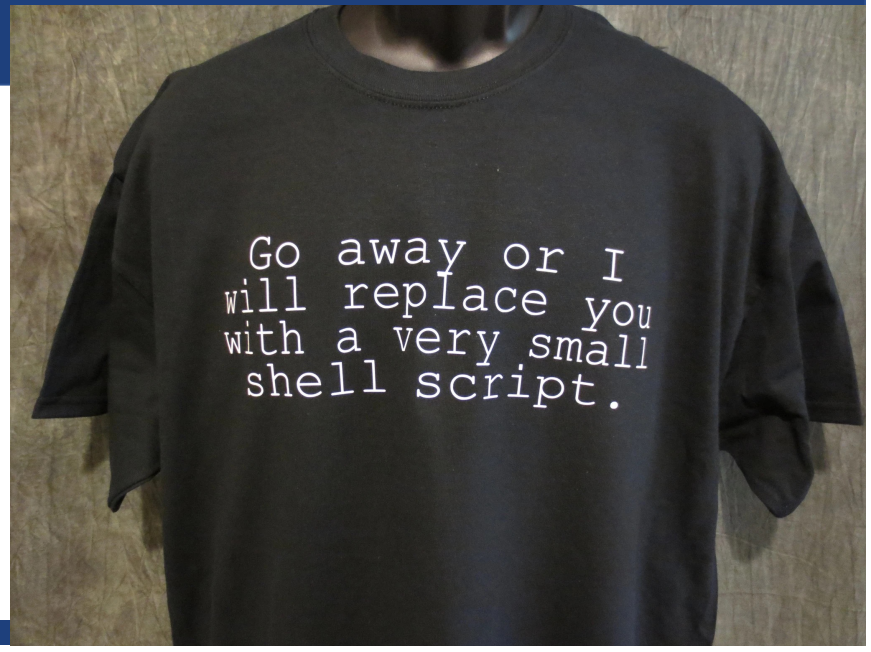
PennState

CMPSC 311 - Introduction to Systems Programming

Shell Programming

Professor
Suman Saha
(Slides are mostly by *Professor Patrick McDaniel* and *Professor Abutalib Aghayev*)



Shell programming

PennState

- aka "shell scripting," "Bash scripting"
- What is it?
  - Series of commands
  - Programming with programs
- What for
  - Automating
  - System administration
  - Prototyping

A sample script: `shello`

PennState

- First line: interpreter
  - The `#!` is important!
- Comment: `#` to end-of-line
- Give the file execute permission

  ```
  chmod +x shello
  ```
  - Not determined by file extension
  - Typical: `.sh` or none
- Run it

  ```
  ./shello
  ```

specify the interpreter

```
#! /bin/bash

# Greetings!
echo Shello world

# Use a variable
echo Shello "$USER"

# Set a variable
greetz=Shellutations
echo "$greetz world"
```

cannot have space

use the content of the variable

Shell variables

PennState

- Setting/unsetting
  - `export var=value` — set variable
  - No spaces!
  - `unset var` → unset variable
- Using the value
  - `$var` — e.g. `echo "$var"`
- Untyped by default
  - Behave like strings
  - (See `help declare` for more)

Special variables

PennState

- Change shell behavior or give you information
  - `PWD`: current directory
  - `USER`: name of the current user
  - `HOME`: the current user's home directory
    - Can usually be abbreviated as a tilde (~)
  - `PATH`: where to search for executables
  - `PS1`: Bash prompt (will see later)

e.g `PS1="C:/>"`
example to change PS1

Exercise

PennState

- Make a directory `~/bin`
- Move `shello` script there
- Prepend the directory to your `$PATH`
  - `PATH=~/bin:$PATH`
- Change to home dir
- Run by typing `shello`

