egrep for Linguists

Nikolaj Lindberg
STTS Södermalms talteknologiservice
http://stts.se

Contents

1 Introduction (Zzzz. . .)

The following pages are intended as a starting point for the empirically inclined linguist who wants to make acquaintance with some basic Unix tricks useful in e.g. corpus studies. Everything on the following pages is (perhaps more accurately!) described elsewhere [2, 4, 8, 9]. However, with the exception of [2], the Unix literature is not written for linguists, and the examples of the different commands are often far fetched from a linguistic text processing perspective.

If you are already familiar with the Unix (or Linux) operating system and its basic programs, these pages will not have much to offer you. On the other hand, if you do not know *anything* about Unix (not even things like logging in, starting a text editor or a new terminal window), you should probably pick up these things before continuing — fortunately, these things are easy to learn. It should be noted that this compendium is not a substitute for a Unix textbook; it is rather an attempt to point out the usefulness of a few simple text processing tools.

Many of the text files used in the examples below were found on the Internet. Again and again, a file called `sonnets.txt` is used. It contains the Sonnets of Shakespeare, and was derived from the Complete Works of William Shakespeare, as electronically published by Project Gutenberg [6]. This file and some of the examples have been edited slightly, for example by removing leading blanks. One (mundane) reason for using the Sonnets in the examples is the fact that the lines are short enough to nicely fit the page, while e.g. newspaper text might be harder to present without a lot of editing.

The SUSANNE corpus [11], some 130 000 words of manually analysed written English, is another freely available electronically published text source used in several of the examples. A few times, a file called `newstext` is used in the examples. It contains some 300 000 words of British newspaper text, dumped from a CD-ROM [7]. Another text, `bonk.html`, is simply a HTML file from a (now defunct) WWW site [1].

A shortcoming of the tools presented below, is the fact that the input text material is supposed to be formated in such a way that each record to process is found on one single line in the input file. For more serious text processing, the Ruby programming language [10] is quite useful (but way out of scope for this quick introduction).

Chapter 2 below introduces the `egrep` program, which is used to find strings in text files by means of search patterns, *regular expressions*. Chapters 3 to 8 introduce some simple, very handy Unix commands which all perform some well-defined task, such as sorting the lines of a text file, e.g. in alphabetic order, down-casing upper-chase characters, removing duplicate lines of a sorted file, etc. In Chapter 9, it is shown how these simple commands can be combined into a *pipeline*, a sequence of commands, to perform interesting work. Chapters 10 and 11 introduce yet two other commands, `sed` for doing "search and replace" and the `cat` command, useful for sending the contents of one or more files down a pipeline. Chapter 12 is meant to illustrate how useful the tools presented in previous chapters can be when combined, e.g. to create word frequency lists. Finally, Chapter 13 will show you how to save a frequently used sequence of commands in a *shell file*, and execute this file instead of typing a long sequence of commands over and over again. In the appendix, a list of essential Unix commands is found.

2 egrep

The egrep program is used to scan files for character strings (e.g. words). Its basic function is to go through a text file line for line, and print all *lines* matching a **search pattern** or **regular expression** to 'standard out(put)'. Printing something to standard output often means simply outputting text to the terminal window from which egrep is run, but the result from an egrep command can also be saved in a text file. Each matching line is printed once, even if the search pattern matches more than one part of the line.

There are different versions of egrep, which behave slightly different. The version described below is GNU egrep (version 2.0), but many of the things described should apply to other egreps as well. If your egrep program lacks some of the features described below, the GNU version is freely available [5].

Many of the commands presented in this section can also be executed by the grep program, a predecessor to egrep. If you want to save yourself the typing of an *e*, use grep instead (there are differences between these programs, but by sticking to egrep, the present writer does not have to care about these details).

2.1 Regular expressions unexplained

The regular expressions (search patterns) presented here are not particular to egrep, but have a more general use—e.g. in sed (see Section 10) and in Perl, Ruby, Java and other programming languages. The strange sounding term 'regular expression' is akin to another equally strange term, 'regular language', the set of strings one can describe with the help of regular expressions.[1] However, the formal (mathematical) properties of regular expressions are of no concern in what follows.

For the present purpose, a sloppy 'definition' of regular expression like 'a regular expression is the thing you use in egrep' or 'a regular expression can be used instead of a list of strings' will suffice. As an example of the latter statement, the regular expression `[Ww]ork(s|ing|ed)?` is a shorter way of saying 'Work or work or Works or works or Working or working or Worked or worked'. With a little bit of luck, the reader will get some kind of intuitive picture of what regular expressions are from the following pages.

2.2 Running egrep

In the examples below, an egrep command is typed on a line starting with a dollar sign, and the output from the command is given below. The dollar sign is not typed by the user, but denotes the prompt in the terminal window. (The actual prompt in your terminal window will probably look different.)

An egrep command consists of the regular expression one wants to test on each *line* of a text file, plus the name(s) of the file(s) one wants to search. Thus, to run egrep, one types the word egrep followed by a search pattern followed by one or more file names.

Maybe it should be pointed out that a *line* can be of any length, more or less. In Unix, a line is a sequence of characters ending with a newline character (often denoted by `\n`). This might pose a problem when using such a simple tool as egrep, since the line might be an unsuitable format (e.g. if the lines are very long, or if sentences span several lines, etc).

[1] Curiously enough, there is a whole book devoted solely to the construction of search patterns [4].

2.3 Reporting lines containing your favourite character string

The most basic use of `egrep` is to search for a fixed string of characters, e.g. a word. The following command prints all lines of the text file `sonnets.txt` containing the string `star` to standard output (to the screen, i.e.):

```
$ egrep star sonnets.txt
```

and the output might look something like this:

```
Not from the stars do I my judgement pluck,
And constant stars in them I read such art
Whereon the stars in secret influence comment.
Let those who are in favour with their stars,
Till whatsoever star that guides my moving,
When sparkling stars twire not thou gild'st the even.
Before these bastard signs of fair were born,
And by and by clean starved for a look,
It is the star to every wand'ring bark,
It might for Fortune's bastard be unfathered,
And beauty slandered with a bastard shame,
Nor that full star that ushers in the even
```

where the underlined parts highlight the hits in the above example, which are not actually underlined in the output from `egrep`.

It can be observed that some of the `star`s were really ba`star`ds, as it were, which illustrates the fact that the above command is not sensitive to the context of the string matched. In order to report only lines containing the *word* `star`, there are two possibilities: either you change the way `egrep` behaves by using a special 'switch', or you can use a more complex regular expression. Both methods will be explained presently.

2.4 Match complete words

A **switch**, or **option**, is a character (or string), preceded by a dash, `-`, which modifies the behaviour of a program. `egrep` can take several different switches, some of which will be explained below (pages 12 and 36).

Let us go back to the `star` example above, which output lines which we did not want, if we were looking for the word *star*. By adding the `-w` switch to the `egrep` command we can narrow down the search for `star` to match only full words; the `-w` switch tells `egrep` to behave differently, forcing the program to match only full words and not substrings (parts) of words. The command

```
$ egrep -w star sonnets.txt
```

will result in the output

```
Till whatsoever star that guides my moving,
It is the star to every wand'ring bark,
Nor that full star that ushers in the even
```

This time, we do no longer find `bastard` and `starved` among the hits. However, also the lines containing `stars` are lost. If we are looking for also the plural instances of the word, we need to extend the search pattern somehow.

Unfortunately, the `-w` switch might not work correctly when dealing with words including 'silly' characters such as *å*, *ä*, *ö*, etc (depending on how the computer is configured).

2.5 Two ways of seeing `stars`

We have now seen an example of a simple search pattern and how `egrep` can be modified by adding a switch to the command. More important than the different switches `egrep` recognises are the special symbols, meta characters, used to create complex regular expressions.

Disjunction

There are different ways to match both `star` and `stars` within a single search pattern. The most straightforward way might be to tell `egrep` to look for either the string `star` or the string `stars` with the help of a disjunction, expressed with the vertical bar, `|`. This time the search pattern has to be **quoted** in order for the program to know where the search pattern starts and ends; outside of a regular expression, the vertical bar has a special meaning to the Unix system (see Section 9).

```
$ egrep -w 'star|stars' sonnets.txt
Not from the stars do I my judgement pluck,
And constant stars in them I read such art
Whereon the stars in secret influence comment.
Let those who are in favour with their stars,
Till whatsoever star that guides my moving,
When sparkling stars twire not thou gild'st the even.
It is the star to every wand'ring bark,
Nor that full star that ushers in the even
```

Yet an example of the use of `|` is in order. This time, in order for `egrep` to be able to interpret the regular expression correctly, the scope of the disjunction has to be specified with the help of parentheses. The command

```
$ egrep 'Achilles (heel|tendon)' INFILE
```

matches any lines containing either the string `Achilles heel` or the string `Achilles tendon` (with exactly one space between the words). Without the parentheses, the regular expression would mean something differently (namely. . . ?).

