A Brief POSIX Advocacy

Shell Script Portability

ARNAUD TOMEÏ

Arnaud Tomeï is a self-taught system administrator who first worked for the French social security administration as a consultant, where he discovered portability issues between AIX and RHEL Linux. He currently works for a hosting and services company in the south of France, specializing in Debian GNU/Linux and OpenBSD, administering 600 systems and the network. arnaud@tomei.fr

Automating things is the most important task a system administrator has to take care of, and the most practical or at least widespread way to do that is probably by writing shell scripts. But there are many flavors of shell, and their differences are a big concern when you have a heterogeneous environment and want to run the same script with the same result on every machine (that's what any sane person would expect). One option is to write POSIX-compliant shell scripts, but even the name might be confusing because POSIX normalizes a lot of UNIX-related things, from system APIs to standard commands, so I will try to clear things up.

The Bestiary of /bin/sh

One remarkable characteristic of Unixes since their beginning is the separation between the base system and the command interpreter. Beside architectural considerations, it allowed a wide diversity of programs to exist, and even to coexist on the same system with the choice given to the users on which one to use.

The first shell available was not surprisingly developed by Ken Thompson, and while it remained the default only for a couple of years, it laid the basis for the functionalities we use today: pipes, redirections, and basic substitutions. It was rapidly improved by Steve Bourne [1] in 1977 and developed into the now widely known Bourne shell. But another competing implementation was released in 1978, the C shell, written by Bill Joy to be closer to the C syntax and to have more interactive features (history, aliases, etc.). Sadly, those two syntaxes were incompatible.

That's when the Korn shell emerged; developed by David Korn and announced at the 1983 summer USENIX conference, it was backward-compatible with the Bourne shell syntax and included a lot of the interactive features from the C shell. Those two main characteristics made ksh the default shell on many commercial versions of UNIX, and made it widely known and used. No major alternative shell was written, and a stable base was reached with the release of ksh88. A new version was shipped in 1993, ksh93, which brought associative arrays and extensibility of built-in commands. Due to its popularity, the Korn shell has seen a lot of forks, including the "Public Domain KSH" pdksh, which shipped on OpenSolaris, most of the open source BSD variants, and even graphic-enabled versions like dtksh [2] and tksh [3].

It took until the late '80s and the beginning of the '90s to see two new shells released: bash in 1989 and zsh in 1990. The first was an effort from the GNU Project to have a free software equivalent of the Bourne shell for the GNU operating system, and the second was a student project of Paul Falstad's [4]. They are both backward-compatible with the Bourne shell but aim at providing more advanced functions and better usability.

A Step by Step Normalization

Back in 1988, the IEEE Computer Society felt the need to standardize tools and APIs to maintain compatibility between systems and started to write what was going to be commonly known as "POSIX," the IEEE Std 1003.1-1988, or ISO/IEC 9945 standard. This document defined very low-level mandatory characteristics of what could be called a UNIX, and

was the foundation of what we now know. It was further expanded to the point where four standards were necessary: POSIX.1, POSIX.1b, POSIX.1c, and POSIX.2, with even longer official denominations. The interesting part for our purposes is the 1992 revision (POSIX.2 also known as IEEE Std 1003.2-1992), which defined the behavior of the shell and the syntax of the scripting language. This norm is based on what was the most available shell at the time which, given the time frame, was still ksh88.

All those standards were finally merged as the result of a vendor consortium (if you thought it was already complex, search for The Open Group history) into one document in 1994: the Single UNIX Specification. The standards have since all become available under the same IEEE Std 1003.1 standard, divided into four sections. The shell scripting language is defined by the XCU chapter, along with standard tools (e.g., grep, sed, or cut) with their options, and those specifications are now maintained both by the IEEE Computer Society and by The Open Group.

Testing Code Portability

Modern shells like bash, zsh, or ksh will all be able to run POSIX-compatible scripts with no modifications, but will not fail when facing nonstandard options or constructs. For example, bash has a POSIX-compatibility mode that can be triggered in three different ways: calling it directly with the `--posix` argument, setting the `POSIXLY_CORRECT` environment variable, and calling `set -o posix` in an interactive session; none of these methods, however, will cause bash to fail to run a script containing a test between double brackets, a bash-only construct, or use the `-n` argument for `echo`. Reading the full XCU specification before writing a script is not even remotely conceivable: the specification's table of contents alone is already 4867 lines long (I'm serious) [5].

> Although setting the `POSIXLY_CORRECT` variable will not make bash behave as a strictly POSIX shell, it will enable other GNU tools like df or tar to use 512-byte blocks (as specified by the norm) instead of one kilobyte by default, which might be useful for a backup script designed to run between Linux and BSD, for example.

Installing all available shells and running the intended script with all of them might sound crazy but is a serious option if you want to look after really specific cases where strict POSIX compliance is not mandatory but portability is.

