Chapter 4

Programming in Perl

4.1 More on built-in functions in Perl

There are many built-in functions in Perl, and even more are available as *modules* (see section 4.4) that can be downloaded from various Internet places. Some built-in functions have already been used in chapter 3.1 and some of these and some others will be described in more detail here.

4.1.1 split and join (+ qw, x operator, "here docs", .=)

```
1  #!/usr/bin/perl
2
3  $sequence = ">|SP:Q62671|RATTUS NORVEGICUS|";
4
5  @parts = split '\|', $sequence;
6  for ( $i = 0; $i < @parts; $i++ ) {
7    print "Substring $i: $parts[$i]\n";
8  }
```

- `split` is used to split a string into an array of substrings. The program above will write out

  ```
  Substring 0: >
  Substring 1: SP:Q62671
  Substring 2: RATTUS NORVEGICUS
  ```

- The first argument of `split` specifies a regular expression, while the second is the string to be split.

- The string is scanned for occurrences of the regexp which are taken to be the boundaries between the sub-strings.

- The parts of the string which are matched with the regexp are not included in the substrings.
- Also empty substrings are extracted. Note, however that trailing empty strings are removed by default.
- Note that the `|` character needs to be escaped in the example above, since it is a special character in a regexp.
- `split` returns an array and since an array can be assigned to a list we can write:

splitfasta.pl†

```
#!/usr/bin/perl

$sequence=">|SP:Q62671|RATTUS NORVEGICUS|";

($marker, $code, $species) = split '\|', $sequence;
($dummy, $acc) = split ':', $code;
print "This FastA sequence comes from the species $species\n";
print "and has accession number $acc.\n";
```

splitfasta.pl†

- It is not uncommon that we want to write out long pieces of text using many `print` statements. This can be a bit cumbersome. An alternative is to use *here docs*. Anywhere a string is expected you can write `<<` directly followed by some tag. Perl will then take the following lines of text and put them together into a single string (including new-line characters) until it finds the same tag again alone on a line. Any variable inside this string will be interpolated as in a normal double-quoted string.
- We can hence rewrite the previous script as follows (note that the `;` cannot be on the same line as the closing tag).

splitfasta2.pl

```
#!/usr/bin/perl

$sequence=">|SP:Q62671|RATTUS NORVEGICUS|";

($marker, $code, $species) = split '\|', $sequence;
($dummy, $acc) = split ':', $code;
print <<ENDTAG
This FastA sequence comes from the species $species
and has accession number $acc.
ENDTAG
;
```

splitfasta2.pl

- Using the pattern of space ' ' for the `split` command is special. It will split on any white space, deleting both leading and trailing white space. It is useful when parsing rows of data. In the next script the `@numbers` array will contain the numbers 1,2,5,6,8. Also, the ' ' pattern will automatically remove any last newline character occurring in the string to split. Split using the ' ' pattern is extremely useful!

splitwhite.pl†

```
#!/usr/bin/perl

$row = "   1 2  5 6    8     ";
@numbers = split ' ', $row;
print "Array numbers contain: @numbers\n";
```

splitwhite.pl†

- `join` is in some sense the inverse of `split`. It takes an array and concatenates the elements into a string.
- The first argument to `join` is a string piece (the *glue*) inserted between the concatenated strings. The string piece can, of course, be an empty string. The second argument is the array to be concatenated.
- If an array is interpolated inside a string literal it is turned into a scalar with `join ' ', @array`.
- `join` can also be called with more than two arguments, in which case the first is the glue, and all the rest are (arrays of) strings to be concatenated:

```
@arr = qw(1 2 3 4 5);
print join('<<', 0, @arr, 6), "\n";
```

- The above snippet will print out `0<<1<<2<<3<<4<<5<<6`. Note here we use `()` around the `join` function since it appears in the `print` function.
- The `qw` keyword is used to create a list of strings in a simplified way. Instead of the normal list `("The", "quick", "brown", "fox")` you can do `qw(The quick brown fox)` and get the same result. You can use any character as delimiters instead of `()`, eg. `qw|The quick brown fox|` (note that the character pairs `()`, `[]`, `{}` and `<>` are special in that we use `qw<The quick brown fox>` etc.)
- Remember that you can also use the `.` (dot) operator to concatenate strings, as well as standard variable interpolation.
- There is also a `.=` operator appends the string on the right to the scalar on the left. Hence the following will print out `Hello World!`

```
$hello = "Hello";
$hello .= " " . "World";
$hello .= "!";
print "$hello\n";
```

- Finally, we have the `x` operator which repeats the string on the left the number of times given on the right. Hence the following will print out "`EchoEchoEchoEcho`".

```
$str = "Echo" x 4;
print "$str\n";
```

4.1.2 push, pop, shift, unshift and splice

- To add elements to an array one can use the `push` and `unshift` functions.
- The first argument of `push` is the array to be increased. All following arguments will be added as elements to the end of the array. If any of these arguments are arrays, all the corresponding elements will be added.
- `unshift` works in the same way except the arguments are added to the beginning of the array. The name of this function will become clearer below.
- The following snippet will print out "`0 1 2 3 4 5 6 7 8`" (note the order in which the arguments to `unshift` are added).

```
1 @arr = (2, 3, 4, 5, 6);
2 push @arr, 7, 8;
3 unshift @arr, 0, 1;
4 print "@arr\n";
```

- Note that accessing elements beyond the end of an array will also add elements, but this is may result in holes, eg. the following will print out "`0 1 2 3 4  6`":

```
1 @arr = (0, 1, 2, 3, 4);
2 $arr[6] = 6;
3 print "@arr\n";
```

- The `pop` function will remove the last element of an array and return it.
- Similarly, `shift` will remove the first element of an array and return it. The elements of the array are *shifted* one step to the left and the zeroth element *falls off* the array. Similarly, the `unshift` function will shift the array to the right, making room for new elements in the beginning.
- The following snippet will produce "`40 (1 2 3)`":

```
1 @arr = (0, 1, 2, 3, 4);
2 print pop(@arr);
3 print shift(@arr);
4 print " (@arr)\n";
```

- `push`, `pop`, `shift` and `unshift` can be thought of special cases of the more general `splice` function for manipulating arrays.