Zero or one of the preceding item

Disjunctions can often be expressed with the help of the vertical bar, but in the *star/stars* case there is a simpler way. Only one character differs in the strings searched for, and this fact can be expressed with the help of a question mark, `?`, a special character which means 'zero or one instance of the preceding item'. If it appears after the `s` in the following way, it means 'look for zero or one instance of `s` at the end of `star`':

```
$ egrep -w stars?  sonnets.txt
```

This command produces exactly the same output as the command using the vertical bar does. Yet an example of the use of `?`: the regular expression `e-?mail` matches both the string `email` and the string `e-mail`.

Parentheses are used for grouping characters in conjunction with the `?` special character just as it is used for grouping characters in a disjunction (*cf* `(heel|tendon)` above). In the following example, which matches either the string `burn` or the string `burning`, the parentheses decide the scope of the `?` meta-character:

```
$ egrep -w 'burn(ing)?'  sonnets.txt
Lifts up his burning head, each under eye
And burn the long-lived phoenix, in her blood,
Nor Mars his sword, nor war's quick fire shall burn:
My most full flame should afterwards burn clearer,
```

By combining `|` and `?`, we can match any of the strings `burn`, `burning`, `burns`, `burned` or `burnt`:

```
$ egrep -w 'burn(ing|s|ed|t)?'  FILENAME
```

We have now encountered a switch (`-w`), which modifies the behaviour of the program, the disjunction (`|`) and one of the 'quantifiers' (`?`), and it has been shown how these things can be used together. The important notion of grouping, with the help of parentheses, has also been exemplified.

Next, a symbol which makes the `-w` switch redundant will be presented.

2.6 End of the word

If one wants to match not a whole word, but the end or beginning of it, the `-w` switch is no good, but there is a way of telling `egrep` that a search pattern should match word boundaries. For example, if one is interested in finding all lines with words ending in *ing*, the command `egrep ing FILENAME`, will of course print any lines containing words ending in *ing*, but also lines containing the words *single*, *things*, *kingdom*, etc. The solution is to use the `\b` symbol, which 'matches the empty string at the edge of a word', which means that it actually does not match any character at all, but rather a position *between* a digit or an alphabetic character and a 'non-word' character, a character which is not a digit or a letter (e.g. a space, period, dash, tab, etc). The `\b` word boundary symbol simply recognises the end or beginning of (what is typically) a word. A 'word' is a string of one or more letters or digits, surrounded by a delimiter such a space character, period, comma, exclamation mark, etc. `\b` matches also at the beginning or end of a line. Thus

```
$ egrep 'ing\b' FILENAME
```

will print only lines containing strings ending in *ing*. By now, you might have realized that e.g. `egrep -w eagle` is equivalent to `egrep '\beagle\b'`.

2.7 Anchoring at the beginning or end of a line

The `^` and `$` symbols have a function somewhat similar to `\b`; they are used to match the *beginning* and the *end* of a line, respectively, but do not match any actual character. Imagine that one wants to find all lines starting with the words *Hate* or *Death* or *Sin*. The `^` symbol can be used to 'anchor' a regular expression at the beginning of a line, and `(Hate|Death|Sin)\b` matches the words looked for (the `\b` is used here to exclude e.g. *Single* or *Hates* from the hits).

```
$ egrep '^(Hate|Death|Sin)\b' sonnets.txt
Sin of self-love possesseth all mine eye,
Death's second self that seals up all in rest.
Hate of my sin, grounded on sinful loving,
```

Note that the regular expression matches *Death* in *Death's*, (since there is a non-word character (*'*) following the string matched).

Likewise, `$` anchors the search patter at the *end* of a line. For example, the command `egrep 'love$'` will extract all lines ending in *love*.

2.8 Character classes

Regular expressions would not be very useful if you could not generalise over sets of characters. For this end, there are pre-defined character set symbols, such as the 'word-character symbol', `\w`, matching characters of which words are typically made up. You can also create your own character set by enumerating a set of characters, such as the vowels, inside square brackets.

Any letter or digit

The `\w` symbol matches any digit or alphabetic character in the range *a-z*, upper and lower case. For example, `'e\we\we'` matches strings of three *e*s with any alphabetic characters between the *e*s:

```
$ egrep 'e\we\we' sonnets.txt
Whose fresh repair if now thou not renewest,
Receiving nought by elements so slow,
Are both with thee, wherever I abide,
For when these quicker elements are gone
Pity me then, and wish I were renewed,
```

The complement of `\w` is written `\W`, and matches a non-alphanumeric character.

Defining your own character sets

`\w` is in fact a synonym for the character set `[A-Za-z0-9]`, where `A-Z` means 'all upper case characters from A to Z', etc (*cf* Section 3). Just as `\w` matches a single alphanumeric character, a set of characters enumerated inside square brackets matches one character of the set; `[aouei]` matches a lower case vowel, and `[aouei][aouei][aouei][aouei][aouei]` matches five adjacent lower case vowels (e.g. *queueing*).

Any character but

'Every character *but*' can be expressed with the help of the caret: `[^aoueiAOUEI]` matches any character *but* the upper or lower case vowels (in other words, it matches not only the rest of the alphabet, but also spaces, tabs, etc).

The caret should occur first in the set, immediately after the opening bracket. As an example, the regular expression `\b[^aoueiyAOUEIY]+\b` might match a word without any vowels (*Bbrrnngg*, *1935*, *Mrs*, *20th*, etc.). The `+` 'repeat character' is presented below.

Do not confuse the `^` used in character classes with the use of the same character for anchoring a regular expression at the beginning of a line (see 2.7). For example, `^[^s]` will match lines beginning with any character except `s`, i.e., the caret does not mean 'beginning of line' inside of square brackets, `[^...]`, but at the beginning of a character class, it means *any character but.*

Any character at all

The dot, `.`, is a very important 'character class', which matches *any character at all.* It will be presented below, in conjunction with the `*` quantifier, since these two are often used together to match any number of any characters.

2.9 'Quantifiers'

The character sets become really useful when used together with the different 'quantifiers', or repetition symbols, one of which has already been presented: `?`. A character, string of characters (enclosed in parentheses), character set, etc, can be repeatedly matched a desired number of times with the help of the `?`, `*`, `+` and `{...}` special characters presented below. These special characters help to make it possible to construct very compact search patterns. However, it should be remembered that a regular expression always matches the longest string possible (a fact which is easy to forget, and which might result in unwanted or surprising results).

One or more

A typical use of the 'one or more' character, `+`, is to match a word (a sequence of one or more alphanumeric characters): `\w+`. For example, lines including 'multi-hyphen-words', *fin-de-siecle*, *down-to-earth*, etc., can be found with the help of the command

```
$ egrep '\w-\w+-\w' INFILE
```

The plus sign means 'one or more instances of the preceding item', so `\w+` means 'a sequence of one or more alphabetic characters or numbers' (a word, i.e.). The `-` matches the hyphen. Since the pattern starts and ends with a single instance of `\w`, the regular expression matches just part of the 'multi-hyphen words', e.g. the underlined part of *f<u>in-de-s</u>iecle.* In order to match the whole thing (useful in some cases), `'\w+(-\w+)+-\w+'` can be used, where the third `+` 'quantifies' the subexpression inside parentheses, and could be paraphrased with something like 'one or more instances of `-\w+`' (i.e. at least one *hyphen-word* sequence).

Notice that both of the regular expressions above match not only e.g. *fin-de-siecle*, *hand-in-hand*, *down-to-earth*, etc., but also e.g. *1-2-5* since `\w` matches a letter or a digit. Since `(-\w+)+` matches any number of *hyphen-word* sequences, not only strings of three components will be found, but also e.g. *two-for-the-price-of-one*. To only match a sequence of three words delimited by hyphens, use `\w+-\w+-\w+`.

Any number of times

Let's go back to the *-ing* example: `'ing\b'` matches e.g. *thing*, *sing*, *ring*, etc. If one looks for words with an *-ing* inflection, these hits are not desired. One can narrow down the search by only looking for words ending in *ing*, where at least one vowel (including *y*) precedes the ending. The set of vowels is defined by enumerating them inside square brackets: `[aoueiy]`. The regular expression `'[aoueiy]ing\b'` matches a vowel followed by *ing*, e.g. *being*, *unseeing*, *going*, etc. In order for the regular expression to match verbs of the *-ing* form in which a consonant precedes the ending (*singing*, etc), the `\w` symbol could be used to match a consonant, since it matches any one alphabetic character. The desired regular expression will match any string ending in *ing*, preceded by some characters of which at least one is a vowel. The regular expression `'[aoueiy]\wing\b'` matches strings where *-ing* is preceded by any alphabetic character, preceded by a vowel. However, this will exclude *being* from the hits (why?). The problem is solved by using the star, `*`, which matches 'zero or any number of the preceding item'. The regular expression is modified to match zero or any number of alphabetic characters between the vowel and the ending:

```
$ egrep '[aoueiy]\w*ing\b' sonnets.txt
Making a famine where abundance lies,
And tender churl mak'st waste in niggarding:
Then being asked, where all thy beauty lies,
Were an all-eating shame, and thriftless praise.
Proving his beauty by succession thine.
Nature's bequest gives nothing but doth lend,
And being frank she lends to those are free:
...
```

The underlining is there to stress the fact that the regular expression does not match the whole words. To match complete words, something like `'\b\w*[aoueiy]\w*ing\b'` could be tried. As can be seen, most of the lines seem to contain what was looked for, but also *nothing* appears—as would also e.g. *darling*.

