Faculty of Computer Science, Dalhousie University *12/13-Sep-2023*

CSCI 4152/6509 — Natural Language Processing

Lab 1: FCS Computing Environment, Perl Tutorial 1

Instructor: Vlado Keselj
Location: Goldberg CS 134(u)/CS 143(g)
Notes authors: 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 `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)

The main learning objectives of this lab are:

- Make sure that all students are familiar with their CSID and how to login to the `timberlea` server
- Refresh your memory about some basic Unix-like (or Linux) commands, also known as the Unix-like Command-Line Interface (CLI) or shell
- Learn basics of the Perl programming language

The first objective is to make sure that all students are familiar with the basic FCS (Faculty of Computer Science, Dalhousie University) computing environment. As a start, 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.

Note 1: If you do not know your CSID, you can look it up and check its status at the following URL: `https://csid.cs.dal.ca`

Note 2: Once you learn your CSID, use it in the rest of the lab whenever you see the string: `<your_csid>`

Slide notes:

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`

The lab will be evaluated as a part of an assignment (Assignment 1), which will be posted in full later. The lab submission deadline will be the same as the assignment deadline, and it will be at least one week after the lab scheduled time.

Step 1: Logging in to the server `timberlea`

The first step is to login to the server `timberlea` using the SSH (Secure SHell) protocol. The server `timberlea` (`timberlea.cs.dal.ca`) is the main Linux server used by FCS at Dal to support teaching in general and particularly undergraduate teaching. All students and faculty have accounts on this server with your CSID name and password. The following is a short summary of some options that use can to login to the server depending on your computer and operating system:

- 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 <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`

In this step we assume that you may be using a Linux, Mac, or Windows computer (desktop or laptop), and we will provide some basic instructions for each of these environments below.

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

```
ssh <your_csid>@timberlea.cs.dal.ca
```

where `<your_csid>` is your CS userid. You will be prompted for your CSID password, after typing it you should be logged in into the server assuming that your password was correct.

Important Note on Entering Commands: It is important that you type the commands as given in the boxes like the one above. **DO NOT copy and paste them from the PDF document,** because you may paste some incorrect characters and the command will not work properly.

Note about unknown keys: It is possible that you will get a warning the first time you login that the public key of the server `timberlea` is not known and whether you want to accept it. The message should be like this:

```
The authenticity of host 'timberlea.cs.dal.ca (129.173.22.43)' can't be established.
ED25519 key fingerprint is SHA256:/myX/hSyWyz/q/l4P9mxPAcwhfGr2gpEoh743dA0uJM.
This key is not known by any other names
Are you sure you want to continue connecting (yes/no)?
```

You should accept it. The warning is given in an unlikely chance of a man-in-the-middle attach. If you want to be really careful, you can compare the fingerprints above to the fingerprints you got, and if they match, you should be safe. They should match if the same key (ED25519) is reported, but it is possible that your `ssh` program chooses a different key. Other two possible keys and their fingerprints of the `timberlea` server are:

```
ECDSA ... SHA256:9ruRg3IIm01a54gXqUDJHu+Ss34c57tA5a3encCp4qM
RSA   ... SHA256:X92KGKagx2NZ2L18ew4dEHqXPu2CEyeYsk2fb7JD99Q
```

(If you are interested in checking these fingerprints yourself, you can use the command `ssh-keygen -lf` on the `.pub` files in the directory `/etc/ssh/` on `timberlea`.)

On Mac: The Mac OS is a Unix-like system, and similarly to Linux, you can first start a Terminal. A way to find the 'Terminal' application is to click on the search image in the upper 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 `timberlea` server by typing:

```
ssh <your_csid>@timberlea.cs.dal.ca
```

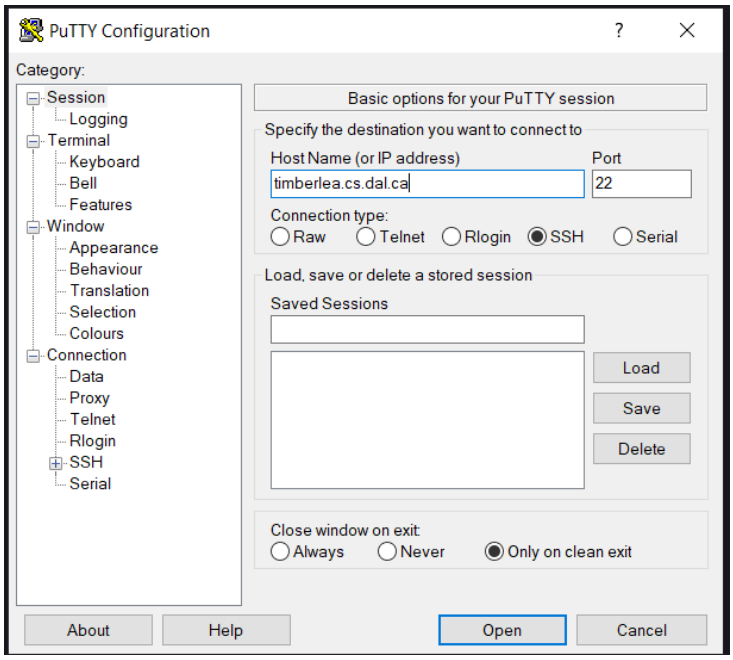
where `<your_csid>` is your CS userid.

