Chapter 3

Tools for finding and displaying text

This chapter will introduce some basic techniques and operations for use in data-intensive linguistics. We will show how existing tools (mainly standard UNIX tools) and some limited programming can be used to carry out these operations.

Some of the material on standard UNIX tools is a revised and extended version of notes taken at a tutorial given by Ken Church, at Coling 90 in Helsinki, entitled “Unix for Poets”. We use some of his exercises, adding variations of our own as appropriate.

One of Church’s examples was the creation of a KWIC index, which we first encountered in chapter 5 of Aho, Kernighan and Weinberger (1988). In section 4 we discuss this example, provide variations on the program, including a version in Perl, and compare it with other (more elaborate, but arguably less flexible) concordance generation tools.

There are more advanced, off-the-shelf tools that can be used for these operations, and several of them will be described later on in these notes. In theory, these can be used without requiring any programming skills. But more often than not, available tools will not do quite what you need, and will need to be adapted. When you do data-intensive linguistics, you will often spend time experimentally adapting a program written in a language which you don’t necessarily know all that well. You do need to know the basics of these programming languages, a text editor, a humility, a positive attitude, a degree of luck and, if some or all of that is missing, a justified
belief in your ability to find good manuals. The exercises towards the end of this chapter concentrate on this kind of adaptation of other people’s code. 

This chapter therefore has two aims:

1. to introduce some very basic but very useful word operations, like counting words or finding common neighbours of certain words;
2. to introduce publicly available utilities which can be used to carry out these operations, or which can be adapted to suit your needs.

The main body of this chapter focuses on particularly important and successful suite of tools — the UNIX tools. Almost everybody who works with language data will find themselves using these tools at some point, so it is worth understanding what is special about these tools and how they differ from other tools. Before doing this we will take time out to give a more general overview of the tasks to which all the tools are dedicated, since these are the main tasks of data-intensive linguistics.

3.0.1 Search tools for data-intensive linguistics

The general setup is this: we have a large corpus of (maybe) many millions of words of text, from which we wish to extract data which bears on a research question. We might for example, contra Fillmore, be genuinely interested in the distribution of parts-of-speech in first and second positions of the sentence. So we need the following

- A way of specifying the sub-parts of the data which are of interest. We call the language in which such specifications are made a query language.

- A way of separating the parts of the data which are of interest from those which merely get in the way. For large corpora it is impractical to do this by hand. We call the engine which does this a query engine.

Either: a way of displaying the extracted data in a form which the human user finds easy to scan and assess.

Or: a way of calculating and using some statistical property of the data which can act as a further filter before anything is displayed
We would like a tool which offers a flexible and expressive query language, a fast and accurate query engine, beautiful and informative displays and powerful statistical tests. But we can’t always get exactly what we want.

The first design choice we face is that of deciding over which units the query language operates. It might process text a word at a time, a line at a time, a sentence at a time, or slurp up whole documents and match queries against the whole thing. In the UNIX tools it was decided that most tools should work either a character at a time or (more commonly) a line at a time. If we want to work with this convention we will need tools which can re-arrange text to fit the convention. We might for example want a tool to ensure that each word occurs on a separate line, or a similar one which does the same job for sentences. Such tools are usually called filters, and are called to prepare the data the real business of selecting the text units in which we are interested. The design strategy of using filters comes into its own when the corpus format changes: in the absence of filters we might have to change the main program, but if the main business is sheltered from the messiness of the real data by a filter we may need to do nothing more than to change the filter.

The filter and processor architecture can also work when units are marked out not by line-ends but by some form of bracketing. This style is much used when the information associated with the text is highly-structured. The Penn Treebank (described in Marcus et al. (1993)) uses this style. The current extreme of this style uses SGML (Standard Generalized Markup Language Goldfarb (1990)), essentially an a form of the bracketing-delimits-unit style of markup, with all sorts of extra facilities for indicating attributes and relationships which hold of units.

The next design choice is the query language itself. We could demand that users fully specify the items which the want, typing in (for example) the

---

1 Note that the statistical mechanism can operate in either of two modes, in the first mode it completely removes data which it thinks the human user, while in a second, more conservative mode it merely imposes an order on the presentation of data to the user, so nothing is ever missed, provided only that the user is persistent enough.

2 For the purposes of this discussion it doesn’t matter much whether the information has been created by a human being and delivered as a reference corpus, or whether an earlier tool generates it on the fly from less heavily-annotated text.
word in which they are interested, but that is of very limited utility, since you have to know ahead of time exactly what you want. Much better is to allow a means of specifying (for example) “every sentence containing the word ‘profit’”. Or again “every word containing two or more consecutive vowels”. There is a tradeoff here, since the more general you make the query language, the more demanding is the computational task of matching it against the data.

