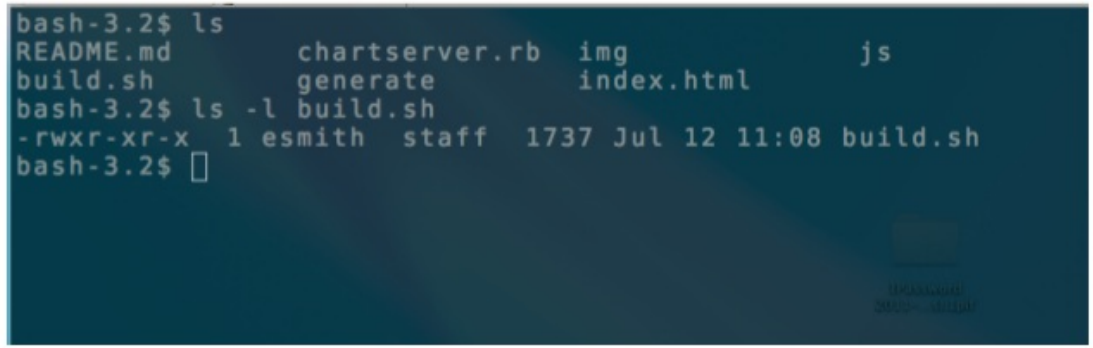
Shellcraft

building a better command line

what do you mean, shell?

servant of two masters

- CLI - command interpreter, exposes operating system
- script interpreter - programming interface, batches commands



Unix shell is famously difficult to master...

- steep learning curve
- arcane, terse commands
- not user friendly

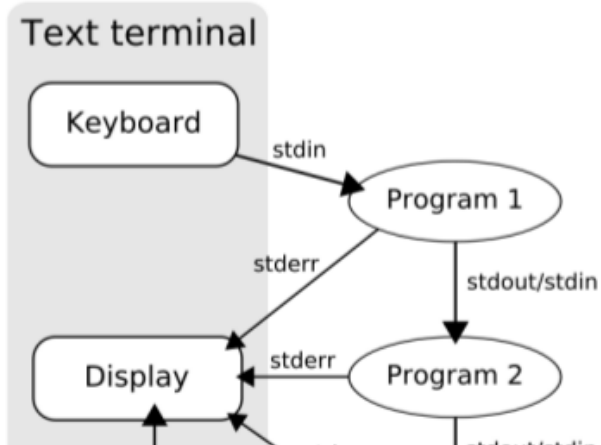
> *Unix never says 'please'.*
> *-- Rob Pike*

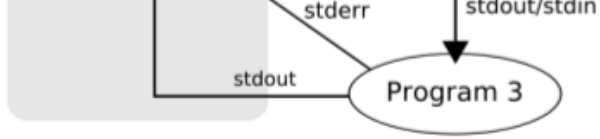
yet, it's often the most powerful tool.

The Unix philosophy:

> *Write programs that do one thing and do it well.*
> *Write programs to work together.*
> *Write programs to handle text streams, because that is a universal interface.*
>
> *-- Doug McIlroy, the inventor of Unix pipes (1972)*

pipes - Unix's killer app





example: what's my MAC addres

```
1  $ ifconfig eth0
2
3  eth0      Link encap:Ethernet  HWaddr f8:
4            inet addr:10.80.100.35  Bcast:1
5            inet6 addr: fe80::20b:dbff:fe9
6            UP BROADCAST RUNNING MULTICAST
7            RX packets:408752002 errors:0 c
```

```
          TX packets:388745488 errors:0 d
          collisions:0 txqueuelen:1000
          RX bytes:123203059 (123.2 MB)
          Interrupt:16
```

example: what's my MAC addres

```
$ ifconfig eth0 | grep HWaddr

eth0      Link encap:Ethernet  HWaddr f8:
```

'grep' shows only lines that contain some matching expressio

example: what's my MAC addres

```
1 $ ifconfig eth0 | grep HWaddr | awk '{pri
```

```
2
3 f8:1e:df:e6:a9:13
```

'awk' is grep on steroids.

example: what's my MAC addres

```
$ ifconfig eth0 | grep HWaddr | awk '{pri

F8:1E:DF:E6:A9:13
```

'tr' *translates* one set of characters to another.

