An Introduction to Shell Scripting

Anja Gerbes

Goethe University, Frankfurt am Main
Center for Scientific Computing

August 28, 2018
Assumptions

Before starting, you should...

... know how to use a text editor like emacs or vi/vim

... have basic knowledge of UNIX:

- some basic commands like `ls`, `cd`, ...
- processes, kernel, etc
What is UNIX shell?
Welcome to a new world!

sh  Bourne-Shell

csh  C-Shell

ksh  Korn-Shell

bash  Bourne-Again-Shell

We will restrict ourselves to bash

To find all available shells in your system type following command:

$ cat /etc/shells

Note!

Each shell does more or less the same, with differences in command syntax, or built-in functions, …

To find your current shell type the following command:

$ echo $SHELL
Why shell scripting?

- Need to manage computers remotely?
- Need to perform complex operations on lots of files?
- Need to repeat the same operations on a lot of machines?
Why shell scripting?

- Need to manage computers remotely?
- Need to perform complex operations on lots of files?
- Need to repeat the same operations on a lot of machines?

Shell scripting is the answer!!!
Why shell scripting?

- Need to manage computers remotely?
- Need to perform complex operations on lots of files?
- Need to repeat the same operations on a lot of machines?

Shell scripting is the answer!!!

...or maybe not, BUT
Why shell scripting?

- Need to manage computers remotely?
- Need to perform complex operations on lots of files?
- Need to repeat the same operations on a lot of machines?

... or maybe not, BUT

Shell scripting glues together

- the power of UNIX and
- the power of programming
## What is a shell script?

- A Text File
- With Instructions
- Executable, if wanted
Writing Bash Scripts

- **Shebang**
  
  ```bash
  #!/bin/bash
  ```

- **Comments**
  
  ```bash
  #This text will be ignored
  ```

- **Make script executable**
  
  ```bash
  chmod +x myscript.sh
  ```

- **Execute Script**
  
  ```bash
  ./myscript.sh
  ```

- **Also (no need to turn on x bit)**
  
  ```bash
  bash myscript.sh
  ```
A simple example of shell script with arguments

```bash
#!/bin/bash

#This is a comment
echo "Hello, $1 $2"
echo "Greetings from $0"
echo "Welcome back!"
```

$ bash simple.sh
Hello,
Greetings from simple.sh
Welcome back!

$ bash simple.sh Hans
Hello, Hans
Greetings from simple.sh
Welcome back!

$ bash simple.sh Max Born
Hello, Max Born
Greetings from simple.sh
Welcome back!
Command Line and Exit Status

- The command line is the interface from the shell to an external command (executable).
- The exit value is the interface from the command to the shell.

```bash
$ ls aAa
ls: cannot access aAa: No such file or directory
$ echo $?  
2

But
```
```
$ touch aAa
$ ls aAa
aAa
$ echo $?  
0
```
#!/bin/bash

```bash
echo "What is your name?"
read uname

echo "Welcome $uname"
```
Possible startup files

- `/etc/profile` is executed automatically at login
- The first file found in the list
  - `~/.bash_profile`,
  - `~/.bash_login`, or
  - `~/.profile`

  is executed automatically at login
- `~/.bashrc` is executed by login and nonlogin shells.
Filename Metacharacters

* match any string of zero or more characters
? match any single character
[abc...] match any of the enclosed chars; hypens for ranges ([a-z])
[!abc...] match any chars not enclosed
~ home directory of current user
~name home directory of name
~+ current working dir
~– previous working dir
Tricky issue: see `man bash`, under QUOTING

**Double quotes: "**
Everything between the initial " and the closing " is taken literally, except for

- $ variable substitution will occur
- ` command substitution will occur
- \ it will escape the next character (can also escape ")

**Single quotes: ’**
Everything between the initial ’ and the closing ’ is taken literally

- another ’ cannot be embed a single quoted strings
Quoting

` or $()

Command substitution: expands to what is inside

Examples of quoting

$ echo 'Single quotes "protect" double quotes'
Single quotes "protect" double quotes

$ echo "Well, isn’t that \"special\"?"
Well, isn’t that "special"?

