Bash Programming Reference

1 General Notes

- Words in italics denote concepts that are further explained, often in the same section, but possibly elsewhere - in that case a cross-link is provided.
- Roman `[ ] *` stand for themselves; in italics they denote optional elements (*[]*: 0 or 1) or repetition (***: one or more).
- By "ordinary" parameters I mean non-array, non-nameref parameters.

2 Operation

1. Input
2. Tokenizing
3. Parsing
4. Expansions
5. Redirections
6. Execution
7. (Wait)

3 Tokenizing

Split input into *tokens*, which are either *words* or *operators*:

- **words** do not include un*quote*d (8) *metacharacters* (`space`, `tab`, `newline`, `|`, `&`, `;`, `(`, `)`, `<`, `>`; see 7). The following words are *reserved* and have special meaning:
  `! case coproc do done elif else esac fi for function if in select then until while { } time [[ ]]`
- **operators** contain at least one unquoted metacharacter, and are either *control* or *redirection* operators
  - control operators are `newline`, `||`, `|`, `&&`, `&`, `|&`, `;`, `;;`, `;&`, `(`, and `)`.
  - redirection operators are `>`, `>>`, `>|`, `>&`, `&>`, `&>>`, `>&-`, `<&`, `<&-`, `<<`, `<<-`, `<<<`, `<>`

IOW, tokens are delimited by whitespace or by **metacharacter/non-metacharcter boundaries**, e.g. `ls|wc` → `ls`, `|`, `wc` because `|` is a metacharacter whereas letters aren't, so `ls|wc` cannot be a word; by contrast, `ls-l` is parsed as single word, so space is needed to obtain two tokens: `ls -l`.

4 Parsing

The tokens are parsed into *commands*, of which there are the following four kinds (see the shell grammar for details):

- Simple commands: *`[assignment*] program [arg*] [&;]`*
- Pipelines: `cmd` *`[| cmd]*`* (or `|&` to redirect `stderr` too)
- Lists: ≥ 1 pipelines; precedence: `&&`,`||`; `&`,`;`
  - `ppln1` *`[&& ppln2]`* 2 iff 1 succeeds
  - `ppln1` *`[|| ppln2]`* 2 iff 1 fails
  - `ppln1` *`[;`* `ppln2]* 2 waits for 1
  - `ppln1` *`[&`* `ppln2]]` doesn't wait (bg)
- Compound commands (9):
  - Loops
  - Conditionals
  - Groupings

5 Expansions

1. Brace (5.1)
2. Tilde (5.2), parameter (5.3), arithmetic (5.4), command (5.5), process (5.6)
3. Word splitting (aka Field splitting – 5.7)
4. Filename expansion (aka Globbing – 5.8)
5. Quote removal (5.9)

- Abbreviated below as B, T, V, A, C, P, W, F, and Q, respectively.
- **Rule of thumb**: expansions that can increase the number of words (B, W, F) do not occur where a single word is expected.

5.1 Brace Expansion

A prefix and suffix (both possibly empty) are affixed to each of a set of strings. This is either an explicit *list*, or a *sequence*.

5.1.1 `{,}` List

Generate lists of strings, normally with a common prefix, suffix, or both.

expression	**value**	**comment**
`pr{A,B,C}`	`prA prB prC`	prefix
`{A,B,C}su`	`Asu Bsu Csu`	suffix
`pr{A,B,C}su`	`prAsu prBsu prCsu`	both

5.1.2 `{..}` Sequence

Generate sequences, possibly with a common prefix, suffix, or both. Specify start, stop, and an optional step.

expression	value	comment
`{1..5}`	`1 2 3 4 5`	
`f{1..4}.c`	`f1.c f2.c f3.c f4.c`	affixes
`{5..1}`	`5 4 3 2 1`	reverse
`{01..5}`	`01 02 03 04 05`	fixed width
`{1..10..2}`	`1 3 5 7 9`	step
`{a..e}`	`a b c d e`	character

Nesting is possible: `{a,b,c{1..3}}` → `a b c1 c2 c3`

Note: contrary to *filename expansion*, the generated strings do **not** have to match filenames.

