A Guide to Unix Shell Quoting

New URL: http://rg1-teaching.mpi-inf.mpg.de/unixffb-ss98/quoting-guide.html
Please update your bookmarks.

Contents

1 Why is quoting necessary? The levels of interpretation

For most non-trivial programs, certain characters or strings have a special meaning. For example, the Ctrl-C character has a special meaning for the terminal interface: it causes an interrupt, rather than inserting a literal Ctrl-C character. The »`|`« sign has a special meaning for the Bourne shell: it serves to connect two processes by a pipeline. The »`-`« sign as the first character of an argument has a special meaning for an application program like »`mv`«: it tells »`mv`« that this argument should be interpreted as an option, rather than a file name. Quoting becomes necessary, whenever we want to remove this special meaning from a character, that is, when we want it to be interpreted like any other character.

For example imagine that we have accidentally created a file whose name consists of a hyphen, a vertical bar, and a Ctrl-C. When we want to rename this file, we have to keep in mind that our input is interpreted first by the terminal interface, then by the shell, and finally by »`mv`«,

and that each of them may misinterpret our input. To see how our input must be protected, or *quoted*, we proceed in reverse order:

One way to prevent »`mv`« from misinterpreting »`-`« as an option is to precede »`-`« with »`./`«. So »`mv`« should be invoked as

```
mv ./-|^C foo
```

(where `^C` denotes a Ctrl-C character). If we want the shell to invoke »`mv`« in this way, we have to ensure that it does not misinterpret the vertical bar. This can be done, for instance, by preceding it with a backslash. So the input of the shell should look like

```
mv ./-\|^C foo
```

Finally, to prevent the terminal interface from misinterpreting the Ctrl-C character, we have to precede it with a Ctrl-V. In other words, the sequence of keystrokes

```
m v Space . / - \ | Ctrl-V Ctrl-C Space f o o Return
```

does the job.

Note that the quoting rules of the terminal interface, the shell, and »`mv`« are completely different. A backslash would not prevent »`mv`« from interpreting the hyphen as an option character; typing Ctrl-V in front of the vertical bar would not prevent the shell from interpreting »`|`« as a pipeline character; and so on.

2 Terminal interface quoting

Terminal interface quoting is easier and more uniform than shell quoting or application quoting. The terminal input system has a function called `LNEXT` that causes the special meaning of the next character to be ignored. Usually, the `LNEXT` function is bound to Ctrl-V (but this can be configured differently), so the character sequence Ctrl-V Ctrl-C can be used to input a literal Ctrl-C without generating a SIGINT signal, and the character sequence Ctrl-V Ctrl-? can be used to input a literal Ctrl-? (Delete) character without deleting the preceding character. On some fairly old Unix systems, there is no `LNEXT` function. In this case, it is still possible to quote *some* control characters (usually Ctrl-D, Ctrl-U, Ctrl-?) by prefixing them with a backslash, but this technique is not universally applicable.

Shells that do their own command line editing rather than relying on the capabilities of the terminal driver (such as bash, ksh, or tcsh) usually mimic the `LNEXT` function of the terminal driver. Thus Ctrl-V still works as a quoting character.

3 Shell quoting

3.1 Bourne shell

3.1.1 Which quoting method protects which characters?

Bourne shell uses three characters for quoting: single quotes (`'`), double quotes (`"`), and backslashes (`\`). (The backquote (`` ` ``) is used for command substitution rather than for quoting, the acute accent (´) has no special function in the shell syntax.)

What is the difference between single and double quotes? And why is there more than one quoting method? The simple answer is

- 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 double quotes, backslashes, dollar signs, and backquotes, 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.

Note that it is perfectly legal to quote the third character with double quotes, the fifth to seventh with single quotes, and the rest not at all, so that

```
12'4"$\89
```

is quoted like

```
12"'"4'"$\'89
  \_/ \___/
```

But this is only half of the story. To understand the other half, one has to understand how the shell interprets its input. (If you prefer an easy rule of thumb to a complicated explanation, proceed to Sect. 3.1.11.)

3.1.2 Input interpretation: an overview

Before the shell executes a command, it performs the following operations (check the manual for the details):

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

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

2. Parameter (variable) and command substitution:
   `$HOME` -> `/usr/home/walter`
   `` `date` `` -> `Thu Jun 18 12:40:45 MET DST 1998`

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

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

   concerns: Space, Tab, Newline (cf. Sect. 3.1.4)

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

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

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

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

The order is important: Characters that are introduced during, say, step 2 may be processed further during step 3 and 4, but they are irrelevant for step 1 (cf. Sect. 3.1.3).