- `splice` takes three or more arguments, where the first is the array to manipulate. The second specifies the index of the first element to be removed and the third how many elements to remove. All the following arguments are inserted instead of the ones which were removed.
- In the following snippet, one element starting from index 3 (the number 7) will be removed and the numbers 3 and 4 will be inserted resulting in the output "`0 1 2 3 4 5 6`":

```
1 @arr = (0, 1, 2, 7, 5, 6);
2 splice @arr, 3, 1, 3, 4;
3 print "@arr\n";
```

- We can convince ourselves that `splice @arr,0,1` is the same as `shift @arr`
- If the second argument is negative it will count elements from the end of the array. Hence `splice @arr,-1,1` is the same as `pop @arr`.
- `splice @arr,@arr,0,$x` is the same as `push @arr,$x` since the array in the second argument is converted to a scalar giving its length. I.e. *no elements are removed from the end of the array and a scalar is added instead.*
- Finally `splice @arr,0,0,$x` is easily shown to be the same as `unshift @arr,$x`.
- The return of the `splice` function depends on the context: In list context, it returns the elements removed from the array. In scalar context, it returns the last element removed, or "undef" if no elements are removed.

4.1.3 The map function

A for/foreach loop is convenient in situations where a fixed number of list elements is to be processed. E.g.

```
1 foreach $item ( @res ) {
2    push @sqr_res, $item * $item;
3 }
```

- All items in the array `@res` are squared and placed in the new array `@sqr_res`.
- An alternative to the loop is to use the built-in `map` function.
- This function is specifically aimed at situations where you want to process a list of values to create a new one. The code for the above example looks like this when using the `map` function:

```
1 @sqr_res = map { $_ * $_ } @res;
```

- The `map` function evaluates the block once for each item in the `@res` array, with `$_` aliased to a different original list element each time. The result of each such block evaluation is put into the resulting array `@sqr_res`.
- Some benefits are obvious, less code and easier to understand. The `map` function is also more efficient than using loop constructions.
- Here are a few more examples:

map1.pl†

```
1 #!/usr/bin/perl
2
3 @mm = qw(jan feb mar apr may jun jul aug sep oct nov dec);
4
5 @uc_mm = map {uc $_} @mm;
6
7 print "@uc_mm\n";
```

map1.pl†

- Here each element in the `@mm` array are made into uppercase using the `uc` function, stored in the `@uc_mm` array and then printed.

map2.pl†

```
1 #!/usr/bin/perl
2
3 @num = (3, 9, -4, 10, -6);
4
5 @num2 = map { $_ >= 0 ? sqrt $_ : ()} @num;
6
7 print "@num2\n";
```

map2.pl†

- In the last example only the positive elements from the `@num` array are transformed using the `sqrt` mathematical function.
- The construction `A ? B : C` may be new and should be read as, "if `A` is true then `B` else `C`.

4.1.4 Sorting

- `sort` will take the list given as argument and return an array where the elements are sorted.
- By default the elements are sorted in lexical order defined by the `cmp` operator.

- This means that the following snippet will output "`0 1 2 23 3 4 5 6`"

```
@arr = (0, 1, 2, 6, 5, 3, 4, 23);
@sorted = sort @arr;
print "@sorted\n";
```

- If another ordering is required, it can be provided by a *block* of code as first argument. In this block the scalars `$a` and `$b` are set to the elements to be compared and the code should result in a negative integer value if `$a` comes before `$b`, positive if `$a` comes after `$b` and zero if they are equal. This is exactly what the `cmp` operator does for strings. The following example will hence produce the same output as the previous snippet:

```
@arr = (0, 1, 2, 6, 5, 3, 4, 23);
@sorted = sort {$a cmp $b} @arr;
print "@sorted\n";
```

- Note that there is no comma between the first and second argument.
- The array given as argument to `sort` is left untouched.
- If we want to sort the elements numerically to get the output "`0 1 2 3 4 5 6 23`", we can do the following:

```
@arr = (0, 1, 2, 6, 5, 3, 4, 23);
@sorted = sort {$a - $b} @arr;
print "@sorted\n";
```

- Instead of the subtraction we could use the `<=>` operator which returns -1, 0, or 1 if the left side is less than, equal to or larger than the right side respectively.
- It is rather common to want to sort the entries in a hash, since these have no defined order intrinsically.
- In the following we assume that we have scanned a number of newspaper articles and collected in a hash the number of times a given name has occurred.

```
%namecount = ( "Gandalf", 1523, "Prins Valiant", 32,
               "Superman",  993, "Mattias Ohlsson", 0 );

foreach $name ( sort keys %namecount ) {
  print "$name: $namecount{$name}\n";
}
```

- The resulting output is

```
Gandalf: 1523
Mattias Ohlsson: 0
Prins Valiant: 32
Superman: 993
```

- To get the items sorted in descending number of occurrences we can instead use a block of code as follows:

```
1 %namecount = ( "Gandalf", 1523, "Prins Valiant", 32,
2                "Superman",  993, "Mattias Ohlsson", 0 );
3
4 foreach $name ( sort { $namecount{$b} <=> $namecount{$a} } keys %namecount ) {
5   print "$name: $namecount{$name}\n";
6 }
```

- Rather than a block of code, the `sort` function can also take a subroutine as argument.

