Rich's sh (POSIX shell) tricks

This page is meant as a repository for useful tricks I've found (and some I've perhaps invented) for scripting the POSIX shell (with some attention to portability to non-conformant shells as well, scattered here and there). I am a strong believer that Bourne-derived languages are extremely bad, on the same order of badness as Perl, for programming, and consider programming sh for any purpose other than as a super-portable, lowest-common-denominator platform for build or bootstrap scripts and the like, as an extremely misguided endeavor. As such you won't see me spending many words on extensions particular to ksh, Bash, or whatever other shells may be popular.

Printing the value of a variable

```
printf %s\\n "$var"
```

The "\\n" may be omitted if a following newline is not desired. The quotation marks are essential. The following is **NOT** a valid substitute:

```
echo "$var"
```

NEVER use echo like this. According to POSIX, echo has unspecified behavior if any of its arguments contain "\" or if its first argument is "-n". Unix™ standards fill in this unspecified area for XSI-conformant implementations, by specifying nasty undesirable behavior that no one wants ("\" is interpreted as a C-string-literal style escape), and other popular implementations such as Bash interpret argument values other than "-n" as special options even when in "POSIX compatibility" mode, rendering then nonconformant. The jist of this is:

	Output		
Command	**POSIX**	**Unix**	**Bash**
echo "-e"	"-e "	"-e "	" "
echo "\\n"	*unspecified*	" "	"\n "
echo -n hello	*unspecified*	"hello"	"hello"
echo -ne hello	"-ne hello "	"-ne hello "	"hello"

These issues mean `echo "$var"` can sting you whenever you do not have strict guarantees about the contents of `var` (for example that it contains a nonnegative integer). Even if you're a GNU/Linux-centric jerk who thinks all the world is a Bash and you don't care about portability, you'll run into trouble someday when you happen to read a "-n" or "-e" or "-neEenenEene" into `var` and suddenly your script breaks.

If you're really attached to the name "echo" and want to use it in your scripts, try this function which 'repairs' echo to behave in a reasonable way (pretty much like the Bash echo command, but with the added stipulation that the last argument will never be interpreted as an option, so that `echo "$var"` is safe even when `var`'s contents look like an option):

```
echo () (
fmt=%s end=\\n IFS=" "

while [ $# -gt 1 ] ; do
case "$1" in
[!-]*|-*[!ne]*) break ;;
*ne*|*en*) fmt=%b end= ;;
*n*) end= ;;
*e*) fmt=%b ;;
esac
shift
done

printf "$fmt$end" "$*"
)
```

Dropping this code in at the top of your script will likely fix all the subtle bugs due to the utter brain damage of the standard echo command. Or, if you think the whole idea of echo having options is preposterous, try this simpler version (use of "$*" instead of "$@" is very intentional here):

```
echo () { printf %s\\n "$*" ; }
```

You never imagined printing the value of a variable could be so difficult, eh? Now you see why I say Bourne-derivative languages should never be used for serious programming...

Reading input line-by-line

```
IFS= read -r var
```

This command reads a line of input, terminated by a newline or end of file or error condition, from stdin and stores the result in var. Exit status will be 0 (success) if a newline is reached, and nonzero (failure) if a read error or end of file terminates the line. Robust scripts may wish to distinguish between these cases. According to my reading of POSIX, the contents of var should be filled with the data read even if an error or premature end of file terminates the read, but I am uncertain whether all implementations behave as such and whether it is strictly required. Comments from experts are welcome.

One common pitfall is trying to read output piped from commands, such as:

```
foo | IFS= read var
```

POSIX allows any or all commands in a pipeline to be run in subshells, and which command (if any) runs in the main shell varies greatly between implementations — in particular Bash and ksh differ here. The standard idiom for overcoming this problem is to use a here document:

```
IFS= read var << EOF
$(foo)
EOF
```

Reading input byte-by-byte

```
read dummy oct << EOF
$(dd bs=1 count=1|od -b)
EOF
```

This command leaves the octal value of a byte of input in the variable oct. Note that dd is the only standard command which can safely read exactly one byte of input with a guarantee that no additional bytes will be buffered and lost. Aside from failing to be portable, `head -c 1` may be implemented using C stdio functions with buffering.

