Faculty of Computer Science, Dalhousie University *27-Sep-2024*

CSCI 4152/6509 — Natural Language Processing

Lab 3: Perl Tutorial 3

Lab Instructor: Sigma Jahan and Tymon Wranik-Lohrenz
Location: Mona Campbell 1108 (10am)/Goldberg CS 134 (4pm)
Time: Friday, 10:04–11:25 and 16:05–17:25
Notes authors: Vlado Keselj and Magdalena Jankowska

Perl Tutorial 3

Lab Overview

- We will finish Perl Tutorial with this Perl Tutorial 3
- In this lab you will learn more about
  - arrays
  - hashes
  - references
  - modules

Files to be submitted:

1. `lab3-array-examples.pl`
2. `lab3-test-hash.pl`
3. `lab3-letter_counter_blanks.pl`
4. `lab3-out_letters.txt`
5. `lab3-word_counter.pl`
6. `lab3-out_word_counter.txt`
7. `lab3-word_counter2.pl`
8. `lab3-ngram-output.txt.gz`

Step 1. Logging in to server timberlea

As the first step, as in previous labs, you need to login to the server `timberlea`, create a lab directory, and enter this directory:

1-a) Login to the server `timberlea`

First, login via an ssh connection to the server timberlea; e.g., using the program PuTTY from Windows, or `ssh` from Mac or Linux.

1-b) Change directory to `csci4152` or `csci6509`

Change your directory to `csci4152` or `csci6509`, whichever is your registered course. This directory should have been already created in your previous lab.

Create the directory `lab3` and change your current directory to be that directory:

1-c) `mkdir lab3`
1-d) `cd lab3`

This is the directory where you should keep files from this lab.

Arrays

Arrays

– An array is an ordered list of scalar values
– Array variables start with @ when referred in their entirety; examples:

```
my @animals = ("camel", "llama", "owl");
my @numbers = (23, 42, 69);
my @mixed   = ("camel", 42, 1.23);
```

– When referring to individual elements, use notation such as:

```
$animals[0] = 'Camel';
$numbers[4] = 70;
$mixed[1]++;
```

Arrays or Lists

– Perl arrays are dynamic, also called lists
Perl arrays are dynamic in a sense of extensibility: we can easily make them larger or shorter without a need to deal with details of memory management. This is why they are also called lists.
– Some examples:

```
my @a = (); # creating an empty array
$a[5] = 10; # array extended to: '','','','','',10
```

– In the above example, simply by assigning a value to `$a[5]` we are expanding an empty array to an array of length 6 (from `$a[0]` to `$a[5]`), with automatically filling in empty values of the elements from `$a[0]` `$a[4]`. More precisely, the empty values are actually the so-called `undef` Perl values, which will normally be presented as empty strings in a string, or 0 in a numerical expression. If you use the directive '`use strict;`' or '`use warnings;`', you would be warned if they are used before assigning a more concrete value.

```
$a[-2] = 9; # use of negative index, array is now
            # '','','','',9,10
```

– We can use a negative index to access the elements from the right end; e.g., `$a[-1]` is the last elements, `$a[-2]` the second last, etc.

```
print $#a;            # 5, index of the last element
print scalar(@a);     # 6, length of the array
for my $i (0..$#a) { # printing all elements
  print $a[$i]."\n"; }
```

Iterating over Arrays

There are several way of how we can use a for-loop to iterate over an array:

– The loop `foreach` (or its synonym `for`)

```
my @a = ("a", "b", "c");
foreach my $element (@a)
{ print $element; }
```

- the default variable `$_` can be used in the `foreach` loop

```
foreach (@a)
{ print; }
```

- or using `for` and the index

```
for (my $i=0; $i<=$#a; $i++) {
   print $a[$i]; }
```

The last example of the for-loop uses C-like form, similar to Java and some other languages. Namely, in the code:

```
for (my $i=0; $i<=$#a; $i++) {
   print $a[$i]; }
```

part `my $i=0` declares a local variable `$i` and sets it initially to 0. The loop is repeated while `$i` is less or equal `$#a`. The variable `$#a` is a special variable which denotes the last index of the array `@a`. Since array indexing starts generally from 0, it means that `$#a` is equal to $n - 1$, where n is the number of elements of array `@a`. The part `$i++` means that the index `$i` is incremented by 1 after each iteration of the loop.