4.1.5 Standard mathematical functions

- Perl also include the most common mathematical functions in its standard library. These all have reasonable straight forward names. Here are some examples.
- `abs` gives the absolute value of a number, `int` gives the integer part of a floating point value, `hex` and `oct` takes strings as argument and interpret them as hexadecimal and octal based integers and returns the corresponding integer.
- `sin`, `cos` and `tan` are the standard trigonometric functions. `atan2` takes an x and y value as arguments and returns the corresponding angle to the x-axis (the inverse of sin and cos can be trivially implemented in terms of this).
- `exp` and `log` are the standard exponential and natural logarithm functions.
- `rand` returns a pseudo-random floating point number between zero and the (positive) argument.

4.1.6 Other built-in functions

- As mentioned in the beginning, there are a large number of built-in functions available in Perl. Below is a small list of functions that may be of interest for you. Some will be dealt with later and some have been mentioned in a previous chapter.

  `index substr lc lcfirst uc reverse sprintf exists keys`
  `grep values print printf open each close -X`

- All of these functions are described in detail in the manual pages for Perl. Under Linux you can do `man perlfunc` to find information about all functions and `perldoc -f 'function'` to find information for a specific one.

4.1.7 Follow-up tasks 4-1

1. Execute the code snippets in this section and make sure you understand them. Play with them and try out whatever comes to mind. In particular you may want to change the first argument to `split` in the first example from `\|` to `|`. Also it is instructive to look at all different ways to use `splice`.

2. Use the Perl man pages (`man perlfunc`) to find out how to use the functions `index`, `substr` and `reverse`. Write small pieces of code which use these functions and make sure you understand them.

3. In `man perlfunc` you will also find a set of mathematical functions, such as `cos`, `log`, `sqrt` etc. Write small scripts where some of these are used.

4. Write a program which reads through the file `protein_sequences.fasta` and counts the number of occurrences for each amino acid[1]. (Remember to ignore the header lines.) Display the result sorted alphabetically according to the amino acid code letter, numerically in increasing and decreasing number of occurrences.

4.2 User-defined functions

So far we have written our scripts as a sequence of statements, read from the beginning to end. Although some repetitive tasks have been put in loops, such scripts sometimes becomes too long to be readable. For this reason it is good to subdivide (cf. section 2.10) the task to be performed in the script into *functions*. The main part of the script can then be made fairly compact, just calling a handful of functions which hide the nitty-gritty details. If someone is interested in these details he/she can look at the function definitions to see what happens.

- **Important:** As we increase the complexity of our Perl programs we will now add the `-w` flag to the Perl command (first row of every Perl script). This flag will tell Perl to warn you when it sees something suspicious going on in your program.

  From the "Learning Perl" book *"Of course, warnings are generally meant for programmers, not for end-users. If the warning won't be seen by a programmer, it probably won't do any good. And warnings won't change the behavior of your program, except that now it will emit gripes once in a while. If you get a warning message you don't understand, look for its explanation in the perldiag manpage."*

[1] See also follow-up exercise 3-5

4.2.1 Defining and calling simple functions

```
1  #!/usr/bin/perl -w
2
3  sub writeit {
4      print "\$it = $it\n";
5  }
6
7  $it = 11;
8  &writeit;
9  $it = $it/2;
10 writeit();
```

- The script above defines a function called `writeit`.
- Functions are sometimes referred to as *subroutines*, hence the `sub` which tells Perl that what follows is a definition of a function.
- The declaration starts with a name followed by a *function body* in curly brackets.
- Every time the function is called, the statements in the function body are executed.
- There are a few ways of calling a function, writing the function name prepended with an `&` sign, as in the first example above. The preferred way however, is to write the name followed by a (possibly empty) list of arguments enclosed in parenthesis, as in the second example.
- The `writeit` function will simply write out the content of the `$it` scalar, whatever its value happens to be when the function is called. The output of the previous script will be

  ```
  $it = 11
  $it = 5.5
  ```

- The function above was defined in the beginning of the script. We could also define it in the end of the program and this is advisable to improve the readability of the program.

subend.pl†

```
1  #!/usr/bin/perl -w
2
3  $it = 11;
4  writeit();
5  $it = $it/2;
6  writeit();
7
8  sub writeit {
9     print "\$it = $it\n";
10 }
```

subend.pl†

- From now on we will only call functions using the parenthesis notation. Not only is this the most clear way when we later deal with arguments, it is also the way most other languages deals with function calls.

4.2.2 Functions with arguments

- In the previous example, the function printed out the value of a particular variable. This is not very versatile. To allow the function to write out any variable we can do as follows:

subpar.pl†

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

$it = 11;
writevar($it);
$a = $it/2;
writevar($a);

sub writevar {
   $var = $_[0];
   print "The value was: $var\n";
}
```

subpar.pl†

- When the function is called with a given argument, that argument will be available in the function body from the *magic* `@_` array. The first (in this case the only) argument will be in the first element. If more arguments are given they will be placed in the following elements of `@_`.

subsum.pl†

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

$x = 7;
$y = 2;
sumvars($x, $y);
$x = -3;
sumvars($x, $y);
sumvars($x, 8);

sub sumvars {
   $a1 = $_[0];
   $a2 = $_[1];
   print "The sum is: ", $a1 + $a2, "\n";
}
```

subsum.pl†

- This function takes two arguments, available in `$_[0]` and `$_[1]`, respectively, in the function body.

