# Lab # 4

## Deadline

Please refer to the dropbox.

## Question

Please refer to the word file on Shell scripts. Please read the document carefully and write the summary in a maximum of 5 pages.

Please note that shell scripts will NOT be there for the final exam.

## Answer

Many built-in commands are available in the command line interface to traverse and work with files and folders on a system. Some of the most commonly used commands include: "ls" to list directory contents, "pwd" to print the current working directory, "cd" to change directories, "wc" to count words in a file, "mkdir" to create a new directory, "rm" to remove files or directories, "rmdir" to remove empty directories, "grep" to search for a pattern in files, "sort" to sort lines of text, and "find" to search for files in a directory hierarchy. Each command has a unique set of parameters and options that can be used to modify how it behaves. For example, "ls" and the "-l" option can be used to display extensive information about files and directories, while "find" and the "-type" option can be used to look for files of a certain type.

Redirection and piping are two potent methods for modifying command input and output in Linux and Unix-like operating systems. Users can alter the default input or output of a command from the keyboard or display to a file or another command via redirection. There are three different types of redirection operators: >, which routes a command's output to a file and overwrites its contents if the file already exists; >>, which routes a command's output to a file and appends it to the end of the file if the file already exists; and, which routes a command's input from a file. Contrarily, piping allows users to design a series of commands where each command accepts input from the previous command and produces output for the following command by connecting the output of one command to the input of another command using the | symbol. In a Unix-like environment, redirection and piping are frequently used to filter, sort, search, and alter text data.

Metacharacters are special characters that can be used to match patterns in text processing or pattern matching. They have a specific meaning in a computer language or context. Metacharacters are used by regular expressions to match specific patterns in a string of characters. Metacharacters include symbols like the dot (. ), asterisk (), plus (+), square brackets ([]), and parentheses (()). Metacharacters are used in shell scripting to carry out redirection and piping actions. Metacharacters can be used for a variety of purposes, such as matching a single character with a dot (. ), matching zero or more instances of a character with an asterisk (), and matching a particular group of characters with square brackets ([]). You can match the start or end of a line with the caret () or dollar sign ($) metacharacters, respectively.

Variables can be declared, given values, and exported in shell scripting so that other processes can access them. Simply give a variable a name and an equal's sign-delimited value to declare it. The variable name can then be used to obtain the value. Use the export command and the variable name to export a variable. Shell scripts can employ variables in a variety of ways, such as conditional statements and loops, to make them more dynamic and adaptable. Variables can also be imaginatively used, such as in ASCII art, to give scripts more individuality.

A shell variable that is predefined by the shell or operating system is known as a system-defined shell variable. These variables, which are accessible to all processes and users on the system, often have uppercase names like $HOME, $PATH, $USER, $SHELL, and $PWD. They serve as a source of system data and configuration options, and they could significantly affect how the system functions. A user-defined shell variable, on the other hand, is one that is set by the user in the current shell or in a script. These variables often have lowercase names like $my var, $my other var, $name, $age, and $filename and, unless exported to the environment, are only accessible within the current shell session or script. They can be edited or removed by the user and are often used to hold information or configuration settings unique to a given script or session.

With interactive shell scripts, users are prompted for input using the read command. The output can be adjusted, or computations can be made using this input. The read command is used in two interactive shell script examples. Whereas the other asks for a number and calculates its square, the first asks the user for their name and a choice from a menu of ASCII art possibilities. The first example's script is made more engaging and fascinating by the usage of ASCII art. Generally, interactive shell scripts that react creatively to user input can be made using the read command.

In shell programming, the read command can be used to invoke interactive scripts that take the user's input and respond in a variety of ways. As two instances, one asks the user for a number to display a multiplication table along with ASCII art of a calculator, while the other greets them with personalised ASCII art depending on their supplied name. Scripts can be made more interesting and remembered by ASCII art. You may write a variety of interactive shell scripts that are both helpful and entertaining using the read command.

Here are a few instances of shell scripts that carry out various operations using command-line arguments. In the first illustration, the wc command is used by a script to count the

number of lines in a file after receiving a filename as an argument. The second script displays the total of two numbers that are provided as inputs. The third script uses the rev command to show a string's reverse after accepting it as an argument. The final script utilises the ls command to list the contents of a directory and takes a directory name as a parameter. To give the output some visual interest, each script also includes ASCII art. These examples show how shell scripts may be written to carry out tasks by using command-line arguments.

Positional parameters are used in shell scripting to save the command-line arguments given to a script. Any more arguments are saved in $@ or $*, while the first argument is stored in $1, the second in $2, and so on up to $9. A shell script that takes two command-line arguments and uses them to customise a welcome is an example of using positional parameters. Moreover, positional parameters can be set using the set command within a shell script, which is useful for changing command-line inputs in the middle of a script.

Shell scripting's command substitution function makes it possible to use a command's output as an argument for another command or as a value for a variable. This capability is required when you want to keep the results of an operation for later use or use them in another command. For instance, you may count the number of files in a directory and save the result in a variable by using the ls command to list the files in a directory and the wc programme to count the number of lines in the output. Furthermore, the set command sets positional parameters from within a script, and positional parameters save the command-line arguments supplied to a shell script. By utilising these elements, you can produce scripts that are more dynamic and adaptable and that can contain ASCII graphics to improve their appeal.

Programmers can create conditional and repetitive code by using shell control structures like test, if, case, for loops, while loops, and until loops. Expressions are evaluated by the test command, which then produces a Boolean result indicating whether the test was successful or unsuccessful. Depending on the outcome of the test, if statements will run a block of code conditionally. While loops repeatedly run a code block while a specific condition is true, for loops iterate over a list of values and execute a code block for each value. Like the while loop, the until loop constantly runs the block of code until the condition is met. Each control structure can be tailored to carry out functions according on the programmer's needs, and they can be applied in a variety of ways to enhance the usefulness of shell scripts.

A collection of commands to be run when a signal is received can be specified using the trap command in shell scripting. Signals are notifications that an operating system or another process sends to a process, and they can be used to affect how a script behaves. The trap command can be used to provide signal handlers for signals, which can then be used to run code or carry out bespoke operations. The trap command in a shell script can be used to configure signal handlers for a variety of events, including keyboard interrupts, failures, and exceptions. Moreover, adding ASCII graphics can add fun and intrigue to shell programs.

Writing reusable and modular code in shell scripts requires the use of shell functions. They enable you to isolate a certain set of operations or functionality and call it repeatedly from

various script sections. We may observe various shell functions in the given examples, including display number, factorial, random cat, and sum diff. These functions show off a variety of talents, including the ability to display a number with a decoration, compute a number's factorial through recursion, produce a cat's ASCII art at random, and execute operations on two numbers. Shell functions have the potential to make difficult operations easier to complete and make shell scripts easier to read.