Text Editing (continued)

BASICS OF THE UNIX/LINUX ENVIRONMENT

Command line editors

ed

Don't even think of using ed, if you accidently type it, enter ^D to get out.

Command line editors

edit

Don't even think of using edit, if you accidently type it, enter "exit" to get out.

Command line editors

sed

sed:

powerful command line text editor. ("the ultimate" stream editor, non-interactive).

It takes standard in, edits each line, and spits it to standard out.

It uses regular expressions for pattern matches.

I don't use this often.

sed:

sed has several commands, but most people only learn the substitute command: *s*.

The substitute command changes all occurrences of the regular expression into a new value.

A simple example is changing "day" in the "old" file to "night" in the "new" file:

```
%sed s/day/night/ <old >new
```

sed:

```
%echo day | sed s/day/night/
night
```

It does what you tell it.
(law of unintended consequences)

```
%echo Sunday | sed 's/day/night/'
Sunnight
```

sed:

4 parts to substitute command

s Substitute command

/../../ Delimiter

day Regular Expression Pattern
Search Pattern

night Replacement string

sed:

Most examples of sed are incomprehensible

```
sed 's/[^ ]*/(&)/' <old >new
sed 's/[^ ][^ ]*/(&)/g' <old >new
sed 's/^\([^:]*\):[^:]:/\1::/' </etc/passwd >/etc/password.new
```

count the number of lines in three files that don't begin with a "#:"

```
sed 's/^#.*//' f1 f2 f3 | grep -v '^$' | wc -l
```

sed:

Is very useful when you really need it.

vi and vim
(use the same command set as ed/edit/sed!
This is Unix, reuse the same tools.)

to start it up

```
%vim [name-of-file]
```

vi and vim are have what is called a “modal” interface.

They have two modes

“normal” = command mode
insert = input mode

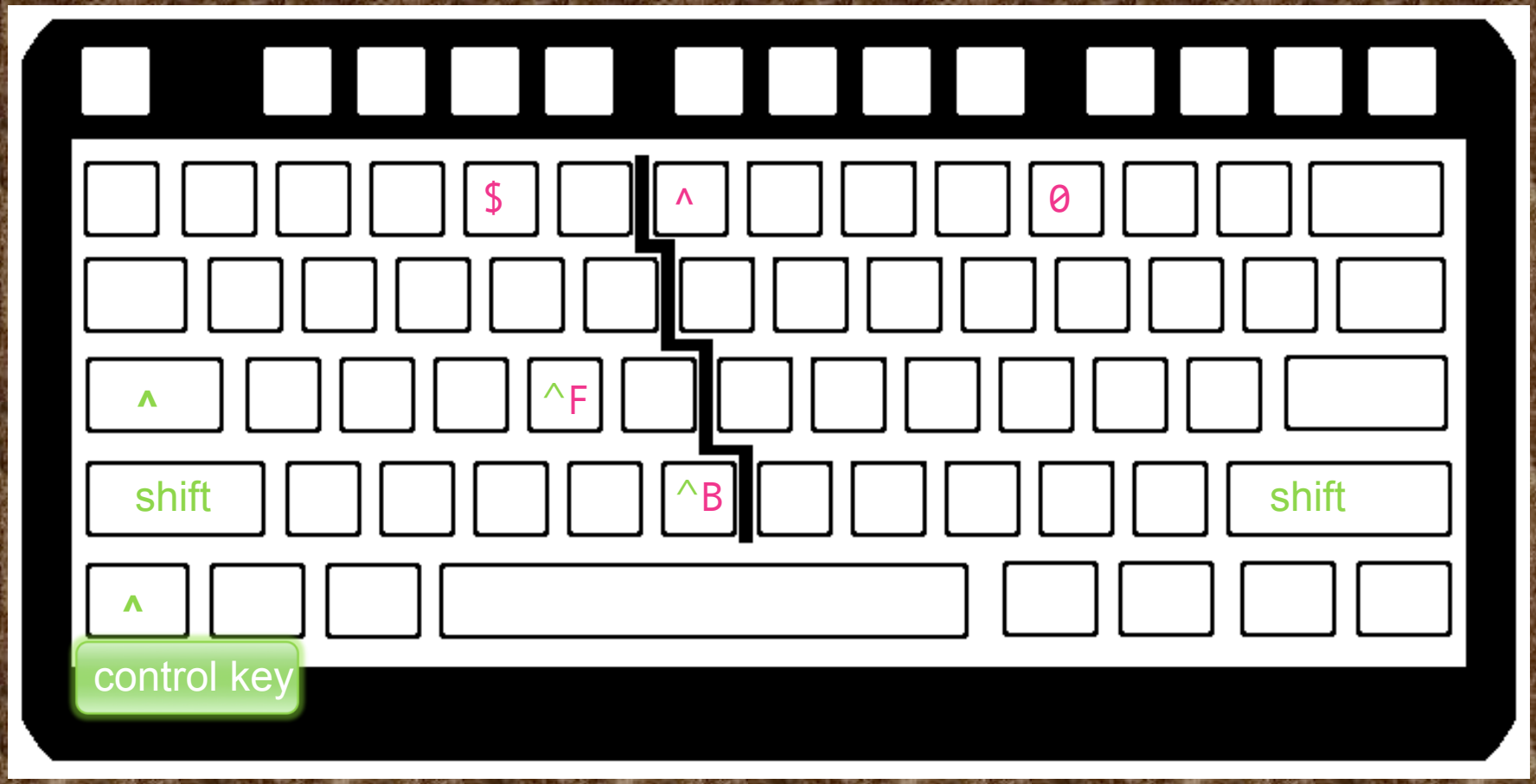
Entering text takes place in insert mode and the editing power comes to the fore in command mode.

Use **esc** to return to command mode from insert mode.

/Users/hdeshon/Projects/SUMATRA/bin

```perl
# use perl
eval 'exec perl -S $0 "$@"'
if 0;

use lib "$ENV{ANTELOPE}/data/perl" ;

use Datascope ;


#FILT data/2804/WMQ.BHZ_00.2005:044:01:29:22
#P 443.46 0 pP 456.21 0 sP 462.81 4      u 443.46 0 u 446.36 3 u 447.76 3 u 461.21 2 u 468.46 3
#print "123456789012345678901234567890123456789012345678901234567890123456789\n";
#print "         1         2         3         4         5         6\n";

$in=shift;
$events=shift;
$out=shift;

open(EV,"$events") or die;
while(<EV>) {
        chomp;

        $evid=(split /\s+/,$_)[0];
        ($oyear{$evid},$omo{$evid},$oday{$evid},$ohr{$evid},$omin{$evid},$osec{$evid},$olat{$evid},$olon{$evid},$odepth{$evid},$omag{$evid})=
                        (split /\s+/,$_)[2,3,4,5,6,7,8,9,10,11];
        $stime = "$omo{$evid}/$oday{$evid}/$oyear{$evid} $ohr{$evid}:$omin{$evid}:$osec{$evid}";
        $time{$evid} = str2epoch($stime);

}
close(EV);

open(IN,"$in") or die;
open(OUT,">$out") or die;
@line=<IN>;

$oldevid=0;
ILOOP: for($i=0;$i<@line;$i=$i+2) {
        chomp $line[$i];
        chomp $line[$i+1];
        $n=0;$m=0;$p=0;$c=0;
        @goodP="";@goodpP="";@goodpwP="";@goodsP="";
        @badP="";@badpP="";@badpwP="";@badsP="";
        @newtv="";$ct=0;

        #find origin time of sac file
        $file=(split /\s+/,$line[$i])[1];
        $tmp= `getsachdr $file kzdate kztime`;
        ($ayear,$ajday,$ahr,$amin,$asec)=(split /\s+/,$tmp);
        if ($ajday<10) { $ajday="0".$ajday; }
        if ($ajday<100) { $ajday="0".$ajday; }
        $string="$ayear$ajday $ahr:$amin:$asec";
        print "$string\n";
        $atime=str2epoch($string);

        #rest
                                                                                1,1          To
```