How does quoting influence the behaviour of the shell? If a string/character is quoted by single quotes or by a backslash, then none of the operations above occurs (except for quote removal, of course). If a string is quoted by double quotes, then parameter and command substitution still occur, but syntax analysis, blank interpretation and filename generation are prohibited.

As a consequence, there *is* a difference between `$XYZ` and `"$XYZ"`, (or between `` `date` `` and `` "`date`" ``), though in both cases, the parameter is replaced by its value: If the value of `$XYZ` contains globbing or whitespace characters, they are left unchanged in `"$XYZ"`. In `$XYZ`, however, globbing characters are expanded and whitespace is interpreted as a separator.

Example:

Suppose that the current directory contains the files `foo.1`, `foo.2`, and `bar` (and nothing else). Then in

```
$ XYZ='abc f*'
$ grep $XYZ bar
```

the parameter `$XYZ` is first replaced by »`abc f*`«, then the space inside the value of `$XYZ` is interpreted as a separator, and finally the »`f*`« expands to »`foo.1`« »`foo.2`«. So this is equivalent to

```
$ grep abc foo.1 foo.2 bar
```

(i.e., grep for »`abc`« in the files »`foo.1`«, »`foo.2`«, and »`bar`«).

Now let's put the parameter into double quotes:

```
$ XYZ='abc f*'
$ grep "$XYZ" bar
```

The parameter substitution occurs even within double quotes, but the space and the `*` inside the value of `$XYZ` are now protected by the quotes, so no further interpretation takes place here. This is therefore equivalent to

```
$ grep 'abc f*' bar
```

(i.e., grep for »`abc f*`« in the file »`bar`«). Note that this is what a programmer usually intends.

3.1.3 Common misconceptions

It is important to realize that parsing takes place before parameter and command substitution. The result of parameter or command substitution is therefore subject to blank interpretation and filename generation (unless protected by double quotes), but it is not re-parsed.

Example:

Suppose that the current directory contains the files `foo.1`, `foo.2`, and `bar` (and nothing else). Then in the command sequence

```
$ XYZ='f* ; cat bar'
$ echo $XYZ
foo.1 foo.2 ; cat bar
```

parameter substitution produces

```
echo f* ; cat bar
```

and filename generation yields

```
echo foo.1 foo.2 ; cat bar
```

Since the semicolon was not present at parsing time, it is taken literally and does not work as a command separator. Therefore »`echo`« is called with the five arguments »`foo.1`«, »`foo.2`«, »`;`«, »`cat`«, and »`bar`«.

Similarly, 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:

Example:

Consider the command sequence

```
$ XYZ='ghi jkl'
$ cat abc\ def $XYZ
```

Parameter substitution in the second command produces

```
cat abc\ def ghi jkl
```

The backslash quotes the space before »`def`«. As `$XYZ` was not enclosed in double quotes, the space before »`jkl`« is subject to blank interpretation; hence cat is

invoked with three arguments »`abc def`«, »`ghi`«, and »`jkl`«.

What happens if we precede the space between »`ghi`« and »`jkl`« in the assignment to `$XYZ` with a backslash?

```
$ XYZ='ghi\ jkl'
$ cat abc\ def $XYZ
```

Now parameter substitution produces

```
cat abc\ def ghi\ jkl
```

The first backslash was already present when the »`cat`« command was parsed, thus is quotes the following space. The second backslash, however, is the result of parameter substitution and was not present at parsing time. It is taken literally and does *not* quote the following space. So the space before »`jkl`« is still subject to blank interpretation and cat is invoked with three arguments »`abc def`«, »`ghi\`«, and »`jkl`«. There is no way to include quoting in the value of a parameter to compensate for the missing double quotes around `$XYZ`.