You can actually live without the `+` quantifier presented above, since it can be simulated by using the the star; `\w\w*` matches exactly the same thing as `\w+` does.

Any number of any characters

The dot, `.`, is used to match any character (except newline). For example, `egrep e.e.e.e` matches any line in which there are four *e*s intervened by any other character:

```
$ egrep 'e.e.e.e' sonnets.txt
The lovely gaze where every eye doth dwell
```

As can be seen, `.` matches also the space.

To find lines in which two specific words are repeated, possibly with other words in between, the period can be used to match every character in between the words. For example, instances of '*as... as*' are matched by the regular expression `'\bas\b.*\bas\b'`:

```
$ egrep '\bas\b.*\bas\b' sonnets.txt
And die as fast as they see others grow,
Thou art as fair in knowledge as in hue,
Thou art as tyrannous, so as thou art,
And sealed false bonds of love as oft as mine,
Who art as black as hell, as dark as night.
```

As explained above, the star, `*`, means 'zero or more instances of the preceding item', and when put after the period, it means 'zero or more instances of any character'.

The underlining of the last line in the example output above illustrates the important fact that `*` is *greedy*; it matches the longest string possible. As already pointed out, a regular expression matches the longest possible string.

An exact number of times

Extending the `egrep` command with the `-E` switch makes it possible to state exactly how many times the preceding item should be matched.

```
$ egrep -E '[AOUEIaouei]{5}' INFILE
```

matches five consecutive upper or lower case vowels. If `-E` is omitted, the `{5}` sequence matches a left curly brace, the digit *5* and a right curly brace.

With the help of the `-E`(xtended) regular expressions, one can also search for e.g. *between zero and four* instances of the preceding item. An example: in order to find all examples of *sources... said* where the dots represent between zero and four intervening words, the following command could be used:[2]

```
$ egrep -E '\b[Ss]ources (\w+ ){0,4}said\b' newstext
```

which matches e.g. the following

```
yesterday, Whitehall sources said the Government may be forced to sus
          Leadership sources said last night the new initiative would
  British diplomatic sources in Paris said the joint flypast is inten
   Senior Tory Party sources said there were practical difficulties
                     Sources close to Hizbollah said in Beirut last n
                     ...
```

The character set `[Ss]` is used to match both *Sources* and *sources.* Presently, a switch which makes `egrep` ignore the upper/lower-case distinction will be presented.

`{4,}` matches the preceding item repeated four times or more; `{,4}` matches no more than four times (thus, `{0,4}` and `{,4}` mean the same thing).

[2]The file `newstext` contains a sample from a British newspaper corpus [7] (and the output above is edited to be easier to read).

2.10 Back referencing

In the *as... as* example above, the same string (`as`) was matched twice. There is a mechanism for **back-referencing** which can be used instead of repeating parts of a regular expression. To find all lines in which the word *nothing* occurs at least twice, the regular expression `'\b(nothing)\b.*\b\1\b'` can be used. The `\1` symbol is a back-reference to the string matched by the part of the regular expression inside parentheses:

```
$ egrep '\b(nothing)\b.*\b\1\b' sonnets.txt
To me are nothing novel, nothing strange,
```

If there are more than one pair of parentheses, `\2` matches the second pair, etc. (If parentheses are embedded, `\2` will refer the to second pair of parenthesis from the left, etc).

There is a subset of the 'multi-hyphen' word sequences (*cf* Section 2.8) in which one of the words is repeated, as e.g. in *hand-in-hand*, *arm-in-arm*, *door-to-door*, etc. Lines including these can be extracted thus:

```
$ egrep '\b(\w+)-\w+-\1' INFILE
```

Notice that `\b` is necessary; otherwise the search pattern will match lines containing a *word-hyphen-word-hyphen-word* sequence where the last character of the first word is the same as the first character of the third word—the `(\w+)` part will match only a single character (e.g. in *tit-for-tat*). (The `\b` can be substituted for the `-w` option: `egrep -w '(\w+)-\w+-\1'`.)

The parentheses are used *both* for grouping (see Section 2.5) and back reference.

2.11 More switches

Ignore upper/lower case

Most of the examples above will only match lower-case characters, `'[aoueiy]\w*ing\b'` does not match SEEING, etc. One could of course enumerate also the upper-case vowels and add `ING` to the search pattern, but it is much simpler to just add the `-i` switch (`i`gnore upper/lower case distinctions):

```
$ egrep -i '[aoueiy]\w*ing\b' sonnets.txt
```

which will match e.g. *SEEING*, *seeing*, *NOTHING*, or even *NOtHiNg*, etc. Likewise, `egrep -iw 'gods?'` matches lines in which *God*, *gods*, etc, occur. (If you do not remember what `?` means, see Section 2.5.)

Count numbers of lines

Sometimes one wants to know in how many lines a pattern is found, rather than see the actual lines. By using the `-c` switch, `egrep` reports the number of lines matched:

```
$ egrep -ic '[aoueiy]\w*ing\b' sonnets.txt
309
```

where `309` is the number of lines in which `'[aoueiy]\w*ing\b'` was found. Notice that it is the number of *lines* in which a pattern is found, not necessarily the number of *times* the pattern was found, since the same string can occur more than once on a single line. Thus `-c` should not be used to count e.g. the number of occurrences of a word (unless, of course, there is only one word on each line!).

Report lines *not* matching

The lines in the input file which did *not* match the search pattern are found with the help of the `-v` switch. This switch tells `egrep` to print those lines not matching the regular expression, and in combination with the `-c` switch, the number of lines in which the pattern was *not* found will be printed:

```
$ egrep -icv '[aoueiy]\w*ing\b' sonnets.txt
2317
```

As can be seen, the switches are just appended after the `-` in any order.

Print context

Sometimes it is handy to look also at at few lines before and/or after a line matched by a regular expression. For example, to print two lines before and after a matching line, `-2` can be used:

```
$ egrep -2 Death sonnets.tst
As after sunset fadeth in the west,
Which by and by black night doth take away,
Death's second self that seals up all in rest.
In me thou seest the glowing of such fire,
That on the ashes of his youth doth lie,
```

To print some preceding or following context, the `-B` or `-A` switches are used:

```
$ egrep -B2 Death sonnets.tst
As after sunset fadeth in the west,
Which by and by black night doth take away,
Death's second self that seals up all in rest.
```

```
$ egrep -A2 Death sonnets.tst
Death's second self that seals up all in rest.
In me thou seest the glowing of such fire,
That on the ashes of his youth doth lie,
```

2.12 Match characters with special meaning

Some characters have a special meaning when they occur in a regular expression. For example, parentheses are used for grouping (see e.g. `'Achilles (heel|tendon)'` in Section 2.5 above). To find lines with parentheses in them, the backslash, `\`, is used to remove the special meaning of the parenthesis: `\(` means 'match a left parenthesis':

```
$ egrep '\(' sonnets.txt
The eyes (fore duteous) now converted are
Which this (Time's pencil) or my pupil pen
```

```
So should my papers (yellowed with their age)
In thy soul's thought (all naked) will bestow it:
For then my thoughts (from far where I abide)
Which like a jewel (hung in ghastly night)
And each (though enemies to either's reign)
(Like to the lark at break of day arising
Then can I drown an eye (unused to flow)
But if the while I think on thee (dear friend)
...
```

To match any of the characters of special meaning to `egrep`, prefix it with a backslash. These characters include `)`, `|`, `*`, `.`, `\`, `?` and `+`. When the `-E` option is used, also `{` and `}` need to be prefixed with a backslash in order to escape their special meaning.

So far, single quotes have been used to delimit regular expressions which should otherwise confuse the Unix system (e.g. `'Achilles (heel|tendon)'`). If one wants to match a single quote, `'''` will not work, since the shell (the program which interprets and executes commands) reacts as if the single quote one wants to match closes the fist single quote, and then there is one left (and there will be an error). Instead, double quotes can be used: `"'"`.

If a search pattern **starts with a hyphen**, e.g. one used to find words ending with *-like* (*flamingo-like*, *hawk-like*, *umbrella-like*, etc), the hyphen needs to be prefixed by a backslash: `'\-like'`. If the hyphen appears anywhere else but first in the regular expression it does not need to be backslashed.

`man egrep`

Try the command `man egrep` (or `man grep`); it will present the man(ual) page for `egrep`. It tells everything explained above and more, in fewer and even more incomprehensible words. (Type `q` to quit the man-page.)

