Bash, a command line interface for interacting with the operating system, was created in the 1980s. Other popular shells are *zsh* and *fish*.

1 Programming in Bash

1.1 Shebang

The shebang (`#!`) at the head of a script indicates an interpreter for execution, as in `#!/bin/bash`. Lines starting with a `#` (with the exception of shebang) are comments and thus won't be executed.

1.2 Quoting and literals

Single quotes '' preserve the literal value of characters enclosed within them. A single quote may not appear between single quotes, even when escaped, but may appear between **double quotes** "".

They work similarly, with an exception that the shell expands any variables that appear within them.

1.3 Variables

Variable names are case sensitive. They can contain digits and underscores as well, but a name starting with a digit is not allowed. Example:

```
var="kind"
echo ${var}ness # kindness
```

Special variables:

1. $0: name of the script itself.
2. $1, $2, $3, ...: the first, second, etc. argument. `shift` removes first argument and advances rest of them forward.
3. $* and $@ denote all the positional parameters.
4. $#: the number of positional parameters
5. $?: exit status of last executed command.
6. $$: the process ID of the shell.
7. $!: the process ID of last executed command.

To read a line of input, use `read` shell built-in.

- ☐ **read** reads a line from the stdin:
  - n returns after reading *n* characters,
  - n displays a prompt.

1.4 Expansions

"After the command has been split into tokens, these tokens or words are expanded or resolved. There are eight kinds of expansion performed, which we will discuss in the next sections, in the order that they are expanded."

1.4.1 Brace expansion

Brace expansion is used when we need to generate all possible string combinations. Both of the commands produce the same output:

```
echo {I,really,love,dots}.
echo I. really. love. dots.
```

Warning: it does not expand the variables (`$var`), which is done later, but supports ranges (sequences) of characters:

```
echo {a..t}
a b c d e f g h i j k l m n o p q r s t
```

and (maybe zero paded or with an increment rate) integers, assuming the Bash version is 4 or newer:

```
echo {01..10..1}.~
01. 02. 03. 04. 05. 06. 07. 08. 09. 10.
```

There is a tilde expansion as well. The expressions ~ and ~<user> expand to the home directory of the current (or given) user.

1.4.2 Parameter expansion

1. `${var^}`, `${var,}` convert first character to upper and lowercase. `${var^^}`, `${var,,}` do the same to all characters.

   `${var~}`, `${var~~}` are undocumented now: they reverse the case.

   In case of the array expansion, every expanded element changes case, no matter what.

2. `${var#pattern}` removes the `pattern` from the beginning of the string, if possible.

   It's greedy variant is `${var##pattern}`.

   `${var%pattern}` and `${var%%pattern}` do the same, but from the end of the string.

   Application: extracting parts of a filename.

3. `${var/pattern/string}` performs a single search and replace operation.

   `${var//pattern/string}` searches for all occurrences of the pattern and replaces them.

4. `${#var}` returns length of the string.

5. `${var:offset:length}` skips first `offset` characters from `var` and truncates the output to given length. `:length` may be skipped.

   Negative values separated with extra space are accepted.

6. `${var:-value}` uses a default value, if `var` is empty or unset.

   `${var:=value}` does the same, but performs an assignment as well.

   `${var:+value}` uses an alternative value if `var` isn't empty or unset!

1.4.3 Command substitution

To execute commands in a subshell and then pass their standard output, use `$( commands )`.

1.4.4 Arithmetic expansion

The arithmetic expression `$(( ... ))` is evaluated and expands to the result. Bash guarantees that the output will be a one-word integer.

1.4.5 Process substitution

This kind of substitution: `<( ... )` and `>( ... )` (not specified by POSIX!), where input or output of a command appears as a temporary file, is performed simultaneously with the following: arithmetic and parameter expansions, command substitution.

1.5 Streams

There are always three default files open:

1. *stdin* (the keyboard, file descriptor 0),
2. *stdout* (the screen, file descriptor 1) and
3. *stderr* (error messages output, file descriptor 2).