On Windows: If you use a Windows environment, you will need to use a program named `PuTTY` to login to the `timberlea.cs.dal.ca` server. There are other alternative programs that can be used for ssh login, such as MobaXterm, WinSSH, and FileZilla, and any of them could be used as well. They are generally newer than PuTTY and have richer functionality.

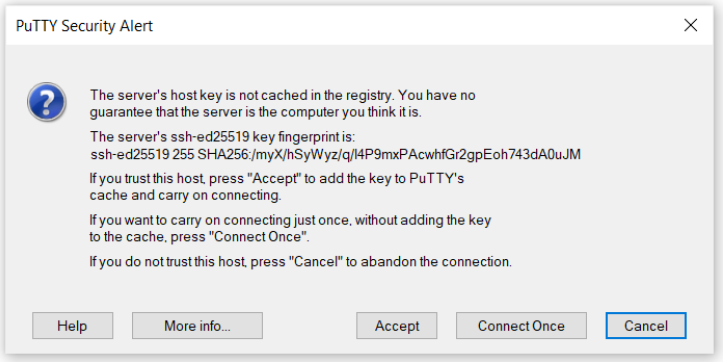
If you do not have PuTTY installed on your computer, you can easily install it. It is a well-known and freely available program, available at
`http://www.chiark.greenend.org.uk/~sgtatham/putty`
The same link is available from the page at `https://www.putty.org`.

Running PuTTY

– Double-click the PuTTY icon, and the following window should appear:



You should fill in the basic information: `timberlea.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. This is an equivalent warning mentioned earlier in the case of using `ssh` program with Linux. 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. This initial warning should look like this:



If the same key is chosen, then the fingerprint should match the fingerprint shown in the image.

As mentioned before, the valid fingerprints of the keys of the server `timberlea` as of September 2022 are:

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

Review of Some Linux Commands

We will now briefly review some Linux command-line commands that you can use on the `timberlea` 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. After your login, the server will print out some messages to your terminal, and it will print a short prompt string expecting your commands. After you type a command and press enter it will get executed on the server, you may get some output, and the server will wait for your next command. In this process, you are communicating with a program on the server called *shell* (also known as the "command line"). It is important to be aware that the shell is always associated with a directory in the file system of the server, and the commands will read and write to files usually depending on this directory, which is also known as a folder. We will call this your *current working directory*, and after your login your initial current directory is always your home directory.

In the labs in this course, it is frequently important to know what is your current working directory, and you can always use the command 'pwd' to see this directory. The command stands for pwd—print working directory. 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 the 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. 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. If you are not familiar with the command 'chmod' and file permissions in Unix-like systems, you should read more about them. For example, 'man chmod' command gives a manual page about the 'chmod' function, or there are many useful pages on the Web discussing them.

Step 4: lab1 directory. Enter the command:

```
cd csci6509
```

or

```
cd csci4152
```

to change your current working directory 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 path that looks as follows (with possible differences at the beginning):

`/users/cs/<your_csid>/csci4152/lab1`

or

`/users/cs/<your_csid>/csci6509/lab1`

The part "`/users/cs/`" may vary in case that there are some changes on the server, but the part "`<your_csid>/csci6509/lab1`" (or "`<your_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.

Note about emacs and other editors: 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:

`hello.pl`

```
#!/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`.

Note: In the above block of code with gray background, the first separated line containing '`hello.pl`' is just the name of file and you should not type it into the file. The rest of the lines are the contents of the file.

Emacs can provide syntax highlighting, or also called font highlighting, which can be on or off. If you do not see font highlighting and would like to have it, you can enable it by typing in Emacs: `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)`
If you run the command `M-x global-font-lock-mode` once more, you would turn off font highlighting, which is equivalent to the following command in the `.emacs` file in your home directory:
`(global-font-lock-mode 0)`

Step 6: Running a Perl program. Exit Emacs and enter into the command line:

```
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
```

This ends a brief introduction into the FCS server environment. If you wanted to finish your session you would need to type the command '`exit`', but we will not do this yet but continue with the Perl tutorial.

Perl Tutorial 1

In the next few labs we will cover some basic elements of the Perl programming language. Currently, the Python language is used more in general NLP research and we will cover it later. Perl is still often the best choice for lower-level text processing, simpler text manipulation, especially when we can exploit significantly the power of regular expressions. We will not go into some more advanced Perl features, such as Object-Oriented style, or libraries more oriented towards system, networking, and other application areas.

In the tutorial up to this point, you already saw a simple hello-world program in Perl named `hello.pl`.

Slide notes:

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`

Slide notes:

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 32, 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

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 your current lab directory, and the content of the file should be:

hello.pl

```
#!/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: (`hello.pl`) You need to submit the program `hello.pl` to be marked. You can submit the program by running the following command on timberlea:

```
submit-nlp hello.pl
```

You will be prompted to enter your CSID and password.

