Faculty of Computer Science, Dalhousie University *15/17-Jan-2020*

CSCI 4152/6509 — Natural Language Processing

Lab 1: FCS Computing Environment, Perl Tutorial 1

Lab Instructor: Dijana Kosmajac, Tukai Pain
Location: rm1102, Mona Campbell building
Notes copyright: Vlado Keselj, Magdalena Jankowska

FCS Computing Environment

Slide notes:

Lab Overview

- An objective: Make sure that all students are familiar with their CSID and how to login to the `bluenose` server
- Refresh your memory about Unix-like Command-Line Interface
- Introduction to Perl
- Note 1: Replace *CSID* with your CSID (Dalhousie CS id, which is different from your Dalhousie id)
- Note 2: If you do not know your CSID, you can look it up and check its status at: `https://www.cs.dal.ca/csid`

An objective of this lab it to make sure that all students are familiar with the basic FCS (Faculty of Computer Science, Dalhousie University) computing environment. This means that you should know your CSID. This is a separate Computer Science user id, in addition to the the Dalhousie ID and Banner number. Among other things, the CSID is needed when logging in into the Computer Science servers and when using the GitLab repository.

Step 1: Logging in to the server `bluenose`

Step 1: Logging in to server bluenose

- You can choose Windows, Mac or Linux environment in some labs
- Windows: you will use `PuTTY` program
- On Mac: open a Terminal and type:
  `ssh` *CSID*`@bluenose.cs.dal.ca`
  (instead of *CSID* use your CS userid)
- On Linux: similarly to Mac, you open the terminal and type the same command:
  `ssh` *CSID*`@bluenose.cs.dal.ca`

Your first step is to login into the `bluenose` server. You can use a lab computer if there are any available in the lab, or your own laptop. We assume that you may be using a Linux, Mac, or Windows computer, and some basic instructions for each of these environments will be provided.

On Linux: In a Linux environment, you should open a terminal with a shell and type the following command to login to the bluenose server:

```
ssh CSID@bluenose.cs.dal.ca
```

where *CSID* is your CS userid. You will be prompted for your CSID password, after which you will be logged in into the bluenose server.

On Mac: The Mac OS is a Unix-like system, and similarly like in Linux, you can first start a Terminal. A way to find the 'Terminal' application is to click on the search image in th eupper right corner and type 'Terminal', and then find the `Terminal` application. Another way to find the Terminal application is to look into the Mac applications folder. Once you open the terminal, you can login to the `bluenose` server by typing:

```
ssh CSID@bluenose.cs.dal.ca
```

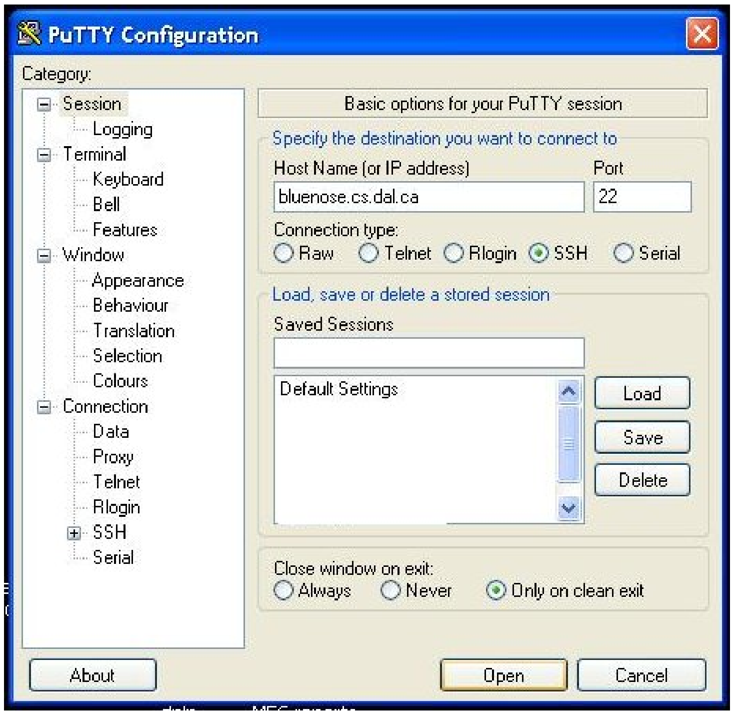
where *CSID* is your CS userid.

On Windows: If you use a Windows environment, you will need to use a program named `PuTTY` to login to the `bluenose.cs.dal.ca` server. There are other alternative programs that can be used for ssh login, such as MobaXterm, and any of them could be used as well.

