Unix Tools
Lecture Notes

Kevin Scannell and John Paul Montano

Contents

Lecture 1.

Before anything else, you should install Linux on your own computer. When this course was first offered at Saint Louis University in 2014, the students were able to use accounts on a university Linux server (turing.slu.edu) via an ssh client, but most of them chose to run Linux as their day-to-day operating system during the semester. This is by far the best choice and the fastest way to learn. On a Windows computer you can do any of the following:

- Dual boot Linux/Windows
- Run Linux through a VM
- Cygwin

A Mac is, roughly-speaking, “Unix under the hood”. You can open the Terminal app and access a Unix command line, and do most of what we want in this course. Unfortunately, the default Unix tools that come on a Mac, while broadly compatible, are somewhat different from the ones on most Linux systems (Mac uses the so-called “BSD” tools, Linux distros use the “GNU” tools). Fortunately, it’s pretty easy to install the GNU tools on a Mac.

Unix is an operating system, created at Bell Labs (AT&T) in the late 60’s, early 70’s. It came out of an earlier effort (MIT, GE, Bell Labs) to develop a time-sharing operating system called

MULTICS. There were all sorts of problems with MULTICS, so the AT&T people pulled out of the joint effort, and created Unix. (Get it?). The main drivers were Ken Thompson and Dennis Ritchie.

The history since then is a long and sordid one not worth getting into now. Basically, through the 80's and 90's there were lots of commercial vendors selling a variety of "Unix-like" operating systems. In 1991, Linus Torvalds started working on an open source version of Unix that would run on commodity PCs. This is the version of Unix that would become "Linux".

Open source is a big part of the culture of the Linux world and the software development that goes on around it. Everything we use on a typical Linux system is free software. We'll talk later about doing development on github as a way of sharing your work with others and collaborating with people who are perhaps far away.

We'll start by going over some of the basics of how to use the command-line (navigating the file system, creating, copying, removing files, etc.) but will do so quickly so we can get to the real content of the course.

In all of the examples below, we'll write a $ to represent the command-line prompt. So, for example, the command for listing the files in the current directory looks like this:

```
$ ls
```

The important point for true beginners is that you only need to type the `ls` (and then press Enter or Return) because the dollar sign is just the prompt. It's also important to be aware that, in practice, this prompt will look different on different systems, depending on your configuration. For example, it's common that the prompt displays the name of the directory you're in, e.g.

```
data$ ls
```

or sometimes also your username and the name of the server:

```
kscanne@turing:~$ ls
```

The "command-line" or "terminal" is really a program that accepts input from the keyboard, interprets it as one or more commands, executes those commands, and displays any output. This program is called a "shell", and the dominant shell (and the one you'll use by default on Linux unless you try hard not to) is "bash". (This is another joke – it's the "bourne again shell" since it's an improvement of the earlier Bourne shell).

Commands you enter, like `ls`, are really the names of (usually little) programs. They typically have short, obfuscated names because we have to type them a lot – be thankful! Everything is case-sensitive.

Read-only navigation commands for exploring the filesystem:

```
$ ls
$ ls -l
```

The second example is typical; programs often take “command-line options”; these usually consist of a hyphen and a single letter, like the “`-l`” in the second example.

Show the present working directory:

```
$ pwd
```

Go into the subdirectory named “xyz”:

```
$ cd xyz
```

Go to my home directory:

```
$ cd
```

There are special shorthand filenames that you can use when navigating your filesystem. For example, “`..`” always refers to the parent directory of the current directory. So, this command takes you up to the parent directory:

```
$ cd ..
```

Similarly, “`.`” is shorthand for the current working directory. “`cd .`” therefore has no effect. But, the single dot is very useful in other contexts, e.g. copying things to the current directory.

A tilde `~` is shorthand for you home directory. So

```
$ cd ~
```

has the same effect as

```
$ cd
```

Most of the commands we’ve introduced here have lots and lots of complicated command-line options (that, honestly, are rarely used in practice). You can learn about these, or about how to use any other Unix tools, by reading the manual page for a particular command:

```
$ man ls
$ man pwd
```

etc.

Ok, now we enter the danger zone and learn some commands that actually change your filesystem. BEWARE: Unix is notoriously unforgiving. If you delete a file using the command-line tools, the file’s gone forever. No “recycle bins”. No tears. Toughen up.

To create a new subdirectory of the current directory:

```
$ mkdir data
```

And then go into the new directory:

```
$ cd data
```

You can create a new, empty file called `file.txt` as follows:

```
$ touch file.txt
```

If the file exists already, then `touch` just updates its “last modified” timestamp, which is sometimes useful.

Remove file X:

```
$ rm X
```

Copy file X to Y:

```
$ cp X Y
```

Move (rename) file A to B:

```
$ mv A B
```

Move file A into directory `dir` (the slash is optional):

```
$ mv A dir/
```

You can also create a new file by “redirecting” output of a command into a file; for example:

```
$ ls > file.txt
```

We'll do a lot of this in the lectures that follow.

Or, you can use a text editor.

```
$ vim
$ emacs
```

Pick one.

Sometimes the output of a shell command will contain many lines and scroll off the top of the screen; e.g. if you use `ls` as above, and a directory contains many files. A nice way to "slow down" the output is to pipe it through the Unix tool `more`:

```
$ ls | more
```

Pressing the spacebar moves one page down in the output. Pressing Enter moves down one line. And, "q" stops the output and returns you to the command line. There's a similar command `less` that has some additional functionality as well. In many of the examples throughout these lectures, you'll find it useful to pipe the output through `more` or `less` in this way, but we'll leave that up to you in specific examples.

Finally, a couple of indispensable tricks when working at the command line:

First, you can use the up- and down-arrows to navigate backward and forward through the history of commands you've typed into the current session. You'll use the up-arrow extensively while experimenting with the examples in these notes, and in constructing commands of your own.

Second, in many contexts one is able to use the tab key to auto-complete commands or filenames at the shell prompt. For example, if there is a file `data.txt` in the current directory, and no other files beginning with the letter "d", then you can simply type:

```
$ touch d
```

and then press tab, and the rest of the filename will autocomplete.

The one other thing you'll need to work through this course is a set of example files, which is available here:
https://kevinscannell.com/files/examples.zip

You can download and open the zip file all from the command line:

```
$ wget https://kevinscannell.com/files/examples.zip
$ unzip examples.zip
$ cd examples
$ ls
```

Let's start with a cool example that will illustrate the power of the techniques we'll learn in this course. If you've ever been to an American Super Bowl party, you may have participated in a "Super Bowl Square" pool. This is a 10x10 grid, with rows and columns labelled with the digits 0...9. Everyone pays an entry fee and gets to choose some number of squares in the grid, usually by writing their initials in the squares. At the end of the game, you take the last digit of the two teams' scores and that determines the row/column of the winning square. You too can win Super Bowl Squares more often than chance, THANKS TO THE UNIX TOOLS. Let us explain.

You'll find a file called `nfl96.dat.txt` in the examples folder (original source: http://www.amstat.org/publications/jse/datasets/nfl96.dat.txt) that contains American football scores for the 1996 season. Here's how you can quickly find the best squares to choose at your next Super Bowl party:

```
$ cat nfl96.dat.txt | sed 's/^ *//' | sed 's/^[^ ]* *//' |
  sed 's/  */\t/g' | cut -f2,4 | sed 's/^[0-9]*\([0-9][^0-9]\)/\1/' |
  sed 's/\([^0-9]\)[0-9]*\([0-9]\)$/\1\2/' | sort | uniq -c |
  sort -k1,1 -r -n
```

This may look complicated, but it's really not. For now, don't worry about the details; what's important is the philosophy of the Unix tools and programming using pipelines. We came up with the code above one step at a time:

```
$ cat nfl96.dat.txt
```

This just dumps the original file to the screen

```
$ cat nfl96.dat.txt | sed 's/^ *//'
```

This dumps the same file, but modified slightly (removes any leading spaces from each line). The vertical bar is a "pipe"; the idea is that the output of the first program (everything to the left of the pipe) is sent through the pipe and becomes the input to the next program. The power comes when you string together many programs with pipes:

```
$ cat nfl96.dat.txt | sed 's/^ *//' | sed 's/^[^ ]* *//'
```

This further removes the date from the beginning of each line. And so on…

After executing the full command, the first few lines of your piped result should look as follows:

```
     12 0       4
     11 0       7
     10 4       7
      9 3       7
...
```

The first line of our piped result (i.e., "`12 0        4`") indicates that the final digit of each of the two football teams' scores (0 and 4 in this case) appear most often. 12 times, to be exact. The second line of our piped result (i.e., "`11 0        7`") indicates that the football teams' respective final digits, 0 and 7, appear next most frequently. A total of 11 times, in this case. And, so on.

