Shell Scripting

Xiaoxu Guan

High Performance Computing, LSU

September 20, 2017

#!/bin/bash

(?<=^ >) (?=[a-zA-Z])

Overview

- What is a shell?

Overview

- What is a shell?
- What is **bash** and why do we need **bash**?

Overview

- What is a shell?
- What is **bash** and why do we need **bash**?
- Bash Shell Scripting
  - Linux Internal and External Commands;
  - Shell Parameters;
  - Standard Input/Output, and Exit Status;
  - Meta Characters; Control, and Logical Operations;
  - Quotes; Group Commands;
  - Special Parameters; Shell Arrays;
  - Pattern Matching; Arithmetic Operations;
  - Control Flow: Testing and Looping;
  - Aliases and Functions;
  - Regular Expressions;

Overview

- What is a shell?
- What is **bash** and why do we need **bash**?
- Bash Shell Scripting
  - Linux Internal and External Commands;
  - Shell Parameters;
  - Standard Input/Output, and Exit Status;
  - Meta Characters; Control, and Logical Operations;
  - Quotes; Group Commands;
  - Special Parameters; Shell Arrays;
  - Pattern Matching; Arithmetic Operations;
  - Control Flow: Testing and Looping;
  - Aliases and Functions;
  - Regular Expressions;
- **Summary and Further Reading**

What is a shell?

- We are using shell almost every day!
- Shell is a fundamental interface for users or applications to interact with the Linux OS and kernels;
- Shell is a special program that accepts commands from users' keyboard and executes it to get the tasks done;
- Shell is an interpreter for command languages that reads instructions and commands;
- Shell is a high-level programming language (compared to C/C++, Fortran, ...);
- It serves as a bridge between the Linux kernels and users/applications;
- Don't be confused with the Linux commands that need to be ran in a shell;

What is a shell?

- Shell has many different flavors from a number of aspects;
- At the system level: `(non-)login shells`,
  `(non-)interactive shells`;
- At the implementation level: `sh`, `bash`, `csh`, `tcsh`, `zsh`, · · ·
- Login shell is the shell where you land once you login into a Linux machine. Non-login shell might build on the top of login shell (executing by the shell name).
- It sets up system-wide variables (`/etc/bashrc` and `/etc/profile`) and user-specified variables (`~/.bashrc` and `~/.bash_profile`, if available).
- `$ echo $0` (or `ps -p $$`)
  `-bash` (login shell) or `bash` (non-login shell);
- The default shell is **bash** (**B**ourne-**A**gain **Sh**ell) on most Linux/Unix/MaC OSs;

What is bash and why do we need bash?

- Modern shells are very sophisticated today. You can reuse or modify the commands you ran before; define our own shortcuts for the commands; programmable . . .;
- **GNU Bash** is one of the GNU open source projects;
- Bash is the effectively “standard”, and probably the most popular shell;
- It’s very useful for Linux/Unix system administration;
- Bash, Python, Perl, and Ruby;
- Many startup scripts were written in Bash. Bash works better as it’s closer to OS;
- Learning Bash helps us better understand how Linux/Unix works;
- It’s not hard to master, but it might not be simple either (a lot of pitfalls): a **quick-and-dirty** way;

What is bash and why do we need bash?

- **Bash** shell scripting is **not** for everything:
  - Lack of rich data structures;
  - Heavy-duty floating point operations;
  - Extensive file operations (line-by-line operations);
  - Potential incompatibilities with other shells, if portability is critical;
  - Plotting, . . . ;
- Bash incorporates many features from other shells (`ksh` and `csh`); Significant improvements over `sh`;
- Enhance your productivity and efficiency;
- Bash supports filename globbing, redirections, piping, command history, substitution, variable, etc;
- Good for the tasks that repeat many times with minor or no changes in input parameters;

Example Scripts

- This's the list for the example scripts in the tarball:

  01-hello-world.bash
  02-quotes.bash
  03-case-v0.bash
  04-case-v1.bash
  05-for-loop-all-headers.bash
  06-for-loop-primes.bash
  07-while-loop-sleep.bash
  08-addition-v0.bash
  09-addition-v1.bash
  10-quadratic-function.bash
  11-arrays.bash
  12-alias-for-loop.bash

