

Programming in Shell 1

Command-line parsing and meta-characters.

Command execution.

Shell variables.

Jan Trdlička



Czech Technical University in Prague, Faculty of Information Technology
Department of Computer Systems

- 1 Shell behavior
- 2 Command-line parsing
 - Quoting
 - Comment
 - Command line splitting
 - Special characters
 - Word splitting
 - File name substitution
 - Input/output redirection
- 3 Command Execution
- 4 Shell Variables

Example: variable setting and simple command

Variable setting

separators
(space / TAB)

end of line
(new line)

The original line:

```
a=Hello      b=world
```

Line after parsing:

```
a=Hello      b=world
```

1st word
variable definition

2nd word
variable definition

Simple command

separators

separators

end of line

The original line:

```
echo      $a      $b
```

Line after parsing:

```
echo      Hello      world
```

1st word
command

2nd word
1st argument

3rd word
2nd argument

Shell behavior

• Variables

```
A="Hello world"
```

```
B="Number of logged-in users:  $(finger | tail -n+2 | wc -l)"
```

• Commands

```
ls -la /
```

```
echo "$A"
```

```
finger | tail -n+2 | wc -l
```

```
:(){ [ $1 -gt $2 ] && echo $1 || echo $2; }; : 1 3; : 2 1
```

• Meta-characters

- Characters that represent something other than its literal meaning (depending on the context).
- #, " ", ' ', \, \$, |, {}, (), [], ...

- Shell repeatedly performs the following steps
 - ① Command-line reading.
 - ② Command-line parsing.
 - ③ Variable setting and command execution.
- During the command-line parsing, the meta-characters are interpreted/replaced by the shell in the "exact" order.
- **Command-line parsing order**
 - ① Quoting: `\`, `'...'`, `"..."`.
 - ② Comments: `#`.
 - ③ Command line splitting: `;`, `&`, `|`.
 - ④ Special characters: `~`, `'...'`, `$`, `$(...)`, `$((...))`, `{...}`.
 - ⑤ Word splitting: space, tab, new-line.
 - ⑥ Pathname expansion: `*`, `?`, `[...]`, `[!...]`, `[^...]`.
 - ⑦ I/O redirection: `<`, `>`, `>>`, `<<END`.
 - ⑧ Options and arguments setting.

Quoting

- Turning off the special meaning of meta-characters.
- **Backslash **
 - It turns off the special meaning of the following character.

```
echo \$HOME = $HOME
```

- **Apostrophes '...'**
 - They turn off the special meaning of all characters between them (except apostrophe).

```
echo 'Meta-characters: $ \ * < > | & ; ...'
```

- **Double quotes "..."**
 - They turn off the special meaning of all characters except
 - `\`,
 - `$variable`, `$(cmd)`, `$((exp))`,
 - `'cmd'`.

```
echo "\$HOME is $HOME." # variable substitution
```

```
echo "Today is $(date +%A)" # command substitution
```

```
echo "Today is 'date +%A' "
```

```
echo "1 + 1 = $((1+1))" # arithmetic expansion
```

- Hashtag #

- Line/word beginning with a # indicate that the # and a text after is comment.
- Example of script

```
#!/bin/bash

# Example of script

uname -s    # OS
uname -m    # hw architecture

echo Hello world#this is not comment
```

Command line splitting

- Simple command

- Sequence of optional variable assignment followed by blank-separated words.

```
A="Hello world"
```

- Command name followed by options and arguments.

```
ls -l /etc
```

- Pipeline

- Sequence of one or more commands separated by the character `|`.
- The standard output of the previous command is redirected as the standard input of the next command.

```
ls -a | wc -w
```

- List

- Sequence of one or more pipelines separated by one of the operators `;`, `&`, `&&` or `||`, and optionally terminated by one of `;`, `&`, `newline`.

```
printf "Number of files in $PWD:" ; ls -a | wc -w
```

- Compound command

- `(list)`, `{list}`, `((expression))`, `[[expression]]`, `for`, `while`, `until`, `if`, `case`, ...