5.2 Tilde Expansion

When an unquoted `~` occurs at the beginning of a word, all characters between that `~` and either the first unquoted `/` or the end of the word constitute a *tilde prefix*, expanded as follows:

tilde prefix	value
`~`	`$HOME`
`~user`	`user`'s `$HOME`
`~+`	`$PWD`
`~-`	`$OLDPWD`

5.3 Parameter Expansion

Parameter expansion is introduced by `$` (optional in *shell arithmetic* (12) contexts). For *setting* parameters, see 10.

5.3.1 Ordinary Parameters

expansion	value
`${`*var*`}`	value of `var` (as a string)
`$var`	short for `${var}`, ok iff not prefix of longer word
`${#var}`	string length of `$var`

5.3.2 Handling null and unset

A parameter is *null* if its value is the **empty string**; it is *unset* if it has **no value**.

expansion	value
`${var-def}`	if `var` is unset, return `def`, otherwise return `$var`
`${var=def}`	if `var` is unset, return `def` *and set `var` to `def`*, otherwise return `$var`
`${var+def}`	if `var` is unset, return empty, otherwise return `def`
`${var?msg}`	if `var` is unset, expand `msg` and write it to `stderr`, then exit (unless interactive).

Operators `:-`, `:=`, `:+` and `:?` work like their colon-less counterparts, but they check for null as well as unset. `def` and `msg` undergo TVCA expansion.

5.3.3 Expansions involving `@` or `*`

Expansions of the form `$@`, `${`*ary*`[@]}`, `${!`*ary*`[@]}`, `${`*ary*`[@]`*pos*`:`*len*`}` as well as their `*` variants mean "all the elements" in some collection (the positional parameters, the values in an array, the keys in an array, or the values in an array slice, respectively). They expand either to a **single word** containing all the values, or to **separate words**, one per value, depending on context and quoting:

- in contexts where **word splitting is not done** (see 5.7.2), both `@` and `*` constructs expand to a single word containing all values, irrespective of quoting.
- in contexts where word splitting is done,
  - *unquoted* `@` and `*` constructs both expand to separate words, the expanded words are themselves subject to word splitting and filename globbing.
  - *quoted* `*` constructs expand to a single word containing all values, while `@` constructs expand to separate words; in neither case does further word splitting or globbing occur.
- when the expression expands into a single word, the values are separated by spaces in `@` constructs, and by the first character of `$IFS` (or nothing if null) in `*` constructs.
- when the expression is embedded in a word, and expands to separate words, the part of the word before (resp. after) the expression remains prefixed (resp. suffixed) to the first (resp. last) word, *e.g.* if `a=(1 2 3)` then `"X${a[@]}Y"` expands to `X1 2 3Y`.

Note: do not confuse the above expansions with `${`*ary*`[@]/`*tgt*`/`*rep*`/}` and other similar forms. The latter are *substitutions* (5.3.7) and always expand to separate words.

5.3.4 Positional Parameters: `$1`, `$2`, ...

At the top level, `$1`, `$2`, ... contain the program's 1st, 2nd, etc. arguments (if any, otherwise unset); in a function body they refer to the function's arguments. They can be reset, but only with `set` and `shift`. Use braces for arguments beyond 9th: `$10` = `${1}0` ≠ `${10}`.

Within a *function* (14) body, these refer to the arguments (if any) passed to the function.

5.3.5 Some Special Parameters

All are read-only.

param	meaning
`$0`	pathname of script
`$*`, `$@`, `"$*"`, `"$@"`	positional parameters (= arguments). See 5.3.3.
`$#`	number of positional parameters
`$?`	exit status of last foreground pipeline
`$!`	PID of last asynchronous command

5.3.6 Substring Expansion

expansion	value
`${str:pos}`	substring of `$str` starting at `pos`
`${str:pos:len}`	`$len`-character substring of `$str` starting at `pos`.

`pos` and `len` undergo arithmetic expansion - if negative, they both count as *position* from the end of `$str` (use whitespace to avoid confusion with `:-` (`${str: -1}` ≠ `${str:-1}`).