These **streams** can be **redirected**:

1. `cmd > file` redirects to a file (overwrites),
2. `cmd >> file` appends instead,
3. `m>n` (or `m>&n`) redirects a file descriptor to a file (or another file descriptor),
4. `&>file` redirects stdout and stderr to a file,
5. `:> file` truncates file to zero length,
6. `|` (pipe) serves as a command chaining tool.

Here document is a section of a source code file that is treated as if it were a separate file:

```
cat <<EOF > /path/to/your/file
   This line will write to the file.
EOF
```

1.6 Control flow statements

The one-line constructs `&&` and `||` work not like and, or ($\wedge$, $\vee$), but the *if – then – else* statement.

1.6.1 Conditionals

Here at least one statement must be specified inside every block, but one can use a single colon (:) as a null statement to avoid rewriting the code.

```
if condition; then
  commands
elif second_condition; then
  some_commands
else
  other_commands
fi

select word in "Bash" "Haskell" "Python"
do
  echo "Your language is $word".
done
```

There is also a case instruction:

```
case $language in
  bash)
    echo "Bourne Again Shell!"
  ;;
  python|haskell)
    echo "Python or Haskell!"
    exit 1
  ;;
  *)
    echo "Unknown language!"
  ;; # optional
esac
```

1.6.2 Testing conditions

Remember that `test` command follows symbolic links (except for the `-h` test).

1. **File tests**:
   (a) `-e` file exists, `-s` file is nonempty,
   (b) `-d` directory, `-f` regular file, `-h` symlink,
   (c) `-b` block device, `-c` character device,
   (d) `-p` named pipe, `-S` socket.
2. **File permissions**:
   (a) `-r` readable, `-w` writable, `-x` executable,
   (b) `-u` setuid, `-g` setgid, `-k` sticky bit.
3. **String tests**: `-z` empty, `-n` nonempty.
4. **Arithmetic tests**:
   (a) `-eq` $=$, `-ne` $\neq$,
   (b) `-lt` $<$, `-gt` $>$,
   (c) `-le` $\leq$, `-ge` $\geq$.

1.6.3 Loops

```
for var in "the first" "the second"; do
  echo "${var}"
done

for (( i = 1; i <= 10; i++ )); do
  echo "i = ${i}."
done # C-style

while read myline; do
  echo "It says ${myline}"
done < some_file
```

As Bash Guide for Beginners by M. Garrels says:

1. the `break` statement is used to exit the current loop before its normal ending.
2. the `continue` statement resumes iteration of an enclosing while, until, select or for loop.

author: Remigiusz Suwalski,
date: July 30, 2017

2 Shell style guide

The following notes are meant to be summary of a style guide written by Paul Armstrong and too many more to mention (revision 1.26).

Bash is the only shell scripting language permitted for executables. Bash should only be used for simple wrapper scripts or small utilities.

Executables should have no extension, libraries must have a `.sh` extension and should not be executable. SUID and SGID are *forbidden* on shell scripts.

All error messages should go to `STDERR`, a function to print out error messages along with other status information is recommended:

```
err () {
  echo "[$(date +%Y-%m-%d\ %T)]: $@" >&2
}
```

Comments. Start each file with a description of its contents. Any function that is in a library or not both obvious and short, must be commented. Comment tricky, interesting or important parts of code. Use TODO comments for temporary or good enough but not perfect code and short-term solutions.

The following are required for any new code:

1. Indent 2 spaces, no tabs.
2. Maximum line length is 80 characters.
3. Long pipelines should be split one per line.
4. Indent case alternatives by 2 spaces.
5. Always quote strings containing variables, command substitutions or spaces.

Use quotes rather than filler characters if possible. Use an explicit path when doing wildcard expansion of filenames. Avoid `eval`. Use process substitution or for loops in preference to piping to while. Finally, `[[ ... ]]` is preferred over `[` and `test`.

Naming conventions. Function and variable names should be lower case, with underscore to separate words. Constants and environment variable names should be all caps, declared at the top of the file. Use `readonly` or `declare -r` to ensure they're read only. Declare function specific variables with `local`. A function called main is required for scripts long enough to contain at least one other function.