Conversion to some escaped format (in this case octal) is necessary because the read command deals with text files. It cannot handle arbitrary bytes; in particular there is no way to store a NUL byte in a shell variable. Other issues with non-ASCII bytes may exist as well depending on your implementation and your locale. It's possible to modify this code to read several bytes at a time, but take care to account for all the various bad behavior of the od program such as condensing long runs of zeros.

Conversion of the octal back to binary data can be accomplished via the next sh trick.

Writing bytes to stdout by numeric value

```
writebytes () { printf %b `printf \\\\%03o "$@"` ; }
writebytes 65 66 67 10
```

This function allows specification of byte values in base 8, 10, or 16. Octal and hex values must be prefixed with 0 or 0x, respectively. If you want the arguments to always be treated as octal, for example when processing values read by the previous trick for reading binary data, try this version:

```
writeoct () { printf %b `printf \\\\%s "$@"` ; }
```

Be aware that it will break if your octal values are larger than 3 digits, so don't prepend a leading 0. The following version is much slower but avoids that problem:

```
writeoct2 () { printf %b $(printf \\%03o $(printf 0%s\  "$@")) ; }
```

Using find with xargs

GNU fans are accustomed to using the -print0 and -0 options to find and xargs, respectively, for robust and efficient application of a command to all results of the find command. Without GNU extensions, the output of find is newline-delimited, meaning there is no way to recover the actual pathnames found if some of the pathnames contain embedded newlines.

If you don't mind having your script break when pathnames contain newlines, at least make sure that the misprocessing that will result cannot lead to a compromise of privilege, and then try the following:

```
find ... | sed 's/./\\&/g' | xargs command
```

The sed command here is mandatory. Contrary to popular belief (well, it was popular enough that I mistakenly believed it for a long time), xargs does NOT accept newline-delimited lists. Rather it accepts shell-quoted lists, i.e. the input list is separated by whitespace and all internal whitespace must be quoted. The above command simply quotes all characters with backslashes to satisfy this requirement, protecting embedded whitespace in filenames.

Using find with +

Of course the much smarter way to use find to efficiently apply commands to files is with -exec and a "+" replacing the ";":

```
find path -exec command '{}' +
```

This causes find to place as many filenames as will fit on the command line in place of the "{}", each as its own argument. There is no issue with embedded newlines being misinterpreted. Sadly, despite its presence in POSIX for a long time, the popular GNU implementation of find did not support "+" for the longest time, and so its use is rather unportable in practice. A reasonable workaround would be to write a test for support of "+", and use ";" in place of "+" (with the naturally severe loss in efficiency) on broken systems where find is nonconformant.

Here is a command which must succeed on any POSIX conformant system but which will fail if find lacks support for "+" due to a missing ";" argument:

```
find /dev/null -exec true '{}' +
```

This takes advantage of the fact that "/dev/null" is one of the only three non-directory absolute pathnames guaranteed by POSIX to exist.

Portable version of find -print0

```
find path -exec printf %s\\0 '{}' +
```

Portability is subject to the above notes on GNU find's lack of support for "+" until recent versions, so it's probably a good idea to fallback to using ";" instead of "+" if necessary.

Note that this trick is probably useless, since the output is not a text file. Nothing but GNU xargs should be expected to parse it.

Using the output of find -print robustly

Despite the embedded-newline field-separator-emulation issue of `find`, it is possible to parse the output robustly. Just remember, "slash dot saves the day." For each absolute path being searched, prefix the initial "/" with "/.", and likewise prefix each relative path to be searched with "././" — the string "/./" then becomes a magic synchronization marker for determining if a newline was produced as a field separator or due to embedded newlines in a pathname.

Processing the output is left as an exercise for the reader.

Getting non-clobbered output from command substitution

The following is not safe:

```
var=$(dirname "$f")
```

Due to most commands writing a newline at the end of their output, Bourne-style command substitution was designed to strip training newlines from the output. But it doesn't just strip one trailing newline; it strips them all. In the above command, if `f` contains any trailing newlines in the last directory component, they will be stripped, yielding a different directory name. While no one sane would put newlines in directory names, such corruption of the results could lead to exploitable vulnerabilities in scripts.

