Shell scripts in 20 pages

A guide to writing shell scripts for C/C++/Java and unix programmers

Russell Quong

April 10, 2001— Document version 2001a

Keywords: Shell documentation, Shell tutorial, Shell beginners, Guide to shell scripts. (For internet search engines.)

This is a work in progess; you will find unfinished sections, paragraphs and even sentences.

Contents

This is a work in progess; you will find unfinished sections, paragraphs and even sentences.

1 Introduction

A shell is a program that reads commands and executes them. Another name for a shell is a *command interpreter*.

1.1 My own history with Unix shells

I started using `csh` many years ago as an undergraduate. I could not figure out the `/bin/sh` syntax, in particular `${var:-val}`. Despite encountering many mysterious /bin/sh scripts and having to use make, which uses `/bin/sh`, I resisted and wrote csh shell scripts and used the `tcsh` as my login shell. Finally, I couldn't stand csh scripting any more and "re"-learned /bin/sh.

2 The Operating System

A typical home user is completetely insulated from the OS. So when someone says, "I really like computer XXX (e.g. the Mac or Windows 95 or Unix)", they are NOT talking about the operating system. Rather they are talking about the user interface or *UI* on top of the OS.

Externally, an operating system is a programming API, typically in the C programming language. The API lets some other program do low level operations like:

Unix API call	Description
exec	run a program, given a fully specified command
open	open file or some other I/O stream
read/write	read or write data to a file descriptor

Directly interacting with the OS is incessantly tedious, as the OS is very picky and works at a low level. Its akin to communication via Morse Code.

Instead, people use graphical environments (like the Mac or Win32) or command line interpreters like Unix *shells* or the (very minimal) MSDOS prompt.

3 Interactive Use of shells

If you type commands to the shell, you are running an *interactive shell.* The shell features beneficial for interactive use are different from those needed when running a script, in which the shell reads preset commands.

For interactive use, I prefer `bash` and `tcsh`, because they have easily accessible filename and command completion and good editing capabilities. Note the phrase *completes* means that if you partially type a word, the shell will either

On a ...	the shell does ...
unique match	finishes the rest of the word
multiple matches	shows all possible completions of the partial word

The feature I rely on in order from most important to least important are

What	Keys	Description
filename completion	TAB	completes partially typted file,path names
command history access	CNTL-n and CNTL-p	fetch a previous command and edit it
command completion	TAB	completes the command name
CDPATH		variable of directories to search when you type `cd`

3.1 Testing if it is an interactive shell

All shells read a startup file, when they err startup, in which you can set and customize various settings (variables, aliases, functions, prompt, terminal settings).

If you "login" you get an interacitve shell and you probably want to (heavily) customize its use. However if you run a command remotely, say via the Unix rsh, rcp or rsync commands, you start a non-interactive remote shell to run the remote command and you usually to set the path correctly. In particular, you **must not** print any messages when the remote shell start up.

To test if a shell is interactive, (i) test for the existance of the shell prompt string or (ii) run `tty -s` which returns true (0) for an interactive shell, as there is an underlying tty.

3.2 Setting your prompt

Set the PS1 (prompt string 1) variable. In PS1, the following escape sequences can be used. I have listed only the most useful; see the man page for a full listing.

\h	hostname	\u	user name
\H	hostname.domainname	\w	current working directory (CWD)
\n	newline	\W	basename of CWD
\r	carriage return	\!	history number of the current command
\s	shell name	$	if UID is 0 (root), use a '#', else use a '$'

My personal preference for PS1 is "`\h \! \w\$ `".

3.3 Real or physical paths (bash)

In the presense of symbolic links and home directories, bash by default uses the logical directory structure. To force bash to show the actual, real or physical directory structure use `cd -P .`

4 Processes or jobs or tasks

Each command run by a shell is a separate child process, whether run interactively or via a script. The child process inherits various values, such as (i) who is running the command, (ii) the current directory, and (iii) the environment variables.

5 Command line processing

The command line parameters to a script are stored in the nearly identical variables `$*` and `$`. The following table summarizes the variables you would use for command line processing. For the example values, assume you wrote a ba/k/sh script `/usr/bin/args.sh` and ran it as shown below.