~/bin

~/bin :$PATH

~/bin: $PATH

~/bin: $PATH

Shell initialization

PennState

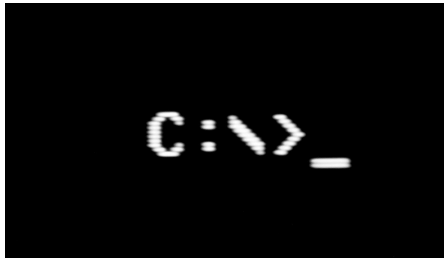
- Set custom variables
- At startup, bash runs shell commands from `~/.bashrc`
  - Just a shell script
- This script can do whatever you want

~/.bashrc

Fun with prompts

PennState

- Main prompt: `$PS1`
- Special values
  - `\u`: username
  - `\h`: hostname
  - `\w`: working dir
  - `\e`: escape (for colors)
  - many others
- This is something you can set in your `.bashrc`



Very detailed treatment: http://www.tldp.org/HOWTO/Bash-Prompt-HOWTO/

Special characters

PennState

```
echo Penn State is #1
echo Micro$oft Windows
echo Steins;Gate
```

comments

for variable

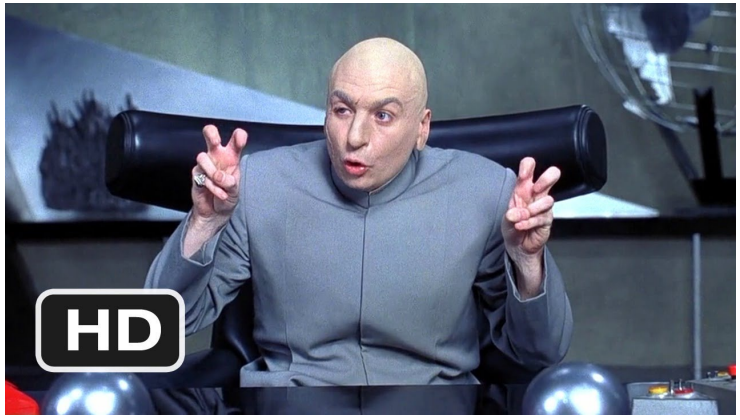
for seperate

- What happened?
- Many special characters
  - Whitespace
  - #$*&^?!~'`"\{}[]<>()|;
- What if we want these characters?

Quoting

PennState

- Removes specialness
- Hard quotes: `'...'`
  - Quote everything except closing `'`
- Soft quotes: `"..."`
  - Allow variables (and some other things)
  - Good practice: `"$var"`
- Backslash (escaping)
  - Quotes next character



Arguments

PennState

- In C: `argc` and `argv[]`
- Split at whitespace
  - How can we override this?
- Arguments to a script
  - `./script foo bar`
  - `$#` is the same as `argc`
  - `"$@"`: all args
  - `"$1"` to `"$9"`: individual

```
#! /bin/bash
echo Number of args: $#
echo Arg All: $@
echo Arg 1: $1
echo Arg 2: $2
```

Debug mode

PennState

- Shows each command
  - Variables expanded
  - Arguments quoted
- Run with `bash -x`
  - Temporary – just for that run
  - `bash -x shello`
  - Use `-xv` for even more info

Exercises

PennState

- Make a script that:
  - Prints its first argument doubled

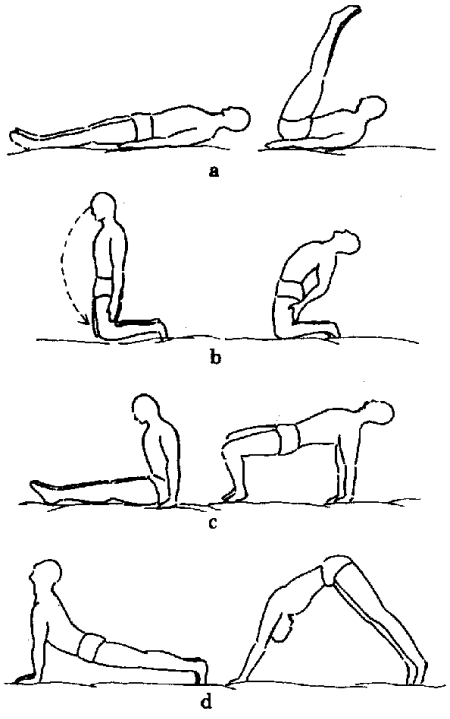
```
./script1 foo
foofoo
```

```
#! /bin/bash
echo $1 $1
```