demo: mac address

strengths of the Unix model

```
1 $ ifconfig eth0 | grep HWaddr | awk '{pri
```

strengths

- easy to assemble
- infinitely flexible
- easy to reason about

strengths are also weaknesses

```
1 $ ifconfig eth0 | grep HWaddr | awk '{pri
```

strength

- easy to assemble
- infinitely flexible
- easy to reason about

weakness

strengths are also weaknesses

```
1 $ ifconfig eth0 | grep HWaddr | awk '{pri
```

strength

- easy to assemble
- infinitely flexible
- easy to reason about

weakness

hard to read

strengths are also weaknesses

```
1 $ ifconfig eth0 | grep HWaddr | awk '{pri
```

strength

- easy to assemble
- infinitely flexible
- easy to reason about

weakness

hard to read

some assembly is alwa

strengths are also weaknesses

```
1 $ ifconfig eth0 | grep HWaddr | awk '{pri
```

strength

- easy to assemble
- infinitely flexible
- easy to reason about

weakness

hard to read

some assembly is alwa

transformations are opa

fixating on weakness becomes li

```
1 $ ifconfig eth0 | grep HWaddr | awk '{pri
```

weakness limiting belie

- hard to read
- some assembly is always required
- transformations are opaque

- transformations are opaque

fixating on weakness becomes li

```
1 $ ifconfig eth0 | grep HWaddr | awk '{pri
```

weakness	limiting belie
• hard to read	shell scrip

- some assembly is always required
- transformations are opaque

fixating on weakness becomes l

```
1 $ ifconfig eth0 | grep HWaddr | awk '{pri
```

weakness limiting belie

weakness

- hard to read
- some assembly is always required
- transformations are opaque

limiting belie

shell scrip

Unix shell

fixating on weakness becomes li

```
1 $ ifconfig eth0 | grep HWaddr | awk '{pri
```

```
$ ifconfig eth0 | grep inet addr | awk '{pri
```

weakness	limiting belie
• hard to read	shell scrip
• some assembly is always required	Unix shell
• transformations are opaque	good for t

changing POV

Bernhardt's talk inspired me to revisit my approach to using the shell.

Insight:

statements freeze perspectives, questions open them up.

jedi mind trick: inverting limits

invert a limiting belief, and turn it into an open-ended question

inverting limits

```
1 $ ifconfig eth0 | grep HWaddr | awk '{pri
```

limiting belief

empowering

- shell scripts are ugly
- Unix shell isn't REAL programming
- good for throw-away code

inverting limits

```
1 $ ifconfig eth0 | grep HWaddr | awk '{pri
```

limiting belief

- shell scripts are ugly
- Unix shell isn't REAL programming
- good for throw-away code

empowering

what wou

inverting limits

```
1 $ ifconfig eth0 | grep HWaddr | awk '{pri
```

limiting belief	empowering
• shell scripts are ugly	what wou
• Unix shell isn't REAL programming	what if it
• good for throw-away code	

inverting limits

```
1 $ ifconfig eth0 | grep HWaddr | awk '{pri
```

limiting belief	empowering
• shell scripts are ugly	what wou
• Unix shell isn't REAL programming	what if it
• good for throw-away code	what if sh

empowering question

what would beautiful shell look like?

- readable
- understandable
- documented

composed method

striving for **readable**, **understandable** code often results in many small, well-named methods.

-what Kent Beck refers to as the *Composed Method* pattern

apply composed method

before

```
1 $ ifconfig eth0 | grep HWaddr | awk '{pri
```

after

```
1 get_mac_address () {
2   ifconfig $1 | grep HWaddr | lastcol | u
3 }
4
5 lastcol () {
6   awk '{ print $NF }'
```

composed method

```
1 get_mac_address () {
2   ifconfig $1 | grep HWaddr | lastcol | u
3 }
```

```
4
5  lastcol () {
6    awk '{ print $NF }'
7  }
8
9  uppercase () {
10   tr '[:lower:]' '[:upper:]'
11 }
```

benefits

reality check

why aren't all shell commands composed in this style?