The solution to this problem is very simple: add a safety character after the last newline, then use the shell's parameter substitution to remove the safety character:

```
var=$(command ; echo x) ; var=${var%?}
```

In the case of the dirname command, one also wants to remove the single final newline added by dirname, i.e.

```
var=$(dirname "$f" ; echo x) ; var=${var%??}
```

Of course there is an easier way to get the directory part of a pathname, provided you don't care about some of the odd corner-case semantics of the dirname command:

```
var=${f%/*}
```

This will fail for files in the root directory, among other corner cases, so a good approach would be to write a shell function to consider such special cases. Note, however, that such a function must somehow store its results in a variable. If it printed them to stdout, as is common practice when writing shell functions to process strings, we would run into the issue of "$(...)" stripping trailing newlines once again and be back where we started...

Returning strings from a shell function

As can be seen from the above pitfall of command substitution, stdout is not a good avenue for shell functions to return strings to their caller, unless the output is in a format where trailing newlines are insignificant. Certainly such practice is not acceptable for functions meant to deal with arbitrary strings. So, what can be done?

Try this:

```
func () {
body here
eval "$1=\${foo}"
}
```

Of course `${foo}` could be replaced by any sort of substitution. The key trick here is the eval line and the use of escaping. The "$1" is expanded when the argument to eval is constructed by the main command parser. But the "${foo}" is not expanded at this stage, because the "$" has been quoted. Instead, it's expanded when eval evaluates its argument. If it's not clear why this is important, consider how the following would be bad:

```
foo='hello ; rm -rf /'
dest=bar
eval "$dest=$foo"
```

But of course the following version is perfectly safe:

```
foo='hello ; rm -rf /'
dest=bar
eval "$dest=\$foo"
```

Note that in the original example, "$1" was used to allow the caller to pass the destination variable name as an argument the function. If your function needs to use the shift command, for instance to handle the remaining arguments as "$@", then it may be useful to save the value of "$1" in a temporary variable at the beginning of the function.

Shell-quoting arbitrary strings

Sometimes it's necessary to put a string in a shell-quoted form, for instance if it needs to be expanded into a command that will be evaluated with eval, written into a generated script, or similar. There are several methods, but many of them fail if the string contains newlines. Here is a version that works:

```
quote () { printf %s\\n "$1" | sed "s/'/'\\\\''/g;1s/^/'/;\$s/\$/'/" ; }
```

This function simply replaces every instance of «'» (single quote) within the string with «'\"» (single quote, backslash, single quote, single quote), then puts single quotes at the beginning and end of the string. Since the only character whose meaning is special within single quotes is the single quote character itself, this is totally safe. Trailing newlines are handled correctly, and the single quote at the end doubles as a safety character to prevent command substitution from clobbering the trailing newlines, should one want to do something like:

```
quoted=$(quote "$var")
```

Working with arrays

Unlike "enhanced" Bourne shells such as Bash, the POSIX shell does not have array types. However, with a bit of inefficiency, you can get array-like semantics in a pinch using pure POSIX sh. The trick is that you do have one (and only one) array — the positional parameters "$1", "$2", etc. — and you can swap things in and out of this array.

Replacing the contents of the "$@" array is easy:

```
set -- foo bar baz boo
```

Or, perhaps more usefully:

```
set -- *
```

What's not clear is how to save the current contents of "$@" so you can get it back after replacing it, and how to programmatically generate these 'arrays'. Try this function based on the previous trick with quoting:

```
save () {
for i do printf %s\\n "$i" | sed "s/'/'\\\\''/g;1s/^/'/;\$s/\$/' \\\\/" ; done
echo " "
}
```

Usage is something like:

```
myarray=$(save "$@")
set -- foo bar baz boo
eval "set -- $myarray"
```

Here, the quoting has prepared "$array" for use with the eval command, to restore the positional parameters. Other possibilities such as `myarray=$(save *)` are also possible, as well as programmatic generation of values for the 'array' variable.

