TEXT PROCESSING COMMANDS: FIND AND GREP

find: print the location of a particular file or files with a filename matching name
To find all Makefiles in Elizabeth's 2270SP04 directory:
```
        find ~ekwhite/2270SP04 –name Makefile
```
To find all C++ files in my csci2270 directory:
```
        find ~/csci2270 –name '*.cxx'
```
To find all header files in Elizabeth's class directory:
```
        find ~ekwhite –name '*.h'
```
You may need to add the option –print on some machines to see the output:
```
        find ~ekwhite –name '*.h' -print
```
The find command doesn't work according to quite the same rules as the regular UNIX commands we discussed last week—it's inherently recursive (so it looks in all subdirectories underneath the current one) without needing an –R flag.

grep: search files for a particular piece of text.
You can specify a file or some files to start in. To find files containing the string 'poly' in last week's lab:
```
        grep -l 'poly' ~ekwhite/2270SP04/lab6/*
```
To find files containing the string 'reserve' in last week's lab, and print the line number for each match:
```
        grep -n 'reserve' ~ekwhite/2270SP04/lab6/*
```
It's possible to send the grep command off on long, fruitless searches (especially if you forget to supply a filename); if it seems to be taking forever, try Control-c to cancel.

REGULAR EXPRESSIONS

To get the most out of find and grep, we can write REGULAR EXPRESSIONS (REs for short) to match patterns of characters, instead of single strings. REs are a product of some fairly theoretical (or mathematical) computer science, and this is only a quick and dirty introduction, not a thorough treatment. See Chapter 6 of *UNIX in a Nutshell*, 3rd ed., by A. Robbins, published by O'Reilly, Cambridge, 1999 for more on this.

The simplest regular expressions look like regular text:
```
        poly                ## matches poly inside of words or by itself
```
and you can use * to stand for anything:
```
        pol*                ## matches poly, polo, polenta, etc.
        *pol*               ## matches the above plus Interpol, apolitical, etc.
```
But that alone isn't so powerful—we'd like a way to look for more general strings. You can mix up any of the matching techniques described below:

Certain common sets are defined for regular expressions, like [A-Z], the set of capital letters, [A-Za-z] or [0-9], the set of digits. Suppose we wanted to find files that contained the word poly or Poly; we could define the first letter as one of the set P or p:
```
        [Pp]oly
```
Or we could broaden the search, like this:

```
P[aeiou]ly        ## matches paly, pely, pily, poly, or puly
```

We can request any member that isn't in a set by negating it with ^:

```
P[^aeiou]ly       second letter is a consonant, uppercase, or non-letter
                  (it's NOT a, e, i, o, or u)
```

We can match any character with a ., so that

```
c.f
```

matches a sequence starting with c and ending with f, with a single letter in between, and

```
c….f
```

matches a sequence starting with c and ending with f, with 4 letters in between.

We can also use the * here to define "zero or more matches" to a RE. If

```
[A-Z]
```

matches any capital letter, then

```
[A-Z]*
```

matches zero or more capital letters. This is a pretty broad matching! In general, you'll want to match "one or more" of something. To match one or more capital letters using a RE, say

```
[A-Z][A-Z]*
```

To match zero or more letters, use

```
.*
```

We can also use 'positional anchors' to match text at the beginning of a line (like a variable declaration). Notice that ^ means something different inside and outside the [ ]!

```
^polynomial
```

Or we can ask to match at the end of the line:

```
polynomial$
```

Or we can match both (so it's the only thing on the line):

```
^polynomial$
```

To try this stuff out, you can grep through some of my text files, which are in ~ekwhite/2270SP04/lab7. The Napoleon file is an old English nursery rhyme and other three files are some very strange poems by Edward Gorey. You can look at them by typing:

```
emacs ~ekwhite/2270SP04/lab7/gashlycrumb_tinies &
emacs ~ekwhite/2270SP04/lab7/limerick1 &
emacs ~ekwhite/2270SP04/lab7/limerick2 &
emacs ~ekwhite/2270SP04/lab7/napoleon &
```

RE EXERCISES

Here's a simple example that doesn't involve a regular expression, but a string:

```
grep –l 'Bonaparte' ~ekwhite/2270SP04/lab7/*
```