Bash Shell Commands

- Linux **internal** and **external** commands;
- Internal commands: builtin in the shell and no external executables needed (`cd`, `pwd`, `echo`, `type`, `source`, `bg`, ...);
- External commands: they are executable, separate files in your `$PATH` (`ls`, `mv`, `cp`, `rm`, ...);
- `$ compgen -b` (or `-a, -k, -e`)
  # list builtins (`-b`), aliases (`-a`), keywords (`-k`),
  # exported variables (`-e`), etc.
- `$ type command_name [which command_name]`
- The internal commands run faster than external ones;
- Commands include aliases, bash scripts (functions), builtins, keywords, external commands;
- All these can be called in a bash script (but pay attention to `aliases`);

Shell Parameters

- **Shell parameter** is a placeholder to store a value: variables and special parameters; all parameters can be assigned with values and can also be referenced;
- Substitution (parameter expansion) means referencing its value stored in that parameter;
- Builtin variables: `$BASH`, `$BASH_VERSION`, `$HOME`, `$PATH`, `$MACHTYPE`, `$SHLVL`, . . ., `$0`, `$1`, etc;
  `$my_variable` or `${my_variable}` [Don't use `$( )`]
- Variable assignment (avoid using `$`, `@`, `#`, `%`) and substitution `$`:
- Be careful with whitespaces (but why?);
- Shell variables are case sensitive;
- Bash variables are `untyped` (`typeless`)! It depends on whether a variable contains only `digits` or not;
- Define a constant: `$ readonly abc=456 ; abc=123`

Standard Input and Output

- Redirection facilities (`<` and `>`);
- $ `my_script < file.inp > results.out`
  # Input (`<`) from `file.inp` and
  # Output (`>`) to `results.out`;
- File descriptor `0` is for the standard input (`STDIN`), `1` for the standard output (`STDOUT`), and `2` for the standard error (`STDERR`);
- `$ my_script 1> script.out 2> script.err`
- A single file holding all error and output messages (two non-standard options: `>&` or `&>`)
- Remember that the default value for `>` is the `1`, and `0` for `<`;
- Compare `1>&2` and `2>&1` (note the spaces);
- The double greater-than sign `>>` means to append the output;

Exit Status

- The builtin exit command can be used to (1) terminate a script, and (2) return a value for the last executed command;
- The return value is called the exit status (or exit code). Here zero stands for a successful return (true), while non-zero value means errors (false or error code);
- It can be used to debug your scripts ($?); All functions, aliases, commands, bash scripts, . . ., return an exit code;
- Don't be confused with return, though both are the shell builtins;

  (1) return is used in a function to optionally assign the function to a particular integer value;

  (2) Function calls also return the information on whether it was successful or not. This is the exit status;

Bash Meta Characters

Meta Char.	Explanation
#	Are comments that will not be executed
&	Puts the command in the background
$	Expansion; (1) referencing the content of variable, (2) command substitution `$( )`, (3) arithmetic computation `$(( ))`
\	Escape meta characters; Protect the next character from being interpreted as a meta character
;	Connects two or more commands in a single line
;;	To mark the end of case statement
~	Means home directory

Bash Meta Characters

Meta Char.	Explanation
`" "`	(Double quote) protects the text from being split allows substitution
`' '`	(Single quote) protects the text from being split doesn't allow the substitution (**literal** meaning)
`:`	A null command (do-nothing operation, `true`)
`|`	The pipe operator
`( )`	The group command starting a **sub-shell**;
`{ }`	The group command **not** starting a **sub-shell**
`[`	Test (builtin, what about `]`?)
`[[ ]]`	Test (keyword)

Bash Meta Characters

Meta Char.	Explanation
␣␣	Separates the command/arguments and argument/argument
` `	Enclosed text treated as a command (output)
(( ))	Means arithmetic expression
$(( ))	Means arithmetic computation
< >	Redirections
!	Reverses an exit status or test (negate)
.	(1) Source command and (2) hidden filenames
/	Forward slash to separate directories

Control and Logical Operators

Operation	Explanation
`&&`	The logical `AND`
`||`	The logical `OR`
`!`	The logical `NOT` (depending on the exit status)
`Ctrl-A`	Moves cursor to the beginning of the command line
`Ctrl-E`	Moves cursor to the end of the command line
`Ctrl-D`	Log out of a shell
`Ctrl-Z`	Suspends a foreground job
`Ctrl-P`	Repeats the last executed command
`Ctrl-L`	Clears up the screen (`clear`)
`Ctrl-K`	Clears up the text from the cursor to the end of line
`Ctrl-R`	Searches through `history` command

