Slide 1

Administrivia

- Reminder: Homework 4 due today. Homework 5 due next Monday.
- Homework 3 grades mailed. Sample solution linked from “lecture topics” page.

Slide 2

Minute Essay From Last Lecture

- Most people came fairly close on the regular-expression question.
- Most people had seen regular expression, usually in Scala.

Slide 3

Text Editors Revisited

- Some text editors (`vim` among them) allow you to “filter” text through an external program.
- One thing this allows is building on-the-fly scripts — construct in `vim` the lines to execute, then execute them with, e.g., `:%!sh`. (No need to save unless you want to reuse another time.)

Slide 4

On-the-fly Scripts, Continued

- I like this “on-the-fly scripting” for various kinds of file moving/renaming operations — use `r!ls` to get a list of files, “massage” with various editing operations, then execute as above. I find this works well as a way of dealing with filenames containing spaces — relatively easy to add double quotes around names. A useful idiom employs a simple regex and `&` to reference the matched text, e.g.,

```
:%s/.*/mv -v "&" targetdir/
```

- (Of course I could also use a `bash` loop, and sometimes I do, but — whatever seems easiest for the particular use case?)

Slide 5

Text Editors Revisited, Continued

- I also use `vim`'s ability to record and play back "macros" fairly regularly. To do this: Start recording with `q` plus a single letter. End with another `q`. Play back with `@` and the single letter.

  (Somewhere sometime I think I remember a comment to the effect that with regard to certain repetitive tasks there were two kinds of people — the ones who write macros and the ones who write a regular expression. I do both, depending on the situation.)
- It can be tricky to record in a way that will "play back" effectively, but when this works, it works well.
- I use this sometimes when I need to make the same edit to several files.

Slide 6

Regular Expressions — Recap

- Regular expressions can be complex — constructs include "character classes", repetition, "or", and back references.
- Can be very powerful but also very cryptic.

Slide 7

Regular Expressions — Example

- As an example, consider taking a “class roster” as produced by TigerPaws (showing student name, ID number, e-mail address, etc.,) and extracting from it just the student’s last name and e-mail address. Example line:

  ```
  Lastname, Firstname M.  1234567 lastname@whatever        SR       New
  ```

  Here’s a `vim` command to do that:

  ```
  :%s/\(.\+\),.*\d\d\d\d\d\d\d\s\(\S\+\)\s.*/\2 \1/
  ```

  (Admittedly it did take a few tries to get right!)
- Translation of expression being changed: Any number of characters up to a comma (capture this as group 1); a comma, any number of characters, seven digits plus a space/tab; one or more nonwhitespace characters (capture this as group 2); a space/tab and any number of characters.

Slide 8

Regular Expressions — Example

- As another example, consider revising that little script that computes factorials using a recursive shell function. Really would be nice if it rejected invalid input.
- Can do this fairly easily with `grep` and regular expression. Package this as a function in a separate file and “source” it from the factorial script. (See “sample programs”.)

Slide 9

Scripts Versus Programs

- One of the parts of grading I've tried to semi-automate is adding up point deductions and computing grade.
- For grading on paper, I wrote a script that takes a total number of points and a list of deductions/additions and computes a total. I call it from `vim` to compute and record. (This works because I keep grade information in text files. I don't like spreadsheets!)
- For grading electronically, at one time I had scripts that would take one of my "scores and comments" files and compute a total from point deductions/additions in the file, but at some point it got unwieldy, and I wrote a program. (In Perl, if you're curious. It seemed like a Perl kind of job?)

Slide 10

The `make` Utility

- Motivation: Most programming languages allow you to compile programs in pieces ("separate compilation"). This makes sense when working on a large program — when you change something, just recompile parts that are affected.
- Idea behind `make` — have computer figure out what needs to be recompiled and issue right commands to recompile it.
- (I think all of you have seen this, in CSCI 1120? but review.)

Makefiles

Slide 11

- First step in using `make` is to set up "makefile" describing how files that make up your program (source, object, executable, etc.) depend on each other and how to update the ones that are generated from others. Normally call this file `Makefile` or `makefile`.

  Simple example on "sample programs" page.
- When you type `make`, `make` figures out (based on files' timestamps) which files need to be recreated and how to recreate them.

Useful Command-Line Options

Slide 12

- `make` without parameters makes the first "target" in the makefile.
  `make foo` makes `foo`.
- `make -n` just tells you what commands would be executed — a "dry run".
- `make -f otherfile` uses `otherfile` as the makefile.

Slide 13

Defining Rules

- Define dependencies for a rule by giving, for each “target”, list of files it depends on.
- Also give the list of commands to be used to recreate target.

  NOTE!: Lines containing commands must start with a tab character. Alleged paraphrase from an article by Brian Kernighan on the origins of UNIX:

  > The tab in makefile was one of my worst decisions, but I just wanted to do something quickly. By the time I wanted to change it, twelve (12) people were already using it, and I didn’t want to disrupt so many people.

Slide 14

Phony Targets

- Normally targets are files to create (e.g., executables), but they don’t have to be. So you can package up other things to do . . .
- Example — many makefiles contain code to clean up, e.g.:

```
clean:
        -rm *.o main
```

  To use — `make clean`.

Slide 15

Variables in Makefiles

- You can also define variables, e.g.:
  - List of object files needed to create an executable. Then use this list to specify dependencies, command.
  - Pathname for a command, options to be used for all compiles, etc.
- (Used in example.)

Slide 16

Predefined Implicit Rules

- `make` already knows how to “make” some things — e.g., `foo` or `foo.o` from `foo.c`.
- In applying these rules, it makes use of some variables, which you can override.
- A simple but useful makefile might just contain:

  ```
  CFLAGS = -Wall -pedantic -O -std=c99
  ```
- This can also make longer makefiles not as long. (Revised example.)

Minute Essay

- I’ve described some of the ways I use some of the tools discussed. How about you — anything we’ve talked about thus far that you’ve been able to put to use to do something that helps you?