It should give you the name of a file containing this text (napoleon). Typing

```
grep –l 'fennis' ~ekwhite/2270SP04/lab7/*
```

should give you no matches, because this string doesn't exist in my directory. But

```
grep –l ‘[Ff]ennis’ ~ekwhite/2270SP04/lab7/*
```
should give you the name of a file containing this text.
If you want to see particular lines from a file, you can use the –n option:
```
grep –n ‘.*unnery’ ~ekwhite/2270SP04/lab7/*
```
should print you the 3 lines of the file limerick2 that contain matches for this text.
If you say
```
grep –n ‘^A’ ~ekwhite/2270SP04/lab7/gashlycrumb_tinies
```
you’ll see a line detailing the sad fate of Amy.
If we wanted to use a RE for this, we could also say:
```
grep -n '^[A][A]*' ~ekwhite/2270SP04/lab7/gashlycrumb_tinies
```
Since the poem’s alphabetically based, what RE-based line would tell you the awful end of the people whose names begin with M and N? What RE-based grep command would print all of the lines in the poem?
To use a RE to find a file containing a period is a problem. To see this, try:
```
grep -n '.' ~ekwhite/2270SP04/lab7/*
```
What’s the result? Why do you match so many lines that don’t have a period? Now try:
```
grep -n '\.' ~ekwhite/2270SP04/lab7/*
```
Using the backslash forces grep to interpret the period as a period.
To find lines ending with characters other than letters and numbers, we can use negation to exclude letters and numbers:
```
grep -n '[^A-Za-z0-9]$' ~ekwhite/2270SP04/lab7/*
```
This gives back the lines ending with punctuation marks.

We can also use REs in the find command. Suppose that we’re looking for a file whose name contains the string tiny or tinies, but we aren’t sure which. We can make a RE to search for either the string ‘tiny’ or ‘tini’:
```
find ~ekwhite/2270SP04/lab7 -name '*tin[iy]*'
```
This should return the gashlycrumb_tinies file.
Or we can look for files with names ending in numbers:
```
find ~ekwhite/2270SP04/lab7 -name '*[0-9]'
```
Or we can look for files with names that do not end in numbers:
```
find ~ekwhite/2270SP04/lab7 -name '*[^0-9]'
```
We can look for files that contain characters other than letters and numbers:

Regular expressions are particularly useful for searching for partial filenames or searching a bunch of files for a particular bit of text. You can also employ them in search & replace operations to edit text, which we’ll hopefully discuss later.

AND ONE OTHER MATTER: TERMINATING RUNAWAY PROCESSES

ps: identify processes like netscape or emacs running on a particular machine
kill: force a process to end when it’s stopped behaving

Occasionally, you will have a runaway process and you will need to kill it. For instance, Netscape often locks up on its own for no apparent reason. You may not be able to log off a machine properly if you still have processes running. Or you may get nasty email from CSOps indicating that you have ‘zombie’ processes running on the machines. It’s polite to kill these, since they can still be using memory resources and slowing everyone else down. To fix a hung process on a machine you aren’t logged onto at present, you need to log onto another machine and connect by using ssh to the machine with the hung process.

Suppose that the process is hung on the machine splashdance. We’d log onto another machine, and then connect to it remotely by typing:

```
ssh splashdance
```

and giving the password again.

Next, we’d use ps to inquire about what’s running on the machine. The –e option indicates that you want all processes to display (I have no idea why this isn’t the default), and the –l option requests a long listing. The command is used like this:

```
ps –el
```

Used on its own, ps –el returns a series of listings, like the partial one shown below:

```
ssh splashdance [~] % ps -el
F S   UID   PID  PPID  C PRI  NI ADDR    SZ WCHAN  TTY          TIME CMD
4 S     0     1     0  0  75   0    -   344 schedu ?        00:00:03 init
1 S     0     2     1  0  75   0    -     0 contex ?        00:00:00 keventd
1 S     0     3     1  0  75   0    -     0 schedu ?        00:00:00 kapmd
1 S     0     4     1  0  95  19    -     0 ksofti ?        00:00:00 ksoftirqd_CPU0
1 S     0     9     1  0  85   0    -     0 bdflus ?        00:00:00 bdflush
1 S     0     5     1  0  75   0    -     0 schedu ?        00:00:00 kswapd    ….
```

You’ll see an integer number marked PID in the output. This is the process id, which is how the computer identifies the processes; there’s also a short name for each one.

We want to use grep to search the listing from ps –el for the name or the PID of our runaway process. Suppose a Netscape process is hung. We’d send the output text from ps –el right into the input for grep, using a pipe (|). We’ll get into pipes more later.

```
ps –el | grep netscape
```

and we’d get a line like

```
0 S 56789  8967   1 98  75  0    - 7681 schedu ?   6-22:13:05 netscape
```

The fourth number from left to right is the process id (8967 for this example, but of course likely different for you). If you are killing a process because you have gotten mail from trouble, you should match the PID they give you in the email to this number. We use the PID when we ask the shell to kill the job:

```
kill -TERM 8967
```

Then we check again in a few seconds to see if the process is still running:

```
ps –el | grep netscape
```

and we hope for a blank line, which tells us it was successfully killed. If not, we can try

```
kill –HUP 8967
```

and check for the process again, or

```
kill –SEGV 8967
```

or (if nothing else works)

```
kill –KILL 8967
```