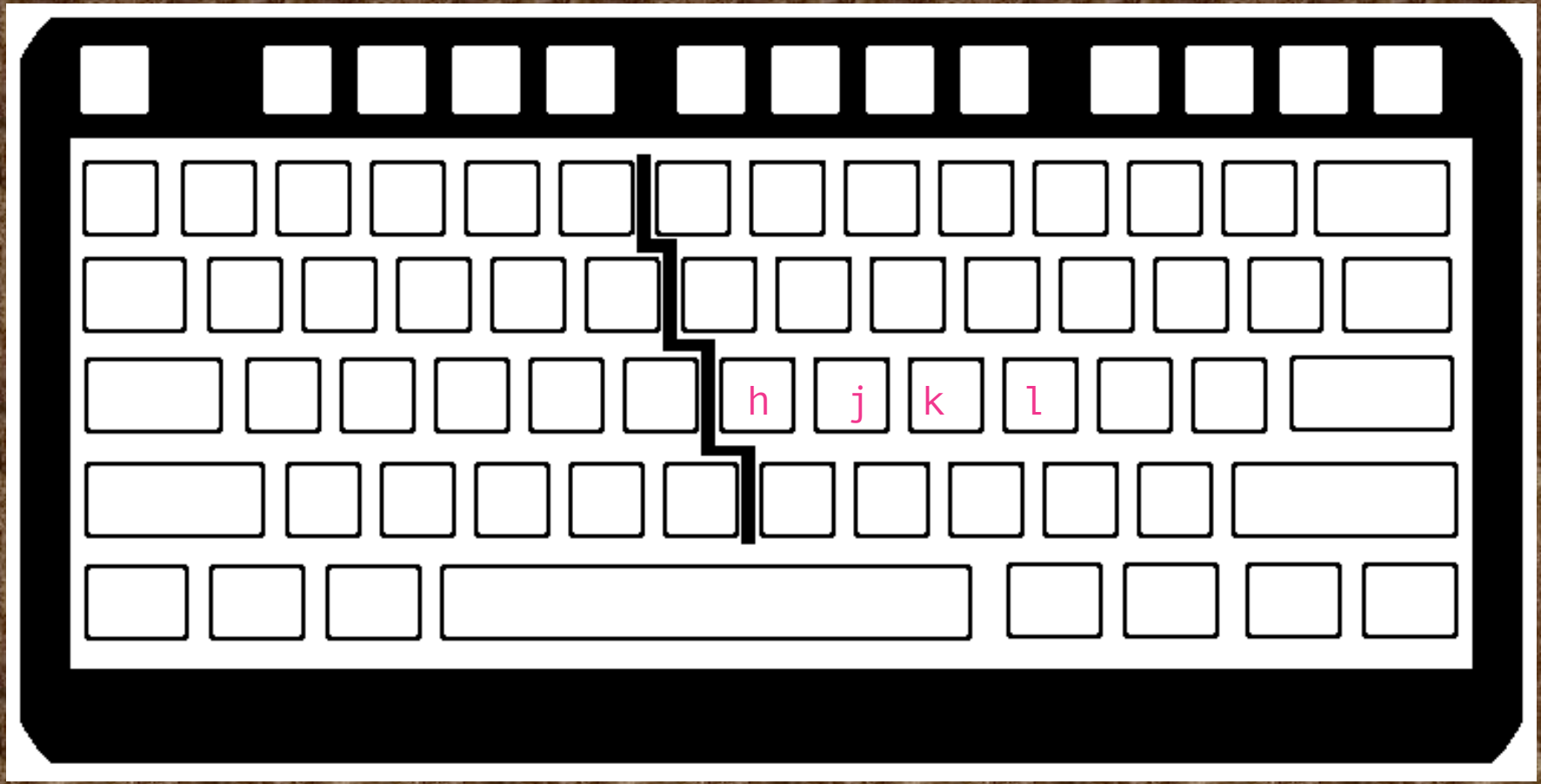
moving the cursor



$ -- go to end of line (eol)
0 -- go to beginning of line (bol)
^ -- go to first character at bol

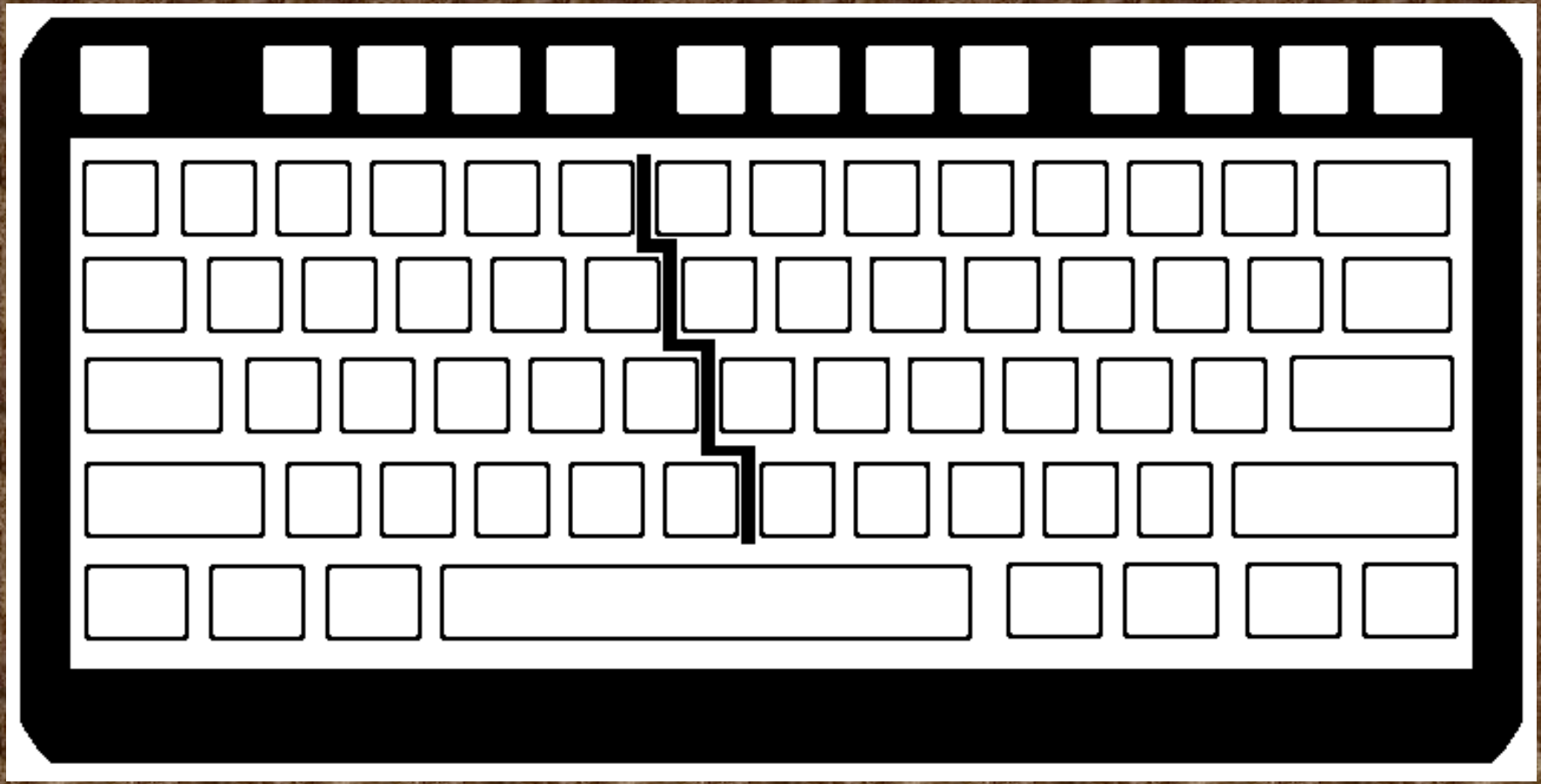
^F -- scroll screen forward
^B -- scroll screen backwards

