Text Filter Notes

These programs modify their input in some way. They can easily find something in a long file, or to get only some columns from a table. Those covered in the *learn* tutorials have that listed in italics under 'Notes'.

- To avoid unclarity, sometimes ␣ is used for a space.
- A number in parentheses after the name, such as `date(1)` or `fgets(3)`, refers to a section of the on-line manual. To learn more about `time(2)`, run '`man 2 time`'. Some parts of these pages require consulting the manual.
- Except for inside regular expressions, things in brackets, [and], are optional. The notation '`ls [filenames]`' means that `ls(1)` will list information about files if names are given, otherwise it will list the current folder.

Print this sheet out and fill it in while doing the *learn* tutorials and when reviewing notes after class.

1 Regular Expressions

Many Unix programs use *regular expressions* (abbreviated REs) to specify strings of characters with less typing.
For example, if changing 'colour' to the American spelling 'color', searching for `colour` may miss instances at the beginning of a sentence, where it would have an uppercase 'C'. Search for `[Cc]olour` would find those instances.. The square brackets indicate any one of the characters listed, either an uppercase or lowercase 'c'.

1.1 Standard REs

These are recognised by almost all programs that use REs, and are the default for `grep(1)` and `sed(1)`.
This pattern language is not the same as that used for shell filename matches; be careful not to get them confused.

Character	Meaning	Notes
`.` (dot)	Any single character	*(grep 0.1e)* the shell uses ?
`[`*list*`]`	any single character on list	*(grep 0.1b)* can take ranges ([a-z])
`[^`*list*`]`	any single character not on list	*(grep 0.1d)*
`*`	zero or more of the match to the left	*(grep 0.1g)* includes zero!
`^`	beginning of line	*(grep 0.1c)*
`$`	end of line	*(grep 0.1f)*

Note: remember that `*`, which you may know as 'Kleene Star', means 'zero or more'; `grep 'q*' myfile.txt` will match every line in `myfile.txt`, even blank lines, because every line has zero or more `q`s on it.

1.2 Extended REs

These extensions can be used in `awk(1)` and `egrep(1)`, and can be used in `grep(1)` or `sed(1)` with the `-E` flag.

Character	Meaning	Notes
a`|`*b*	multi-character alternatives	
(*thing*)	form a group (treat *thing* as a unit)	see below
`?`	0 or 1 of thing to left	
`+`	1 or more of thing to left	
{`n`}	exactly `n` of the match to the left	
{`x`,`y`}	between `x` and `y` of the match to the left	

Note about groups:

`ab{1,3}` matches `ab`, `abb`, and `abbb`. The `{1,3}` applies only to the `b`.
`(ab){1,3}` matches `ab`, `abab`, and `ababab`. The `{1,3}` applies to the group `ab`.
`a|b{1,3}` matches `a`, `b`, `bb`, and `bbb`. The `{1,3}` applies only to the `b`.
`(a|b){1,3}` matches `aba`, `ab`, `baa`, and many more. The `{1,3}` applies to the group `a|b`.

1.3 Character Classes

Some groups of frequently-used characters can be referred to by name.
Each matches a single character when used in a list specified with brackets as in Section 1.

Note that list brackets are still necessary:

`grep '[:alpha:]'` will match only characters in the set {a, l, p, h, :} (includes colon, excludes everything else).

`grep '[[:alpha:]]'` will match all upper- and lower- case letters.

The names for the classes correspond to those listed in the manual page for `isalpha(3)`, except for the '`is`'. For example, in an RE, the class `[:upper:]` will match characters considered uppercase by the `isupper()` function.

The named character classes are mostly self-explanatory:

Class	Meaning	Notes
`alnum`		
`alpha`		
`blank`		
`cntrl`		
`digit`		
`graph`		
`lower`		
`print`		
`punct`		
`space`		more than space and tab
`upper`		
`xdigit`		

1.4 Special Escape Sequences

There are some shortcuts which can make expressions simpler. The most commonly used are:

Shortcut	Meaning	Notes
`\<`		
`\>`		
`\w`		same as `[_[:alnum:]]`
`\W`		same as `[^_[:alnum:]]`

1.5 Using Special Characters with the Shell

Some of the characters used in REs have special meaning to the shell; if you want to type `a*` as part of an expression, it would be annoying to have the shell put in all the filenames that start with `a`.