To force one more run through the parsing/substitution/globbing procedure, the »`eval`« command can be used:

Example:

Let us replace »`cat`« in the previous example by »`eval cat`«:

```
$ XYZ='ghi\ jkl'
$ eval cat abc\ def $XYZ
```

As above, parameter substitution produces

```
eval cat abc\ def ghi\ jkl
```

where the first backslash quotes the following space and the second backslash is taken literally. Thus »`eval`« is invoked with four arguments »`cat`«, »`abc def`«, »`ghi\`«, and »`jkl`«. Now »`eval`« concatenates its arguments separated by spaces, yielding

```
cat abc def ghi\ jkl
```

This string is parsed once more. At this time, the *quoted* backslash becomes a *quoting* backslash and thus the following space becomes protected, whereas the space after »`abc`« is no longer protected. Hence »`cat`« is called with three arguments »`abc`«, »`def`«, and »`ghi jkl`«.

3.1.4 Blank interpretation revisited

In Section 3.1.2, we have written that during the blank interpretation phase, spaces, tabs and newlines are interpreted as separators between words. This is indeed the default behaviour, but it can be changed: The characters at which the input is split are exactly those found in the value

of the shell parameter `$IFS` (IFS = Internal Field Separators). For instance in the command sequence

```
$ A='a:b:c   d'
$ echo $A
a:b:c d
```

»`echo`« is invoked with the two arguments »`a:b:c`« and »`d`«. Now let's change `$IFS` from its default value (a string consisting of a space, a tab, and a newline) to a colon, i.e.,

```
$ A='a:b:c   d'
$ IFS=:
$ echo $A
a b c   d
```

Then the value of `$A`, i.e., »`a:b:c   d`« is split no longer at the spaces, but at the colons, thus »`echo`« is invoked with three arguments »`a`«, »`b`«, and »`c   d`«. In particular, setting `IFS=''` prevents blank interpretation completely.

See Sect. 3.2.1 for differences between bash, ksh and sh.

3.1.5 Disabling filename generation

The »`-f`« option to the »`set`« command makes it possible to disable filename generation:

```
$ echo *
bar foo.1 foo.2
$ set -f
$ echo *
*
```

Using the command

```
$ set +f
```

the default behaviour can be restored.

This feature was added to the Bourne shell in System III; almost all Bourne shells used today support it. It is particularly useful in situations, where we *want* blank interpretation (so that enclosing a certain parameter in double quotes is not a solution) but nevertheless want to prevent filename generation.

3.1.6 Assignments, `case` commands, indirection

There are some particular situations in which neither blank interpretation nor filename generation occurs: in assignments, between the keywords »`case`« and »`in`« of case commands, and after input/output redirection operators such as »<« or »>>«. Note that in all these cases the shell syntax allows only a single word, not a sequence of words, so that blank interpretation or expansion of globbing characters might result in something syntactically illegal. Double quotes around parameters or backquoted commands serve only to prevent blank interpretation and filename generation, hence they are unnecessary in the situations mentioned above. For instance, the double quotes in the following three examples may be omitted:

```
case "$A" in ...

D="$A/$B/$C"

cat result > "`date`"
```

On the other hand, quotes remain necessary if, say, the value after »=« contains literal whitespace (which is detected already at parsing time, not during blank interpretation):

```
D="$A $B $C"
```

See Sect. 3.2.1 for differences between bash, ksh and sh.

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

In the Bourne shell, the positional parameters (or command line arguments) can be accessed individually as `$1`, `$2`, ... To access the whole list of positional parameters, the two special parameters `$*` and `$@` are available. Outside of double quotes, these two are equivalent: Both expand to the list of positional parameters starting with `$1` (separated by spaces). Within double quotes, however, they differ: `$*` within a pair of double quotes is equivalent to the list of positional parameters, separated by *quoted* spaces, i.e., `"$1 $2 ..."`. On the other hand, `$@` within a pair of double quotes is equivalent to the list of positional parameters, separated by *unquoted* spaces, i.e., `"$1" "$2" ...`. (This is the behaviour if `$IFS` has its default value (space, tab, newline). If `$IFS` has a non-standard value, the evaluation of `$*`, `$@`, `"$*"`, and `"$@"` is highly obscure, non-intuitive, badly documented, and varying between different shells. Avoid it.)

What happens if the list of positional parameters is empty (i.e., `$#` equals 0)? As one should expect, `$*` and `$@` expand to nothing and `"$*"` expands to one empty argument. The question is: what should `"$@"` be? Originally, it expanded to one empty argument, just as `"$*"`. But this is somewhat inconsistent: it contradicts the usual rule that `"$@"` yields exactly the list of all positional parameters originally passed to the current program, i.e., a list whose length equals `$#`. (See Sect. 3.1.8 for a fix.) In most Bourne shells used today, the evaluation of `"$@"` has been regularized, so that `"$@"` behaves in the way a programmer expects: it expands to the list of all positional parameters, and in particular it is expands to nothing if the list of positional parameters is empty.

