Environnements et Outils de Développement
Cours 4 — Shell scripting basics

Stefano Zacchiroli
zack@pps.univ-paris-diderot.fr

Laboratoire PPS, Université Paris Diderot

2013–2014

URL http://upsilon.cc/zack/teaching/1314/ed6/
Copyright © 2013–2014 Stefano Zacchiroli
License Creative Commons Attribution-ShareAlike 4.0 International License
http://creativecommons.org/licenses/by-sa/4.0/deed.en_US
BY-SA

Outline

1. Program execution
2. Command composition
3. Expansion
4. Control structures

Shell and shell script

A shell is a program that sits between yourself and the services offered by the operating system (e.g. running programs).

- shells can be used interactively
  - with read-eval-print loops
  - ...you've done this thousands of times already!
- shells can be programmed
  - to develop real programs (AKA "scripts")
  - in a programming language known as shell script
    - (in fact, each shell has its own shell scripting language; we will use the Bash shell—`/bin/bash`—in these slides)

Scripts are fundamental tools to automate tasks (system administration, development, day-by-day productivity, etc).
Learn shell script, love shell script. You will be more productive.

Hello, world !

A simple shell script

```
#!/bin/bash
echo -n Hello,
echo " World!"
```

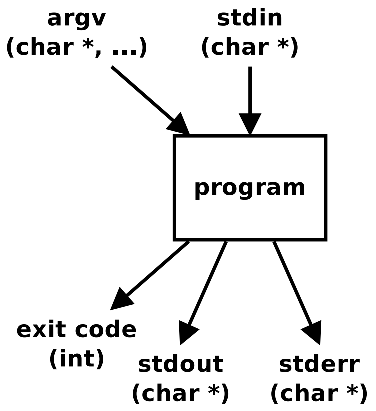
```
$ ls -l hello
-rw-r--r-- 1 zack zack 42 feb 18 11:15 hello
$ cat hello
#!/bin/bash
echo -n Hello,
echo " World!"
$ chmod +x hello
$ ls -l hello
-rwxr-xr-x 1 zack zack 42 feb 18 11:15 hello
$ ./hello
Hello, World!
$
```

Sommaire

1. Program execution
2. Command composition
3. Expansion
4. Control structures

The "API" of general UNIX programs

Program execution on UNIX implements a very simple API.



Programs :

- are executed passing command-line arguments (strings)
  - 1st argument = program name
- return a numeric exit code
- read user input via standard input
  - usually : the keyboard
- write "ordinary" output to standard output
  - usually : the screen
- write "extraordinary" output (e.g. error messages) to standard error
  - usually : the screen too

Program execution — examples

```
$ ls
code
main.tex
Makefile
$
```

argv = {"ls"}
exit code = 0 (== 0, i.e. *“everything is fine”*)
stdin = ""
stdout = "code\nmain.tex\nMakefile\n"
stderr = ""

Program execution — examples (cont.)

```
$ ls doesn-t-exist
ls: cannot access doesn-t-exist: No such file or
directory
$
```

argv = {"ls", "doesn-t-exist"}
exit code = 2 (!= 0, i.e. *“we’ve got a problem”*)
stdin = ""
stdout = ""
stderr = "ls : cannot access doesn-t-exist : No such [. . .]\n"

Program execution — examples (cont.)

```
$ cat
foo
foo
bar
bar
baz
baz
CTRL-D
$
```

argv = {"cat"}
exit code = 0
stdin = "foo\nbar\nbaz\n"
stdout = "foo\nbar\nbaz\n"
stderr = ""

Program execution — examples (cont.)

```
$ ./hello one two three
Hello, World!
$
```

argv = {"./hello", "one", "two", "three"}
exit code = 0 (default exit code)
stdin = ""
stdout = "Hello, World!\n"
stderr = ""

Note : hello doesn't *use* its command line arguments, but that didn't stop us from passing them. Other programs will be less forgiving than hello.

