Faculty of Computer Science, Dalhousie University *22/24-Jan-2020*

CSCI 4152/6509 — Natural Language Processing

Lab 2: Perl Tutorial 2

Lab Instructor: Dijana Kosmajac, Tukai Pain
Location: rm1102, Mona Campbell building

Perl Tutorial 2

Lab Overview

- Use of Regular Expressions in Perl
- This topic is discussed in class, we will see some more examples in this lab
- The second part of the lab includes some practice with Regular Expressions
- Practice with processing Character N-grams

The lab starts with a presentation about regular expressions in Perl and some additional Perl functions. After that there will be practice exercises to be done on the bluenose server.

Files to be submitted:

1. `matching.pl`
2. `matching-data.pl`
3. `word-counter.pl`
4. `replace.pl`
5. `ngram-output.txt.gz`
6. `line-count.pl`

Some References about Regular Expressions in Perl

- To read more (e.g., on bluenose):
  - `man perlrequick`
  - `man perlretut`
  - `man perlre`
- Same information on:
  `http://perldoc.perl.org/perlrequick.html`
  `http://perldoc.perl.org/perlretut.html`
  `http://perldoc.perl.org/perlre.html`
- Used for string matching, searching, transforming
- Built-in Perl feature

Introduction to Regular Expressions

- A simple example:

```
if ("Hello World" =~ /World/) {
  print "It matches\n";
} else {
```

```
    print "It does not match\n";
}
```

Regular Expressions: Basics

- A simple way to test a regular expression:

```
while (<>)
{ print if /book/ }
```

  prints lines that contain substring 'book'
- /chee[sp]eca[rk]e/ would match: cheesecare, cheepecare, cheesecake, cheepecake
- option /i matches case variants; i.e., /book/i would match Book, BOOK, bOoK, etc., as well
- Beware that substrings of words are matched, e.g.,
  "That hat is red" =~ /hat/; matches 'hat' in 'That'

RegEx — No match

```
if ("Hello World" !~ /World/) {

  print "It doesn't match\n";

} else {

  print "It matches\n";

}
```

Character Classes (1)

/200[012345]/	match one of the characters
/200[0-9]/	character range
/From[^:!]/	match any character but : or !
/[^a]at/	does not match 'aat' or just 'at' but does 'bat', 'cat', '0at', '%at, etc.
/[a^]at/	matches 'aat' or '^at'
/[^a-zA-Z]the[^a-zA-Z]/	multiple ranges
/[0-9ABCDEFa-f]/	match a hexadecimal digit

Character Classes (2)

.	(period) any character but new-line
\d	any digit; i.e., same as [0-9]
\D	any character but digit
\s	any whitespace character; e.g., space, tab, newline
\S	any character but whitespace; i.e., printable
\w	any word character (letter, digit, underscore)
\W	any non-word character; i.e., any except word characters

Some more examples:

/\d\d:\d\d:\d\d/	matches a hh:mm:ss time format

`/[\d\s]/` matches any digit or whitespace
`/\w\W\w/` matches a word char, followed by non-word char, followed by word char
`/..rt/` matches any two chars followd by 'rt'
`/end\./` matches 'end.'

Word Boundary Anchor (`\b`)

- `\b` is word boundary anchor. It matches inter-character position where a word starts or ends; e.g., between `\w` and `\W`
- Examples:

```
$x = "Housecat catenates house and cat";
$x =~ /cat/       matches cat in 'housecat'
$x =~ /\bcat/     matches cat in 'catenates'
$x =~ /cat\b/     matches cat in 'housecat'
$x =~ /\bcat\b/   matches 'cat' at end of string
```

^ $

```
"housekeeper" =~ /keeper/; # matches
"housekeeper" =~ /^keeper/; # doesn't match
"housekeeper" =~ /keeper$/; # matches
"housekeeper\n" =~ /keeper$/; # matches

"keeper" =~ /^keep$/; # doesn't match
"keeper" =~ /^keeper$/; # matches

"" =~ /^$/; # ^$ matches an empty string
```