2.1 Other useful resources

If you want to understand Bash better, consider one of the following webpages:

1. `http://wiki.bash-hackers.org`
2. `http://mywiki.wooledge.org/BashGuide`

Avoid guides by TLDP (The Linux Documentation Project) at all costs.

3 Emacs shortcuts in Bash

For detailed information on emacs editing mode, visit `http://readline.kablamo.org/emacs.html` or type `man readline`.

`Ctrl-l` clears the screen.

`Ctrl-a` moves to the start of the line,
`Ctrl-e` moves to the end of the line.

`Ctrl-u` deletes to the beginning of the line,
`Ctrl-k` deletes to the end of the line,
`Ctrl-w` deletes to the start of the word,
`Ctrl-y` pastes text from the clipboard.

`Alt-r` undoes all changes to the line,
`Ctrl-r` searches incrementally up the history,
`Ctrl-xe` invokes an editor to write commands,
`Alt+.` inserts the last argument of last command.

4 Text processing (grep, sed, awk)

This is an expanded description of three powerful text processing tools: `grep`, `sed` and `awk`.

See also `ack`, an optimized for programmers tool like grep, at `https://beyondgrep.com/` or learn Perl.

4.1 grep – pattern search enginge

The ed command `g/re/p` was used to **g**lobally search a **r**egular **e**xpression and **p**rint.

☐ **grep** prints lines matching a pattern:
`E` interprets pattern as an extended regexp,
`F` does not recognize regexps,
`P` interprets pattern as a Perl regexp,

`c` prints a count of matching lines instead,
`m` exits after finding … matches,
`o` prints only matched parts of lines,
`n` prints line numbers as well,
`r` reads all files under each directory.

`e` uses a "regexp" pattern,
`f` obtains patterns from a file,
`i` ignores case disctinctions,
`v` inverts the sense of matching,
`w` selects only lines with whole words matches,
`x` selects matches exactly matching whole line.

`A` prints … lines of trailing content,
`B` prints … lines of leading content,
`C` prints … lines of both contents.

4.2 sed – stream editor

■ **sed** filters and transforms text:
`-e` adds a script to the commands to be executed,
`-i` edits files in place. With suffix supplied, makes backup (`sed -i[suffix] ...`),
`-n` suppresses auto- printing of pattern space,
`-r` accepts extended regular expressions.

The simplest usage is `sed 's/foo/bar/g'` which substitutes (`s`) strings globally (`g`). There are other options, including:

`a` appends line before,
`d` deletes line,
`i` inserts line before,
`p` prints line,
`w` writes pattern space to a file.

Default delimiter `/` can be replaced by any other. This is useful when regular expression already contains `/`. Addresses allow limiting to given line numbers:

1. `1-10` first ten lines
2. `$` the last line
3. `10~2` even lines starting from the 10th.

One can also use regular expressions:

```
sed -e '/:/s/ /_/g'
```

replaces spaces with underscores in lines containing a colon. Negation may be obtained with `!s`.

4.3 awk – Aho, Weinberger, Kernighan

☐ **awk** is a language used as a data extraction and reporting tool. General form of its code:

```
#! /bin/awk
pattern {actions} # comment
```

Supported types of pattern are

1. `BEGIN` for initialization,
2. regular expressions (enclosed in `/ /`, single slashes): for example `/[a-z0-9_-]{3,16}/`,
3. awk expressions, comparison operators (<, >, <=, >=, ==, `!=`, tilde: matches, `!` negates),
4. `END` for final actions.

Awk is weakly typed: variables can be treated either as numeric values or strings, which are not represented as one-dimensional arrays of characters! Important variables include:

1. **FS**: field separator (tab and space by default),
2. **OFS**: output field separator,
3. **RS**: record separator (new line),
4. **NR**: number of the current record,
5. **NF**: number of fields in the current record,
6. **$0**: the entire input record,
7. **$1**, **$2**, …: fields in the current record.