Variable	Meaning	Ex: /usr/bin/args.sh -t two "let's go"
`$*`	Command line args	-t two let's go
`$`	Command line args	-t two "let's go"
`$#`	Number of args	3
`$0`	Name of script	/usr/bin/args.sh
`$1`	First arg in $*	-t
`$2`	Second arg in $*	two
`$3`	Third arg in $*	let's go
`$4`	Fourth arg in $*	(empty)

To parse command line arguments, I prefer using the `case` construct, instead of `getopts` in ba/ksh, because it is easier to understand, handles all flag situations and will work in `sh` too.

Here is a more realistic example for a command that takes five possible flags, in any order. For the `-n` flag, we set a shell variables to remember the state; this technique is common.

Flag	Description
-o OUT	send output to file OUT
-n	show what you would do but do not do it
-v	give more output, each -v increases verboseness
-l	same as -verbose
-version	show the version and quit

Here is the code snippet. Notice the `shift 2` and the `$2` for the -o flag. Notice that any flag beginning `-ver` is considered the same are `-version`.

```
nflag=0
vlevel=0
OUT=

while [ $# -gt 0 ]; do
  arg=$1
  case $arg in
   -o ) OUT=$2 ; shift 2 ;;
   -n ) nflag=1 ; shift ;;
   -l | -v ) vlevel=$(( vlevel+1 )) ; shift ;;
   -ver* ) echo "Version $version"  ; exit 1 ;;
   * ) echo "Saw non flag $arg" ; break ;;
  esac
done

... continue processing remaining args ...
```

6 Variables

Assign to variables using = with no surrounding space between variable name and value. Access the value of variable by using a $ before the variable name, e.g. `$showWarnings`.

```
 $ color=red                    # correct
 $ color= red                   # WRONG, space after equal
 $echo I want a $colorder than $color shirt  # I want a redder than red  shirt
```

If there is any ambiguity what the variable name is, you can use `${varname}`. In the preceding example, see how we echo'ed "redder".

Shell variables contain string values, though you can use the values numerically. Normal variables are local/private to the shell and are only accessible (or visible) to the shell in which they are set.

6.1 Environment (or public) variables

Public or *environment variables* are accessible by all child processes/jobs of the shell.

6.2 Common environment variables

PATH	dirs to search for commands
SHELL	path of shell
TERM	terminal type
USER	user (login) name
HOME	home dir of user
PS1	main interactive prompt (ba/k/sh)
CDPATH	dirs to search when you do a cd or pushd

6.3 Conditional Assignment

Many times we want a conditionally assign a value to a variable VVV. The syntax `VVV=${ZZZ:-DefaultVal}` is equivalent to

```
if [ "$ZZZ" != "" ]; then
  VVV=$ZZZ
else
  VVV=DefaultVal
fi
```

Thus we assign the value of $ZZZ to VVV if ZZZ has a value, otherwise we assign DefaultVal.

7 Expansions

The shell will expand the following strings.

Expansion	You type	the shell generates
Tilde	~	your home directory ($HOME)
Tilde	~alison	home directory for user `alison`
Brace	{1,blue,dot.com}	1 blue dot.com
Brace	x{0,11}y{2,33,}z	x0y2z x0y33z x0yz x11y2z x11y33z x11yz

7.1 Globbing

On a command line, the following characters have special meaning. This process is called *globbing*.

*	Any sequence of characters not containing a /
?	Any single character
[aeiou]	Any single a, e, i, o or u character
[~aeiou]	Any character not a, e, i, o or u

A leading * and ? will not match a leading `.`, to prevent from matching `.` and `..` which would normally cause havoc.

The shell reads command line arguments and applies globbing to the list of filenames in the current directory, by default. Thus most people think globbing is the same as file name expansion.