We recommend reproducing the long command above one step and a time, and trying to figure out what each piece of the pipeline is doing. Don't forget to use the up-arrow to minimize retyping. A single shell command made up of a long pipeline like the one above is called a "one-liner".

The philosophy is that you have some input data (the original file with the scores) and you'd like to modify it into some output data (a sorted list of the best squares for the Super Bowl pool). You can get from the input to the desired output one small step at a time. The Unix tools like `sed`, `cut`, `sort`, and `uniq` in this example make it easy to do this.

Get used to experimenting. A good bit of text-processing with "real world" data involves some trial-and-error. A typical process is to add a command to the end of a pipeline and examine the output. If it worked the way you expected, you can press the up-arrow at the shell prompt to retrieve the command you just used, and add more commands if needed. If it didn't work the way you wanted, you can again use the up-arrow and tweak the command until you get it right.

Lecture 2.

Recall that the kind of text-processing we did last time is called "pipelining" or "pipeline programming". This is the fundamental idea in this whole course. Once you're good at this, you'll tend to think about all programming problems in the same way.

Principles:

- Everything we do is processing a stream of plain text.
- We initiate a stream somehow, often from an existing file.
- Streams are (almost) always processed *line-by-line.* This is critically important to understand. All of the so-called Unix Tools are designed to work on one line of text at a time.
- The stream is "piped" through one or more Unix Tools (sometimes called "filters", but we prefer the more refined terminology below) until the stream looks the way we want it to.

Pipelines have been part of Unix almost since the beginning. There's a great interview with Doug McIlroy, the head of the department at Bell Labs where Unix started, which explains how pipelines came to be, and how their implementation was followed (the very next day) by an "orgy of one-liners." We've reproduced an excerpt at the end of these notes in an appendix since it echoes very nicely some of the general Unix philosophies that we're trying to convey in these lectures.

There are three kinds of tools, roughly speaking:

- Mappers, which modify lines in the stream in some way
- Filters, which keep only certain lines of the stream
- Reducers, which "accumulate" or "collapse" the stream to a result

If you know something about Google's "Map-Reduce" architecture, or if you've studied the bits about stream processing in SICP, you'll understand the choice of terminology here.

To get started, we'll need a plain text file to work with. There's a file `en.txt` in the examples folder which contains a list of about 100,000 English words. There appears one English word on each line of the file. (A slightly modified version of the file can be found here: http://www-01.sil.org/linguistics/wordlists/english/wordlist/wordsEn.txt)

We'll almost always create a stream using the program `cat`, which simply dumps the contents of the file to the terminal:

```
$ cat en.txt
```

If you just want the first 100 lines of the file, you can get them using the Unix tool `head`:

```
$ head -n 100 en.txt
```

Alternatively, you can create the stream using `cat` and then use `head` as a filter in the sense above using the pipe symbol "`|`":

```
$ cat en.txt | head -n 100
```

This pipeline has the same effect as the previous command. Even though it's more typing (and possibly slightly slower), we prefer to use `cat` at the beginning of pipelines since it better illustrates the general philosophy here: create streams of text from files, and then apply a sequence of mappers, filters, and reducers which all read from standard input and write to standard output. We do this in our everyday lives, and will also do it in the notes that follow.

Similar to `head`, the filter `tail` keeps only the specified number of lines at the end of the stream:

```
$ cat en.txt | tail -n 100
```

The filter `tac` reverses a stream:

```
$ cat en.txt | tac
```

(Again, most people would just type `tac en.txt`, but I'm weird).

And, `shuf` is a Unix tool that reads in lines of text, randomizes them, and outputs them:

```
$ cat en.txt | shuf
```

Next, let's apply the Unix tool `egrep` as a filter. It keeps only the lines that "match" the pattern that was provided; in this case any word containing the substring "good":

```
$ cat en.txt | egrep 'good'
```

Here's the thing... you can match patterns that are much more complicated than fixed strings like "good". And, this is where the real power comes in; by choosing these patterns well, you can select exactly what you want from the text stream in quite sophisticated ways. These patterns are called "regular expressions", and there's an extensive (and beautiful) theory of *finite state automata*, worked out in the 1940's and 1950's, that describes the sets of strings that can be matched by regular expressions. And, the best part is that these theoretical results and algorithms underlie the actual implementations of Unix tools like `egrep`, and explain why they're so fast!

Before we get into this too deeply, if you have some familiarity with Unix you may have heard of the tool `grep`, which very much like the `egrep` we're using here. `egrep` stands for "extended `grep`" and it uses a somewhat more convenient syntax for describing regular expressions (more on this momentarily). So you can simplify your life by just sticking with `egrep` even in cases where the behavior would be the same. (And, if you're curious what `grep` stands for, you can check out the Wikipedia article).

Examples of regular expressions:

Maybe you've heard the rule "i before e, except after c." Let's prove that it's false:
```
$ cat en.txt | egrep 'cie'
$ cat en.txt | egrep '[^c]ei'
```

The pattern `[^c]` means "any single character other than c."

Square brackets *without* a caret (`^`) in a pattern mean "exactly one of any of the characters inside the brackets." So, you can find words with three (or more!) vowels in a row as follows:

```
$ cat en.txt | egrep '[aeiou][aeiou][aeiou]'
```

Curly braces can be used to indicate a desired number of repetitions. There are a handful of words in English (like "queueing") with five consecutive vowels. We can find them like this:

```
$ cat en.txt | egrep '[aeiou]{5}'
```

How about long sequences of consonants (e.g., "latchstring")?

```
$ cat en.txt | egrep '[^aeiouy]{6}'
```

A range of numbers is allowed inside the curly braces:

```
$ cat en.txt | egrep '[^aeiouy]{5,7}'
```

For Scrabble players:

```
$ cat en.txt | egrep 'q[^u]'
```

A period/full-stop (`.`) in a pattern matches any single character:

```
$ cat en.txt | egrep 'z.z'
```

An asterisk in a pattern is called a "Kleene star"; it means "match 0 or more of the preceding pattern." So, this first example will give words with consecutive k's, or with k's separated only by vowels:

```
$ cat en.txt | egrep 'k[aeiou]*k'
```

Sometimes you want "one or more" of a pattern. You could write `[aeiou][aeiou]*` to get one or more vowels, but this is common enough that there’s a special notation for it; a plus sign instead of an asterisk:

```
$ cat en.txt | egrep 'k[aeiou]+k'
```

The pattern below will match any line containing a character other than a letter a-z.

```
$ cat en.txt | egrep '[^a-z]'
```

You’ll see there are several words that contain an ASCII apostrophe. To this point, we’ve used apostrophes around the pattern when using `egrep`, but you can also use double quotes. So, if the pattern contains an apostrophe, use double quotes, and vice versa. For example, to match the words containing an apostrophe directly:

```
$ cat en.txt | egrep "'"
```

The default behavior is to print any line which contains a match *anywhere in that line*.

You can require the whole string to match with “beginning of line” and “end of line” anchors:

```
$ cat en.txt | egrep '^[aeiou]+$'
```

In a Facebook post a couple of years ago, a colleague observed that not all words ending in “-ly” are adjectives or adverbs formed by adding “ly” to an existing word in the way that “friendly” or “gladly” are formed, and gave the (funny) examples “barfly” and “prickly”. You can search for four- and five-letter examples like this:

```
$ cat en.txt | egrep '^.{2,3}ly$'
```

Let’s do our first real pipeline, by adding on additional `egrep`s:

```
$ cat en.txt | egrep '^anti' | egrep -v "'s$"
```

Instead of filtering out the lines that don’t match, `egrep -v` filters out the lines that *do* match.

You may have heard of the 1939 novel “Gadsby” by Ernest Vincent Wright. There are about 50,000 words in the book, none of them containing the letter “e”. Here are the words Wright could choose from:

```
$ cat en.txt | egrep -v 'e'
```

You can count them by extending the pipeline with the Unix tool `wc -l`, which outputs the total number of lines it reads from standard input:

```
$ cat en.txt | egrep -v 'e' | wc -l
```

`wc` is a good example of a “reducer” in the sense we introduced above.

Next, “capturing parentheses”!