If you do not have PuTTY installed on your computer, you can easily install it. It is a well-known and freely available program, avalable at
`http://www.chiark.greenend.org.uk/~sgtatham/putty`

Running PuTTY

- If you use Windows, then one option is to use PuTTY to login to the `bluenose` server
- Double-click the PuTTY icon, and the following window should appear:



Host Name (or IP address): bluenose.cs.dal.ca
Port: 22
Connection type: Raw Telnet Rlogin [✓ SSH] Serial
Close window on exit: Always Never [✓ Only on clean exit]

You should fill in the basic information: `bluenose.cs.dal.ca` for the Host Name. Make sure that the port number is 22; i.e., Connection type is SSH. You click 'Open' and the login process should start. You are likely to receive a warning about an unknown host key. Normally, this is something that you should be careful about and try to make sure that the offered fingerprint matches the fingerprint of the server, but in a relatively secure network you can accept this connection. Once accepted, the host key is stored with PuTTY and this warning should not appear again.

Review of Some Linux Commands

We will now briefly review some Linux command-line commands that you can use on the `bluenose` server. If you already know Linux or other type of Unix-like system, you should be familiar with the following commands and you can quickly try them out. Otherwise, these commands are a good start into learning more about basics of using a Linux file system.

Step 2: pwd. Immediately after login, your current directory will be a directory known as your home directory. The program that you are communicating with in a terminal after logging in is called shell (known also as 'command line' in some operating systems). Our current directory is associated with our currently executing shell. To print our current working directory we can use the command 'pwd'. Enter the command:

```
  pwd
```

to show the path name of your current working directory. It is a good idea to frequently use this command to verify that your current working directory is the one you think it is, or to find out what is your current working directory.

If you are not familiar with the Unix (Linux) command line, you should read about it: there are a lot of resources on the web. A useful way to learn more about a command is to use command 'man'. For example, if you type:

```
  man pwd
```

you can read about the command 'pwd'. The command 'man' stands for *manual pages.* You can exit the 'man pwd' command by pressing the key 'q'.

Step 3: mkdir, ls, chmod. Depending on which course number you registered for (CSCI4152 or CSCI6509) you should create a directory named csci4152 or csci6509. To use an example, we will show the commands with the 6509 number, but you should use the appropriate number. First you can create a directory for the course work using the following command:

```
mkdir csci6509
```

or

```
mkdir csci4152
```

depending on the course number.

Enter the command:

```
ls
```

to display the files (including subdirectories) in your working directory. Do you see the name of the directory you just created?

Now, enter the command:

```
chmod go-rx csci6509
```

or

```
chmod go-rx csci4152
```

This command will prevent other users from entering and seeing the content of this directory.

Step 4: lab1 directory. Enter the command:

```
cd csci6509
```

or

```
cd csci4152
```

to change your current working directory to to the directory you created in the previous step. Use the `pwd` command to verify that you performed this task successfully.

Create a directory with the name `lab1` in your current working directory, and change the current working directory to the directory `lab1`. Verify that you have performed these tasks successfully. If you type the command `pwd` it should display the following path:

/users/cs/*CSID*/csci4152/lab1

or

/users/cs/*CSID*/csci6509/lab1

The part "`/users/cs/`" may vary in case that there are some changes on the server, but the part "***CSID***`/csci6509/lab1`" (or "***CSID***`/csci4152/lab1`") should be exactly as shown.

Step 5: Prepare `hello.pl` in emacs or some other editor. Now we are going to write a simple Perl program and run it. This will be a simple "Hello world!" program, which prints a line saying "`Hello world!`".

The first step is to write the source code.

We can use `emacs` editor for this task. If you are more comfortable with some other editor, you can use it. For example, the editors `pico` and `nano` are user-friendly, but less suitable for serious programming as they are less powerful in terms of functionality. The editors `vi` and `vim` are quite powerful, but with possibly a bit steeper learning curve. In this course, we will provide most help with emacs.

A file with filename `filename` is opened in the editor emacs using command '`emacs filename`' or in the editor pico using the command '`pico filename`'.

If you are not familiar with emacs, it should be relatively easy to start using it. A few basic instructions will be explained below, and there are many resources on the Internet that can be used. One available resource on the Internet is the following tutorial. It is very useful, although somewhat long:
`http://www2.lib.uchicago.edu/keith/tcl-course/emacs-tutorial.html`
You can start reading it and stop once you are comfortable using emacs to edit a small program.