- As we may have several arguments it is more convenient to "unpack" the argument array in the following way: `($a1, $a2) = @_;`. This will be the preferred way.

subsums.pl†

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

@arr = ( 0, 1, 2, 3, 4 );
sumvars(@arr);

sumvars(1, 2, 3, 4);

sub sumvars {
   $sum = 0;
   foreach $arg ( @_ ) {
      $sum += $arg;
   }
   print "The sum is: $sum\n";
}
```

subsums.pl†

- In this example the function `sumvars` is called with an array as the argument.
- The elements of the list or array are made available in the `@_` variable in the function body.

4.2.3 Functions returning a value

- All functions returns a value. If none is explicitly specified, the value of the last statement executed in the function body will be the value.
- It is possible to specify what should be returned with a `return` statement.
- When a `return` statement is executed in a function body, the specified value is returned immediately and statements which may come later in the function are ignored.

sumfn.pl†

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

@arr = ( 0, 1, 2, 3, 4 );
$x = sumvars(@arr);
print "The sum is: $x\n";
print "Half the sum is: ", sumvars(1, 2, 3, 4)/2, "\n";

sub sumvars {
   $sum = 0;
   foreach $arg ( @_ ) {
      $sum += $arg;
   }
   return $sum;
}
```

sumfn.pl†

- The value returned from a function call can be assigned to a variable, or used directly in an expression. Note that function calls are not interpolated in double-quoted strings.

- A function may return more than one value. In general it can return a list of values:

sumnfn.pl†

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

($x, $y) = sumvars(0, 1, 2, 3, 4);
print "The sum of the $y elements is: $x\n";

sub sumvars {
   $sum = 0;
   $n = scalar @_;
   foreach $arg ( @_ ) {
      $sum += $arg;
   }
   return ($sum, $n);
}
```

sumnfn.pl†

- A function can be thought of a a mini-program. In a normal Perl program data is typically taken from the standard input and the results are printed to the standard output. For a function the input data is given as arguments and are available in the `@_` array, while the output is delivered as a return value.
- We know that the statement `$n = @_` assigns `$n` to be the length of the array `@_`. To be more explicit one can write `$n = scalar @_`, which forces a scalar context of the `@_` array, meaning the length of it.

4.2.4 Namespaces and variable scopes

- When writing long scripts it is not uncommon that one accidentally uses the same name for two different variables.
- Especially when using many functions it may even be desirable to use the same name for different variables.
- So far we have used only *globally* declared variables, which means that accidents may happen:

sumnfn2.pl†

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

($sum, $n) = sumvars(0, 1, 2, 3, 4);
($sum2, $n2) = sumvars(5, 6, 7);
print "The first list had $n elements and the sum was: $sum\n";
print "The second list had $n2 elements and the sum was: $sum2\n";

sub sumvars {
   $sum = 0;
   $n = scalar @_;
   foreach $arg ( @_ ) {
```

```
12        $sum += $arg;
13     }
14     return ($sum, $n);
15  }
16
```

sumnfn2.pl†

- In this example both lists will be reported to have three elements and the sum 18. The problem is that in the second call to `sumvars`, the values assigned to `$sum` and `$n` will be overwritten since they are globally declared.
- To avoid this we can make the `$sum` and `$n` used in `sumvars` *local* to the function, so that inside the function body these names refers to different variables than the ones in the main body of the script. For this we use the keyword `my`:

sumnfn3.pl†

```
1  #!/usr/bin/perl -w
2
3  ($sum, $n) = sumvars(0, 1, 2, 3, 4);
4  ($sum2, $n2) = sumvars(5, 6, 7);
5  print "The first list had $n elements and the sum was: $sum\n";
6  print "The second list had $n2 elements and the sum was: $sum2\n";
7
8  sub sumvars {
9     my $sum = 0;
10    my $n = @_;
11    foreach $arg ( @_ ) {
12       $sum += $arg;
13    }
14    return ($sum, $n);
15 }
16
```

sumnfn3.pl†

- Every time `sumvars` is executed two new variables named `$sum` and `$n` will be created which have nothing to do with the variables in the main body.
- The `my` keyword is only given before the first use of the variable.
- A variable declared with `my` cannot be accessed outside of the function where it is declared. Such a variable is said to be in the functions local *scope* or *namespace*.
- If a variable declared with `my` in a function is given a value in one call, that value will **not** be remembered the next time the function is called.
- A variable declared with `my` inside any block of statements (enclosed in curly brackets) is local to that block and cannot be accessed outside.
- A variable declared with `my` inside the a loop and the value is set in one iteration, that value will not be remembered in the next iteration.

- If there are blocks of statements inside another block of statements, a `my`-variable declared in the outer block will be accessible in the inner block, but not vice versa.

```
$i = "hello";
for ( my $i = 0; $i < 10;  ++$i ) {
   print $i;
}
print "$i\n";
```

- A `my`-variable declared in a `for`-loop initialization is local to the loop although it is declared outside the actual `for`-block. The snippet above will print out "`0123456789hello`".
- To help the programmer to think about the scope of the variables, the statement "`use strict;`" can be issued in the beginning of the script. In that case Perl will consider it an error if a variable is introduced without qualifying it with `my`.
- **IMPORTANT**: For the rest of this course we will always use "`use strict;`" in our scripts, and all variables will be declared with `my`.
- A `my`-declared variable in the global scope (ie. in the main script body outside any enclosing blocks) will be available inside the functions defined later. Hence the following script will also print out "`0123456789hello`".