moving the cursor



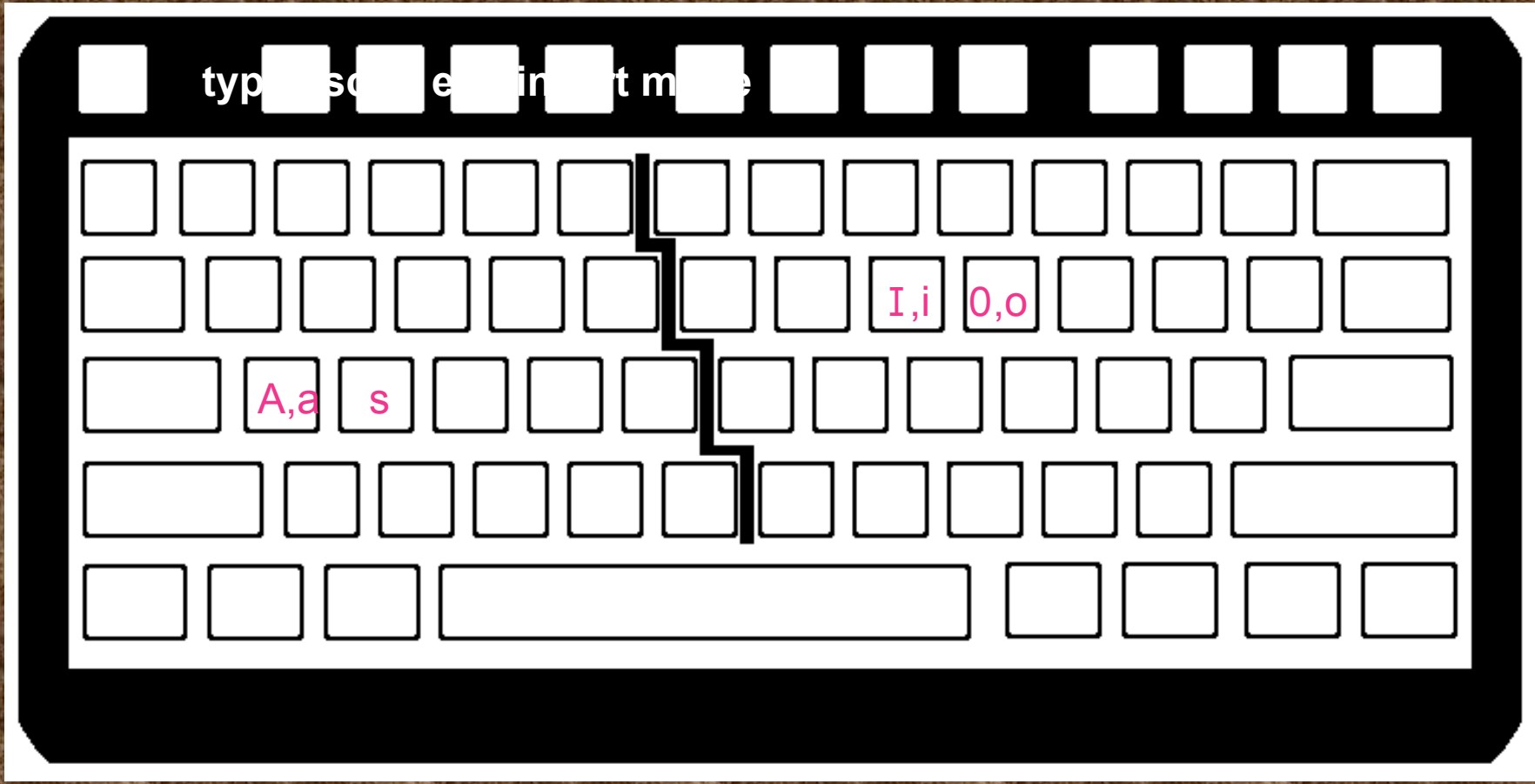
The “h”, “j”, “k”, and “l” keys move the cursor left, down, up, and right respectively. This is very fast and efficient for a touch typer.

moving the cursor



Use the arrow keys. (slow – have to take you right hand off the keyboard.)