Sommaire

1. Program execution
2. Command composition
3. Expansion
4. Control structures

Command composition

Based on the main ingredients of the program execution API—exit codes and standard I/O channels—we can compose shell commands in interesting ways :

conditional execution executing a command only if another one "succeeded" (exit code == 0) or "failed" (exit code != 0)

redirection using the standard output of a program as the standard input of another one (or diverting stdin/stdout to/from a file)

Conditional execution

- *cmd1 && cmd2* — execute *cmd2* iff *cmd1* succeeds
- *cmd1 || cmd2* — execute *cmd2* iff *cmd1* fails

success means : exit code == 0 ; *failure* means : exit code != 0

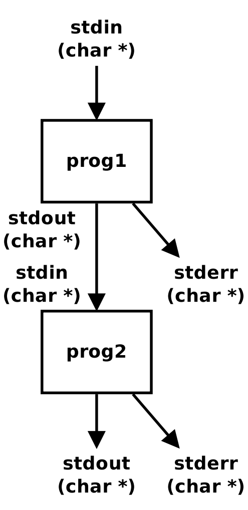
```
$ ls && echo EUREKA
code
main.tex
Makefile
EUREKA
$ ls doesn-t-exist && echo eureka
ls: cannot access doesn-t-exist: No such file or directory

$ ls doesn-t-exist || echo DOH
ls: cannot access doesn-t-exist: No such file or directory
DOH
$
```

Pipeline

cmd1 | *cmd2* — execute both *cmd1* and *cmd2*
+ make *cmd1*'s stdout be *cmd2*'s stdin

```
$ ls | wc --lines
3
$ ls | sort --reverse
Makefile
main.tex
code
$ cat /etc/passwd | cut -f 1 -d : | sort
[...]
backup
bin
daemon
games
[...]
root
[...]
$
```



Redirection to/from file

- *cmd* > *file* — execute *cmd*, writing its stdout to *file*
- *cmd* < *file* — execute *cmd*, using *file* as its stdin

```
$ cat < hello
#!/bin/bash
echo -n Hello,
echo " World!"
$ echo "Hello, World!" > test-redir.txt
$ cat test-redir.txt
Hello, World!
$
```

```
$ cat < hello | sort --reverse > not-a-script
$ cat not-a-script
echo " World!"
echo -n Hello,
#!/bin/bash
$
```

Sommaire

1. Program execution
2. Command composition
3. Expansion
4. Control structures

Expansions

As a language, shell script has a less clear parsing/execution distinction than many other languages. Instead, evaluation of shell scripts goes through a series of expansions (similar to meta-programming in other languages) that iteratively transform script statements.

see : Shell Command Language, *The Open Group Base Specifications Issue 6*, IEEE Std 1003.1, 2004 Edition. `http://pubs.opengroup.org/onlinepubs/009695399/utilities/xcu_chap02.html`

Some expansion phases :

- variable expansion
- wildcard expansion
- arithmetic expansion
- command expansion
- ...

Variable expansion

```
$ year=2097
$ galaxy="far far away"
$ echo "It’s year $year, in galaxy ${galaxy}..."
It’s year 2097, in a galaxy far far away...
```

Cmdline arguments can be accessed via special variables $0, $1, ...

```
$ cat hello2
#!/bin/bash
echo "Hello, $1 World!"
$ ./hello2
Hello,  World!
$ ./hello2 wonderful
Hello, wonderful World!
```

Variable expansion

```
$ year=2097
$ galaxy="far far away"
$ echo "It's year $year, in galaxy ${galaxy}..."
It's year 2097, in a galaxy far far away...
```

Cmdline arguments can be accessed via special variables $0, $1, ...

```
$ cat hello2
#!/bin/bash
echo "Hello, $1 World!"
$ ./hello2
Hello,  World!
$ ./hello2 wonderful
Hello, wonderful World!
```

Exercise : explain the following behavior

```
$ ./hello2 truly wonderful
Hello, truly World!
$ ./hello2 "truly wonderful"
Hello, truly wonderful World!
```

Wildcard expansion

Specific meta-characters, like * and ?, are used as shortcuts for space-separated lists of file names. Collectively, those characters are called wildcards and are subject to wildcard expansion.