8 Arithmetic

In ksh,bash use `$(( expression ))` to perform arithmetic operations. Note that inside `$(( expression ))`, you do not need to prefix variables with a $.

```
echo 'using $(( var + 1 )) style'
i=0 j=0 k=0 ll=0
while [ $i -le 4 ]; do
  echo $i $j $k $ll
  i=$(( $i + 1 ))         # OK to use $i
  j=$(( j + i ))          # just 'j' is fine, too
  ll=$(( k += i ))
done
```


9 Embedding verbatim text

If you need to print out text nearly verbatim, e.g. you need to generate a standard 40-line disclaimer, then use a *here document*. The general notation is as follows, where you can use any string of your choice to replace `END_DELIMITER`.

```
cat <<-END_DELIMITER
  verbatim text
  ...
END_DELIMITER
```

The optional minus sign - before END_DELIMITER tells bash to ignore beginning whitespace in each line of the verbatim text, so you can indent this text. The shell will evaluate shell variables, backtick expansion and (bash) arithmetic expressions in the verbatim text. To suppress this evaluation, put single quotes around END_DELIMITER.

Here is a more realistic example, where we generate an HTML header to the file `$out`.

```
htmldoc=...
cat >> $out <<-EOS
  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">
  <HTML><HEAD>
  <TITLE>The HMTL doc: $htmldoc on 'date'</TITLE>
  </HEAD>
EOS
```

10 Process management

The shell can manage several processes (jobs/tasks). The current or *selected* job (which must be suspended or running the background) is marked with a + when you type `jobs`. A foreground process is one that is currently running and has control of the terminal. E.g. your keyboard input goes to the foreground process.

Command	What
jobs	list the jobs running on this shell
bg [Proc]	run selected job in the background
fg [Proc]	run selected job in the foreground
CTL-Z	suspend the current job

For example, I had three netscape's and some other processes running and typed jobs and got:

```
hostest 972 ~/bin$ netscape &     # jobs5
hostest 973 ~/bin$ fg 6           # resume running less
CTL-Z                             # suspend less
hostest 974 ~/bin$ jobs
[1]   Running                 ( cd ~; netscape-v304 ) &  (wd: ~)
[3]   Running                 oclock &  (wd: ~)
[4]   Running                 netscape -geometry =720x700 &  (wd: ~/ftp)
[5]-  Running                 netscape -geometry =720x700 &  (wd: ~/ftp)
[6]+  Stopped                 less -c -s -M ../summary-01-21-01.out  (wd: /tmp/tmp/tmp)
hostest 974 ~/bin$ fg
```

In a shell script, the `wait` command will wait for all background jobs to finish before proceeding.

```
...
commandOne -a -x -b &
commandTwo -vv file1 file2 &
wait    # waits for the two previous commands to finish
...
```

11 Relational operators

12 Syntax of control structures

When issue control constructions such as for, while or if statements, the ba/k/sh shells need either a `newline` or a `;` (semi-colon) delimiters to parse your input. In the following, ▪ means either a `newline` or a `;`. Thus the following four if-statements are all equivalent.

```
if EXPR $▪$ then STMT(S) $▪$ fi   # general syntax
if [ -f /bin/mv ]$▪$ then echo "looks like unix" $▪$ fi
if [ -f /bin/mv ]; then echo "looks like unix" ; fi
if [ -f /bin/mv ]; then
  echo "looks like unix"
fi
if [ -f /bin/mv ]
    then
  echo "looks like unix" ; fi
```

The control constructs are

if	if EXPR ▪ then STMT(S) ▪ fi
if else	if EXPR ▪ then STMT(S) ▪ elif EXPR ; then STMT(S) ▪ fi
for	for VAR in LIST ▪ do STMT(S) ▪ done
while	while EXPR ▪ do STMT(S) ▪ done
case	case VALUE in [[PATTERN [— PATTERNS]) STMTS ;;] esac

12.1 If

12.2 Case

The case statement lets you determine if a string SSS, which is almost always contained by a variable VVV, matches any of several ”cases”. For example we test which state a traffic signal is in via:

```
case $trafficLight in
  red ) echo "stop" ;;
  yellow | orange ) echo "decision time..." ;;
  green ) echo "GO" ;;
  default ) echo "Unknown color ($trafficLight)" ;;
esac
```