Useful GNU `egrep` switches	
`-1, -2, ...`	print one, two, etc, lines of context before and after the match
`-c`	count matching lines
`-f`	read search patterns from a file instead of from the command line
`-h`	do not print the file names in front of the matching lines when searching more than one file
`-i`	ignore upper/lower case distinctions
`-n`	print also the line number of the match
`-v`	print the inverted result (all lines *not* matching)
`-w`	match full words
`-x`	the regular expression should match the whole line, and not only part of it
`-E`	extend the regular expressions to interpret `{5}` as *repeat 5 times*, etc

Useful symbols in GNU `egrep` regular expressions	
`\1, \2, ...`	match expression inside first pair of parentheses, inside second pair, etc
`\b`	word boundary
`\w`	number, alphabetic character (same as `[A-Za-z0-9]`)
`\W`	the opposite to `\w`
`^`	beginning of line
`$`	end of line
`|`	disjunction
`?`	zero or one (of the preceding item)
`*`	zero or any number
`+`	one or more
`.`	any character (excluding newline)
`\`	change meaning of special characters (e.g. `\\b` means ‘a backslash followed by `b`’, and not ‘word boundary’)
`[...]`	character set (e.g. the vowels: `[auoei]`)
`[^...]`	any character *but* (e.g. anything but the vowels: `[^auoei]`)
`{n}, {m,n}`	repeat a specified number of times (requires the `-E` switch)

3 tr

Sometimes it is useful to be able to put all words in a text file on a line of their own, e.g. when 'egreping' for words of some special properties, or when one wants to produce a frequency list.

A simple way to turn a text file into a list of words, one word on each line, is to turn every space in the text into a newline character. The `tr` program 'translates' characters, and can be used for translating every space in a text into a newline character. The command `tr ' ' '\012' < INFILE` turns the text in `INFILE` into a list of words. The `<` symbol is the Unix way to tell `tr` to read its input from a file. The `' '` part means 'a space' and `'\012'` is a code for the newline character. Some versions of `tr` accept `'\n'` instead of `'\012'`.

Likewise, `tr 'A-Z' 'a-z' < INFILE` translates all uppercase letters in INFILE into the lower-case equivalents. `A-Z` is a simpler way of saying `ABCDEFGHIJKLMNOPQRSTUVWXYZ`, etc (the same thing goes for digits: `0-9` is the same as `0123456789`). `tr 'a-z' 'A-Z'` turns all lower case characters into upper case.

If the file `one_line.txt` has the following contents

```
Kilgore Trout owned a parakeet named Bill
```

the command `tr ' ' '\012' < one_line.txt` yields

```
Kilgore
Trout
owned
a
parakeet
named
Bill
```

and the command `tr 'A-Z' 'a-z' < one_line.txt` results in

```
kilgore trout owned a parakeet named bill
```

The command `tr ' ' '\012' < INFILE` above will result in empty lines if there are multiple spaces in the text, since each space will be translated into a newline character. The following command turns a text file into a word list, similar to the example above, but removes multiple spaces, and other non-alphanumeric characters.

```
$ tr -cs '[a-zA-Z0-9]' '\012' < INFILE
```

The `-c` switch means 'translate all characters *not* in the character set' (`[a-zA-Z0-9]` in this case). The `-s` switch is used to 'squeeze' repeated characters, e.g. `tr -s ' ' < INFILE` replaces sequences of repeated spaces with one space. Unfortunately, different implementations of `tr` can behave differently. In some, `tr -cs '[a-zA-Z0-9]' '[\012*]' < INFILE` should be used (check the `man` page by typing `man tr`).

Since all characters not enumerated in the character set will be lost (exchanged for newlines), the two versions will treat e.g. *she'll* or *out-of-touch* differently. (In the first case, only spaces are changed into newlines, whereas in the second case, e.g. `'` or `-` will also be substituted for a newline.)

When turning upper case characters into lower case for another alphabet than *a-z*, e.g. an alphabet including *å*, *ä*, *ö*, the additional characters are enumerated inside square brackets (*cf* Section 2.8)

```
$ tr '[A-ZÅÄÖ]' '[a-zåäö]' < INFILE
```

where the additional characters must appear in the same order in both sets.

There are also predefined symbol sets. A different way of turning uppercase characters into lower case is to use `[:upper:]` and `[:lower:]`:

```
$ tr '[:upper:]' '[:lower:]' < INFILE
```

which should also take care of e.g. *å*, *ä*, *ö* if your machine can handle these.

The lion-hearted reader is referred to `man tr` for more details.

In all examples in the compendium, the arguments to e.g. `tr` are inside single quotes. Depending on which command line interpreter ('shell') you use, you might be able to leave out the single quote characters, saving yourself a few key-strokes.

4 sort

The `sort` program sorts the lines of a file. If the file contains a single word on each line, the result will be a word list in alphabetic order. If the lines of a file starts with a number, the command `sort -n INFILE` will result in a list sorted numerically. In other words, the commands `sort` and `sort -n` will produce different results for e.g. the following file (the `more` command is used to display the contents of a text file, one screenful at a time):

```
$ more file.txt
1 small onion, skinned and finely chopped
65 g (2 1/2 oz) freshly grated Parmesan cheese
600 ml (20 fl oz) Bechamel sauce
115 g (4 oz) plain flour
```

In the first case below, lines are treated as character strings, sorted in ASCII order: the line starting '`600 ml ...`' will appear before the one starting '`65 g ...`', etc (since the character 0 has a lower ASCII number than 5 has). In the second example (`-n`), the digits at the beginning of the lines are treated as numbers.

```
$ sort file.txt
1 small onion, skinned and finely chopped
115 g (4 oz) plain flour
600 ml (20 fl oz) Bechamel sauce
65 g (2 1/2 oz) freshly grated Parmesan cheese

$ sort -n file.txt
1 small onion, skinned and finely chopped
65 g (2 1/2 oz) freshly grated Parmesan cheese
115 g (4 oz) plain flour
600 ml (20 fl oz) Bechamel sauce
```

By adding the `-r` switch, the output is presented in `r`everse order:

```
$ sort -nr file.txt
600 ml (20 fl oz) Bechamel sauce
115 g (4 oz) plain flour
65 g (2 1/2 oz) freshly grated Parmesan cheese
1 small onion, skinned and finely chopped
```

If one has a text file with different **fields** on each line, the file can be sorted with one of the fields as the key. The file `freq_list` contains (the top of) a frequency list created from some text file, and there are two fields on each line, separated by a space. The first field contains the number of occurrences of a word, and in the second field the word is found.

```
$ more freq_list
1642 the
872 and
```

```
729 to
632 a
595 it
552 she
545 i
513 of
462 said
411 you
398 alice
```

The command `sort +1` will sort the file in ASCII order with the *second* field as key field.

```
$ sort +1 freq_list
632 a
398 alice
872 and
545 i
595 it
513 of
462 said
552 she
1642 the
729 to
411 you
```

Finally, the `-f` switch makes `sort` ignore the difference between upper and lower case characters (it '`folds`' lower case to upper case). If the switch is left out, `sort` will put e.g. *Zoroaster* before *asparagus*.

`man sort` reveals a host of different options.

5 uniq

Though it looks misspelt, `uniq` is used to remove duplicate lines from a **sorted** file. With the `-c` switch, `uniq` removes duplicate lines but counts the number of times a line occurs. Given a sorted file with the following contents

```
$ more sorted_file
are
are
are
are
argue
argued
argument
argument
argument
argument
arguments
arithmetic
arm
arm
arm
arm
arm
arm
arm
```

a `uniq` command can be used to produce a frequency list:

```
$ uniq -c sorted_file
4 are
1 argue
1 argued
4 argument
1 arguments
1 arithmetic
7 arm
```

Combined with `sort`, `uniq` can be used to produce frequency lists with one single, wonderful *pipeline* of commands (see Section 12.1).

6 paste

If two files containing a single column of words each are `paste`d together, the result will be a file with two columns. If the file `file_a` contains the words

```
$ more file_a
The
weapon
the
pirates
however
was
capacity
astonish
```

and file `file_b` contains

```
$ more file_b
chief
of
sea
,
,
their
to
.
```

the two files can be merged with the command

```
$ paste file_a file_b
The       chief
weapon    of
the       sea
pirates   ,
however   ,
was       their
capacity  to
astonish  .
```

`paste` can be used e.g. for producing lists or two-word sequences, bigrams, explained in Section 12.2. (It can also be handy for producing tables.)

7 tail and head

The commands `tail` and `head` are used to print a number of lines from the end and beginning of a file, respectively. `tail -40 INFILE` prints the last 40 lines of the file `INFILE`, while `tail +40 INFILE` prints the rest of the file from line number 40; `tail +2` prints the whole file except for the first line, and so on. The default number of lines is 10: `tail INFILE` prints the last 10 lines of `INFILE`, and `head INFILE` prints the first 10 lines.

The file `J10` is 2 381 lines long and contains a part of the SUSANNE corpus[3]. The command `head -18` prints the first 18 lines of the document

```
$ head -18 J10
J10:0010a  -  YB     <minbrk>     -                    [Oh.Oh]
J10:0010b  -  II21   Apart        apart                [O[S[P:m[II=.
J10:0010c  -  II22   from         from                 .II=]
J10:0010d  -  AT     the          the                  [Ns.
J10:0010e  -  NN1c   honeybee     honeybee             .Ns]P:m]
J10:0010f  -  YC     +,           -                    .
J10:0010g  -  RR     practically  practically          [Np:s[D.
J10:0010h  -  DBa    all          all                  .D]
J10:0010i  -  NN2    bees         bee                  [NN2&.
J10:0010j  -  CC     and          and                  [NN2+.
J10:0010k  -  NN2    bumblebees   bumble<hyphen>bee    .NN2+]NN2&]Np:s]
J10:0020a  -  VV0i   hibernate    hibernate            [V.V]
J10:0020b  -  II     in           in                   [P:h.
J10:0020c  -  AT1    a            a                    [Ns.
J10:0020d  -  NNL1n  state        state                .
J10:0020e  -  IO     of           of                   [Po.
J10:0020f  -  NN1n   torpor       torpor               .Po]Ns]P:h]S]
J10:0020g  -  YF     +.           -                    .
```

Adding the 'save in file' character, `>`, and a file name, e.g. `J10_excerpt`, creates a little file containing the first 18 lines of the `J10` document

```
$ head -18 J10 > J10_excerpt
```

More on *redirecting output* is found in Section 9.