- * — any number of (non-/) characters that appear in the name of existing files
- ? — a single character, as above

```
$ ls
auto   code      main.log  main.out  main.pdf.mk  main.tex  main.vrb
build  main.aux  main.nav  main.pdf  main.snm     main.toc  Makefile
$ echo *
auto build code main.aux main.log main.nav main.out main.pdf main.pd
$ echo main.p*
main.pdf main.pdf.mk
$ echo main.t??
main.tex main.toc
$ echo foo*
foo*
```

Arithmetic expansion

`$(( expr ))` — evaluate *expr* as an arithmetic expression and expand to its value

```
$ year=2037
$ echo $(( $year + 1 ))
2038
$ year=$(( $year * 2 - 42 ))
$ ls -l $(( $year / 37 )).*
-rw-r--r-- 1 zack zack 5 feb 18 15:42 108.cfg
-rw-r--r-- 1 zack zack 4 feb 18 15:42 108.txt
```

Arithmetic expansion

`$(( expr ))` — evaluate *expr* as an arithmetic expression and expand to its value

```
$ year=2037
$ echo $(( $year + 1 ))
2038
$ year=$(( $year * 2 - 42 ))
$ ls -l $(( $year / 37 )).*
-rw-r--r-- 1 zack zack 5 feb 18 15:42 108.cfg
-rw-r--r-- 1 zack zack 4 feb 18 15:42 108.txt
```

Exercise

list the expansion phases that the commands above go through

Command substitution

- *$(cmd)* — executes *cmd* and substitute the $(cmd) string with cmd's stdout before continuing execution
- *'cmd'* — same (but harder to nest)

```
$ date
Mon Feb 18 15:49:31 CET 2013
$ date +%s
1361198971
$ now=$(date +%s)
$ echo $now
1361198971
$ date -d @1361198971
Mon Feb 18 15:49:31 CET 2013

$ one_week_later=$(date -d @$(( $(date +%s) + 60*60*24*7 )))
$ echo $one_week_later
Mon Feb 25 15:54:22 CET 2013
```

(see date(1) for a simpler way to do the same)

Sommaire

1 Program execution

2 Command composition

3 Expansion

4 Control structures

If/then/else

- `if` *cmd1* `;` `then` *cmd2* `;` `else` *cmd3* `;` `fi` — execute *cmd1* then, depending on its exit code, execute either *cmd2* (*cmd1* succeeded) or *cmd3* (*cmd1* failed)
- multiline syntax :

```
if cmd1 ; then
    cmd2
else
    cmd3
fi
```

Note : as it happened for conditional composition, it is the exit code of the guard that determines which branch is executed not, e.g., its stdout/stderr.

test

The `test` program is very useful in if/then/else guards. Some noteworthy usages :

- `test -f` *file* — true (i.e. exit code == 0) if *file* exists
- `test -z` *string* — true if *string* is the empty string
- `test` *s1* `=` *s2* — true if *s1* and *s2* are equal as strings
- `test` *x* `-lt` *y* — true if *x* is smaller (less than) *y*, when interpreted as integers
  - other integer comparison operators : -eq, -ne, -ge, -gt, -le

Tip

`!` *cmd* — logically negate the exit code of *cmd*

test — examples

```
$ test -f annoying-file.txt && rm annoying-file.txt
$ ! test -f file-i-need.txt && echo foo > file-i-need.txt
```

```
$ cat hello3
#!/bin/bash
if test -z "$1" ; then
    echo "Usage: hello3 MESSAGE"
    exit 1
fi
echo "Hello, $1 World!"
$ ./hello3
Usage: hello3 MESSAGE
$ ./hello3 wonderful
Hello, wonderful World!
```

test — examples (cont.)

```
$ cat debug
#!/bin/bash
debug="yes"
if test "$debug" = "yes" ; then echo "D: enter script" ; fi
echo "Hello, World!"
if test "$debug" = "yes" ; then echo "D: exit script" ; fi
$ ./debug
D: enter script
Hello, World!
D: exit script
```

test — examples (cont.)

```
$ cat debug
#!/bin/bash
debug="yes"
if test "$debug" = "yes" ; then echo "D: enter script" ; fi
echo "Hello, World!"
if test "$debug" = "yes" ; then echo "D: exit script" ; fi
$ ./debug
D: enter script
Hello, World!
D: exit script
```

Always add quotes around `test` operands ! I.e. don't do this :

```
if test $debug = "yes" ; then echo "D: enter script" ; fi
```

Why ?

test — examples (cont.)

```
$ cat debug
#!/bin/bash
debug="yes"
if test "$debug" = "yes" ; then echo "D: enter script" ; fi
echo "Hello, World!"
if test "$debug" = "yes" ; then echo "D: exit script" ; fi
$ ./debug
D: enter script
Hello, World!
D: exit script
```

Always add quotes around `test` operands ! I.e. don't do this :

```
if test $debug = "yes" ; then echo "D: enter script" ; fi
```

Because : (exercise : explain why the error happens)

```
$ debug=""
$ if test $debug = "yes" ; then echo "D: enter script" ; fi
-bash: test: =: unary operator expected
```

A nicer syntax for `test`

Hack

- `/usr/bin/[` — is a symbolic link (i.e. an alias) to `test`
- `]` can be safely added at the end of `test`'s expressions

What gives ?