3.1.8 `${VAR+...}`

If the parameter `$XYZ` is unset or set to the empty string, then `$XYZ` expands to nothing, but `"$XYZ"` expands to an empty argument (just as `''` or `""`). What can we do, if we want to have the usual effect of double quotes (i.e., no blank interpretation, no filename generation), except that an unset parameter should expand to nothing?

The `${VAR+...}` construct can be used to solve our problem. Generally, `${XYZ+word}` is evaluated to »`word`«, if the parameter `$XYZ` is set, and to nothing, otherwise. In particular, `${XYZ+"$XYZ"}` is evaluated to `"$XYZ"`, if the parameter `$XYZ` is set, and to nothing, otherwise. It is also possible to treat a parameter set to the empty string in the same way as an unset parameter; in this case we have to put a »`:`« before the plus sign.

Using `${VAR+...}` it is also possible to pass the list of all positional parameters to a subprogram in a portable way:

```
${1+"$@"}
```

expands to nothing, if `$#` equals 0 (i.e., if `$1` is undefined), and otherwise to `"$@"`, that is `"$1" "$2" ...`. This works also for shells in which `"$@"` expands to one empty argument if `$#` equals 0.

3.1.9 Here documents

A *here document*

```
command <<word
...
...
```

is a special type of redirection that instructs the shell to take some part of the current source file as standard input for »`command`«. The delimiter »`word`« is subject to parameter and command substitution. All lines of the current source up to (and not including) the first following line containing only »`word`« constitute the standard input for »`command`«.

If any part of »`word`« is quoted, then no additional processing is done on the lines constituting the here document, hence the output of

```
$ A=123
$ cat <<'qwerty'
bcd\
$A
qwerty
```

consists of the two lines

```
bcd\
$A
```

If no part of »`word`« is quoted, then parameter and command substitution occurs, every newline preceded by a backslash is removed, and every dollar sign, backquote, or backslash must be quoted by a backslash, thus

```
$ A=123
$ cat <<qwerty
bcd\
$A
qwerty
```

produces the output

```
bcd123
```

If the special form »`<<-`« of the redirection operator is used, then all leading tab characters are stripped from the input lines (before the input lines are compared with »`word`«.)

See Sect. 3.2.1 for differences between bash, ksh and sh.

3.1.10 Details of command substitution

When the shell encounters a string between backquotes

```
`cmd`
```

it executes cmd and replaces the backquoted string with the standard output of cmd, with any trailing newlines deleted. (There is no way to preserve these trailing newlines!)

Quoting inside backquoted commands is somewhat complicated, mainy because the same token is used to start and to end a backquoted command. As a consequence, to nest backquoted commands, the backquotes of the inner one have to be escaped using backslashes. Furthermore, backslashes can be used to quote other backslashes and dollar signs (the latter are in fact redundant). In the backquoted command is contained within double quotes, a backslash can also be used to quote a double quote. All these backslashes are removed when the shell reads the backquoted command. All other backslashes are left intact. (Usually, nested backquoted commands can be avoided using variable assignments.)

See Sect. 3.2.1 for differences between bash, ksh and sh.

3.1.11 The golden rules of Bourne shell quoting

So, what are the *right* quotes to use in a shell script? It depends. But for those who prefer an easy rule to a complicated explanation, there are some rules of thumb that work very well in practice:

- Parameters and backquoted commands that should be interpreted by the shell are enclosed in double quotes. Single quotes are also protected by double quotes (or by a backslash).
- The proper way to pass the list of all positional parameters (or command line arguments) to a subprogram is »`some_prg ${1+"$@"}`«. (For most newer Bourne shells, »`some_prg "$@"`« has the same effect.)
- Everything else that might be maltreated by the shell is protected by single quotes.