Performance can often be dramatically improved by indexing the corpus with information which will speed up the process of query interpretation. The index of a book is like this, freeing you from the need to read the whole book in order to find the topic of interest. But of course the indexing strategy can break down disastrously if the class of queries built into the index fails to match the class of queries which the user wants to pose. Hence the existence of reversed-spelling dictionaries and rhyming dictionaries to complement the usual style. ) is a tool which relies heavily on indexing, but in doing so it restricts the class of queries which it is able to service. What it can do is done so fast that is a great tool for interactive exploration by linguists and lexicographers.

One of the most general solutions to the query-language problem is to allow users to specify a full-fledged formal grammar for the text fragments in which they are interested. This gives a great deal of power: and Corley et al. (1997) has shown that it is possible to achieve adequately efficient implementations. It requires pre-tagged text, but otherwise imposes few constraints on input. Filters exist for the BNC, for the Penn treebank, and for Susanne.

**Note for Edinburgh readers:**
And it is an Edinburgh in-house product, for which it is pretty easy to write new filters. And Frank Keller knows about it…

. We will be working with Gsearch for the first assessed exercise.

See on the Web for more documentation. A sample grammar is in table ??.

Finally, there are toolkits for corpus processing like that described in: McKelvie et al. (1997), which we call LT-NSL or LT-XML, depending (roughly) on the wind-direction which offer great flexibility and powerful query languages for those who are able and willing to write their own tools. Packaged in the form of `sggrep`, the query language is ideally suited for search over corpora which have been pre-loaded with large amounts of reliable and hierarchically structured annotation.
Search tools for data-intensive linguistics

% File: Grammar
% Purpose: A fairly simple Corset grammar file describing NPs and PPs
% Author: Steffan Corley
% Date: 6th December 1996

#defterm "tag" % Saves writing

np --> det n1+ pp*
np --> n1+ pp*
np --> np conj np

n1 --> adj* noun+
n1 --> adj* gen_noun n1
n1 --> n1 conj n1

gen_noun --> noun genitive_marker

pp --> prep
%%
% BNC specific part
%%
npdet --> <AT.>* % Determiner
adj --> <AJ.>* % Adjective
adj --> <ORD.>* % Ordinal
noun --> <NN.>* % common noun
noun --> <NP.>* % proper noun
genitive_marker --> <POS.>* % Saxon genitive
prep --> <PR.>* % Preposition

conj --> <CJ.>* % Conjunction

ofp --> of np

Table 3.1: a sample Gsearch grammar
Note for Edinburgh readers:
And it is an Edinburgh in-house product whose development you
can influence. The query language adds some things to that of
xwic, but also lacks some of what xwic has.

See for further documentation.

It probably isn’t worth going into great detail about ways of displaying
matched data, beyond the comment that visualisation methods are impor-
tant if you want human beings to make much sense of what is provided.

3.0.2 Using the UNIX tools

So back to the practicalities of the UNIX tools... This chapter is interactive.
To follow this chapter, make sure you have the text files exatext1, exatext2
and exatext3, and code files wc.awk and wc.perl in a directory where you
have write permission. If you don’t have these files, you can take another
plain text file and call it exatext1. You will be told the contents of the
other files in the course of this chapter, so you can create them yourself.

Note for Edinburgh readers:
UNIX commands work differently between different UNIX operat-
ing systems. You may on occasion need to execute a few com-
mands to set things up. These Edinburgh local things will be
highlighted like this comment here.

3.1 Sorting and counting text tokens

A tokeniser takes input text and divides it into “tokens”. These are usually
words, although we will see later that the issue of tokenisation is far more
complex than that. In this chapter we will take tokenisation to mean the
identification of words in the text. This is often a useful first step, because
it means one can then count the number of words in a text, or count the
number of different words in a text, or extract all the words that occur
exactly 3 times in a text, etc.

UNIX has some facilities which allow you to do this tokenisation. We start
with tr. This “translates” characters. Typical usage is

tr chars1 chars2 < inputfile > outputfile
which means “copy the characters from the input file into the output file and substitute all characters specified in chars1 by chars2”.

For example, tr allows you to change all the characters in the input file into uppercase characters:

```bash
tr 'a-z' 'A-Z' < exatext1 | more
```

This just says “translate every a into A, every b into B, etc.”

**Note for Edinburgh readers:**

On different UNIX systems the options for tr are slightly different. Read the man-page to get the details. For the purposes of this tutorial, you should use the UNIX command alias tr /usr/ucb/tr (or equivalent if you use a non-standard shell) to get a version of tr which works as described here.

Similarly,

```bash
tr 'aiou' e < exatext1 | more
```

changes all the vowels in exatext1 into es.

You can also use tr to display all the words in the text on separate lines. You do this by “translating” everything which isn’t a word (every space or punctuation mark) into newline (ASCII code 012). Of course, if you just type

```bash
tr 'A-Za-z' '\012' < exatext1
```

each letter in exatext1 is replaced by a newline, and the result (as you can easily verify) is just a long list of newlines, with only the punctuation marks remaining.