Another way that you can learn Emacs is to simply type `emacs` in your PuTTY terminal. This will start the editor, with an initial 'splash' screen explaining basic commands. You can also notice in the 'splash screen' that by pressing `C-h t` ; i.e,., Control-h and then 't', you can start a tutorial, and this is also a way to learn emacs.

The emacs documentation uses notation `C-x` for `Ctrl-x`, i.e., Control-x key. Additional special way for pressing keys is `M-x`, which stands for Meta-x. Some keyboards have a special Meta key which is used, but more commonly this is obtained by pressing the Alt key and 'x' in the same time, or by pressing Esc and 'x' one after another (not in the same time!).

Here are the most important emacs commands:

– to start editing a (possibly new) file in emacs: type in the terminal
  `emacs filename`
  where `filename` is the name of the file
– to exit emacs: type `C-x C-c` sequence (remember, this means Control-x Control-c).
– to save the current file: type `C-x C-s`

Now, we can go back to editing the Perl program, which we will call `hello.pl` and which should print string "`Hello world!`" on a line by itself. You can start by typing:
`emacs hello.pl`
in the shell command line. Now, in the emacs window, type in the following program:

```
#!/usr/bin/perl

print "Hello world!\n";
```

After finishing the program, you can exit emacs using `C-x C-c`. You will need to confirm that you want to save the file by pressing 'y' on the emacs question "Save file…".

If you want to save the file without exiting, use `C-x C-s`.

Also, if you do not see font highlighting, you may want to try enabling that by typing `M-x global-font-lock-mode` (remember, `M-x` is produced with key combination Alt+x or Esc x one after another). This can be set as default by adding the following line to the emacs initialization file `.emacs` in your home directory:
`(global-font-lock-mode 1)`

Step 6: Running a Perl program. Exit emacs and enter:
`perl hello.pl`
to run you program. The program should produce one line of output: ‘`Hello world!`”.

To run the program in this way, we did not even need the first line of the program (`#!/usr/bin/perl`); but this line is important in the second way to run the program, as follows. First, we change the file permissions allowing the user to execute the program:
`chmod u+x hello.pl`
Then, we run the program in the following way:
`./hello.pl`

Perl Tutorial 1

Perl Tutorial

– Over next couple of labs we will go over a basic Perl tutorial
– Learn basics of Perl programming language
– You already wrote and ran a simple Perl program `hello.pl`

Finding Help

– Web: `perl.com`, `CPAN.org`, `perlmonks.org`, ...
– `man perl`, `man perlintro`, ...
– books: the “Camel” book:
“Learning Perl, 4th Edition” by Brian D. Foy; Tom Phoenix; Randal L. Schwartz (2005)
Available on-line on Safari at Dalhousie

Step 7: Basic Interaction with Perl

– You can check the Perl version on bluenosee by running ‘`perl -v`’ command:
```
perl -v
This is perl 5, version 16, subversion 3 (v5.16.3)...
```
– If you use the official Perl documentation from `perl.com` documentation site, choose the right version.
– Test your assignment programs on bluenose if you developed them somewhere else.

Executing Command from Command-Line

– You can execute Perl commands directly from the command line
– Example, type:
```
perl -e 'print "hello world\n"'
```
– and the output should be: `hello world`
– A more common way is to write programs in a file

Write Program in a File

As we saw already, the most common way is to write Perl programs in files and execute them from there. By this time you should have the following program named `hello.pl` already created in the directory:

```
#!/usr/bin/perl

print "hello world\n";
```

Once you save the file you can run it using the command:
```
perl hello.pl
```
You can also run it directly. First you need to set the executable permission of the file with:
```
  chmod u+x hello.pl
```
and then you can run it using:
```
  ./hello.pl
```

Submit: You need to submit the program `hello.pl` to be marked. You can submit the program by running the following command on bluenose:

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

You will be prompted to enter your CSID and password.

Direct Interaction with an Interpreter

Interacting with an interpreter program in Perl is not common, but it is available if you want to try the immediate effect of some Perl commands. To run interpreter, we actually need to run the perl debugger using the command-line command:
```
perl -d -e 1
```
As mentioned in class, Perl interpreter normally operates by reading the whole program, compiling it just in time for execution into an intermediate form, and executing this intermediate form. If we want to test Perl execution command by command, one option is to run the Perl debugger using the command '`perl -d -e 1`'.