For those who know what they are doing:

- If you are absolutely sure that the value of the parameter contains neither blanks nor globbing characters, you *may* omit the quotes. This applies for instance to the parameters `$$`, `$#`, `$?`, and `$!`. (Never do this for command line arguments!)
- If you really want the value of the parameter or backquoted command to be interpreted as a list, with embedded blanks as separators (and with expansion of globbing characters), you *must* omit the double quotes. (This occurs rather infrequently.) If you want blank interpretation but no filename generation, use the »`set -f`« command; if you want filename generation but no blank interpretation, set the parameter `$IFS` temporarily to `''`.

- If the parameter `$XYZ` is unset or set to the empty string, then `$XYZ` expands to nothing, but `"$XYZ"` expands to an empty argument (just as `''` or `""`). If you want to have the usual effect of double quotes, except that an unset parameter should expand to nothing, use `${XYZ+"$XYZ"}`. If additionally you want an empty parameter to be treated in the same way as an unset parameter, put a `:` before the plus sign.

- In assignments and case statements, neither blank interpretation nor filename generation takes place; thus double quotes around parameters or backquoted commands are redundant. (In the Bourne shell, the same applies to i/o redirections. Notice, however, that bash and ksh differ here, hence for portability reasons it is better not to omit the double quotes.)

3.2 Other shells

3.2.1 bash, ksh

Quoting in bash or ksh is similar to Bourne shell quoting. Some differences are due to the fact that the number of interpretation steps in these shells is significantly larger: Apart from parameter and command substitution, there is also alias substitution, history substitution (only bash), brace expansion (only bash), tilde expansion, arithmetic expansion, and process substitution (not on all Unices). Arithmetic expansion behaves in the same way as parameter and command substitution: it takes place also within double quotes. The rules for alias substitution and history substitution are slightly confusing. Consult the manual for details. In bash, both are enabled by default only in interactive shells.

Both bash and ksh offer an alternative syntax for backquoted commands. Instead of

```
`command`
```

they allow us to write

```
$(command).
```

This form avoids most of the quoting troubles of backquoted commands.

Some further differences:

Both bash and ksh perform blank interpretation only if the argument is non-constant, that is, if it contains a parameter or backquoted command:

```
bash$ IFS=,
bash$ A='a,b,c,d'
bash$ set $A
bash$ echo "$1"
a
bash$ set x,y,$A
bash$ echo "$1"
x
bash$ set x,y
bash$ echo "$1"
x,y
```

By contrast, the Bourne shell splits on `$IFS` even if the argument is constant:

```
$ IFS=,
$ set x,y
$ echo "$1"
x
```

In the Bourne shell, `$*` within double quotes expands to all positional parameters separated by quoted spaces. In both bash and ksh, the positional parameters are separated by the first character of `$IFS`:

```
$ set q w e r t
$ IFS=,:
$ echo "$*"
q w e r t

bash$ set q w e r t
bash$ IFS=,:
bash$ echo "$*"
q,w,e,r,t
```

In contrast to the Bourne shell, both bash and ksh perform filename generation after input/output redirection operators; bash also performs blank interpretation. If the result of (blank interpretation and) filename generation is more than one word, they produce an error. For instance, if the current directory contains exactly one file »`foobar`« whose name matches »`foo*`«, then in bash or ksh,

```
echo > foo*
```

overwrites »`foobar`«, whereas the same command in sh creates a new file »`foo*`«.

Neither bash nor ksh perform parameter or command substitution on the delimiter of a here document: The output of

```
bash$ A=abc
bash$ cat <<$A
q
abc
$A
```

consists of the two lines

```
q
abc
```

In the Bourne shell, line 4 (containing »`abc`«) would already have terminated the here document.

3.2.2 csh, tcsh

Quoting in csh or tcsh is noticeably different from quoting in Bourne-like shells. The main differences one should observe are the following:

In the Bourne shell, a backslash can be used to protect `"`, `$`, or `` ` `` within double quotes. In csh and tcsh, one cannot put a literal `"`, `$`, or `` ` `` into double quotes, neither with a backslash nor

without. If one needs one of these characters literally, it has to be quoted with single quotes or *only* with a backslash. (In tcsh (but not in csh), a double quote can be protected within double quotes by a backslash, if the `backslash_quote` option is set. This does not work for `$` and `` ` ``, though.)

In contrast to the Bourne shell, a newline character within single or double quotes is permitted only if it is preceded by a backslash. A non-backslashed newline within single or double quotes leads to a syntax error; a backslashed newline within single or double quotes becomes a literal newline; a backslashed newline outside of single or double quotes is an argument separator.