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

my $i = "hello";
for ( my $i = 0; $i < 10;  ++$i ) {
  print $i;
}
printi();

sub printi {
  print "$i\n";
}
```

4.2.5 References as function arguments

- In general all arguments given when calling a function are available in the function body in `@_`.
- If you modify one of the elements in `@_` in a function, the corresponding scalar variable given as argument will be modified as well. Hence the following will write out "`HELLO WORLD!`".

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

my $x = "Hello";
my $y = "World";
toupper($x, $y);
print "$x $y!\n";

sub toupper {
  for ( my $i = 0; $i < @_; $i++ ) {
    $_[$i] =~ tr/[a-z]/[A-Z]/;
  }
}
```

- If an array is passed as an argument, its elements will be available in the function body in `@_`. Again if the function modifies an element in `@_`, the corresponding element in the array passed as argument is changed as well, so using the following snippet in the script above would also write out "`HELLO WORLD!`".

```
my @arr = qw(Hello World);
toupper(@arr);
print "$arr[0] $arr[1]!\n";
```

- The actual array given as argument is, however, not changed in the function, only its elements.

popchop.pl†

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

my @arr = (0, 1, 2, 3, 4);
popchop(@arr);
print "@arr\n";

sub popchop {
   pop @_;
   shift @_;
}
```

popchop.pl†

- Here the `popchop` function will take away the first and last element of the **local** `@_` array, but the array given as argument is left untouched, and the output will be `0 1 2 3 4`.

- To manipulate an array as such in a function we must pass a *reference* to it as argument, rather than its elements.

- A reference can be thought of as the address of the place in the memory where a variable is stored.
- The following script will output `1 2 3`

popchop2.pl

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

my @arr = (0, 1, 2, 3, 4);
popchop(\@arr);
print "@arr\n";

sub popchop {
  my ($ref) = @_;
  pop @{$ref};
  shift @{$ref};
}
```

popchop2.pl

- A reference is a scalar!
- The `$ref` scalar is assigned the reference given as argument, and to access the array the argument refers to we need to use `@$ref` or to enhance the readability we can use `@{$ref}`.
- Similarly, an element of the array can be accessed by eg. `${$ref}[1]`.
- When calling the function we need to specify with the '\' operator that we want to pass a reference to the array as argument, rather than the elements of the array.
- **We will talk much more about references in section 5.3**.

4.2.6 Follow-up tasks 4-2

1. As usual, go through all examples in this section and make sure you understand them. Play around with them, eg. try passing different kinds of arguments to the functions used. What happens if you give a string literal to the `toupper` function?
2. Add the `-w` flag and `use strict;` to the ones where it was not already used and make appropriate changes to make them run without errors and warnings.
3. Write your own `join` function which works just like the built-in one. Write a small script to test it.
4. Write a function which takes an array as argument and removes all empty or undefined elements and replaces elements which are not numbers with a number giving the length of the string.
   Hints:
   - You need to pass the array as a reference.

- Use `splice` to remove an element in the middle of the array.
- When looping over an array make sure nothing unexpected happens when removing an element.
- Use the built-in function `defined` to see if an element is actually defined. (see `perldoc -f defined`)
- '`\D`' in a regexp matches any non-digit character.

Assuming the function is called `purgecount`, the following code snippet should output "`5|5|2|0|4|42`".

```
1 my @arr = ("hello", 5, "oo", 0, "", "zero" );
2 $arr[10] = 42;
3 purgecount(\@arr);
4 print join('|', @arr), "\n";
```

4.3 Accessing things outside a script

We have already seen how to access files and *pipes* inside a script. Here we will look a bit more on how to run programs from the script and how to access *environment variables* and *command-line arguments*.

4.3.1 Command-line arguments

- We have already seen the `<>` operator (with no file handle specified) will read from the files specified as command-line arguments.
- The arguments can also be accessed as strings through the predefined array `@ARGV`.

```
1 #!/usr/bin/perl -w
2 use strict;
3
4 for ( my $i = 0; $i < @ARGV; $i++ ) {
5    print "arg $i: \"$ARGV[$i]\"\n";
6 }
```

- Running the above script with the arguments "`Hello World`" will print out

  ```
  arg 0: "Hello"
  arg 1: "World"
  ```

- Note that accessing the arguments directly like this may be messy if `<>` is also used. Every time `<>` needs to open a new file it `shifts` `@ARGV` and uses the shifted element as the file name.

- The following script will scan the filename given as first argument and count the number of times the word given as second argument occurs in the file. If the script is called `countwords.pl` executing it with `./countwords.pl countwords.pl filename` will report four occurrences.

countwords.pl

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

my $filename = $ARGV[0];
my $word = $ARGV[1];
open my $FILE, '<', $filename or die "No such file: $filename";
my $count = 0;
while ( my $line = <$FILE> ) {
   while ( $line =~ /\b$word\b/g ){
      $count++;
   }
}
close $FILE;
print "The word $word occurred $count times in $filename\n";
```

countwords.pl

- A common problem when providing a Perl program which accept command-line arguments is to check that the user supplies the correct number and type of arguments. Some arguments may eg. be optional, some arguments may need to be decomposed, etc. (See eg. `man perlrun` to check out which arguments can be given to the Perl interpreter itself).

- In the rest of this section we will assume we have a subroutine `sprotseq` which takes two arguments: the filehandle of a SwissProt file with proteins and an accession number. This function will scan the file and find the corresponding protein and return its sequence.

- Here is a script using this function, giving the user some freedom when specifying the command-line arguments.

scansprot.pl

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

my $filename = "Swissprot.dat";
my @accnr;
my $countonly = 0;

# Loop until the @ARGV array is empty
while ( @ARGV ) {
   my $arg = shift @ARGV;
   if ( $arg eq "-f" ) {
      $filename = shift @ARGV;
   }
   elsif ( $arg eq "-c" ) {
      $countonly = 1;
   }
```

```
17     else {
18        push @accnr, $arg;
19     }
20  }
21
22  foreach my $acc ( @accnr ) {
23     open my $FH, "<", $filename or die "Cannot open $filename";
24     my $seq = sprotseq($FH, $acc);
25     my $cnt = 0;
26     while ($seq =~ /\w/g) {$cnt++;}
27     if ( $countonly ) {
28        print "Number of amino acids in $acc is ", $cnt, "\n";
29     } else {
30        print "The sequence of $acc is\n$seq\n";
31     }
32     close $FH;
33  }
34
35  sub sprotseq {
36     my ($filehandle, $accnr) = @_;
37     #
38     # Code
39     #
40  }
```

scansprot.pl

- Note that also filehandles can be used as arguments in functions.
- In this script the command-line arguments are scanned by `shift`ing them one by one. Each argument is by default treated as an accession number which will be searched for in the default file `Swissprot.dat`
- If the user supplies a "`-f`" argument, the following argument will be treated as a filename to use instead of the default file (note the extra `shift`ing of `@ARGV` in this case.
- In addition if the user supplies a "`-c`" the actual sequences are not written out, but only the length of the sequences.