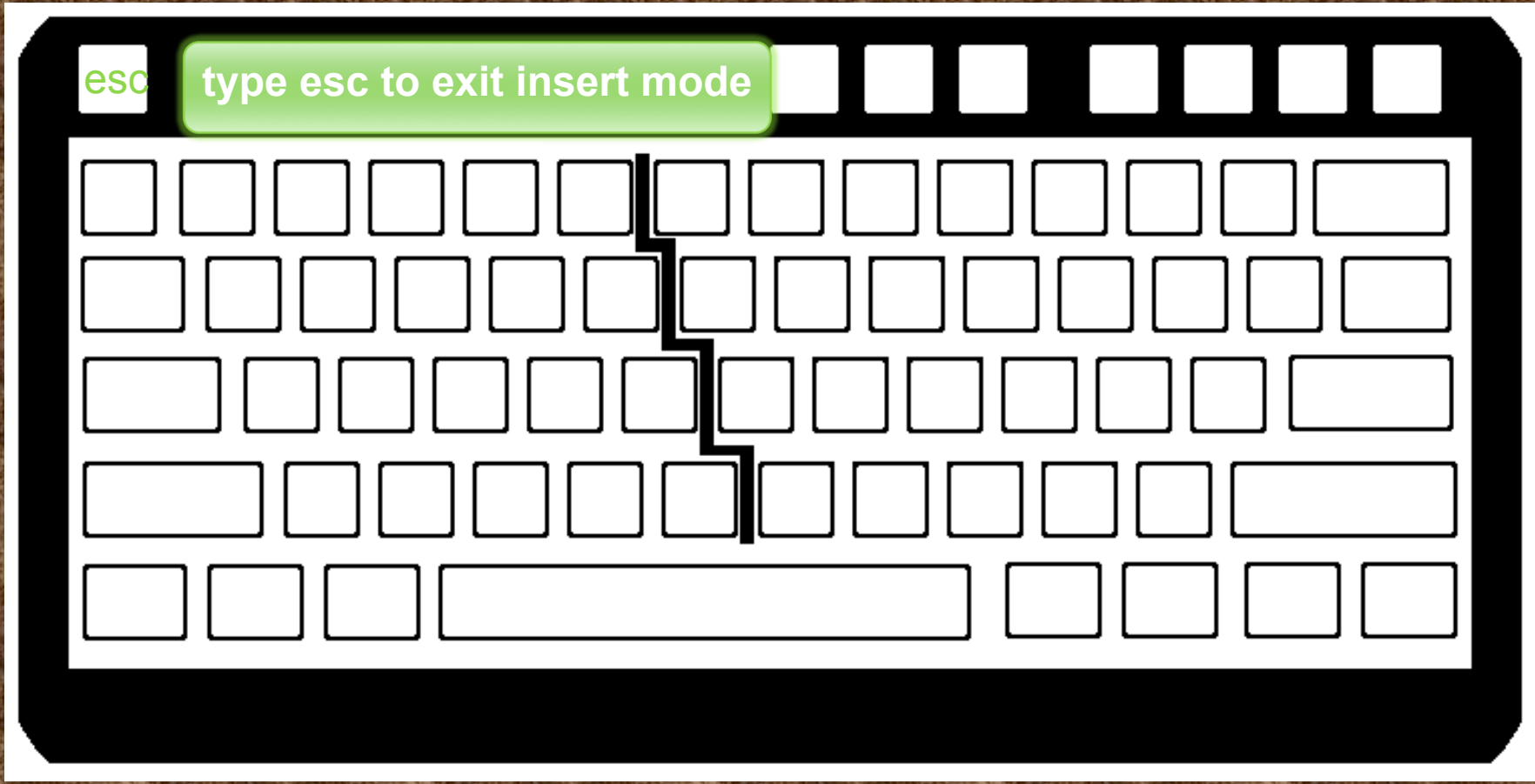
to enter insert mode



i -- insert
a -- append
s -- substitute

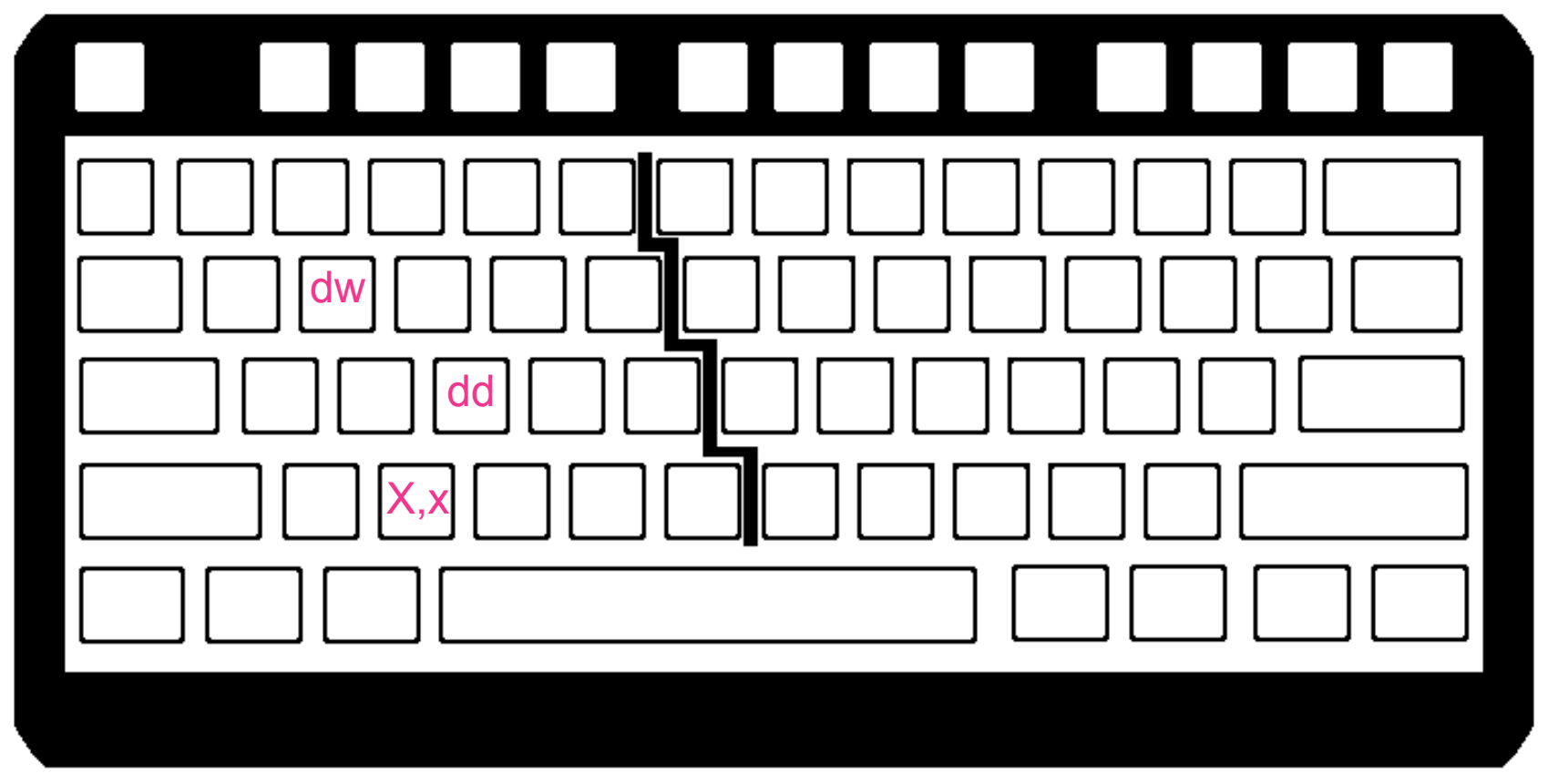
A -- append at eol
I -- insert at bol
O – start new line & insert

to exit insert mode



esc to exit insert mode.