The next example finds words that contain the *same* vowel occurring twice in a row:

```
$ cat en.txt | egrep '([aeiou])\1'
```

The idea is that, first, the parentheses “capture” the *text* that’s matched by the pattern inside the parentheses. The `\1` serves as a “back reference” - to that actual *text* (e.g., `e`) and *not* to the *pattern* (e.g., `[aeiou]`) - that then matches that captured *text* (e.g, an `e` versus an `a`, `e`, `i`, or `u`). The key thing to understand is why this is different from:

```
$ cat en.txt | egrep '[aeiou][aeiou]'
```

which matches an `a`, `e`, `i`, or `u` followed by an `a`, `e`, `i`, or `u`; which is quite different from matching a captured `a`, `e`, `i`, or `u`, immediately followed by that *same* captured letter.

The following pattern will match words containing the same vowel three times in a row:

```
$ cat en.txt | egrep '([aeiou])\1\1'
```

And, this pattern will look for the occurrence of a consonant appearing three times in a row:

```
$ cat en.txt | egrep '([^aeiou])\1\1'
```

This pattern will look for words like “couscous” and “murmur” which are made up of a substring repeated twice:

```
$ cat en.txt | egrep '^(.+)\1$'
```

Since this matches some two-letter words like “cc”, how could you tweak it to discard those? You could do it by changing the pattern slightly, or by extending the pipeline to filter out the words you don’t want....

Now, notice that in every case, `egrep` returns the entire line that matches. Sometimes, you just want it to print the part of the line that matches. Use the “`-o`” flag for this:

Find all diphthongs:

```
$ cat en.txt | egrep -o '[aeiou]{2}'
```

Then count them:

```
$ cat en.txt | egrep -o '[aeiou]{2}' | sort | uniq -c | sort -r -n
```

Or even a character frequency list:

```
$ cat en.txt | egrep -o '.' | sort | uniq -c | sort -r -n
```

Or predictive text statistics – what is the character most likely to follow “ca” in English?

```
$ cat en.txt | egrep -o 'ca.' | sort | uniq -c | sort -r -n
```

(Incidentally, there are no examples of “caa” or “cax” in this word list – just `sort | uniq` to see that). Techniques like this can be used to create predictive text input systems for cell phones.

What other experiments can you think of?

Simple tutorials: http://regexone.com/
Crossword puzzles: http://regexcrossword.com/
Regex golf: http://regex.alf.nu/ (see also http://xkcd.com/1313/)
The t-shirt: http://store-xkcd-com.myshopify.com/products/i-know-regular-expressions

---

Lecture 3.

Next, we’ll do a similar statistical analysis of text, but this time using “real” texts which can be grabbed from Project Gutenberg (where texts have been proofread) or the Internet Archive (where texts have been scanned and OCR’d, but not checked).

In the examples folder, you’ll see `macbeth.txt`, which is a version of Shakespeare’s *Macbeth* from Project Gutenberg (original source: http://www.gutenberg.org/ebooks/1129.txt.utf-8)

Before hacking on the text, it’s worth opening it in a text editor and seeing what’s there; in this case you’ll see some legal text, copyright notices, etc. It’s important to be aware of this if you’re analyzing word frequencies; if this were a serious study, you’d want to clean up the source file and remove this kind of cruft.

```
$ vim macbeth.txt
```

Remember that `egrep`, etc. work at the level of lines by default, so you get the *whole line* that contains a match by default:

```
$ cat macbeth.txt | egrep 'ious '
```

will give all lines that contain a word ending in “ious”, followed by a space. To output just those words, you could use this:

```
$ cat macbeth.txt | egrep -o '[A-Za-z]*ious '
```

Here’s an attempt at getting all of the words in the file:

```
$ cat macbeth.txt | egrep -o '[A-Za-z]+'
```

Then, do the now-idiomatic frequency list thing:

```
$ cat macbeth.txt | egrep -o '[A-Za-z]+' | sort | uniq -c |sort -r -n
```

Note some stuff of interest – “s” and “d” and “ll” appear as “words” of high frequency because we didn’t include apostrophes as word characters. You see names in all-caps which are mostly the names of people speaking. And, there’s plenty of Elizabethan English in the top 200 most frequent words: “thou”, “thee”, etc.

Using Unix tools, can you figure out who speaks the most?

A first cut…

```
$ cat macbeth.txt | egrep ' [A-Z]+\. '
```

From this, notice that the Gutenberg editors are consistent about names starting two spaces from the beginning of the line:

```
$ cat macbeth.txt | egrep '^  [A-Z]+\. '
```

but note that we’ve lost “LADY MACBETH”, “OLD MAN”, etc. because of spaces, so:

```
$ cat macbeth.txt | egrep -o '^  [A-Z ]+\. ' | sort | uniq -c |
  sort -r -n
```

Lecture 4.

Next, let's actually start changing files around.

First, `tr` is a pipeline tool for "translating" characters. Typical uses:

```
$ cat macbeth.txt | tr "a-z" "A-Z"
```

```
$ cat macbeth.txt | tr -s '*'
```

It's time we have a heart-to-heart about Unix vs. DOS line endings. On Windows machines, the end of line in a text file is marked by two characters: a carriage return (CR) and a line feed (LF). That's decimal ASCII values 13 and 10, respectively. In the more refined Unix world, we mark the end of line with just an LF. Sadly, DOS line endings are quite common in the wild, and they're a huge pain if you're using the Unix tools. (Matching at the end of a line with $ won't work the way you expect it to, for example.) So, you should always check for them and strip them (as demonstrated below) before working in earnest on a file.

For example, the `macbeth.txt` in the example folder (originally from Project Gutenberg) has DOS line endings. You can see this by opening it in `vim` for example:

```
$ vim macbeth.txt
```

and you'll see something like this at the bottom of the screen:

```
"macbeth.txt" [dos] 3196L, 129093C
```

Or there's a great hacker tool called `xxd` that shows you the hex values of every byte in the stream you pass through it:

```
$ cat macbeth.txt | xxd
```

Look for `0d0a` in the output!

One way to strip the carriage returns is to use `tr`:

```
$ cat macbeth.txt | tr -d "\r" > macbeth-unix.txt
```

Note that we've redirected the output to a new file, which we'll use in all of the examples below.

`sed` is a tool for doing replacements also, but instead of single characters for characters, you can match and replace any regular expression! So, in other words, it gives you the text-matching power of `egrep` with the ability to actually change the text. It's clearly the best Unix tool by far; we use it every day. Even better, it was written by a Saint Louis University grad, Lee McMahon, who worked at Bell when Unix was first developed.

We could use `sed` to try and "modernize" Macbeth:

```
$ cat macbeth-unix.txt | sed 's/thee/you/'
```

But, note that this only performs the substitution on the first instance of "thee" on each line. To apply it to every instance of "thee", use the "`g`" flag at the end of the command:

```
$ cat macbeth-unix.txt | sed 's/thee/you/g'
```

Modernize more words in a longer pipeline:

```
$ cat macbeth-unix.txt | sed 's/thee/you/g' | sed 's/hath/has/g' |
  sed 's/thou/you/g'
```

Now these matches occur *anywhere,* even in the middle of a word. How could we check if this is going to be a problem? How about grepping in en.txt? Doing so shows us that "thee" and "hath" are probably ok, but "thou" is a problem. We could also grep for `'thou[a-z]'` in macbeth-unix.txt for more concrete evidence.

We *could* say

```
$ cat macbeth-unix.txt | sed 's/ thou / you /g'
```

and be safe, but we'd miss examples followed by punctuation!

Let's kill leading spaces:

```
$ cat macbeth-unix.txt | sed 's/^ *//'
```

And, kill trailing spaces:

```
$ cat macbeth-unix.txt | sed 's/ *$//'
```

And, then, force one space after a period.

```
$ cat macbeth-unix.txt | sed 's/\.  */. /g'
```

We can refer back to the (full) matched pattern with a '`&`'. For example, to insert a blank line before each speech:

```
$ cat macbeth-unix.txt | sed 's/^  [A-Z]/\n&/'
```

Or you might note that there are stage directions in square brackets. We can kill those as follows:

```
$ cat macbeth-unix.txt | sed 's/\[[^]]*\]//g'
```

Lecture 5

There's a huge amount of useful data out on the web that's "trapped" in semi-structured HTML pages. The process of extracting this data into a more structured format is usually called "screen-scraping." There are libraries for parsing HTML documents in all major programming languages, but here we want to demonstrate just how easy it is to get the information you want from a page *without* doing a full parse, by just using the Unix tools.