When you run it, you should get an output as follows:

```
Loading DB routines from perl5db.pl version 1.28
Editor support available.

Enter h or 'h h' for help, or 'man perldebug' for more help.

main::(-e:1):1
  DB<1>
```

Now you can enter a couple Perl commands to see them executed. For example, you can try:

```
print "hello\n";
print 12*12;
```

You can enter '`q`' to exit the debugger. If you would like to learn more about the Perl debugger, you can start with entering command '`h`' in debugger, for help, and you can read even more by using the command '`man perldebug`'.

Syntactic Elements of Perl

- statements separated by semi-colon '`;`'
- white space does not matter except in strings
- line comments begin with '`#`'; e.g.
  ```
  # a comment until the end of line
  ```

- variable names start with $, @, or %:
  `$a` — a scalar variable
  `@a` — an array variable
  `%a` — an associative array (or hash)
  However: `$a[5]` is 5th element of an array `@a`, and
  `$a{5}` is a value associated with key 5 in hash `%a`
- the starting special symbol is followed either by a name (e.g., `$varname`) or a non-letter symbol (e.g., `$!`)
- user-defined subroutines are usually prefixed with &:
  `&a` — call the subroutine `a` (procedure, function)

Step 8: Example Program 2

- Enter the following program as `example2.pl`:
```
#!/usr/bin/perl
use warnings;

print "What is your name? ";
$name = <>;
chomp $name;
print "Hello $name!\n";
```
- '`use warnings;`' enables warnings — recommended!
- `chomp` removes the trailing newline from `$name` if there is one. However, changing the special variable `$/` will change the behaviour of `chomp` too.
- Test `example2.pl` and you will need to submit it later

Prepare the program `example2.pl` and save it. You can use the `emacs` editor to do this. You should test it to see that it works. Remember to make it first user executable and then run it using the commands:

```
chmod u+x example2.pl
./example2.pl
```

Submit: Submit the program '`example2.pl`' using: `~vlado/public/submit-nlp`

Example 3: Declaring Variables

The declaration "`use strict;`" is useful to force more strict verification of the code. If it is used in the previous program, Perl will complain about variable `$name` not being declared, so you can declare it with: '`my $name`' We can call this program `example3.pl`:

```
#!/usr/bin/perl
use warnings;
use strict;

my $name;
print "What is your name? ";
$name = <>;
chomp $name;
print "Hello $name!\n";
```

You do not have to submit this program, but if you can try it and run it.

Example 4: Declare a variable and assign its value in the same line

```
#!/usr/bin/perl
use warnings;
use strict;

print "What is your name? ";
my $name = <>;
chomp $name;
print "Hello $name!\n";
```

Step 9: Example 5: Copy standard input to standard output

We can call this program `example5.pl`

```
#!/usr/bin/perl
use warnings;
use strict;

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

The operator <> reads a line from standard input, or—if the Perl script is called with filenames as arguments—from the files given as arguments.

Try different ways of running this program:

– Reading from standard input, which by default is the keyboard:
  `./example5.pl`
  In this case the program will read the lines introduced from the keyboard until it receives the `Ctrl-D` combination of keys, which ends the input.
– Reading the content of files, whose names are given as arguments of the script
  Create two simple text documents `a.txt` `b.txt` with a few arbitrary lines each (you can use a text editor to do that).
  Then run the Perl script with the names of these files are arguments:
  `./example5.pl  a.txt b.txt`

Submit: Submit the program '`example5.pl`' using: `~vlado/public/submit-nlp`

Example 6: Default variable

Special variable `$_` is the default variable for many commands, including `print` and expression `while (<>)`, so another version of the program `example5.pl` would be:

```
#!/usr/bin/perl
while (<>) { print }
```

This is equivalent to:

```
#!/usr/bin/perl
while ($_ = <>) { print $_ }
```

Even shorter version of the program would be:

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

Variables

- no need to declare them unless "`use strict;`" is in place
- `use strict;` is a good practice for larger projects
- variable type is not declared (it is inferred from context)
- the main variable types:
  1. Scalars
     - numbers (integers and floating-point)
     - strings
     - references (pointers)
  2. Arrays of scalars
  3. Hashes (associative arrays) of scalars

Single-Quoted String Literals

```
  print 'hello\n';           # produces 'hello\n'
  print 'It is 5 o\'clock!'; # ' has to be escaped
  print q(another way of 'single-quoting');
                       # no need to escape this time
  print q< and another way >;
  print q{ and another way };
  print q[ and another way ];
  print q- and another way with almost
           arbitrary character (e.g. not q)-;
  print 'A multi line
       string (embedded new-line characters)';
  print <<'EOT';
Some lines of text
  and more $a @b
EOT
```

Double-Quoted String Literals

