Unix Fundamentals – Shell Syntax Basics

Marek Kozłowski
Faculty of Mathematics and Information Sciences
Warsaw University of Technology

Never use copy&paste for the following exercises. Retype all commands manually!

Don't just read those exercises and examples. Do them, please! Then make sure you understand how and why they work. Refer to a respective presentation or ask the teacher shall the need arise.

Leading $ denotes an ordinary user's prompt (don't type it in!). A command to be entered follows it.

Be careful with spaces: don't add any extra ones nor omit if there are some!

1. **Shells**

   (a) Which bash version is used on your workstation? Let's check:
   `$ bash --version`

   (b) By default you are using `bash`. Are there any other modern shells installed on your workstation: `tcsh`, `ksh`, `zsh`? Try to run them, for example:
   `$ tcsh`
   For quitting a shell execute:
   `$ exit`
   Note that default prompts for other shells may differ.

   (c) You may redefine the default prompt in Bourne-compatible shells by executing:
   `$ PS1="my new prompt here "`
   Remember not to place any spaces before nor after '`=`' !
   Close your terminal emulator window and open a new one for resetting your prompt to the default (and more useful) value.

2. **Entering a command**

   (a) Verify commands and options case-sensitivity (`ls` *directory contents*):
   `$ ls`
   `$ ls --version`
   Those won't work:
   `$ Ls`
   `$ ls --Version`

   (b) Remember that blank separators are a requisite. This won't work:
   `$ ls--version`

   (c) Note that # in string literals doesn't start a comment. Let's check:
   `$ echo "Hello"`
   `$ #echo "Hello"`
   `$ echo #"Hello"`
   `$ echo "#Hello"`

   (d) Unterminated string literals are not allowed. Users are asked to continue in the next line.
   `$ echo \`
   `> "Hello"`
   `$ echo "Hello`
   `> "`

In case of problems you may close and re-open a terminal emulator. You may also use the `Ctrl-c` key combination.

3. **Running simple commands**

   (a) The '`echo`' command is implemented twice: as a bash built-in and as a binary program:
   `$ echo "Something"`
   `$ /bin/echo "Something"`
   They work almost always the same way, however in some rare cases small differences do exist – compare:
   `$ echo --help`
   `$ /bin/echo --help`
   Honestly, such implementation is error-prone and may lead to unpredictable side effects. In short: it is a design error.

   (b) Run all commands from examples given on slides 13–16.

   (c) Get help on the '`ls`' command:
   `$ ls --help`
   What is the long option equivalent to '`-l`'? Oups!
   Take a look at the information on exit status (at the end). Remember: 0 denotes success/true!

   (d) Display help on '`touch`':
   `$ touch --help`

   (e) Create a file with the name '`--help`':
   `$ touch -- --help`
   and delete ($\mathtt{r}_e\mathtt{m}_{ove}$) it:
   `$ rm -- --help`

   (f) Create three files: '`file1`', '`file2`', '`file3`' with a single command:
   `$ touch file1 file2 file3`
   and delete them:
   `$ rm file1 file2 file3`

4. **Bash features**

   (a) Display all commands recognized by your shell that start with the letter '`b`'. How about '`c`'?
   Tips: Use auto-completion for this purpose. Double press the <TAB> key if necessary. Press <SPACE> if you see: '`--More--`'.

   (b) How many filenames in '`/etc`' start with '`pa`'? Remember: only command arguments are extended to filenames so precede the path with some (any) command introduced during preceding exercises.

   (c) Experiment with a bash history. Use:
   `$ history | grep` *somepattern*
   for only those lines that contain a *somepattern*. Note that this command modifier (a filter) also works for all other commands.

   (d) What does the '`!!`' command do? Let's check.

   (e) Make sure your home directory is the default (working) one and display the content of your history file:
   `$ cd`
   `$ cat .bash_history`
   Clear your current bash session history:
   `$ history -c`
   Delete the content of your history file (`.bash_history`) – clear the history permanently:
   `$ echo -n > .bash_history`

5. **Special characters**

(a) C$_{hange}$ d$_{irectory}$ and display a new p$_{ath\ to}$ w$_{orking}$ d$_{irectory}$:
```
$ cd /etc; pwd
$ cd /nonexisting; pwd
```
Don't display path to working directory if a directory change failed:
```
$ cd /nonexisting && pwd
```
If you can't enter a non-existing directory then enter your home directory (if we don't specify where to go then '`cd`' goes home):
```
$ cd /nonexisting || cd && pwd
```
Remember that commands are processed in order, from left to right.