$ echo "You have ‘ls | wc -l’ files in ‘pwd’"
You have 84 files in /home/gerbes

$ x=100
$ echo "The value of \$x is $x"
The value of $x is 100

$ echo "$a"
$a
I/O Redirection

<table>
<thead>
<tr>
<th>fd</th>
<th>Name</th>
<th>Abbr.</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>standard input</td>
<td>stdin</td>
<td>Keyboard</td>
</tr>
<tr>
<td>1</td>
<td>standard output</td>
<td>stdout</td>
<td>Screen</td>
</tr>
<tr>
<td>2</td>
<td>standard error</td>
<td>stderr</td>
<td>Screen</td>
</tr>
</tbody>
</table>

Simple redirection

- `cmd > file` sends output to `file` (overwrite)
- `cmd >> file` sends output to `file` (append)
- `cmd < file` `cmd` takes input from `file`
- `cmd1 | cmd2` a *pipe*: output of `cmd1` is input of `cmd2`
I/O Redirection

More redirection

\[ cmd << \text{text} \] \textit{here document}

\[ cmd >& n \] sends \textit{cmd} output to file descriptor \textit{n}

\[ cmd m>& n \] Same as previous, but output that would normally go to file descriptor \textit{m} is sent to file descriptor \textit{n} instead

\[ cmd 2>file \] sends standard error to \textit{file}, standard output remains the same (screen)

\[ cmd >& file \] sends both standard output and standard error to \textit{file}

\[ cmd &>> file \] appends both standard output and standard error to \textit{file}
I/O Redirection

In practice

$ cat # it takes input from keyboard and output goes to screen (also errors)
hello world
hello world
$ cat > my_dummy_file # now std output goes to a file
hello world, again
$ cat < my_dummy_file # input comes from file; no need to press ctrl-d to exit cat
hello world, again
How does it look like in terms of commands?

```
$ echo "Hello world!" | wc -c
13
```
$ echo This \a very \long \command line.

This is a very long command line.
Variable Assignment

- letters, digits, underscores
- case sensitive
- may not start by a digit
- assignment of variables with the = operator
- no spaces between name and value
- multiple assignments in one line
  
  ```
  name=John lastname=Smith age=99
  ```
- Convention: uppercase names used/set by the shell
- default: all variables are strings
- declare -i
Variable Substitution

```
var=value  sets var to value
${var}    Use value of var
${var:-value} Use var if set, otherwise, use value
${var:=value} Use var if set, otherwise, use value and assign value to var
${#var}  Use the length of var
${!var}  Use value of var as name of variable whose value should be used (indirect reference)
```

```
$ a=CC b=DD A=a
$ echo ${!A}
CC
```
Some Variables

$HOME absolute path of the home directory
$HOSTNAME name of the computer
$PATH list of paths where the executables are looked for
$PWD current working directory
$OLDPWD previous working directory
Some Special Shell-Variables

$0 first word (command name)

$n individual positional arguments on command line

$*, @$ all arguments on command line

$# number of command line arguments

$$ PID of the active shell

$! PID of last background command

$? Exit value of last executed command
Variables and the Environment

$ env
[...variables passed to sub-programs...]

$ NEW_VAR="Yes"

$ echo $NEW_VAR
Yes

$ env
[...PATH but not NEW_VAR...]

$ export NEW_VAR
$ environment
[...PATH and NEW_VAR...]
Arithmetics

- Only Integer Arithmetics
- `let` command:

```
let expressions
(( expressions ))
```

Examples:

```
$ let i=0# variables do not need preceding "$"
$ let i=i+1# spaces not allowed
$ echo $i
1

$ let "i = i + 1" # quotes must be added if expression contains spaces
$ echo $i
2

$ (( i += 1 )) # (( ... )) does quoting for you
$ (( i *= 7 )) # Arithmetic operators taken from the C language
$ echo $i
21
```
$((  )) for Math

$(( ... )) to assign to a variable the result

```
$ a=$(( 1 + 2 ))
$ echo $a
3

$ echo $(( 2 * 3 ))
6

$ echo $(( 1 / 3 ))
0
```
Several ways to get history

- `history` command
- line-edit mode
- `fc` command
- C-shell-style history
line-edit mode

- history treated like a file
- lines can be modified before executing
- set -o emacs or set -o vi

<table>
<thead>
<tr>
<th>emacs</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>up or ctrl+p</td>
<td>previous command</td>
</tr>
<tr>
<td>down or ctrl+n</td>
<td>next command</td>
</tr>
<tr>
<td>ctrl+r</td>
<td>get previous command containing <em>string</em></td>
</tr>
<tr>
<td>ctrl+s</td>
<td>get next command containing <em>string</em></td>
</tr>
</tbody>
</table>
Control Constructs

- if
- for
- while

How do we write conditions in bash?
The easiest way: use the `test` command
Logic

```bash
test condition
[ condition ]
[[ condition ]]
```

- `[ . . ]` and `[[ . . ] ]` must be surrounded by spaces
- `[[ . . ] ]` word splitting and filename expansion disabled

```bash
$ test 1 -lt 10
$ echo $? 0

$ test 1 == 10
$ echo $? 1
```
test

[]

[1 -lt 10]

[]

[[ "this string" != "this" ]]

()]

((1 < 10))

[ -e filename ]

Much more!