[3]The entire corpus is freely availably on the Internet [12]. The annotation scheme of the corpus is described in [11].

8 cut

In the SUSANNE corpus the columns (fields) are separated by tabs. If one wants to print only one or a few of the fields, the `cut` command can be useful. The file `J10_excerpt` contains a tiny sample of the corpus (see Section 7). To excerpt the fourth field, for example, the following command can be used:

```
$ cut -f4 J10_excerpt
<minbrk>
Apart
from
the
honeybee
+,
practically
all
bees
and
bumblebees
hibernate
in
a
state
of
torpor
+.
```

In a similar manner, if one is interested in the third to the fourth field of the file, the following command is given:

```
$ cut -f3-4 J10_excerpt

YB      <minbrk>
II21    Apart
II22    from
AT      the
NN1c    honeybee
YC      +,
RR      practically
DBa     all
NN2     bees
CC      and
NN2     bumblebees
VV0i    hibernate
II      in
AT1     a
NNL1n   state
IO      of
NN1n    torpor
YF      +.
```

If a different character than the tab is used to `d`elimit the fields, the `cut` command is told so with the help of the `-d` switch. The command `cut -f1,5 -d' ' FILE` would print fields one and five from the file `FILE`, in which the fields are separated by spaces.

9 Redirecting and pipelining

The seasoned Unix user probably feels hampered when confronted with a operating system not featuring pipelines, the ability to build complex commands out of simple ones. On the other hand, the Unix beginner might be intimidated by the multitude if cryptic commands, and the fact that typically one has to type them on the command line rather than click on a friendly icon. However, the written word, in this case in form of the commands one types on the command line, is far more expressive than the language of icons. In this chapter, a few simple examples of pipelining are given, while subsequent chapters will present more and more complex ones.

9.1 Redirect to file

The `<` character tells for example the `tr` program from where it should take its input (see Section 3, page 16). The `>` character is used to save the output from a command in a file. For example, the command `tr ' ' '\012' < INFILE > OUTFILE` substitutes all spaces in the file `INFILE` for newline characters and prints the result to the file `OUTFILE` (the contents of `INFILE` is not changed in any way). If the file `OUTFILE` does not exist, it will be created, and if it already exists, its contents will be overwritten (this last point is not always true; in some Unix configurations, one has to use the symbol `>!` to over-write an existing file). If one wants to append the output from a program to the end of an existing file rather than over-write the file, the `>>` characters are used.

Here follows a simple (but quite useful) example of how `>` can be used to redirect the output from a program to a file. The `echo` program takes a string as its argument and outputs this string to standard output—it simply "echoes" a string:

```
$ echo I will only say this once
I will only say this once
```

A file containing a single line of text can be created by `echo`ing a string and redirect it to a file. The command

```
$ echo Kilgore Trout owned a parakeet named Bill > one_line.txt
```

creates the file `one_line.txt` in which the text between the `echo` command and the `>` symbol is found. (Try the above command, and view the contents of `one_line.txt` with the help of `more` or `cat`.)

9.2 Pipelines

One of the most useful properties of Unix, is the ability to redirect, or 'pipe', output from one program to be used as input to a second program. The vertical bar, `|`, is used to pipe output through commands.[4] Below, a few examples of **pipelines** are given—more examples are found in Section 12.

When producing a word list from a text file, it can be wise to turn all upper-case characters into lower-case; if the word list should be used to create a frequency list, one does probably not want different counts for *My* and *my*, etc. The following command reads the contents of the file `one_line.txt`, changes all upper-case characters into the lower-case equivalents,

[4]This use of the vertical bar should not be confused with how it is used in regular expressions.

and pipes the result through a command which translates all spaces in the text into newline characters. The result is a list of lower-cased words.

```
$ tr 'A-Z' 'a-z' < one_line.txt | tr ' ' '\012'
kilgore
trout
owned
a
parakeet
named
bill
```

To get the word list sorted in ASCII order, it is a good idea to pipe the output from the command above to the `sort` program:

```
$ tr 'A-Z' 'a-z' < one_line.txt | tr ' ' '\012' | sort
a
bill
kilgore
named
owned
parakeet
trout
```

If the result should be saved in a file, the `>` symbol followed by a file name, e.g. `sort_list`, is added:

```
$ tr 'A-Z' 'a-z' < one_line.txt | tr ' ' '\012' | sort > sort_list
```

Sometimes it is handy to be able to send a text string into a pipeline without reading the input from a file:

```
$ echo 'I like to shout.'  | tr '[a-z.]'  '[A-Z!]'
I LIKE TO SHOUT!
```

Notice how cleverly the period is translated into an exclamation mark.

10 sed

sed can be used to 'find and replace' strings, and the strings to substitute can be formulated as regular expressions. In the following example, in the file manswrld, the word man is substituted for woman. (cat is used to view the contents of the file.)

```
$ cat manswrld
It's a man's man's man's world.
```

To substitute a string, s/.../.../ is used. The first example shows that only the first matching string on a line is substituted:

```
$ sed s/man/woman/ manswrld
It's a woman's man's man's world.
```

By adding g at the end of the replacement operator, the matched string is replaced globally:

```
$ sed s/man/woman/g manswrld
It's a woman's woman's woman's world.
```

In contrast to egrep, the standard behaviour of sed is to print all input lines, also those not matching a regular expression. This means that the lines in which no strings have been substituted by a substitution command (such as the one above) will also appear in the output, together with those lines where strings have been substituted. Here follows an example of a regular expression, used to remove the mark-up tags from a HTML encoded file. It can be used for turning HTML files (e.g. home pages on the WWW) into plain text files. HTML tags start with a < character and end with >. Consider a file bonk.html in HTML format [1].

```
<HTML>
<HEAD>
<TITLE>The Home Page of Bonk Business</TITLE></HEAD><BODY>
<CENTER>
<IMG SRC="etusivu.gif" ALIGN="MIDDLE">
</CENTER>
<H1><CENTER><A HREF="bonk.mpg">
Welcome!</A></H1></CENTER>
<H2>
Introduction by Alvar Gullichsen Bsc(CT), Head of Product Development BBI.</H2>
<P>
"There is nowhere to be but up".  These words of wisdom from Pär Bonk, architect of global recovery
from the Great Depression, should be remembered as we clamber dazed from the wreckage of the
20th Century.  We live in exciting times, a handful of years from the celebration of a new millennium.
Bonk Business Inc. stands poised on the axle of time for a quantum leap into the future.

We are proud of our heritage, working hard for today, and infinitely curious about the future.
Join us on our journey.  Here at Bonk Business Inc. we regard laughter as one of our most profitable
assets.  Laughter penetrates that last difficult five centimetres of the communication process
like anchovy oil.

That is our message.  Enjoy!  <P>

<CENTER>

<IMG SRC="a1893.gif" ALIGN="MIDDLE">

</CENTER>

...
```

In order to remove the HTML tags (e.g. for preprocessing a text before producing a word frequency list), character string starting `<` and ending `>` is replaced by 'the empty string'—nothing at all, i.e.:
`sed 's/<[^<]*>//g' bonk.html`

```
The Home Page of Bonk Business

Welcome!

Introduction by Alvar Gullichsen Bsc(CT), Head of Product Development BBI.

"There is nowhere to be but up".  These words of wisdom from Pär Bonk, architect of global recovery
from the Great Depression, should be remembered as we clamber dazed from the wreckage of the
20th Century.  We live in exciting times, a handful of years from the celebration of a new millennium.
Bonk Business Inc. stands poised on the axle of time for a quantum leap into the future.  We
are proud of our heritage, working hard for today, and infinitely curious about the future.  Join
us on our journey.  Here at Bonk Business Inc. we regard laughter as one of our most profitable
assets.  Laughter penetrates that last difficult five centimetres of the communication process
like anchovy oil.  That is our message.  Enjoy!
...
```

As pointed out on page 11, `*` is greedy and matches as much as it can: `<.*>` matches all of
`<LI><A HREF="today.html">Bonk Today</A></LI>` while `<[^<]*>` matches only the underlined parts: `<LI>` `<A HREF="today.html">`Bonk Today`</A>` `</LI>`. `<[^<]*>` matches any string beginning `<` and ending `>` with any number of any characters *except* `<` in between. This is the standard solution to make `*` 'non-greedy'. (Please note that the above example is not a fool-proof way of removing HTML mark-up, and will not always be adequate.)

