if ( @arr1 ~~ @arr2 ) {
5       say "identical arrays";
6   }
```

- Let's show a couple of examples involving hashes also.

```
1   %hash = ('A' => 'Ala', 'C' => 'Cys', 'Lys' => 'K');
2
3   if ('C' ~~ %hash) {
4       say "key 'C' exists in %hash";
5   }
6
7   if (/y/ ~~ %hash) {
8       say "key with character 'y' exists in %hash";
9   }
```

This is checking keys in the hash. As with arrays one can also do `hash ~~ hash` which will check if the two hashes have identical set of keys.

In the `perlsyn` man page you can find more details).

### 3.4.11 Follow-up tasks 3-4

1. Execute code snippets in this section and **make sure** you understand them.
2. Make a Perl script which counts the occurrences of the word *the* in a file. It should not count words such as *theory*, but it should count *The*.
3. The Unix program `find` will search through the given directory (and all sub-directories recursively) and write out all filenames (their relative or full path). '`find /`' will list all files on the computer with their full path (there are quite a lot of files – hit *ctrl-c* if you get tired). Make a script to extract all files containing Perl scripts (i.e. with the suffix `.pl`), write out their names (without the full path) and count them.

4. An HTML file contains hyper links on the form
   `<a href="http://www.blah.duh/some/document.html">` or
   `<A HREF="http://www.blah.duh/some/document.html">` where the address in double quotes is a *URL*. Create a script which extracts all such URLs from a document. Try it on the page
   `http://home.thep.lu.se/~mattias/index.html` which you can retrieve with the `wget` command[3].

   You should get out the following links:

```
/~mattias/images/category_holiday_16.png
/~mattias/images/working-16.png
/~mattias/include/style.css
http://www.lu.se/
http://cbbp.thep.lu.se
mailto:&#109;&#97;&#116;&#116;&#105;&#97;&#115;&#64;&#116;&#104;&#101;&#112;.&#108;u.se
publications/
teaching/fytn06
http://www.biology.lu.se/education/courses/advanced-courses/bioinformatics-and-sequence-analysi
teaching/binp13
http://www.createhealth.lth.se/
http://www.lumi.lu.se
mailto:&#109;&#97;&#116;&#116;&#105;&#97;&#115;&#64;&#116;&#104;&#101;&#112;.&#108;u.se
```

   Hint: In a regexp the `"` character is not special. You need to match the initial `href="`, the final `"` and anything in between (which is not a `"`). Don't forget the `g`-qualifier to get several matches on each line.

## 3.5 Filehandles

Filehandles are used to interact with files, console input, socket connections, ... We have already encountered the three predefined handles `STDIN`, `STDOUT` and `STDERR`.

### 3.5.1 User defined filehandles

- To access a file we must create our own filehandle and assign the file to it.

fread.pl

```
1  #!/usr/bin/perl
2
3  $fname = "protein_sequences.fasta";
4
5  open $MYFILE, $fname  or die "Cannot open '$fname': $!";
6
7  while ( $line = <$MYFILE> ) {
8     print "$.: $line";
```

[3] like this: `wget -q -O - http://home.thep.lu.se/~mattias/index.html`

```
9  }
10 close $MYFILE;
```
fread.pl

- This will open the file, assign it to our filehandle `$MYFILE` and the `while` loop will print the content to screen with line numbers.
- It is possible to use the bare word `FILE` as the filehandle, but it can lead to bugs that are difficult to track down. Instead we treat the filehandle as a scalar variable (e.g. `$FILE`). The use of capitals is just a way of indicating that it is a filehandle.
- The `die` statement is needed to check the return code from `open` and inform the user if something went wrong. It is in general a good idea to check return codes, at least when debugging.
- `$!` is a magic variable which contains the last error message given by a built-in Perl function.
- `$.` is a magic variable which always contains the number of times `<>` operator has been called for the file handle for which it was last called.
- The `/` in the file name above is in Unix style, but works seamlessly in other operating systems where other characters are used.
- We can of course write to files as well.

fwrite.pl
```
1  #!/usr/bin/perl
2
3  $in = "protein_sequences.fasta";
4  $out = "script.out";
5
6  open $INFILE, '<', $in or die "Cannot open '$in': $!";
7  open $OUTFILE, '>', $out or die "Cannot open '$out': $!";
8
9  while ( $line = <$INFILE> ) {
10    print $OUTFILE "$.: $line";
11 }
12 close $INFILE;
13 close $OUTFILE;
```
fwrite.pl

- Here we have three arguments to the `open` function. For the `$OUTFILE` the second argument, `'>'`, is used to tell that it is an output file.
- For `$INFILE`, `'<'`, tells the `open` function that we open a filehandle to read from. This is the default behaviour when using only two argument to the `open` function.
- The expression `$line = <$INFILE>` means that the scalar `$line` stores each line of the input file in the while loop.
- It is also easy to append information to a file:

fappend.pl