For starters, grab a page from Wikipedia; really, any page in any language's Wikipedia will do, but for definiteness we've included this one (downloaded 7 Jan 2015) in the example folder:

```
$ wget https://en.wikipedia.org/wiki/Boots_Riley
```

To do this properly without parsing, it's important to remember that the Unix tools work line-by-line, but it's perfectly fine for HTML tags, etc. to span multiple lines. To deal with this, we usually start a pipeline by stripping out all newlines (both CR and LF, in case of DOS line endings).

See if you can come up with pipelines that answer the following questions. (The answers to these questions are below, at the end of this lecture.):

1. How many HTML tags (both open and close tags) are in the file?
2. How many of each type of tag (<a>, <li>, <div>, etc., ignoring attributes)?
3. Is the number of open tags of each type equal to the number of close tags of each type?
4. Find all links to other pages on the same Wikipedia.
5. Then, search for pages that are linked more than one time. (It's bad style to link the same article multiple times in the body of an article. So, this is a kind of quality check.)

The next exercise is to create a geojson file from a CSV file of latitude and longitude pairs. Geojson is a text-based file format for storing and exchanging map information; you can read

more about it here:
http://geojson.org/

When taught to a classroom of students, we have them tell us their hometowns, and enter them into a batch geocoding service, like this one:
http://www.findlatitudeandlongitude.com/batch-geocode/

which gives a .csv file back. If you're working through these notes alone, we've included a .csv file in the examples called `user2ll.csv`, which contains latitude/longitude pairs for about 300 Māori Twitter users, as extracted from their user profiles. (We've used datasets like this to create maps of Twitter conversations in a number of indigenous and minority languages, e.g. here or here.)

We'd like to turn this .csv file into a geojson file that can be used to plot a map.
Here's a sample geojson file that just contains simple points expressed as lat/long pairs:

```
{ "type": "FeatureCollection",
  "features": [
       { "type": "Feature",
       "geometry": { "type": "Point", "coordinates": [-39.512382, -18.970382] },
       "properties": { "weight": "1" }
       },
       { "type": "Feature",
       "geometry": { "type": "Point", "coordinates": [-33.512382, -19.970382] },
       "properties": { "weight": "1" }
       },
       { "type": "Feature",
       "geometry": { "type": "Point", "coordinates": [-38.512382, -10.970382] },
       "properties": { "weight": "1" }
       }
  ]
}
```

Note (in the geojson spec) that longitude comes before latitude, so we need to swap the order of the coordinates from the file `user2ll.csv` (or the coordinates that you got back from the geocoding service).

Here's the whole pipeline:

```
$ cat user2ll.csv | sed 's/^[^,]*,//' | sed 's/^\([^,]*\),\(.*\)$/{
"type": "Feature", "geometry": {"type": "Point", "coordinates": [\2,
\1] }, "properties": { "weight": "1" } },/' | tr -d "\n" | sed
's/,$/]}/' | sed 's/^/{ "type": "FeatureCollection", "features": [/'
```

The first `sed` command removes the usernames from the source file.

The second `sed` command does all of the hard work; it takes each latitude/longitude pair and turns it into the chunk of geojson that looks like `{ "type": "Feature" ... },`
Note that we use capturing parentheses (i.e., `sed 's/^\([^,]*\),\(.*\)$/` ...) when matching the latitude and longitude, and we refer back to them as `\1` and `\2` respectively; this is how we reverse their order.

The last step is to prepend the `{ "type": "FeatureCollection"` bit, and to tack on the closing `]}` ... at the end of the file. We do this the easy way, by stripping out all new lines, putting the entire stream onto a single line, and then using `sed` to add stuff at the beginning and end of that line. Note that we need to strip off the extra comma at the end of the last Point feature.

You can paste the output of this command into a mapping site, like this one:
http://geojsonlint.com/

This site will also report any errors in your geojson file, which is a good way of testing.

General remarks on these tools: the "Unix Philosophy"....

- Small, simple tools that do one thing well
- Clean interfaces (read stdin, write to stdout)
- No interactivity
- Process plain text vs. binary formats
- No unnecessary output (or it'll be a pain to process further)
- Don't delete from stream unnecessarily; "pass it on"

Answers to the screen-scraping exercises above:

1: Number of tags, total:
```
$ cat Boots_Riley | tr -d "\r\n" | egrep -o '<[^>]+>' | wc -l
```

2: Number of each kind of tag:
```
$ cat Boots_Riley | tr -d "\r\n" | egrep -o '<[^>]+>' |
  egrep -o '^<[A-Za-z0-9]+' | sed 's/^<//' | sort | uniq -c |
  sort -r -n
```

3: Check that number of close tags matches number of open tags of the same type. Need to be careful to filter out tags of the form `<br />`, etc.
```
$ cat Boots_Riley | tr -d "\r\n" | egrep -o '<[^>]*[^/]>' |
  egrep '^<[A-Za-z/]' | sed 's/<\([A-Za-z0-9]*\) [^>]*>/<\1>/' |
  sort | uniq -c | sort -r -n
```

4. Links to other pages on the same Wikipedia:

```
$ cat Boots_Riley | tr -d "\r\n" | egrep -o '<a[^>]+>' |
  egrep -o 'href="/wiki/[^"]+"' | sed 's/^href="\/wiki\///' |
  sed 's/"$//'
```

5. Find pages on the same Wikipedia that are linked more than once:

```
$ cat Boots_Riley | tr -d "\r\n" | egrep -o '<a[^>]+>' |
  egrep -o 'href="/wiki/[^"]+"' | sed 's/^href="\/wiki\///' |
  sed 's/"$//' | sort | uniq -c | egrep -v ' 1 '
```

Lecture 6

Next, working with tabular data (.csv, .tsv) using `cut`, `paste`, and `join`.

The goal of this lecture will be to create a “heatmap” showing the number of tweets coming from certain locations. We’ll start with two .csv files (from the example folder); the first, `tweets.csv`, has two fields in each row:

```
username,numberoftweets
```

and the second, `user2ll.csv`, is the same one we used in the previous lecture. It has three fields in each row:

```
username,lat,long
```

Warmup; how to kill all but username?

```
$ cat user2ll.csv | sed 's/,.*//'
```

How to kill last field? Here’s a first naive attempt:

```
$ cat user2ll.csv | sed 's/,[^,]*//'
```

but, this kills the second field!

Here’s a better way:

```
$ cat user2ll.csv | sed 's/,[^,]*$//'
```

Or, it's easy to use `cut` to pick the fields you want to display:

```
$ cat user2ll.csv | cut -d ',' -f 1
$ cat user2ll.csv | cut -d ',' -f 2
$ cat user2ll.csv | cut -d ',' -f 2,3
```

Now, `paste` lets you put things back together.

First, we'll `cut` up our `user2ll.csv` file into two separate files, each containing a column of data. Then, we'll `paste` the two columns back together:

```
$ cat user2ll.csv | cut -d ',' -f 1 > firstcolumn.txt
$ cat firstcolumn.txt
$ cat user2ll.csv | cut -d ',' -f 2 > secondcolumn.txt
$ cat secondcolumn.txt
$ cat firstcolumn.txt | wc -l
$ cat secondcolumn.txt | wc -l
$ cat user2ll.csv | wc -l
```

`Paste` defaults to tab-separated output:

```
$ paste firstcolumn.txt secondcolumn.txt
```

But, of course you can customize with a command-line option:

```
$ paste -d ',' firstcolumn.txt secondcolumn.txt
```

Next is `join`. The main difficulty with using `join` in practice is that you need to sort on the join field, as we'll illustrate below.

We've seen `sort` before:

```
$ cat user2ll.csv | sort
```

But, this sorts the whole line, since the comma is regarded as just another character to sort. This is not what we want in this case.

To sort on the first field (and have the comma regarded as a delimiter for that first field):

```
$ cat user2ll.csv | sort -t ',' -k 1,1 > users-sorted.csv
$ cat tweets.csv | sort -t ',' -k 1,1 > tweets-sorted.csv
$ join -t ',' tweets-sorted.csv users-sorted.csv
```

Let’s turn this into data for a “heat map” of tweets:

```
$ join -t ',' tweets-sorted.csv users-sorted.csv |
  sed 's/^[^,]*,//' | sed 's/^\([0-9]*\),\(.*\)$/\2,\1/'
```

You can then upload the output to a site like http://cartodb.com/ to create a heatmap or other visualizations.

tsv files are sometimes easier since tabs are the default delimiter for most of these tools. If you have csv files, you can can always convert:

```
$ cat tweets.csv | tr "," "\t"
```

or:

```
$ cat tweets.csv | sed 's/,/\t/g'
```

Lecture 7

Most modern machine translation engines are based on statistical models. Broadly speaking, the idea is to assemble a large corpus of texts and their translations, align them sentence-by-sentence, and then gather statistics about which words and phrases translate to which words and phrases.