Quotes

- That's where the confusion arose;
- **Three** types of quotes in Bash: **(1)** double quotes `" "`, **(2)** single quotes `' '`, and **(3)** backticks `` ` ` ``;
- Double quotes (`" "`): Allow substitution to occur, and protect the text from being split; Weak form quoting in the sense of the bash interpretation for characters in pattern matching;
- Single quotes (`' '`): Protect the text in its literal meaning, any interpretation by Bash is ignored, and protect the text from being split; Strong form quoting; A single quote may not appear between other single quotes; No escaping happens in single quotes;
- Backticks (`` ` ` ``): Enclosed text runs as a command (output);

Examples on Control and Logical Operators

- `$ ls -ltr ; echo `pwd``
- `$ my_script && echo `pwd``
- `$ echo `pwd` || echo "$USER"`
- `$ ps aux | grep "$USER"`
- `$ :  > my_script.log`
- `$ cat {file.1,file.2,file.3} > allfiles.123`
- `$ a=456 ; { a=123; }; echo $a` # not a sub-shell
- `$ a=456 ; ( a=123; ); echo $a` # starts a sub-shell
- `$ echo $?`
- `$ a=456 ; b=123 ; [ $a -eq $b ] ; echo $?`
- `$ a=456 ; b=123 ; ! [ $a -eq $b ] ; echo $?`
- `$ a=456 ; b=123 ; [[ $a -eq $b ]] ; echo $?`
- `$ test $a -eq $b ; echo $?` # the same as [[ ]]

Group Commands

- Group commands `{ }` and `( )`:
- Remember { } are the keywords, but ( ) are not;
- Two options for `{ }`:

(1) Multiple-line version (separated by a **newline**) is

```
{
 <command_1>
 <command_2>
 <command_3>
 }
```

(2) Single-line version (separated by ; and **whitespace**) is

```
{␣<command_1> ; <command_2> ; <command_3> ; }
```

- Don't forget that `{ }` doesn't invoke a sub-shell (i.e., in the same shell);

Group Commands

- Group commands `{ }` and `( )`:
- Remember `{ }` are the keywords, but `( )` are not;
- Two options as well for `( )`:

(1) Multiple-line version (separated by a **newline**) is

```
(
 <command_1>
 <command_2>
 <command_3>
 )
```

(2) Single-line version (separated by `;` and **whitespace**) is

```
(<command_1> ; <command_2> ; <command_3> )
```

- Don't forget that `( )` does invoke a sub-shell (i.e., like a child process); At the end, the semicolon is an optional to `( )`;

Group Commands

- Differences between { } and ( )

(1) Starts a **sub-shell** or not;

(2) Variable **scope** (visibility): variables in a sub-shell are not visible to its parent shell (parent process). They are local to the child process. However, we can use `export` to transfer the values of variables from the parent shell to a child shell. But a child process cannot export variables back to its parent shell. This is also true for changing directories.

(3) Semicolon (**;**) at the **end** of command: in fact semicolon is a **statement terminator** that tells shell this is the end of the statement. If a command terminates properly by itself, there are no needs to add **;** at the end;

- Advantages of launching multiple sub-shells;

Questions on Sub-shell?

- Sub-shell is like a child process spawned by the parent process (shell); Multiple and concurrent sub-shells are supported;
- **Q1**: Can we safely quit a sub-shell and get back to the parent shell?
- **Q2**: How can I bring the values of variables from a sub-shell back to its parent shell?
- **Q3**: In what cases would be it useful to spawn sub-shells?
- **Q4**: Is there any other benefit to spawn a sub-shell?

- The short answers to **Q2** and **Q3** are to get **files** involved!
- Remember all `STDIN`, `STDOUT`, or `STDERR`, and external files on disks are called **files**;

Special Parameters

Operation	Explanation
`$?`	Exit status of the last command
`$$`	Process ID (keyword)
`$0`	Command name; `$1` is the first argument, etc
`$#`	Number of the positional arguments
`$@`	Expansion of all the positional arguments, a list of every words (`"  "`)
`$*`	Expansion of all the positional arguments a single string containing all of them (`"  "`)

- These are very useful, particularly, in functions, when arguments need to be parsed;