More about Array Functions (Operators)

We will now cover a number of the most important predefined Perl functions for working with arrays. The first function **push** can be used to add one or more elements to the end of an array, and the function **pop** can be used to remove one element from the end of an array:

- **push** `@a,`*elements*`;` or **push(**`@a,`*elements*`);`
- Example:

```
@a = (1,2,3);          # @a = (1, 2, 3)
push @a, 4;            # @a = (1, 2, 3, 4)
```

- Built-in function **push** adds elements at the right end of an array
- Built-in functions generally do not require parentheses, but they are allowed, and sometimes needed to resolve ambiguities
- **pop** `@a;` removes and returns the rightmost element

```
$b = pop @a;       # $b=4, $a = (1, 2, 3)
```

Functions: shift, unshift, scalar; Array Flattening

Similarly to **pop** and **push**, the functions **shift** and **unshift** remove and add elements from an array, but from the left side.

- **shift** `@a;` removes leftmost element

```
@a = (3, 1, 2);
$b = shift @a;           # $b=3, $a = (1, 2)
```

- **unshift** `@a,` *elements*`;` adds at the left end

```
unshift @a, 5;           # @a = (5, 1, 2)
```

- Array flattening in Perl: If we try to put an array inside another array in a simple way, Perl will flatten the arrays into one array. If we really want to have an arrays of arrays, we must use references, which will be covered later.

```
@a = ((1,2,3),4,(5));     # @a = (1, 2, 3, 4, 5)
@a = (0,@a,6);            # @a = (0, 1, 2, 3, 4, 5, 6)
```

- **scalar** `@a;` returns length of an array
  It may be confusing why is a function for the length of array called **scalar**. There is no actually a predefined function in Perl dedicated only to array length. The function **scalar** simply enforces a scalar context on any expression, and a scalar context on an array forces it to be evaluated to the number of elements of an array. For this reason, for example, the following code works as shown:

  ```
  @a = (1,2,3); @b = ('a', 'b');
  print @a+@b;  # prints 5 because @a and @b evaluates to 3 and 2 in
                # the scalar context
  ```

- Remember that you can always used parentheses if you prefer, e.g.:

  ```
  $n = scalar(@a);
  unshift(@a,5);
  push(@a,1,2,3);   # etc.
  ```

Revisiting Function print

- **print**, like many other functions, can take a list of arguments; e.g.:

  ```
  print 'Print ', 'a ', 'list', "\n";
  # prints: Print a list   (with a newline at the end)
  ```

  The arguments are printed sequentially without any spaces
- This is why we have:

  ```
  @a = (1, 2, 3);
  print @a;       # prints: 123
  print "@a";     # prints: 1 2 3
  ```

- The second example uses string interpolation, where an array is expanded by inserting a space between elements.
- Note that we do not use comma when printing to a file handle:

  ```
  print STDERR "print can use a file handle\n";
  print $fh    "printing to a file";
  ```

The function **chomp** is another function that we saw before, which also can take a list as an argument. We saw that this function removes a trailing new-line character from a given string. If given a list as an argument, as in

```
chomp @a;
```

the function removes trailing new-line characters from all elements of `@a`.

Function sort

- **sort** `@a;`   sorts an array

  ```
  @a = sort (5, 1, 2);  # @a = (1, 2, 5)
  ```

- **sort** `BLOCK @a;`   can be used to modify sorting criterion
  `BLOCK` decides how to compare two variables `$a` and `$b` by returning a result similar to `$a-$b`; i.e., negative means `$a<$b`, zero means `$a=$b`, and positive means `$a>$b`, with the final increasing order

- predefined operators <=> (numeric) and `cmp` (string) are usually useful; some simple examples:

  ```
  @a = (2, 1, 10, ’b’, ’a’)
  @a = sort @a; # string order: 1, 10, 2, a, b
  @a = sort {$a<=>$b} @a; # numeric order: a b 1 2 10
                          # (a, b treated as zeros)
  @a = sort {$b cmp $a} @a; # inverse string order:
                            # b a 2 10 1
  ```