Let’s use the Unix tools to put together a parallel corpus of texts from the Bible. A good source of Bible texts in many languages is the “Unbound Bible” site:
http://unbound.biola.edu/index.cfm?method=downloads.showDownloadMain

We’ve included the Lithuanian and Māori bibles from this site in the examples folder (in the files `lithuanian_utf8.txt` and `maori_utf8.txt` respectively). Browsing these files, you’ll see that they have one verse per line, and six tab-separated fields on each line, the last being the text of the verse. In trying to align these verse-by-verse, we immediately run into difficulties, in particular because the number of lines in the two files are different. Remember this is the real world, and the real world is noisy! Basically, some verses are included in the Lithuanian Bible that aren’t present in the Māori one, and vice versa.

How could we figure out which are the “problem” verses? The Unix tools!

What we do is:

- create a stream from the two files using `cat`;
- strip the CR's and comments;
- use `cut` to select the fields that uniquely identify the book, chapter, and verse;
- count the number of times each one appears; and,
- keep the handful that appear only once. (Most of them appear twice.)

```
$ cat lithuanian_utf8.txt maori_utf8.txt | tr -d "\r" |
  egrep '^[^#]'| cut -f 1,2,3 | sort | uniq -c | egrep ' 1 '
```

This pipeline should output the seven verses that appear in only one of the two files. This number is small enough that you could probably just open the files in a text editor and remove the mismatched verses manually. But, let's take this opportunity to teach you a nice feature of `egrep`. The idea is to convert these verse numbers into regular expressions that you can store in a file, and then use `egrep` to filter out any lines matching a pattern in the file. To this end, let's extend the pipeline above a bit, and redirect the output to a file `patterns.txt`:

```
$ cat lithuanian_utf8.txt maori_utf8.txt | tr -d "\r" |
  egrep '^[^#]'| cut -f 1,2,3 | sort | uniq -c | egrep ' 1 ' |
  sed 's/^ *1 //' | sed 's/^/^/' | sed 's/\t/./g' > patterns.txt
```

Have a look at `patterns.txt`. You'll see that we're anchoring the patterns to the beginning of a line with `^`, and we're using `.` to match the tab characters.

Next, using the `-f` flag to `egrep` will allow us to specify a file containing patterns:

```
$ cat lithuanian_utf8.txt | tr -d "\r" | egrep -v -f patterns.txt |
  egrep -v '^#' | cut -f 6 > lt.txt
$ cat maori_utf8.txt | tr -d "\r" | egrep -v -f patterns.txt |
  egrep -v '^#' | cut -f 6 > mi.txt
```

As a final exercise, imagine that we need to combine these two texts into into a single .csv file containing one verse per row, the first column being Lithuanian and the second Māori. Since the texts themselves contain commas, we'll need to enclose each verse in double quotes. (And, fortunately, neither of the texts contains ASCII double quotes, so we don't have to worry about escaping those.) Using `paste` and `sed`, we can accomplish exactly that:

```
$ paste lt.txt mi.txt | sed 's/^\(.*\)\t\(.*\)$/"\1","\2"/'
```

Lecture 8

Let’s start with some familiar territory; we’ll process another HTML file from the English Wikipedia (`Linguistics` in the examples folder) and extract some useful info from it using `egrep` and `sed`.

As you may know, Wikipedia articles contain links to articles on the same topic in other languages. We can scrape these from the English Wikipedia to get translations of a given word into many languages. (These so-called “interwiki” links are now managed as part of the wikidata project. Getting the links from wikidata is probably the best way to do what we’re doing here, but we still like the method we’ll use below as a nice illustration of HTML scraping.)

We can build this up step-by-step:

```
$ cat Linguistics
$ cat Linguistics | egrep 'interlanguage-link'
$ cat Linguistics | egrep 'interlanguage-link' |
  egrep -o 'title="[^"]*'
$ cat Linguistics | egrep 'interlanguage-link' |
  egrep -o 'title="[^"]*' | sed 's/^title="//'
```

To remove the language names, we want to get rid of the hyphen and everything that follows. But, be careful... this is actually a Unicode dash, not an ASCII hyphen. This sort of thing happens quite often in the real world! You’ll also often see weirdness like non-breaking spaces that are difficult to distinguish from regular space characters. Remember that `xxd` and a good understanding of Unicode are your friends!

```
$ cat Linguistics | egrep 'interlanguage-link' |
  egrep -o 'title="[^"]*' | sed 's/^title="//' | sed 's/ – .*//'
```

Now, you can use the up-arrow to repeat this command any time you want (if the `Linguistics` file is in the current directory). But, what about if you log out later? You can save the command in a file! Copy the command (without the prompt), and paste it into a file named `Linguistics.sh`.

You’ve just created your first *shell script.* It’s a (one-line) computer program, no different than writing a Perl or Python program. To run it, use:

```
$ bash Linguistics.sh
```

This is the analogue of running a python program with `python program.py`.

Sidebar on `history`. The `history` command outputs the history of all commands you’ve entered at the command-line during the current session. The cool thing is that the output of `history` is a text stream like any other, and you can process it with any of the Unix tools:

```
$ history
```

To find every time you used the command `wget`:

```
$ history | egrep wget
```

You can even select a bunch of previous commands using `egrep`, and rerun them by saving them in a shell script:

```
$ wget http://en.wikipedia.org/wiki/Irish_phonology
$ wget http://en.wikipedia.org/wiki/Saint_Louis_University
$ wget http://en.wikipedia.org/wiki/Indigenous_Tweets
```

Let’s say you accidentally wipe out these three files that you’ve just downloaded:

```
$ rm Irish_phonology Saint_Louis_University Indigenous_Tweets
```

Rerun the `wget` commands like this:

```
$ history | egrep wget
$ history | egrep wget | egrep -v history
$ history | egrep wget | egrep -v history | sed 's/^ *[0-9]* *//'
$ history | egrep wget | egrep -v history |
  sed 's/^ *[0-9]* *//' > wgetter.sh
$ bash wgetter.sh
$ ls
```

Something awesome just happened here; we “wrote” a new program (a shell script) using a program. More on this in a sec.

Next, the `Linguistics.sh` program is nice, but only deals with a single Wikipedia article. We can do the same thing with any article. Let’s generalize:

```
$ mv Linguistics.sh Multilingual.sh
```

We’d like to be able to say, for example:

```
$ bash Multilingual.sh Mathematics
```

and have it output a bunch of “translations” of the given term. To do this, open the script in your favorite text editor:

```
$ vim Multilingual.sh
```

and do two things: first, insert a line

```
wget "http://en.wikipedia.org/wiki/${1}"
```

at the beginning of the file. Then, change `cat Linguistics` to `cat "${1}"` at the beginning of the pipeline. The variable `${1}` refers to the first “command-line” argument passed to the shell script (“`Mathematics`” in the example above). We should also probably add a check to ensure that the user has provided the correct number of command-line arguments, but we won’t worry about that now.

```
$ bash Multilingual.sh Justin_Bieber
$ bash Multilingual.sh Justin_Bieber | wc -l
$ bash Multilingual.sh Science
$ bash Multilingual.sh Baseball
$ bash Multilingual.sh Beatles
$ bash Multilingual.sh Linux
$ bash Multilingual.sh Propaghandi
```

This keeps the downloaded files in the current directory. You might be tempted to add a line like `rm "${1}"` at the end of the script; this is exceedingly dangerous, though, without some *very* careful checking of the command-line argument. (A user passing a string containing a wildcard might wipe out their whole hard drive.)

The output from `wget` is a bit annoying. Programs on the command line can actually send their output to two different places. The text streams we’ve worked with to this point are being sent to “standard output”, a.k.a. “stdout”. Some text can go to what’s called “standard error”, or “stderr”. This is what you’re seeing from `wget`.