13 Reading input

14 Some useful functions

15 Tips for writing scripts

15.1 Seeing variables

Here is a handy ba/k/sh function that prints out the values of variables given their *names*. I list it first since I use it often.


```
   # showVals varname [ varname(s) ]
showVals () {
  for i in $*; do
    eval echo "\ \ \#\#  $i=\($$i\)"
  done
}
...
showVals USER HOME PS1 outFile nflag
```

15.2 Doing glob matching

In bash and sh you must use case. Here is a handy function, `globmatch`, that lets you glob match anywhere.

```
  # Ex:  matches [ -q ] string globpattern
  # Does $1 match the glob expr $2 ?
  #     -q flag = set return status to 0 (true) or 1 (false)
  #  no -q flag = echo "1" (true) or "0" (false)
  # Unfortunately, the return status is opposite from the echo'ed string
globmatches () {
  if [ $1 = "-q" ]; then
    shift
    case "$1" in
      $2 ) true ;;
      * ) false ;;
    esac
  else
    case "$1" in
      $2 ) echo 1 ; true ;;
      * ) echo 0 ; false ;;
    esac
  fi
}

if globmatches -q $file "*.tar" ; then
  echo "Found a tar file"
elif globmatches -q $file "*zip" ; then
  echo "Found a zip file"
if
```

15.3 Extracting data

You will often get data with multiple fields or *words*. To extract and print the K-th word, where the first word is K=1, use either of

```
  set  - ... | echo $K        # purely shell based solution
  ... | awk '{ print $K; }'   # requires awk or nawk or gawk
  ... | cut -f K -s ' '       # least preferred method
```

For simple tasks, using the shell built in `set` is easiest. It is better to use awk (or gawk or nawk) because awk handles words separate by spaces and tabs correctly. The `cut` program (as of 2001) is quite stupid and assumes precisely one space between words.

To extract the K-th, M-th and P-th words, use either of

```
awk '{ print $K, $M, $P; }'
cut -f K,M,P -s ' '
```

15.4 Precede debugging/verbose messages with common prefix

I personally like '#' because this is the comment character for both scripts and perl. Sometimes, one shell script generates a second script, in which case I must precede optional messages with a comment character.

```
echo "# Do not modify this script.  Auto-generated by master script $0"
...
echo "# FYI, variable color=$color"
```

15.5 No full paths

Do not put full paths in your script, because if the path is wrong, say on a different OS/platform, you have to change all the paths in your script. Instead augment the PATH as necessary. E.g. if your script need to run `/usr/ucb/whoami`, then put the following in your script. On a different platform, you only have to augment the PATH differently.

```
PATH=/usr/ucb:$PATH
...
... whoami ...
```

15.6 Simple regular expression substitution

To change or *substitute* the text FROMX to TOX, use `sed`. You can specify regular expressions for FROMX. A

```
sed -e "s/FROMX/TOX/" # subst first occurrence
sed -e "s/FROMX/TOX/g"        # subst all occurrences
  # strip off domain name (remove .in20pages.com)
echo "speedster.in20pages.com" | sed -e "s/[.].*$//"
  # keep domain name (remove speedster.)
echo "speedster.in20pages.com" | sed -e "s/^[^.]*[.]//"
```

15.7 Floating point math and base calculations

Use `dc`, the postfix or RPN calculator, or `bc` which takes human familiar infix notation. I strongly prefer `dc`. In the following examples, I store the results in variables `rx`, `rhex` and `wacko`. In `dc`, the commands `i`, `o`, `k` mean set the input base, output base, calculation precision, respectively. In `dc`, `p` means print the top of the stack.

```
    # calculate (2.718 + 1.414) / (3.141 - 2)  to 5 decimal places
rx=‘echo 5 k  2.718 1.414 + 3.141 2 - / p | dc‘
  # convert 12345 to hex (base 16)
rhex=‘echo 16 o  12345 p | dc‘
  # convert 12345 base 7 to octal (base 8) [All your base are belong to us]
wacky=‘echo 7 i 8 o  12345 p | dc‘
```

15.8 One liners

- Use `pushd` and `popd` to change and restore the current directory.
- Use `mkdir -p` to create directories.
- Use `case` to do glob matching.