The best solution, usually, is to use single quotes: `grep 'a*' myfile.txt`. (*morefiles 1.1b*)

If you want to look for a single quote, you can switch to double quotes: `grep "case 'b':" *.c` will look in every C program for the string `case 'b':` (the shell will act on the unquoted star to do a filename match).

To type a control character, such as control-H (notation ^H), you first type ^V (as in *verbose*).
If you type **`grep ^V^H memo2.txt`**, it will run **`grep ^H memo2.txt`**, looking for `backspace` characters in the file `memo2.txt`. You can type a literal ^V by just typing it twice. (*morefiles 1.1c*)

2 The GREP Commands

grep(1) filters text lines according to patterns; it will include lines that match the pattern, and discard lines that do not match. *(morefiles 1.1b)*

The format for running `grep` is: `grep [option flags] pattern [filenames]`

If option flags are omitted, `grep` searches for the pattern as typed.

If filenames are omitted, `grep` reads from standard input (either the keyboard or a pipeline).

In addition to regular `grep`, there are two associated commands, `fgrep` and `egrep`.

fgrep(1) ('fast grep') does not use regular expressions; it looks for exactly the string you type. This saves a small amount of startup time needed to process the RE, and lets you easily search for asterisks, brackets, and other special characters. (Such symbols are common to many programming languages.)
`fgrep` is the same as `grep -F`.

egrep(1) ('extended grep') allows you to use extended regular expressions, which gives you more power to specify what you are searching for. (Extended regular expressions are described in Section 1.2 above.)
`egrep` is the same as `grep -E`.

Useful flags to `grep` include:

Flag	Description	Notes
`-E`	act like `egrep`; use Extended REs	
`-F`	act like `fgrep`; use no REs	
`-c`	count lines which would be printed, but don't print them	*grep 1.0a*
`-i`	ignore case	*grep 1.0b*
`-l` (ell)	list filenames that have lines that would be printed, but don't print them	
`-n`	print the line numbers along w/ the lines	*grep 1.0c*
`-o`	show only the part of the line that matches	
	suppress some error messages (do not warn about directories)	
`-v`	invert the search; print lines that do **not** match	*morefiles 6.1e*
`-w`	match words instead of just substrings	*grep 1.0d*

The **grep** commands also support longer flags, most usefully `--color=auto` . This is often set up as an alias to the shell so it's always on, by adding a line `alias grep='grep --color=auto'` to the file `.bashrc` in the user's home directory, or even in a system-wide startup file so everyone always has it.

Color highlighting can help debug regular expressions, because it makes clear exactly what is being matched. (See also Section 3.2.3.)

3 sed: Stream Editor

`sed(1)` accepts editor commands, and runs each command on every line of input.
The format for running `sed` is: `sed [option flags] command [filenames]`
If option flags are omitted, `sed` executes the commands and prints out every line.
If filenames are omitted, `sed` reads from standard input (either the keyboard or a pipeline).
One useful option flag is `-e`, which allows you to have more than one expression, telling sed to run multiple commands on each line of input.

3.1 sed Option Flags

Useful flags to `sed` include:

Flag	Description	Notes
`-E`	use Extended REs (see Section 1.2 above)	
`-e`		
`-f`		
`-i [ext]`		Best avoided.
`-n`		
`-s`		

3.2 sed Commands

Commands to `sed` have three sections, in the following format:

`[address]instruction[modifiers]` (Note: no spaces between sections.)

1. The `address` tells `sed` which line(s) this instruction applies to. If omitted, the instruction will be applied to every line of input. Addresses can be either line numbers or RE patterns (see Section 3.2.1).

2. The `instruction` is what you want to do to lines that are operated on. The most common instructions are:

Instruction	Description	Notes
`s`		see Section 3.2.2, below
`p`		mostly used with `-n`
`_____`	delete this line	
`q`		

   Multiple instructions can be given on one line if separated by semicolons (';').

3. The `modifiers` give options to affect how the instruction operates. If omitted, the default is used. (These modifiers are mostly used with the `s` instruction.)

Instruction	Description	Notes
`#`		
`i`		
`_____`	do all occurrences	
`_____`	do the opposite	

3.2.1 Addresses

An `address` can be a line number ($ means 'last line') or an RE pattern in slashes.

- If no address is given, the instruction is applied to all lines.

  `'s/foo/bar/'` substitute the first `foo` with `bar` on all lines.