Available numerical functions:

1. `int`,
2. `sqrt`, `exp`, `log`,
3. `sin`, `cos`, `atan2`,
4. `rand` (uniformly distributed value from [0, 1)), `srand` (sets the seed (by default, time of day) for random number generator).

String/text functions:

1. `length` returns number of characters.
2. `split` divides string into pieces separated by a separator.
3. `sprintf` returns string printed by `printf` (which is C-like function) without printing.
4. `gsub` searches for all of the longest, leftmost, nonoverlapping substrings and replaces them (globally). `sub` replaces only first match.
5. `index` searches for the first occurrence of a string. `match` accepts regexes too.
6. `tolower`, `toupper` convert case.

4.4 Regular expressions

1. POSIX character classes:
   (a) `[:lower:]` = `[a-z]`
   (b) `[:alpha:]` = `[a-zA-Z]`
   (c) `[:alnum:]` = `[a-zA-Z0-9]`
   (d) `[:word:]` = `[A-Za-z0-9_]`
   (e) `[:digit:]` = `[0-9]`
   (f) `[:xdigit:]` = `[A-Fa-f0-9]`
   (g) `[:blank:]` = `[ \t]`
   (h) `[:space:]` = `[ \t\r\n\v\f]`
   (i) `[:cntrl:]` = `[\x00-\x1F\x7F]`
   (j) `[:graph:]` = `[\x21-\x7E]`
   (k) `[:print:]` = `[\x20-\x7E]`
   (l) `[:ascii:]` = `[\x00-\x7F]`
2. Repetitions:
   (a) `*`: 0 or more, `+`: 1 or more, `?`: 0 or 1,
   (b) `{a, b}`: at least *a*, at most *b*.
3. Anchors:
   (a) `^`: start of line,
   (b) `$`: end of line,
   (c) `\<`: start of word,
   (d) `\>`: end of word.
4. Other:
   (a) `one|two`: one or two,
   (b) `(one)`: a group,
   (c) `$n`: *n*th group,
   (d) `[abcd]`, `[a-d]`: ranges,
   (e) `[^abcd]`: negation (not `[abcd]`).

author: Remigiusz Suwalski,
date: July 30, 2017

5 Unix utilities and shell builtins

5.1 File system

■ **cat** concatenates and prints files:
A shows all nonprinting characters,
b numbers nonempty output lines,
s suppresses repeated empty output lines.
■ **tac** does the same in reverse.
□ **rev** reverses lines characterwise.
■ **nl** numbers lines of files:
s adds "string" after line number,
w uses "number" columns for line numbers.

■ **chgrp** changes group ownership.
■ **chmod** changes permissions of a file:
ugoa of the owner, group, other or all users,
+-= adds, removes or sets selected file mode bits,
rwx selects file mode bits: read 4/write 2/execute 1.
■ **chown** changes owner of a file.
□ **umask** sets file mode creation mask.
■ **touch** changes file timestamps:
a only the access time,
m only the modification time,
t uses custom stamp instead of current time,
c does not create files.

■ See also: **cksum** (CRC checksums), **md5sum**.
■ **shasum** prints or checks SHA message digests:
a algorithm: 1, 224, 256, 384, 512, 512224 or 512256,
b reads in binary mode,
c checks SHA sums read from the "files".
■ **wc** prints newline, word and byte counts (`lwc`):
m prints the character counts,
L prints the maximum display width.

■ **dd** converts and copies a file:
1. `if=` reads from a file,
2. `of=` writes to a file,
3. `bs=` up to "bytes" bytes at a time,
4. `count=` copies only "n" input blocks.

