Appendix B

Tools

B.1 Simple Unix Commands

Before we can do all the fancy statistics on texts, we need to access the raw data. Unix provides a number of built-in commands that are very useful to do so. In the following a list of elementary commands valuable for text preprocessing is given. The online man pages on your Unix machine will give you more information on the specific commands.

Command	Usage
grep	search a file for a pattern
egrep	search a file for disjunctive patterns
sort	sort and/or merge files
uniq	report repeated lines in a file
tr	translate characters
wc	display a count of lines, words and characters in a file
cut	cut out selected fields of each line of a file
paste	merge same lines of several files or subsequent lines of one file
join	merges those lines of a file that have the same key
comm	select or reject lines common to two sorted files
cat	concatenate files
tail	writes the last part of a file to standard output
split	split a line by indicated split characters and particularly useful:
split -n	split a file into pieces of n lines
sed	stream editor

Unix also comes with a number of programming languages of which the following are popular for text handling:

Awk is a pattern scanning and processing language that is preferably used for database construction.

Lex or its successor **flex** is a language for lexical analysis. Typically tokenizer are written in lex or flex. Lex programmes can be interleaved with C code. They are compiled into C, and thus are rather efficient.

Perl is a language for easy manipulation of text, files, and processes.

It borrows capabilities from shells and C, and subsumes awk and sed. Perl is compared to C easy to learn, but not very efficient.

From the following examples[1] you might get an idea about what you could do with the above commands. A good guide for novices is Ken Church's *Unix for poets* [Church 1994].

For those who are not used to Unix, the following is to understand the syntax of the command lines occurring in the examples below:

```
<                  read from input file
>                  write to output file
|                  pipe
| more             pipe to more;
                   more displays standard output screenwise
```

The following is a list with the most essential characters to build search patterns with:

```
+          one or more instances of a specified pattern
*          zero or more instances of a specified pattern
?          at most one instance of a specified pattern
.          any single character
[a-z]      any one character from the small alphabet
[A-Z]      any one character from the ?big alphabet
[0-9]      any one number between 0 and 9
^          begin of line
$          end of line
|          disjunction of patterns
\          indicates that the following character is
           interpreted as such
```

B.2 Split up a text using tr

- Split up a text by blanks, and write each word (i.e. string of characters surrounded by blanks) in a single line to a file. The input file is infile, the output file is outfile. \012 is the ASCII code for new line.

  ```
  tr ' ' '\012'  < infile >  outfile
  ```

- Get rid of empty lines.

  ```
  tr -sc 'A-Za-z' '\012' < infile > outfile
  ```

Now you have a file, lets call it wl, containing your input text as word list. With the word list you can do other fancy things.

B.3 Sort word list: sort, uniq

- Sort word list by dichtionary order.

  ```
  sort -d wl
  ```

[1]The material is taken from an introductory course on corpus linguistics held by the authors at Uppsala University.

- Sort word list by dichtionary order, and get rid of duplicates. Write output to outfile.

```
sort -d wl |  uniq > outfile
or
sort -du wl > outfile
```

- Sort word list by dictionary order, get rid of duplicates, and give a count of how often each word appeared in the word list.

```
sort -d wl | uniq -c > outfile
```

- Sort word list according to word frequency (numeric order).

```
uniq -c wl  | sort -d > outfile
```

B.4 Merge counts for upper and lower case: tr, sort, uniq

- Sometimes you just want to know how often a specific word occurs in a text. And you do not want to make a distinction between upper and lower case letters.

```
tr 'A-Z' 'a-z' wl | uniq -c | sort -d > outfile
```

B.5 Count lines, words, characters: wc

`wc infile` counts lines, words, characters

`wc -l infile` counts lines

`wc -w infile` counts words

`wc -m infile` counts characters

B.6 Display the first n lines of a file: sed

`sed 5q infile` displays the first 5 lines of the file infile

`sed 50q infile` displays the first 50 lines of the file infile

B.7 Find lines: grep, egrep

`grep gh`	`find lines containing 'gh'`
`grep '^gh'`	`find lines beginning with 'gh'`
`grep 'hg$'`	`find lines ending with 'hg'`
`grep '^gh.*hg$'`	`find lines beginning with 'gh' and ending with 'hg'`
`grep -v gh`	`delete lines containing 'gh'`
`grep -v '^gh'`	`delete lines beginning with 'gh'`
`grep -v 'gh$'`	`delete lines ending with 'gh'`
`grep -v '^gh.*hg$'`	`delete lines beginning with 'gh' and ending with 'hg'`

Note with egrep you can specify disjunctive search patterns; for instance:

`egrep 'ful$\|ly$|ant$'`	`find lines ending with 'ful', 'ly', or 'ant'`

B.8 n-grams: tail, paste

Suppose we have got a text file with each word on a single line. Now we can easily create all sorts of n-grams simply by using tail and paste. wl contains the original word list, nextwords contains the original word list reduced by the first element, nextnextwords contains the original word list reduced by the first two elements. The files bigrams, and trigrams contain all bigrams or trigrams respectively occurring in the text.

- Create bi-grams: tail, paste

  ```
  tail +2 wl > nextwords
  paste wl nextwords > bigrams
  ```

- Create tri-grams:

  ```
  tail +3 wl > nextnextwords
  paste wl nextwords nextnextwords > trigrams
  ```

- ...

B.9 Manipulation of lines and fields: awk

Despite **awk** is a general purpose programming language it is intended for shorter programs, especially for easy manipulation of lines and fields. Note, awk is fully integrated into Perl. Warning: awk also comes as **nawk**, and **gawk**.

The following is useful to know:

- Fields are addressed by $ fieldnumber

$1	first field
$NF	last field
$(NF-1)	penultimate field

- Some operators

  ```
  >, <, >=, <=, ==, =!, etc.
  ```

  For more information see the man pages.

- With -F you specify the field seperator, e.g. -F: specifies : as current field seperator. Blank is the default field seperator.

- Select fields by position

  The following command line specifies : as the current field seperator, and prints the 4th field of each line (record) to standard output. The input file is infile.

  ```
  awk -F: '{print $4}' infile
  ```

- Filter fields by numerical comparison

  The following command line matches fields where the numerical value of the first field is larger than 2, and prints the contents of the 3rd field of the according record to standard output. The file seperator is set to default (i.e. blank). The input file is infile.

  ```
  awk '$1 > 2 {print $3}' infile
  ```

- Filter fields by string comparison

  The following command line matches lines where the first field is identical to the last one, and prints the contents of the 3rd field to standard output. The file seperator is set to default (i.e. blank). The input file is again infile.

  ```
  awk '$1 == $NF {print $3}' infile
  ```