Shell Programming

Matt Ruffalo

April 16, 2013

CASE WESTERN RESERVE UNIVERSITY EST. 1826

Overview

What

- POSIX shells provide full-featured programming languages in addition to executing what you tell them to
- Can use shell scripting constructs to automate some things
- Aided by small utility programs/commands
- Some commands are shell “builtins” instead of separate executables on disk

Overview

Why

- Convenience
  - Each line of a shell script is directly executed by the shell; it's easy to run a program and capture its output
- Portability: shell scripting can be treated as a lowest common denominator
  - Perl is everywhere but the differences between versions can be important
  - It was unpleasant to get Python 3 installed on Case's high performance computing cluster
  - In some environments a shell may be all you have (*e.g.* Linux `initrd` images)

Overview

How

- Bash provides many constructs common in other languages (`if`, `while`, `for`, `case`, functions, . . .)
    - Bash's manpage is *huge*
    - Syntax in other shells (`zsh`, (`t`)`csh`) is usually not the same
- Variable assignment is straightforward (all strings though)
- Can make a file full of commands and run it with *e.g.* `bash script.sh`
- Pipes ("|") are your friend

Variables

Assignment, Usage

```
$ name=value
$ echo $name
value
$ path=/
$ ls $path
bin boot cdrom dev etc home lib media mnt opt ...
```

- "=" must directly follow the variable name in assignments
- Variables aren't visible in subprocesses unless you export `variable=value`

Math

Arithmetic Expansion

- Math goes inside `$(( ))`

```
$ echo $((4 * 2 ** 30))
4294967296
```

Process Execution & Output

Command Substitution

- Run a program and obtain what's written to its standard output with `$(command)`
- Or use backticks: `` `command` ``
- Backticks don't nest well (*i.e.* at all); I prefer `$()`

```
$ # Add current directory to PATH:
$ export PATH=$PATH:$(pwd)
```

Useful Programs

Unix systems ship with a *ton* of small useful programs

- `grep`: search for patterns in standard input or files, print matching lines to stdout
- `sed`: perform regex-defined substitutions on data from stdin; write to stdout
- `[` (a.k.a. `test`): perform various checks on files or strings
  - This is an actual executable that's often in `/usr/bin`, but it's also a shell builtin
- `head`, `tail`: print first/last *n* lines of input
- `sort`: sorts input

Useful Programs

Many commands are built in to bash

- `unset`: remove a variable assignment
- `trap`: intercept signals; run a command when the signal is received
- `read`: read from standard input, store result(s) in variable(s)

Return Codes, Boolean Evaluation

Central concept: *return codes*

- Programs return a value after execution
- Convention: 0 means success, nonzero means failure

Conditional execution (basics)

- Run a second command iff the first one succeeds: `command && echo "command succeeded"`
- Run a second command iff the first one fails: `command || echo "command failed"`

Conditional Execution

Semantics aren't surprising

```
if condition; then
    # do things
elif condition; then
    # do other things
else
    # different things
fi
```

- Check if a string is empty:

```
$ if [ -z $var ]; then echo "unset"; fi
unset
```

while loops

Repeated if

```
while condition; do
    # things
done
```

- Run something every 2 seconds:

```
while [ 1 ]; do
    command
    sleep 2
done
```

for loops

Not the same as C

```
for name in sequence; do
    echo $name # or do something more useful
done
```

- Convert all JPEGs to PNGs:

```
$ for f in *.jpg; do
> convert $f $(echo $f | sed 's/jpg/png/')
> done
```

Something that I do reasonably often

```
# mount /dev/sda1 /mnt/btrfs
# cd /mnt/btrfs/snapshots
# root_count=$(($(ls -d root* | wc -l) - 1))
# home_count=$(($(ls -d home* | wc -l) - 1))
# for f in $(ls -d root* | head -n $root_count;
> ls -d home* | head -n $home_count); do
> btrfs subvolume delete $f; done
# cd
# umount /mnt/btrfs
```

case

Check input against repeated patterns

```
case $value in
  0)
    do_if_0
    ;;
  1|2)
    do_if_1_or_2
    ;;
  *)
    do_otherwise
    ;;
esac
```

- There are options other than `;;` for ending each case; check bash manpage for details

Functions

No explicit arguments

```
func_name()
{
        do_thing $1
        do_other_thing $@
}
```

- Access function arguments through numbered variables `$1`, `$2`, . . . or special variables `$@` or `$*`
- Call with *e.g.* `func_name arg1 arg2`