Lecture 3 - The Shell

January 27, 2015

Compass

- talked about commands and pipelines
- first drill done
- assignment 1 due in 9 days
- today: shell and scripting
- throughout: more commands
- vm updated

Command Tips for HW1

Think in bulk.

It's natural, particularly based on early training, to think in pieces: iterate over this data, extract that, compute the other thing.

Many high-level systems do not reward that level of thought.

Instead, think about bulk set/collection operations:

- select these lines
- merge these sets
- sort that output
- think about *what*, not *how*

Other systems reward this kind of thinking:

- matlab/R/numpy
- databases

Commands and Parsing

We've seen **commands**, and built them into **pipelines**.

What does the shell do?

1. Split (*tokenize*) the input

2. Evaluate commands and pipelines
    - pass arguments to commands
    - route input and output

Most commands are programs (look at `/usr/bin`); some are *builtins*.

Arguments and Splitting

Let's write a small C++ program:

“‘c++ #include

int main(int argc, char *argv[]) { for (int i = 0; i < argc; i++) { std::cout <<“argv[” <<i <<“]:” <<argv[i] <<std::endl; } return 0; } “‘

And compile it, and run it:

```
c++ -o printargs printargs.cpp
./printargs
```

Our program gets run with one argument — its name. Try some more:

```
./printargs hackem muche
```

So here is what the shell does:

- split line on whitespace
- treat 1st word as command, run it

What if we want to have a space? (create demo file)

2 solutions:

- escape the space: `cat foobie\ bletch.txt`
- quote the argument: `cat 'foobie bletch.txt'`

Multiple Commands

The shell splits the line into words, and runs it. Basically.

Multiple commands can appear on a line:

- `A | B`: pipe A's output into B
- `A; B`: run A and then B
- `A && B`: run A, and then if it 'succeeds', run B
- `A || B`: run A, and then if it 'fails', run B

What is success?

Our program returns an integer from `main`. This is called the *exit code*. 0 is success, anything else is failure. Can also exit program with `exit()`, passing an exit code.

So: `A && B` runs B if A exits with code 0.

A couple little programs: `true` and `false` exit with code 0 and nonzero, respectively.

```
true && echo hi
false && echo hi
true || echo hi
false || echo hi
```

Scripts

- So far we've been typing in the terminal.
- We can write a *shell script* containing what we would type.
- Each command / pipeline is run.
- Keeps going after failure (by default)

```
echo hello
./printargs test the program
echo goodbye
```

How do we run this script?

- `sh script.sh`

This runs it using `sh`, the standard script interpreter. Often you'll see `/bin/sh`, that's the default location.

Running it with `sh` is annoying; it isn't treated like a normal command.

On Unix, executables can specify an interpreter to use in the first line of the file:

```
#!/bin/sh
```

Now we can make the file *executable* (`chmod +x script.sh`), and run it (`./script.sh`).

This tells Unix: when running this file, use the specified interpreter instead of trying to execute it directly.

The interpreter can be anything that takes the script as its first argument:

- `bash`
- `python`
- `perl`
- `tcl`
- `lua`
- `Rscript`
- `csh` (if you like pain)

It must be a full path - `/bin/sh` is always available; if we don't know the full path, we can say `#!/usr/bin/env python`.

Variables

We can set variables in shell:

```
foo=bar
echo "$foo"
```

- `echo` prints to standard out
- `$var` includes value of the variable
- variable substitution works inside double quotes (*not* single quotes)
- general rule: always quote variable expansions
  - there are some exceptions, but default should always be quoted.

There are some special variables:

- `$PATH` is where to look for programs
- `$?` is the exit status of the last program
- `$@` is the command line arguments (to a shell script)
- `$1` is the first command line argument

Environment Variables

Some variables are inherited by other programs: the *environment variables*.

- For use, there is no difference between a shell variable and an environment variable - `$` substitutes both of them.
- Make a shell variable an environment variable with `export`

Can also set them on the command line:

```
PKGMF=base.mf ./pc-export-packages base
```

Control Structures

Strait-line programs can be really powerful, especially in shell. But we want more!

for *arg* in *words*; do
commands
done

Each time through *commands*, the variable *arg* is bound to one of the words in *words*.

```
for arg in foo bar bletch; do
    echo $arg
done
```

Let's rewrite printargs in shell:

```
#!/bin/sh

for arg in "$@"; do
    echo "arg: $arg"
done
```

We have a special case here: `"$@"` does not expand to one word, it expands to all the command line arguments for the function, one per word.

Handling Arguments

- The variable `$@` has all the arguments
- The variables `$1`, `$2`, etc. have individual ones (`$0` has the script name)
- The `shift` built-in moves variables up

So we can rewrite our function body:

```
while [ -n "$1" ]; do
    echo "argument: $1"
    shift
done
```

What's up with `[`? It is how we write boolean expression tests.

- `[` introduces a test, `]` concludes it
- `-n` checks whether its next argument is non-empty (`-z` tests for empty)

More Control Strutures

We've seen two

- `for`
- `while`

also:

- `if`
- `case`

`if` and `while` are alike. They take a command (often `[`), and consider *success* to be true.

```
if [ condition ]; then
    commands
elif [ condition ]; then
    commands
else
    commands
fi
```

These control structures use some kind of *delimiter* to enclose blocks:

- for `while` and `for`, it's `do` and `done`
- for `if`, it's `then` and `fi`

`case`

The `case` structure lets us match a value:

```
case foo in
    foo) echo "found";;
    bar) echo "what?";;
esac
```

Each case terminates with `;;` (this is required)

Substitution and Expansion

So far, we've seen one kind of substitution: variable substitution.

`$VAR` expands to a variable. `"$VAR"` contains it within a single word (with a special case for `$@`).

There are other substitutions and expansions (all in the shell):

- glob expansion: `*.sh` → all shell scripts (as separate words)
- `$(cmd)`, expanding to the output of a command (gets split into words)
  - historical syntax: “`.
  - again, enclose in quotes: `"$(code)"`

Quoting

- single quotes quote everything (backslashes and all)
- double quotes suppress word splitting, but allow interpolation (and sometimes escapes)
- can also escape: `\$FOO`

Always quote your variables, unless you know you don't want to.

Where Commands Come From

- executables (compiled programs, scripts)
- shell builtins / control structures
- shell *functions*
- shell *aliases*

Shell functions:

```
hello()
{
    echo "Hello, world"
}
```

- behave like scripts, except they're in the same shell.
- have their own `$@`, `$1`, etc.
- do *not* have their own `$0`

Why `$0`?

Why do programs receive their name as the first argument?

- to locate the program (it has enough path to find it)
- some programs change behavior based on name (e.g. `egrep` is `grep -E`)

Shell History

- `/bin/sh` - Bourne (earlier Thompson)
- csh, because Bourne was bad for interactive use
- tcsh grew from csh
- ksh improved Bourne for interactive use
- also ash
- bash is the GNU shell, Bourne-based, borrowed lots of csh features
- POSIX standardized mostly Bourne
  - `/bin/sh` is a Bourne shell
  - It is usually POSIX, except on Solaris
- We will be targeting POSIX (Solaris has one)
- zsh also merges csh niceties with improved Bourne syntax
- fish is something else
- so is rc