Grep, sed, awk

1 grep

Short for "global/regex/print" (a common sed command). Will print full line on match. Examples:

```
grep <option>* "<regex>" [<file>|<dir>]
grep blair /etc/passwd      # simple string search
grep -i blair users.txt     # case insensitive
grep '[0-9][0-9][0-9]' *    # 3-digit regex
grep -l 'd3\.' *.js         # print files, ¬ lines
grep -n 'pub' mycode.r      # line numbers
grep -B5 -A5 'pub' *.r      # print lines bef, aft
grep -v 'bob' /etc/passwd   # lines ¬ ∋ bob
ls -al | grep '^d'          # pipe: prnt all dirs
egrep 'apple|orange' *      # extended regex
grep -E 'apple|orange' *    # ibid
grep -rl 'null' .           # recursive file search
find . -type f -exec grep -il 'foo' {};
find+grep
```

2 sed

Deriving from "line-editor" **ed**, *hence its similarities to* **vi**, *which was influenced by the same.*

Invocation

```
sed <option>* <script> <text_file>
```

To specify [multiple] "instructions" inline:

```
sed '<instruct>' [-e '<instruct>']* <text_file>
```

Scripts combine instructions, each of the form:

```
<address>?<cmd>/<text>      ...or, for /s cmd:
<address>?s/<regex>/<substit_str>/<flags>
```

The most common command-line options are:

-e	editing instruction to follow
-f	script filename follows
-n	suppress automatic (every line) output

Syntax Examples

```
# A comment          # comment
s/.../a long\        # line ...
line, contin'd here  # ...continuation
s/into paren/(&)/    # backref entire match
s/\(\w\w*\) (\w\w*\) (\w\w*\) /\3 \2 \1/
                     # swap captured backrefe-
rences
y/abcdef/ABCDEF      #/transfm (capitalize)
/<H[1-6]>/{          # for html headers ...
  n                  # ...admit next line →
PS
  /^$/d              # ...and delete it if
blank
}                    # and done
$r closing.txt       # read in & append clo-
sing
/South$/w file       # ≡ grep '/South$/' >
file
100q                 # print 100 lines & quit
```

Note: **a**, **i**, and **c** *require new text to be on a new line, started with a line-continuing backslash \:*

```
/<Larry addr>/i\     # insert on ...
4700 Cross Court\    # ...multiple lines ...
French Lick, IN      # ...and finishing here
```

Commands (aka "Procedure")

Each script line is an "instruction" containing only 1 "command". Commands that admit only simple addresses (not ranges) are marked†*:*

s	substit.	d	delete
a	append†	i	insert†
c	change	l	list (ascii)
y	transform	p	print
=	print line #†	n	next line
r *f*	read† file *f*	w *f*	write file *f*
q	quit		

"Flags" for substitution command ∈:

<*n*>	replace *n*th occurrence ($n \in \mathbb{N}$)
g	makes Δs globally (on line)
p	print pattern-space contents
w <f>	write pattern-space to file *f*

Addressing (aka "Pattern")

Every **sed** *"instruction" starts with an "address," (aka a "pattern") indicating lines to which to apply the subsequent "command." A missing address defaults to "all lines." For the* s/ *command, a missing 'replace' regex defaults to the addressing regex. ∃ 3 means of addressing: a line number; a single regex; or a range of lines delimited by line #s or regex's or a combination thereof.*

```
d                    # no address: delete all lines!
1d                   # delete 1st (# indicated) line
$d                   # del last (special '$' addr)
line
/^$/d                # del blank lines (addr is a
regex)
50,$d                # range: del past line 50
1,/^$/d              # del to 1st empty (complex
rng)
/^#/!d               # ! inverts preceding address
```

Use braces {} to nest (AND) addresses and / or to apply multiple commands to the same address:

```
/<from>/,/<to>/{     top-level address range
  <instructn>*       each w/ its own [sub]addr
  <instructn>*       ...and on its own line
}                    } requires a separate line
```