Why aren't all shell commands composed in this style?

tensions between *interactive* CLI and *batch* script interpreter:

- feedback- REPL prints intermediate results that scripts lack
- editing context- prompt (1D) vs text editor (2D)

two environments, two

adaptations

- interactive CLI is the native habitat for building up pipelines
- non-interactive script files are the native habitat for shell functions

can we have the best of both worlds?

empowering question, revised

what if it were easier to ^ interactively ~~apply modern coding practices~~ compose shell functions?

blur the lines between prompt and script

draft functions from command lir

```
$ ifconfig eth0 | grep HWaddr |  awk '{ p

00:0b:db:93:0a:08

$ draft get_mac_address

$ get_mac_address

00:0b:db:93:0a:08
```

revise functions from command

```
$ revise get_mac_address
```

opens definition in text editor

```
get_mac_address ()
{
  ifconfig $1 | grep HWaddr |  awk '{ pri
}
```

re-loads function into memory on save

```
$ get_mac_address eth0

00:0b:db:93:0a:08
```

demo: draft / revise

use draft() and revise() for **interactive-style function composition** from the prompt

empowering question

what if shell programs were always available for refactoring?

- never throw code away
- automatic, transparent versioning
- quickly re-open functions for revision

local Git repository

draft() and revise() auto-commit with local Git repo

- function persistence across shell sessions
- revisions available via normal git commands
- share functions with team via shared repo access

empowering question

what would beautiful shell look like?

- readable
- understandable
- **documented???**

towards better documentation

Comments are helpful. Shell scripts allow them, but in-memory functions don't.

```
$ foo () {
    # this is a comment
```

```
2       # this is a comment
3       echo 'foo'
4   }
5
6   $ typeset -f foo
7
8   foo () {
9       echo 'foo'
10  }
```

towards better documentation

A sneaky shell comment would:

- be a valid shell statement
- allow arbitrary text as arguments
- have zero side effects

null command, :

> *Since the values passed on the command line of the null command are ignored, the null command can be used as a comment.*

```
: This is a comment
```

> *However, there is no advantage to using this--it is more of a novelty than a practical technique*

a personal challenge

a personal challenge

```
REM () {
    :
}

foo () {
    REM 'a BASIC remark'
    echo 'foo'
}

$ typeset -f foo

foo () {
    REM 'a BASIC remark'
    echo 'foo'
}
```

more than a comment

```
1 REM () {
2       :
3 }
```

features

- function names are arbitrary
- create as many as you need
- simplest possible shell function
- retrievable at run-time

run-time introspection

```
$ typeset -f foo

foo () {
    REM 'a BASIC remark'
    echo 'foo'
}

$ typeset -f foo | grep REM | lastcol

'a BASIC remark'
```

key idea: code is also data

functional metadata

key/value metadata for shell functions!

establish keyword convention

```
1 # cite() magically builds metadata functi
2
3 cite about author example group param ver
```

get_mac_address, reprised

```
1 get_mac_address () {
2     about 'retrieves mac address for a gi
3     param '1: network interface'
4     example '$ get_mac_address eth0'
5     group 'network'
6
7     ifconfig $1 | grep HWaddr | lastcol |
8   }
```

get_mac_address, reprised

```
lastcol () {
 about 'prints the last column of space-d
 example '$ echo "1 2 3" | lastcol   # pr
 group 'filters'

 awk '{ print $NF }'
}
```

get_mac_address, reprised

```
1 uppercase () {
2   about 'converts lowercase characters to
3   example '$ echo "abc" | uppercase   # p
4   group 'filters'
5
6   tr '[:lower:]' '[:upper:]'
```

```
}
```

putting metadata to work

```
$ reference get_mac_address

get_mac_address      retrieves mac address
```

```
parameters:
                        1: network interface
examples:
                        $ get_mac_address eth
```

use case: api reference

putting metadata to work

```
$ glossary filters
```

```
$ glossary filters

lastcol                 prints the last colum
uppercase               converts lowercase ch
```

use case: list commands by group

putting metadata to work

```
$ all_groups

filters
network
```

use case: list available groups

persist shell functions with meta

```
$ write get_mac_address lastcol uppercase
$ echo "get_mac_address $1" >> getmac.sh
$ sh ./getmac.sh eth0

F8:1E:DF:E6:A9:13
```

demo: composure

simple network monitor

links

blogs

Gary Bernhardt

Kent Beck

Martin Fowler

reference

The Unix Chainsaw (31 minutes)

Blinn, Bruce. *Portable Shell Programming*. New Jersey: Prentice Hall, 1996.