Faculty of Computer Science, Dalhousie University *20-Sep-2024*

CSCI 4152/6509 — Natural Language Processing

Lab 2: Perl Tutorial 2

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, Magdalena Jankowska, Jacek Wolkowicz

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
- Overview of some Perl input/output functions and some hands-on exercises

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 `timberlea` server.

Slide notes:

Lab Evaluation

- The lab will be evaluated as a part of an assignment with the same submission deadline as the assignment, which will be at least one week after the lab.
- Files to be submitted by the end of the lab are:
  1. `lab2-matching.pl`
  2. `lab2-matching-data.pl`
  3. `lab2-word-counter.pl`
  4. `lab2-replace.pl`
  5. `lab2-line-count.pl`

Regular Expressions

We will cover basics of using regular expressions in Perl in this section. For more information, you can find a lot of additional information on Internet. Some web pages are given below, and they correspond to the manual pages (command 'man') on `timberlea`, or any other Linux servers that have Perl installed.

Some References about Regular Expressions in Perl

- To read more (e.g., on timberlea):
  - `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

Let us look at a simple example of using a regular expression in Perl:

Introduction to Regular Expressions

– A simple example:

```
if ("Hello World" =~ /World/) {
  print "It matches\n";
} else {
  print "It does not match\n";
}
```

– What is the output of this code snippet?

The regular expressions are by default delimited with slash characters (`/.../`) in Perl. The operator match '`=~`' is used to match a string with a regular expression.

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`'

If the match operator (`=~`) is not used, it is assumed that the regular expression is matched with the default variable (`$_`). So the code '`print if /book/`' means the same as '`print $_ if $_ =~ /book/`', which is the same as '`if ($_ =~ /book/) { print $_ }`'.

RegEx — No match

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

  print "It doesn't match\n";

} else {

  print "It matches\n";

}
```

The operator no-match (`!~`) is a negation of the match operator (`=~`).

Character Classes

A character class is a regular expression that matches any character from a given set. A character class typically starts and ends with square brackets (‘`[`’ and ‘`]`]), but there are also some specially designated character classes.

Character Classes with Brackets

`/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, Special notation

`.`	(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
```

Anchors ^ and $

```
"housekeeper" =~ /keeper/;    # match
"housekeeper" =~ /^keeper/;   # no match
"housekeeper" =~ /keeper$/;   # match
"housekeeper\n" =~ /keeper$/; # match

"keeper" =~ /^keep$/;    # no match
"keeper" =~ /^keeper$/;  # match

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

Disjunction (or Alternatives, 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". Groups can be nested.
/(19|20|)\d\d/; # match years 19xx, 20xx, or xx

"20" =~ /(19|20|)\d\d/; # matches null alternative
    # /(19|20)\d\d/ would not match
```

Iterations

```
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 and
                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*/  letters a-z, spaces, and maybe digits
/(\w+)\s+\1/    match doubled words (back reference)
/y(es)?/i       'y', 'Y', or case-insensitive 'yes'
```

The example above `/(\w+)\s+\1/` may be unclear since it uses the back-reference feature that we did not mention yet. The back reference used is `\1`, which means that this part must be equal to the part captured by the

first pair of parentheses in the regular expression. Hence, this regular expression will match a word, followed by a number of whitespace characters, followed by that same word; e.g., '`alpha alpha`'.

Captures (or Extractions)

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

# Another way to capture substrings:
($h, $m, $s) = ($time =~ /(\d\d):(\d\d):(\d\d)/);

/(ab(cd|ef)((gi)|j))/;
 1  2      34          # opening parentheses order

/\b(\w\w\w)\s\1\b/;    # use of backreferences
```

When a string is matched with a regular expression, we can capture parts of the matched string which correspond to parts of regular expressions between parentheses. In this way, parentheses have a double-role: on one side, we used them for grouping frequently needed with disjunctions and iterations, and in the same time they are used to capture certain parts of the matched string.

Selective Grouping

```
# may want to use grouping but no substring capture
# use modified grouping: (?:regex)

# E.g.: 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+))?)/;
```

Greediness in regex Matching

```
# by default: left-most longest match (greedy)

$x = "the cat in the hat";

