latex input: mmd-article-header Title: Bash Syntax Notes Author: Ethan C. Petuchowski Base Header Level: 1 latex mode: memoir Keywords: Bash, Unix, Linux, Shell, Command Line, Terminal, Syntax CSS: http://fletcherpenney.net/css/document.css xhtml header: copyright: 2015 Ethan C. Petuchowski latex input: mmd-natbib-plain latex input: mmd-article-begin-doc latex footer: mmd-memoir-footer

## The Environment

Essentially the shell's *global state*, inherited by every child process of this shell.

To change the environment for a *single command*, prefix the command with your settings

```
CLASSPATH=/bin:/usr/bin java MyProgram
```

## Variables

Assign a value to a variable

```
my_var=24
```

Make an existing variable **read-only**

```
readonly my_var
```

Add variable from this script to Bash's *global environment*

```
export my_var
```

Print the environment

```
env
# or
export -p
```

*Remove* variable/function from the shell

```
unset my_var
unset -f my_fctn    # for functions
```

### Expansion operators

| Operator | Meaning |
|---:|---|
| `${#my_var}` | Return **number of characters** in the value of `my_var` |
| `${my_var:-default}` | Return **default** value if variable is undefined |
| `${my_var:=default}` | **Set** variable *and* return it if it is undefined |
| `${my_var:?"message"}` | If variable is *null* or *undefined*, **exit** and print message |
| `${my_var:+value}` | Return `value` if `my_var` *is* defined |
| `${variable#pattern}` | Delete *shortest* match from *beginning* (only) of var's value, and return the rest (note: the `variable` itself is left unaltered) |
| `${variable##pattern}` | Delete *longest* match from *beginning*, return the rest |
| `${variable%pattern}` | Delete *shortest* match from *end*, return the rest |
| `${variable%%pattern}` | Delete *longest* match from *end*, return the rest |

#### Examples

```
vble=/my/long/path_to.thing
echo ${vble#/*/} # => long/path_to.thing
echo ${vble##/*/} # => path_to.thing
```

## Script/Function Parameters

### Special variables

Try echoing these.

| Variable | Meaning |
|---:|---|
| `$!` | PID of the most recent background command |
| `$$` | PID of the (current) script file or bash terminal |
| `$?` | Most recent foreground pipeline **exit status** |
| `$#` | Number of arguments passed to shell script/function |
| `$*/$@` | All command-line arguments (no quoting applied) |
| `"$*"` | All command-line arguments as a *single* string |
| `"$@"` | All cmd-line args, each wrapped in quotes |

| | |
|---|---|
| **:** | Alias for `true`; deprecated |

**Functions for manipulating parameters**

Replace supplied positional parameters with your own set of parameters

```
set first and third arguments
```

Shift all arguments *left*, replacing $1 with $2 and so on

```
shift [#args to shift]
```

## Arithmetic

The shell evaluates the arithmetic expressions inside and places the result back into the text of the command.

This is done as you'd expect, and it has *everything* you're used to, like |, ||, <<, +=, ++, etc.

```
echo $((2 + 3))    # => 5
```

**Booleans** are `1 = true` and `0 = false`

```
echo $((2 && 3))   # => 1
```

**Exponentiation** is done with **, like in Python

```
echo $((2 ** 3))   # => 8
```

**Note:** `$((...))` is defined by POSIX and is therefore available in all normal shells. KSH and BASH also have `((...))` and `let ...` but those should probably not be used seeing as they make the code less portable.

## If

General form (based on Algol 68)

```
if cond
then
    # what to do
elif cond
    # something
else
    # otherwise
fi
```

`test expr` is a synonym for `[ expr ]` (spaces required)

Test if `$file` is a directory

```
if [ -d "$file" ]
```

String comparison

```
if [ "$file" = "myfilename" ]
```

Multiple boolean checks

```
if [ "$file" = "myfilename" ] || [ "$file" = "another/name" ]
```

## Case

- Check if a variable is one of many values.
- Patterns for catching the variable *can* contain wildcard characters.