Examples on Control and Logical Operators

- Let's consider a real bash script using `if/fi`, `[[ ]]`, and `case`

```
#!/bin/bash
# how to use case in bash script.  # Jan 30, 2016
echo
 # check $1 for year.
if [[ ($1 -ne "2015") && ($1 -ne "2016") ]] ; then
      echo "Year must be 2015 or 2016!"
      echo "Quit!"
      echo
      exit 1
fi
# select year.
case "$1" in
      2015) echo "==== $1 Calendar ===="; cal $1 ;;
      2016) echo "==== $1 Calendar ===="; cal $1 ;;
esac
exit 0
```

#! is called shebang (shabang or hashbang)

Shell Arrays

- Array variable contains multiple variables, and array index starts from zero;
- Bash supports one-dimensional arrays, and sparse array;
- `$ my_array=("Alice" "Bill" "Cox" "David")`
- `$ my_array[0]="Alice" ; my_array[1]="Bill" ; my_array[5]="John" ; my_array[10]="Collin"`
- Explicit declaration of array: `$ declare -a my_array`
- Referencing an array element `${my_array[1]}`, etc;
- Get the number of arrays elements: `${#my_array[@]}`
- List all arrays elements: `${my_array[*]}` or `${my_array[@]}` (" " might be needed);
- Destroy array variables:
  `$ unset my_array` or `$ unset my_array[2]`

Pattern Matching in Shell

- Pattern matching was designed for
  (1) selecting filenames;
  (2) certain strings satisfying a desired format;

Pattern matching includes
- Globs,
- Extended Globs,
- Regular Expressions (ver. $\geqslant$ `3.0`).

- Globs:
  (1) `*` : matches any strings including the null string;
  `$ echo *` or `$ ls *`
  (2) `?` : matches any single character;
  `$ echo a?` or `$ rm ??`
  (3) `[...]` : matches any one of enclosed characters;
  `$ ls a[123]*.dat` or `$ rm [!a-z]*.dat`
- Bash also provides extra features through Extended Globs;
- By default, these features (extended globs) were turned off;

Arithmetic Operations (`let` and `bc`)

- Arithmetic expansion with `(( ))` [works only for integer operations]: `$ echo $((3-5))` or `$ ((n -= 2))`
- Arithmetic evaluation: `let` (builtin for integers only) or `bc` (external for high-precision floating-point operations);
- Be careful with whitespaces in `let`;
  `$ let t=1 + 4; echo $t` or `$ let t="1 + 4"; echo $t`
- `$ t=3.4 ; t=`echo "$t + 1.2" | bc`; echo $t`
- `$ t=3.434059; t=`echo "scale=5;$t^4" | bc`; echo $t`
- `$ echo "ibase=10;obase=2;256" | bc`
  # convert between decimal and binary numbers
  # (`ibase` = input base, `obase` = output base);
- `$ echo "c(4)" | bc -l` # compute `cos(4)` in radians;
  # `"-l"` loads predefined math libs; the default scale is `20`;

Test Constructs

- Every programming language needs to have good test constructs;
- Bash supports **three** types of test constructs: `test`, `[`, and `[[ ]]` ;
- Builtin commands `test` and `[`, while `[[ ]]` are keywords;
- All test constructs return an exit status: success (`0`) or false (`non-zero` value);
- Bash provides `if/then/fi` construct to support conditional branching;
- However, note Bash `if/then/fi` can be used alone without involving `test` constructs;

```
if grep -q "toys" my_file.dat
    then echo "Mom, I found it."
fi
```

General Form of if/then/fi construct

- Note elif is identical to else if;
- Multi-line version using [:

```
if [ condition_1 ]
then
   command_1
   command_2
elif [ condition_2 ]
then
   command_3
   command_4
else
   command_5
fi
```

- Or using test with relevant options:

```
if test condition_1
```

General Form of `if/then/fi` Construct

- Note `elif` is identical to `else if`;
- Single-line version using `[[ ]]`:

```
if [[ condition_1 ]] ; then
        command_1 ; command_2
                else if [[ condition_2 ]] ; then
        command_3 ; command_4
                else if [[ condition_3 ]] ; then
        command_6 ; command_7
else ; command_8
fi
```

- Bash shell offers comparison operators for both integers and strings; however, note the differences!
- For integer comparison: `-eq` (is equal to); `-ne` (not equal to); `-ge` (means $\geqslant$); `-lt` (less than), etc;

