Institute for Networking and Security Research
Department of Computer Science and Engineering
Pennsylvania State University, University Park, PA

Shell Programming (Part 2)

Devin J. Pohly <djpohly@cse.psu.edu>

Some references

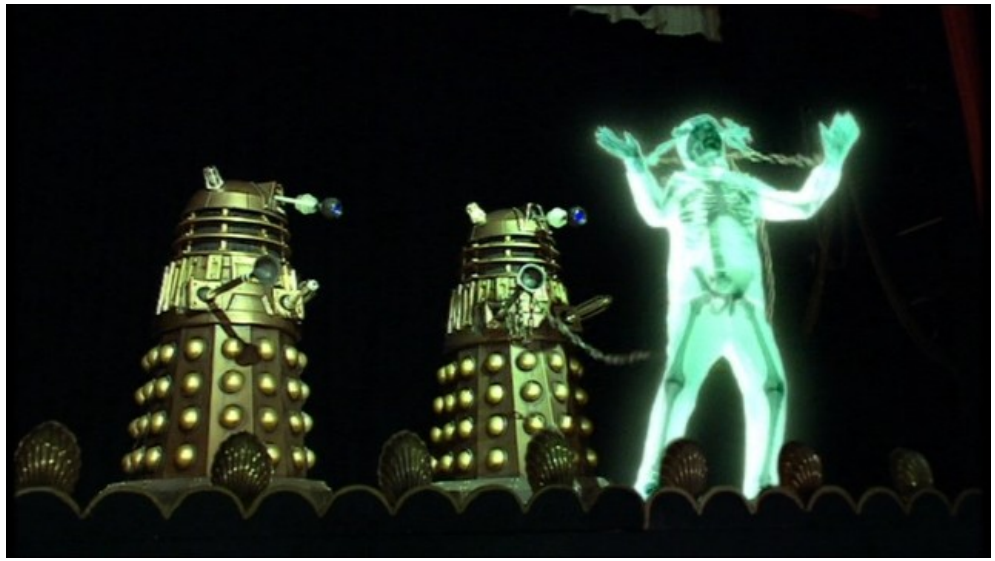
- Advanced Bash-Scripting Guide
  - http://tldp.org/LDP/abs/html/
  - Actually a great reference from beginner to advanced
- commandlinefu.com
  - Lots of gems, somewhat more advanced
  - Fun to figure out how they work
- Bash man page
  - `man bash`
  - Very complete, once you're used to reading man pages

Code for today

```
$ wget -U mozilla tiny.cc/311shell2
$ tar -xvzf 311shell2
$ cd shell2
$ make
```

How to kill a process

- Today we're learning some loops
- If it starts to run away, Ctrl-C is your friend
  - Sends a signal that ends the process
    - More on signals later...
  - Works on many different programs, as long as they were started from the command line
  - Displayed as ^C



Return from `main`

- In C, the main function always returns an `int`
  - Used as an error code for the entire process
  - Same convention as any other function
    - Zero: success
    - Nonzero: failure, error, killed by a signal, etc.
- Also known as the *exit status* of the process

Exit status in scripts

- `$?`: get exit status of the previous command
- The exit status of a script comes from the last command it runs
  - Or use the `exit` builtin to exit early, e.g. `exit 1`
- `! cmd` reverses the value: 0 for failure and 1 for success
  - Works just like the ! ("not") operator in C

Status sample program

```
$ ./status 0
$ echo $?

$ ./status 2
$ echo $?

$ ! ./status 2
$ echo $?

$ ./status -1
$ echo $?
```

```c
#include <stdlib.h>

int main(int argc, char **argv)
{
    // Quick-and-dirty int conversion
    return atoi(argv[1]);
}
```