Functions split and join

- **split** /*regex*/, *string*; splits a string into array using a breaking regex pattern

  ```
  $s = "This is a sentence.";
  @a = split /[ .]+/, $s; # @a=(’This’,’is’,’a’,
                          #     ’sentence’)
  ```

- **join** *string*, *array*; joins array elements into a string by inserting given string between elements

  ```
  @a = (1, 2, 3);
  $s = join ’ <> ’, @a; # $s = ’1 <> 2 <> 3’
  ```

- We can also use parentheses: `join(`*string, array*`)`
- Array arguments can be given directly; e.g., `join ’<>’, 1, 2, 3;`
- And several arrays are flatten in one array; e.g., `join ’<>’, @a, @b;`

The above **split** function breaks the second string argument `$s` into pieces by separating them with matches obtained from the regular expression. In this example, the regular expression matches either a space or a period. The same result on this string would be obtained a more general regular expression such as `/\W+/`, which could be used for a simple tokenization.

The function **split** can take a third, optional argument, *LIMIT*, which would limit the number of elements that the string would be broken into.

As another example for the function **join**, consider a task where we need to concatenate variables `$login`, `$pwd`, `$uid`, `$gid`, `$gc`, `$home`, and `$sh` by inserting the a colon `’:’` character between them. It could be achieved with the command:

```
$rec=join(’:’,$login,$pwd,$uid,$gid,$gc,$home,$sh);
```

This example may look familiar to you if you have seen the format of the `/etc/passwd` file in Linux.

Functions grep and map

- **grep** `expr`, `@list` or **grep** `BLOCK @list` returns all elements of a list for which expression or block evaluates to true, when `$_` is given as the element of the list; example

  ```
  @a = grep {$_>0} @b; # @a gets positive elements
                       # of @b
  @foo = grep {!/^#/} @bar; # @foo gets all elements
      # (strings) from @bar that do not start with #
  ```

- **map** `expr`, `@list` or **map** `BLOCK @list` returns results of expression or block when applied on each element of the list, where `$_` given as element; example

  ```
  @a = map {lc $_} @b; # @a gets elements of @b in
                       # lowercase letters
  ```

- See `man perlfunc` for more about Perl functions

Step 2: Example with Arrays

- Type and test the following program in a file named ‘`lab3-array-examples.pl`’

```
my @animals = ("camel", "llama", "owl");
my @numbers = (23, 42, 69);
my @mixed   = ("camel", 42, 1.23);

print "animals are @animals
that is: $animals[0] $animals[1] $animals[2]\n";
print "There is a total of ",$#animals+1," animals\n";
print "There is a total of ",scalar(@animals),
      " animals\n";

$animals[5] = 'lion';
print "animals are @animals\n";
```

Submit: Submit the program `lab3-array-examples.pl` using the `submit-nlp` command.

Associative Arrays (Hashes)

Associative arrays or hashes are structures that associate some scalars called keys with other scalars called values. They are known in other programming languages also as maps (in Java) and dictionaries (in Python).

Associative Arrays (Hashes)

- Similar to array; associates keys with values
- Examples of use: phonebook, translation dictionary
- English-French translation example:

```
$fr{'one'}   = 'un'; $fr{'two'} = 'deux';
$fr{'three'} = 'trois';
$w = 'two';
print $w." in French is ".$fr{$w};
# Output: two in French is deux
$w = 'four';
print $w." in French is ".$fr{$w};
# Output: four in French is
# because $fr{'four'} is not assigned
```

Hashes: Declarations, Functions

- Hashes use `%` similarly like arrays use `@`, example

```
my %p;
my %p = (); # start with empty hash
my %p =
  ('one' => 'first', 'two' => 'second');
```

- The previous example shows how we can initialize a hash
- **exists** `$hash{$key}` checks if a hash element exists
- **delete** `$hash{$key}` deletes a hash element

- **keys** `$hash` gives an array of all keys
- **values** `$hash` gives and array of all values
- Keys and values are not returned in any particular order

Counting Words, Characters, and similar

- Hashes are convenient for counting words or other similar strings
- For example, if whenever we find a word `$w` in text

  ```
  $count{$w} += 1;  # add 1 to word count
  ```

  collects word counts in the hash `%count`
- If a word is not in the hash, the above expression adds it to the hash, and adding 1 makes the initial count 1, because an undef value is treated as a zero
- A sorted list of words can be obtained with

  ```
  @words = sort keys %count;
  ```