$x =~ /^(.*)(at)(.*)$/;
  # matches:
  # $1 = 'the cat in the h'  (left-most longest)
  # $2 = 'at'
  # $3 = '' (0 characters match)
```

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

Shortest Matches (Minimizing Greediness)

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

a*?      # match 'a' 0 or more times, but as few times
         # as possible

a+?      # match 'a' 1 or more times, but as few times
         # as possible

a{n,m}? # match at least n and not more than m times,
         # but as as few times as possible

a{n,}?  # match at least n times, but as few times as
         # possible

a{n}?   # match exactly n times; so a{n}? is equivalent
         # to a{n}
```

Look-aheads, Look-behinds

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

$x =~ /cat(?=\s)/; # look-ahead
      # matches 'cat' in 'housecat'
@catwords = ($x =~ /(?<=\s)cat\w+/g); # look-behind
      # 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)/; # negative look-behind and
                           # negative look-ahead
```

Look-aheads and look-behinds are relatively advanced features, which are rarely used. Their behaviour can usually be obtained by using regular expressions without them, but their use can result in shorter and less complex code.

Replacements: `s/regex/replacement/`

```
# General format: s/regexp/replacement/modifiers
# 1-letter modifiers, also called flags or options
```

```
$x = "Time to feed the cat!";
$x =~ s/cat/hacker/;
  # $x now 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/\bcat\b/dog/g; # modifier 'g' used
                       # to replace all matches
```

More Replacement Examples

```
$x = "I batted 4 for 4";
$x =~ s/4/four/; # does not replace all 4s:
                 # $x contains "I batted four for 4"

$x = "I batted 4 for 4";
$x =~ s/4/four/g; # flag "g" (global) replaces all:
                  # $x contains "I batted four for four"

$x = "Bill the cat";
$x =~ s/(.)/$ch{$1}++;$1/eg; # flag "e" (evaluate)
         # counts characters, and final $1 simply
         # replaces char with itself

# Printing characters by frequency, sorted:
print "frequency of '$_' is $ch{$_}\n"
  for sort {$ch{$b} <=> $ch{$a}} keys %ch;
```

The last example uses hashes (or associative arrays), which we did not cover yet in more detail, so we will give here a more detailed explanation. First, the substitution uses the option 'e' in replacement, which stands for 'evaluation'. This means that for each match in the first part of the substitution command, it is replaced with the result of the code in the replacement part. This code keeps counts of different characters in the hash `%ch` (a particular element of the hash is `$ch{$1}`), and it leaves the character unchanged by having the last expression `$1`. The rest of the code prints characters and their frequencies, sorted by their frequency. The use of the `sort` function will be explained later.

Slide notes:

End of Regular Expressions

- This is the end of the review of regular expressions in Perl
- After this point, there are Hands-on Exercises that you need to complete

Hands-on Exercises with Regular Expressions in Perl

Step 1. Logging in to server timberlea

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

Step 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
```

Step 1-c: Change directory to `csci4152` or `csci6509`
Change your directory to `csci4152` or `csci6509`, whichever is your registered course.

Step 1-d: Create directory `lab2` and enter it:

```
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`', in other words, you should enter this directory.

Step 2: Testing Regular Expressions

- Create file called `lab2-matching.pl` with the following content

```
#!/usr/bin/perl

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

This is a simple file which reads the lines from the standard input, and prints them if they contain the string 'book'. You can notice the while-loop which reads lines and stores each line in the default variable ('`$_`') before executing the iteration. The 'print' command prints the default variable since it is not given an argument, and you can also notice an inverted use of the if-statement where the action comes before 'if' and the condition comes after. Perl allows for this construct, which is not common in other languages.

- Make it executable and run it using commands:

```
chmod u+x lab2-matching.pl
```

```
./lab2-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 `./lab2-matching.pl < some_file.txt`. In both cases, the input to the program is coming from the standard input. The Perl diamond operator (<>) also allows that the input file is specified as an argument; for example, as `./lab2-matching.pl some_file.txt`, although this is not what is considered the standard input.

Submit: Submit the program `lab2-matching.pl` using the `submit-nlp` command. Remember that the command is executed in the following way on `timberlea`:

```
submit-nlp lab2-matching.pl
```

You will be prompted to enter your CSID and password.

Step 3: Using DATA

Write a program called `lab2-matching-data.pl` with the following content. You should first read and make sure to understand the code. An explanation of the code is given after the code below.

```
#!/usr/bin/perl
# Program: lab2-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 '`$re`'). After any regular expression match, the special variables `$`` `, `$&`, and `$'` are set to the part of string before match (`$`` `), the part of string matched (`$&`), and the part of string after the match (`$'`).

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.