■ **cp** copies files and directories:
b makes a backup of existing destination files,
f removes an existing destination file if needed,
i prompts before overwrite,
n does not overwrite existing files,
L always follows symlinks in "source",
P never follows symlinks in "source",
p preserves timestamps, mode, ownership,
r copies directories recursively,
s makes symbolic links instead,
l hard links files instead,
t copies all "source" arguments into "directory",
T treats "destination" as a normal file,
u copies only newer source files,
v explains what is being done.
■ **mv** moves (renames) files:
b makes a backup of existing destination files,
i prompts before overwriting,
f does not prompt before overwriting,
n does not overwrite existing destination files.
t moves all "source" arguments into "directory",
T treats "destination" as a normal file,
u moves only newer source files,
v explains what is being done.
■ **rm** removes files or directories:
f never prompts,
i always prompts,
r removes directories and their contents.
■ See also: **rmdir** (directories removal), **shred**.
■ **mkdir** makes directories (`-p`: with parents as needed, no error if existing).

■ **df** reports file system disk space usage:
h prints size in powers of 1024,
i list inode information instead of block usage,
t limits listing to file systems of given type,
x limits listing to file systems not of given type,
T prints file systems types.

■ **du** estimates file space usage:
a writes counts for all files, not just directories,
c produces a grand total,
d the depth at which summing should occur,
h prints sizes in human readable format,
s diplays only a total,
X excludes files that match pattern.

□ **file** determines file type.

□ **find** searches for files in a directory hierarchy.
1. Tests:
`-name` base of file name,
`-iname` case insensitive name,
`-group`, `-user` ownership
`-perm 755`, `-perm /u=x` permissions
`-size +5M -1G` size between 5MB and 1GB
`-amin -60` accessed in last hour
`-cmin`, `-mmin`: created, modified,
`-mtime +7` modified over a week ago
`-type d` directories only,
`-type f` files only,
`-empty` empty files or directories only,
2. Example (deletes files larger than 5 megabytes):
`find / -size +5M -exec rm -f {} \;`
□ See also: **whatis**, **whereis** and **locate**, whose database can be updated with **updatedb**.

◪ **fsck** checks and repairs a Linux filesystem:
a automatically repairs (without any question!),
t specifies the type(s) of filesystem to be checked,
A tries to check all filesystems in one run,
M skips mounted filesystems,
R skips the root filesystem.

■ **ln** makes hard links between files (only in the same file system, not between directories):
s makes symbolic links instead.

■ **ls** lists directory contents:
a does not ignore entries starting with dot,
F appends indicator to entries,
h prints human readable sizes,
i prints the index number of each file,
l prints permissions, number of hard links, owner, group, size, last-modified date as well,
r reverses order while sorting,
R lists subdirectories recursively,
S sorts by file size (largest first),
t sorts by modification time (newest first),
□ **tree** lists tree-like contents of directories.

□ **mount** mounts a filesystem.

■ **pwd** prints name of current directory.

■ **pv** monitors progress of data through a pipe.

□ **tar** stores and extracts files from a disk archive:
c creates a new archive,
x extracts files,
t lists the contents of an archive,
v verbosely lists files processed,
j bzip2 compression,
z uses zip/gzip (gz compression),
f uses archive file or device (???),
k does not replace existing files when extracting.

■ **tee** duplicates pipe content:
a appends to the given files, does not overwrite,
i ignores interrupts.

5.2 Processes

■ **chroot** changes the root directory of the calling process and their children.

□ **at** schedules commands to be executed once, at a particular time in the future: it accepts times of the form `HH:MM`, `midnight`, `noon` or `teatime`; `MMDD[CC]YY`, `MM/DD/[CC]YY`, `DD.MM.[CC]YY` or `[CC]YY-MM-DD` (the specification of a date must follow the specification of the time of day). You can also give times like `now + 3 hours`.

□ **bg** resumes suspended jobs in the background.
□ **fg** resumes suspended jobs in the foreground.
□ **jobs** lists the active jobs.
□ **cmd &** runs command in the background.

□ **crontab** maintain individual users' crontab files. See also **cron**: a daemon that executes scheduled commands.

◪ **kill** sends a `TERM` signal to a process.
□ **killall** kills processes by name.

□ **ps** reports a snapshot of the current processes:
e selects all processes,
f does full-format listing,
m shows threads (see also: `H`, `-L`, `-T`)
C selects processes by command name,
p selects processes by PID,
u selects processes by EUID or name.
□ **pstree** displays a tree of processes.

□ **nice** changes process priority.

