Shell Syntax Basics

Marek Kozłowski

Faculty of Mathematics and Information Sciences
Warsaw University of Technology

Outline of the Lecture

Outline of the lecture

Shells and Commands

- Most commands are implemented as small, standalone programs.
- Programs' syntax is shell-independent and up to respective authors (although some conventions do exist).
- File names, device names, streams, etc. are shell-independent too.
- Shells my differ on:
  - configuration files,
  - additional features,
  - some operators and special characters,
  - variable defining and handling,
  - built-in commands,
  - control statements.
- Numerous shells may be installed and offered to users at the same time.

Command Prompt

- Command prompt (in short: *prompt*) is some string followed by a blinking cursor that indicates shell's readiness to interact with a user.
- By convention prompts of normal users end with the dollar ($) or percent (%) while the root's prompt ends with the hash (#) symbols.

  ✍ *Default prompts may include user name, host name etc. and may be coloured:*

  Default prompts for users and the root in Gentoo Linux

  ```
  me@myhost ~ $
  myhost ~ #
  ```

- Throughout this course we will use the $ and the # symbols in our examples to indicate commands executed by any user and the root user respectively.

Entering Commands

- The following general rules apply to any Unix shell:
  - Command names, file names, options, variables, etc. are case-sensitive.
  - Commands should fit into single lines.
  - Single backslash (\) at the end or non-matched quotes allow continuing a command on the next line.
  - Hash (#) forces all remaining characters on that line to be ignored (commented out).
  - Sequences of blanks are used as word (parameter) separators.
  - There are ways of protecting literal meanings of all special characters.

Bourne Shell vs C-shell

- *Bourne shell* (*sh*):
  - default AT&T shell released before commercialization of AT&T Unix,
  - syntax influenced by ALGOL 68,
  - preferred for scripting / programming.
- *C shell* (*csh*):
  - BSD shell with some improvements (history, aliases, file name completion etc.),
  - set as a default shell in BSDs,
  - coherent, C-like syntax,
  - nowadays usually replaced with an improved version: *tcsh*.

Modern Shells

- *Bourne shells* (sh-compatible shells):
  - *Korn shell* (*ksh*) – sh extension with some csh features included,
  - *Bourne-again shell* (*bash*) – *bashing together the features of sh, csh and ksh* (see the next slide),
  - *zsh* – assumed to be an improvement of ksh (or bash); feature-rich but complex,
- *fish* – friendly interactive shell; not fully sh- nor csh-compatible,
- *tcsh* – improved csh shell (usually supersedes csh).

Bourne-again (or: Born-again) Shell (bash)

Bash key features:

- part of the GNU project,
- default shell for almost all Linux/GNU distributions and Mac OS X (since 10.3),
- available under most Unix-like systems,
- POSIX compliant,
- Bourne shell (sh) backward compatible,
- incorporates useful Korn shell (ksh) and C shell (csh) extensions.

We assume bash as a default shell throughout this course, however, unless otherwise stated, the informations refer to any shell.

Outline of the lecture

Commands

- As a command name we may type in:
  - an internal shell's command (a *built-in*),
  - a name of a *binary* program,
  - a name of a *script* (an executable ASCII file that needs to be interpreted by an external program).
- From the user's point of view built-ins, binaries and scripts are undistinguishable.
- If there are shell built-ins and programs with the same names then built-ins take precedence.
- Shell searches some list of directories for executables (programs and scripts). The first one found is then executed.

Command Syntax

Command syntax (commonly accepted convention)

```
$ command [options] [filenames]
```

- By commonly accepted convention options can be given in any sequence, however they should precede file names.
- By the same convention two formats for options are used:
  - short options (POSIX syntax),
  - long options (GNU syntax).

Short Options (POSIX)

- A short option consists of a dash (`-`) followed by a single letter.

  ✍ *In the following example we list (`ls`) the content of the `/etc` directory using long format (`-l`):*

  Listing a directory contents using a long format

  ```
  $ ls -l /etc
  ```

- Short options can be grouped.