General Form of `if/then/fi` Construct

Operation	Explanation
`-z`	zero-length string (null string)
`-n`	string is not null
`!n`	is not equal to
`<`	is less than
`>`	is greater than

- Similarly to C, Bash uses the ASCII code for string comp.;
- Bash also supports nested `if/then/if` constructs:

```
if [[ condition_1 ]] ; then
    if [[ condition_2 ]] ; then
        command_1 ; command_2
    fi
fi
```

General Form of `if/then/fi` Construct

File Operation	Explanation
`-e`	existence or not
`-d`	directory or not
`-f`	regular file
`-s`	zero-size file
`-nt`	f1 `-nt` f2 (if file f1 is newer than f2)
`-O`	you're the file owner

```
my_file=$1  # $1 is the 1st arg. of the script.
 if [[ -f my_file ]] ; then
    do something here
 else
    do something else here
 fi
```

Loop Constructs

- Bash also supports rich loop constructs so they become more powerful;
- Loop blocks are the repeatable code blocks that having same structures (`for-in-do-done`);
- Bash provides three types of loop constructs: `for`, `while`, and `until` loops; Again `for`, `in`, `while`, `until`, `do`, and `done` are all the bash builtin keywords;

```
for arg in [list]
do
    command_1
    command_2
done
```

- `[list]` can be any valid multiple variables including wild cards (* and ?), or even a valid output from a command;

Loop Constructs

- The other example:

```
for i in {a..z}
   do
      command_1
      command_2
done
```

- Bash also supports C-style three-expression loops;
- Using double parentheses `((  ))`;

```
maxt=200
for (( t=0; t <= maxt; t++ ))
do
      command_1
      command_2
done
```

- The `for`-loop rules are the same as those of C;

Loop Constructs

- `while` loop:
- This needs to be combined with a `test` (`true` or `false`);
- `while-[[ ]]-do-done`

```
a=200 ; b=100
while [[ "$a" -gt "$b" ]]
do
  echo "a = $a"; a=`expr $a - 1`
done
```

- The `while`-loop is similar to those of C: `test` at the loop top;
- This complements the functionality provided by `for` loop, for instance, in the cases of unknown iterations before entering a loop;

Loop Controls

- Similarly to C, we can use the builtin commands `break` and `continue` to control loop behaviors;
- `break` jumps out of loops (break the loop), while `continue` jumps to the next loop iteration within the loop. In both cases, some commands are skipped in the loop;

```
a=1 ; counter=0 ; amax=1000;
while [[ "$a" -lt "$amax" ]]
do
  a=$(($a+2))
  let counter=couter+1
  if [[ "$a" -ge 100 ]] ; then
      break
  fi
done
echo "counter = $counter"
```

Aliases

- Alias is a lightweight shortcut for a long or complicated command (substitution/expansion);
- Bash provides limited support for aliases;
- That doesn't mean we cannot use aliases in bash scripts; we can use them in some simple cases;
- All aliases can be replaced by functions;
- Using both double quotes (`" "`) and single quotes (`' '`) should be fine;

```
$ alias rm='rm -i'
$ alias smic="ssh -XY xiaoxu@smic.hpc.lsu.edu"
```

- Restrictions on aliases:
  (1) Aliases cannot be expanded recursively;
  (2) Aliases would not work in the constructs of if/then, loops, and functions;

Bash Functions

- Bash functions are similar to other languages like functions in C and functions/subroutines in Fortran;
- C-style definition:

```
    name_function ()
      {
        command_1
        command_2
        command_3
      }
```

- Single-line version:

```
funct () { command_1 ; command_2 ; command_3 [;] }
```

- The difference from C: in Bash scripts the function definition must appear earlier than where the function call happens;

Bash Functions

- Local and global variables in bash functions;

```
#!/bin/bash
 name_function ()

   {
# this's a local variable.
   local locl_var=123
# this's a global variable.
         glob_var=456
   }
name_function
echo "locl_var = $locl_var"
echo "glob_var = $glob_var"
```

- Declare local variables through `local`;
- A local variable is invisible outside of function;

Passing Arguments to Bash Scripts

- Bash scripts accept arguments at run-time;
- `$0` stands for the command (including script, function, etc);
- `$1` stands for the first argument passed to the `$0`;
- `$2` stands for the second argument passed to the `$0`;
- `...`;
- `${10}` stands for the 10th argument passed to the `$0`;

Operation	Explanation
`$$`	the PID of the current process
`$?`	the exit code of the last executed command
`$*`	the list of arguments fed the current process
`$#`	how many arguments in `$*`

Function Example: Addition

Below is an example for the function to add two integers:

addition-v0.bash

```
#!/bin/bash

echo $1  $2
a=$1
b=$2
echo $#
function_add() {
function_add=$(($a+$b))
}
function_add
echo "sum =" $function_add
```

- `$ ./addition-v0.bash 12 34`
- Can we make it work better (say, for addition of any numbers, and defensive programming)?