□ **pgrep**, **pkill** looks up or signals processes based on name and other attributes.

□ **time** runs programs and summarizes system resource usage.

□ **iostat** monitors CPU and disk usage.
□ **vmstat** monitors memory usage.
□ **top** displays Linux processes.
□ See also: **htop** (Hisham top based on ncurses).

5.3 User environment

□ **clear** clears the terminal screen.
□ **env** runs programs in modified environment.
□ **exit** terminates the calling process.
□ **finger** looks up user information.
□ **history** displays the history list.
□ **mesg** displays messages from other users.

□ **passwd** changes user password:
d deletes (empties) an account's password,
e expires an account's password,
n minimum days to change password,
w warning days before password expire,
x maximum days a password remains valid.
□ **pwgen** generate pronounceable passwords:
s generates hard to memorize passwords,
y includes special characters,
n includes numbers,

◪ **su** changes user ID or becomes superuser.
□ **sudo** executes a command as superuser:
u as a different user.

□ **hostname** shows/sets the host name:
i displays the network address.
■ **uname** prints system information:
a all information, in the following order:
s the kernel name,
n the network node hostname,
r the kernel release,
v the kernel version,
m the machine hardware name,
p the processor type,
i the hardware platform,
o the operating system.

■ **uptime**: how long has system been running?

◪ **wall** writes a message to all users,
□ **write** sends a message to another user.

□ **pinky** is a lightweight version of finger.
□ **who** shows who is logged on,
□ **w** does the same, shows what they are doing,
□ **whoami** prints effective userid.

author: Remigiusz Suwalski,
date: July 30, 2017

5.4 Text processing

☐ **awk**, **grep** and **sed** have been described earlier.

■ **comm** compares two sorted files line by line.
■ **shuf** generates random permutations:
e treats each "arg" as an input line,
i treats each number .. through .. as an input line,
n outputs at most "count" lines,
r output lines can be repeated (with -n).
■ **sort** sorts lines of text files:
c checks for sorted input,
f folds lower case to upper case characters,
g compares general numerical values,
h compares human readable numbers,
k sorts via a key,
n compares string numerical values,
r reverses the results,
s stabilizes the sort.
■ **tsort** performs topological sort.
■ **uniq** omits repeated lines:
c prefixes lines by the number of occurences,
d only prints duplicate lines, one for each group,
f avoids comparing first fields,
i ignores differences in case,
s avoids comparing first characters,
w compares no more than *n* characters.

■ **cut** prints selected parts of lines:
--complement complements the selection,
c selects only these characters,
d uses "delim" instead of Tab for field delimeter,
f selects only these fields,
s does not print lines not containing delimeters.
■ **join** joins lines of two files on a common field.
■ **paste** merges lines of files.
d reuses characters from "list" instead of tabs,
s pastes one file at a time, not in parallel.
■ **tr** translates or deletes characters:
c uses the complement of "set1",
d deletes characters, does not translate,
s replaces each sequence of a repeated character that is listed in the last specified "set" with a single occurrence of that character.

☐ **diff** compares files line by line:
y outputs in two columns,
i ignores case differences,
w ignores all white space.

☐ **fmt** is a simple optimal text formatter,
☐ **fold** wraps each line to fit in specified width.

■ **head** outputs the first (last) part of files:
c the first "num" bytes,
n the first "num" lines,
■ **tail** the last "num" bytes:
c the last "num" bytes,
n the last "num" lines,
f outputs appended data as the file grows,
s sleeps for "n" seconds between iterations.
■ **split** splits a file into pieces:
a generates suffixes of length "n" (default 2),
b puts "size" bytes per output file,
d uses numeric (not alphabetic) suffixes,
l puts "number" lines/records per output file,
n generates "chunks" output files.
■ See also: **csplit**.

☐ **more** pages text too large to fit on one screen, allows scrolling down, not up (deprecated!).
☐ **less** is an enhanced version of more:
+F monitors the tail of a file which is growing.

☐ **vim** is an advanced text editor, too complex to be explained here. See also **emacs** or **nano**.