(b) What directory contents is listed by executing the following command?
```
$ ls /./etc/./../usr/.././.
```
Well, single dots don't do anything so we may denote it as:
```
$ ls /etc/../usr/../
```
We enter sub-directories of the top-most directory then we go one level up, so it is equivalent to:
```
$ ls /
```

(c) Are those commands equivalent?
```
$ ls .
$ ls
```
Yes, they are.

(d) Using *Geany* create a C program *hello world* and compile it (the *brick* icon). Open a terminal emulator. Are those two command equivalent?
```
$ ./myprogram
$ myprogram
```
No, they are not. If there is no path specified your shell seeks for executable files only in directories specified by some list (aka `PATH` variable). Your working nor home directories are not on that list. In the first case you specify a path (a relative path). There is no path in the second one.

(e) Are those commands equivalent?
```
$ ls /etc
$ ls etc
```
No, they are not. The first one specifies an absolute path. There second one tries to list the '`etc`' subdirectory of your working directory.

(f) Create several files with similar names, let's say:
```
$ touch f1 f2 f3 f4 f5 f6
```
and delete them using a wildcard:
```
$ rm f?
```

(g) Let's create a new directory (m$_a$k$_e$ dir$_{ectory}$) and enter it for this exercise:
```
$ mkdir Somedir; cd Somedir
```
Try to guess and check what files are created with the following commands:
```
$ touch \$ ; ls
$ touch "'" ; ls
$ touch \~ ; ls
$ touch '\*' ; ls
$ touch "\"\"" ; ls
$ touch \# ; ls
```
Conduct experiments on your own.
Finally change back to your home directory and delete Somedir and its contents (recursively):
```
$ cd .. ; rm -r Somedir
```

(h) Create a file with a name containing spaces:
```
$ touch "file name with spaces"
```
Start typing and then press <TAB> for auto-completion:
```
$ ls file
$ ls "file
```

6. **Variables**

   (a) Remember not to insert separators:
   `$ SOMEVAR="something"`
   What happens if you insert spaces?
   `$ SOMEVAR = "something"`

   (b) Does use of unassigned variables result in an error?
   `$ echo $UNASSIGNED`
   Absolutely not. They are assigned empty strings.

   (c) Can variables substitute any text? Yes they can:
   `$ SOMEVAR=ls`
   `$ $SOMEVAR`
   `$ SOMEVAR=l`
   `$ ${SOMEVAR}s`

   (d) Check if '∼' and '`$HOME`' point to the same directory:
   `$ echo $HOME`
   `$ echo ∼`
   Yes, they do.

   (e) Check if '∼' and '`$HOME`' in double quotes point to the same directory:
   `$ echo "$HOME"`
   `$ echo "∼"`
   No, they don't. It's due to bash interprets '`$`' and doesn't interpret '∼' in double quotes.

   (f) How many shell variables are environment ones too ('`| sort`' sorts lines lexicographically)? Let's see:
   `$ set`
   `$ env | sort`

   (g) Check values of important variables listed on the slide #43.

   (h) Are new variables inherited by child processes?
   `$ SOMENEW="something"`
   `$ bash`
   `$ echo $SOMENEW`
   No, they are not unless we `export` them.

   (i) Is the `LANG` variable inherited by child processes?
   `$ LANG=pl_PL.UTF-8`
   `$ bash --version`
   Yes, it is, because it is an environment variable.
   If you can't speak Polish close and re-open your terminal emulator which resets all variables to their default settings.

   (j) Add current directory (denoted as '.') to the PATH variable:
   `$ PATH=${PATH}:.`
   or alternatively:
   `$ PATH=.:$PATH`
   And check it:
   `$ echo $PATH`

   (k) Do to the exercise (5d) again.

7. **Configuring bash**

   (a) Check if '`.bashrc`' and '`.bash_profile`' are present in your home folder:
   `$ ls -a`
   If not then copy the templates:
   `$ cp /etc/skel/.bash_profile ∼`
   `$ cp /etc/skel/.bashrc ∼`

(b) Let bash remember more commands in history. Open the file '`.bashrc`' in your home directory (use *mousepad* or *leafpad*) and add the following commands at the end:
```
HISTSIZE=2048
HISTFILESIZE=2048
```
It will work for all bash instances started since then.