  - Prints its first four args in brackets, one per line

```
./script2 "foo bar" baz
[foo bar]
[baz]
[]
[]
```

```
#! /bin/bash
echo [$1]
echo [$2]
echo [$3]
echo [$4]
```



Redirecting input/output

PennState

- Assigns stdin and/or stdout to a file
- `echo hello > world`
- `echo hello >> world`
- `tr h j < world`

translate "h" by "j" taken all elements from "world"

Pipelines

PennState

- Connect stdout of one command to stdin of the next

```
echo hello | tr h j
... | rev
... | hexdump -C
... | grep 06
```

- UNIX philosophy at work!

echo hello
→ hello
echo hello | tr h j
→ jello
reverse
echo hello | tr h j | rev
→ ollej

Some references

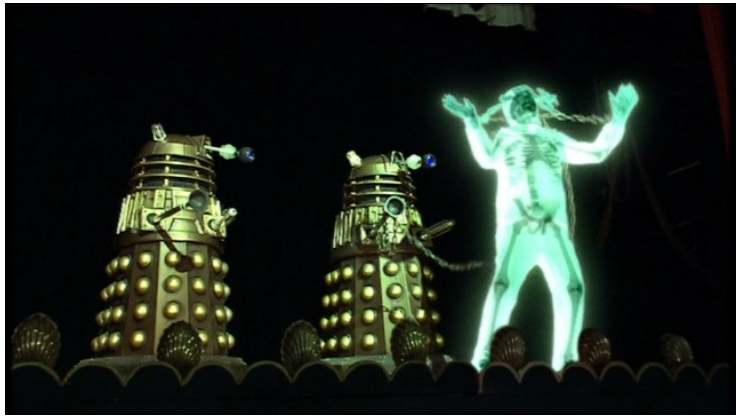
PennState

- Advanced Bash-Scripting Guide
  - http://tldp.org/LDP/abs/html/
  - 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`

How to kill a process

PennState

- 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`

PennState

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

Status sample program

PennState

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

$ ./status 2
$ echo $?

$ ! ./status 2
$ echo $?

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

Whatever is return from status.c will be stored in "?" question mark

```c
#include <stdlib.h>

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

Custom prompt for today

PennState

- You can include `$?` in your prompt
- Now try:

```
./status 42
```

Exit status in scripts

PennState

- `$?`: 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
  - Exactly like the ! ("logical not") operator in C

Conditionals

PennState

- 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.

e.g.
```
if grep include status.c > /dev/null ; then
    echo pattern include appears in file status.c
else
    echo pattern include does not appears in file status.c
fi
```

Test commands

PennState

- 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`

e.g. test -e status.c (exist or not)
echo $?
→ 0 (true)
1 (false)

What do these do?

PennState