In the Bourne shell, backquoted commands should be enclosed in double quotes to prevent blank interpretation and globbing. In csh and tcsh, double quotes prevent spaces and tabs in the output of a backquoted command from being interpreted as word separators; newlines, however, are always taken as word separators, even if enclosed in double quotes. (However, like in a Bourne shell, a trailing newline is always discarded.)

Analogously to backquoted commands, most parameters should be enclosed in double quotes in the Bourne shell to prevent blank interpretation and globbing. In csh and tcsh, a parameter in double quotes produces a syntax error if the value of the parameter contains a newline. Parameters should therefore *not* be included in double quotes; instead of »`"$var"`« use »`$var:q`« to prevent blank interpretation and globbing.

In csh and tcsh, an exclamation mark can only be quoted with a backslash, even within single or double quotes. This holds not only for interactive shells, but even for scripts.

4 Application quoting

In this context, *application* means any external command that is called from a shell (and even some shell built-in commands that behave syntactically like external commands). Application quoting is application-dependent. There are no general rules: Some techniques are very widely usable, others are limited to a particular program. In some programs, there is no need at all for a quoting mechanism, in others, there is need but still no quoting mechanism.

4.1 Options

Most Unix commands distinguish an option from another argument by looking at its first character: If it starts with a »`-`«, it's an option, otherwise, it's not. (Some commands use additional characters besides »`-`«: occasionally »+«, rarely others. The following paragraphs apply to this case analogously.) What can we do if we need to persuade a command to consider an argument starting with »`-`« as a non-option?

A wide-spread convention is to interpret the special option »`--`« as the end of the option list and every further argument as a non-option. This convention is very handy but, unfortunately, there are still many commands around that do not obey it. One should not even rely on the fact that all implementations of a certain command behave in the same way. The »`rm`« command is a well-known example: Some »`rm`« implementations interpret »`--`« as the end of option list, some

use »-« instead (which is a bad idea since »-« commonly denotes standard input), some »rm« implementations accept both, and some accept neither.

For filenames starting with »-« there is a portable and fairly general way to pass them to a command without having them misinterpreted as an option. The key idea is to exploit the fact that a relative filename »foo« denotes the same file as »./foo« (i.e., »foo« in the current directory ».«). Hence, if a filename starts with »-«, it is sufficient to put »./« in front of it to have it correctly interpreted as a filename:

```
$ mv ./-bar baz
```

If the filename is unknown, say a positional parameter in a script, its first character is most easily checked with a case statement:

```
case $1 in
  -*) FILE=./$1 ;;
  *)  FILE=$1   ;;
esac
mv "$FILE" /tmp/baz
```

4.2 `echo`

The »echo« command exists in numerous incompatible versions. Some implementations accept the option »-n« to suppress the trailing newline; some implementations accept various backslash-escapes for control characters (including »\c« to suppress the trailing newline), and some accept both. In those implementations that accept backslash-escapes, there is usually no way to disable the interpretation of backslashs. To output arbitrary strings in a reliable manner, it is therefore recommended to use »printf« instead of »echo«.

4.3 `test`, `expr`

Both »test« and »expr« suffer from the same problem: under certain circumstances, they cannot tell operands from operators or parentheses.

Example:

> Consider the following two commands:
>
> ```
> test \( -r "$A" \) -o "$B1" = "$C"
> test \( "$B2" = "$C" -o -w "$A" \)
> ```
>
> Obviously, the programmer's intention was, in the first case, to test whether the file $A is readable or whether the strings $B1 and $C are equal, and in the second case, to test whether the strings $B2 and $C are equal or whether the file $A is writable. (In both cases, the parentheses are in fact redundant. However, we could easily add some more conditions so that the parentheses become necessary.)
>
> Now let us assume that the parameters are initialized as follows:

```
A='='
B1=-w
B2=-r
C=')'
```

Then in both cases, after the shell has processed the command, »`test`« is called with exactly the same arguments:

```
( -r = ) -o -w = )
```

So there is absolutely no way for »`test`« to figure out what the programmer intended.

In fact, implementations of »`test`« tend to fail even on inputs that *can* uniquely be parsed. Therefore two workarounds are recommended:

- A relative filename »`foo`« denotes the same file as »`./foo`« (i.e., »`foo`« in the current directory »`.`«). Hence, if a filename might be misinterpreted by »`test`«, it is sufficient to put »`./`« in front of it to have it correctly interpreted as a filename (cf. Sect. 4.1).