But for a more generic situation, using a minimal Bourne-compatible shell is a quicker solution: if you are using Debian or a derivative you can use dash, which is installed by default now, or even install posh (Policy-compliant Ordinary SHell) to test the script against, as they will exit with an error when encountering a nonstandard syntax. On almost all other systems (e.g., AIX, HP-UX, *BSD, and Solaris/Illumos), a ksh derivative will be available. Since the XCU standard was written when ksh88 was the most widespread interpreter, chances are that your script will be well interpreted on most platforms if it runs with ksh: granted it is ksh88 and this might not be the case on all systems.

One other option, coming again from the Debian project, is the Perl script `checkbashisms` [6], originally designed to help the transition of the default system shell from bash to dash. It allows for some exceptions by default, as it checks for conformance against the Debian policy [7] first (which allows `echo -n`, for example), but can be forced to be strictly POSIX:

```
$ checkbashisms --posix duplicate-fronted.sh
possible bashism in duplicate-frontend.sh line 144 (echo -n):
      echo -n "Updating server list ..."
possible bashism in duplicate-frontend.sh line 157 (brace
expansion):
mkdir -p $wwwpath/{www,log,stats}
[...]
```

`checkbashisms` has one big limitation, however: it does not check for external tools and their arguments, which can be nonportable.

Finally, there is Shellcheck [8], a tool that does a lot more than just checking portability but also warns you about stylistic errors, always true conditions, and even possible catastrophic mistakes (`rm $VAR/*`). Shellcheck also has an online version with a form to submit the script if you don't want to install the Haskell dependencies required to run Shellcheck.

Built-ins

Some of the errors the previous tools would point out are frequently part of the shell syntax itself, which is often extended for ease of use, but at the expense of compatibility.

read

The `-p` option of `read` is a good example of an extended shell built-in that is frequently used in interactive scripts to give input context to the user:

```
read -p "Enter username: " username
echo "$username"
```

On bash or zsh, it would output something like this:

```
$ ./test.sh
Enter username: foo
foo
```

But it will fail on dash, posh, or ksh because `-p` is not available:

```
$ ./test.sh
read: invalid option -- 'p'
```

Another nonstandard extension of `read` is the special variable `$REPLY`, which contains user input if no variable name is provided:

```
read -p "Enter username: "
echo "$REPLY"
```

This code will also fail on other interpreters:

```
$ ./test.sh
test.sh:2: read: mandatory argument is missing
```

A better version of the above examples would be to use `printf` and explicitly name the variable:

```
printf "Enter username: "
read username
```

Which will give the same output as `read -p` on all shells.

echo

On the last example given with `read`, an alternative would have been to use the following code:

```
echo -n "Enter username: "
read username
```

because `echo -n` does not output a newline. But this option is not portable either, and interpreters on which it is available will likely support the `-p` option of `read`. Actually, the POSIX `echo` does not support any option: as stated by The Open Group, "Implementations shall not support any options."

Some operands are supported, however, and a workaround to suppress the newline would be to insert `\c` at the end: `echo` immediately stops outputting as soon as it reads this operand. But this method, although POSIX-compliant, is not portable either, at least with bash and zsh (only when zsh is called as /bin/sh):

```
$ ./test.sh
Enter username: \c
foo
```

Those two interpreters don't process operands unless echo is followed by the `-e` option, in contradiction with the POSIX specification. That's why it's often recommended to use `printf` instead of `echo`. A rule of thumb is to use `echo` only when no option or operand is needed, or to print only one variable at a time.

getopts

Yes, with an s. Unlike `getopt`, the platform-dependent implementation, `getopts` is well defined and will behave consistently across different systems, with one big limitation: long options are not supported.

test

The `test` built-in, or `[ ]`, obviously has many useful options, too many to be listed here, but two of them were deprecated (actually they were not part of the POSIX norm but of the XSI extension) and are still in use: the binary operators AND and OR, noted `-a` and `-o`.

```
[ "$foo" = "$bar" -a -f /etc/baz ]
[ "$foo" = "$bar" -o -f /etc/baz ]
```

Because they were ambiguous, depending on their position in the expression, and could be confused by user input, they have been marked obsolescent. Moreover, they could be easily replaced by the equivalent shell operators: `&&` and `||`. Another nonportable syntax often used is the bash extended `test`, delimited by double brackets, which must also be avoided for POSIX scripts.