Syntax

```
case $1 in
-f)
    # code
    ;;  # like "break"
-d | --directory)  # multiple options
    # code
    ;;
*)                 # catch-all (not required)
    ;;  # not required here
esac
```

## Looping

### For

```
for i in *.[ch]
do
    # something
done
```

Loop over command-line arguments

```
for i
do
    case $1 in
    -f)
        # etc.
        ;;
    # etc.
    esac
done
```

### While and Until

```
while condition
do
    stuff   # *break* and *continue* are allowed
done

until condition
do
    stuff
done
```

**WARNING:** as noted in the Google Bash styleguide, variables modified in a while loop do not propagate to the parent because **a while loop's commands run in a subshell**. The implicit subshell in a pipe to while can make it difficult to track down bugs. The workaround is to

```
last_line='NULL'
your_command | while read line; do
  last_line="${line}"
done

# This will output 'NULL'
echo "${last_line}"
```

Their first solution is to use for loop, but that is only possible if the input will *never* contain spaces or special characters (i.e. it is also not user input).

Their second solution uses "process substitution" with redirected output

```
total=0
last_file=
while read count filename; do
  total+="${count}"
  last_file="${filename}"
done < <(your_command | uniq -c)

# This will output the second field of the last line of output from
# the command.
echo "Total = ${total}"
echo "Last one = ${last_file}"
```

But what the heck is that `<()` construct? The most succinct explanation comes from this redirections cheat sheet, which has *many* useful explanations, and states, and was written by that guy who is substack's friend

- `cmd <(cmd1)` -- redirect stdout of `cmd1` to an anonymous fifo, then pass the fifo to `cmd` as an argument. Useful when `cmd` doesn't read from `STDIN` directly.
- `cmd < <(cmd1)` -- redirect stdout of `cmd1` to an anonymous fifo, then redirect the fifo to `STDIN` of `cmd`.
- `cmd <(cmd1) <(cmd2)` -- redirect `STDOUT` of `cmd1` and `cmd2` to two anonymous fifos, then pass both fifos as arguments to `cmd`. Best example:

  ```
  diff <(find /path1 | sort) <(find /path2 | sort).
  ```

- `cmd1 > >(cmd2)` -- run `cmd2` with its `STDIN` connected to an anonymous fifo, then redirect `STDOUT` of `cmd` to this anonymous pipe.

### POSIX-Style Command-Line Arguments

Use `getopts` to allow getting CLAs like

```
grep -vnf --long-one=24
```

Here's how you'd implement something like that

```
file=
verbose=
quiet=
long=

while getopts "$@" opt
do
    case $opt in
    f)
        file=$OPTARG
        ;;
    v)
        verbose=true
        quiet=
        ;;
    esac
done
```

## Functions

```
my_func() {
    my code
    return 2  # set exit-status to 2 (failing)
}
```

Note that if you modify a global variable in a function, this modification is actually modifying that variable for real.

## `$(c)` vs `backtick(c)` vs `eval c`

- `$(c)` and `backtick(c)` are (at least practically) the same, they **capture the output**.
- `eval c` **interprets the text** you give it as a bash command.

## Subshells and Code Blocks

**Subshell** commands are wrapped in parentheses and are run in a separate process. The main advantage is that state changes in the subshell (e.g. via `cd`) don't affect the parent.

```
tar -cf - . | (cd /newdir; tar -xpf -)
```

A **code block** is like a subshell, but runs in the shell's current process, and state changes *do* affect the shell's state. These don't seem all that useful.

## Jobs

```
$ vim

# you type
^z  # stop (pause) process

[1]+ Stopped    vim

$ jobs
[1]+ Stopped    vim

$ fg  # back to vim
^z

$ less somefile.txt
^z
[2]+ Stopped    less somefile.txt

$ jobs
[1]- Stopped    vim
[2]+ Stopped    less somefile.txt

$ fg    # back to less
^z

$ fg %1 (or) fg 1  # back to vim
^z

$ kill 2  # raw number means pid, but pid:2 is not a child
bash: kill: (2) - No such process

$ kill %2
[2]- Terminated: 15   less somefile.txt

$ fg    # vim is only job left
^z
```

If you have a job that's taking too long and you want to **move it to the background**, you can do `CTRL-Z` to `STOP` it, then do

```
$ bg %JOB_NO
```

and the shell will run it as a background jobs, as though you had run it with

```
$ command for background execution &
```