  `'d'` delete all lines (not actually very useful).

- If one address is given, the instruction is applied only to the corresponding line(s).

  `'12s/foo/bar/'` substitute the first `foo` with `bar` on line 12 only.
  (`12` is the address; `s/foo/bar/` is the instruction.)

  `'$s/foo/bar/'` substitute the first `foo` with `bar` on the last line only.
  (`$` is the address, a special entry for 'last line'; `s/foo/bar/` is the instruction.)

  `'/^z/s/foo/bar/'` substitute the first `foo` with `bar` on any line that starts with `z`.
  (`/^z/` is the address; `s/foo/bar/` is the instruction. Pattern addresses go in slashes.)

  `'5q'` quit after line 5 (same as `head -5`)

  `'/^z/q'` quit after the first line that starts with z.

- If two addresses are given, the instruction is applied from the first line that matches the first address to the first line that matches the second address.

  `sed -n '11,16p'` - print lines 11-16.
  (The `-n` flag tells sed not to print the line unless the `p` instruction is present.)

  `sed -n '11,16p; 16q'` same as above, but faster.
  (Doesn't process the rest of the file.)

  `sed '/^n/,$d'` delete from the first line that starts with `n` to the last line.
  This can be done faster as `sed -n '/^n/!p ; /^n/q'`:
  suppress default printing, print all lines that don't start with `n`, quit on first line that does. (This is faster because it doesn't process the whole file.)

3.2.2 The s Instruction: Substitution

`sed` is often used to process many lines with substitution on each line. The usual format for the `s` instruction is:
`s/pattern/replacement/[modifiers]`

> **Note:** substitution is most often done using slashes as delimiters, but any character other than backslash ('\') or newline can be used after the `s`. This lets you easily substitute slashes: to replace '`a/b`' with '`a or b`', you can specify colons as delimiters: `'s:a/b:a or b:'` will work. `sed` will use the colons as delimeters, because that was the character after the `s`. This also helps avoid confusion when patterns have backslashes.
>
> This does not apply to address patterns: address patterns specifying lines must always be in slashes.

When making substitutions, parts of a pattern can be grouped with backslashed parents, and then the groups can be reference in the replacement string with backslashed numbers.
This: `sed 's:\(.*\), \(.*\):\2 \1:'` will turn `aaa, bbb` into `bbb aaa`.

The pattern is: '`\(.*\), \(.*\)`':		The replacement is:	
`\(.*\)`	first group ('`aaa`')	`\2`	second group ('`bbb`')
`,␣`	comma-space ('`,␣`')	`␣`	space ('`␣`')
`\(.*\)`	second group ('`bbb`')	`\1`	first group ('`aaa`')

The first group is everything before the comma and space (`aaa`), and the second group is everything after the comma and space (`bbb`).

In the replacement, `\1` is the first group, and `\2` is the second.

Note: the * is said to be 'greedy': takes the biggest chunk it can.

The above command: `sed 's:\(.*\), \(.*\):\2 \1:'` will turn `aaa, bbb, ccc` into `ccc aaa, bbb`.

The pattern is: '\(.*\), \(.*\)':		The replacement is:	
\(.*\)	first group ('aaa, bbb')	\2	second group ('ccc')
,␣	comma-space (',␣')	␣	space ('␣')
\(.*\)	second group ('ccc')	\1	first group ('aaa, bbb')

The first group is everything before the *last* comma and space (aaa, bbb), and the second group is everything after the *last* comma and space (ccc).

The first grouping gets all of `aaa, bbb` , since there's a comma-space after that, and the * is greedy.

You can repeat groupings: `sed 's:\(.*\) \(.*\):\2, \1 \2:'` will turn `James Bond` into `Bond, James Bond` . (Why?)

3.2.3 Debugging REs and SED Operations

Patterns can be tested by using groups:
`sed 's/\(b.*,\)/|\1|/'`
will replace the part of the line that matches the pattern with itself, putting a pipe character ('|') before and after the match. (This is similar to using `grep --color=auto`, as described in Section 2.)
In addition, SED's '=' and 'p' commands, can be used to debug SED operations:
`sed '= ; p ; s/\(.*\), \(.*\)/\2 \1/' names.txt` (Note: no `-n` flag.)
'=' causes the current line number to printed, and 'p' causes the line as it is to be printed, and then the 's' instruction is run. Without the `-n` flag, the result of the `s` flag is also printed. This helps to see exactly what happens.