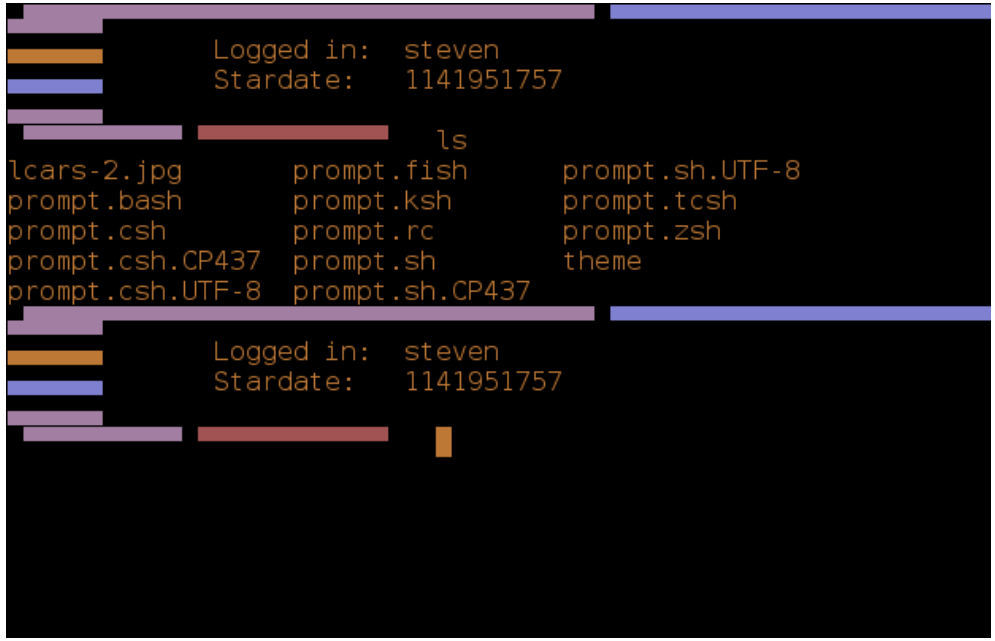
Custom prompt for today

- You can include `$?` in your prompt
  - I personally like this – it lets me know for sure when something fails
- For today, let's do this:

```
source newprompt
```

- Now try:

```
./status 42
```



Test commands

- Builtin commands that test handy conditions
- `true`: always succeeds
- `false`: always fails
- Many other conditions: `test` builtin
  - Returns 0 if test is true, 1 otherwise
  - Full list: `help test`

What do these do?

```
$ test -e status.c
$ test -e asdf

$ test -d status.c
$ test -d /etc

$ test 10 -gt 5
$ test 10 -lt 10
$ test 10 -le 10
$ test 12 -ge 15
```

Useful tests

- `test -e file`
  - True if file exists
- `test -d dir`
  - True if dir exists and is a directory
- `test -z "$var"`
  - True if var is empty (zero-length)
- `test -n "$var"`
  - True if var is nonempty
- `test str1 = str2`
- `test num1 -gt num2`
  - or `-lt`, `-ge`, `-le`, `-eq`, `-ne`