As in `egrep`, back referencing is possible, and it can be used in the string which replaces one matched by the regular expression in a substitution. In some of the lines of Shakespeare's sonnets, an expression inside parentheses is found. Let us assume that one for some (obscure) reason is interested in investigating just the words appearing inside parentheses (given that there is *one* opening and *one* closing parenthesis only on a single line—strings inside parentheses spanning two or more lines will not be found, and lines with more than one pair of parentheses might also be treated incorrectly). As a first step, a file containing only the interesting lines, `parenlines` is created with the help of `egrep`:

```
$ egrep '\(.*\)' sonnets.txt > parenlines
```

The file `parenlines` now includes all lines in `sonnets.txt` which have a `(` followed by any characters followed by a `)` in them.[5] The use of the `>` to create an output file is explained in Sections 3 and 9. Let us take a look at the first 10 lines of the new file (`head` prints the first 10 lines of its input, see Section 7):

```
$ head parenlines
The eyes (fore duteous) now converted are
Which this (Time's pencil) or my pupil pen
```

[5]The command `egrep -c '\(.*\)' sonnets.txt` will report the fact that there are 42 lines matching this description in the Sonnets.

```
So should my papers (yellowed with their age)
In thy soul’s thought (all naked) will bestow it:
For then my thoughts (from far where I abide)
Which like a jewel (hung in ghastly night)
And each (though enemies to either’s reign)
Then can I drown an eye (unused to flow)
But if the while I think on thee (dear friend)
And thou (all they) hast all the all of me.
```

To pick out just the pieces one wants, the text on either side of the parentheses is removed:

```
$ sed 's/.*(\(.*\)).*/\1/' parenlines | head
fore duteous
Time’s pencil
yellowed with their age
all naked
from far where I abide
hung in ghastly night
though enemies to either’s reign
unused to flow
dear friend
all they
```

Of course, these commands could be written as a single pipeline, without the need of the temporary file `parenlines`:

```
$ egrep '\(.*\)' sonnets.txt | sed 's/.*(\(.*\)).*/\1/'
```

Actually, one could use `sed`’s `-n` option together with the `p` flag right away, only outputting lines matching the regular expression:

```
$ sed -n 's/.*(\(.*\)).*/\1/p' sonnets.txt
```

If one considers the regular expression `.*(\(.*\)).*` from an `egrep` perspective, the fact that the parentheses matched in the input text do not appear in the output should be puzzling. The answer to the mystery is that in `egrep`, a parenthesis *without* a backslash has a special meaning (back-reference), while a parenthesis with a backslash in front of it loses its special meaning. In `sed` it works the other way around.

The two `.*` sequences at the beginning and end of the regular expression are there to make sure that all of the line is matched. This way, the whole line is substituted for the part of it which is matched by the regular expression inside the back-reference parentheses.

11 cat

`cat` is used to print files to standard output, and can be used to con`cat`enate files by applying it to several files and redirecting the output to a single file. For example, `cat fileA fileB fileC` prints the contents of `fileA`, `fileB` and `fileC` in that order, and by adding `> OUTFILE` to the command, the contents of all three files will be printed to `OUTFILE`.

In combination with the Unix wild-card characters, `?` and `*`, `cat` can be used to send the contents of several files into a pipeline. For example,

```
$ cat * | tr 'A-Z' 'a-z' > low
```

turns all upper case characters in all the files of the current directory to lower case, and the result is printed to the file `low`.

Lastly, yet another typical example of a pipeline using `cat` to send the contents of several files down a pipeline, as is if the different files were really one huge file. A command like `egrep -c bumblebee *` reports the number of times the search pattern matches in *each file* in the current directory. To get the total numbers of matches in all of the files in the current directory, the following pipeline can be tried:

```
$ cat * | egrep -c bumblebee
```

Be careful when using the star like this, since it matches *any* file in the current directory, also e.g. back-up files which are sometimes produced when editing files (e.g. those recognised by the tilde, ~, at the end of the file name). If all the files you want to send down a pipeline end with the same extension, e.g. the file extension `.txt`, it can be suffixed to the star accordingly: `cat *.txt`. Meow.

Huge text files are often stored in a compressed format. If one or more files are compressed using the `gzip` program, the `zcat` program will print the contents of the `gzip`ed files just the same way as `cat` print the contents of a normal, uncompressed filed. In other words, by using `zcat`, one needs not uncompress compressed files before sending them into a pipeline.

12 Examples of (more or less forbidding) pipelines

In the following sections, a few examples of how complex pipelines can be built up from the commands presented in previous sections are given. Though some of the examples look quite terrible, it is not that bad if you build them up step by step, testing each part of the pipeline, inspecting its output, before adding more commands to it.

In some of the examples, temporary files for storing intermediate results are used.

12.1 Word frequency lists

It is easy to produce word frequency lists by combining the programs presented in previous sections. If the text file does not contain sequences of multiple spaces, tabs, etc, the following series of commands (pipeline) will produce a frequency list, with the most frequent words at the top of the list:

```
$ tr 'A-Z' 'a-z'<TEXTFILE|tr ' ' '\012'|sort|uniq -c|sort -rn
```

To save the result in a file, `> OUTPUTFILE` is added at the end of the pipeline, where `OUTPUTFILE` is the name one wants the file to have. To inspect the frequency list, rather than to save it in a file, the output might be piped to the `more` or `head` program instead :

```
$ tr 'A-Z' 'a-z'<TEXTFILE|tr ' ' '\012'|sort|uniq -c|sort -rn|more
```

(The two above examples also show that no spaces are needed to delimit the parts of a pipeline.)

Instead of reading the contents from a file with the help of `<`, the `cat` program can be used: `cat TEXTFILE | tr 'A-Z' 'a-z' | tr ' ' '\012' |....`

Let us return to the SUSANNE corpus, exemplified above (page 22). The following pipeline, in which `cut -f5` picks out field 5 from all the corpus files, can be used to produce a list of the 15 most frequent lemma forms (look-up head word forms) of the corpus:

```
$ cut -f5 ???|sort|uniq -c|sort -rn|head -15
28377 -
9641 the
4883 be
4702 of
3375 and
3163 to
3038 a
2854 in
1703 he
1429 have
1335 that
1147 for
1117 it
 911 as
 895 with
```

All SUSANNE files have names three characters long (e.g. `J10`). The three question marks, `???`, denote all files in the current directory with a three-character name.

If one wants different counts for words with different morphological analyses (e.g. *wind* (noun) and *wind* (verb)), the following pipeline, in which `cut` extracts the wanted fields (3 and 5) of the corpus files, can be useful:

```
$ cut -f3,5 ??? |sort|uniq -c|sort -rn|head -15
9616  AT      the
6880  YC      -
6584  YF      -
4466  IO      of
3368  CC      and
3070  YG      -
2951  AT1     a
2648  II      in
1968  YB      -
1794  TO      to
1374  VBZ     be
1368  PPHS1m  he
1295  IIt     to
1233  VBDZ    be
1180  YIL     -
```

Once again we return to the *multi word sequences* of two or more hyphens. In order to produce a frequency list of such word sequences, the following pipeline, which presents the top 10 'hyphen words', can be used (the file `newstext` contains a sample of a British newspaper corpus [7]):

```
$ cat newstext |tr -cs '[a-zA-Z0-9-]' '\012'|egrep '\w-\w+-'|sort|
uniq -c|sort -rn|head -10
     11 21-year-old
      9 19-year-old
      8 28-year-old
      8 27-year-old
      6 24-year-old
      6 12-year-old
      5 two-year-old
      5 black-and-white
      5 30-year-old
      5 23-year-old
```

The hyphen must be included in the `tr` character set, since

```
tr -sc '[a-zA-Z0-9-]' '\012'
```

substitutes all characters for newlines *except* those listed inside the square brackets.

Almost all of the examples in the output are of the *-year-old* type. Now that we know that this is the most common type, we can take a look at how the top-list would look *without* the *-year-olds*. The command `egrep -v 'year-old'` prints all lines *not* including the string *year-old*:

```
$ cat newstext|tr -cs '[a-zA-Z0-9-]' '\012'|egrep '\w-\w+-'|
egrep -v year-old|sort|uniq -c|sort -rn|head
      5 black-and-white
      4 state-of-the-art
      4 brother-in-law
      4 Wem-ber-lee
      3 vis-a-vis
      3 up-to-date
      3 up-and-down
      3 two-and-a-half
      3 over-the-counter
      3 off-the-record
```

In this pipeline `egrep` was called twice.

12.2 Bigrams

In computational linguistics, a *bigram* is usually a sequence of two consecutive words in a text. (A tripple of words is called a trigram, and any number of consecutive words are denoted *ngrams.*)

A frequency list of all bigrams in a text will show the most frequent two-word sequences of the text. Bigrams can be interesting e.g. in lexicographic work.[6] In previous sections, tools for creating word lists (`tr`), printing part of a file (`tail`), pasting files together (`paste`), and producing frequency lists (`sort`, `uniq`) have been presented. These programs can be combined in order to produce a (frequency) list of bigrams. The process, which is described in e.g. [2], includes the following steps (which are all exemplified presently):

1. Downcase all upper case characters (optional).
2. Turn the original text file into a list of words.
3. Copy the word list to a new file and delete the first line, (the first word, i.e.) of the new file.
4. Paste the two word list files together to a file of bigrams.
5. Produce a frequency list based on the file of bigrams.