One could also generate an 'array' variable from the output of the find command, either using a cleverly constructed command with the -exec option or ignoring the possibility of newlines in pathnames and using the sed command for prepping find's results for xargs.

```
findarray () {
find "$@" -exec sh -c "for i do printf %s\\\\n \"\$i\" \\
| sed \"s/'/'\\\\\\\\''/g;1s/^/'/;\\\$s/\\\$/' \\\\\\\\/\"
done" dummy '{}' +
}
```

Such a script allows things like:

```
old=$(save "$@")
eval "set -- $(findarray path)"
for i do command "$i" ; done
eval "set -- $old"
```

Note that this duplicates the intended functionality of the horribly-incorrect but often-cited "`for i in `find ...` ; do ...`" construct.

Does a given string match a given filename (glob) pattern?

```
fnmatch () { case "$2" in $1) return 0 ;; *) return 1 ;; esac ; }
```

Now you can do things like:

```
if fnmatch 'a??*' "$var" ; then ... ; fi
```

So much for needing Bash's "[[" command...

Counting occurrences of a character

```
tr -dc 'a' | wc -c
```

This will count the number of occurrences of the character "a", by deleting all other characters. However, it's not clear what `tr -dc` does when encountering noncharacter bytes in the input; POSIX is unclear on this matter and implementations likely differ. Foundational logicians will appreciate this as a practical real-world difficulty in working with set complements and the universal set.

Instead, try the following:

```
tr a\\n \\na | wc -l
```

The `wc -l` command counts the number of newline characters seen, so using `tr` to swap occurrences of "a" with newlines allows `tr` to count "a"s instead.

Overriding locale categories

The following does not necessarily work as intended:

```
LC_COLLATE=C ls
```

This is because LC_ALL may be present in the environment, overriding any of the category-specific variables. Unsetting LC_ALL also provides incorrect behavior, in that it possibly changes all of the categories. Instead try:

```
eval export `locale` ; unset LC_ALL
```

This command explicitly sets all category-specific locale variables according to the implicit values they receive, whether from LANG, the category variable itself, or LC_ALL. Your script may subsequently override individual categories using commands like the one at the top of this section.

Keep in mind that the only values a portable script can set the locale variables to is "C" (or its alias "POSIX"), and that this locale does not necessarily have all the properties with which the GNU implementation instills it. Things you can assume in the "C" locale (with the relevant category in parentheses):

- Ranges like [a-z] work in glob patterns and regular expressions, and are based on ASCII codepoint ordering, not natural-language collation nor bogus ASCII-incompatible character sets' (e.g. EBCDIC) ordering. This also applies to character ranges for the `tr` command. (LC_COLLATE)
- The `sort` command sorts based on ASCII codepoint order (LC_COLLATE).
- Case mapping for "I"/"i" is sane, no Turkish mess (LC_CTYPE).
- Dates are printed in the standard traditional Unix ways. (LC_TIME)

And things which you cannot assume or which may be "more broken" in the "C" locale than whatever the existing locale was:

- Bytes outside the "portable character set" (ASCII) are not necessarily characters. Depending on the implementation they may be noncharacter bytes, treated as ISO Latin-1 characters, treated as some sort of abstract characters without properties, or even treated as constituent bytes of UTF-8 characters. This affects whether (and if so, how) they can be matched in globs and regular expressions. (LC_CTYPE)
- If LC_CTYPE is changed, other locale categories whose data depends on the character encoding (for instance, LC_TIME month names, LC_MESSAGES strings, LC_COLLATE collation elements, etc.) have undefined behavior. (LC_CTYPE)
- It is unclear whether POSIX specifies this or not, but the GNU C library's regular expression engine historically crashes if LC_COLLATE is set to "C" and non-ASCII characters appear in a range expression.

As such, it's sometimes safe to replace individual categories such as LC_COLLATE or LC_TIME with "C" to obtain predictable output, but replacing LC_CTYPE is not safe unless you replace LC_ALL. Replacing LC_CTYPE may on rare occasion be desired to inhibit odd and dangerous case mappings, but in a worst case scenario it could entirely prevent access to all files whose names contain non-ASCII characters. This is one area where there is no easy solution.