deleting text



x -- delete character behind cursor

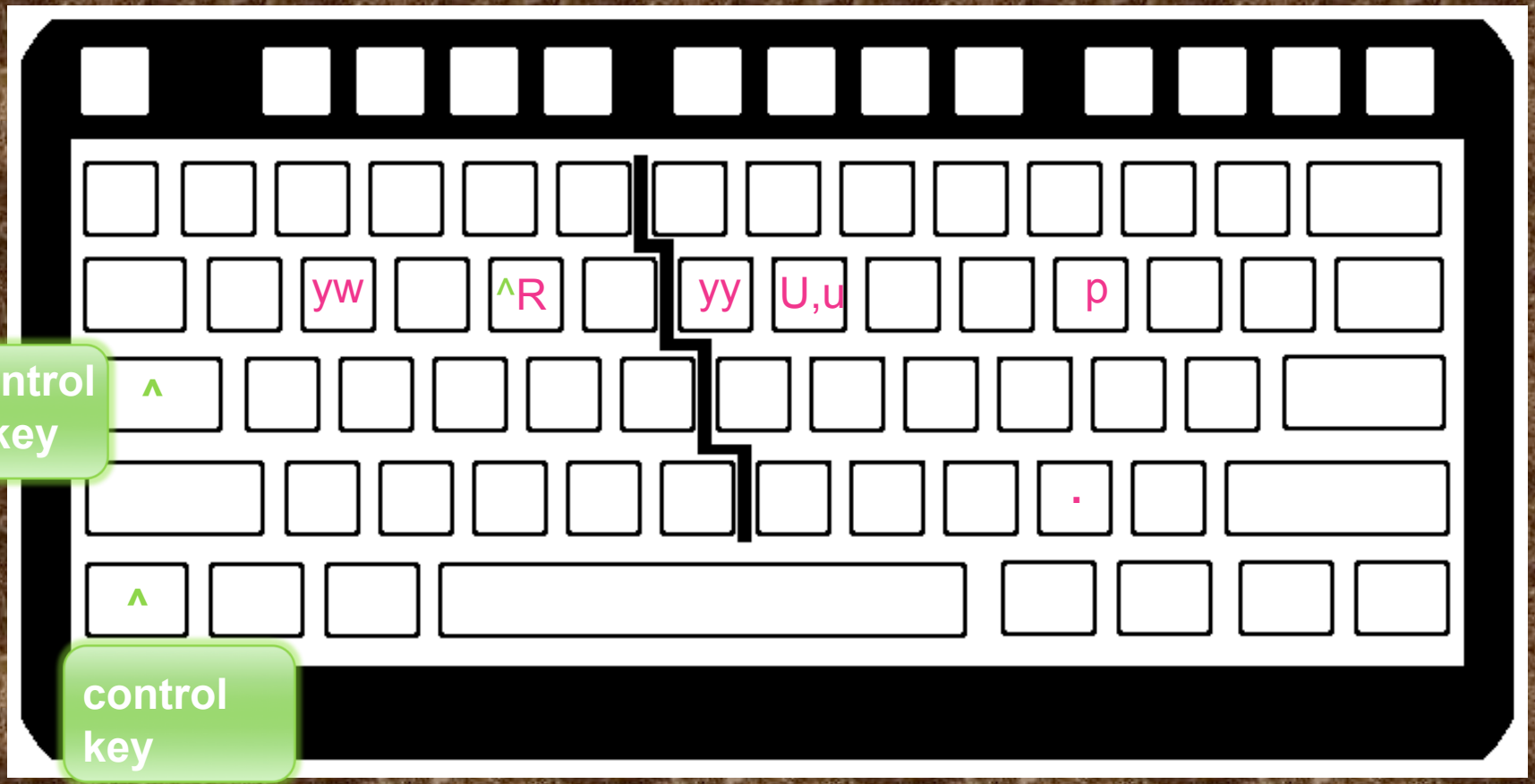
X -- delete character before cursor

dw -- delete word

dd -- delete line

Xdd -- delete next X lines

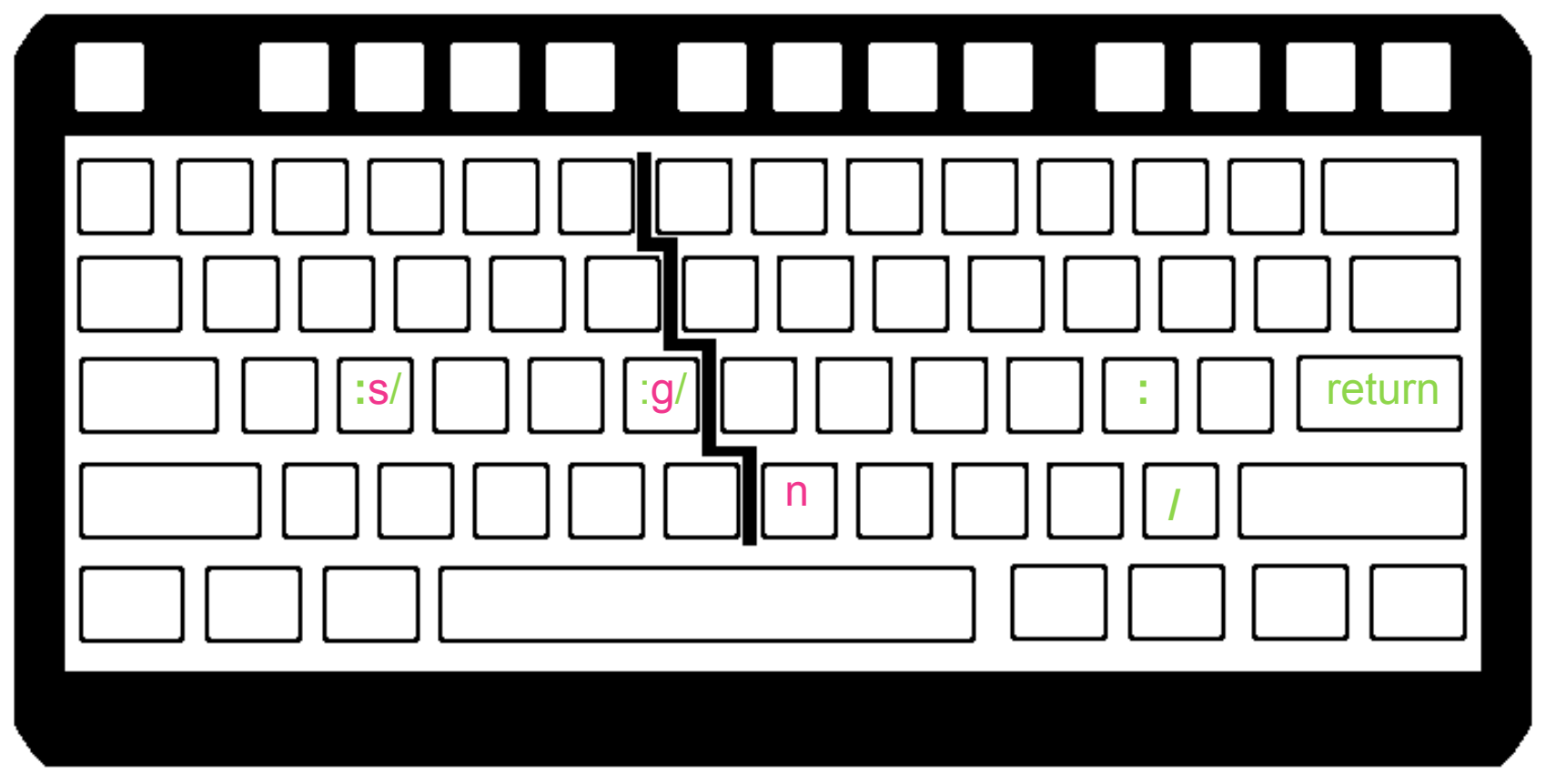
copy, paste, undo and redo



yy -- copy the line (yank)
yw -- copy the word (yank)
p -- paste the line or word after the cursor

u -- undo change
U -- undo all changes to the line
^R -- redo change
. -- repeat last command

search and replace

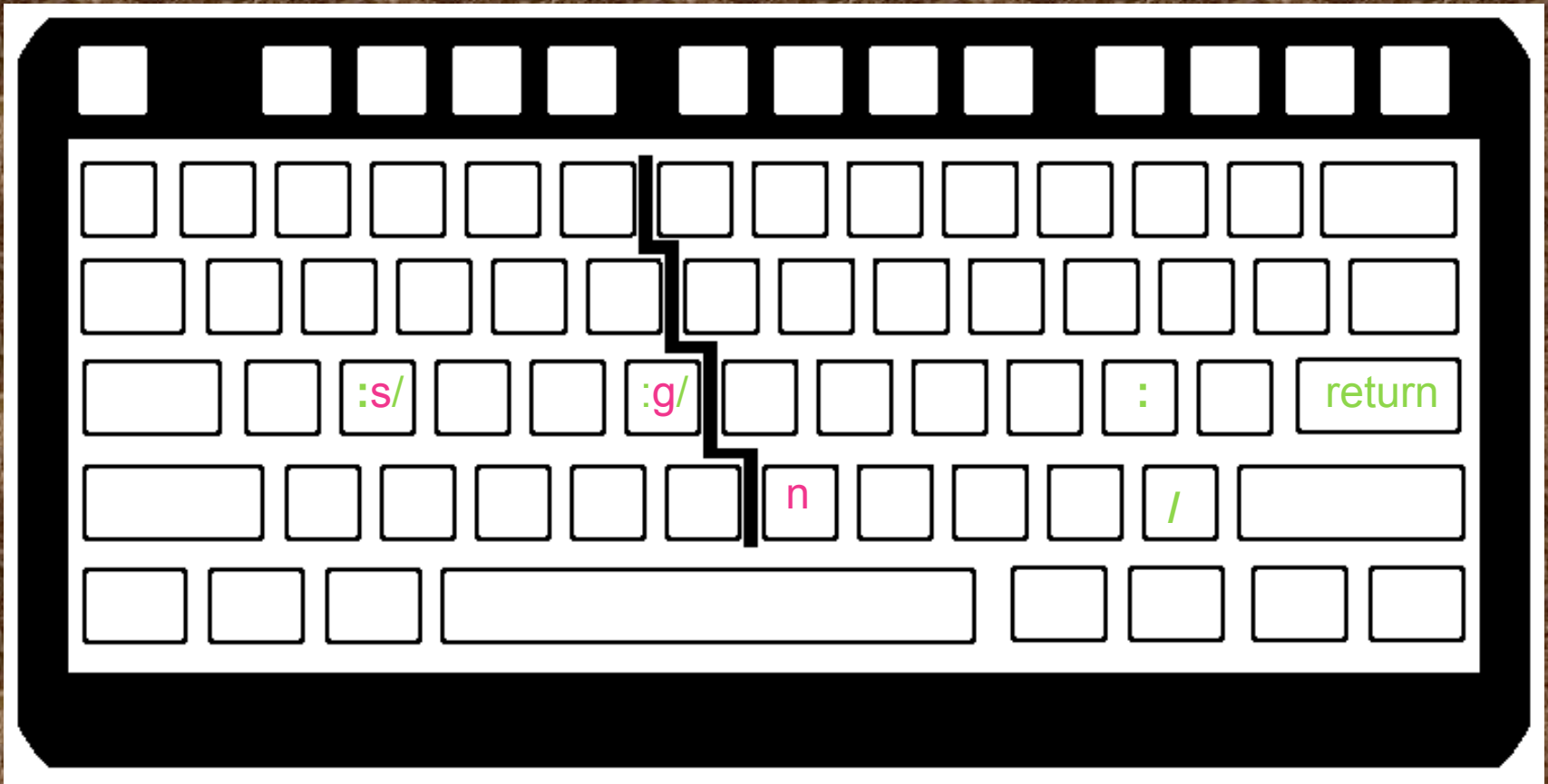


/[word or words] -- search for the next instance of the word or words

n -- go to next instance of word or words

Uses regular expressions for the pattern matching.

search and replace



:s/[old]/[new] -- substitute old with new string; cursor is on old string

:gs/[old]/[new]/g -- globally find old, substitute all old with new

Uses regular expressions for the pattern matching.

When you want to search for a string of text and replace it with another string of text, you can use the syntax

:[range]s/search/replace/[g][c][i].

Range can be -
n,m for lines n to m
n,$ for lines n to last (1,$ for whole file)
or % for whole file

g – global in the line, c – confirmation, i – ignore case.

:[range]s/search/replace/[g][c][i].

The range, global and confirm fields are optional.

if you just run

:s/search/replace/

it will search only the current line and match/replace <u>only the first occurrence</u> of the match.

Ex with range specified, plus "g" at end is for global (on line) replace (all matches on line, not just first)

:8,10 s/search/replace/g

If you want to search an entire file, and replace all matches, you can use % to indicate the whole file as the range, and g for all matches on each line:

:%s/search/replace/g

To match a word

/ word / is a good attempt at a match, but does not get the word when followed by punctuation for example.

\<word\> (have to escape the < and >, don't need the /'s) now matches the word.

Say you want to find a string and append something to it.

Try this.

s/run/&s/

Will match <u>run</u> and produce <u>runs</u>.

The & represents the match.

Say you want to find a string and append something to it.

Try this.

\1 is first match, \2 is second.

So this will also do it.

:%s/\(run\)/\1s/

You need the (), which need to be escaped, around <u>run</u>

The \(. . . \) delimiters are used to inform the editor that the text that matches the regular expression inside the parentheses is to be remembered for later use (in the \1).

Compress multiple occurrences of blank lines into a single blank line

```
:v/./,/./-j
```

Use :helpgrep '\/,\/' *.txt for an explanation.