5.3.7 Substitutions

expansion	value
`${str/tgt/rep}`	replace *1st* instance of `tgt` with `rep`
`${str//tgt/rep}`	replace *all* instances of `tgt` with `rep`
`${str/#tgt/rep}`	replace `tgt` with `rep` if `tgt` matches start of `str`
`${str/%tgt/rep}`	replace `tgt` with `rep` if `tgt` matches end of `str`
`${str#tgt}`	delete shortest match of tgt at start of `str` (##: longest)
`${str%tgt}`	delete shortest match of tgt at end of `str` (%%: longest)

`rep` and `tgt` undergo T, V, C, A (see 5); `tgt` (expanded) is a glob (13) pattern.

5.3.8 Arrays

sub below ("subscript") stands for either an integer or a string, for indexed or associative arrays, respectively.

expression	value
`${array[sub]}`	the value of array *array* at index or key *sub*.
`${a[@]}`, `${a[*]}`	`${a[1]} ... ${a[n]}`
`"${a[*]}"`	`"${a[1]}s${a[2]}s...${a[n]}"`; separator `s` is the 1st char of `$IFS` (`space` if unset, empty if null)
`"${a[@]}"`	`"${a[1]}" "${a[2]}" ... "${a[n]}"`
`"${#array[sub]}"`	length of `"${array[sub]}"`
`${#array[sub]}`	length of `${array[sub]}`
`${#a[@]}`, `${#a[*]}`	number of elements in *a*.
`${!a[@]}`, `${!a[*]}`	like `${a[@]}`, but with the **subscripts** instead of the values

5.3.9 Indirect Expansion

Variables can hold the names of other variables.

expansion	value
`${!name}`	(if *name* is a *nameref*) the name of the variable referenced by *name* (if *name* is not a *nameref*) the expansion (TVCA) of `$name` taken to be a parameter name

5.4 Arithmetic Expansion

`$((expr))` evaluates to the value of `expr` according to *shell arithmetic* (12).

5.5 Command Substitution

`$(command)` evaluates to the output of `command` (run in a subshell).

5.6 Process Substitution

Allows the substitution of a process for a filename argument.

- for reading: `<(list)`, *e.g.* `diff <(sort f1) <(sort f2)`
- for writing: `>(list)`, *e.g.* `tee > >(wc -l) <f1 | gzip -`

5.7 Word Splitting

aka "field splitting"

The result of **most** (but see 5.7.2) *unquoted* expansions is split using the characters in `$IFS`. Any `space`, `tab`, or `newline` found in `$IFS` is called *IFS whitespace*.

- sequences of `IFS` whitespace separate fields
- `IFS` whitespace at the beginning or end of fields is removed

- any non-whitespace character in `$IFS` separates fields (resulting in empty fields if these characters are adjacent)
- if `IFS` is unset, it behaves as if it were `<space><tab><newline>` (the default)
- if `IFS` is null, no word splitting is performed.

Example If `var=' A  B::C #D'`, then we have

IFS	$var	#words
(null)	' A B::C #D'	1
(default or unset)	A, B::C, #D	3
':'	' A ', '', 'C D#'	3
': '	A, B, '', C, '#D'	5
'#'	' A B::C ', D	2
'#: '	A, B, '', C, 'D'	5

5.7.1 Null Argument Removal

`""` and `''` are kept unchanged, except when occuring as part of words (in which case they are removed); but *unquoted* (8) null words resulting from expansion are removed, *e.g.* (assuming `var` is unset or null):

expression	expansion
'', ""	''
''A, A''	A
$var	nothing
"$var"	''

5.7.2 No Word Splitting...

Word splitting **does not occur** in: assignments (`=`) (except arrays: see 11), `""`, `$(())`, `case`, `(())`, `<<<`, `[[]]` or in words not resulting from expansion.

5.8 Filename Expansion

Words resulting from *word splitting* (5.7) and containing `*`, `?` or `[` are treated as *patterns* (13) and matched against filenames ("globbing"). A pattern expands to the list of matching filenames, if any; otherwise it remains unchanged.