```
$ 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
```

directory or not

Useful tests

PennState

- `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`

Shorthand tests

PennState

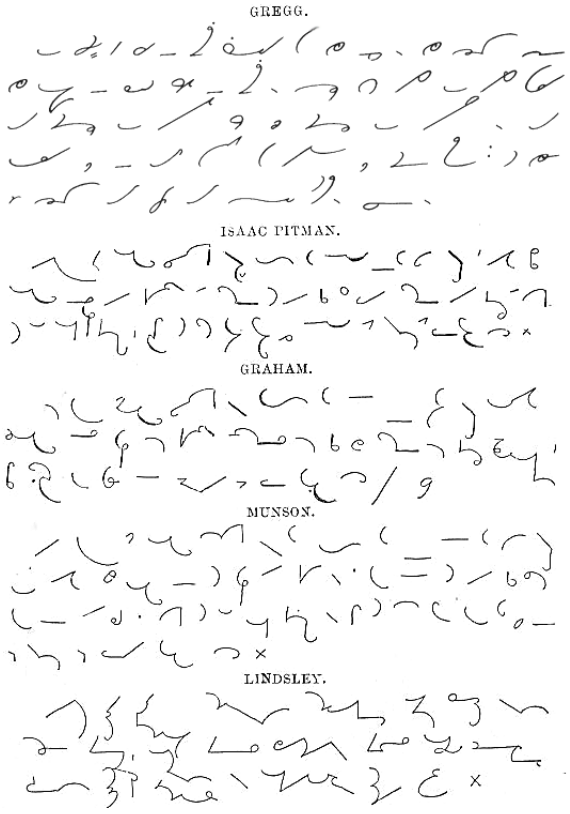
- Shorthand test: `[[ … ]]`
  - Workalike for `test`
- For example:

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

⇒ can drive

⇒ can drive

- Now say age=3 and try again



Command lists

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

```
do_this; do_that
```

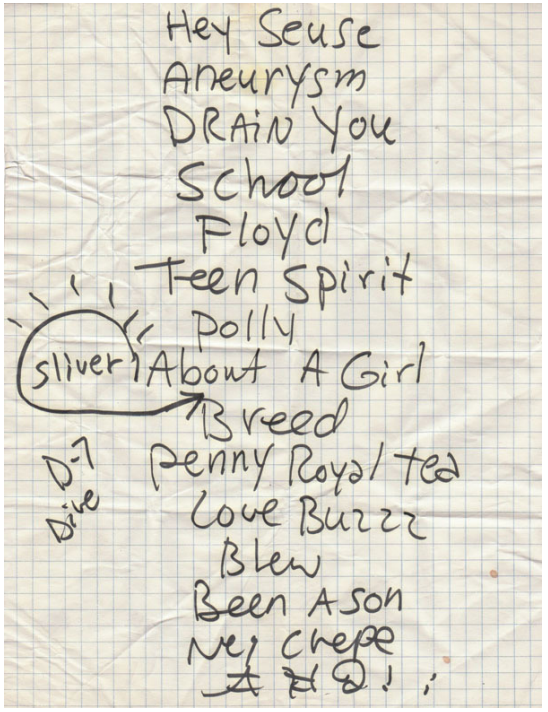
- Shortcutting command lists
  - `&&` stops after failure
  - `||` stops after success
  - Examples:

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

&&

||

;



Try it out

PennState

```
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
```

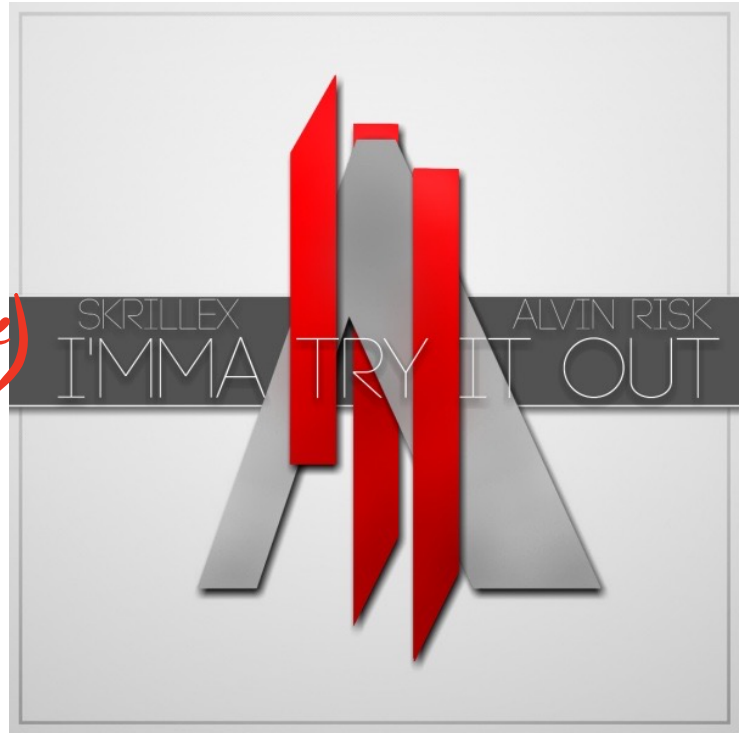
one

empty (no return)

(empty)

four

(0 for true
1 for false)



Conditional loops

PennState

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

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

(which loop check condition and then run the loop body)

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

like do while loop in C
(do while first run the loop body and then check the condition)

Conditional practice

PennState