I'll break down this incredible collapse-multiple-blank-lines command for everyone, now that I finally figured out how it works.
First, however, I'll rewrite it this way to illustrate that some of those slashes have totally different meaning than others:

```
:v_._,/./-1join
```

Note that to delimit expressions like these, just about any symbol can be used in place of the typical slashes... in this case, I used underscores. What we have is an inverse search (:v, same as :g!) for a dot ('.') which means anything except a newline. So this will match empty lines and proceed to execute [command] on each of them.

```
:v_._[command]
```

The remaining [command] is this, which is a fancy join command, abbreviated earlier as just 'j'.

```
,/./-1join
```

The comma tells it to work with a range of lines:

```
:help :,
```

With nothing before the comma, the range begins at the cursor, which is where that first blank line was. The end of the range is specified by a search, which to my knowledge actually does require slashes. The slash and dot mean to search for anything (again), which matches the nearest non-empty line and offsets by {offset} lines.

```
/./{offset}
```

The {offset} here is -1, meaning one line above. In the original command we just saw a minus sign, to which vim assumes a count of 1 by default, so it did the same thing as how I've rewritten it, but simply with one character fewer to type.

```
/./-1
```

There is a caveat about join that makes this trick possible. If you specify a range of only one line to "join", it will do nothing. For example, this command tells vim to join into one line all lines from 5 to 5, which does nothing:

```
:5,5join
```

In this case, any time you have more than one empty line (the case of interest), the join will see a range greater than one and join them together. For all single empty lines, join will leave it alone.

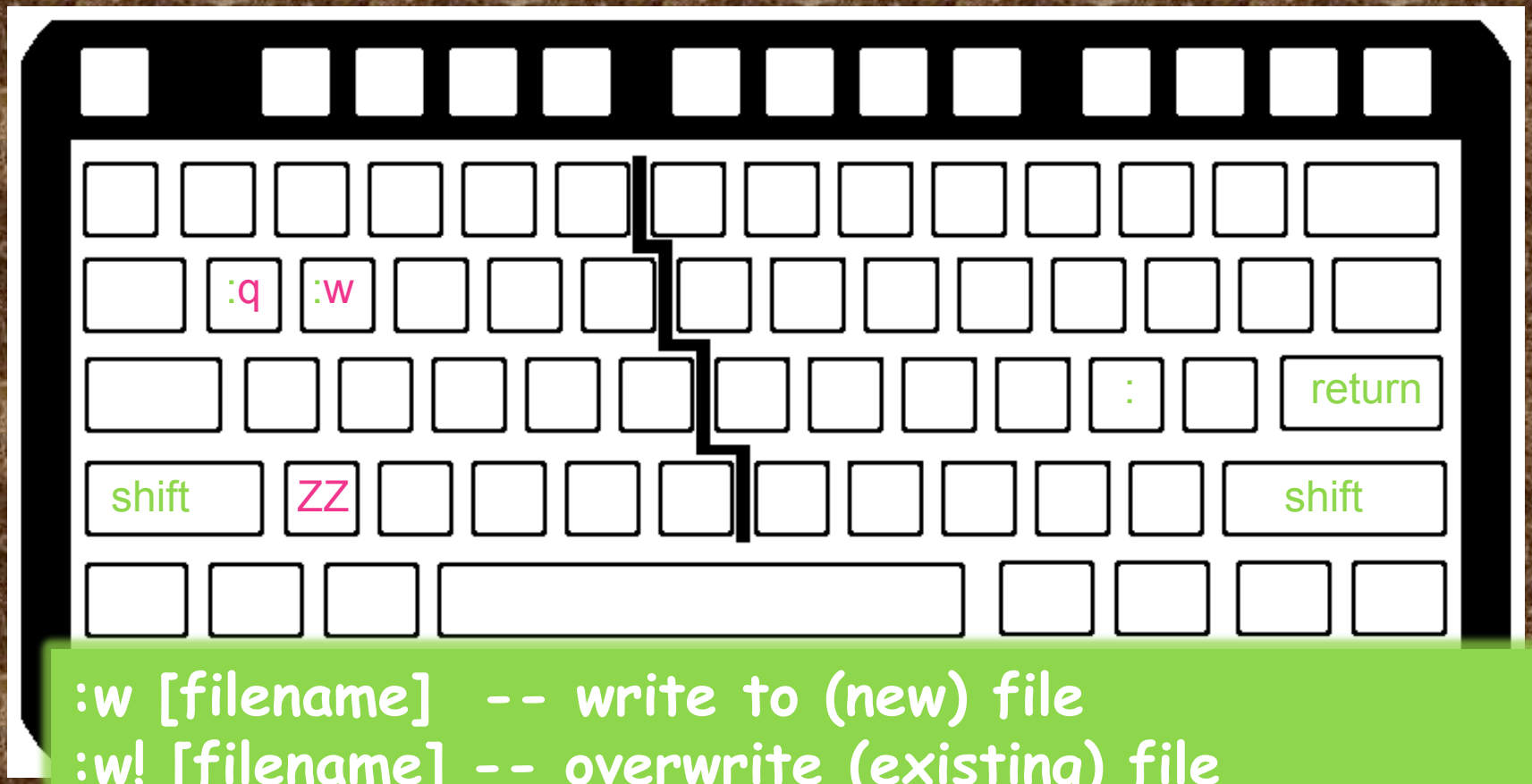
There's no good way use a delete command with :v/./ because you have to delete one line for every empty line you find. Join turned out to be the answer.
This command only merges truly "empty" lines... if any lines contain spaces and/or tabs, they will not be collapsed. To make sure you kill those lines, try this:

```
:v/^[^ \t]\+$/,/^[^ \t]\+$/-j
```

Or, to just clean such lines up first,

```
:%s/^[ \t]\+$//g
```

saving and exiting vi/vim



:w [filename] -- write to (new) file
:w! [filename] -- overwrite (existing) file
:wq -- write and quit
:q --- quit
:q! --- quit without saving
ZZ -- write and quit

other useful features in vi/vim

:![unix command] -- allows you to run standard unix commands without exiting vim; very useful with GMT

Example

```
:!ls *.SAC
```

In command mode the ":" tells vi that we are doing a command from the ed/edit/sed command list.

If you look in the man pages for vi or vim, it will refer you to them for the command descriptions.

other useful features in vi/vim

:set hlsearch -- will highlight all instances of a string when using */[word]* to search

>aB -- indent the block/loop defined by {} when cursor is located within the block in question

:sp -- split the screen
^WW -- use to move from one split screen to the next; useful when writing subroutines within the same file

other useful features in vi/vim

: set number or :set nonumber -- turn line numbers on/off

:X -- jump to line number X example *:1*

There are whole books on vi and vim. We are just scratching the surface.

Once you learn one of these, you tend to use them instead of the “word” like editors.

Manipulating & Printing Files

BASICS OF THE UNIX/LINUX ENVIRONMENT

CERI Printers

Long Building

3892_grad -- B & W printer in grad area

3892_hpcolor -- Color printer in grad area

3892_hpxlfp -- Poster printer in grad area

House 3

3876_langston -- B & W printer near Steve's office

3876_hpcolor -- Color printer near conference room

3876_grad – B & W duplex printer in Sun lab

CERI Printers (Continued)

House 2

3890_hpcolor – Color printer in copier room

3890_copy – B & W printer in copier room

House 1

3904_tek -- Color printer

3904_hallway -- B & W printer

Printing Commands

lpr: submit files for printing

```
%lpr -P3892_grad file.txt
```

Printing Commands

lpq: show printer queue status useful to find out if other jobs are before yours.

```
%lpq -P3892_grad
3892_grad is ready and printing
Rank     Owner    Job      File(s)       Total Size
active   hdeshon 146      junk.pdf    108544 bytes
```

Identifies the job.

lprm: cancel print job (by number)

```
%lprm -P3892_grad 146
```

lpstat: printer status information useful for finding out printer names on Macs, which are not necessarily the same as on the Unix system

```
%lpstat -a
_3876langston accepting requests since Wed Aug 27 13:11:36 2008
hp_color_LaserJet_4600 accepting requests since Mon Aug  4 11:50:47 2008
```

Some more useful commands

BASICS OF THE UNIX/LINUX ENVIRONMENT

Additional useful commands

wc: word count

```
%wc suma1.hrdpicks
37753  253998 3561084 suma1.hrdpicks
```

Reports number of lines, words (separator=space), and characters in the file.

Additional useful commands

cmp: compare files

```
alpaca.ceri.memphis.edu496:> cmp hw1.txt hw1a.txt
hw1.txt hw1a.txt differ: char 175, line 12
alpaca.ceri.memphis.edu497:>
```

No output if the same, else reports byte and line numbers at which the first difference occurred (starts at 1).

Additional useful commands

diff: show differences between two files

```
alpaca.ceri.memphis.edu498:> diff hw1.txt hw1a.txt
12c12
< 2) [2] Create a directory in your account for this course -
you might call it something like ESCI7205.
---
> 2) [2*] Create a directory in your account for this course -
you might call it something like ESCI7205.
14c14
```

Sometimes useful (if files completely different is mess). Less than sign, suck, for file 1, greater than sign, spit, for file 2. (if have extra lines, will re-synch, afterwards.)