We have a tiny text file, `aiw.txt` (in which no line breaks occur in the middle of a word, i.e. no line ends in the middle of a word):

```
$ more aiw.txt
And so it was indeed:  she was now only ten inches high, and her face brightened
up at the thought that she was now the right size for going though the little
door into that lovely garden.
```

The following command puts each word in the text on a separate line, and prints the result to the file `aiw_list`:

[6]In [3], a nice example of how semantically related words are explored with the help of bigrams and a rather simple statistical method is found.

```
$ tr 'A-Z' 'a-z' < aiw.txt | tr -cs '[a-z0-9]' '\012' > aiw_list
```

The next step is to produce a copy of the word list file, with the first line (word) missing. This file will be `aiw_list2`:

```
$ tail +2 aiw_list > aiw_list2
```

Now, we can paste the two lists together, and save the result in the file `bigrams`:

```
$ paste aiw_list aiw_list2 > bigrams
```

Let us inspect the resulting list of bigrams:

```
$ more bigrams
and so
so it
it was
was indeed
indeed she
she was
was now
now only
only ten
ten inches
inches high
high and
and her
her face
face brightened
brightened up
up at
at the
the thought
thought that
that she
she was
was now
now the
the right
right size
size for
for going
going though
though the
the little
little door
door into
into that
```

```
that lovely
lovely garden
garden
```

The list of bigrams only serves to illustrate how the process works, but is really too small to be of any use. (However, the reader might verify that two bigrams occur more than once.) A file of bigrams produced according to this method is easily turned into a bigram frequency list with the help of the `sort | uniq -c | sort -nr` pipeline of Section 12.1.

12.3 Simplistic key-word extraction

By deleting all of the most over-all frequent words (mostly function words[7]) from a text and extracting the most frequent remaining words, you will (with a little bit of luck) get a list of key-words of the text. The explanation as to why we can find key-words in a text by deleting the function words, is that the function words are common to every text, but that we should expect that content words, central to a specific text, should be repeatedly used in this text, but perhaps not in other texts dealing with other subjects. (This is the basic idea behind a method for finding key-words suggested in [14].)

To be able to remove the most frequent words, a list of these, a **stop word list**, is needed. It can be obtained by taking the top of a frequency list, based on a reasonably large text sample. How to produce a frequency list is shown in Section 12.1 (and 13). The command `freq` will be used as shorthand for `sort | uniq -c | sort -rn` (see Section 13 for an explanation as to how this is possible).

Suppose that the 150 most frequent words of the SUSANNE corpus (see Section 7) would do as a stop word list and that the current directory contains the corpus files, each of which has a file name three characters long, `???`. The stop word list can be produced thus:

```
$ cut -f4 ???  | tr '[:upper:]'  '[:lower:]'  | freq |
head -150 | cut -f2 > stopwordlist
```

where `cut -f4` cuts out all tokens (the words of the actual text), `tr` down-cases all characters, `freq` produces the word frequency list, and `head` chops off the 150 most frequent items. The last `cut` picks out the words, but not the frequency numbers.

As can be seen on page 22, some of the lines in the SUSANNE corpus do not contain plain words in the token field (field four), but also things like `<minbrk>`, with non-alphabetic characters. The stop word list thus includes some non-words, like `<minbrk>`, `+,`, etc. (but this is okay, since we want to treat these as stop words anyhow; we do not want these as potential key-words). The top 20 words ('words' in a broad sense) of the SUSANNE corpus are

```
$ head -20 stopwordlist
the
+,
+.
of
and
```

[7]Words of the closed word classes, typically articles, prepositions, conjunctions, auxiliaries and the like.

```
to
-
a
in
<minbrk>
he
that
is
was
<ldquo>
+<rdquo>
for
it
+<hyphen>
as
```

Now, let us try to extract key-words from individual files of the corpus. The files `J09` and `J10` are two of the files of the SUSANNE corpus, each dealing with a different subject matter. All stop words have to be removed from these files, and a frequency list should be produced from the surviving words. In Section 2.11 `egrep -v`, which prints lines *not* matching the search pattern, was presented. It is time to introduce a few more switches: `-f`, `-F` and `-x`, the first of which tells `egrep` to read the search patterns from a file (the stop word list in this case) and not from the command line. The second switch, `-F`, tells `egrep` to treat the search pattern as a string of characters without any special meaning (as an alternative, the command `fgrep` can be used). This is needed since the stop word list has entries containing characters of special meaning to `egrep`, e.g. `+`. Lastly, the `-x` switch means that a search pattern must match a complete line, not just part of it, in order to be a hit. In other words, `egrep -vixFf stopwordlist FILE` means 'print only those lines in `FILE` *not* identical to any of the lines in `stopwordlist`'.

The top 20 words of `J10` after deleting the stop words:

```
$ cut -f4 J10 |  egrep -vixFf stopwordlist |
tr '[:upper:]' '[:lower:]' | freq | head -20
     11 bees
     10 pollen
     10 bumblebees
      9 queen
      8 species
      7 young
      7 willow
      7 nest
      7 female
      7 beebread
      7 bee
      7 andrena
      6 summer
      6 queens
```

```
          6 life
          6 larvae
          6 honey
          5 psithyrus
          5 females
          5 eggs
```

which can be compared to the top list of file `J09`

```
     $ cut -f4 J09 |  egrep -vixF -f stopwordlist |
     tr '[:upper:]' '[:lower:]' | freq | head -20
          30 1
          22 activity
          19 2
          18 saline
          18 group
          17 cells
          15 used
          12 serum
          12 dialysis
          12 buffer
          12 antibody
          11 sera
          11 fractions
          11 anti
          10 using
           9 gradient
           9 albumin
           8 samples
           8 sample
           8 regions
```

The two files obviously deal with different subject matters. A realistic stop word list should perhaps be produced more carefully than by blindly chopping off the 150 most common items of a corpus.

The previous sections have presented quite a number of different commands and switches. Remember that one does not have to remember all this. There are manual pages available, and they are invoked with the `man` command (see Page 14). These might be quite hard to understand, by if one has already learnt how to use a command, but forgotten the details, the man pages are often useful.

13 Shell files

The following is a rudimentary introduction to **shell scripts**, and is only intended to point out the fact that one can produce small programs by combining the programs available in the Unix system and putting the result in a 'shell file' or 'shell script'. More on shell files is found in e.g. [8].

13.1 `freq`

When it turns out that one types the same sequence of commands over and over again, one might want to consider producing a shell file with the commands, and run this 'program' instead of typing the commands. In Section 12.1 the pipeline `sort | uniq -c | sort -rn`, which produces a frequency list of its input, was used several times. By putting the pipeline in a file, called e.g. `freq`, preferably by using a text editor, or by the command

```
echo 'sort | uniq -c | sort -rn' > freq
```

and making the file **executable** (described below), one can now type `freq` instead of `sort | uniq -c | sort -rn`.

However, this tiny 'program' does not behave correctly if it is not called as part of a pipeline. For example, as part of the pipeline

```
cat TEXTFILE | tr -cs '[:alnum:]' '[\012*]' | freq | more
```

`freq` works fine, but called with one or more files as its argument(s), e.g. `freq WORDLISTFILE`, it will not work. The reason is that the shell file `freq` does not 'know' that it is supposed to read its input from a file when called this way.

To remedy this, the `freq` shell file is modified slightly:[8]

```
$ cat freq
sort $* | uniq -c | sort -rn
```

where `$*` is a **variable** which denotes the argument(s) with which `freq` is called. In the example above, `$*` would get the value `WORDLISTFILE`. If `freq` is called with e.g. the three arguments `file1 file2 file3` instead, the `$*` variable will represent a list of the files `file1 file2 file3`, etc. `sort $*` executes the `sort` command on the list of files. If the shell file is used as part of a pipeline, the variable will not get any value, and the `sort` command will get its input from the pipeline instead.

Given that `freq` is made executable (see below), the `freq` script will now work both as part of a longer pipeline, where its input comes from some other command, or with one or more files as its argument(s). Remember that for `freq` to produce meaningful output, its input should be something that is meaningful to make a frequency list out of, such as lists of words or bigrams, etc.

Often one does not want different counts for capital words, so it might be a good idea to throw in a down-casing `tr` command as well:

```
$ cat freq
cat $* | tr [:upper:] [:lower:] | sort | uniq -c | sort -rn
```

(or you could save this as a different shell file, called e.g. `freq_lc`, *lc* for 'lower case').

[8]In case you have forgotten: the `cat freq` command is used here to view the contents of the file `freq`.

Output from bigramfreq			
42 my love	23 the world	19 of the	14 in thee
33 thou art	23 my heart	18 that thou	14 in me
29 my self	22 when i	18 beauty s	14 do not
29 in the	22 of thy	17 and in	14 do i
28 that i	20 thy self	16 to thee	14 and all
27 in my	20 of my	16 to my	13 to make
26 love s	20 in thy	16 to me	13 of your
25 to be	20 i have	16 time s	13 not to
24 to the	20 for my	16 mine eye	13 mine own
24 i am	19 that which	14 in their	...