  ✍ *The following commands are equivalent (the `-A` option includes hidden files for listing):*

  Listing all files in a directory using a long format

  ```
  $ ls -l -A /etc
  $ ls -A -l /etc
  $ ls -lA /etc
  $ ls -Al /etc
  ```

Key-value Options

- Some options require additional values.

  ✍ *We display the `/etc/passwd` file content sorted. We use the colon symbol as a field separator (`-t :`), we sort by the 3rd field (`-k 3`) and apply numerical sorting (`-n`):*

A file sorted numerically by the third field (colon separated)

```
$ sort -t : -k 3 -n /etc/passwd
```

- Key-value options can be grouped provided that values directly follow the options they refer to and no ambiguity occurs.

  ✍ *The following commands are equivalent:*

A file sorted numerically by the third field (colon separated)

```
$ sort -t : -k 3 -n /etc/passwd
$ sort -t: -k3 -n /etc/passwd
$ sort -n -k3 -t: /etc/passwd
$ sort -nk3 -t: /etc/passwd
```

Long Options (GNU)

- Long options are specified as: `--option-name` .
- Most GNU commands define equivalent short and long options. In that case they may be used interchangeable.

  ✍ *The following commands are equivalent:*

  Listing all files in a directory

  ```
  $ ls -A /etc
  $ ls --almost-all /etc
  ```

- Short and long options can be used together and in any sequence but long options cannot be grouped.

  ✍ *In this example we use both short and long options:*

  Listing all files in a directory using a long format

  ```
  $ ls -l --almost-all /etc
  ```

Long Key-value Options

- Long key-value options are specified as: `--option-name=value` (IMPORTANT: no blanks must precede nor follow the '=' symbol).
- Mandatory values for short options are mandatory for long options too.

The following commands are equivalent:

A file sorted numerically by the third field (colon separated)

```
$ sort -t : -k 3 -n /etc/passwd
$ sort --field-separator=: --key=3 --numeric-sort
/etc/passwd
```

Standard GNU Options

- Programmers are encouraged to define long options equivalent to short ones.
- GNU users may assume that two long options are defined for each command. Those are:
  - `--help`,
  - `--version`.

File Names and Options – Ambiguity

- Options should precede file names.
- Usually the first command parameter which doesn't start with the dash is considered to open a file names list.
- Double dash (`--`) may be used as an explicit end-of-options marker in case of ambiguity.

`touch` creates an empty file of a given name if it doesn't exist. We'd like to create a file named `--help` :

Double dash as an end-of-options marker

```
$ touch --help
  help on the 'touch' command
$ touch -- --help # '--help' is a file name
```

File Names

- If it makes any sense a file name can be substituted with a file or directory list.
- If a directory name is missing many commands assume *this* (*current*, *working*) directory.
- Numerous commands in lack of needed file name parameters operate on standard input (default: keyboard) for reading and standard output (default: screen) for writing.

Outline of the lecture

Bash History

- Bash keeps the history of user's commands. By pressing the ↑ and ↓ keys users are able to quickly navigate through it.
- The `history` displays enumerated list of recent commands. Then we can recall a command by specifying its number: `!`*number* . The shortcut: `!!` refers to the latest command.

✍ *The last two commands are equivalent:*

Recalling a command by number

```
$ history
  1 ls -lA /etc
  2 sort -t : -k 3 -n /etc/passwd
$ !2
$ sort -t : -k 3 -n /etc/passwd
```

Cleaning Bash History

- For the current session bash stores history in memory.
- Current bash session history can be wiped out by: `history -c` .
- On closing a bash session the history is appended to the bash history file (`.bash_history`) in user's home directory.
- After wiping out cache we may wish to remove a history file content too.

We don't remove a file. We just remove its content:

Emptying a bash history file

```
$ echo -n > ~/.bash_history
```

Bash Auto-completion

- We can start typing a word then press the <TAB> key. Bash will try to complete the word. The following rules apply:
  - the first word is completed to a valid command,
  - next words are completed to valid file names,
  - in case of ambiguity press <TAB> twice – all possible completions are displayed.