Additional useful commands

sort: alphabetical or numeric sort function

sort alphabetically

```
alpaca.ceri.memphis.edu525:> more samgps.dat
PELD -33.14318 -70.67493 CAP [5] 1993 1997 1998 1999 2002
CHILE OKRT
COGO -31.15343 -70.97526 CAP [3] 1993 1996 2002 CHILE OKRT
MORA -30.20823 -70.78971 CAP [3] 1993 1996 2002 CHILE OKRT
MOR2 -30.20823 -70.78971 CAP [?] CHILE OKRT
TOFO -29.45939 -71.23842 CAP [4] 1993 1996 2001 2002 CHILE
OKRT
SILA -29.24037 -70.74956 CAP [3] 1993 1996 2002 CHILE OKRT
HUAS -28.47848 -71.22235 CAP [3] 1993 1996 2002 CHILE OKRT
. . .
alpaca.ceri.memphis.edu526:> sort samgps.dat
ABAC -24.433 -66.217 SAGA [-] ARGENTINA NORT
ABEL -25.667 -65.483 SAGA [-] ARGENTINA NORT
ACOL -30.78337 -66.21338 CAP [3] 1993 1997 2000 ARGENTINA OKRT
ACPM -33.447181 -70.537434 CAP2 [c] continuous (2005-) CHILE
ADLS -26.08449 -67.4191 CAP [2] 1993 1997 ARGENTINA OKRT
AGAL -24.317 -66.467 SAGA [-] ARGENTINA NORT
. . .
```

```
alpaca.ceri.memphis.edu506:> more flong.dat
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

sort alphabetically

```
alpaca.ceri.memphis.edu513:> sort flong.dat | head -4
1
10
11
12
```

Sort numerically

```
alpaca.ceri.memphis.edu514:> sort -n flong.dat | head -4
1
2
3
4
```

Sort numerically on second column

```
alpaca.ceri.memphis.edu530:> sort -n -k 2 samgps.dat | head -5
W01A -87.41565 -149.43328 WAGN [2] 2002 2005 OKRT
W01B -87.41518 -149.44311 WAGN [2] 2002 2005 OKRT
W02A -85.61192 -68.55633 WAGN [3] 2002 2005 2008 OKRT
W02B -85.61185 -68.55546 WAGN [2] 2002 2005 OKRT
W13B -83.12942 159.50532 WAGN [1] 2003 OKRT
```

Read the man page to see what else it will do.

```
NAME
     sort - sort, merge, or sequence check text files

SYNOPSIS
     /usr/bin/sort  [-bcdfimMnru]  [-k keydef]   [-o output]
[-S kmem]  [-t char] [-T directory] [ -y  [kmem]] [-z recsz]
[+pos1  [-pos2]] [file...]

     /usr/xpg4/bin/sort [-bcdfimMnru] [-k keydef] [-o output]
[-S kmem]  [-t char] [-T directory] [ -y  [kmem]] [-z recsz]
[+pos1  [-pos2]] [file...]
```

From now one, you will be expected to read the man pages for all the commands we have used or will use to see how to use them and what they will do.

Time

cal: displays a calendar

Default is current month

Will also display the year

Good way to figure out day of year (often incorrectly called julian day) using the -j flag

```
smalleys-imac-2:geolfigs smalley$ cal
    September 2009
Su Mo Tu We Th Fr Sa
        1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30

smalleys-imac-2:geolfigs smalley$ cal -j
       September 2009
 Su  Mo  Tu  We  Th  Fr  Sa
         244 245 246 247 248
249 250 251 252 253 254 255
256 257 258 259 260 261 262
263 264 265 266 267 268 269
270 271 272 273
```

Time

date: displays date and time

```
%date
Wed Aug 27 17:12:01 CDT 2008

%date -u -r 10
Thu Jan  1 00:00:10 UTC 1970
```

Basic Math

bc: basic math calculator

+, -, *, /, %, ^, sqrt

also test Boolean expressions and >,<,!=, etc.

quit or CNTL-D to exit

expr: evaluate the expression

more powerful, command line calculator for integer math and string comparison

units: unit conversion

Job Control

top: lists all processes currently running

ps: process status, another way to display process identification numbers (PID)

```
alpaca.ceri.memphis.edu585:> ps -aef
     UID   PID  PPID  C   STIME TTY      TIME CMD
    root     0     0  0   Jun 13 ?       0:04 sched
    root     1     0  0   Jun 13 ?       0:10 /etc/init –
. . .
rsmalley  9790  9580  1 23:17:45 pts/12 0:00 ps -aef
rsmalley  9578  9575  1 18:50:33 ?       0:04 /usr/lib/ssh/sshd
```

kill: allows you to hard kill processes by PID

```
%kill -9 9578
```

CNTL-Z: suspends the current job (use to end man program).

bg: resume job but runs it in the background (initially set on the command line by adding an & to the end of the command/ script).

fg: resume job and runs it in the foreground.

jobs: lists all jobs running in the background, including their PIDs.

Finding/Searching

find: search for files

Say I want to see if I have a file "Volcanoes.dat" in my "dem" and "bin" subdirectories.

```
alpaca.ceri.memphis.edu513:> find ~/dem -name Volcanoes.dat
/gaia/home/rsmalley/dem/Volcanoes.dat
alpaca.ceri.memphis.edu514:> find ~/bin -name "*olcanoes*"
alpaca.ceri.memphis.edu515:>
```

OK, that's nice, but not yet too useful (I could have cd'd into dem and done an ls, no need for a new command).

Finding/Searching

find: search for files

To make this really useful, we need a way to search for patterns in the filenames (or within files).

Enter Regular Expressions.

A regular expression is a set of characters that specify a pattern.

So now we are going to have two kinds of special characters, or metacharacters.

Those that mean something special to the shell (such as the "$" on a shell or environment variable or the "/" in a path).

Those that are used to specify a pattern.

And will need a way to "turn off", or escape, the special meaning.

Say I want to look for all files that start with a "v" or "V", and have any "extension" (the ".dat", part of the file name).

```
alpaca.ceri.memphis.edu514:> find ~/dem -name *olcanoes*
find: No match.
```

This did not work for some reason.

The find command is not "seeing" the wildcard "*". (The shell got hold of it first and did something with it.)

We have to "escape" the shell's interpretation of the "*", so it gets passed to find to be used as a wildcard (regular expression) there.

```
alpaca.ceri.memphis.edu515:> find ~/dem -name \*olcanoes\*
/gaia/home/rsmalley/dem/Volcanoes.dat
/gaia/home/rsmalley/dem/volcanoes
/gaia/home/rsmalley/dem/volcanoes.f
alpaca.ceri.memphis.edu516:> find ~/dem -name '*olcanoes*'
/gaia/home/rsmalley/dem/Volcanoes.dat
/gaia/home/rsmalley/dem/volcanoes
/gaia/home/rsmalley/dem/volcanoes.f
alpaca.ceri.memphis.edu517:> find ~/dem -name "*olcanoes*"
/gaia/home/rsmalley/dem/Volcanoes.dat
/gaia/home/rsmalley/dem/volcanoes
/gaia/home/rsmalley/dem/volcanoes.f
alpaca.ceri.memphis.edu518:>
```

There are three ways to escape metacharacter interpretation.

Backslash "\", escapes the next character from interpretation [the first time \ is encountered], i.e. the next character is treated as a regular character.

```
\*olcanoes\*
'*olcanoes*'
"*olcanoes*"
```

Works for all programs (the shell is just another program).

```
\*olcanoes\*
```

So the splat is not used as a wildcard by the shell (all the files in the directory), the first program to encounter it, and it is passed as a * to the program find where it is (finally) used as a wildcard (any combo of characters).

The backslash "\", is the strongest method to escape a character.

It works everywhere.

If you want to place text on two or more lines for readability, but the program expects one line, you need a line continuation character. Just use the backslash as the last character on the line:

```
% echo This could be \
a very \
long line\!
This could be a very long line!
%
```

This *escapes* or *quotes* the end of line (eol) character, so it no longer has a special meaning. (In the above example, the backslash before the exclamation point is necessary if you are using the C shell, which treats the "!" as a special character.)

Another example of the thought processes involved in taking advantage of the power of Unix.

What would you enter if you were looking for a file named "*olcanoes"?

(rhetorical question).

Next two methods.

Protect metacharacters from interpretation by the shell only.

Single quotes.

"quote", "escape", or "protect" everything inside them from the shell.