5.9 Quote Removal

All *unquoted* (8) `\ ' "` that did not result from an expansion (IOW, were already present before the Expansions phase (2, 5)) are removed.

6 Redirections

Before a command is run, its input and output streams may be *redirected* (to other files, or to nothing, etc. – see table below). Redirections may appear before, after, or even within the command. There may be more than one, and they are evaluated from left to right – **order matters**.

TODO the following `[]` should be italicized.

operator	behaviour
`[m]<file`	open `file` for reading on fd `m`
`[n]>[|]file`	open `file` for writing on fd `n`; `>|` ignores `noclobber`
`[n]>>file`	open `file` for appending on fd `n`
`&>file`	`>file 2>&1`
`>&file`	preferred form of `&>file`
`&>>file`	`>>file 2>&1`
`[m]<&int`	make fd `m` a copy of input fd `int`.
`[m]<&-`	close fd `m`.
`[n]>&int`	make fd `n` a copy of output fd `int`.
`[n]>&-`	close fd `n`.
`[m]<&int-`	`[m]<&int int<&-`
`[n]>&int-`	`[n]>&int int>&-`
`[m]<>file`	open `file` for reading and writing
`[m]<<<string`	expansion (all but W and F - see 5) of `string` on fd `m`

- `[n]>file` **erases** ("clobbers") `file` if it exists (and creates it otherwise)
- `m` defaults to `0` (`stdin`), `n` to `1` (`stdout`).
- `file` and `int` undergo all expansions; `int` must expand to an integer.

6.1 Here Documents

Read from the script itself until a line consisting exactly of `delimiter`. This becomes the input of fd `m` (defaults to `0`).

```
[m]<<[-]end
   code
delimiter
```

- `end` undergoes no expansion
- `delimiter` is `end` after quote removal
- `end` unquoted: `code` is expanded (V, C, A (5)) – use `\` to escape, `\newline` ignored)
- `end` with quotes (`'` or `"`, anywhere): `code` is not expanded.

---

7 Special Characters

The following characters can have special meaning.

characters	function/category
`space tab newline`	whitespace metacharacters
`| & ( ) < > ;`	other metacharacters
`* ? [ ]`	glob/test
`" ' \`	quoting
`$ ` `	expansion (V, C, A - cf. 5)
`#`	comment
`=`	assignment
`!`	logical NOT

Some characters can be special in several contexts.

Special characters regain their normal ("literal") status if *quoted* (8). **Rule of thumb**: a character is special (in a given context) iff quoting it makes a difference.

8 Quoting

Makes special characters literal.

- \ ("escape") makes the next character literal, except `newline`
- all characters between ' are literal; no ' can occur between '
- all characters between " are literal, except `$`, `` ` ``, \ (in this case, only before `" $ ` \ newline`)
- ANSI-C: `\n`, `\t` within `$'string'` → `newline`, `tab`, etc.

---

9 Compound Commands

Compound commands control program flow. They begin and end with a *reserved word* or *control operator* (3).

9.1 Loops

9.1.1 *for* loop - variant 1

`for` *name* [`in` *words...*] `do` *list*; `done`

words... are expanded (all expansions - 5); *list* is executed once for each item in the resulting list, binding *name* to each in turn. If `in words...` is omitted, it defaults to `in "$@"`.

9.1.2 *for* loop - variant 2

`for ((`*expr1*`;`*expr2*`;`*expr3*`)); do` *list*`; done`

expr1 is evaluated once; then *expr2* is evaluated repeatedly until it is 0; as long as it is nonzero *list* then *expr3* are evaluated. *expr[123]* are evaluated using *shell arithmetic* (12).

9.1.3 `while` loop, *until* loop

`while` *list1*`; do` *list2*`; done`

Execute *list2* while the exit status of *list1* is zero.

`until` *list1*`; do` *list2*`; done`

Execute *list2* while the exit status of *list1* is nonzero.