4.3.2 Environment variables

- In Linux (but also in other OS's) there are some information available in *environment variables*. As an example, if you give a command in a terminal window, the shell will try to find the corresponding program file by looking in the directories specified by the `PATH` environment variables. (Do "`echo $PATH`" to see what is in that variable.)
- You can issue the command `printenv` to see all the environment variables in your shell and what they are set to.
- You can introduce new environment variables with the `export` command like this: `export MYNAME="Mattias Ohlsson"`. This environment variable is then accessible, not only in the shell by `$MYNAME`, but also inside all programs started from that shell.

- Permanent environment variables can be set in the `.bashrc` file in your home folder. (There are also other stuff in that file.)
- Inside a Perl script, the environment variables can be accessed with the pre-defined hash `%ENV`, where the `keys` are the variable names and the `values` are the corresponding values. The following snippet will write out your home directory:

```
1 my $home = $ENV{'HOME'};
2 print "$home\n";
```

4.3.3 Running programs from within a script

- Pipes have been discussed previously. Here we will revisit these and introduce a few more ways of executing other programs from within a Perl script.
- If the second argument in an `open` statement is `-|` the third argument should be a program and the output of the program is available in the associated file handle.
- If the second argument in an `open` statement is `|-` the third argument should be a program and the input to this program is coming from whatever is printed on the filehandle.
- The full SwissProt file is so large it is usually stored in compressed form. The standard way of compressing files is with `gzip` (see `man gzip`). To avoid compressing and uncompressing a file each time it is needed one can use the program `zcat` which takes the files given as arguments and uncompresses their content and sends the result to the standard output.
- `gzip`-compressed files are given the suffix `.gz` and we can change our script to check if the filename ends in `.gz` and in that case read from a pipe from `zcat` rather than from the file itself.

scansprot2.pl

```
1  #!/usr/bin/perl -w
2  use strict;
3
4  my $filename = "Swissprot.dat";
5  my @accnr;
6  my $countonly = 0;
7
8  # Loop until the @ARGV array is empty
9  while ( @ARGV ) {
10    my $arg = shift @ARGV;
11    if ( $arg eq "-f" ) {
12       $filename = shift @ARGV;
13    }
14    elsif ( $arg eq "-c" ) {
15       $countonly = 1;
16    }
```

```
17    else {
18       push @accnr, $arg;
19    }
20 }
21
22 my $mode = '<';
23 if ( $filename =~ /\.gz$/ ) {
24    $filename = "zcat $filename";
25    $mode = '-|';
26 }
27
28 foreach my $acc ( @accnr ) {
29    open my $SPROTFILE, $mode, $filename or die "Cannot open $filename";
30    my $seq = sprotseq($SPROTFILE, $acc);
31    my $cnt = 0;
32    while ($seq =~ /\w/g) {$cnt++;}
33    if ( $countonly ) {
34       print "Number of amino acids in $acc is ", $cnt, "\n";
35    } else {
36       print "The sequence of $acc is\n$seq\n";
37    }
38    close $SPROTFILE;
39 }
```

scansprot2.pl

- In the above script the scalar `$mode` is either `<` or `-|` depending on the suffix of the Swissprot file.

- An alternative to reading from a pipe is to use the *back-ticks* operator.

- Anything between two ‘ characters are interpolated as a double-quoted string and the corresponding command is executed in a sub-shell. All output from the execution is collected and returned as a single string, including newline characters and all, or as an array where each line in the file corresponds to one element in the array.

```
1 my $filename = "/usr/local/lib/swissprot/sprot-full.dat.gz";
2 my $fullfile = ‘zcat $filename‘; # As a scalar
3 my @fullfile = ‘zcat $filename‘; # As an array, lines are elements in the array
```

- This snippet would put the whole of the (compressed) system-wide SwissProt file in the `$fullfile` scalar as a single character string. (Don't do this – you are likely to exhaust the RAM of your computer which would possibly crash.)

- if you want to run a program but are not interested in its in or output you can use the `system` function. This function takes a string as argument and the corresponding command is executed in a sub-shell. The input to the command read from `STDIN` of the script and the standard output and error are sent directly to the the `STDOUT` and `STDERR` handles respectively.

- `system` returns the *value* of the command. The value of a program executed from a shell is not often used explicitly, but all programs returns an integer. By convention a program will return zero if except if something went wrong – in that case a non-zero error code is returned.

4.3.4 Follow-up tasks 4-3

1. As usual go through the examples in the text and play around with them. In the examples scanning a SwissProt file you can use the working implementation of `sprotseq` below (also available at the course homepage).

sprotseq.pl

```
sub sprotseq {
   my ($filehandle, $accnr) = @_;

   while ( my $line = <$filehandle> ) {
      # First we are only interested in the AC line which contains the
      # given accession number.
      next unless ( $line =~ /^AC.*\b$accnr;/ );

      # If we end up here we have found the correct protein entry.  We
      # now scan until we find the SQ line which preceeds the actual
      # sequence
      while ( my $line = <$filehandle> ) {
         if ( $line =~ /^SQ/ ) {

            # The following lines contains the sequence.
            # When we hit the // line the sequence is complete.
            my $seq = "";
            while ( my $line = <$filehandle> ) {
               last if ( $line =~ /^\/\// );
               $seq .= $line;
            }
            # Note that we can return already here
            return $seq;
         }
      }
   }
}
```

sprotseq.pl

You can use the full SwissProt file available in

`/usr/local/lib/SwissProt/sprot-full.dat` (**large file, 2.6 Gb**)
`/usr/local/lib/SwissProt/sprot-full.dat.gz` (479 Mb)

or a small subset of it in

`/usr/local/lib/SwissProt/sprot-subset.dat`
`/usr/local/lib/SwissProt/sprot-subset.dat.gz`

If you haven't seen a SwissProt file before, look through it and you will find that each entry starts with an `ID` line and ends with a `//` line. Inside the entry the accession code(s) are given in the `AC` line, and all lines after the `SQ` line until the end of the entry contains the amino acid sequence.