Command line splitting – examples

- `cmd1 ; cmd2`

- Commands are executed sequentially; the shell waits for each command to terminate in turn.

```
date ; sleep 2 ; date
```

- `cmd1 & cmd2 &`

- The shell executes command in parallel.

```
date & sleep 2 & date &
```

- `cmd1 | cmd2`

- The standard output of `cmd1` is connected via a pipe to the standard input of `cmd2`.
- Commands are executed in parallel.
- The return status of a pipeline is the exit status of the last command.

```
ls -a | wc -w
```

- `(cmd1 ; cmd2)`

- Commands are executed sequentially by the subshell.

```
ps ; ( ps ; ps ; ) > out.txt
```

```
date ; ( LC_ALL=zh-TW.UTF-8 ; date ) ; date
```



Command line splitting – examples

- `{ cmd1 ; cmd2 ; }`

- Commands are executed sequentially by current shell.

```
ps ; { ps ; ps ; } > out.txt
```

```
date ; { LC_ALL=zh-TW.UTF-8 ; date } ; date  
LC_ALL=C
```

- Exit status/return code

- Number passed from a child process to a parent process when it has finished executing.
- Meaning of exit code
 - Success: 0
 - Error: 1,2,3,..., 255

- `cmd1 && cmd2`

- `cmd2` is executed if and only if `cmd1` returns an exit status of zero.

```
date && echo "Everything is OK"
```

- `cmd1 || cmd2`

- `cmd2` is executed if and only if `cmd1` returns a non-zero exit status.

```
date foo || echo "Bad argument"
```

Special characters

- **Brace expansion**

- Mechanism by which arbitrary strings may be generated.
- **String list:** pattern $\{s_1, s_2, \dots, s_n\}$ is replaced by strings s_1, s_2, \dots, s_n .

```
$> echo {aa,bb,cc}  
aa bb cc
```

```
$> echo X{aa,bb,cc}Y  
XaaY XbbY XccY
```

```
$> echo {a,b}{1,2}.{c,txt}  
a.c a.txt a1.c a1.txt a2.c a2.txt b.c b.txt b1.c b1.txt  
b2.c b2.txt
```

- **Range list:** pattern $\{x..y[..incr]\}$ is replaced by range of values (x and y are either integers or single characters, and incr, an optional increment, is an integer).

```
$> echo {1..10}  
1 2 3 4 5 6 7 8 9 10
```

```
$> echo {1..10..2}  
1 3 5 7 9
```

Special characters

- Home directory abbreviation by tilde

- `~` represents a home directory of the current user.
- `~username` represents a path to home directory of the given user.

- Variable substitution

- Shell replaces the variable substitution with the content of the variable.
- `$variable` or `${variable}`

- Command substitution

- Shell executes a command and replaces the command substitution with the standard output of the command.
- `' cmd ' ...` old syntax
- `$(cmd) ...` better syntax

- Arithmetic expressions

- Shell evaluates an arithmetic expression and replaces the arithmetic expression with the result.
- `$((expression))`

Word splitting

- Shell splits the content of command-line into words.
- Default word separators are
 - `space`,
 - `tab`,
 - `new-line`.
- Default word separators can be change by shell variables `IFS`.

```
#!/bin/bash

OLDIFS="$IFS"                # remember current value of IFS
IFS=":"                      # set new value of IFS

# Read lines from the file /etc/passwd
while read name foo uid gid comment home shell
do
    echo "name = $name  uid = $uid"
done < /etc/passwd

IFS="$OLDIFS"                # set previous value of IFS
```

File name substitution

- Asterisk *

- Matching zero or more characters, except the leading . (period) of a hidden file.

```
ls -l *.txt *.c
```

- Question mark ?

- Matches exactly one character, except the leading . (period) of a hidden file.

```
ls -l *.???
```

- Square brackets []

- Matching one character in the set or in the range, except the leading . (period) of a hidden file.

```
ls -l file[0-9].html  
ls -l file[367].html
```

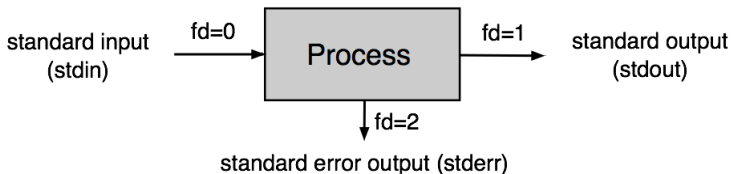
- Square brackets [^] or [!]