A nicer syntax for test

Hack

- /usr/bin/[— is a symbolic link (i.e. an alias) to test
-] can be safely added at the end of test's expressions

```
if [ -z "$1" ] ; then
    echo "Usage: hello3 MESSAGE"
    exit 1
fi

if [ "$debug" = "yes" ] ; then
    echo "D: enter script"
fi

if ! [ -f "$datafile" ] ; then
    echo "init:0" > "$datafile"
fi
```

- see test(1) manpage for more details

grep — the swiss army knife of text search

`grep` *`regexp file...`* — search all given *files* for lines that matches the regular expression *regexp*

- output matching lines on stdout
- exit code is 0 iff at least one matching line is found

- you know regular expressions already!
- see regex(7) manpage for the actual syntax

grep — examples

```
$ cat hamburger.txt
bread
lettuce
tomato
meat
cheddar
bacon
bread

$ grep '^b' hamburger.txt
bread
bacon
bread
$ grep 'd$' hamburger.txt
bread
bread
$ grep '[ht]' hamburger.txt
lettuce
tomato
meat
cheddar
```

grep — examples (cont.)

```
$ cat onion-check
#!/bin/bash
recipe="hamburger.txt"
if ! grep --quiet '^onions$' "$recipe" ; then
   echo "Oops! no onions..."
   echo "onions" >> "$recipe"
   echo "Fixed! Onions are now safely(?) back in"
fi

$ ./onion-check
Oops! no onions...
Fixed! Onions are now safely(?) back in
$ grep onion hamburger.txt
onions
$ ./onion-check
$
```

Alternative (and more cumbersone) check :

```
if ! [ -z "$(grep '^onions$' $recipe)" ] ; then
```

For/foreach loops

- `for` *`var`* `in` *`list`*`...; do` *`cmd`*`; done` — loop over *list*. . . , binding at each iteration *var* to the current value. At each iteration, execute *cmd* (that usually uses $var to reference *var*).
- multiline syntax

```
for var in list ... ; do
    cmd
done
```

Pure for loops can be built using command substitution and the seq program :

```
$ for i in `seq 1 5` ; do echo $i ; done
1
2
3
4
5
```

For/foreach loops — example

```
$ cat explain-recipe
#!/bin/bash
if [ -z "$1" ] ; then echo "Usage: $0 RECIPE" ; exit 1 ; fi
recipe="$1"
step="0"
for ingredient in $(cat $recipe) ; do
    step=$(( $step + 1 ))
    echo "step #${step}: add $ingredient"
done
echo "All done: congratulations!"
```

```
$ ./explain-recipe hamburger.txt
step #1: add bread
step #2: add lettuce
step #3: add tomato
step #4: add meat
step #5: add cheddar
step #6: add bacon
step #7: add bread
All done: congratulations!
```

While loops

- `while` *cmd1* `;` `do` *cmd2* `;` `done` — execute *cmd2* repeatedly, as long as *cmd1* succed (usual semantics : test first, then execute)
- multiline syntax

```
while cmd1 ; do
    cmd2
done
```

```
i=$1
fact=1
while [ $i -gt 0 ] ; do
   fact=$(( $fact * $i ))
   i=$(( $i - 1 ))
done
echo $fact
```

While true. . .

Tip

The program true does nothing and return a 0 exit code.
It can then be used to write infinite loops.

```
$ while true ; do echo "I’m bored" ; done
I’m bored
I’m bored
I’m bored
I’m bored
I’m bored
I’m bored
I’m bored
I’m bored
I’m bored
I’m bored
[...]
```

While/read loops

read *var* — read a line from stdin and store it in variable *var*

Poor men's cat :

```
$ while read line ; do echo $line ; done
foo
foo
bar
bar
baz
baz
CTRL-D
$
```

References

Mendel Cooper
Advanced Bash-Scripting Guide : An in-depth exploration of the art of shell scripting
http://tldp.org/LDP/abs/html/

Arnold Robbins and Nelson H.F. Beebe
Classic Shell Scripting
O'Reilly Media, 2005.

GNU
bash(1) manual page
http://man.cx/bash