If you redirect output, you’re just redirecting standard output; so

```
$ bash Multilingual.sh Propaghandi > p.txt
```

dumps the list of translations into the file `p.txt`, but you still see the output from `wget`. Again, that’s because it’s going to stderr, not stdout. You can redirect everything to one place as follows:

```
$ bash Multilingual.sh Propaghandi &> p.txt
```

And, if you want to silence a command completely and don’t want to keep the output in a file, you can “redirect to dev-null” (a.k.a. “the great bitbucket in the sky”) by changing the `wget` command in `Multilingual.sh` to this:

```
wget "http://en.wikipedia.org/wiki/${1}" &> /dev/null
```

Lecture 9

You can even have shell scripts that work in the middle of a pipeline. This is a big idea! The Unix tools themselves are useful exactly because each one does something small and does it well, and each reads from stdin and writes to stdout so they can be piped together. But, this ecosystem of tools is not closed... *you can define your own Unix tools*. And, the most useful and reusable tools are themselves filters, i.e. they read from stdin and write to stdout, exactly because they can then be used in pipelines.

A nice example is the idiomatic `sort | uniq -c | sort -r -n` business we've seen over and over in previous examples. Put this in a file called `counter.sh`:

```
$ vim counter.sh
```

You could run this with `bash counter.sh` but it would just sit there apparently not doing anything. Why? Try this first:

```
$ sort
```

It's working, but the text stream it's reading from is "standard input", a.k.a. the keyboard. We can type a few lines and hit Ctrl-D and it'll work as expected. To use `counter.sh` in earnest, you just pipe an interesting text stream into it:

```
$ bash Multilingual.sh 'Justin_Bieber' | bash counter.sh
```

Now for more "programs that write programs". What if we want not just the names of the articles in other languages, but we want to get those articles themselves?

This pipeline will get a list of URLs pointing to those documents:

```
$ cat Linguistics | egrep 'interlanguage-link' |
  egrep -o 'href="[^"]*'  | sed 's/^href="/http:/'
```

So, we've turned a text stream coming from a Wikipedia article in HTML into a new text stream that has some useful data, a list of URLs. But, it gets better. What if we do this:

```
$ cat Linguistics | egrep 'interlanguage-link' |
  egrep -o 'href="[^"]*'  | sed 's/^href="/http:/' |
  sed 's/^/wget /' > mygetter.sh
```

We’ve just turned a text stream coming from a Wikipedia article into a *program* that does something awesome. It’s an example of what some people call “higher-order programming”.

Now, executing

```
$ bash mygetter.sh
```

will accomplish precisely what we’ve set out to do: fetch all of the actual documents being linked to within the “Linguistics” Wikipedia article.

We can even just pipe the stream commands directly to the bash interpreter without saving them as a shell script:

```
$ cat Linguistics | egrep 'interlanguage-link' |
  egrep -o 'href="[^"]*'  | sed 's/^href="/http:/' |
  sed 's/^/wget /' | bash -s
```

Here’s a final example... suppose we have a directory containing a bunch of `.htm` files and want to rename them all as `.html`:

```
$ ls *.htm | sed 's/\(.*\)\.htm/mv \1.htm \1.html/' > mover.sh
```

Be sure it looks good!

```
$ vim mover.sh
```

If so, run it:

```
$ bash mover.sh
```

Lecture 10

What we've done so far is "hacking" on the command line. Basically, we create a text stream, tweak it with pipeline commands, examine the output, then add more to the pipeline until the output looks how we want it to.

This isn't how disciplined software engineers work, however.

For example, once you have a pipeline you're happy with, you ought to save it in a script (either as a .sh file, or using other methods which we're about to discuss) so that you can reproduce the same behavior later.

The other thing we want to do is "Version Control". A Version Control System is a piece of software that developers use to track all changes in all data files and source code files in a project.

Why use version control?

- Makes collaboration easier. Shared, up-to-date code base.
- Tracks history of every change with log message explaining the reason for it. Log messages often refer to bug reports.
- Can serve as a (maybe suboptimal) method of effort tracking.
- Allows “diff” between any two versions.
- Essential for debugging and “root cause analysis”. Find the exact commit that introduced a given bug.
- Most tools allow “blame view” – a way to look at a file with each line annotated by the person who added that line, and a cross-reference to the commit that added the line.
- Backup of all files. Some systems use a “centralized repo”, while others, like git, use a “distributed” model – which means that no one copy of the repo is more important than others, theoretically.
- Allows you to maintain multiple “branches” of a project. Stable branch, “next” branch for next stable release, “trunk” or bleeding edge branch, etc.
- Branching allows fearless experimentation. Create a branch, try something, and if it doesn’t work, discard the branch. If it does work, “merge” in with the main development branch. And, if you change your mind after merging, you can still “revert” at any time.

We’ll use `git` which has become the de facto industry standard.

`git` is often (but not always) used in conjuction with the website http://github.com/ which is a central place for storing your `git` repositories so that a group of people can collaborate on a single piece of code.

Before proceeding, if you don’t have an account on github, you should create one and then make sure you have `git` installed on your Linux machine, by typing:

```
$ git --version
```

If you’re new to `git`, you’ll avoid errors in the future by specifying your name and email address (for the log files) as follows:

```
$ git config --global user.email "you@example.com"
$ git config --global user.name "Your Name"
```

Now, let’s create your first repository, in a subdirectory of your home directory:

```
$ cd
$ mkdir MyProject
$ cd MyProject/
$ git init
```

The repo is empty:

```
$ ls
```

But there are some “hidden” data files in a subdirectory called `.git`:

```
$ ls .git
```

Let’s create some interesting files and commit them to the repository. To do that, we want to detour and talk about the Unix tools `find` and `xargs` for a moment...

Returning to the examples folder, there’s a subdirectory `irishblogs` which contains a bunch of text files:

```
$ cd ~/examples
$ ls irishblogs
```

(Reminder – how to figure out how many!)

```
$ ls irishblogs | wc -l
```

`find` is a bit like `ls` but much more powerful. By default it recurses over all subdirectories under the given directory:

```
$ find irishblogs
```

And, you can specify “filters”:

```
$ find irishblogs -name '12*'
$ find irishblogs -name '15?.txt' | sort
```

We won’t go into them all, but you can filter by modification time, file size, etc.

```
$ man find
```

What if we want to do the same shell command to every file? (We saw something like this toward the end of the previous lecture, Lecture 9, when we moved a bunch of files using our “higher order programming” `mygetter.sh` bash script.)

```
$ find irishblogs -name '12*' > FILE
$ cat FILE
$ cat FILE | sed 's/^/egrep " agus " /' > FILE.sh
$ cat FILE.sh
$ bash FILE.sh
```

With the command `xargs` you can do all of this in one command. `xargs` is a pipeline tool, so it reads from standard input like everything else. But, it expects a text stream consisting of file names, one per line. So, quite often we use `find` to pipe filenames into it. What `xargs` does is execute the given command on each file that it gets.

```
$ find irishblogs -name '12*' | xargs egrep " agus "
```

Since our `FILE.sh` bash version above didn’t print the file names of the text files, let’s reproduce that behavior by including the `-h` flag with `egrep`:

```
$ find irishblogs -name '12*' | xargs egrep -h " agus "
```

(Also, note that this was faster than the version with bash!)

To concatenate together all of the files whose names start with “12”:

```
$ find irishblogs -name '12*' | xargs cat
```

And, now, let’s commit some data to the repository:

```
$ find irishblogs -name '1*' | xargs egrep -h ' agus ' | shuf > FILE
$ mv FILE ~/MyProject
$ cd ~/MyProject
$ git add FILE
$ git commit -m "Adding data file"
$ git log FILE
```

Now, let’s hack on it... Let’s say we want to find references to years (1994, 2014, etc.) in this file. We’ll begin building up this pipeline:

```
$ cat FILE | egrep -o '[0-9]+' | egrep '^(19|20)[0-9][0-9]$' | sort |
  uniq -c | sort -r -n
```

Ok. This pipeline (starting with the first `egrep)` is nice and useful, something you might imagine using in other contexts. So, we might save it in a script, calling it `yearfinder.sh`:

```
$ vim yearfinder.sh
$ cat FILE | bash yearfinder.sh > commonyears.txt
$ git add yearfinder.sh
$ git commit -m "new script"
```

Now, let's say that at some point we become unhappy with `commonyears.txt`. For example, maybe our requirements have changed and we need to include years from the 1800's. Or, perhaps we want the year to appear first, followed by the frequency.

So, we tweak `yearfinder.sh` to our liking.