Another feature demonstrated by the program is the use of the subroutine `testre`. It is first defined starting with the keyword '`sub`', and we see that the arguments are taken using the command '`shift`'. The subroutine is later called using identifer `&subre`.

You should test the program by making it executable (`chmod` command as earlier) and by running it without any arguments. You can also extend the program with more regular expressions and more DATA lines.

Submit: Submit the program `lab2-matching-data.pl` using the `submit-nlp` command.

Step 4: Counting words

Write a program called `lab2-word-counter.pl` with the following content:

```
#!/usr/bin/perl
# lab2-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 letter sequences, one after another, in the default variable `$_` as long as the matches are found on this line.

Make the program executable and run it using commands:

```
chmod u+x lab2-word-counter.pl
./lab2-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 `./lab2-word-counter.pl < some_file.txt`. The program will also work as you specify a file name as an argument, as in `./lab2-word-counter.pl some_file.txt`, thanks to the diamond operator. After the input ends, the program will print out the total number of words.

Submit: Submit the program `lab2-word-counter.pl` using the `submit-nlp` command.

Step 5: Simple Task 1

Write a program called `lab2-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
# lab2-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 `lab2-replace.pl` using the `submit-nlp` command.

Some String Functions

- Side note: `man perlfunc` gives a lot of information about different Perl functions
- **chomp** *string*; removes trailing newline from the string if it exists
- Like all predefined Perl functions, **chomp** can be used with parentheses as well, as in: **chomp**(*string*);
- **chomp;** applies `chomp` to the default variable (`$_`), like most other functions
- **length** *string*; string length
- **index***(str,substr[,offset])* returns position of the substring *substr* in the string *str*, starting from offset *offset*; if *offset* is not included, 0 is assumed; returns -1 if substring not found
- **substr***(str,begin[,len])* returns substring of string *str* starting from *begin*, with length *len*; if *len* is missing, returns to the end of string *str*

Some String Functions: sprintf

- **sprintf***(format, @arguments)* an elaborate function to create a string based on a given format with provided list of arguments; similar to the C function `printf`, more information provided in `man perlfunc`

Basic Input and Output (I/O) in Perl

Review: Standard Input and Standard Output

- Remember that *standard input* and *standard output* (and *standard error*) have a precise meaning in the Linux or Unix environment
- When a program reads *standard input* it reads keyboard by default
- When a program writes to *standard output* it prints to the screen terminal
- Redirection operators such as '<' and '>' can be used to redirect standard input from a file, or standard output to a file
- Redirection operators are used in the command line and do not depend on a programming language

Basic I/O in Perl

- We have seen basic "diamond" operator <> for reading input
- The diamond operator <> behaves in a special way:
  - if the program is not given arguments, the diamond operator reads the standard input
  - if the program is given arguments, the diamond operator treats the first argument as the file name, opens the file, and reads it; when finished, it will open the next file using the next argument as the file name
- 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 6: Count Number of Lines

- Write a program `lab2-line-count.pl`
- Usage: `./lab2-line-count.pl file.txt`
- Output: `file.txt has 124 lines`

- Remember to include a file header comment
- Submit `lab2-line-count.pl` using `nlp-submit`

Write a Perl program named `lab2-line-count.pl` which counts number of lines in a given file. The program is run using the line:

```
./lab2-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.

Of course, if we specify a file with a different file name; e.g., `f2.txt` having 250 lines, then the output of the command:
`./lab2-line-count.pl f2.txt`
should be:
`f2.txt has 250 lines`

Some hints: In order to run the program as `./lab2-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. A common way to open a file for readin in Perl, as we just saw is `open(my $fh, '<', $fname);`. We can read each line of the file by using the command `<$fh>` within a while loop in order to read to the end of 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.

Do not forget to include the file header comment, which should have the format as shown below:

```
#!/usr/bin/perl
# CSCI4152/6509 Fall 2024
# Program: lab2-line-count.pl
# Author: Vlado Keselj, B00123456, vlado@dnlp.ca
# Description: The program is a part of Lab2 required submissions.
```

You can test the program on the program file itself, or other files in the directory. The output should show the number of lines which is the same as when running the command '`wc`' (word count).

Submit: Submit the output file `lab2-line-count.pl` using the `submit-nlp` command.

Step 7: End of the Lab

- Make sure that you submitted all required files:
  `lab2-matching.pl`, `lab2-matching-data.pl`, `lab2-word-counter.pl`, `lab2-replace.pl`, `lab2-line-count.pl`
- End of the lab.