Note about Web submission: There is also a way to submit your program from your computer directly using the web interface available at: `https://web.cs.dal.ca/~vlado/csci6509/login/`
You can use this interface only if you copy the program to your local computer. You will need your CSID and password to login, and then you should click 'File submission' menu tab, and submit the file. You can also access the list of submitted files by clicking the 'Submit list' tab. You can submit the same file many times before the deadline, and only the last submission before the deadline will be marked. **Be careful** to name the files exactly as specified, including the case (uppercase or lowercase) of the letters, otherwise your file will not be counted as submitted.

Using the web interface means that you can do labs on your own computer and submit everything from there. However, the programs will be tested on the `timberlea` machine and you will lose marks if the program does not work properly on `timberlea`. Hence, you should make sure that the program run correctly on `timberlea`.

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. The standard way of using Perl is to write a full program in a file and run it separately. When we run a Perl program, the Perl interpreter will read the whole file, compile it into an intermediate form, and then execute it. It is not the most convenient way to quickly test the Perl commands, and a direct interaction with the interpreter may be useful.

To interact with the Perl interpreter, a common way is actually to run the Perl debugger. This is not something that you are required to try, but you can if you are interested. The command to start the debugger is the following: '`perl -d -e 1`' or shorter '`perl -de1`'.

If you try this command, the debugger will produce output as follows, or similar:

```
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

We will first go over some basic syntactic elements of the programming language Perl. We assume that you already know a programming language and have some experience with programming. If not, you can start with Perl or any other language, but you should follow then a more detailed step-wise tutorial to learn it.

Slide notes:

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)

The following are some basic syntactic elements of Perl:

Statements: The Perl statements are separated by semi-colon (`;`). They must be always separated by a semi-colon. The last statement in a block does not have to have a semicolon at the end, but it is common to have it. Whitespace in code, such as space characters, tabs, and new-line characters, do not matter, other than token separators, or if they are inside literal strings.

Comments: Perl uses line comments starting with the hash symbol (`#`), such as in:
`# a comment until the end of line`

Variable names: The variable names start with special symbols: `$` for scalar variables (numbers and strings), `@` for array variables, and `%` for associative arrays (known as hashes, maps in Java, or dictionaries in Python). For example:

```
$a = 5;          # example of a scalar variable
$a = 'hello';
@b = (1, 2, 3); # example of an array
print $b[0];    # however $b[0] is an element of the array (first)
%c = ( 'jane' => 1, 'joe' => 2); # hash example
print $c{'jane'};  # element of a has (prints 1)
```

The special variable symbol at the beginning is also known as a *sigil*, and it is followed by an identifier (starting with a letter and then followed by letters, digits, or underscore); for example `$varname`. The sigil can also be followed by a special symbol, as in `$!`, but those variable names are usually reserved for some special purposes, so do not use them unless you known this purpose. The user-defined subroutines should be prefixed with the sigil `&`, although this is not mandatory, as in `&a(1,2)`. There is a difference between using the sigil `&` at the beginning of a subroutine name or not, so you should always use it until you learn the difference.

Step 8: Example Program 2

- Enter the following program as `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

Prepare the program `lab1-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 lab1-example2.pl
./lab1-example2.pl
```

Submit: Submit the program '`lab1-example2.pl`' using: `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 `lab1-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 `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`

Variables

Slide notes:

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

In many operations that require a variable, if a variable is not named, then the operation is done on the default variable by default.

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 and Operators

Slide notes:

String Literals and Operators

- We will over over string literals and main operators
- 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

A string can span multiple lines. This is a very convenient feature but it is also a source of many syntactic errors that are hard to find. Namely, if we start a string and not finish it, the Perl interpreter will keep reading source code for multiple lines until it can finish the string, and this usually leads to a hard-to-understand syntactic error in the code.

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: ><
```

We can notice that the logical true value in Perl normally corresponds to number or string 1, and the false value to an empty string.

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 repeated:

```
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 2022
# 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'; #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

The Perl control structures, i.e., control-flow structures, are very similar to other programming languages, with maybe a few specific details. They look very much like C, C++, or Java, with blocks delimited with braces `{` and `}`, and logical expressions delimited by parentheses `(` and `)`.

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'

The `goto` statement is not used that often. It is similar to C and will move control to a label in code followed by a colon (e.g., `L:`).

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

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. 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";
```

- **Remember to add a file header comment**

Submit: Submit the program '`lab1-task2.pl`' using: `submit-nlp`

This is the end of Lab 1.