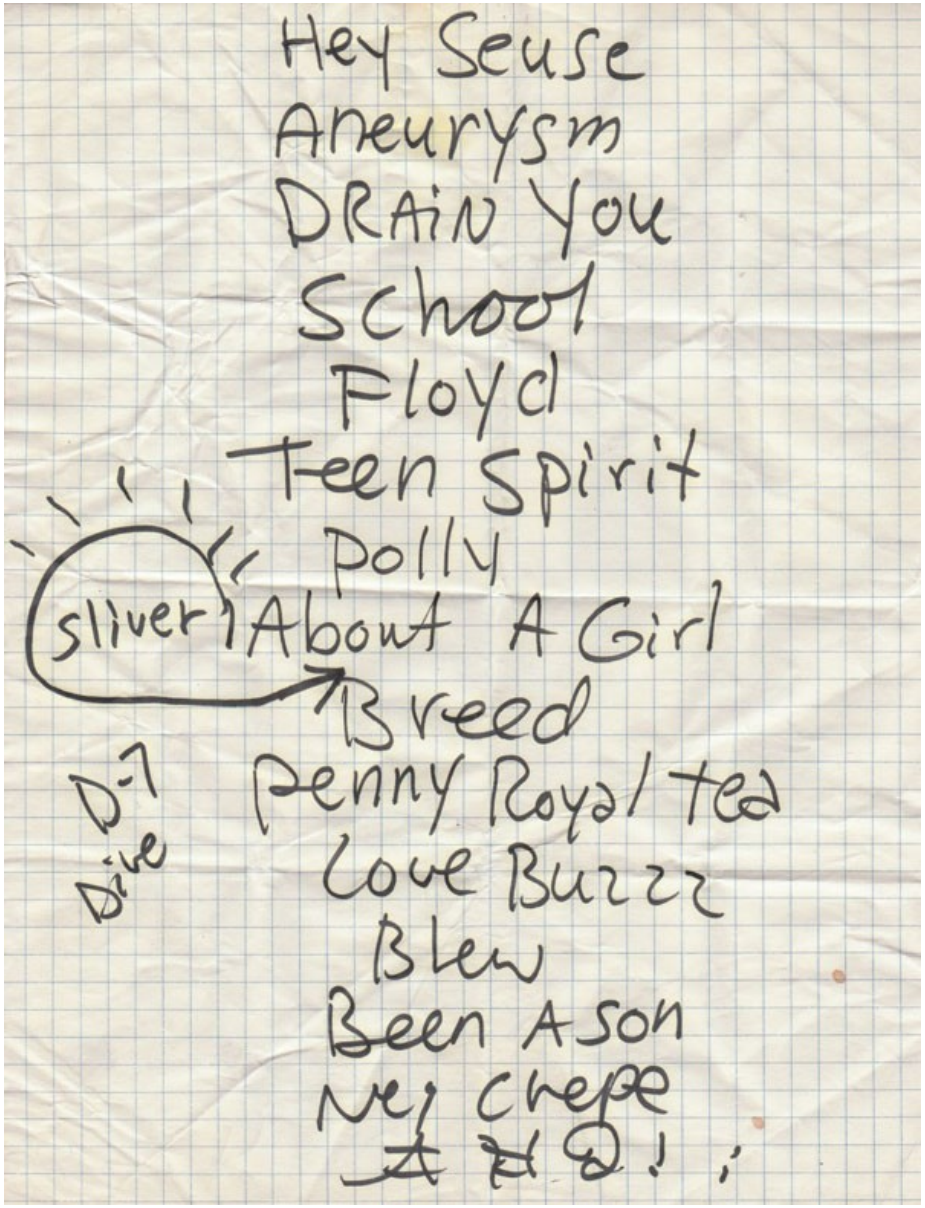
Command lists

- Simple command list: `;`
  - Runs each command regardless of exit status
  - Example:

```
do_this; do_that
```

- Shortcutting command lists
  - `&&` stops after failure
  - `||` stops after success
  - Examples:

```
foo && echo success
bar || echo failed
```