Design Principles & Usage Patterns

∃ *various use-cases for* **sed**, *including:*

- *Multiple edits to the same file*
- *[Search/replace] Δs across a set of files*
- *Extract relevant content from a file(s)*
- *Manipulate a file for ephemeral use in a pipe*

Text Δs can be difficult; helpful practices include:

- *Start with checklist of desired changes, then write a script to make one & repeat*
- *Learn about your data, using* ***grep*** *first to see context in which target phrases appear*
- *Longer regexs reduce matches: ∃ an implied recall vs precision tradeoff*
- *Manually inspect Δs,* ***diff****ing input ag. out*
- *Iteratively test ag ever larger sample text*
- *It's also important that script doesn't work where you don't want it to*

Pattern- and Hold- spaces

These are temporary buffers, used while processing a single line of input text. **sed** *normally processes all script instructions for each line of text before proceeding to the next line of text. The "pattern space" stores the edited line as successive instructions change it; the "hold space" is for temporary storage. The following commands manipulate the PS, HS buffers or use those to maniuplate* **sed***'s control flow. Non-cap variants clobber exiting buffers; capitalized ones append only:*

h	PS→hold-space	H	PS→Hold-space
g	get: HS→PS	G	Get: HS→PS
N	Next line → PS	D	Del 1st PS line
P	Print 1st PS line	x	exchange HS, PS
b	branch to <lab>	t	test to <lab>

b *and* **t** *branch to the given "label" (defined by* : my_lab*), while keeping the current PS, HS;* **t***, conditionally upon success of previous* **/s** *cmd.*

3 awk

Evolved out of grep, sed; both of which evolved out of line editor **ed***.*

Invocation

```
awk '<instruction>*' files
awk -f <script> <files>
awk '<instruction>*' -
```

The most common options are:

-f	script filename to follow
-F	Δed Field separator to follow
-v	variable assignment follows

Example invocations:

```
awk '{ print }' in.file      # print each line
awk '{ print $1 }' my.txt    # print all 1st fields
awk -f myscript high=99 file #      setting vars
awk -f script -f /dir/lib    # link a lib (2nd -f)
shell> cat myscript          # script is all awk
shell> #!/usr/bin/awk .      #. direct "she-bang"
shell> myscript.sh file.in   # invoke; pass args
awk '...' $*                 # pass script params
```

Input from pipe using non-default separator:

```
ls | awk -F"\t" '{ print NR ": " $1 }'
```

Main Input Loop

Processing proceeds in a "main input loop," which iterates over all lines in all input files, running the script against each line. Each script "instruction" is comprised of a pattern and block of actions:

```
<pattern> { <action>* }
```

Patterns can be regex or line numbers, eg:

```
/^$/ { print "blank line!" }  # match regex
NR == 1 { print $0 }          # conditional pat-
tern
BEGIN { FS="," }              # special section
$5 ~/TX/ { print ... }        # regex matches
fld?
$5 ~myvar { print ... }       # ibid, using var
$5 !~/MA/ { print ... }       # ibid, negated
```

Each record is split into fields. While the main loop processes, variables **NR**, **NF**, *respectively contain index number of the current record,*

and total number of fields within this record. Eg, to count line number and swap first & last fields:

```
{ print NR, NF, $1 }      # use of fields
```

Expressions

Arithmetic operators:

```
  +    -    *    /    %    ~    **
```

Assignment Operators:

```
  ++   --        +=   -=       *=
  /=   %=   ~=   **=
```

Relational Operators:

```
  <    >    <=   >=   ==   !=   ~   !~
```

Boolean Operators:

```
  !!   &&   !
```

```
 /^$/ { print x += 1 }       # count lines

NF == 6 && NR > 1 { ...# compound expr
```

Variables

Basic variable assignment, manipulation:

```
x = 1                       # basic assignment
z = "Hello"                 # strings
y = x + 1                   # math
y += 1                      # assign operator
```