```
print "Backslash combinations are interpreted in
       double-quoted strings.\n";
print "newline after this\n";
$a = 'are';
print "variables $a interpolated in double-quoted
       strings\n";
# produces "variables are interpolated" etc.
```

```
@a = ('arrays', 'too');
print "and @a\n";
# produces "and arrays too" and a newline

print qq{Similarly to single-quoted, this is also
         a double-quoted string, (etc.)};
```

Scalar Variables

- name starts with `$` followed by:
  1. a letter and a sequence of letters, digits or underscores, or
  2. a special character such as punctuation or digit
- contains a single scalar value such as a number, string, or reference (a pointer)
- do not need to worry whether a number is actually a number or string representation of a number

```
$a = 5.5;
$b = " $a ";
print $a+$b;
```

(11)

Numerical Operators

- basic operations: `+ - * /`
- transparent conversion between int and float
- additional operators:
  `**` (exponentiation), `%` (modulo), `++` and `--` (post/pre inc/decrement, like in C/C++, Java)
- can be combined into assignment operators:
  `+= -= /= *= %= **=`

String Operators

- `.` is concatenation; e.g., `$a.$b`
- `x` is string repetition operator; e.g.,
  `print "This sentence goes on"." and on" x 4;`
  produces:

```
This sentence goes on and on and on and on and on
```

- assignment operators:
  `=  .=  x=`
- string find and extract functions: `index(str,substr[,offset])`, and `substr(str,offset[,len])`

Comparison operators

```
Operation                Numeric  String
--------------------------------------------
less than                  <         lt
less than or equal to     <=         le
greater than                >        gt
greater than or equal to  >=        ge
equal to                  ==        eq
not equal to              !=        ne
compare                   <=>       cmp
---------------------------------------------
```

Example:

```
print ">".(1==1)."<";  # produces: >1<
print ">".(1==0)."<";  # produces: ><
```

Remember: Operators cause conversions between numbers and strings

Example:

```
my $x=12;

print $x+$x;  #produces 24
print $x.$x;  #produces 1212

print ">".($x > 4)."<";  # produces: >1<
print ">".($x gt 4)."<";  # produces: ><
```

Step 10: Simple task 1

Create a Perl script named `task1.pl` that prints to the standard output 20 of the following lines:

```
Use \n for a new line.
```

The number 20 should be defined as a variable within the script.

Submit: Submit the program ‘`task1.pl`’ using: `~vlado/public/submit-nlp`

What is true and what is false — Beware

```
print ''      ?'true':'false'; #false
print 1       ?'true':'false'; #true
print '1'     ?'true':'false'; #true
print 0       ?'true':'false'; #false
print '0'     ?'true':'false'; #false
print ' 0'    ?'true':'false'; #true
print 0.0     ?'true':'false'; #false
print "0.0"   ?'true':'false'; #true
print 'true'  ?'true':'false'; #true
print 'zero'  ?'true':'false'; #true
```

The false values are: `0`, `''`, `'0'`, or `undef`
True is anything else.

<=> and cmp

`$a <=> $b` and `$a cmp $b` return the sign of `$a - $b` in a sense:

-1	if `$a < $b`	or `$a lt $b`,
0	if `$a == $b`	or `$a eq $b`, and
1	if `$a > $b`	or `$a gt $b`.

Useful with the `sort` command

```
@a = ('123', '19', '124');
@a = sort @a;              print "@a\n"; # 123 124 19
@a = sort {$a<=>$b}   @a; print "@a\n"; # 19 123 124
@a = sort {$b<=>$a}   @a; print "@a\n"; # 124 123 19
@a = sort {$a cmp $b} @a; print "@a\n"; # 123 124 19
@a = sort {$b cmp $a} @a; print "@a\n"; # 19 124 123
```

Boolean Operators

```
Six operators:  &&  and
                ||  or
                !   not
```

Difference between `&&` and `and` operators is in precedence: `&&` has a high precedence, `and` has a very low precedence, lower than =, `,` Similarly for others

```
$x = $a || $b;    #better construction
$x = ($a or $b); #requires parenthesis
```

Can be used for flow control (short-circuit) - for this purpose `or` is better than `||`

```
some_func $a1, $a2 or die "some_func returned false:$!";
some_func($a1, $a2) ||
         die "some_func returned false:$!";
```

Range Operators