Try it out

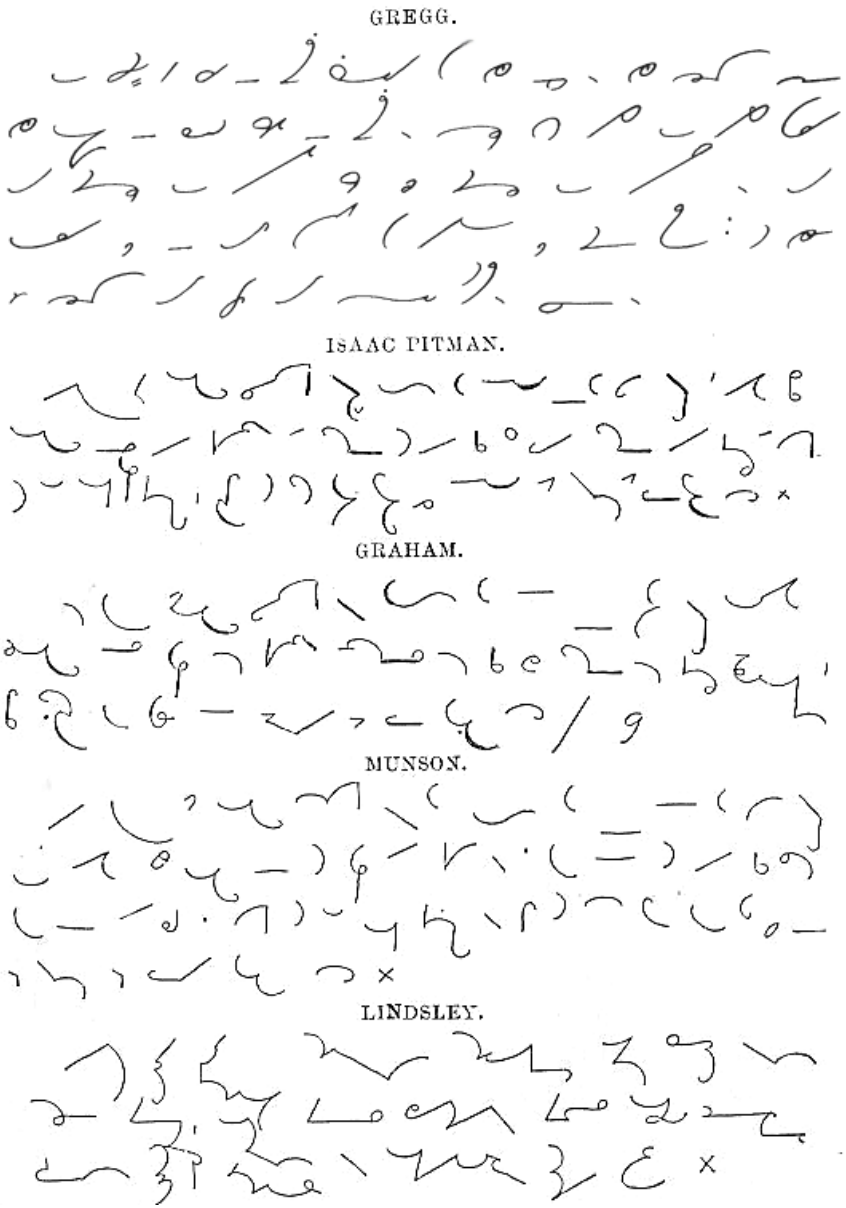
```
true && echo one
true || echo two
false && echo three
false || echo four
test -e Makefile && make
cat dog || echo bird
./status 4 && echo 4
./status 0 && echo 0
cat dog; cat status.c
touch status.c; make
make clean && make
```

Shorthand tests

- Shorthand test: `[[ ... ]]`
  - Workalike for `test`
- For example:

```
age=20
test $age -ge 16 &&
echo can drive
[[ $age -ge 16 ]] &&
echo can drive
```

- Now say age=3 and try again



Conditionals

- Exit status is used as the test for `if` statements:

```
if list; then
   cmds
fi
```

- Runs *list*, and if the *exit status* is 0, then *cmds* is executed
- There are also `elif` and `else` commands that work the same way.

Conditional loops

- You can write a `while` loop using the same idea:

```
while list; do
   cmds
done
```

- Runs *list*, *cmds*, *list*, *cmds*, *list*... for as long as *list* succeeds (exit status 0)
- Similarly, the `until` loop will execute as long as *list* fails

Conditional practice

```
if ! [[ -e foo ]]; then
  echo hello > foo
fi

while [[ "$x" -lt 99999 ]]; do
  echo "$x"
  x="1$x"
done

if cat foo; then
  echo Same to you
fi

if cat dog; then
  echo Woof
fi
```

For statement

- The `for` loop is "for-each" style:

```
for var in words; do
   cmds
done
```

- The *cmds* are executed once for each argument in *words*, with *var* set to that argument

For example... (get it??)

```
for a in A B C hello 4; do
  echo "$a$a$a"
done

for ext in h c; do
  cat "hello.$ext"
done
```

For example... (get it??)

```
for a in A B C hello 4; do
  echo "$a$a$a"
done

for ext in h c; do
  cat "hello.$ext"
done
```

Globbing

- Funny name for *wildcards*
  - (Comes from "glob<u>al</u> command")
- * means any number of characters:

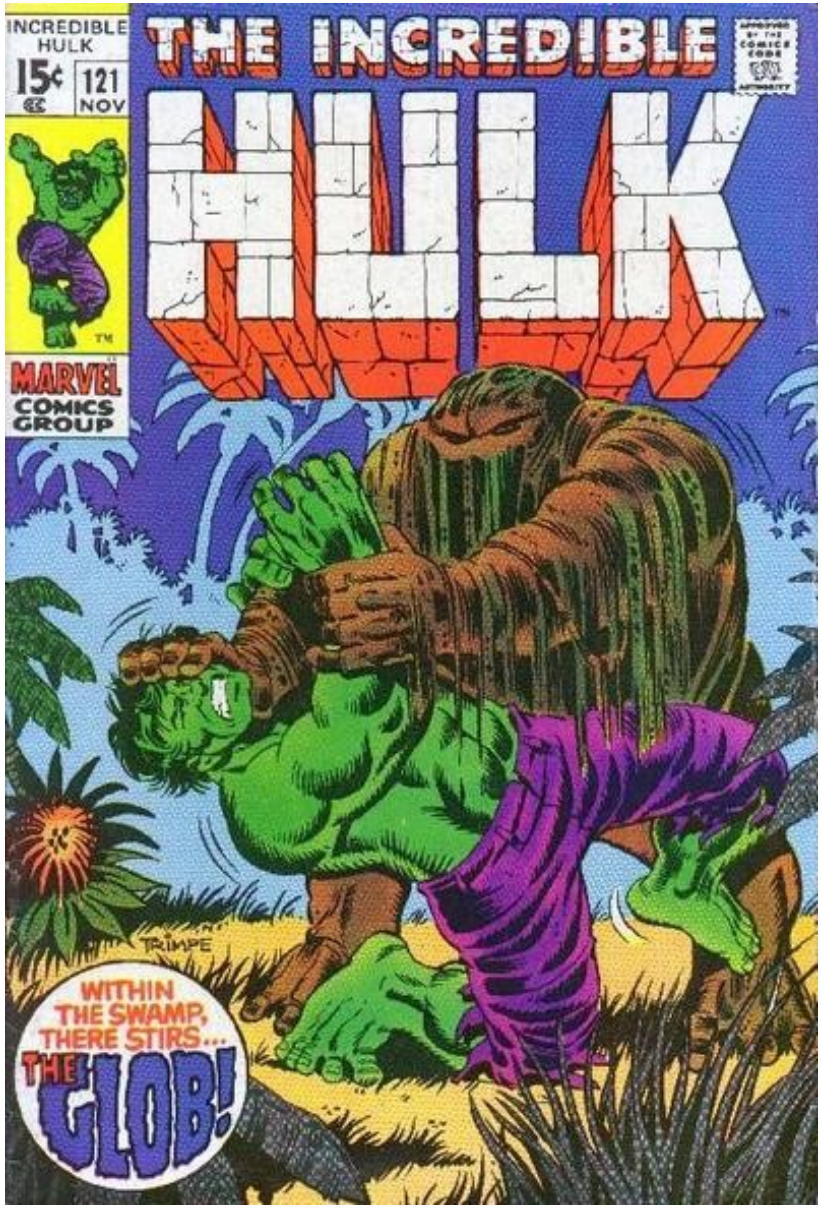
```
$ echo *
$ echo *.c
```

- ? means any one character:

```
$ echo hello.?
```

- Bulk rename:

```
for f in hello.*; do
  mv "$f" "$f.bak"
done
```



Some more useful tools

- `touch` *`foo`*: "modify" the file *foo* without really changing it
- `sleep` *`t`*: wait for *t* seconds
- `fgrep` *`string`*: filter stdin to just lines containing *string*
- `find . -name '*.c'`: list all the .c files under the current directory
  - Many other things you can search for; see `man find`
- `file` *`foo`*: determine what kind of file *foo* is
- `wc`: counts words/characters/lines from stdin
- `bc`: command line calculator

Exercises

- Print out “foo” once per second until ^C’d
- Find all the .png files in dir/
- Find all the files which are actually PNG graphics in dir/
- Use a pipe and bc to calculate the product of 199 and 42