2. Change the `sprotseq` function so that it returns the sequence without spaces and newline characters. You can use `scansprot2.pl` as a template for this and the exercises below.

3. If you specify an accession number that does not exist in the Swissprot file the program still prints out the this number but no sequence. Modify the script so that it will print out a message saying that the accession number was not found.

4. To scan through the full SwissProt file takes some time. Rather than going through the file once for each requested accession number, it may be better to give all accession numbers as argument to the function and search for all of them and returning a hash where the found sequences are keyed with the corresponding accession number. Change the function and the rest of the script accordingly. Here are some hints:

   - The first argument to the `sprotseq` is the filehandle and the second should be an array of all the accession numbers.
   - Join together the accession codes into a regexp so that it would look something like `^AC.*\b(acc1|acc2|acc3);` and save the `AC` line in a variable.
   - After you have read in a sequence check which accession number was actually found in the `AC` line and store it in a hash.
   - Note that you should now continue scanning the file after you have found a sequence and only in the end you should return the hash.

4.4 Modules

The `sprotseq` function in the previous section is an example of a function which could be used in many programs and it is, of course, rather simple to cut-and-paste it into every script where it is needed. An alternative is to put the subroutine in a `module` which can be imported into a script, making it shorter and more readable. We will start with a simple example and then go on to look at one of the standard modules provided with the Perl program.

Perl comes with hundreds of more or less useful modules, and thousands more are provided on the net by Perl programmers around the world. We will talk more about modules in chapter 6.

4.4.1 A simple user-provided module

- Consider the function in the following script which returns the sum and average of its arguments.

sum.pl†

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

my ($sum, $av) = sumarr(0, 1, 2, 3, 4, 5);
print "Sum = $sum, Average = $av\n";

sub sumarr {
   my $sum = 0;
   foreach my $arg ( @_ ) {
      $sum += $arg;
   }
   return ($sum, $sum/scalar @_);
}
```

sum.pl†

- We can put the function by itself in a file called eg. `sumfn.pm` like this:

sumfn.pm

```
sub sumarr {
   my $sum = 0;
   foreach my $arg ( @_ ) {
      $sum += $arg;
   }
   return ($sum, $sum/scalar @_);
}

1;
```

sumfn.pm

- Perl will treat this file as a module of functions which can be used inside other scripts.
- Note the `1;` in the end of the file. This is because Perl requires the module to have a value (which is not *false*).
- Note also that the filename conventionally should have the `.pm` suffix (rather than `.pl`).
- We can now use the function in a script:

sum2.pl†

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

my ($sum, $av) = sumarr(0, 1, 2, 3, 4, 5);
print "Sum = $sum, Average = $av\n";
```

sum2.pl†

- The `use sumfn;` instructs Perl to look for a module file called `sumfn.pm` and make available the functions defined there.

- Perl will look for the file in some standard directories (typically under `/usr/lib/perl5`), but it will also look in the current working directory.
- If your module is placed somewhere else you can specify additional directories where you want Perl to look by eg. `use lib "/home/users/mattias/mymodules"`.

4.4.2 Using the `Getopt::Std` module

- The standard `Getopt::Std` module contains a few functions for extracting simple command-line options. To use these functions in a script you need to do "`use Getopt::Std;`"
- Since this is a standard module, Perl will know where to find it, and you do not need to specify a search directory.
- Just as all other standard modules in Perl, there is documentation available by doing `perldoc Getopt::Std`. The resulting man page will look something like this:

```
NAME
       getopt, getopts - Process single-character switches with switch clustering

SYNOPSIS
           use Getopt::Std;

           getopt('oDI');    # -o, -D & -I take arg.  Sets $opt_* as a side effect.
           getopt('oDI', \%opts);    # -o, -D & -I take arg.  Values in %opts
           getopts('oif:');  # -o & -i are boolean flags, -f takes an argument
                             # Sets $opt_* as a side effect.
           getopts('oif:', \%opts);  # options as above. Values in %opts

DESCRIPTION
       The getopt() function processes single-character switches with switch
       clustering. Pass one argument which is a string containing all switches
...
```

- In the following we will use the `getopts` function.
- With the argument `'oif:'`, `getopts` will look at the first element in `@ARGV` and if it is `-o`, `-i` or `-f` it will `shift @ARGV`. If `-f` was found it will put the next argument in the hash provided in the function call with the key `'f'` and `shift @ARGV` again. If the argument found was `-o` or `-i` it will put `1` in the hash with the key `'o'` or `'i'` respectively.
- This is very similar to the way we scanned the command-lines *by hand* in the previous section. The only difference is that `getopts` will stop scanning the command line as soon as it finds an argument which is not one of the specified options.
- We can now change the beginning of our SwissProt script like this:

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

my $filename = "/usr/local/lib/swissprot/sprot-subset.dat";
my $countonly = 0;
my %opts;

getopts('cf:', \%opts);
$filename = $opts{'f'} if defined($opts{'f'});
$countonly = 1 if defined($opts{'c'});

my @accnr = @ARGV;

# The rest of the script as before ...
```

- The `defined` function checks if a key is present in a hash.
- `getopts('cf:',\%opts)` uses the `\` operator to pass a reference to the `%opts` hash to the function. In this way the function can introduce new entries in the hash.
- If the reference to the hash is not given to `getopts`, the corresponding values are put in semi-magic variables called `$opt_f` and `$opt_c` which needs to be declared in a special way:

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

my $filename = "/usr/local/lib/swissprot/sprot-subset.dat";
my $countonly = 0;
our($opt_f, $opt_c);
getopts('f:c');

$filename = $opt_f if defined($opt_f);
$countonly = $opt_c if defined($opt_c);

my @accnr = @ARGV;

# The rest of the script as before ...
```

- Note `our($opt_f, $opt_c);` which is needed to declare package variables.