4 Translating Characters with TR

The `tr` command takes two character sets on the command line, and replaces any character found in the first set with the corresponding character in the second. Characters not in the first set are passed through unfiltered.
`tr` understands ranges of characters, `tr A-Z a-z` will convert uppercase letters to lowercase letters, but digits, whitespace, and punctuation will be left unchanged. (*Set1* is all the uppercase letters, and *Set2* is all the lowercase letters, and they are replaced one-for-one.)
Useful flags to `tr` include:

Flag	Description	Notes
-c (or -C)	complements the set of characters in string	
-d	delete characters in the first set from the output	
-s	squeeze out duplicates	
-t	truncates set1	

Because `tr` operates on characters, not lines, the newline character does not get any special treatment. `tr '\n' '_'` will replace all newlines with underscores, and `tr ' ' '\n'` will replace all spaces with newlines.
`tr` understands the character classes from Section 1.3, which means all punctuation can be changed to spaces with a command such as: `tr '[:punct:]' '␣'`

The command `tr 'a-z' '_'` will turn all lowercase letters into underscores because if *Set2* is shorter than *Set1*, the last character in *Set2* is repeated to fill in the gap.

This means a typing error in *Set1* can lead to disaster. This: `tr 'A-z' 'a-z'` (note lowercase `z` in *Set1*) will convert all uppercase letters to to lowercase, and will convert all lowercase letters to `z`. *Set1* has more than 26 characters, so the last character of *Set2* is repeated as necessary to make up the difference.

5 Removing Duplicates with UNIQ

The `uniq` command removes pairwise duplicate lines from the input. That is, if the current line is identical to the last line printed, the the current line is omitted. Duplicate lines will appear if a different line is between them.

Useful flags to `uniq` include:

Flag	Description	Notes
`-c`	tells how many times a line was repeated	
`-d`	Only print lines that have adjacent duplicates.	
`-i`	allows case-insensitive comparisons	

6 Using SORT

↳ ascending

SORT sorts its input and then prints it out. By default, it uses alphabetical order and processes an entire line.

Specific columns ('keys') can be specified with the `-k` flag, in this format: **-k#[options][,#]**

where # is the column number this key starts at, options are below, and the second # is the column this key stops at, if you don't want to process an entire line.

There are a number of options, which can be used as flags for the entire line, or for specific keys.

Flag	Description	Notes
`-b`	ignores blanks @ start of line	
`-c`	Check file is sorted, but make no changes.	
`-d`	considers only blanks & alphanumeric characters	
`-f`	folds lower to upper case characters (ignore case)	
`-k`	Specify sort fields (see below)	
`-M`	Sort as months, Jan-Dec	
`-n`	sorts file w/ numeric data present inside	
`-r`	sorts input file in reverse order	
`-t`	use provided separator to identify the fields	
`-u`	sorts & removes duplicates	

6.1 Specifying Sort Keys

The `-k` option tells sort what to sort and how.

`-k2n`	Sort column 2 to the end of the line, as numbers.
`-k4r`	Sort column 4 to the end of the line, in reverse order.
`-k3nr,3`	Sort column 3 *only*, as numbers, in reverse order.
`-k1,2`	Sort columns 1 and 2 alphabetically, ignore everything else.

More than one key specification can appear on a line; later specifications are examined only if earlier ones match.

`-k3,3 -k2,2`	Sort column 3; if two lines are the same in column 3, sort those lines by column 2.
`-k3nr,3 -k1,1 -k2n,2`	Sort column 3 in decreasing order as numbers; if two lines are the same in column 3, sort those lines by column 1 alphabetically; if two lines are the same in column 3 and column 1, sort them by column 2 as increasing numbers.

7 Awk

AWK is a Turing-complete programming language, with sufficiently compact syntax that it is often used just from the command line. It borrows the SED syntax for matching lines, and uses the extended regular expression language as described in Section 1.

The most common flags to `awk` are:

Flag	Description	Notes
`-F`	tells awk what field separator to use	
`-f`	read awk program source from source-file	

AWK's line format is: `condition { commands }` . If the `condition` is true, the `commands` are executed. This gives an automatic `if-then` structure, with two shortcuts:

1. If the `condition` is ommitted, it defaults to `true`, and the `commands` are executed.
2. If the `commands` are omitted, it defaults to `print`.