```
..   - creates a list in list context,
       flip-flop otherwise
...  - same, except for flip-flop behaviour

@a = 1..10;   print "@a\n"; # out: 1 2 3...
```

Control Structures

- Unconditional jump: `goto`
- Conditional:
    - `if-elsif-else` and `unless`

- Loops:
  - while loop
  - for loop
  - foreach loop
- Restart loop: ‘next’ and ‘redo’
- Breaking loop: ‘last’

If-elsif-else

```
if (EXPRESSION) {
  STATEMENTS;
} elsif (EXPRESSION1) {  # optional
  STATEMENTS;
} elsif (EXPRESSION2) {  # optional additional elsif’s
  STATEMENTS;
} else {
  STATEMENTS;  # optional else
}
```

Other equivalent forms, e.g.:

```
if ($x > $y) { $a = $x }
$a = $x if $x > $y;
$a = $x unless $x <= $y;
unless ($x <= $y) { $a = $x }
```

While Loop

```
while (EXPRESSION) {
  STATEMENTS;
}
```

- `last` is used to break the loop (like `break` in C/C++/Java)
- `next` is used to start next iteration (like `continue`)
- `redo` is similar to next, except that the loop condition is not evaluated
- labels are used to break from non-innermost loop, e.g.:

```
L:
while (EXPRESSION) {
   ... while (E1) { ...
         last L;
}  }
```

next vs. redo

```
#!/usr/bin/perl

$i=0;
while (++$i < 5) {
    print "($i) "; ++$i;
    next if $i==2;
    print "$i ";
} # output: (1) (3) 4

$i=0;
while (++$i < 5) {
    print "($i) "; ++$i;
    redo if $i==2;
    print "$i ";
} # output: (1) (2) 3 (4) 5
```

For Loop

```
for ( INIT_EXPR; COND_EXPR; LOOP_EXPR ) {
   STATEMENTS;
}
```

Example:

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

Foreach Loop

Examples:

```
@a = ( 'lion', 'zebra', 'giraffe' );
foreach my $a (@a) { print "$a is an animal\n" }

# or use default variable
foreach (@a) { print "$_ is an animal\n" }

# more examples
foreach my $a (@a, 'horse') { print "$a is animal\n"}

foreach (1..50) { print "$_, " }
```

`for` can be used instead of `foreach` as a synonym.

Subroutines

```
sub say_hi {
    print "Hello\n";
}


&say_hi();  # call
&say_hi;    # call, another way since we have no params
say_hi;     # works as well
            # (no variable sign = sub, i.e., &)
```

Subroutines: Passing Parameters

When a subroutine is called with parameters, a parameter array `@_` within the subroutine stores the parameters.

The parameters can be accessed as `$_[0]`, `$_[1]` but it is not recommended:

```
sub add2 { return $_[0] + $_[1] } #not recommended

print &add2(2,5);  # produces 7
```

Subroutines: Passing Parameters (2)

Recommended: copy parameters from `@_` to local variables:

– using shift to get and remove elements from the array `@_`
  With no arguments, shift within a subroutine takes `@_` by default (outside of a subroutine, shift with no arguments takes by default the array of parameters of a script `@ARGV`)
```
sub add2 {
    my $a = shift;
    my $b = shift;
    return $a + $b;
}
```
– or copy the whole `@_` array
```
sub add2 {
    my ($a, $b) = @_;
    return $a + $b; }
```

Subroutines: Passing Parameters (3)

You can define a subroutine that will work with variable number of parameters.

Example:

```
sub add {
  my $ret = 0;
  while (@_) { $ret += shift }
  return $ret;
}
print &add(1..10); # produces 55
```

Step 11: Simple task 2

Create a Perl script named `task2.pl` that defines a subroutine `conc`. The subroutine takes two parameters and returns a string that is the concatenation of the two parameters, but such that the two input parameters are ordered alphabetically in the resulting string, i.e., the input parameter that is first in the alphabetical order appears first in the output string of the joined parameters. For example, when the subroutine `conc` is called as `conc('ccc','aaa')` as well as when it is called as `conc('aaa', 'ccc',)`, in both cases it should return the string `aaaccc`

In addition to the subroutine the script should include the following four lines for testing the subroutine:

```
print &conc('aaa','ccc');
print "\n";
print &conc('ccc','aaa');
print "\n";
```

Submit: Submit the program '`task2.pl`' using: `~vlado/public/submit-nlp`