The grep, sed, and awk Commands

The `grep` Command and Regular Expressions

Normal invocation syntax:

```
grep PATTERN FILE1, ..., FILEn
```

If it finds a line with the PATTERN, it outputs it. So it is similar to `cat`, but more selective. In fact, if you use a pattern that matches every character, `grep` can behave identically to `cat`:

```
grep '.*' file
```

CHEATSHEET: Partial of `man grep`

```
NAME
       grep, egrep, fgrep, rgrep - print lines that match patterns

SYNOPSIS
       grep [OPTION...] PATTERNS [FILE...]
       grep [OPTION...] -e PATTERNS ... [FILE...]
       grep [OPTION...] -f PATTERN_FILE ... [FILE...]

DESCRIPTION
       grep  searches  for  PATTERNS  in  each  FILE.  PATTERNS is one or more
       patterns separated by newline characters, and  grep  prints  each  line
       that  matches a pattern.  Typically PATTERNS should be quoted when grep
       is used in a shell command.

       A FILE of “-” stands  for  standard  input.   If  no  FILE  is  given,
       recursive  searches  examine  the  working  directory, and nonrecursive
       searches read standard input.

       In addition, the variant programs egrep, fgrep and rgrep are  the  same
       as  grep -E,  grep -F,  and  grep -r, respectively.  These variants are
       deprecated, but are provided for backward compatibility.

OPTIONS
   Pattern Syntax
       -E, --extended-regexp
              Interpret PATTERNS as extended regular  expressions  (EREs,  see
              below).

   Matching Control
       -e PATTERNS, --regexp=PATTERNS
```

```
              Use  PATTERNS  as the patterns.  If this option is used multiple
              times or is combined with the -f (--file) option, search for all
              patterns  given.   This  option can be used to protect a pattern
              beginning with “-”.

       -f FILE, --file=FILE
              Obtain patterns from FILE, one per line.  If this option is used
              multiple  times  or  is  combined with the -e (--regexp) option,
              search for all patterns given.  The  empty  file  contains  zero
              patterns, and therefore matches nothing.
```

Basic Regular Expressions

`grep` actually uses **Basic regular expressions (BREs)**, a simpler form of the more familiar regex, **extended regular expressions (EREs)**. It also only matches against single lines at a time.

- Most printable characters like letters and numbers match themselves.
- Control characters like `*` need to be escaped in order to match themselves e.g. `\*` and `\\`.
- `.` as a control character matches every single character
- `^` matches the start of the line
- `$` matches the end of the line
- `[]` matches a single character but only those that are included in the set enclosed in the square brackets

CAUTION: the `*` is a **globbing pattern** to the shell, so it's interpreted with special meaning. Enclose it with *quoting* - don't let the *arguments themselves* be interpreted by the shell.

```
grep 'ab*c*' fo
```

Remember to Quote!

Quoting and little language commands like `grep` go hand-in-hand. Scripts are evaluated by the shell first before being shipped off as arguments to its child programs, so make sure you quote and/or escape everything keeping in mind how your initial string will be **resolved** as it makes its way through each program.

As described in single quoting, single quotes are often used for `grep` expressions because they preserve everything literally - what you see within the quotes is *exactly* what you're giving `grep`.

When your expression gets more complicated however, you will likely have to use double quotes to be able to interpolate regex fragments into the larger expression. In those cases, you'll likely see many instances of double backslashes `\\` since `\` is still a special character within double quotes which much be **escaped** to represent themselves by the time it reaches `grep`.

An example from a practice midterm:

```
atom='[a-zA-Z0-9]+'         # at least 1 alphanumeric character
string='\\"([^"\]|\\.)*\\"' # \"(something)\", where (something) is . OR neither "
```

```
or \
word="($atom|$string)"
words="$word(\\.$word)*"
grep -E "$words" | grep ' '
```

ASIDE: grep Edge Case

Example:

```
grep '*' foo
```

This is a weird case where `*` actually matches itself instead of being treated like a quantifier. grep has some ambiguous edge cases.

But *sometimes* grep yells at you:

```
$ grep '['
grep: Invalid regular expression
$ grep '\['
```

So basically, just use something well-defined.

Extended Regular Expressions

Historically there was another team that came up with alternate syntax to the familiar grep, called egrep. Nowadays you can use the `-E` flag to specify that the pattern is using the extended syntax instead:

```
grep -E PATTERN FILE1, ..., FILEn
```

It provides some features in addition to the BREs:

- The `+` quantifier to match "1 or more occurrences," so `P+` is equivalent to `PP*`.
- The range quantifier: `P{2,5}`. This would match the pattern `P` 2 to 5 times in a row, inclusive.
- Grouping with `()` and combining patterns with `|` to mean logical OR. For example, `(hello|foo)` matches *either* the entire string `hello` or the entire string `foo`.