Iterating over a Hash

- Example of iterating over all key-value pairs in a hash:

  ```
  my %p=('one'=>'first', 'two' => 'second');
  foreach my $k (sort keys(%p)) {
      my $v = $p{$k};
      print "value for $k is $v\n";
  }
  ```

- If we do not care about sorting keys, we can also use the special function **each**:

  ```
  while (my ($k,$v) = each %p) {
    print "$k => $v\n";
  }
  ```

'Barewords' in Keys

- For more convenience, words without quotes, or so-called *barewords,* are allowed to be used in hashes like this:

  ```
  %p = (one => first, two => second);
  $p{three} = 'third';
  ```

- Even a starting minus sign is allowed, and used sometimes:

  ```
  %p = (-one => first, -two => second);
  $p{-three} = 'third';
  ```

- Even the following would work:

  ```
  $p{-three} = third;
  ```

  but not if we defined a subroutine called '`third`', so we should not use it.

For example, if we wanted a larger translation hash from English to French numbers, we could use:

```
my %fr = ( one => un, two => deux, three => trois, four => quatre,
  five => cinq, six => six, seven => sept, eight => huit,
  nine => neuf, ten => dix);
```

Barewords in Perl

- Since Perl uses so much special symbols (sigils) as starting symbols in variables, it is interesting to see how are identifiers without such sigils treated
  We usually do not have to use *barewords* in Perl much, but it is still useful to know how they are treated when we read Perl code.
- *Barewords* are words (i.e., identifiers) that appear without initial special symbol (sigil) such as `$`, `@`, `%`, `&`, and `*`
- If they are not reserved words, or pre-defined function names, they are treated as literal strings; or forbidden if `use strict` is in place
- For example, the following code produces output shown:

```
$s = one.two; print "$s\n"; # out: onetwo
sub one { return "(1)" }
$s = one.two; print "$s\n"; # out:(1)two
sub two { return "(2)" }
$s = one.two; print "$s\n"; # out:(1)(2)
```

Barewords in Keys (2)

- Back to barewords in keys, hash keys have a stronger preference for barewords and are allowed even with ‘`use strict;`’. For example:

```
use strict;

my %a;
sub p { print join(’,’,%a), "\n"; }

%a = (one => ’one’);   p; # out: one,one
sub one { return "(1)" }
%a = ( one => one );   p; # out: one,(1)
%a = ( &one => one );  p; # out: (1),(1)
%a = ( (one) => one ); p; # out: (1),(1)
%a=(); $a{one} = one;  p; # out: one,(1)
%a=(); $a{&one} = one; p; # out: (1),(1)
```

Step 3: Example with Associative Array

- Write, test, and submit the following program in a file called `lab3-test-hash.pl`

```
#!/usr/bin/perl
# File: lab3-test-hash.pl

sub four { return ’sub4’ }
sub fourth { return ’sub4th’ }

%p = (one => first, -two => second);
$p{-three} = third;
$p{four} = fourth;
$p{four2} = ’fourth’;

for my $k ( sort keys %p ) { print "$k => $p{$k}\n" }
```

Submit: Submit the program `lab3-test-hash.pl` using the `submit-nlp` command.

Step 4: `lab3-letter_counter_blanks.pl`

4-a) Copy the following files to your `lab4` directory:
`~prof6509/public/TomSawyer.txt`
`~prof6509/public/lab3-letter_counter_blanks.pl`

If you forgot the copy command on `timberlea` (a Linux system), remember that you can use the following commands:

```
cp ~prof6509/public/TomSawyer.txt .
cp ~prof6509/public/lab3-letter_counter_blanks.pl .
```

The first file is the novel "The Adventures of Tom Sawyer" by Mark Twain that we saw before, and the second file is a sample unfinished program.

4-b) Open the file `lab3-letter_counter_blanks.pl` and fill in three blanks.

The program counts the frequencies of letters (case insensitive) in an input text from standard input or from files specified in the command line. It is supposed to print the letters with their frequencies in the decreasing order of their frequency.