```
'*olcanoes*'
```

Next two methods.

Protect metacharacters from interpretation by the shell only.

Double quotes.

*"*olcanoes*"*

"quote", "escape", or "protect" everything inside them from the shell <u>except variables and backquoted expressions</u> (`)(we will get to that soon), which are expanded by the shell and replaced with their value.

Starts where we are (.), looks there and below.

```
alpaca.ceri.memphis.edu508:> find . -name cap_ice\* -print
./dem/cap_icezooms_.5v2.ps
. . . .
/from_midtown/dem/cap_ice_.5v2.ps
```

Don't need the "-print" anymore (but you may see it). In old days, found the files, but needed instructions on what to do with them (did not automatically send to standard out).

grep: search for a pattern inside files (or standard in).

(general regular expression,
general regular expression processor,
. . .)

highly useful and it is worth your time to sit down with the man page.

Simple examples

Find the string PELD in the file samgps.dat.

grep sends all lines in input (standard in, file [don't need redirect, but can use it], or pipe) that contain the string "PELD" to the standard out.

```
alpaca.ceri.memphis.edu533:> grep PELD samgps.dat
PELD -33.14318 -70.67493 CAP [5] 1993 1997 1998 1999 2002 CHILE OKRT
```

Takes standard Unix "regular expressions", of which we have seen a few.

This finds all the lines that start with a "P" ("^" is the metacharacter for the beginning of a line) and sends them to standard out.

```
alpaca.ceri.memphis.edu534:> grep ^P samgps.dat
PELD -33.14318 -70.67493 CAP [5] 1993 1997 1998 1999 2002 CHILE OKRT
PSTO -28.17157 -69.79377 CAP [3] 1993 1996 2002 CHILE OKRT
PNAZ -26.14822 -70.65368 CAP [3] 1993 1996 2001 CHILE OKRT
```

Finds all the lines with "ARGEN" and sends them to standard out.

```
alpaca.ceri.memphis.edu535:> grep ARGEN samgps.dat
TND2 -37.3 -59.2167 CAP|C1960 [0] ARGENTINA NORT dropped
ZAPX -38.82775 -70.02394 CAP|C1960 [?] ARGENTINA OKRT
```

Finds all the lines with "3 ARGEN" and sends them to standard out.

```
alpaca.ceri.memphis.edu510:> grep "3 ARGEN" samgps.dat
ZAPL -38.82775 -70.02394 CAP|C1960 [4] 1993 1997 1997 2003 ARGENTINA
BSON -42.01391 -71.20485 CAP|C1960 [3] 1993 1997 2003 ARGENTINA OKRT
```

I probably use grep every time I'm on a Unix system!

Regular Expressions

If you master regular expressions, searching for text becomes easy.

Regular expressions are accepted input for grep, sed, awk, perl and other unix commands.

Much like learning the shells, it is all about syntax & we'll just scratch the surface here.

Basic "regular expressions"

. : Matches a single character

```
alpaca.ceri.memphis.edu523:> grep P..D samgps.dat
PELD -33.14318 -70.67493 CAP [5] 1993 1997 1998 1999 2002 CHILE OKRT
MOAT -54.9572 -66.79024 SCARP|CAPP|TDF [4] 1998 2000 2007 ARGENTINA
```

Basic "regular expressions"

"*": Matches zero or more instances of the preceding character

```
alpaca.ceri.memphis.edu529:> grep 'AT1*' samgps.dat
SPAT -22.91555 -68.24654 CAP [3] 1993 1996 2001 CHILE OKRT
CATA -16.30061 -68.46202 CAP [2] 1993 1999 BOLIVIA OKRT
CUER -51.63393 -74.51458 CAP|GFZ|SCARP|TRANSFER|BOAT|SENH [3]
1994 1998 1999 CHILE OKRT
AT09 -30.27979 -68.53166 MATE|CAPP [3] 1997 1999 2004
ARGENTINA OKRT
AT10 -30.28933 -68.54643 MATE|CAPP [4] 1997 1999 2000 2004
ARGENTINA OKRT
AT11 -30.23073 -68.43688 MATE|CAPP [3] 1997 1999 2004
ARGENTINA OKRT
```

Basic "regular expressions"

"*": Matches zero or more instances of the preceding character

```
alpaca.ceri.memphis.edu529:> grep 'AT1*' samgps.dat
```

What were we looking for?

AT…, AT1…,AT11…,AT111…

Basic "regular expressions"

How do we look for anything and everything (zero or more instances of any character, the * wildcard from earlier).

The regular expression "*" does not do it.

Basic “regular expressions”

We have enough information.

All we have to do is think Unix.

Basic “regular expressions”

The “.” represents any character.

The “*” is any number of repetitions (including none or zero) of the preceding character.

Basic “regular expressions”

We now have all the pieces, we just have to put them together in Unix think.

Any guesses?

Basic "regular expressions"

How about

".*"

(dot, splat)

Any character plus zero or more repetitions of any character.

You can think of regular expressions as wildcards on steroids (or LSD).

^ : Represents the beginning of a line

```
alpaca.ceri.memphis.edu534:> cat samgps.dat
PELD -33.14318 -70.67493 CAP [5] 1993 1997 1998 1999 2002
CHILE OKRT
COGO -31.15343 -70.97526 CAP [3] 1993 1996 2002 CHILE OKRT
MORA -30.20823 -70.78971 CAP [3] 1993 1996 2002 CHILE OKRT
MOR2 -30.20823 -70.78971 CAP [?] CHILE OKRT
TOFO -29.45939 -71.23842 CAP [4] 1993 1996 2001 2002 CHILE
SILA -29.24037 -70.74956 CAP [3] 1993 1996 2002 CHILE OKRT
HUAS -28.47848 -71.22235 CAP [3] 1993 1996 2002 CHILE OKRT
PSTO -28.17157 -69.79377 CAP [3] 1993 1996 2002 CHILE OKRT
GRDA -27.71571 -69.55836 CAP [2] 1993 1996 CHILE OKRT
CALD -27.0827 -70.86208 CAP [5] 1993 1996 1999 2001 2002 CHILE
PNAZ -26.14822 -70.65368 CAP [3] 1993 1996 2001 CHILE OKRT
alpaca.ceri.memphis.edu532:> grep ^P samgps.dat
PELD -33.14318 -70.67493 CAP [5] 1993 1997 1998 1999 2002
CHILE OKRT
PSTO -28.17157 -69.79377 CAP [3] 1993 1996 2002 CHILE OKRT
PNAZ -26.14822 -70.65368 CAP [3] 1993 1996 2001 CHILE OKRT
PPST -20.97508 -68.83487 CAP [3] 1993 1996 2001 CHILE OKRT
PSAG -19.6023 -70.21962 CAP [3] 1993 1996 2001 CHILE OKR
```

$: Represents the end of the line

```
file example IND.pha
# 1918 9 22 9 54 49.29  -1.698   98.298   15.0  0.0 0 0
COC          274.71 1 P
MAN          346.71 1 P
ZKW          450.71 1 P
# 1926 6 28 3 23 26.82  -0.128  101.514   15.0  0.0 0 0
COC          303.18 1 P

%grep 'P_*$' IND.pha  | head -n2
COC          274.71 1 P_______
MAN          346.71 1 P_______

or

%grep -c 'P_*$' IND.pha    the -c flag counts matches
          831857
```

Unix think practice.

What represents an empty line?

What represents an empty line?

^$

\ : Escapes the following metacharacter

```
%grep  '\*'  suma.stations | head –n2
*AGD      +11.529000      +042.824000
*        AIS      -37.797000      +077.569000
```

[ ] : Matches members of the sets/ranges within the brackets

```
%grep '[DB]EQ' SUMA.NEW.loc
3478 2005 7 4 16 7 35.23   10.301   93.576   29.9  4.9
0.0 ehb DEQ Md
3480 2005 7 5 1 52 4.16    1.822   97.068   30.0  6.2
6.8 ehb BEQ Md
3481 2005 7 5 7 57 27.19    2.244   94.978   15.7  5.1
4.5 ehb DEQ Md
```

Non-printable characters

The following syntax works with a range of commands and programs that recognize regular expressions (sed, awk, perl, printf, etc)

\t : for a tab character
\r : for carriage return
\n : for line feed or new line.
\s : for a white space

awk (nawk):
[Aho, Kernighan, Weinberger]
new-awk

Powerful pattern-directed scanning and processing language.

So powerful that we will devote a full week to it in the future.

One of the most used Unix tools.