Note that these features means the characters `+ { } ( | )` become **meta-characters** in EREs. In BREs, they would match themselves literally.

NOTE: `{}` is *also* a **globbing pattern** to the shell, where `{a,b,c,...}` expands to `a b c ...`, so remember to quote your regex!

Oh yeah and *apparently* in EREs, you can use the OR `|` operator outside of a group `()`:

```
hello|there
```

EXAMPLE: Write an ERE that only matches numbers between 0 and 255.

```
^(25[0-5]|2[0-4][0-9]|1[0-9]{2}|[1-9][0-9]|[0-9])$
```

TIP: Make sure to remember to include `^$`! That constrains the number of digits you can match, and more importantly, it ensures no partial matching.

Both BREs and EREs

- Bracket expressions `[]` are available in both standards and specify a **character set**.
- **Ranges** in bracket expressions like `[a-z]`. The `^` at the *start* negates the set e.g. `[^a-z]`. To include the literal `^` inside the character set, put it at the end.

Within a set, these characters take on special meanings, so to match them literally:

- To match the `]`, put it at the start of the set like `[]abc]`
- To match the `^`, put it at the end of the set like `[abc^]`
- To match the `* , put it at the end of the set like` [abc^*`

There are also **named character sets** in `grep`:

```
[[:alpha:]]     # match every alphabetic character
[[:alpha]$/]    # match every alphabetic character or $, /
[[:alpha]$/\]   # backslash isn't special inside a bracket
```

Common use cases of regular expressions in things like web applications include:

- Validating phone numbers
- Validating emails
- Extracting or processing such information

The `sed` Command

A command that is a like a generalization of `grep`, `head`, `tail`, etc.

Short for "stream editor". `sed` was designed to let you edit a file of ANY size because it does not use a buffer; rather it uses a **stream**. It's a "programmable incremental editor".

Using sed to emulate previous commands:

```
sed -n '1,10p'      # head -n 10
sed -n '10q'        # head -n 10
sed -n '$p'         # tail -n 1; can't generally because sed is incremental
```

`sed` itself is a little language that has supports its own scripting The invocation pattern is:

```
sed OPTIONS... [SCRIPT] [INPUTFILE...]
```

Stream Manipulation

The SCRIPT can be simple like:

```
sed 2q input.txt
sed 2,4d input.txt
```

- `q` example: **q**uit after the `2`nd line.
- `d` example: **d**elete lines `2` through `4` (basically excluding it from the output).

String Substitution

More commonly, the SCRIPT has the general structure below. By DEFAULT, *sed works at the line level*, replacing the FIRST occurrence of PATTERN with REPLACEMENT *per line*.

```
'[RANGE] s/PATTERN/REPLACEMENT/FLAGS'
```

- Optional RANGE narrows the operation to a range of lines, like `1,3`.
- By default, PATTERN is interpreted as BRE, but can be changed to ERE with the `-E` flag.
- If you used capturing groups in PATTERN, you can retrieve them in REPLACEMENT with `\1`, `\2`, etc.
- FLAGS are zero or more characters, each representing an individual modification. Some examples are:

Flag	Meaning
g	**g**lobal: replace all occurrences per line instead of the first
NUM	replace only the NUM-th (e.g. 1st, 2nd, etc.) occurrence per line
p	if the substitution was made, then print the new pattern space.

This is an example from lecture where you remove trailing whitespace (not sure if I copied it down correctly):

```
sed 's/[[:space:]]*$//; /^$/d'
```

Tutorial Examples

Dummy data in `toppings.txt` from https://www.youtube.com/watch?v=nXLnx8ncZyE:w

```
Pizza topping combos:
1. Spinach, Pepperoni, Pineapple
2. Pepperoni, Pineapple, Mushrooms
3. Bacon, Banana Peppers, Pineapple
4. Cheese, Pineapple
```

Replace the occurrences of `Pineapple` with `Feta` in the buffer, and then output the result.

```
sed 's/Pineapple/Feta/' toppings.txt
```

To modify the file **in-place**, just use the `-i` flag:

```
sed -i 's/Pineapple/Feta/' toppings.txt
```

Deleting every occurrence of a string is as simple as using the empty string for REPLACEMENT:

```
sed 's/Feta//' toppings.txt
```

Delimiters and Escaping

The `/` is just standard practice for the **delimiter**, which can be changed. The delimiter is automatically picked up from the character directly after `s`:

```
s|PATTERN|REPLACEMENT|
s PATTERN REPLACEMENT
s.PATTERN.REPLACEMENT.
```

This comes in handy when you would rather not have to escape characters, like the `/` itself. If you insist, you can use `\` to escape characters, like `\/`.

