6 Shell programming

There is no script for this section. Have a look at the commented shell script(s) on the lecture web page and at the manual pages for sh or bash.

7 Quoting

A more detailed version of this section can be found in *A Guide to Unix Shell Quoting* on the lecture web page.

For most non-trivial programs, certain characters or strings have a special meaning:

Terminal interface:	`Ctrl-C`	$\rightsquigarrow$ interrupt.
Bourne shell:	`|`	$\rightsquigarrow$ pipeline.
mv:	`-`	$\rightsquigarrow$ option.
unlink:	`/`	$\rightsquigarrow$ directory separator.

Quoting = taking away this special meaning from a character.

Levels of interpretation

User input is usually interpreted by several programs or modules successively, e. g.:

Terminal interface → Shell
  → Application program → System call.

Problems:

Which of these programs may misinterpret the input?

Does this program offer a quoting mechanism?

If so: how?

Terminal interface quoting

Usually: `Ctrl-V` quotes the next character
(can be changed using `stty`).

Some shells do their own command line editing rather than relying on the capabilities of the terminal driver
(such as bash, ksh, or tcsh).
Usually they mimic the terminal driver
$\rightsquigarrow$ `Ctrl-V` still works as a quoting character.

Note: Terminal interface quoting is relevant only for interactive shells, not for shells that read their input from a file.

The usual quoting characters in Unix editors are `Ctrl-V` (vi style) and `Ctrl-Q` (emacs style).

Shell quoting: Bourne shell

Bourne shell uses three characters for quoting:

- `'` single quote, apostrophe.
- `"` double quote, quotation mark.
- `\` backslash, reverse slash, reverse solidus.

The backquote or grave accent `` ` `` is used for command substitution rather than for quoting;
the acute accent ´ has no special function in the shell syntax.

A backslash (\) protects the next character, except if it is a newline. If a backslash precedes a newline, it prevents the newline from being interpreted as a command separator, but the backslash-newline pair disappears completely.

Single quotes (`'...'`) protect everything (even backslashes, newlines, etc.) except single quotes, until the next single quote.

Double quotes (`"..."`) protect everything except `"`, \, `$`, `` ` ``, until the next double quote. A backslash can be used to protect `"`, \, `$`, or `` ` `` within double quotes. A backslash-newline pair disappears completely; a backslash that does not precede
`"`, \, `$`, `` ` ``, or newline is taken literally.

Input interpretation

Before the shell executes a command, it performs the following operations (in this order!):

1 Syntax analysis (Parsing):
remove comments, split input into words, detect quoting, detect variables, detect keywords and control structures, . . .

concerns: `#`, Space, Tab, Newline, `'`, `"`, `\`, `` ` ``, `$VAR`, `=`, `;`,
`&`, `|`, `>`, `>>`, `(`, `{`, `for`, `while`, `do`, `if`, . . .

2 Parameter (variable) and command substitution:
`$HOME` → `/usr/home/walter`
`` `date` `` → `Mon Jan  7 16:30:18 CET 2008`

concerns: `$VAR`, `` `...` ``

3 Blank interpretation (Word Splitting):
if previous substitutions have introduced further whitespace characters, split into words.

concerns: Space, Tab, Newline

4 Filename generation (Globbing):
`*.c` → `input.c main.c output.c`

concerns: `*`, `?`, `[...]`

5 Quote removal:
remove all quoting characters detected at parsing time.

concerns: `'`, `"`, `\`

Overview:

1 Syntax analysis.
2 Parameter and command substitution: `$VAR`, `` `...` ``.
3 Blank interpretation: Space, Tab, Newline.
4 Filename generation: `*`, `?`, `[...]`.
5 Quote removal: `'`, `"`, `\`.

Different ways of quoting:

No quotes:	perform 1, 2, 3, 4, (5).
Double quotes:	perform 2, 5.
Single quotes/Backslashs:	perform 5.

Rules of thumb

Parameters and backquoted commands should usually be enclosed in double quotes.

Single quotes are also protected by double quotes
(or by a backslash).

Everything else that might be maltreated by the shell is
protected by single quotes.

The result of parameter or command substitution is subject to blank interpretation and filename generation (unless protected by double quotes), but it is not re-parsed.

Single/double quotes or backslashes count as quoting characters only if they are detected at parsing time, not when they are the result of parameter or command substitution.

`${VAR+...}`

If `$XY` is unset or set to the empty string, then

`$XY` expands to nothing,
`"$XY"` expands to an empty argument (just as `''` or `""`).

How can we get the usual effect of double quotes, except that an unset parameter should expand to nothing?

The special construct `${XY+word}` is evaluated to `word`,
if the parameter `$XY` is set, and to nothing, otherwise.

$\leadsto$ Use `${XY+"$XY"}`.

The list of positional parameters: `$*` and `$@`

The positional parameters (or command line arguments) can be accessed individually as `$1`, `$2`, ...

To access the whole list, the two special parameters `$*` and `$@` are available:

```
$*    ⤳  $1 $2 ...
$@    ⤳  $1 $2 ...
"$*"  ⤳  "$1 $2 ..."
"$@"  ⤳  "$1" "$2" ...
```

Rule: Use either `"$@"` or (to cater for older shells) `${1+"$@"}`.

Special cases

Assignments, case commands, indirections:

Double quotes around parameters can occasionally be omitted.

Here documents (`command <<word ...`),
command substitution (`` `...` ``):

Check the manual.

`IFS=''` $\rightsquigarrow$ prevent blank interpretation.

`set -f` $\rightsquigarrow$ disable filename generation.

Rules of thumb revisited

Parameters and backquoted commands should usually be enclosed in double quotes. The list of all positional parameters is accessed via `${1+"$@"}`. (For newer Bourne shells, `"$@"` is sufficient.)

Single quotes are also protected by double quotes (or by a backslash).

Everything else that might be maltreated by the shell is protected by single quotes.

If you know what you are doing:

If the value of a parameter cannot contain blanks or globbing characters (e. g.: `$$`, `$#`), double quotes may be omitted.

If the value of the parameter or backquoted command should be interpreted as a list (with blanks as separators and with expansion of globbing characters), double quotes must be omitted.

If an unset parameter should expand to nothing, use `${XY+"$XY"}`.

In assignments and case statements, double quotes may sometimes be omitted.

Shell quoting: Other shells

bash, ksh: similar to sh

csh, tcsh: chaotic

rc, es: easier, but different.

Application quoting

Options:

`--`: end of options (not universally supported),

relative filenames: `-foo` → `./-foo`

`test`, `[ ... ]`, `expr`:

relative filenames: `-foo` → `./-foo`

string comparison: `"$A" = "$B"` → `"xx$A" = "xx$B"`

use `[[ ... ]]` instead.

use `case` instead.

`set`:

without arguments:
prints values of all currently defined parameters.

with options (`-x`, `+x`):
turns on/off corresponding option of the current shell.

with non-option arguments:
assigns to positional parameters.

good news: `--` works as usual, but only if followed by at least one argument.

alternatively: use dummy element and shift it away.

`echo`:

System-V `echo` interprets certain character sequences:
`\a`, `\b`, `\n`, `\t`, ...

use `printf` instead.

use `cat` with here document instead.