Figure 1: *The top of a bigram frequency-list based on the sonnets.*

13.2 `bigramfreq`

In Section 12.2, a simple method for producing bigrams was presented. The commands in that section, together with a pipeline for producing a frequency list, could be put in a script file, called e.g. `bigramfreq`, and this script could then be executed like an ordinary program, `bigramfreq INFILE(S)`, and the result, a bigram frequency list, would be printed to the screen ('standard output').

```
# bigramfreq
#
# Turn a text file into a frequency list of bigrams.

cat $* | tr -cs [a-zA-Z0-9] '[\012*]' | tr A-Z a-z > tmpfile1
tail +2 tmpfile1 > tmpfile2
paste tmpfile1 tmpfile2 | sort | uniq -c | sort -rn
rm tmpfile1 tmpfile2
```

There are a few things to observe. The lines starting with a `#` is not part of the 'program' proper, but are comments which explain the contents of the file.

The `$*` of the first line of the script is a **variable** which will be substituted for the file name(s) with which the script is called (e.g. `sonnets.txt` or whatever), just as in the `freq` example above.

Lastly, the script produces **temporary files**, `tmpfile1` and `tmpfile2`, which are removed, with the help of the `rm` command of the last line, when no longer needed. The rest of the commands in the script are all explained in Sections 3, 4, 5, 6, 7 and 8.

13.3 Making a file executable

To make a shell file executable, the **file permission** has to be changed. This is accomplished by typing

```
$ chmod +x bigramfreq
```

where `+x` means 'make this file (`bigramfreq`) executable'. If this is not done, the file is considered an ordinary text file, and not a program file, and an error message, such as 'Permission denied' will be the result from trying to run the file as a program. More about file permissions is found by reading `man ls` and `man chmod`.

An alternative to changing the permission of the file, is to call the 'shell' (`sh`) with the shell file (and *its* argument(s)) as its argument:

```
$ sh bigramfreq TEXTFILE
```

A `shell` (command line interpreter) is a program which reads the commands a user types and sends these commands to the operating system for execution. In a Unix system, there are typically different shells to choose from (e.g. `sh`, `bash`, `csh`, `tcsh`), each slightly different. The example shell scripts above should work with any of these (?).

A A handful of Unix commands

The commands enumerated below are not described in any detail; they are listed merely as a hint of what might be useful to know. By typing `man COMMAND`, where `COMMAND` is any of the commands described below, the man(ual) page for the command is displayed (these page are often quite cryptic, though). Most of the commands (programs) can take a host of different options (switches) modifying the behaviour of the command. One does not necessarily have all these commands available on one's own system (but they should all be available for downloading on the Internet (try e.g. [5]).

`cat` display ('print') contents of files. Can be used to concatenate files, or send the contents of one or more files into a pipeline.

`cd` change directory. (`cd ..` takes you to the directory immediately above the current directory, `cd` (without any arguments) takes you to your home directory.)

`chmod` change file permissions. `chmod +x FILE` makes the file `FILE` executable. `chmod -w FILE` write-protects `FILE` and `chmod -r FILE` removes the permission to read a file. `chmod +r FILE` gives permission to write to a file (make changes), etc. Normally, a user can only change the permissions of the files she has created herself.

`compress` compresses files, adding a `.Z` extension to the file names.

`CTRL-c` to exit a program when nothing else helps, hold down the `CTRL` key and type `c`. If this does not help, check out `kill`.

`CTRL-z` stop a process (program) temporarily and put it the background. The stopped process is resumed by the `fg` command. For example, type `man ls`, which shows a man page. Type `CTRL-z`. The `man` command is stopped, and you can use the command line as usual. Type `fg` and the `man` command will reappear.

`cut` pick out fields from a file.

`diff` compare two files to find the differences.

`df` check how much free disk space there is.

`du` check disk usage.

`echo` print a string (to 'standard out'). (If the argument to `echo` starts with a dollar sign, the value of an 'environment' variable is printed; try `echo $PATH` which prints the value of the PATH variable, etc.)

`egrep` search for strings with the help of regular expressions.

`emacs` a useful text editor, popular among programmers. It comes with an on-line tutorial.

`fgrep` search for strings (no regular expressions).

`finger` print log-in information about a user.

`ftp` down-load or upload files on remote machines (e.g. on the Internet).

`ghostview` program for viewing PostScript files.

`grep` almost same as `egrep`.

`gunzip` unzip (uncompress) files compressed with `gzip`.

`gzip` compress files. A file compressed with `gzip` will get the file extension `.gz`.

`head` prints a given number of lines of a file. The default value is 10 lines. (Section 7)

`hostname` prints the name of the machine to which you are logged in.

`kill` get rid of processes (programs) which you cannot terminate with other means. The ID number of the process ('PID') to kill is obtained by running `top` or `ps`.

`killall` terminate all processes, programs, which have this name. For example, `killall firefox` will kill the program `firefox`.

`less` is `more`, more or less.

`ls` list the contents of a directory. `ls -a` lists all files, also those with names starting with a period (e.g. `.login`). `ls -l` prints more information (size, date, permissions).

`man` invokes a manual page for a program, try e.g. `man ls` (or even `man man`).

`more` print the contents of a file to the screen, one screenful at a time. The text is scrolled by typing RET or SPACE.

`mv` move and/or rename file.

`nice` run a command with lower priority (e.g. when running a heavy task on someone else's computer, or when one wants to be able to keep working with other things while the computer runs some big job which would otherwise slow down the computer too much).

`paste` merge files in columns.

`ping` check if a remote machine is running.

`pwd` print the name of the working directory.

`renice` lower the priority of a process (see `nice`).

`rev` reverse the order of characters on each line.

`rlogin` can be used to log in to a different host (computer, i.e.).

`rm` remove files (never to be seen again, unless backed up). For example, `rm file` removes `file`. `rm f*` removes all files (in the current directory) beginning *f*. The present author can witness that the command `rm -r *` is an extremely dangerous command, which recursively deletes all files in the current directory and all its sub-directories.

`rmdir` remove a directory.

`sdiff` compare two files to find the differences.

`sed` can be used e.g. to "find and replace" strings. See Section 10.

`split` split a file into smaller ones. For example, `split -1000 HUGEFILE` splits the file `HUGEFILE` into smaller files, each 1000 lines long.

`su` log in. Followed by a user name, a different user than the one currently logged in can log in.

`tail` prints a number of lines from the end of a file. See Section 7.

`tar` store or extract `tar` archive files. `tar -xf FILE.tar` is used to unpack the archive `FILE.tar`

`telnet` log on to remote machine.

`top` print the processes (programs) running on the machine.

`tr` translate characters, e.g. uppercase to lower.

`uname` print the name of the operating system (`uname -a` prints the system version).

`wc` water closet (counts the words, lines and characters in a text file).

`whereis` (does its best to) report the search path to a program (e.g. `whereis egrep`).

`which` report the search path for a program (e.g., by typing **`which egrep`**, the path to **`egrep`** should appear). Only works for programs which are found by searching the paths defined by the PATH environment variable. (To check the value of your PATH variable, try `echo $PATH`.)

`who` check which users are logged in on the computer.

`whoami` print the user name of the user currently logged in.

`zcat` print the contents of zipped files (same as `cat`, but used on compressed files).

`zgrep` like `grep`, but works on files compressed with `gzip` or `compress`. `grep`? Uh, same as `egrep`, almost.

`zmore` is the same as `more`, but is used on gzipped files (see `gzip` and `gunzip` above).

References

[1] Bonk Business Inc. www homepage. `http://www.telegate.se/bonk/`, 1995.

[2] Kenneth Ward Church. Unix for poets. Seminar slides/notes. Available on-line on the Internet at: `http://pi0995.kub.nl:2080/Paai/Onderw/Ttt/Poets01.html`.

[3] Kenneth Ward Church and Patrick Hanks. Word association norms, mutual information, and lexicography. In *Proceedings from the 27th Meeting of the ACL*, pages 76–83, 1989. Also in *Computational Linguistics* 16.

[4] Jeffrey E.F. Friedl. *Mastering Regular Expressions*. O'Reilly & Associates, Inc., Sebastopol, CA, 1997.

[5] GNU homepage. `http://www.gnu.org/`.

[6] Project Gutenberg, www homepage. `http://www.gutenberg.net/`, 199?

[7] The Independent on CD-ROM, 1 january 1996-30 june 1996. CD–ROM, 1996.

[8] Brian W. Kernighan and Rob Pike. *The UNIX Programming Environment*. Prentice Hall, 1984.

[9] Mats Mellstrand. *UNIX—Grunden till öppna system*. eXmandato AB, Kalmar, 1991. (In Swedish).

[10] The ruby programming language web site. `http://ruby-lang.org/`, 2006.

[11] Geoffrey Sampson. *English for the Computer*. Claredon Press, Oxford, 1995.

[12] The SUSANNE corpus. Available by anonymous `ftp` at `ota.ox.ac.uk` (under the directory `pub/ota/susanne`).

[13] Kurt Vonnegut. *Breakfast of Champions*. Dell, 1975. Paperback edition.

[14] Margareta Westman. Vanliga och ovanliga ord. In B. Molde, editor, *Språkvårdsstudier*. Esselte Studium, Stockholm, 1974. (In Swedish).