✍ *The following pairs of commands give the same results:*

Using the auto-completion

```
$ ec<TAB>
$ echo
$ ls -l /et<TAB>
$ ls -l /etc
```

Outline of the lecture

Compound Commands

- Commands delimited by semi-colon (`;`) on a single line will be run in sequence.
- Each command returns some value indicating success or failure. A command can be run conditionally – depending on the success (`&&`) or failure (`||`) of the previous command.

`cp` copies a file to a new name; `mv` – renames (moves), `rm` – deletes (removes). We delete `file3` or move `file5` depending on the result of copying:

Command sequences

```
$ cp file1 file2 ; rm file1
$ cp file3 file4 && rm file3
$ cp file5 file6 || mv file5 file6
```

Exit Status / Return Value

- Each command (more precisely: process) returns some exit status when it terminates.
- Contrary to C programming language exit status 0 means *success* (logical *true*). In case of success we need no further explanation.
- Other values denote *failure* (logical *false*). They may give more information on the reasons for the failure.

Wildcards and Relative Pathnames

- The following symbols can be used while specifying paths and filenames:

dot (`.`)	current (working) directory,
double dot (`..`)	parent directory,
tilde (`~`)	home directory,
question mark (`?`)	any character,
asterisk (`*`)	any sequence (string).

✍ *List all two-character names in the working directory:*

Using wildcards

```
$ ls ??
```

Absolute and Relative Paths

- Paths that start with / are absolute paths (relative to the topmost directory).

  ✍ *This works independently from the working directory:*

Absolute path

```
$ ls /somepath
```

- All other paths are relative to the working directory.

  ✍ *The following commands are equivalent:*

Relative paths

```
$ ls somepath
$ ls ./somepath
```

Literal Meaning of Single Characters

- Any single character preceded by a backslash (\) is considered by a shell literally – backslash removes its special meaning. It's often referred as *escaping*.

  ✍ *It is possible (but highly not recommended) to name a file * or even ' . Quoting or escaping is necessary to refer to such files. It may be also used for protecting blanks in filenames.*
  Note that # starts comments in this examples:

Escaping special characters

```
$ rm * # remove all files
$ rm \* # remove a single file '*'
$ rm filename with spaces # remove three files
$ rm filename\ with\ spaces # remove a single file
```

Single Quotes

- Single quotes (') preserve literal meaning of all enclosed characters.

  ✍ *Single quotes allow clear notation if there are many special characters to be escaped:*

Using single quotes

```
$ rm a\ b\ c\ d\ e # remove a single file
$ rm 'a b c d e' # remove a single file
```

Double Quotes

- Double quotes (") preserve literal meaning of all quoted characters except: \, $ and ` (on the tilde key):
  - $ allows referring to variable values (see the next section),
  - ` allows so-called command substitution (see the presentation on streams).
- Due to some bugs or poorly documented features bash may interpret some additional characters in double quotes.

Special Characters in File Names

- In file listings file names containing special characters may be shown enclosed in single quotes.
- There is no way to place a single quote inside single quotes.
- In file listings file names containing single quotes may be shown enclosed in double quotes.

✍ *Using double quotes*

What are the file names? ;-)

```
$ touch \"
  '"'
$ touch \'
  "'"
```

Notes on Escaping and Quoting

- Escaping and quoting affect how a shell reads characters.
- Those are transparent to commands.

  ✍ *In all cases `touch` receives `--help`*

Escaping and quoting is for a shell, not a command

```
$ touch --help
$ touch '--help'
$ touch "--help"
$ touch \-\-help
```

- Quotes are parsed from left to right; there is no nesting.

  ✍ *Single quotes have no effect here:*

Parsing quotes from left to right

```
$ echo "'\$'"
 '$'
```

Multi-Line Commands Revised

- By finishing line with \ we escape the end-of-line character.
- Quotes matching is requisite – placing unmatched quotes allows spanning commands over several lines.

Outline of the lecture

Variables as Labels

- As in numerous (all?) scripting languages variables act as object *labels*.
- Variables itself are untyped and don't need to be declared (however in some languages objects being labeled can by strictly typed). In shell we use a single data type – a string.
- Referring to undefined variables doesn't result in an error in shells – empty strings are returned.