☐ **xargs** builds and executes command lines:
0 takes care of filenames with spaces, backslashes.
I replaces occurrences of "string" with names read from standard input.

☐ **yes** outputs a string repeatedly until killed.

5.5 Shell builtins

☐ **alias** allows a string to be substituted for a word.
☐ **cd** changes the shell working directory:
- to the previous directory.
☐ **echo*** displays a line of text:
e enables interpretation of backslash escapes,
n does not output the trailing newline.
☐ **test** checks file types and compares values.
☐ **unset** unsets a shell variable, removing it from memory and the shell's exported environment.
☐ **wait** waits for process to change state.

5.6 Networking

☐ **curl** transfers a URL.
☐ **ftp** is a File Transfer Protocol client.
☐ **wget** is a non-interactive network downloader.
A, R specifies lists of file suffixes or patterns (when wildcard characters appear) to accept or reject,
b goes to background immediately after startup,
c continues getting a partially-downloaded file,
m turns on options suitable for mirroring: infinite recursion and time-stamping,
np does not ever ascend to the parent directory when retrieving recursively,
U identifies as "agent-string" to the HTTP server.
w waits the specified number of seconds between the retrievals (see also --random-wait).

☐ **rlogin** starts a terminal session (which is not encrypted!) on a remote host.
☐ **ssh** connects & logs into a (remote) hostname:
C requests compression of all data,
i uses private key for authentication,
p selects a port to connect to on the remote host,
X enables X11 forwarding.

☐ **dig** interrogates DNS name servers.
x performs a simplified reverse lookup.
☐ **host** is a DNS lookup utility.
☐ **nslookup** is (probably) deprecated! Use **dig** and **host**.

☐ **arp** manipulates the system ARP cache.
☐ **ifconfig** configures a network interface.
☐ **netstat** prints networking subsystem related info: network connections, interface statistics, routing tables, and so on.
a whether sockets is listening or not,
t only TCP connections.
☐ **ping** tests the reachability of a host on an IP network by sending ICMP ECHO_REQUEST:
c stops after sending "count" packets,
n numeric output only, avoids symbolic names for host addresses lookup.
☐ **route** shows and manipulates the IP routing table.
☐ **traceroute** prints a trace of the route that IP packets are travelling to a remote host:
I uses ICMP ECHO for probes.

☐ **rsync** copies files fast (remote or local):
a in archive mode, equivalent to:
g preserves group,
o preserves owner (super-user only)
p preserves permissions,
t preserves modification times,
l copies symlinks as symlinks,
b make backups,
c skip based on checksum,
n performs a dry run without changes made,
r resursively,
u skip newer files on the receiver,
v increases verbosity,
z compresses file data during the transfer,
☐ **scp** copies files securely.

5.7 Hardware

◪ **dmesg** prints/controls the kernel ring buffer.
◪ **lsblk** lists block devices.
☐ **lsof** lists info about files opened by processes.
? **fuser** identifies processes using files/sockets.
☐ **lsusb** listsq USB devices.

5.8 For programmers

■ **g++** compiles, assembles and links C++ files:
o writes the build output to a file named …

5.9 Miscellaneous

☐ **bc** is an arbitrary precision calculator language.
1. echo 'obase=16;255' | bc prints FF,
2. echo 'ibase=2;obase=A;10' | bc prints 2,
3. scale=10 (after bc -l) sets working precision.

☐ **dc** is a reverse-polish desk calculator. One of the oldest Unix utilities, predating even the invention of the C programming language.
◪ **cal**, **ncal** displays a calendar.
e displays date of Easter,
j displays Julian days,
m displays the specified month,
w prints the numbers of the weeks,
y displays a calendar for the specified year,
3 displays the previous, current and next month.
☐ **date** prints or set the system date and time.
■ **seq** prints a sequence of numbers:
w equalizes width by padding with leading zeroes.
■ **sleep** delays for a specified amount of time.
☐ **true**, **false** does nothing, (un)successfully.

author: Remigiusz Suwalski,
date: July 30, 2017