CSCI 4152/6509
Natural Language Processing

Lab 1:

FCS Computing Environment, Perl Tutorial 1

Lab Instructor: Aditya Joshi and Tymon Wranik-Lohrenz

Faculty of Computer Science

Dalhousie University

Lab Overview

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

Lab Evaluation

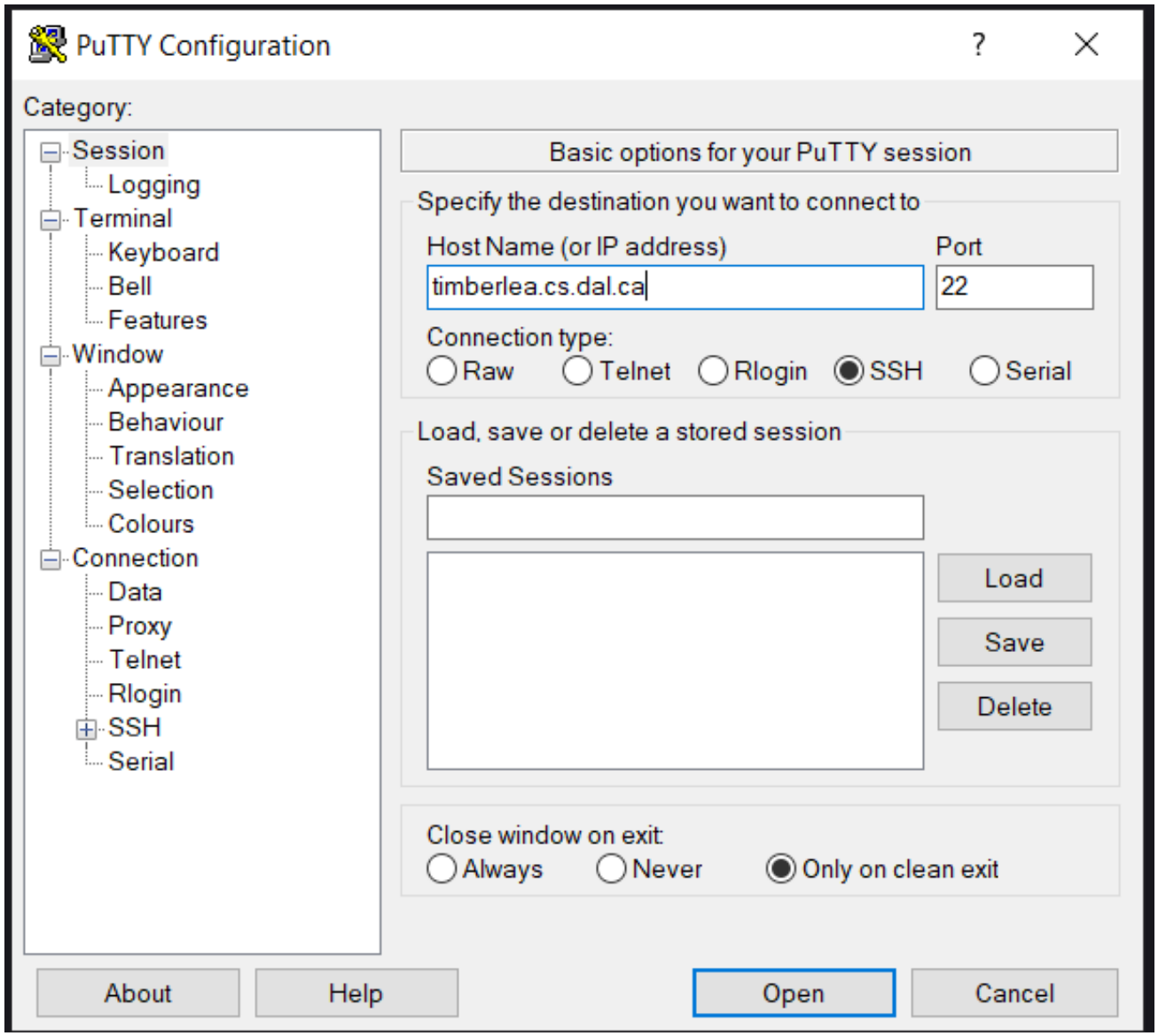
- The lab will be evaluated as a part of the Assignment 1 (A1) with the same submission deadline as the assignment, which will be at least one week after the lab.
- Files to be submitted by the end of the lab are:
  1. `hello.pl`
  2. `lab1-example2.pl`
  3. `lab1-example5.pl`
  4. `lab1-task1.pl`
  5. `lab1-task2.pl`