Regular Expressions (REs or Regexs)

- **RE** is a group of characters (including meta characters) that match specified textual patterns in files;
- Why do we need REs? They are needed by the Linux commands/utilities grep, sed, awk, vi, emacs, etc;
- Don't be confused with Bash **Globs**;
- Character set (no metas), anchor set (specifying the positions), and modifiers (range of characters);

Operator	Explanation
.	Matches any single character
?	The preceding item is optional and will be matched $\leqslant 1$ times
*	The preceding item will be matched $\geqslant 0$ times
+	The preceding item will be matched $\geqslant 1$ times

Regular Expressions (REs or Regexs)

Operator	Explanation
`{N}`	The preceding item matched N times
`{N,}`	The preceding item matched $\geqslant N$ times
`{N,M}`	The preceding item matched $\geqslant N$, but $\leqslant M$ times
`-`	represents the range if it's not first or last in a list or the ending point of a range in a list
`^`	Matches the beginning of a line; also represents the characters not in the range of a list
`$`	Matches the empty string at the end of a line
`\b`	Matches the empty string at the edge of a word
`\<`	Match the empty string at the beginning of word
`\>`	Match the empty string at the end of word

- Matching strings form subsets of the specified pattern;

(1) `.` matches any one character, except a newline (line break); The pattern `ad.pt` ⟵ `adapted`, `iadopt`, `ad␣pt`;

(2) `*` matches any number of repeats of the preceding character (including zero occurrence);
The pattern `10na*` ⟵ `10n`, `210nab`, `10naaa`;

(3) `^` matches the beginning of the line.
The pattern `^ed` ⟵ `I finished the editing of files.`

(4) `$` matches a line ending with a particular pattern;
The pattern `ed$` ⟵ `I finished the editing of files.`

(5) `^$` matches blank lines;

(6) `[^a-c]` matches every single character except `a`, `b`, and `c`;

Regular Expressions (REs)

- More examples? Let's consider the editor vi:
  - Delete all blank lines: `:g/^$/d`
  - Search all 2-digit numbers: `\d\d`
  - Search all non-digit words: `\D`
  - Search all whitespaces: `\s`
  - What shall I find if I search: `the*`?
  - Search all 3-letter words: `\s\w\w\w\s`
  - Search all 3-letter words that start with capital letters: `\u\w\w`
- What do we get to match the following patterns? (`\` to escape)
  - `es\+`
  - `gs\=`
  - `s\{2}`
  - `[0-9]`
  - `.`

Regular Expressions (REs)

- Even more examples? Let's consider the editor `sed`:
- GNU `sed` is a `s`tream `ed`itor; In most cases, `sed` is not sensitive to double or single quotes;

```
$ sed "s/RE/SUB/" my_file.dat
$ echo "shell scripting" | sed "s/[si]/?/g"
$ echo "shell scripting 101" | sed "s/[^0-9]/0/g"
```

- A word (`\w`) in `sed` means any combination of lowercases, uppercases, numbers, and underscores (_);

```
$ echo "shell scripting 101" |  \
  sed "s/\w\w\w\w/=/g"
$ echo "shell scripting:  101 (02/17/2016)" |  \
  sed "s/[[:alnum:]]/+/g" # the same as [a-zA-Z0-9]
$ echo "My cat was educated." |  \
  sed "s/\<cat\>/dog/g"
$ echo "egg" | sed "s/e\+/=/g/"
```

The best way to master shell scripting is to write scripts yourself!

Quotes, Whitespaces, Parentheses, and Meta Characters.

Shell Scripting → REs → sed, awk, vi, emacs, . . .

Advanced Bash Scripting Guide, Mendel Cooper (2008).
The Comprehensive List of Bash References.

Questions?

`sys-help@loni.org`