7.1 AWK Variables

7.1.1 Automatic Variables

When a line is read, a number of variables are automatically created:

Variable	Meaning
`NR`	Number of records (input?) received so far
`NF`	Number of fields on this line
`$0`	The whole input line
`$1, $2, $3 ...`	1st, 2nd, 3rd fields of input line

Note: because the $ notation is used for some builtin variables (and in many programming languages such as Perl and shell scripting), it is a common mistake to put a $ before a builtin AWK variable that does not take one, as in `$NR`. If one of your scripts is not working, check for that.

7.1.2 Programmer Variables

Variables created by the programmer do not have to be declared. In any `commands` section, a variable can be assigned a value and it will be available from that point on, including other `commands` sections.

Variable names follow the usual conventions: alphanumeric characters and underscore, start with a letter.

Be sure to avoid reusing a name from one of the builtin variables.

7.2 AWK Conditions

Conditions can be either Boolean conditions, such as `NR <= 10` , or a pattern, using the extended REs as described in Section 1. As with SED, patterns must be in slashes, such as `/^k/`.

Conditions can refer to either builtin or programmer-created variables.

There is a special syntax for doing an RE match on a variable instead of the entire line:

There are two special conditions:

Condition	When it is true
`BEGIN`	when the first input hasn't been read yet
`END`	when all the input has been read

These are used for any needed setup before reading data, or to print summaries after all data is processed.

7.3 AWK Commands

Command blocks must be between curly braces. Multiple commands should be separated by semicolons. Many familiar structures are supported, including loops, if/then/else, and so on.

Here is a shell session showing an awk program to print odd numbers less than 10:

```
elvis> cat oddnums.awk
# Print odd numbers 1-10
BEGIN {
    for (i = 0; i < 10; i++) {
        if ( i % 2 == 1 ) {
            printf("%d ", i)
        }
    }
    printf("\n")
}
elvis> awk -f oddnums.awk
1 3 5 7 9
elvis>
```

Notes:

- Anything after an unquoted # sign is a comment.
- Curly braces are used for the *if* and *for* blocks.
- The program runs with no input, so the code is in the `BEGIN` block.
- AWK supports *printf()*.
- The `-f` flag is used to specify the program file on the command line.
- Since the two *printf()* calls are not followed by other commands, semicolons are not needed.

7.3.1 AWK Functions

AWK has a number of builtin functions which are always available. Among the more frequently used are:

Function	Operation
`int()`	truncates parameter to integer value
`sqrt()`	returns square root of parameters
`index()`	checks whether sub is substring of str or not (str, sub)
`length()`	returns length of string
`substr()`	returns substring of str @ index start of length l (str, start, l)
`split()`	splits str into fields by regex, fields loaded into arr (array) (str, arr, regex) ↳ if no regex, FS is used

8 HEAD and TAIL

HEAD shows the first 10 lines of a file, or any other number if the `-n` flag is used.

TAIL shows the last 10 lines of a file, or any other number if the `-n` flag is used.

These can be combined with a pipe to pick any line:

```
head -n 1000 myfile.txt | tail -n 1
```

will show line 1000 of `myfile.txt`.

As a shortcut, most versions accept `-#` instead of `-n #`, so the above could be:

```
head -1000 myfile.txt | tail -1
```

Other flags include:

Flag	Description	Notes
`-c`		
`-f`		TAIL only
`-q`		
`-v`		

9 Shell Command Equivalents

Many problems can be solved in more than one way. These three commands:

```
head myfile.txt    sed '10q' myfile.txt    awk 'NR <= 10' myfile.txt
```

all show the first ten lines of `myfile.txt`.

Here is an incomplete table of equivalents. Experiment at a shell to fill it in.

Awk	Grep	Sed
`awk '/`*pattern*`/'`	`grep '`*pattern*`'`	`sed -n '/`*pattern*`/p'`
	`grep -o '^[^␣]*'`	
`awk '{print $2}'`	*(no equivalent)*	
		`sed 's:\(.\).*:\1:'`
	(no equivalent)	`sed -n -e '20q' -e '11,20/p'`
`awk '/^$/{sum += 1} END {print sum}'`		
		`sed '/\*/d'`
	`grep -w foo`	
`awk 'length > 75'`		

 Id: UnixFilters.tex,v 1.14 2020/09/11 16:41:23 kilroy Exp kilroy