4-c) Run the program on the file `TomSawyer.txt` and save the output to the file `lab3-out_letters.txt` (the following is a one-line command):

```
./lab3-letter_counter_blanks.pl TomSawyer.txt >
                                  lab3-out_letters.txt
```

4-d) Submit `lab3-letter_counter_blanks.pl` and `lab3-out_letters.txt`

Submit: Submit the program `lab3-letter_counter_blanks.pl` and the file `lab3-out_letters.txt` using the `submit-nlp` command.

Step 5: `lab3-word_counter.pl`

Write a Perl program `lab3-word_counter.pl` that counts words from an input text from the standard input or files specified in the command line (i.e., as per usual diamond <> operator). All words must be turned into lowercase before counting. We will define a word here as a maximum sequence of Perl "word characters": letters, digits or the underscore, that is as a string captured by the regular expression `\w+`

You may want to copy your program `lab3-letter_counter_blanks.pl` and modify it, so that it counts words instead of letters.

The program must not print all extracted words. Instead, it should provide answers to the following two questions:
1. What are 10 most common words in the input text? and
2. How many words appear only once in the input text?

A word that appears only once in a text is called *hapax legomenon,* or in plural *hapax legomena*.

The program must produce output in the following format:

```
10 most common words are:
word1 word2 word3 word4 word5 word6 word7 word8 word9 word10
The number of hapax legomena is ???
```

where `word1` `word2` etc. are the ten most common words sorted from the most frequent to the least frequent. If some words among these ten most frequent words have the same frequency, then their exact order does not matter.

If input does not have 10 unique words, then print as many as there are, still sorted by frequency from the most frequent down. The string `???` above should be replaced with the number of hapax legomena.

You must include the file comment header as in previous labs in the following format:

```
#!/usr/bin/perl
# CSCI4152/6509 Fall 2022
# Program: lab3-word_counter.pl
# Author: Vlado Keselj, B00123456, vlado@dnlp.ca
# Description: The program is a part of Lab3 required submissions.
```

Run the program on the file `TomSawyer.txt`. Save the output to the file `lab3-out_word_counter.txt`:

```
./lab3-word_counter.pl TomSawyer.txt > lab3-out_word_counter.txt
```

Submit: Submit the files: `lab3-word_counter.pl` and `lab3-out_word_counter.txt`

References

References to Arrays and Hashes

A reference is a scalar pointing to another data structure, usually an array or a hash:

```
my @a=('Mon','Tue','Wed'); # an array
my %h = ('one' => 'first', 'two' => 'second'); # a hash

my $ref_a = \@a; # reference to an array
my $ref_h = \%h; # reference to a hash
```

- References are necessary in Perl to make more complex data structures, such as creating arrays of arrays

Using References (1): Scalar Variables as References

Method 1: If your reference is a simple scalar variable, then wherever the identifier of an array or hash would be used as a part of an expression, one can use the variable that is the reference to the array or the hash, as in following examples:

```
@array=@a;       #using an array
@array=@$ref_a; #using a reference to an array

$element=$a[0];       #using an array
$element=$$ref_a[0]; #using a reference
$$ref_a[0]='xxx';     #using a reference

%hash=%h;       #using a hash
%hash=%$ref_h; #using a reference

$value=$h{'one'};       #using a hash
$value=$$ref_h{'one'}; #using a reference
$$ref_h{'one'}='f';     #using a reference
```

Using References (2): Using Braces { }

Method 2: Regardless whether your reference is a simple scalar or not. As Method 1, but enclose the reference in `{ }`

```
@array=@a;         #using an array
@array=@{$ref_a}; #using a reference
```

```
$element=$a[0];         #using an array
$element=${$ref_a}[0]; #using a reference
```

```
$value=$h{'one'};         #using a hash
$value=${$ref_h}{'one'}; #using a reference
```

While this is optional for simple scalars (i.e., you can use Method 1), this is necessary otherwise — for example when you store references to arrays in a hash `%hash_of_ref_to_arrays`

```
$value=${$hash_of_ref_to_arrays{'one'}}[0];
```

Using References (3): Arrow Operator ->

Method 3: Accessing elements of arrays or hashes using references directly and using the arrow operator ->