Matching - choices

```
"cats and dogs" =~ /cat|dog|bird/; # matches "cat„
"cats and dogs" =~ /dog|cat|bird/; # matches "cat"

"cab" =~ /a|b|c/    # matches "c”
                        # /a|b|c/ == /[abc]/

/(a|b)b/;           # matches 'ab' or 'bb'
/(ac|b)b/;          # matches 'acb' or 'bb'
/(^a|b)c/;          # matches 'ac' at start, 'bc' anywhere
/(a|[bc])d/;        # matches 'ad', 'bd', or 'cd'
/house(cat|)/;      # matches 'housecat' or 'house'
/house(cat(s|)|)/;  # matches 'housecats', 'housecat' or
                    #'house'. Note groups can be nested.
/(19|20|)\d\d/;     # match years 19xx, 20xx, or xx

"20" =~ /(19|20|)\d\d/;
             # matches null alternative
             # '()\d\d', because '20\d\d' can't match
```


Repetitions

`a?` means: match 'a' 1 or 0 times

`a*` means: match 'a' 0 or more times, i.e., any number of times

`a+` means: match 'a' 1 or more times, i.e., at least once

`a{n,m}` means: match at least n times, not more than m times.
`a{n,}` means: match at least n or more times
`a{n}` means: match exactly n times

```
/[a-z]+\s+\d*/
/(\w+)\s+\1/        match doubled words
/y(es)?/i           'y', 'Y', or case-insensitive 'yes'
```


Extractions

```
# extract hours, minutes, seconds
if ($time =~ /(\d\d):(\d\d):(\d\d)/)
{ # match hh:mm:ss format
  $hours = $1;
  $minutes = $2;
  $seconds = $3;
}

($h, $m, $s) = ($time =~ /(\d\d):(\d\d):(\d\d)/);

/(ab(cd|ef)((gi)|j))/;
 1  2      34

/\b(\w\w\w)\s\1\b/;
- backreferences
```


selective grouping

match a number, $1-$4 are set, but we want $1

```
/([+-]?\ *(\d+(\.\d*)?|\.\d+)([eE][+-]?\d+)?)/;
```

match a number faster , only $1 is set

```
/([+-]?\ *(?:\d+(?:\.\d*)?|\.\d+)(?:[eE][+-]?\d+)?)/;
```

match a number, get $1 = entire num., $2 = exp.

```
/([+-]?\ *(?:\d+(?:\.\d*)?|\.\d+)(?:[eE]([+-]?\d+))?)/;
```

– Grouping not exported if (?:regex)

Controlling greediness

```
$x = "the cat in the hat";

$x =~ /^(.*)(at)(.*)$/;
   # matches,
   # $1 = 'the cat in the h'
   # $2 = 'at'
   # $3 = '' (0 characters match)

$x =~ /^(.*?)(at)(.*)$/;
   # matches,
   # $1 = 'the c'
   # $2 = 'at'
   # $3 = ' in the hat'
```

Greediness

`a??` means: match 'a' 0 or 1 times. Try 0 first, then 1.

`a*?` means: match 'a' 0 or more times, i.e., any number of times, but as few times as possible

`a+?` means: match 'a' 1 or more times, i.e., at least once, but as few times as possible

`a{n,m}?` means: match at least n times, not more than m times, as few times as possible

`a{n,}?` means: match at least n times, but as few times as possible

`a{n}?` means: match exactly n times. Because we match exactly n times, a{n}? is equivalent to a{n} and is just there for notational consistency.

