Slide 1

Administrivia

- As noted in e-mail, reading quiz 1 graded and sample solution and grades uploaded. I was generous with points, so I say worth looking at sample solution even if you did well.
- Reminder: Reading Quiz 3 due today; Homework 1 due end of the week.
- More assignments coming soon.

Slide 2

Shell Input as a Programming Language and Shell Scripts

- What `bash` understands is in a sense a programming language, with the shell as its interpreter:
  - Variables (usually untyped).
  - Expressions (arithmetic and logical).
  - Conditionals (if/then/else) and loops.
  - Functions.
- I'll talk about `bash`, but most shells provide similar functionality, just sometimes with different syntax.

Slide 3

Shell Input as a Programming Language — the Good

- Interactive shells are a kind of REPL (read, evaluate, print loop) for the shell's language. So you can use the various features interactively or use them to write "scripts" — in the same way you can test out ideas in Scala's REPL and then use them in programs (except that Scala's REPL is mostly useful for testing/development, whereas using shell features such as loops interactively can be useful).
- Any UNIX/Linux system will have a shell of some sort, I think always one that supports basic `sh` functionality, while which "real" programming languages are available might vary.

Slide 4

Shell Input as a Programming Language — the Bad

- Writing portable scripts is tough. Sticking to the `sh` subset of `bash` helps, as does avoiding GNU-only commands and extensions, but how to do that — yeah well. (It's a little like writing portable C.)
- What you can do is somewhat limited, and scripts of any size are apt to be ugly.
- Advice: For long and complex scripts, a scripting language such as Perl or Python may be a better choice than a shell script.

Slide 5

Shell Input as a Programming Language — the Ugly

- Dealing with spaces (in filenames, e.g.) is a huge pain. Rules for quoting are tricky, and sometimes it seems the only way to get it right is to just try things until something works. (Yuck!)
- There are many weirdnesses having to do with when subshells are created, for example the behavior of `while` and shell variables (more later).

Slide 6

Shell Scripts

- A "shell script" is just a sequence of things you could type at the shell prompt, collected in a (text) file.
- Normally, first line of script is `#!` ("hash bang") followed by path for shell (`/bin/bash`, e.g.), and the file is marked "executable" (with `chmod`). But you can also execute commands in file `anyfile` (even if not marked executable) via `sh anyfile` (or `bash anyfile`).
- With the exception of the first line, lines starting with `#` are comments.
- (`hello` example.)

Slide 7

Shell Variables

- Define/assign variables with, e.g., `myvar="hello"`. (Notice absence of spaces.)
- Reference with, e.g., `$myvar`.
- What's the difference between these and "environment variables" already mentioned? Shell variables are local to the shell, not passed on to child processes. Environment variables are (potentially) available to child processes. Distinction is somewhat blurred in Bourne shells. Convention is that environment variable names are all caps.

Slide 8

Shell Functions and Parameters

- Define functions as described previously(?) — name, parentheses, then function definition in curly brackets. Separate/end commands with `;` or newlines. Can precede with `function`.
- Parameters for functions and shell scripts are positional — `$0` for script name, then `$1`, etc. (much like arguments to C program). `$*` is a list of all parameters; `$#` is the count of parameters, not including `$0`.
- Call functions or shell scripts by giving name and then parameters, separated by whitespace. (If a parameter should include whitespace, use quoting or escape characters.)

Slide 9

Shell Functions and Parameters, Continued

- `fcn-example` example.
- Note that you can do this interactively too! a feature I often find useful.

Slide 10

Command Substitution

- Can "inline" output of one command as parameters of another. Old style uses backquotes, e.g.:

  ```
  vim `find . -name "*.c"`
  ```

  (Note that these are *backquotes*, not single quotes!)

  Or use newer `bash` syntax

  ```
  vim $(find . -name "*.c")
  ```

  (Much easier to nest!)

- The "inlined" command can even be a pipeline. Example:

  ```
  ls -ld $(echo $PATH | sed 's/:/ /g')
  ```

Slide 11

Two More Useful Commands

- `basename` and `dirname` split up pathname into "base" (last level of path) and rest of path.
- Very helpful in combination with command substitution, especially in scripts.

Slide 12

Conditionals and Loops

- (Next time.)

Slide 13

Minute Essay

- Questions?