Loop control: `break [n]`, `continue [n]` - break out, or jump to the next iteration, of *n* nested loops (default 1).

9.2 Conditionals

9.2.1 *if - then - else*

```
if test-list-1; then
  consequent-list-1;
[elif test-list-i; then
  consequent-list-i;* ]
else
  alt-list;
fi
```

Executes lists *test-list-1* to *test-list-n* until one succeeds or all fail. If *test-list-m* succeeds, executes *consequent-list-m*, then stops. If none of the test lists succeeds **and** there is an `else` clause, executes *alt-list*.

`test-list?` is often a `(())` or `[[]]` conditional but may be *any* lists, including a simple command.

The `else` and `elif` clauses are optional, there may be more than one `elif` clause.

9.2.2 `case - in`

```
case word in
pat1 )
list1
;; # or ;& or ;;&
pat2 )
list2
;;
esac
```

`word` is expanded (all but split+glob), then the first `list?` that matches the expansion is run.
`pat?` may be composed of sub-patterns separated by `|`.

operator	behaviour
`;;`	exit the `case` statement
`;&`	execute next clause
`;;&`	try next pattern, run list iff match

`((expr))` : succeeds iff `expr` $\neq 0$ (arithmetic evaluation)

`[[ expr ]]` : succeeds based on the value of the *conditional expression* `expr`. `s == p` does glob-style pattern matching on `p` (can be a simple string). Same for `= !=`. `-eq -ne -lt -gt -le -ge` evaluate their arguments with *shell arithmetic* (12).

9.3 Conditional Expressions

expr	true iff
`-e f`	`f` exists
`-f f`	`f` is a regular file
`-d d`	`d` is a directory

`-r f`	`f` is readable
`-w f`	`f` is writeable
`-x f`	`f` is executable
`f1 -nt f2`	`f1` is newer than `f2` (`-ot`: older)
`-v var`	`var` is set
`-z str`	`str` has length 0
`-n str`	`str` has nonzero length
`str`	`str` has nonzero length
`s == p`	`s` =/matches `p`
`s = p`	`s` =/matches `p`
`s != p`	`s` ≠/doesn't match `p`
`s1 < s2`	`s1` sorts before `s2` (`>`: after)
`a1 -eq a2`	a1 = a2
`a1 -ne a2`	a1 ≠ a2
`a1 -lt a2`	a1 < a2 (`-gt`: >)
`a1 -le a2`	a1 ≤ a2 (`-ge`: ≥)

The following operators (by order of precedence) may be used to combine expressions into new ones:

operator	value/effect
`( expr )`	*expr* ; overrides normal precedence
`! expr`	negates *expr*
expr1 `&&` *expr2*	true if both *expr2* and *expr2* are true
expr1 `||` *expr2*	true if at least one of *expr2* and *expr2* is true

`&&` and `||` evaluate *expr2* only if *expr1* isn't sufficient to determine the value of the whole expression.

10 Parameters

Parameters store values. This section is about setting and unsetting them. For retrieving values, see *parameter expansion* (5.3).

10.1 Scalars

10.1.1 Creation and Assignment

var=*value*

- There are **no spaces** around the = sign
- *value* undergoes all expansions except `W` and `Q`.

10.1.2 Destruction

11 Arrays

One-dimensional, indexed by non-negative integers (*indexed*) or strings (*associative*).

In the sections below, *word*, *word1* etc. undergo all expansions.

11.1 Creation and Assignment

11.1.1 Indexed

- *i*, *i1* etc. below are *shell arithmetic* expressions (often just integers) - may silently evaluate to 0, e.g. on undefined variables. If negative, start from the end (i.e. -1 references the last element).

operation	effect
`declare -a` *array*	(optional) declares *array* to be an indexed array
array`[`*i*`]=`*word*	sets the value of array *array* at index *i* to *word*; creates *array* iff needed.
array`=(`*word1 word2 word3 ...*`)`	creates array *array* with values *word1 word2*, etc. at indexes 0, 1, 2, etc.
array`=([`*i1*`]=`*word1* `[`*i2*`]=`*word2* `[`*i3*`]=`*word3 ...*`)`	as above, but at indexes *i1*, *i2*, *i3*.