Look-aheads, look-behinds

```
$x = "I catch the housecat 'Tom-cat' with catnip";

$x =~ /cat(?=\s)/;
      # matches 'cat' in 'housecat'
@catwords = ($x =~ /(?<=\s)cat\w+/g);
      # matches,
      # $catwords[0] = 'catch'
      # $catwords[1] = 'catnip'
$x =~ /\bcat\b/;
      # matches 'cat' in 'Tom-cat'
$x =~ /(?<=\s)cat(?=\s)/;
      # doesn't match; no isolated 'cat' in
      # middle of $x

$x =~ /(?<!\s)foo(?!bar)/;
```


s///

```
s/regexp/replacement/modifiers

$x = "Time to feed the cat!";
$x =~ s/cat/hacker/;
  # $x contains "Time to feed the hacker!"

$strong = 1  if $x =~ s/^(Time.*hacker)!$/$1 now!/;

$y = "'quoted words'";
$y =~ s/^'(.*)'$/$1/;   # strip single quotes,
                        # $y contains "quoted words"

$x =~ s/(?<=\s)cat(?=\s)/dog/g;
```


s///...

```
$x = "I batted 4 for 4";
$x =~ s/4/four/;   doesn't do it all:
                   $x contains "I batted four for 4"

$x = "I batted 4 for 4";
$x =~ s/4/four/g;
            does it all:
            $x contains "I batted four for four"

$x = "Bill the cat";
$x =~ s/(.)/$ch{$1}++;$1/eg;
            final $1 replaces char with itself

print "frequency of '$_' is $ch{$_}\n"
  for sort {$ch{$b} <=> $ch{$a}} keys %ch;
```


Useful perl functions

See perlfunc

`chomp(@list)` – removes trailing newline from each element of the list

`grep(EXPR,@list), grep BLOCK @list` – evaluates EXPR for each element of the list and returns elements for which EXPR was true:
`@foo = grep {!/^#/} @bar; # weed out comments`
modification of $_ in EXPR modifies the list

`map BLOCK @list` – runs BLOCK for each element of the array and returns a list of results

`join(EXPR,@list)` – joins elements of the list.
`$rec=join':',$login,$pwd,$uid,$gid,$gc,$home,$sh);`

more perl functions

`length(EXPR)` – return the length of the expression

`pop(@list)`
`push(@list,@elements)`
`shift(@list)`
`unshift(@list,@elements)`
`scalar(@list)` – length of the list

`substr(EXPR,BEG,LENGTH)` – selects a fragment from the EXPR

`sprintf(FORMAT,@arguments)` – like in C

`split(PATTERN, STRING, LIMIT)` – splits STRING on a regular expression PATTERN and returns a list of remaining items

`sort BLOCK @list` – sorts list according to BLOCK comparison criterion.

Step 1. Logging in to server bluenose

1-a: Login to the server bluenose
As in previous lab, login to your account on the server bluenose.

1-b: Check permissions of your course directory `csci4152` or `csci6509`:
Important: Before you continue, you must check permissions on your course directory, which is either `csci4152` or `csci6509` depending on the course you are enrolled in. You were supposed to create this directory in the lab 1, but if you did not, you should create it now.

You should check permissions of this directory using the command:

```
ls -ld csci4152
```

or

```
ls -ld csci6509
```

You need to use option `-l` to show the permissions, and the option `-d` to prevent subdirectory listing, which would clutter the output.

The output of the command must start with '`drwx------`', which means that only you have read, write, and execute permissions. If it does not, for example, if it starts with '`drwxr-xr-x`' or similar, you must fix the permissions using the command:

```
chmod 0700 csci4152
```

or

```
chmod 0700 csci6509
```

1-c: 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.

1-d: `mkdir lab2`
`cd lab2`
Now, using the command '`mkdir lab2`' create the directory `lab2`. After this, you should make this directory your current directory '`cd lab2`'.

Step 2: Testing Regular Expressions

- Create file called `matching.pl` with the following content

```
#!/usr/bin/perl

while (<>) {
   print if /book/;
}
```

- Make it executable and run it using commands:

```
chmod u+x matching.pl
./matching.pl
```

Enter some input lines; some of them including the word 'book' and some of them not. Notice how the lines that include the word book are echoed back, while the lines not including it are not.