Don't Forget the Standard Tools

The shell language on its own would not have met its success without all the tools it can use to process files and streams. Did I mention that such tools as grep, sed, and cut are mandatory in the XCU standard? They are, and their necessary options are even listed. But we're used to some options not necessarily being available on all systems.

cut

I've never used this option, but I've seen it in others' scripts a couple of times, so I guess it is worthy to mention that `--output-delimiter` is GNU-specific:

```
cut -f 1,2 -d ':' --output-delimiter ',' foo
```

will work with GNU coreutils but will throw an error on other systems:

```
cut: unknown option -- -
```

The alternative in this case is pretty obvious and straightforward: pipe it to sed.

```
cut -f 1,2 -d ':' foo | sed -e 's/:/,/g'
```

sed

One really useful flag of `sed`, the `-i` option, is sadly not defined, and that can lead to some surprising errors even on systems supporting it: for example, a small script I wrote on my Linux machine to run on my girlfriend's Mac produced the following:

```
$ ./spectro-split.sh lipo-ctrl_1.csv
sed: 1: "lipo-ctrl_1.csv": invalid command code .
[...]
```

In the script, `sed` was used to replace the decimal mark in spectrometry raw data, for later analysis by another tool:

```
sed -i 's/,/./g' $column.txt
```

With GNU `sed`, the `-i` option takes an optional string as a suffix for a backup copy of the file being edited, but on Mac (and FreeBSD) the suffix is mandatory even if empty; here the substitution pattern was misunderstood, so I had to use this more portable (but still non-POSIX) syntax:

```
sed -i '' -e 's/,/./g' $column.txt
```

`read -p "Input:" variable`	The `-p` option is not portable. Actually, the only POSIX option to read is `-r`.
`read; echo $REPLY`	The `$REPLY` special variable is interpreter-specific and is not always available.
`echo -n Foo`	Portable `echo` does not support any option; `printf` should be preferred.
`sed -i "s/foo/bar/" file`	Although really useful, this option is not standard and behaves differently depending on the system.
`cp /etc/{passwd,shadow}`	Brace substitutions are commonly used with bash and zsh but are not available on ksh and POSIX.
`if [[ -e /tmp/random-lock ]]`	Double brackets are bash-specific.
`touch /tmp/$RANDOM.tmp`	The special variable `$RANDOM` is not available everywhere.
`if [ $var1 == $var2 ]`	String comparison takes only one equals sign. Moreover, doubling it might be interpreted as a variable (named "=") assignment, which can't be done in a test.
`foo () { local var1=bar }`	Scoped variables are not defined by the XCU. The `unset` routine might be used instead if necessary.
`foo=((foo++))`	Works only with bash, should be replaced by `foo=$((foo+1))` or `$((foo=foo+1))` when used in another expression (for example, `ls -l $((foo=foo+1))`).
`[ "$foo" = "$bar" -a -f /etc/baz ]`	Should be replaced by `([ "$foo" = "$bar" ] && [ -f /etc/baz ])`
`[ "$foo" = "$bar" -o -f /etc/baz ]`	Should be replaced by `([ "$foo" = "$bar" ] || [ -f /etc/baz ])`
`ls -1 ~/foo`	Often used in interactive sessions, the tilde should be banned from script as it is not expanded by all shells.

Table 1: A cheat-sheet with a quick check for the most common errors in shell scripts

This syntax will run, at least on Linux, Mac, FreeBSD, and OpenBSD, but it will throw an error on AIX, Solaris, and HP-UX, whose `sed` does not know the in-place editing option. An alternative would be to use `perl` if available:

```
perl -pi -e 's/,/./g' $column.txt
```

Or to rely only on POSIX tools:

```
sed -e 's/,/./g' $column.txt > $column.txt.new && mv
$column.txt.new $column.txt
```

This last option might not be prettiest, but it is the most portable and reliable: it does not require external tools or nonstandard options, and it actually does the same as the in-place argument, without risking silent corruption in case of disk space exhaustion.

sort and uniq

Sort is often used in conjunction with uniq, which can only process adjacent lines, and, contrary to most of the previous options we've seen, one of sort's options is often wrongly thought to be nonportable although it is perfectly standard and more efficient:

```
sort -u foo.txt -o bar.txt
```

which is POSIX-compliant and portable, and is more elegant than piping the result into uniq before redirecting the output.

Common Mistakes

Table 1 provides a small cheat-sheet to quickly check for the most common errors in shell scripts:

Conclusion

Even if it requires greater discipline, writing POSIX-compliant scripts, as well as knowing the syntax and the options of the tools used, is a good starting point for portability: it will produce higher quality scripts and, in some marginal cases, might even lead to better performance by using a limited but optimized interpretation. Of course, as in the echo example, even with standards some specific features can interfere, but by sticking closely to the norm, those situations will be limited and trivial to correct most of the time.

Resources

[1] http://www.unix.org/what_is_unix/history_timeline.html.

[2] dtksh was a fork able to manipulate Motif widgets and was included with the CDE desktop.

[3] tksh, like dtksh, was a fork adding graphic capabilities to ksh but with the Tk widget toolkit instead of Motif.

[4] http://zsh.sourceforge.net/FAQ/zshfaq01.html.

[5] http://pubs.opengroup.org/onlinepubs/9699919799/utilities/contents.html.

[6] http://sourceforge.net/projects/checkbaskisms/.

[7] http://www.debian.org/doc/debian-policy/ch-files.html.

[8] http://www.shellcheck.net/about.html.