Removing all exports

```
unexport_all () {
eval set -- `export -p`
for i do case "$i" in
*=*) unset ${i%%=*} ; eval "${i%%=*}=\${i#*=}" ;;
esac ; done
}
```

Using globs to match dotfiles

```
.[!.]* ..?*
```

The first of these two globs matches all filenames beginning with a dot, followed by any character other than a dot. The second matches all filenames beginning with two dots and at least one other character. Between the two of them, they match all filenames beginning with dot except for "." and ".." which have their obvious special meanings.

Keep in mind that if a glob does not match any filenames, it will remain as a single unexpanded word rather than disappearing entirely from the command. You may need to account for this by testing for existence of matches or ignoring/hiding errors.

Determining if a directory is empty

```
is_empty () (
cd "$1"
set -- .[!.]* ; test -f "$1" && return 1
set -- ..?* ; test -f "$1" && return 1
set -- * ; test -f "$1" && return 1
return 0 )
```

This code uses the magic 3 globs which are needed to match all possible names except "." and "..", and also handles the cases where the glob matches a literal name identical to the glob string.

If you don't care about preserving permissions, a simpler implementation is:

```
is_empty () { rmdir "$1" && mkdir "$1" ; }
```

Naturally both of the approaches have race conditions if the directory is writable by other users or if other processes may be modifying it. Thus, an approach like the latter but with a properly restrictive umask in effect may actually be preferable, as its result has the correct atomicity properties:

```
is_empty_2 () ( umask 077 ; rmdir "$1" && mkdir "$1" )
```

Querying a given user's home directory

This does not work:

```
foo=~$user
```

Instead, try:

```
eval "foo=~$user"
```

Be sure the contents of the variable user are safe; otherwise very bad things could happen. It's possible to make this into a function:

```
her_homedir () { eval "$1=~$2" ; }
her_homedir foo alice
```

The variable `foo` will then contain the results of tilde-expanding `~alice`.

Recursive directory processing without find

Since `find` is difficult or impossible to use robustly, why not write the recursion in shell script instead? Unfortunately I have not worked out a way to do this that does not require one level of subshell nesting per directory tree level, but here's a shot at it with subshells:

```
myfind () (
cd -P -- "$1"
[ $# -lt 3 ] || [ "$PWD" = "$3" ] || exit 1
for i in ..?* .[!.]* * ; do
[ -e "$i" ] && eval "$2 \"\$i\""
[ -d "$i" ] && myfind "$i" "$2" "${PWD%/}/$i"
done
)
```

Usage is then something like:

```
handler () { case "$1" in *~) [ -f "$1" ] && rm -f "$1" ;; esac ; }
myfind /tmp handler   # Remove all backup files found in /tmp
```

For each file in the recursive traversal of "$1", a function or command "$2" will be evaluated with the directory containing the file as the present working directory and with the filename appended to the end of the command line. The third positional parameter "$3" is used internally in the recursion to protect against symlink traversal; it contains the expected physical pathname PWD should contain after the `cd -P "$1"` command completes provided "$1" is not a symbolic link.

Seconds since the epoch

Sadly, the GNU `%s` format for `date` is not portable. So instead of:

```
secs=`date +%s`
```

Try the following:

```
secs=$((`TZ=GMT0 date \
+"((%Y-1600)*365+(%Y-1600)/4-(%Y-1600)/100+(%Y-1600)/400+1%j-1000-135140)\
*86400+(1%H-100)*3600+(1%M-100)*60+(1%S-100)"`))
```

The only magic number in here is 135140, the number of days between 1600-01-01 and 1970-01-01 treating both as Gregorian dates. 1600 is used as the multiple-of-400 epoch here instead of 2000 since C-style division behaves badly with negative dividends.

Final remarks

Expect this page of tricks to grow over time as I think of more things to add. It is my hope that these tricks serve to show that is IS possible to write correct, robust programs using the plain POSIX shell, despite common pitfalls, but also that the lengths needed to do so are often extremely perverse and inefficient. If seeing the above hacks has inspired anyone to write a program in a real language rather than sh/Bash/whatever, or to fix corner case bugs arising from the badness of the shell language, I will be happy. Please direct comments, flames, suggestions for more tricks to include, and so forth to "dalias" on Freenode IRC.