Remember that you can end your input from the keyboard by pressing Control-d (`C-d`). This is equivalent to redirecting the input from a file using `./matching.pl < some_file.txt`, or by specifying input as `./matching.pl some_file.txt`.

Submit: Submit the program `matching.pl` using the `submit-nlp` command. Remember that the command is executed in the following way on bluenose:

```
~vlado/public/submit-nlp matching.pl
```

You will be prompted to enter your CSID and password.

Step 3: Using DATA

Write a program called `matching-data.pl` with the following content:

```
#!/usr/bin/perl
# Program: matching-data.pl

sub testre {
  my $re = shift; my $line = shift;
  if ($line =~ /$re/) {
    print "/$re/ MATCH: $`>>>$&<<<$'";
  } else {
    print "/$re/ NOMATCH: $line";
  }
}

while (<DATA>) {
  &testre('book', $_); # testing /book/;
}

__DATA__
This line has book in it.
How about textbook?
This is capitalized word "Book"
```

This program shows a bit more elaborate testing of regular expressions. We use a subroutine 'testre' for this task. It demonstrates that we can embed variables in a regular expression (variable '`$re`'). The program also demonstrates the use of words '`__DATA__`' and '`DATA`' in Perl. The token '`__DATA__`' denotes the end of a Perl script, so content after this token is not read by the compiler. We can use this space to provide some data content which is accessible for reading using the special file handle `DATA`'. Similarly to the use of <> for reading from the standard input, the expressing `<DATA>` reads the next line of input from the DATA part of the program.

You can extend and test this program if you want, with more regular expressions and more DATA lines.

Submit: Submit the program `matching-data.pl` using the `submit-nlp` command.

Step 4: Counting words

Write a program called `word-counter.pl` with the following content:

```
#!/usr/bin/perl
#word-counter.pl
use warnings;
use strict;
```

```
my $tot=0;
while (<>) {
  while (/[a-zA-Z]+/g) {
    $tot++;
  }
}

print "\nTotal number of words: $tot\n";
```

This program searches for words, where we define a word as a maximal sequences of letters. Notice the use of the `//g` modifier, which means a global search for the regular expression. In other words, the condition in the second while loop will keep matching letters, one after another, in the default variable `$_` as loon as the matches are found on this line.

Make the program executable and run it using commands:

```
chmod u+x word-counter.pl
./word-counter.pl
```

Enter some input lines. Remember that you can end your input from the keyboard by pressing Control-d (`C-d`). This is equivalent to redirecting the input from a file using `./word-counter.pl < some_file.txt`, or by specifying input as `./word-counter.pl some_file.txt`. After the input ends, the program will print out the total number of words.

Submit: Submit the program `word-counter.pl` using the `submit-nlp` command.

Step 5: Simple task 1

Write a program called `replace.pl` that reads text from the standard input or from files specified as a command line parameters (i.e., use the operator <>) and for each line replaces all case-insensitive occurrences of the word `book` (i.e, `book`, or `BOOK`, or `Book` or `bOOk` etc.) with lowercase string `book`

You may want to use the `s///` operator, with modifiers that allow for global search and for case insensitive search.

You may want to use the following template of the code that needs filling in with a line.

```
#!/usr/bin/perl
#replace.pl
use warnings;
use strict;

while (<>) { # <> reads one line of input into the default variable

    ### Below you would need to add a missing line of code
    ### that will replace within the default variable
    ### all the occurances of case-insensitive 'book' (book, Book,
    ### BOOK, etc) with the lowercase string 'book'
    ### MISSING LINE GOES HERE

    ### AFTER THE MISSING LINE

    print; #this line prints the default variable $_
}
```

To make program easier, do not use the word boundary anchors. This means that the word 'BOOKING', for example, will be replaced with the word 'bookING'.

Submit: Submit the program `replace.pl` using the `submit-nlp` command.