11.1.2 Associative

- *key*, *key1* etc. below undergo all expansions.[1]

operation	effect
`declare -A` *array*	declares *array* to be an associative array (required, otherwise *array* is indexed).
array`[`*key*`]=`*word*	sets the value of array *array* for key *key* to *word*.
array`=([`*key1*`]=`*word1* `[`*key2*`]=`*word2 ...*`)`	assigns value *word1* to key *key1* of array *array*, etc.
array`=(`*key1 word1 key2 word2 ...*`)`	as above

11.2 Destruction

expression	value
`unset` *a*`[`*sub*`]`	removes the element at index or key *sub* from array *a*.
`unset` *a*	removes the array *a*.
`unset` *a*`[@]`	removes the array *a*.
`unset` *a*`[*]`	removes the array *a*.

12 Shell Arithmetic

Strings are evaluated as integers, using the following operators (by decreasing priority):

operator	meaning
`var++ var--`	post-in(de)crement
`++var --var`	pre-in(de)crement
`+ -`	unary +, −
`! ~`	negation (logical, bitwise)
`**`	exponentiation
`* / %`	×, ÷, remainder

[1] Not documented in the manual, deduced from experimentation.

`+ -`	+, −
`<< >>`	left (right) bitwise shift
`< <= > >=`	numeric comparison
`&`	bitwise AND
`^`	bitwise XOR
`|`	bitwise OR
`&&`	logical AND
`||`	logical OR
`? :`	inline if
`=`	assignment
`,`	sequence

Augmented assignment (`+=`, `|=`, `<<=`, etc.) has the same priority as `=`.

13 Pattern Matching ("Globbing")

pattern	matches
`*`	any string, even empty
`?`	any character
`[...]`	any character in the class ...
`[!...]`	any character NOT in the class ...
`[^...]`	any character NOT in the class ...

13.1 Character Classes

class	matches
`x`	the character `x`
`x-y`	one of `x`, `y`, or any character in between
c1c2	(where *c1* and *c2* are classes): any character in *c1* or *c2*
`:name:`	any character in the predefined character class `name`

13.1.1 Some Predefined Character Classes

name	matches
`alpha`	letters
`alnum`	alphanumeric characters
`digit`	digits
`space`	whitespace
`punct`	punctuation

13.2 Extended Glob

If option `extglob` is set, and `list` is a list of ≥ 1 patterns separated by `|`:

pattern	matches
`?(list)`	zero or one of the pattern(s) in `list`
`*(list)`	zero or more "
`+(list)`	one or more "
`@(list)`	one or more "
`!(list)`	anything *but* one "

14 Functions

Behave like scripts: `return` returns an exit status - otherwise the status is that of the last command in the function; parameters are passed with `$1`, `$2`, `$*`, `$@`, etc.; results through command substitution: `$(f arg1 arg2)...`

14.1 Definition

```
function f() compound-cmd [ redirections ]
f() compound-cmd [ redirections ]
function f compound-cmd [ redirections ]
```

`compound-cmd` is usually just `{ list; }`, but can actually be any compound command
`unset -f` to delete a function
`local var...` makes a variable local to the function (default: global)

---

15 References

- GNU Bash reference manual
- POSIX shell command language definitiion
- When does Bash do split+glob
- All about Bash redirections
- A Guide to Unix Shell Quoting

16 TODO

- `eval` (and maybe `expr`)
- `declare`
- namerefs
- idioms:
  - `read x y < <(...)` (use process substitution to set variables)
  - `$(< file)` is a faster equivalent of `$(cat file)`
- `f -o a1 a2` – you can pass switches and options to functions
- regexps `[[ $var =~ regexp ]]`
- options: one section on `iset` and `shopt`, then mention of affecting options in the other sections (e.g., mention `failglob` in the File Globbing section.
- reserved words (under Parsing)
- best practices
  - `set -u`
  - `set -e`
  - `unset CDPATH` to prevent `cd` from outputting target dir