- The result of a string comparison »`foo = bar`« does not change, if both strings are preceded by the same prefix string, say »`xx`«. (The same holds for inequality tests.) As no operator of »`test`« or »`expr`« starts with »`xx`«, this is again safe.

Example:

As we have seen above, the command

```
test \( -r "$A" \) -o "$B1" = "$C"
```

is not safe. To force `$A` to be interpreted as a filename and `$B1` and `$C` as strings, we can use

```
case $A in
  /*) A0=$A ;;    # absolute filename: don't change
  *)  A0=./$A ;;  # relative filename: prepend »./«
esac
test \( -r "$A0" \) -o "xx$B1" = "xx$C"
```

Similarly,

```
test -z "$A"
```

and

```
test -n "$A"
```

are unsafe in some implementations of »`test`« and should be replaced by

```
test "xx$A" = xx
```

and

```
test "xx$A" != xx
```

The »`test`« command has an alternative syntactic form: The command

```
[ condition ]
```

is equivalent to

```
test condition
```

and it shares its deficiencies. Even if »`[ ... ]`« looks as if it were a part of the shell syntax, it is not: »`[`« behaves like any other external command (though both »`test`« and »`[`« are usually built into the shell for efficiency reasons).

The »`case`« control structure does not suffer from the problems of »`test`«. It is parsed before parameters are evaluated, hence

```
case $A in
  -r)  echo 'parameter $A equals -r' ;;
  ')') echo 'parameter $A equals )' ;;
  -p*) echo 'parameter $A starts with -p' ;;
esac
```

works even if `$A` is set to »`-r`« or »`)`« or to some strange string like »`esac in ;;`«. In situations where both »`case`« and »`test`«/»`expr`« are applicable, »`case`« is therefore preferred.

Besides the »`[`« command, bash and ksh offer a »`[[ ... ]]`« construct. This is similar to »`[ ... ]`«, but, just like »`case`«, it is also parsed before parameters are evaluated and it is therefore robust.

4.4 `set`

The »`set`« command of the Bourne shell has three functions. It is called as

```
set options non-option-arguments
```

where both options and non-option-arguments may be omitted.

- If there are no arguments at all, »`set`« prints the values of all currently defined parameters (except special parameters such as `$1`, `$2`, `$*`, `$$`, `$#`, ...).
- Options start either with »`-`« or with »`+`«. If »`set`« is called with an option starting with »`-`«, it turns on the corresponding option of the current shell. For instance, if

  ```
  set -x
  ```

  is executed in a shell script »`foo`«, the shell behaves as if it had been invoked by

  ```
  sh -x foo ...
  ```

  Options starting with »`+`« turn off the corresponding option of the current shell.
- Non-option-arguments are assigned, in order, to the positional parameters. For instance, if

```
set abc '' def
```

is executed in a shell script »`foo`«, the shell behaves as if it had been invoked by

```
sh [options] foo abc '' def
```

that is, `$1` is set to »`abc`«, `$2` to the empty string, `$3` to »`def`«, and all further positional parameters are unset.

»`set`« obeys the »`--`«-convention mentioned above, hence

```
set -- "$XYZ"
```

can be used to set `$1` safely to `$XYZ`, even if the value of `$XYZ` starts with »`-`«. There remains only one problematic case: One might assume that

```
set --
```

unsets all positional parameters. In some Bourne shells, however, it leaves all positional parameters unchanged. To get the desired effect in a portable way, use a dummy argument and shift it:

```
set foobar
shift
```

In both bash and ksh, »`set --`« works as desired.

4.5 `read`

The »`read`« command of the Bourne shell reads a line from standard input, splits it into words using $IFS, and assigns the words to variables. Backslashs in the input escape the next character; a backslash at the end of the line can be used for line continuation. In most cases, this behaviour is not desired. Newer versions of the Bourne shell implement an option »`-r`« for »`read`« that prevents interpretation of the backslash as an escape or line continuation character.

Uwe Waldmann <uwe@mpi-inf.mpg.de>, 2014-05-02.

Acknowledgments: Thanks to Mark Brader, Greg Ubben, Chet Ramey, Joachim Saul, Jürgen Salk, and Sriraaman Sridharan.