The `awk` Command

In a sense, even more general then `sed`. A scripting language designed to edit text. It's a programming language with variables, arrays, regular expressions, etc.

Here are some examples from lecture:

```
awk '{ x = $0; print "("x")"; }'
awk '{ if ($0 == x) print $0; x = $0 }'  # outputs duplicates
```

Basic Usage

Invocation pattern:

```
awk [OPTIONS] SCRIPT INPUTFILE
```

The SCRIPT has the general structure:

```
'{command args}'
```

awk also works at the line level. The SCRIPT body is assumed to operate individually on *each* line of the input. By default, awk sees each line as a sequence of **fields** delimited by spaces:

```
foo bar baz spam eggs
$1  $2  $3  $4   $5
```

Printing Select Fields

Dummy data `tmnt.txt` from https://www.youtube.com/watch?v=oPEnvuj9Qrl, a file with 4 lines and *3 fields each*:

```
leonardo blue leader
raphael red hothead
michelangelo orange party-animal
donatello purple geek
```

The `print` command prints the content of each line, so alone it is identical to `cat`:

```
awk '{print}' tmnt.txt
```

The `$NUM` syntax selects the NUM-th (starting from 1) field from each line. Paralleling regex capture group 0, `$0` selects the entire line.

```
$ awk '{print $1}' tmnt.txt
leonardo
raphael
michelangelo
donatello
```

You can select multiple fields by delimiting them with commas. The fields will be outputted with spaces in between them:

```
$ awk '{print $1,$3}' tmnt.txt
leonardo leader
raphael hothead
michelangelo party-animal
donatello geek
```

An example of pipeline input, outputting the permission flags next to the file names within the current directory (inspecting `ls -l`, the permissions and names happen to be on columns 1 and 9 respectively):

```
$ ls -l | awk '{print $1 $9}'
-rw-r--r-- lmao
drwxr-xr-x sub
-rw-r--r-- test.el
-rw-r--r-- test.sh
-rw-r--r-- toppings.txt
```

The special variable `$NF` is equivalent to the last field (NF stands for "number of fields") for each line:

```
$ awk '{print $NF}' tmnt.txt
leader
hothead
party-animal
geek
```

Attempting to use a number < 0 results in an error. You are allowed to use a field number greater than what a line has, in which case, the empty string is returned for that line.

Changing Field Delimiter

You can use the `-F` flag to change how awk delimits fields:

```
awk -F':' '{print $5}' /etc/passwd
```

CHEATSHEET: Partial of `man wc`

```
NAME
       wc - print newline, word, and byte counts for each file

SYNOPSIS
       wc [OPTION]... [FILE]...
```

```
       wc [OPTION]... --files0-from=F

DESCRIPTION
       Print  newline,  word, and byte counts for each FILE, and a total line if
       more than one FILE is specified.  A word is a non-zero-length sequence of
       characters delimited by white space.

       With no FILE, or when FILE is -, read standard input.

       The  options below may be used to select which counts are printed, always
       in the following order: newline,  word,  character,  byte,  maximum  line
       length.

       -c, --bytes
              print the byte counts

       -m, --chars
              print the character counts

       -l, --lines
              print the newline counts

       -w, --words
              print the word counts
```

CHEATSHEET: Partial of `man chmod`

```
NAME
       chmod - change file mode bits

SYNOPSIS
       chmod [OPTION]... MODE[,MODE]... FILE...
       chmod [OPTION]... OCTAL-MODE FILE...
```

Each comma-separated MODE is of the pattern `[augo][-+=][rwx]+`:

```
chmod a+x,u-r file1.txt file2.py
```

A single OCTAL-MODE string is 3 octal digits respectively representing the ugo permissions:

```
chmod 755 file3.sh file4.js
```

Assorted Commands

Some `ps` Flags

```
ps -efH
```

- `-e`: Select all processes. Identical to -A.
- `-f`: Do full-format listing.
- `-H`: Show process hierarchy (forest).

Creating File Links

```
# Hard link
ln TARGET LINKNAME
# Soft (symbolic) link
ln -s TARGET LINKNAME
```

Other Commands

Refer to your Assignment 1 for commands like `sort`, `tr`, `comm`, `seq`.

Refer to your Assignment 2 for commands like `shuf`.

Refer to lecture notes for any other commands.