System variables:

ARGC	# arguments on command line
ARGV	Array ∋ cmd line args
CONVFMT	String conversion formatter
ENVIRON	Array ∋ all env vars
FILENAME	Current filename
FNR	NR relative to current file
NF	# fields in current record
NR	# of the current record
OFMT	Output format for numbers
OFS	Output field separator
ORS	Output record separator
RLENGTH	Length of str from match()
RS	Record separator
RSTART	1st pos from match()
SUBSEP	Separator char for array sub

Types

Strings can contain the following escape sequences: **\a**, **\b**, **\f**, **\n**, **\r**, **\t**, **\v**, **\<oct>**, **\x<hex>**, **\c**

Arrays are associative; iterate/test with for/if-in.

```
{ print str1 ": " str2 }    # string concat
grade["blair"]=99           # array assignemnt
split($1, name, " ")# 1st field → name[]
delete grade["blair"]       # edit array
```

Flow Control

Syntax follows C's. Brackets { } are required for multiple (\n-separated) lines of statements. **break** *and* **continue** *have normal meaning.* **next** *and* **exit** *goto next input line and exit the script.*

```
if(<cond>) { <statmt>* }
<expr> ? <statemt> : <statemt>
while( <cond> ) \n  <statemt>*
do \n <statmt>* \n while( <cond> )
for( <statmt>; <test>; <iterate>) { ... }
for( <var> in <array>) { <statmt>* }
```

Minimum working flow-control examples:

```
if(x==y) print x          # one-line if
avg > 65 ? "pass" : "fail"# ternary
while (i <= 4) { ... }    # line per statement
do { ... } while (x<=4)   # line per statment
for( i=1; i <= NF; i++) print i# iterate flds
for(itm in myarray) ...   # iterate array
```

Functions

Arithmetic functions:

cos	exp	int	log	sin
sqrt	atan2	rand	srand	

String functions:

gsub	index	length	split	sprintf
sub	substr	tolower	toupper	

System functions:

getline	read from stdin (see sed version)
system	execute system function
close	close pipes or files script opened
> |	output to file or pipe

Create and call user-defined functions:

```
function <str> ( <var># )name, params
  <statmt>*               # newline separated
  [return <var>]?         # optional return
}                         #
myfunc($1, 4, "mystr")# call myfunc
```

Idioms

Parse data given in "blocks" (records in lines):

```
BEGIN { FS = "\n"; RS = "" }
```

Report headers / footers using special sections:

```
BEGIN { print "Col 1", "\t", "Col 2" }
END { print "Total: ", sum }
```

Dynamic variable binding by shell:

```
awk '{ ... } dir=$(pwd) file1 ...
```

Confusing idiom, $1's mean different things:

```
shell> cat script
shell> #! /bin/bash
shell> awk '$1 == search' search=$1 file
```

printf Formatting

```
printf( <format> [, <argument>]* )
```

Where `<format>` *is a string containing arbitrary text, escaped special characters, and any number of* `<FS>`*s (format specifier), each comprised as follows:*

```
<FS> := %[-]?[0]?[0-9]*(.[0-9]*)?<spec>
```

[-]?	left justify option
[0]?	zero-fill option
[0-9]*	minimum print width
(.[0-9]*)?	decimal precision for floats
<spec>	mandatory, see below:

c	character	d	decimal
i	integer	e	float
E	float	f	float
g	shortest of **e** or **f**	G	shortest of **E** or **F**
o	unsigned octal	s	string
u	unsigned decimal	x	lowercase hex
X	uppercase heX	%	literal %

```
printf("|%10s|")        # |     hello|
printf("|%-10s|")       # |hello     |
printf("%d %s %d", 7, "ate", 9)# 7 ate 9
printf("%09.5f", π)     # 003.14159
printf("%*.*g\n", 5, 3, π)# dynamic format
```