Instead of:

```
$$ref_a[0]
$$ref_h{'one'}
```

one can use:

```
$ref_a->[0]
$ref_h->{'one'}
```

Using References (4): Omitting Arrow Operator

If the arrow -> is between bracketed indexes of arrays or hashes, e.g.,

```
$ref_a->[0]->[10] #$ref_a is a reference to an array
                   #storing references to arrays
$ref_a->[0]->{'k'} #$ref_a is a reference to an array
                    #storing references to hashes
$ref_h->{'one'}->{'k'} #$ref_h is a reference to a hash
                        #storing references to hashes
```

then the arrow between bracketed indexes can be omitted

```
$ref_a->[0][10]
$ref_a->[0]{'k'}
$ref_h->{'one'}{'k'}
```

Using References to Pass Arrays or Hashes to a Subroutine

Arrays and hashes can be passed to a subroutine via references:

```
sub print_array {
    my $ref_a=shift; #takes a reference to an array
                     #as a parameter
    foreach my $element (@$ref_a) {
        print "Element: $element\n"
    }
}
sub add_element {
    my ($ref_a, $element) = @_;
    push(@$ref_a, $element);
}
my @a=('Mon','Tue','Wed'); #array
add_element(\@a,'Thu');
print_array(\@a); # array is changed
```

Passing Arrays or Hashes to Subroutine Directly

We can also pass arrays or hashes directly as list of arguments:

```
sub print_array {
  foreach my $e (@_) { print "Element: $e\n" }
}

sub print_hash {
  my %p = @_;
  foreach my $k (keys %p) { print "$k => $p{$k}\n" }
}

print_array(1, 2, 3, 'four');
print_hash( one=>first, two=>second,
  'any key' => 'some value' );
```

Let us see more closely what happens when we use call:

```
print_hash( one=>first, two=>second, 'any key' => 'some value' );
```

with the function:

```
sub print_hash {
  my %p = @_;
    foreach my $k (keys %p) { print "$k => $p{$k}\n" }
}
```

Since the literal hash `(one=>first, two=>second, 'any key' => 'some value')` is passed as an array, it is flattened into an array, that may look like:
`('one', 'first', 'two', 'second', 'any key', 'some value')`
The order may be different, but each key must be followed with the corresponding value. When the function `print_hash` is executed, this array is received as the array `@_`. After the command `my %p = @_;`, the array is translated again into a hash named `%p`, and the function proceeds.

Step 6: `lab3-word_counter2.pl`

Copy the previous program `lab3-word_counter.pl` to `lab3-word_counter2.pl`. You should now edit the program `lab3-word_counter2.pl` and add a new subroutine to it. The subroutine named `f` should take two parameters: a word and a reference to the hash that stores the frequencies of words. It should return the frequency of the input word (0, if it is not present in the hash).

Test the program `lab3-word_counter2.pl` on the file `TomSawyer.txt` to find the frequencies of the words: 'tom', 'sawyer', and 'huck'.

You should remove the previous output of the program `lab3-word_counter.pl`, and add new code to print output for the three given words. You can add some text, but the program should basically print three lines, and mention the tree words and their frequencies in these three lines.

Submit: Submit the program `lab3-word_counter2.pl`

Using Perl Modules and an Ngrams Module

Perl libraries are usually implemented as modules, and we will look at a particular example of the `Text::Ngrams` module. The use of this module is related to the CNG method for classification, which is or will be discussed in the lectures. This discussion of the module is also somewhat covered in lectures, but we will now go through a hands-on exercise of using it. The Perl module file is called `Ngrams.pm` and there is an associated program called `ngrams.pl`. The module and the program are open-source, and can be found in the CPAN archive, but the newest version are also available on `timberlea`, in the directory `~prof6509/public`. The modules are typically installed system-wide and Perl can automatically find them. They can also be installed on a per-user basis, in a more systematic way or in a more ad-hoc way.

Step 7: Copy `Ngrams.pm` and `ngrams.pl`

We will use here a very local ad-hoc installation. First, you will need to copy the appropriate files to your current directory using the commands:

```
cp ~prof6509/public/ngrams.pl .
cp ~prof6509/public/Ngrams.pm .
```

These files may be already installed on `timberlea`, but to use the appropriate local version, we will do a couple additional operations and checks. Create a subdirectory `Text` and copy the module there:

```
mkdir Text
cp Ngrams.pm Text
```

It is important that the module `Ngrams.pm` is located in the directory `Text`, in other words the pathname from the current directory must be `Text/Ngrams.pm`, because the later execution of the `ngrams.pl` scripts relies on this path.