To reverse the display position of the year and the frequency, we could add a `sed` command to the end of our pipeline contained in `yearfinder.sh`, as follows:

```
egrep -o '[0-9]+' | egrep '^(19|20)[0-9][0-9]$' |
sort | uniq -c | sort -r -n | sed 's/^ *\([0-9]*\) \(.*\)$/\2 \1/'
```

We'll edit and resave `yearfinder.sh` so that it contains only this new pipeline:

```
$ vim yearfinder.sh
```

And, then test:

```
$ cat FILE | bash yearfinder.sh > commonyears.txt
$ cat commonyears.txt
```

If `commonyears.txt` looks better, and each year indeed appears in front of its respective frequency, as we intended, then we can now commit this new version of `yearfinder.sh` to the repository:

```
$ git diff
```

('q' to quit)

```
$ git add yearfinder.sh
$ git commit -m "reformat output"
$ git log
```

We can then make another change, to allow years from the 1800's as well, for example, and repeat the above steps. Test. Verify. Commit.

We'll discuss a more sophisticated workflow, involving "branching" and "merging," in the next lecture.

Side remark – do you want to commit `commonyears.txt` and track all changes? The conventional wisdom is "no," because it can be regenerated from files that are in the repo (namely, `FILE` and `yearfinder.sh`), but it's up to you.

We can change the data file, too. For example, maybe we try to filter out English by deleting lines containing " the ". (Dumb idea. But, whatever.):

```
$ egrep ' the ' FILE
$ cat FILE | egrep -v ' the ' > FILE2
$ mv FILE2 FILE
$ git diff
```

('q' to quit)

```
$ git add FILE
$ git commit -m "kill all lines with the"
$ git log
```

('spacebar' to continue, or 'q' to quit)

```
$ git diff
```

Lecture 11

Let's talk about make and makefiles.

When we left off last time, we'd created a git repo, `~/MyProject`, and were using it to track all changes in our files. Let's add some data to this directory!

There's an example file called `gd-tweets.txt` that contains a sampling of tweets in Scottish Gaelic. Twitter allows some characters with diacritics (accents) to appear within a hashtag, but not others. And, this has an interesting effect on some users' behavior, in that they'll drop diacritics in hashtags that they would typically include otherwise. So, let's say we want to analyse usage of diacritics in hashtags in this collection.

Since we're about to embark on some changes, we get a cup of coffee and begin our workday with a fresh new "branch." We'll name this new branch 'addmakefile'. From within our git repo directory `~/MyProject`:

```
$ git checkout -b addmakefile
```

This starts as an exact copy of the "master" branch. But, then we can change it, test those changes, etc. And, when we're satisfied, we merge the changes back into the master branch. Or, if the changes don't pan out, we can just delete the branch, and we're back to where we started. No harm, no foul!

Back to hacking on the command line... We'll first write something that grabs all hashtags (and, that will get diacritics, too, in a UTF-8 locale):

```
$ cat gd-tweets.txt | egrep -o '#[A-Za-z0-9]+' |
  sort | uniq -c | sort -r -n
```

As disciplined software engineers, maybe we store this pipeline in a script that we can run later:

```
$ vim script2.sh
$ bash script2.sh > hashtags.txt
```

A lot of what we do is structured this way. A text stream gets filtered into something we want to keep; we then save it as a file. Commands of this kind are most conveniently stored in a "makefile".

```
$ vim makefile
```

Then, we add lines that look like this:

```
hashtags.txt: gd-tweets.txt
	cat gd-tweets.txt | egrep -o '#[A-Za-z0-9]+' | sort |
	uniq -c | sort -r -n > hashtags.txt
```

We could also have put `bash script2.sh > hashtags.txt`. But, it's much nicer having all of the commands here in this one file.

That's a TAB before the shell command (i.e., immediately preceding "`cat gd-tweets.txt ...`"); it needs to be there. We've heard this requirement described as "the worst design decision in the history of software."

Additionally, we need to be sure to enter the entire shell command as a single, unbroken line.

Let's also replace `hashtags.txt` at the end of the shell command with `$@`, which stands for "target filename". It's less typing and is convenient if we ever want to change the name of the file later.

So, let's do it all in the makefile:

```
$ rm -f script2.sh
$ rm -f hashtags.txt
$ make hashtags.txt
```

Or, because it's the first (only) target in the makefile, we can just:

```
$ make
```

And, if we run `make` again:

```
$ make
```

It doesn't do anything. Which is to say, `make` is SMART. It checks timestamps on all dependencies (recursively), and only remakes those targets that are out-of-date with respect to their dependencies. For those who've studied algorithms, it does a "topological sort" on the digraph of dependencies to decide what gets built first.

But, if we change the dependency, say by opening `gd-tweets.txt` in a text editor and deleting one of the tweets, or even just by applying `touch` to update its timestamp:

```
$ touch gd-tweets.txt
```

and run `make` again, it does actually rebuild:

```
$ make
```

To add a second target, we add the following lines to the end of our makefile, remembering that a TAB needs to immediately precede the shell command:

```
accentedtags.txt: hashtags.txt
	cat hashtags.txt | egrep '[ÁÉÍÓÚáéíóúÀÈÌÒÙàèìòù]' |
	sed 's/^ *[0-9]* //' > $@
```

And, it's typical to add an "`all`" target at the very top of the makefile:

```
all: accentedtags.txt
```

This way, when we just type `make`, it will try to make any dependencies which are listed there.

Now, we want to strip all accents from the hashtags we extracted from `gd-tweets.txt`, and search for the ASCII versions. For this, we need a way of stripping accents from a text stream. This might be a nice, reusable command that we can put into a script which we'll call `toascii.sh`.

As a fun interlude, we can "write" this script using Unix tools. First, we `echo` a one-line text stream consisting of the accented characters we care about followed immediately by their ASCII versions. Then, using `sed`, we massage that echoed text stream into the script we want:

```
$ echo 'ÁAÉEÍIÓOÚUáaéeíióoúuÀAÈEÌIÒOÙUàaèeìiòoùu' |
  sed 's/\(.\)\(.\)/s\/\1\/\2\/g; /g' |
  sed "s/^/sed '/" | sed "s/$/'/" > toascii.sh
```

With this in place, we can add a new target to the makefile:

```
stripped.txt: accentedtags.txt
	cat accentedtags.txt | bash toascii.sh > $@
```

It's typical to include a "`clean`" target that will remove all generated files whenever `make clean` is executed. Let's add one, then, to the end of our makefile:

```
clean:
	rm -f stripped.txt accentedtags.txt hashtags.txt
```

Since this new functionality seems to work, we might decide to commit these changes to the repository, merge the branch back into master, and start a new cycle with a new branch. You should be on the branch `addmakefile`; we can always check where we are as follows:

```
$ git status
```

Then, we commit the new files to this branch:

```
$ git add makefile toascii.sh
$ git commit -m "add script that strips diacritics"
$ git status
```

and merge the branch back into the master branch:

```
$ git diff master
$ git checkout master
$ git merge addmakefile
$ git diff
$ git branch -d addmakefile
```

When we're ready to continue working, we start the process over again by creating a new branch - which we might name, for example, 'addreporttarget' - just as we did at the start of this lecture when we got a cup of coffee and began our workday with the fresh new branch named 'addmakefile'.

Let's create that new branch, which we'll indeed name 'addreporttarget'. From within our git repo directory `~/MyProject`:

```
$ git checkout -b addreporttarget
```

Next, to complete the analysis of diacritics in hashtags, we want to paste `accentedtags.txt` and `stripped.txt` together, and turn this into a script that searches for each in `hashtags.txt`. In practice, we would work out the pipeline command to do this at the command line in the usual way, and when it works, add it to the makefile. The result would look something like following, which we'll now add to our makefile (immediately preceding `clean:`):

```
search.sh: stripped.txt
	paste accentedtags.txt stripped.txt | sed 's/\t/|/' |
	sed 's/^/egrep " (/' | sed 's/$$/)$$" hashtags.txt; echo/' > $@

report.txt: search.sh hashtags.txt
	bash search.sh > $@
```

We'll update our makefile's "`clean`" targets, as well, to include `report.txt` and `search.sh`:

```
clean:
	rm -f report.txt search.sh stripped.txt accentedtags.txt
	hashtags.txt
```

Since `report.txt` is our ultimate goal, we'll change our makefile's "`all`" target to make `report.txt`:

```
all: report.txt
```

Now, we can run `make`, and check the results of our analysis of the usage of diacritics in hashtags:

```
$ make
$ cat report.txt
```

If everything worked as expected, we just commit and merge to master as usual:

```
$ git status
$ git add makefile
$ git commit -m "add targets for search.sh and report.txt"
$ git status
$ git diff master
$ git checkout master
$ git merge addreporttarget
$ git status
$ git branch -d addreporttarget
```

Lecture 12

When taught to a class, this lecture is run as a contest. The students work in pairs and see who can find the most palindromes in running English text, in a one-hour class period.