4.4.3 Follow-up tasks 4-4

1. Go through the examples. What happens if you change the order of the command-line arguments when using the `Getopt::Std` module? What happens if you remove the `1;` in the `sumfn.pm` file?

2. Put the `sprotseq` function in a module and change the scripts using it accordingly. Try putting the module file in different directories and make sure you can access it in your script.

3. The module `Data::Dumper` is handy for printing any kind of variable. Create a small hash, array and a scalar and print them using this module (e.g. `print Dumper(%myhash)`). Hint: Use perldoc to read the man page for this module (`perldoc Data::Dumper`).

4.5 Rules of thumb and recommendations

So far we have mainly been writing small scripts and programming style has not really been an issue. Since Perl was originally designed to do quick-and-dirty hacks, it is easy to write sloppy and unstructured code - Perl will not complain. In the following chapters we will, however, write longer programs and in the future you may be writing programs and modules to be used by others. It is then important to have well structured and well documented code and to impose a fair amount of discipline when writing it. For that purpose we will here give a set of useful rules and recommendations.

4.5.1 Before starting coding

- Always take plenty of time to think through the problem before starting writing code.
- Use stepwise refinement (see section 2.10) to subdivide the problem into parts.
- Each step in the refinement should typically correspond to a function.
- If any of these functions are general enough, try to put them in a module of a black-box character. In this way you can test and debug the individual parts of your program as it develops rather than writing the whole lot before starting testing and debugging.
- If any of the subtasks you want to do seems to be of a general nature, chances are that someone else has implemented this functionality before. Look in the Perl documentation, or ask your local Perl guru, or look on the net (eg. `http://www.cpan.org`) to see if there are any ready-made modules available. This may save you a lot of work.

4.5.2 Disciplined programming

- Always use the `-w` flag to the Perl interpreter. Make sure that you program never issues any warnings of any kind.

- Start by writing a comment block describing what the script does and how it is done.
- Always `use strict;`
- Avoid using literals in your program – especially constant numeric literals. Such *constants* have a tendency to change, and it is then difficult to later find all the places in the code where it is used. Rather you should put in a variable in the beginning of the program and use that instead.
- A program must be properly indented. Which particular style of indentation chosen is less important, but the indentation must be consistently applied everywhere. When writing scripts in emacs, going through the lines from top to bottom, hitting the tab key on each line will give you a reasonable and consistent indentation.
- Comment your code properly. Don't overdo it, but rather too many than too few comments.

4.5.3 User-friendly programming

- Always test your program extensively. Check what happens if you give strange arguments to functions or stupid command-line arguments. Make sure your program behaves nicely under all conditions. If a program receives strange parameters, write out a helpful message.
- Avoid using external shell commands in your code. If your code is used on another platform (eg. Windows instead of Linux) it may not work anymore. Normally there are Perl modules and functions which do what you want in a platform-independent way. (There are eg. modules available for reading and writing gzip-compressed files) In addition, such modules tend to be faster than calling shell commands.

4.6 Hand-in Exercise 2

You must hand in one of the following exercises.

1. *Convert SwissProt entries to a FastA formatted file.*

The SwissProt database is a highly structured file, where every protein entry starts with a `ID` and ends with a `//`. The SwissProt database sample you should convert is available at the course web page (elegans.swissprot). All, but sequence lines, starts with a two letter tag. All the tags are described at the SwissProt home page[2] but the ones needed here are `AC`, `DE`, `OS`, `RX` and `SQ`.

The FastA format is a one line header starting with a '>' character, followed by the sequence information on the lines following the header line. In this exercise you should compose the header line according to:

[2] http://web.expasy.org/docs/userman.html

`>sp|accession|name OS=organism`
where

- `accession` is the accession number (use the first if more than one)
- `name` is the information found in the `DE` tag. Use the information from either 'RecName', 'AltName' or 'SubName' (in that order).
- `organism` is the information found in the `OS` tag.

Your program should read the file to convert from the command line and the new file should have the extension ".fasta". Your program should also print information to the terminal for each entry that is converted. This information should be the accession number, the PubMed id and the DOI url. This information can be found in the `RX` tag. If more than one `RX` entry use the first one.

Summary: The task is to convert `elegans.swissprot` to FastA format. The header line of the FastA file should be composed from the `AC`, `DE` and the `OS` information. Your output file and the information printed on the terminal should be "close" to the sample output files `sample-elegans.fasta` and `sample-terminal.txt`

2. *Nucleotide sequence to amino sequence converter, all reading frames.*

The task is to convert the nucleotide sequences, found in `ecoli.fasta`, to amino acid sequences. There is a nucleotide triplet to amino acid conversion table available through the course web page as *translation.txt*. The table does not contain the stop codons. Add the missing triplets to your table and let them code for a non-amino acid character (e.g. a '-' (dash) character). Note: You should produce protein sequences for all 6 open reading frames.

Summary: The task is to convert the nucleotide sequences in `ecoli.fasta` to protein sequences. Your result should look **exactly** like the sample output file `sample-ecoli.fasta`. Hint1: You might be helped by studying the `substr` function for sub-string extraction from strings. Hint2: When you consider the reversed sequence don't forget to map (`A->T, C->G, G->C, T->A`).