Step 1: Logging in to server timberlea

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

Running PuTTY

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


PuTTY Configuration
Category:
Session
Logging
Terminal
Keyboard
Bell
Features
Window
Appearance
Behaviour
Translation
Selection
Colours
Connection
Data
Proxy
Telnet
Rlogin
SSH
Serial
Basic options for your PuTTY session
Specify the destination you want to connect to
Host Name (or IP address)
Port
timberlea.cs.dal.ca
22
Connection type:
Raw
Telnet
Rlogin
SSH
Serial
Load, save or delete a stored session
Saved Sessions
Load
Save
Delete
Close window on exit:
Always
Never
Only on clean exit
About
Help
Open
Cancel

Fingerprints of the Current timberlea Keys

```
ED25519   SHA256:/myX/hSyWyz/q/l4P9mxPAcwhfGr2gpEoh743dA0uJM
ECDSA     SHA256:9ruRg3IIm01a54gXqUDJHu+Ss34c57tA5a3encCp4qM
RSA       SHA256:X92KGKagx2NZ2L18ew4dEHqXPu2CEyeYsk2fb7JD99Q
```

- If you want to compare with the fingerprints that you obtain.
- We will go over a review of some Linux commands next.

Review of Some Linux Commands

Step 2: `pwd` — print current working directory
`man pwd` — print `pwd` documentation

Step 3: `mkdir csci4152` or `mkdir csci6509`
`ls` — list directory contents

Make sure others cannot read or enter your directory:

```
chmod go-rx csci6509
```

or

```
chmod go-rx csci4152
```

Step 4: `cd csci6509` or `cd csci4152`

- Make directory `lab1` and change your current directory to it.

Note about editors: instructions for editor `emacs` are provided

- other editors are available on `timberlea` (pico, nano, vi, vim)
- options on using editors from your computer (vscode, sublime)
- options to edit file on your computer and copy to `timberlea`

Step 5: Using emacs prepare `hello.pl`

Use an editor: `emacs` or some other (e.g., `vi`, `pico` ...)

```
emacs hello.pl
```

Prepare the following program:

`hello.pl`

```
#!/usr/bin/perl

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

Step 6: Running a Perl program

```
perl hello.pl
```

Another way to run the program:

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

This ends a brief introduction into the FCS server environment.

Perl Tutorial

- Next few labs will go over a basic Perl tutorial
- Perl is a useful programming language for string-based text processing
- More details about Perl connection with NLP is covered in lectures
- We will not cover more advanced features, such as Object-Oriented style in Perl
- You already wrote and ran a simple Perl program `hello.pl`

Finding More Help about Perl

- From Unix-style command line (e.g., timberlea):
  `man perl`, `man perlintro`, ...
- Many Web resources: `perl.com`, `CPAN.org`, `perlmonks.org`, ...
- Books: e.g., the "Camel" book:
  "Learning Perl" by Brian D. Foy; Tom Phoenix; Randal L. Schwartz (latest seems to be 8th edition, Aug 2021)

  or "Beginning Perl" by Simon Cozens
  `https://www.perl.org/books/beginning-perl`

Step 7: Basic Interaction with Perl

- You can check the Perl version on timberlea by running '`perl -v`' command; e.g.:

```
perl -v
```

```
This is perl 5, version 36, subversion ...
```

- If you use the official Perl documentation from `perl.com` documentation site, choose the right version.
- Test your assignment programs on `timberlea` 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: Example 1

- `hello.pl` should be already in the directory
- Run the program using: `perl hello.pl`
- You can also run it directly
- First, make it executable:

```
chmod u+x hello.pl
```

and then you can run it using:

```
./hello.pl
```

- Submit the program `hello.pl` using the command `submit-nlp` as described in notes

Direct Interaction with an Interpreter

- Not common to use, but available
- Command: `perl -d -e 1`
- Enter Perl statements, for example:

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

- To learn more about debugger: command '`h`'
- Enter '`q`' to exit debugger
- Learning more from the command line:

```
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 `lab1-example2.pl`:

lab1-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 `lab1-example2.pl` and submit it

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, and refuse to run the program. You can declare it with: '`my $name`'
We can call this program `lab1-example3.pl`:

`lab1-example3.pl`

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

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

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

You can declare a varible and assign it an initial value in the same line. The previous program would look like this with that modification:

```
#!/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

- This program named `lab1-example5.pl` reads the standard input and prints it to standard output

`lab1-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:

  `./lab1-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:

  ```
  ./lab1-example5.pl  a.txt b.txt
  ```

Submit: Submit the program '`lab1-example5.pl`' using:
`submit-nlp`

Some General Perl Reference Information

- We will go now over some general Perl information
- Most of the material is just for reference, but there are still some steps that you need to finish for this Lab.

  We will go over language elements such as:
- Variables
- Strings (String Literals)
- Operators and Expressions
- (Complete Step 10)
- Control Structures
- Subroutines (Functions, Procedures)
- (Complete Step 11)

Variables: Summary

- Variable names with sigils:
  - `$a` — scalar,
  - `@b` — array,
  - `%c` — hash
- Variable declarations:
  ```
  my $a = 1;
  ```
- Variable declarations not required by default
- `use strict;` requires variable declarations
- `use strict;` is a good idea for larger projects
- Arbitrary identifies can be used for variable names, as in:
  `$count @pages12a %phone_book_1`
- Variable name can be a sigil and a special character, and such variables are usually special, such as:
  `$_` — the *default* variable

Example 6: Default variable

Special variable `$_` is the default variable for many commands, including `print` and expression `while (<>)`, so another version of the program `lab1-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
```

Variable Types

- The main variable types:
  1. Scalars
     - numbers (integers and floating-point)
     - strings
     - references (similar to pointers)
  2. Arrays of scalars
  3. Hashes (associative arrays) of scalars
- Difference between numbers, strings, and references is not declared but inferred from context

Scalar Variables

- Variable 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
- Scalar variable contains a single scalar value such as a number, string, or reference (a pointer)
- We generally do not worry whether a number is integer, float, or string containing a number; e.g.:

```
$a = 5.5;           # $a contains a float number 5.5
$b = " $a ";        # $b is a string " 5.5 "
print $a+$b;        # $b converted to number
                    # output: 11
```

String Literals

- We will over over string literals (i.e., quoted strings)
- String literals:
  - single-quoted strings preserve strings as they are almost always (except `\'`)
  - double-quoted strings replace (interpolate) characters like `\n` (newline) and variable values like `" $a "`
  - back-quoted strings are an advanced feature to execute a system command and use output in a string; e.g., `` `ls` ``
- We will not look into back-quoted strings now
- A string can span multiple lines

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.)};
```

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;
print $x.$x;

print ">".($x > 4)."<";
print ">".($x gt 4)."<";
```

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 `lab1-task1.pl` that prints to the standard output 20 times the following line:

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

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

File Header Comment

- Since this is the first program that you created and not only copied, you should start to use required file header comment in the following format:

```
#!/usr/bin/perl
# CSCI4152/6509 Fall 2024
# Program: lab1-task1.pl
# Author: Vlado Keselj, B00123456, vlado@dnlp.ca
# Description: The program is a part of Lab1 required submissions.
```

- Use your name, Banner number, and email.
- You can copy the same description.
- You can use course number for which you are registered.
- Your code should follow this comment.

Submit: Submit the program '`lab1-task1.pl`' using: `submit-nlp`

What is true and what is false — Beware

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

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,
```

For example:

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

- The range operator is quite convenient in the list context.

- If the range operator is used in a scalar context, it behaves as a so-called flip-flop boolean variable. It is a more complex and advanced feature that we will not cover.

- Perl also has `...` as a range operator, which differs from `..` only in a scalar context.

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 `lab1-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.

- E.g., `conc('ccc','aaa')` and `conc('aaa', 'ccc')` should both return: `aaaccc`
- Add the following lines to the script:

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

- **Remember to add a file header comment**
- Test the program `test2.pl` and submit it

This is the end of Lab 1.