- Matching one character not in the set or in the range, except the leading . (period) of a hidden file.

```
ls -l [^A-Z]*
```

Input/output redirection

- File descriptor is an abstract indicator for accessing a file (0,1,2,...).
- Every process has 3 standard POSIX file descriptors by default
 - 0 – standard command input (stdin)
 - 1 – standard command output (stdout)
 - 2 – standard command error output (stderr)



- New process inherits file descriptors from his parent process by default.
- User can redefine/redirect every file descriptor by special characters.
- I/O redirection is evaluated from left to right.

Input/output redirection

- `cmd < file`

- Shell executes `cmd`, using `file` as the source of the standard input.

```
wc -l < /etc/passwd
```

- `cmd > file`

- Shell executes `cmd`, placing the standard output in `file`.

```
date > out.txt  
date > out.txt
```

- `cmd >> file`

- Shell executes `cmd`, appends the standard output to the end of the `file`.

```
date > out.txt  
date >> out.txt
```

- `cmd << string`

- Here-document, shell reads lines until the line beginning with the given string, then all lines are sent to standard input.

```
cat > out.txt <<END  
> 1. line  
> 2. line  
> END
```



Input/output redirection

- `cmd 2> file`

- Shell executes `cmd`, placing the standard error output in the file.

```
ls /etc /XYZ > out.txt 2> err.txt
ls /etc /XYZ > out.txt 2> /dev/null
```

- `cmd >&n`

- Shell executes `cmd`, placing the standard output in file defined by file descriptor `n`.

```
ls /etc /XYZ > out.txt 2>&1
```

- `cmd m>&n`

- Shell executes `cmd`, placing the output defined by file descriptor `m` in file defined by file descriptor `n`.

```
ls /etc /XYZ > out.txt 2>&1
```

- The command line is scanned twice by the shell.
- First, the shell interprets the command line when it passes to the eval command.
- Then shell interprets it a second time as a result of executing the eval command.

```
~> unset B ; A='$B' ; B=date ; echo $A  
$B
```

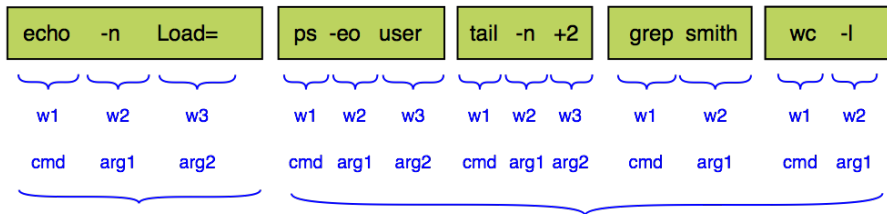
```
~> unset B ; A='$B' ; B=date ; eval echo $A  
date
```

Example: Command Line Parse Order

The original line:

```
echo -n "Load=" > l.txt ; ps -eo user | tail -n +2 | grep $USER | wc -l >> l.txt
```

Line after parsing:



This command is executed first.
The input is from keyboard and
the output is written to file.

This group of commands are executed in parallel after
the termination of the first command.
The output of every command is redirected like input of the next one.
Only the output of the last command is added to the end of file.

Command Execution

- Which program will be executed?
 - Command `type` displays information about command type.

```
type date
type printf
```

- Absolute path to command

```
/usr/bin/echo "Hello world"
```

- Relative path to command

```
echo "Hello world"
```

- Shell function
- Shell built-in command
- Program in filesystem
 - Shell variable `PATH` contains list of directories.
 - The shell makes search through each of these directories, one by one, until it finds a directory where the executable program exists.

Shell Variables

- Local shell variable

- The variable is define only in the current shell.
- Definition: `NAME=VALUE`

- Environment shell variable definition

- The variable is define in the current shell and in all its descendants.
- Definition: `export NAME="VALUE"`

- How to print value of variable?

- By command `echo "$NAME"`.

- How to print list of all variables (locale + environment)?

- By command `set`.

- How to print list of environment variables?

- By command `env`.

Built-in shell variables

- They have special meaning for shell.

HOME	home directory
PWD	current working directory
PS1	prompt definition
PATH	list of directories for command searching
0	program name
1, ..., 9	command line arguments
#	number of command line arguments
\$	process ID of current shell
?	exit status of the last executed command

- They are described in manual page of the shell.

```
~> man bash  
/^ *Shell Variables
```