Every palindrome you find must be given together with an example sentence (or, more simply, just the line in the text file) in which it appears.

The rules: palindromes must (a) be at least four letters; (b) be all lowercase, so in particular no acronyms; and (c) contain at least two different letters, so no “xxxx” or “yyyy”.

(1) Find lots of English text, probably from the web, and save it as plain text.
(2) Create a word list from this.
(3) Find a way to pick out the palindromes from this list. (Hint: `rev`, `paste`, `egrep`; one solution is given below.)
(4) Take the list of palindromes you find, and turn it into a script that goes back and greps for example lines.

Here’s a solution for (3) – if `words.txt` is a wordlist with one word per line:

```
$ cat words.txt | rev | paste words.txt - |
  egrep '^([a-z][a-z][a-z][a-z]+).\1$' |
  egrep -v '^((.)\2+).\1$' | sed 's/\t.*//' > palindromes.txt
```

Lecture 13

For starters, let's hack on a cool dataset; this is the file `randomsample.txt` in the examples folder. It contains one sentence per line in 1474 different languages, found as part of the first author's Crúbadán web crawling project.

Let's explore:

```
$ cat randomsample.txt
$ cat randomsample.txt | wc -l
$ cat randomsample.txt | egrep '^a'
$ cat randomsample.txt | egrep '^a' | wc -l
```

What if we wanted to save each sentence to a separate file in a subdirectory `samples`, filename `<languagecode>.txt`?

To this point, everything we've done has involved linear processing of a text stream. NO TRADITIONAL PROGRAMMING CONSTRUCTS. Maybe you hadn't noticed this. No variables. No loops. No "if" statements. Notice it now – this is crazy, and awesome.

Well, we still don't need traditional programming constructs; we can just use the "programs that write programs" approach and massage the thing into a runnable script. A bit of care is needed here to deal with quotes; an easy hack is to just convert double quotes into entities for the script, and then convert them back when writing the files:

```
$ mkdir samples
$ cat randomsample.txt | sed 's/"/\&quot;/g' |
  sed 's/^\([^\t]*\)\t\(.*\)$/echo "\2" |
  sed "s\/\&quot;\/\\\"\/g" > samples\/\1.txt/'
```

We could save the output of this command as a .sh script and run it, or else pipe it directly to `bash -s` as we did earlier.

But, let's see how to do the same thing with a loop and variables. For starters, open a file called `script.sh` using your favorite editor:

```
$ vim script.sh
```

And, we'll put the following into it:

```
cat randomsample.txt |
while read x
do
  echo "$x"
done
```

The while loop reads each line one-by-one, assigning each to the variable “x”. Inside the loop, we just echo the line to standard output. So, the script above is exactly the same as `cat randomsample.txt`!!

Let’s do some more interesting processing inside the loop. We’ll edit `script.sh` again and change the stuff inside the loop:

```
cat randomsample.txt | egrep '^a' | tr "\t" " " |
while read x
do
  SENTENCE=`echo $x | sed 's/^[^ ]* //'`
  LANGCODE=`echo $x | sed 's/ .*//'`
  echo $SENTENCE > samples/$LANGCODE.txt
done
```

We run the script:

```
$ bash script.sh
```

and, then, check that everything looks good in the `samples` directory. How might we check to see that every file has exactly one line in it?

```
$ find samples -name '*.txt' | xargs wc -l | egrep -v ' 1 '
```

outputs the total number of lines across all files. So, if each file consists of exactly one line, the value returned by the above line of code should match the total number of files, which is the value given to us by the following line of code:

```
$ ls samples/ | wc -l
```

We can also process all of this with a loop... which introduces us to the “if” statement and conditionals. We should be aware that “types” of variables can be weird, and sometimes a numerical comparison might be made between one or more variables containing string data! We’ll replace all of the code in our script.sh file with the following:

```
find . -name '*.txt' |
while read x
do
  LINES=`cat $x | wc -l`
  if [ $LINES -ne 1 ]
  then
```

```
    echo "There is a problem with file $x, it has $LINES lines"
  fi
done
```

Next, let’s talk about “environment variables.”

```
$ printenv
```

Discuss `$HOME`, `$USER`, `$PWD`, `$LANG`. (Briefly.)

We can use these in scripts...

```
find . -name '*.txt' |
while read x
do
  LINES=`cat $x | wc -l`
  if [ $LINES -ne 1 ]
  then
     echo "Hey $USER, problem with file $x, it has $LINES lines"
     echo "Why don't you go to your home directory, $HOME"
  fi
done
```

We can change or set new environment variables:

```
$ export USER=kevin
```

(Though this is probably a bad idea!)

Next, let’s try this:

```
$ locale
```

We’ll see several special environment variables that determine the “international” behavior of programs that we run at the command-line. Let’s walk through and discuss the impact of `LC_MONEY`, `LC_COLLATE`, etc.

We can see the behavior of `LC_COLLATE` by sorting a file containing the following two lines:

```
tu
tá
```

and seeing that you get different results in a UTF-8 locale and the C locale.

```
$ export LC_ALL=C
$ sort
```

We also set the environment for one command:

```
$ LC_ALL=C sort ...
```

Lecture 14

A bit more “non-linear” processing, using `tee`.

First, we’ll generate a frequency list from the collection of text files (Irish language blog posts) in the subdirectory `irishblogs` of the examples folder.

```
$ cd irishblogs
$ ls
$ ls | wc -l
```

Quick word frequency list:

```
$ cat *.txt | egrep -o '[A-Za-z]+' | sort | uniq -c | sort -r -n |
  sed 's/^ *\([0-9]*\) \(.*\)$/\2 \1/' | head -n 100
```

But, when doing statistical analysis of texts, it’s important to look at how many different texts a given word appears in. So, for example, if we have a corpus of 100 books, the word “Ahab” might appear with very high frequency. But, that’s because it’s common only in *Moby Dick*; so appearing in just one of the 100 books makes it less “important.” How, then, might we add a “column” for the number of different text files each word appears in?

```
$ cat *.txt | egrep -o '[A-Za-z]+' | sort | uniq -c |
  sort -r -n | sed 's/^ *\([0-9]*\) \(.*\)$/\2 \1/' |
  head -n 100 | LC_ALL=C sort -k1,1 | tee TEMP | sed 's/ .*//' |
  while read x
  do
    echo $x `egrep -l "\W$x\W" *.txt | wc -l`
  done | LC_ALL=C join TEMP -
```

Lecture 15
Unicode for Linguists.

Appendix

Excerpt from the Doug McIlroy interview mentioned in Lecture 2:

One place that McIlroy exerted managerial control of Unix was in pushing for pipes. The idea of pipes goes way back. McIlroy began doing macros in the CACM back in 1959 or 1960. Macros involve switching among many data streams. "You're taking in your input, you suddenly come to a macro call, and that says, stop taking input from here, go take it from the definition. In the middle of the definition, you'll find another macro call. Somewhere I talked of a macro processor as a switchyard for data streams. ...In 1964, [according to a] paper hanging on Brian's wall, I talked about screwing together streams like garden hoses."

"On MULTICS, Joe Osanna, ... was actually beginning to build a way to do input-output plumbing. Input-output was interpreted over this idea of the segmented address space in the file system: files were really just segments of the same old address space. Nevertheless, you had to do I/O because all the programming languages did it. And he was making ways of connecting programs together."

While Thompson and Ritchie were laying out their file system, McIlroy was "sketching out how to do data processing by connecting together cascades of processes and looking for a kind of prefix-notation language for connecting processes together."

Over a period from 1970 to 1972, McIlroy suggested proposal after proposal. He recalls the break-through day: "Then one day, I came up with a syntax for the shell that went along with the piping, and Ken said, I'm gonna do it. He was tired of hearing all this stuff." Thompson didn't do exactly what McIlroy had proposed for the pipe system call, but "invented a slightly better one. That finally got changed once more to what we have today. He put pipes into Unix." Thompson also had to change most of the programs, because up until that time, they couldn't take standard input. There wasn't really a need; they all had file arguments. "GREP had a file argument, CAT had a file argument."

The next morning, "we had this orgy of `one liners.' Everybody had a one liner. Look at this, look at that. ...Everybody started putting forth the UNIX philosophy. Write programs that do one thing and do it well. Write programs to work together. Write programs that handle text streams, because that is a universal interface." Those ideas which add up to the tool approach, were there in some unformed way before pipes, but they really came together afterwards. Pipes became the catalyst for this UNIX philosophy. "The tool thing has turned out to be *actually successful. With pipes, many programs could work together, and they could work together at a distance."*