Why Variables?

- Shells are programming languages.
- Variables may store long string literals and simplify command notation.
- Variables are used for customizing shell.
- So-called environment variables can be inherited by *child processes*.

Variables in Bourne Shells

- The following information refer to all sh-compliant shells. Csh-based shells use slightly different syntax, based on the C programming language.
- A common practice is using capital letters for variable names.
- Variables are defined by value assignment: `VARIABLE_NAME=value` (IMPORTANT: no blanks may precede nor supersede the '=').
- We refer to values stored in variables by preceding variable names with the dollar symbol: `$VARIABLE_NAME` . For ambiguity avoidance we may also use the form: `${VARIABLE_NAME}` (see next slides).

Managing Variables

Setting a variable

```
$ SOMEVAR="something"
```

Examining a variable

```
$ echo $SOMEVAR
$ echo ${SOMREVAR}
```

Displaying all variables

```
$ set
```

Unsetting a variable

```
$ unset SOMEVAR
$ SOMEVAR=
```

Using Variables – Examples

✍ *echo displays a string interpreted by the shell:*

Managing variables

```
$ VAR1=/etc
$ echo $VAR1
  /etc
$ ls -lA $VAR1
  listing the /etc directory contents
$ unset VAR1
$ echo $VAR1
```

✍ *We don't want the VAR2ERPILLAR variable:*

Curly braces for ambiguity avoidance

```
$ VAR2=CAT
$ echo ${VAR2}ERPILLAR
  CATERPILLAR
```

Bash History Revised

- Bash history is controlled by the following variables:
  - `HISTSIZE` – number of last commands stored in cache,
  - `HISTFILE` – file storing bash history,
  - `HISTFILESIZE` – number of commands stored in bash history file.

Environment

- For each process a set of variables defines its own environment.
- Child processes (processes invoked by the current one) receive a copy of parent's environment during process creation.
- The command `export` adds a shell variable to its environment.

  ✍ *We can export existing variables or join exporting and assigning a value:*

Exporting variables

```
$ VARIABLE1=value1
$ export VARIABLE1
$ export VARIABLE2=value2
```

- `printenv` and `env` print the list of environment variables.
- Bash provides an additional `declare` built-in which allows viewing and managing attributes (status) of all variables.

Important Variables

- Some important environment variables are:

`HOME`	home directory,
`LC_*` , `LANG`	locale settings,
`PATH`	colon-separated list of directories searched for executables,
`PWD`	path to the working (current) directory,
`SHELL`	login shell.

- Other shell variables include:

`IFS`	word separator (default: blanks),
`PS1`	(primary) prompt string description.

Redefining Shell Variables – Examples

✍ *Redefining the default prompt:*

Changing the default prompt

```
$ PS1='Hello> '
Hello>
```

✍ *Adding the working directory in front of the search path:*

Modifying the search path

```
$ PATH=.:$PATH
```

✍ *Selecting the interface language (it should be done before starting the GUI):*

Changing default localization to Polish

```
$ export LANG=pl_PL.UTF-8
```

Outline of the lecture

System vs. User's Configuration Files

- Most configuration files global for the system are placed in the `/etc` (*Edit-To-Configure*) directory.
- Most configuration files use 'shell-friendly' syntax:
  - those are ASCII text files,
  - each line is considered a separate entry,
  - `#` starts a comment,
- Most global settings can be overwritten by user's settings. User's configuration files reside is user's home directory, their names are preceded with dot (which marks them as hidden files).

Bash Configuration Files

- The names of shell initialization files use similar naming scheme. Note that even configuration file names for bash can slightly vary from one system to another. Consult your shell documentation for details.
- The following listing comes from bash documentation:

`/etc/profile`	The systemwide initialization file, executed for login shells
`~/.bash_profile`	The personal initialization file, executed for login shells,
`~/.bashrc`	The individual per-interactive-shell startup file
`~/.bash_logout`	The individual login shell cleanup file, executed when a login shell exits