Using Perl modules and an Ngrams Module

We will now focus on looking at some issues related to using the Perl modules, with 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 bluenose, 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 6: 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 bluenose, 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 7: 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;
#-
# $Revision: 1.26 $
```

```
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 8: 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) 2003-2019 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 9: 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 `lab2` 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 `ngram-output.txt`. You can do it using the following command:

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

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

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

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

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

Step 10: Basic I/O

- We have seen basic "diamond" operator <> for reading input
- For output, we can use `print`
- `printf` can be used for formatted output
- We can also explicitly open and close files using command `open` and `close`
- `print` can be used to print to a file
- Let us look at some examples

Some I/O Code Snippets

We can read the standard input, or from files specified in the command line and print using the following code snippet:

```
while ($line = <>) { print $line }
```

or using the default variable `$_`:

```
while (<>) { print }
```

The following two lines show different behaviour of <> depending on the context:

```
$line = <>;  # reads one line
@lines = <>; # reads all lines,

print "a line\n"; # output, or
printf "%10s %10d %12.4f\n", $s, $n, $fl;
     # formatted output
```

The above examples show the use of the diamond operator (<>) to read input, the use of command `print` to print output, and the command `printf` to print formatted output, in a similar way as in the C programming language. The string after `printf` maybe puzzling if you have not seen the 'printf' function in C. The format specification `%10s` specifies that the first variable after the format string `$s` should be printed in at least 10 characters, then `%10d` specifies that the number `$n` should be printed as a decimal number in at least 10 positions, and `%12.4f` specifies that the number `$fl` is a floating-point number (number with a decimal period), which should be printed in 12 characters, with exactly 4 digits after the decimal point.

Reading from a File

```
my $filename = 'file.txt';

#using file handle $fh

open(my $fh, '<', $filename);

my $line = <$fh>;

print $line;

close $fh;
```

Reading from a File, with Error Check after Opening

```
my $filename = 'file.txt';

#using file handle $fh

open(my $fh, '<', $filename)
    or die "Cannot open file $filename $!";
```

```
my $line = <$fh>;

print $line;

close $fh;
```

Writing to a File

```
my $filename = 'file.txt';

#using file handle $fh

open(my $fh, '>', $filename)
    or die "Cannot open file $filename $!";

print $fh "new first line\n";

close $fh;
```

Appending to a File

```
my $filename = 'file.txt';

#using file handle $fh

open(my $fh, '>>', $filename)
    or die "Cannot open file $filename $!";

print $fh "new last line\n";

close $fh;
```

Step 11: Count Number of Lines

- Write a program `line-count.pl`
- Usage: `./line-count.pl file.txt`
- Output: `file.txt has 124 lines`
- Submit `line-count.pl` using `nlp-submit`

Write a Perl program named `line-count.pl` which counts number of lines in a given file. The program is run using the line:

```
./line-count.pl file.txt
```

where `file.txt` is an existing file. The output of the program must be in the following format to the standard output:

```
file.txt has 124 lines
```

assuming that the file `file.txt` has 124 lines.

Some hints: In order to run the program as `./line-count.pl`, the program must be user-executable and the first line must start with the so-called "hash-bang" combination: `#!/usr/bin/perl` To get the file name from the command line you can use the line:

```
my $fname = shift;
```

at the beginning of the program. One common way to open a named file for reading in Perl is '`open(F, "<$fname")`'. The symbol `F` in this code is called a file handle and you can read from it using `<F>` expression, which reads the next line of the file. The output could be printed using the command:

```
print "$fname has $cnt lines\n";
```

where `$fname` is the variable containing the name of the file, and `$cnt` contains the number of lines of the file.

Submit: Submit the output file `line-count.pl` using the `submit-nlp` command.

Step 12: End of the Lab

- Make sure that you submitted all required files: `matching.pl`, `matching-data.pl`, `word-counter.pl`, `replace.pl`, `ngram-output.txt.gz`, `line-count.pl`
- End of the lab.