Slide 1

Administrivia

- Reminder: Homework 2 due today (written problems in hardcopy by 5pm, programming problems by e-mail by 11:59pm).
- Homework 3 on the Web; due next week.

Slide 2

Minute Essay From Last Lecture

- Most people seem to use `vi` in Linux, because it's what we teach in CS1, though there was a mention of `gedit`. You might try `gvim` if you're working at the console.

Slide 3

Regular Expressions

- From an old Wikipedia definition:

  A regular expression (abbreviated as regexp, regex or regxp) is a string that describes or matches a set of strings, according to certain syntax rules. Regular expressions are used by many text editors and utilities to search and manipulate bodies of text based on certain patterns.

- Idea has roots in formal theory of languages, where the "languages" (sets of strings) described by regular expressions are exactly the ones accepted by finite state automata.

Slide 4

Regular Expressions and UNIX Tools

- Tools that use regular expressions include editors and also text-manipulation commands such as `grep` and `sed`. Also supported in many programming languages, especially ones for scripting (Perl, Python, `bash`, etc.).

- This being UNIX, not all the tools accept exactly the same syntax. POSIX defines two standards, "basic" and "extended". Some tools/languages add more. Simple stuff is very similar in all versions, fortunately. Key difference — in basic syntax, must precede many special characters with "escape character" (backslash).

  Also notice that to keep shell from doing its thing with your regular expressions (which generally you don't want), must enclose in single or double quotes.

Slide 5

Character Literals and Metacharacters

- Most characters represent themselves.

  `hello` matches what?

- Other characters are “special” (metacharacters):

  `^` matches start of line

  `$` matches end of line

  `.` matches any character (except newline)

  To use these as regular character literals, “escape” with a backslash.

Slide 6

Character Literals and Metacharacters, Continued

- Examples of use:

```
grep "hello" foo
grep "^hello" foo
grep "hello$" foo
grep "^hello$" foo
grep "h.llo" foo
grep "h\.llo" foo
```

Slide 7

Character Classes

- Character classes represent “one of these characters”.

  Examples: `[abcd]`, `[0-9]`
- `^` at the start of a list means “any character other than these”:

  Example: `[^abcd]`
- Most tools define some shorthand:

  Examples: `\s` for whitespace, `[:alpha:]` for letter

Slide 8

Character Classes, Continued

- Examples of use:

```
grep 'h[ae]llo' foo
sed 's/[A-Z]/!/g' foo
sed 's/[A-Za-z0-9]/!/g' foo
sed 's/[^A-Za-z0-9]/!/g' foo
sed 's/[[:alnum:]]/!/g' foo
sed 's/[^[:print:]]/!/g' foo
```

Slide 9

"OR" (Alternation)

- UNIX pipe symbol (`|`) separates alternatives. (Must escape in basic syntax.)
  Example: `cat|dog`
- (What about AND? Usually don't need it, or can get the same result another way. For `grep`, pipe one `grep` into another.)
- Example of use:

```
grep 'cat\|dog' foo
```

Slide 10

Quantifiers

- `*` means "preceding character (or group), zero or more times".
  Example: `.*`
- + means "preceding character/group, one or more times". (Must escape in basic syntax.)
  Example: `a+`
- `{N,M}` means "preceding character/group, N to M times". (Must escape curly brackets in basic syntax.)
- Notice that quantifiers are "greedy" — match longest string possible.
- Examples of use:

```
sed 's/[0-9]\+/NUMBERS/g' foo
sed 's/[0-9]\{2\}/NUMBERS/g' foo
sed 's/[0-9]\{1,4\}/NUMBERS/g' foo
```

Slide 11

Grouping in Regular Expressions

- Use parentheses to group. (Must escape them in basic syntax.)

  Example: `(abc)(def)`

  Example: `(abc)*`

- Can then "backreference" groups, with `\1`, `\2`, etc.

  Example: `(abc)(.*)\1`

- Examples of use:

```
sed 's/\(hello\|bye\)\+//g' foo
sed 's/\(^.\)\(.*\)/\2\1/' foo
```

Slide 12

A Few More Tricks

- Angle brackets match beginning/end of word. (Must escape in basic syntax.)

  Example: `<hello>`

  (Notice that this doesn't work on Mac OS X. Instead, one must use "character classes" `[[:<:]]` and `[[:>:]]`.)

- Examples of use:

```
grep '\<bye\>' foo
```

Slide 13

Usage of Regular Expressions, Revisited

- Can use regular expression to search — `grep`, search in `vi`.
- Can also use them to modify — `sed`, search-and-replace in `vi`.
  Backreferences can be useful here!
  Example: `s/\(\^..\)\(.*\)/\2\1`
- Examples of use:

```
sed 's/\(.\+\) \(.\+\)/\2 \1/' foo
```

Slide 14

Where to Learn More

- `man` and/or `info` pages for `sed`, `grep`; `info` page for `regex`.
- Online help for `vim`.
- Books and online references/tutorials . . .
- Useful advice from `vim`'s help:

  > Which of these should you use? Whichever one you can remember.

- There are also programs that offer a GUI-ish environment for trying things out.
  Time permitting I will install one or more on the classroom/lab machines.

Slide 15

A Few More Things

- How to search for / replace a literal backslash? `[\]` works. `\\` also works, but to pass that to `grep`, it appears that you have to enclose the string in single rather than double quotes.
- A student in a previous year pointed out that backslash does seem to have different meanings in different contexts here. From the `info` page from `regex`:

```
     The '\' character has one of four different meanings, depending on
  the context in which you use it and what syntax bits are set (*note
  Syntax Bits::).  It can: 1) stand for itself, 2) quote the next
  character, 3) introduce an operator, or 4) do nothing.
```

  A bit strange, but in practice, I claim one can get used to it.

Slide 16

Minute Essay

- Try writing a regular expression that would match a "license plate" string of the form "one uppercase letter, then two digits, then three uppercase letters". (Hint: Remember that `[A-Z]` matches one uppercase letter. Similar syntax for digits.)

Slide 17

Minute Essay Answer

- A not-so-hard-to-remember answer:

```
[A-Z][0-9][0-9][A-Z][A-Z][A-Z]
```