```
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
```

→ prepend x by 1 : e.g 1 → 11

Command substitution

PennState

- Command substitution allows the output of a command to replace the command itself
  - In recent versions of bash: $(command)
  - More commonly: `command`

- file $(which ls)
- file `which ls`

For statement

PennState

- 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 i in $(seq 1 5); do
  echo $i;
  done
- for i in {1..5}; do echo $i; done

{1..5} ⇒ 1 2 3 4 5

```
if list ; then
   cmd
elif    list ; then
   cmd
else
   cmd
fi
```

```
while list; do
   cmd
done
```

```
for var in word ; do
   cmd
done
```

For example...

PennState

```
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

PennState

- Old name for *filename wildcards*
  - (Comes from "global command")
- * means any number of characters:

```
echo *
```

→ all files

```
echo *.c
```

→ all files with .c

- ? means any one character:

```
echo hello.?
```

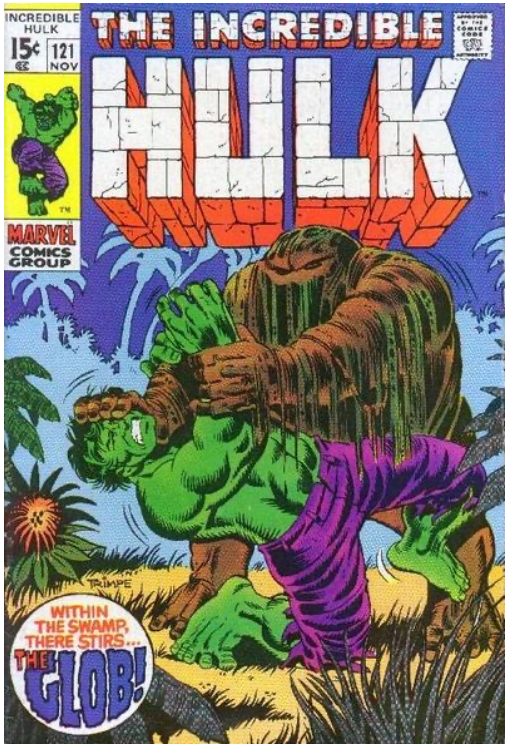
- Bulk rename:

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

remove $f

build $f.bak

e.g remove hello.c → build hello.c.bak



Some more useful tools

PennState

- `touch` *`foo`*: "modify" the file *foo* without really changing it
- `sleep` *`t`*: wait for *t* seconds
- `grep -F` *`string`*: filter stdin to just lines containing *string*
- `find . -name '*.c'`: list all the .c files under the current directory

  find . -name '*.c'

  - 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

Exercise time

PennState

- Print out "foo" once per second until ^C'd
- Find all the .png files in dir/ `find . -name '*.png'`
- Find all the files in dir/ which are *actually* PNG graphics
- Use a pipe and bc to calculate the product of 199 and 42 `echo (199*42) | bc`

哈 哦

哈 哦

哈 哦

swing

flog