Step 8: Checking Modified `ngrams.pl`

The file `ngrams.pl` is a part of the package `Text::Ngrams`, which is publicly available through CPAN. However, the version available here is slightly modified to make sure that it uses the Ngrams module in the current directory. You should verify that you have this version of the file. The beginning of the file `ngrams.pl` should look as follows (you can check this using the command '`more ngrams.pl`'):

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

use strict;
use vars qw($VERSION);
#<? read_starfish_conf(); echo "\$VERSION = $ModuleVersion;"; !>#+
$VERSION = 2.007;#-

use lib ’.’;

use Text::Ngrams;
use Getopt::Long;
...
```

The line ‘`use lib ’.’;`’ is important, since it directs Perl to give priority to first searching for the module in the current directory, and then, if it is not found, to search for other versions that may be available in the system. You can test the program `ngrams.pl` by typing:

```
./ngrams.pl
```

then typing some input, and pressing ‘`C-d`’; i.e., Control-D combination of keyboard keys. For example, if you type input:

```
natural language processing
```

you should get the output:

```
BEGIN OUTPUT BY Text::Ngrams version 2.007

1-GRAMS (total count: 28)
FIRST N-GRAM: N
 LAST N-GRAM: _
-------------------------
_ 3
A 4
C 1
E 2
G 3
I 1
L 2
N 3
O 1
P 1
R 2
S 2
T 1
U 2

2-GRAMS (total count: 27)
FIRST N-GRAM: N A
 LAST N-GRAM: G _
-------------------------
```

```
_ L 1
_ P 1
A G 1
A L 1
A N 1
A T 1
C E 1
E _ 1
E S 1
G _ 1
G E 1
G U 1
I N 1
L _ 1
L A 1
N A 1
N G 2
O C 1
P R 1
R A 1
R O 1
S I 1
S S 1
T U 1
U A 1
U R 1

3-GRAMS (total count: 26)
FIRST N-GRAM: N A T
 LAST N-GRAM: N G _
------------------------
_ L A 1
_ P R 1
A G E 1
A L _ 1
A N G 1
A T U 1
C E S 1
E _ P 1
E S S 1
G E _ 1
G U A 1
I N G 1
L _ L 1
L A N 1
N A T 1
N G _ 1
N G U 1
O C E 1
P R O 1
R A L 1
R O C 1
```

```
S I N 1
S S I 1
T U R 1
U A G 1
U R A 1

END OUTPUT BY Text::Ngrams
```

These are the character n-grams of up to the size 3 of the given text, with their counts.

Step 9: Test that `ngrams.pl` is using the local version of Ngrams module

To test that the program is using the correct version of the module `Ngrams.pm` you can try inserting a 'die' command at the beginning of the module, and see that the program `ngrams.pl` will not work. The beginning of the module should look like:

```
# (c) 2003-2022 Vlado Keselj http://web.cs.dal.ca/~vlado
#
# Text::Ngrams - A Perl module for N-grams processing

die;

package Text::Ngrams;

use strict;
require Exporter;
```

It is important to note that this is the copy of the module in the subdirectory `Text`. After this small test, remove again the line '`die;`' from the `Ngrams.pm` module.

Step 10: Using the Ngram module

We will use now the Ngrams module to extract character n-grams from the the novel "The Adventures of Tom Sawyer" by Mark Twain. The novel is available on bluenose as the file `~prof6509/public/TomSawyer.txt` and you should copy it to your directory `lab3` using the command:

```
cp ~prof6509/public/TomSawyer.txt .
```

Run the script `ngrams.pl` with default parameters, on the file `TomSawyer.txt` and store the output in the file `lab3-ngram-output.txt`. You can do it using the following command:

```
./ngrams.pl TomSawyer.txt > lab3-ngram-output.txt
```

You can take a look at the file `lab3-ngram-output.txt` using the `more` command. Before submitting, you must compress the `lab3-ngram-output.txt` file using the command:

```
gzip lab3-ngram-output.txt
```

which will replace the file with the compressed file `lab3-ngram-output.txt.gz`.

Submit: Submit the output file `lab3-ngram-output.txt.gz` using the `submit-nlp` command.

This is the end of Lab 3.