```
#!/usr/bin/perl

$in = "protein_sequences.fasta";
$out = "script.out";

open $INFILE, '<', $in or die "Cannot open '$in': $!";
open $OUTFILE, '>>', $out or die "Cannot open '$out': $!";

while ( $line = <$INFILE> ) {
   print $OUTFILE "$.: $line";
}
close $INFILE;
close $OUTFILE;
```

fappend.pl

- The only change is `>>` instead of `>`.

### 3.5.2 Pipes

- It is possible to assign the input and output of other programs to filehandles.
- `open $INPIPE, '-|', 'program');` can be used to read the output from another program.
- `open $OUTPIPE, '|-', 'program');` can be used to write to the standard input of another program.
- The following program will use `wget` to open a web page, read from it and extract all links to other web pages. The output is sent to the `sort` program which sorts all lines alphabetically and then pipes it on to the `uniq` program which removes all duplicate lines.

hsort.pl†

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

$cmd = 'wget -q -O - http://home.thep.lu.se/~mattias/index.html';
open $INPIPE, '-|', $cmd;
open $OUTPIPE, '|-', 'sort|uniq';
while ( <$INPIPE> ) {
   while ( /(href|HREF)="([^"]+)"/g ) {
      print $OUTPIPE "$2\n";
   }
}
close $INPIPE;
close $OUTPIPE;
```

hsort.pl†

- We must `close` the output pipe, otherwise that program will continue to wait for input, and our script will wait for the program to end.
- Normal files are closed automatically when the script ends, but is a good habit to always close the filehandle when the reading/writing is done.

### 3.5.3 Follow-up tasks 3-5

1. Execute code snippets in this section and make sure you understand them.

2. Look at the following script and make sure you understand it.

multiply.pl

```
#!/usr/bin/perl

print STDOUT "Multiply number with *3\n";
print STDOUT "Enter a number : ";
chomp ( $input = <STDIN> );

if ( $input =~ /\D/ ) {
   print STDERR "Not a number !!!!\n";
} else {
   print STDOUT 'The answer is ', $input * 3, "\n";
   print STDOUT "The result is also save on the external file 'res.txt'\n";
   open $OUT, '>', 'res.txt' or die "Cannot open file 'res.txt'\n";
   print $OUT 'The answer is ', $input * 3, "\n";
   close $OUT;
}
```

multiply.pl

3. The file `numbers.txt` contains a few numbers (one per line). Write a small program that opens the file using a filehandle and then reads all numbers. The program should also create a new file called `numbers2.txt` where all numbers have been squared.

4. Write a small program that counts the different amino acids in a file containing protein sequences. The program should also compute the frequency of each amino acid (i.e. the fraction compared to the total number of aa). The sequences are found in the FastA file `protein_sequences.fasta`, at the course web page.

## 3.6 Hand-in Exercise 1

In this exercise you will make Perl scripts that make use of:

- scalars, arrays and hashes
- basic input and output
- loops
- simple pattern matching

1. Write a Perl program that defines a hash which associates the three letter abbreviation of each amino acid with its one-letter code. E.g.

```
ALA : A
CYC : C ...
```

Using a `foreach` loop and the `keys` function, display the contents of the hash in the format above. An example output can be found in the file `sample-H1-1.txt`. The file `aminos.txt` contains a list of the abbreviations that should be in your hash. Note: You do not have to read this file from the Perl program, it is just a handy list of all abbreviations!

2. Write a Perl program that reads the file `protseq.txt` from the standard input (`STDIN`) and count the number of occurrences for one selected letter. An example output can be found in the file `sample-H1-2.txt`. Hint: Use a `while` loop for the flow control.

3. Write a Perl program that reads the file `1ECD.fasta` using a user defined filehandle and count the number of occurrences for one selected amino acid letter. The program should ignore the description line. An example output can be found in the file `sample-H1-3.txt`.

4. Write a Perl program that reads the file `1ECD.fasta` using a user defined filehandle and count the number of occurrences for each amino acid. Display result in the form

```
A: 17
D: 9
...
```