see: man test
 Decision Control Constructs

▶ **if** allows the programmer to make a decision in the program based on conditions he specified
▶ If the condition is met, the program will execute certain lines of code
▶ otherwise the program will execute other tasks the programmer specified
▶ different types of conditional statements: file-based, string-based and arithmetic-based conditions
▶ e.g. file-based conditions are unary expressions and often used to examine a status of a file (`-e file` returns true if file exists)

```bash
# see if a file exists
if [ -e /etc/passwd ]
then
  echo "/etc/passwd exists"
else
  echo "/etc/passwd not found!"
fi
```
Looping Control Constructs

- simplify recursive tasks
- optimize any code by providing a way to minimize code
- easier to troubleshoot than unstructured code providing the same output
- types of looping statements: the `for` and `while` loops
# for-in structure

```bash
for i in 1 2 3
do
  echo $i
done
```

# list directory recursively

```bash
for i in /*
do
  echo "Listing $i:"
  ls -1 $i
  read
done
```
C-like Syntax

# syntax of C-style for-loop

```bash
for ((initialization; boolean_test; increment/decrement))
do
  <code>
done
```

# example for C-style for-loop

```bash
LIMIT=10
for ((a=1; a<=LIMIT; a++))
do
  echo -n "$a"
done
```
while separates the initialization, Boolean test and the increment/decrement statement

```bash
# syntax of while-loop

<initialization>
while (condition) do
  <code>
  <increment/decrement>
done
```

```bash
# example for while-loop

a=0; LIMIT=10
while [ "a" -lt "$LIMIT" ] do
  echo -n "$a"
  a=$(( a+1 ))
done
```
Functions

# syntax of functions

name () {
    function’s body
}

# example for functions

fatal () {
    echo "$0: fatal error:", "$@" > &2
    exit 1
}

...

if [ $# = 0 ]
then
    fatal not enough arguments
fi

▶ return to return and exit value to the calling program
▶ exit to really exit
Easy exercises

Exercise 1: back me up!

Write a shell script that backs itself up. The backup’s name should be the original name with a `.back` suffix.

```bash
# Use cat to output the script itself
# Use > to create or append to a file
"cat "$0" > "$0.back"
```
Easy exercises

Exercise 1: back me up!

Write a shell script that backs itself up. The backup’s name should be the original name with a `.back` suffix.

**hint**

Use `cat`
**Exercise 1: back me up!**

Write a shell script that backs itself up. The backup’s name should be the original name with a `.back` suffix.

**hint**

Use `cat`

```
cat "$0" > "$0.back"
```
Exercise 2: reverse

Write a script that reverses the content of a given file given as first argument and writes it to a file appending the .kcab suffix to the original file name.
Easy exercises

Exercise 2: reverse

Write a script that reverses the content of a given file given as first argument and writes it to a file appending the `.kcab` suffix to the original file name.

hint

Use `tac` and `rev`
Easy exercises

Exercise 2: reverse

Write a script that reverses the content of a given file given as first argument and writes it to a file appending the `.kcab` suffix to the original file name.

hint

Use `tac` and `rev`

```
rev $1| tac > $1.kcab
```
Exercise 3: basic argument parsing

Write a shell script that takes 3 arguments and prints them in reverse order. If \(-h\) is given, print also a help message.
Easy exercises

Exercise 3: basic argument parsing

Write a shell script that takes 3 arguments and prints them in reverse order. If `-h` is given, print also a help message.

hint

$1, $2, ...
Easy exercises

Exercise 3: basic argument parsing

Write a shell script that takes 3 arguments and prints them in reverse order. If \(-h\) is given, print also a help message.

hint

$1, $2, ...

```bash
echo "$3 $2 $1"
if [ "$1" = "-h" -o "$2" = "-h" -o "$3" = "-h" ]
then
echo "Some help"
fi
```
Exercise 4: implement a trash

Write a shell script that acts as a *safe delete*. Call it `srm.sh`. Filenames passed as command-line arguments to this script are not deleted, but instead moved to a directory called `~/.TRASH`. Add the following features:

- Upon invocation the script checks the `~/.TRASH` directory for files older than 7 days and permanently removes them.
- If the files are not *gzipped*, the script compresses each file before moving it to the trash.
- Decouple the initial check to another script that should be executed regularly by `cron`. 
Exercise 5: process monitor

Given a process ID (PID) as an argument, this script will check, at user-specified intervals, whether the given process is still running. You may use the `ps` and `sleep` commands.
Write to

▶ hpc-support@csc.uni-frankfurt.de
▶ support@csc.uni-frankfurt.de

in case of general questions about the cluster.

Or directly to us for comments or questions about this course:

frankfurt@hpc-hessen.de

THANK YOU!