What we want is exactly the opposite—we are not interested in the punctuation marks, but in everything else. The option -c provides this:

```bash
tr -c 'A-Za-z' '\012' < exatext1
```

Here the complement of letters (i.e. everything which isn’t a letter) is mapped into a newline. The result now looks as follows:
Text
in
a
class
of
its
own

The
HCRC
Language
Technology
Group

LTG

is
a
technology
transfer
...

There are some white lines in this file. That is because the full stop after a class of its own is translated into a newline, and the space after the full stop is also translated into a newline. So after own we have two newlines in the current output. The option -s ensures that multiple consecutive replacements like this are replaced with just a single occurrence of the replacement character (in this case: the newline). So with that option, the white lines in the current output will disappear.

Exercise:
Create a file exa_words with each word in exatext1 on a separate line.

Solution:
Just type tr -cs 'A-Za-z' '\012' < exatext1 > exa_words

You can combine these commands, using UNIX pipelines (1). For example, to map all words in the example text in lower case, and then display it one word per line, you can type:

tr 'A-Z' 'a-z' < exatext1 | tr -sc 'a-z' '\012' > exa_tokens
3.1. SORTING AND COUNTING TEXT TOKENS

The reason for calling this file *exa_tokens* will become clear later on. We will refer back to files created here and in exercises, so it’s useful to follow these naming conventions.

Another useful UNIX operation is *sort*. It sorts lines from the input file, typically in alphabetical order. Since the output of *tr* was one word per line, *sort* can be used to put these lines in alphabetical order, resulting in an alphabetical list of all the words in the text. Check the man-page for *sort* to find out about other possible options.

Exercise:
Sort all the words in *exatext1* in alphabetical order.

Solution:
Just pipeline the *tr* command with *sort*: i.e. type

```
tr -cs 'A-Za-z' \012 < exatext1 | sort | more
```

Or to get an alphabetical list of all words in lowercase, you can just type

```
sort exa_tokens > exa_tokens_alphab.
```

The file *exa_tokens_alphab* now contains an alphabetical list of all the word tokens occurring in *exatext1*.

The output so far is an alphabetical list of all words in *exatext1*, including duplicates, each on a separate line. You can also produce an alphabetical list which strips out the duplicates, using *sort* -u.

Exercise:
Create a file *exa_types_alphab*, containing each word in *exatext1* exactly once.

Solution:
Just type

```
sort -u exa_tokens > exa_types_alphab
```

Sorted lists like this are useful input for a number of other UNIX tools. For example, *comm* can be used to check what two sorted lists have in common. Have a look at the file *stoplist*: it contains an alphabetical list of very common words of the English language. If you type

```
comm stoplist exa_types_alphab | more
```

you will get a 3-column output, displaying in column 1 all the words that only occur in the file *stoplist*, in column 2 all words that occur only in
exa_types_alphab, and in column 3 all the words the two files have in common. Option -1 suppresses column 1, -2 suppresses column 2, etc.

**Exercise:**
Display all the non-common words in exatext

**Solution:**
Just type `comm -1 -3 stoplist exa_types_alphabetical | more`  
That compares the two files, but only prints the second column, i.e. those words which are in exatext but not in the list of common words.

The difference between word types and word tokens should now be clear. A word token is an occurrence of a word. In exatext1 there are 1,206 word tokens. You can use the UNIX command `wc` (for word count) to find this out: just type `wc -w exatext1`.  
However, in exatext1 there are only 427 different words or word types. (Again, you can find this out by doing `wc -w exa_types_alphabetical`).

There is another tool that can be used to create a list of word types, namely `uniq`. This is a UNIX tool which can be used to remove duplicate adjacent lines in a file. If we use it to strip duplicate lines out of exa_tokens_alphab we will be left with an alphabetical list of all wordtypes in exatext1—just as was achieved by using `sort -u`. Try it by typing

```
uniq exa_tokens_alphab | more
```

The complete chain of commands (or pipe-line) is:

```
tr -cs 'a-z' '\012' < exa_tokens | sort | uniq | more
```

**Exercise:**
Can you check whether the following pipeline will achieve the same

```
tr -cs 'a-z' '\012' < exa_tokens | uniq | sort | more
```

**Solution:**
It won't: uniq strips out adjacent lines that are identical. To ensure that identical words end up on adjacent lines, the words have to be put in alphabetical order first. This means that `sort` has to precede `uniq` in the pipeline.

An important option which `uniq` allows (do check the `man`-page) is `uniq -c`: this still strips out adjacent lines that are identical, but also tells you
how often that line occurred. This means you can use it to turn a sorted alphabetical list of words and their duplicates into a sorted alphabetical list of words without duplicates but with their frequency\textsuperscript{3} Try

\texttt{uniq -c exa\_tokens\_alphab > exa\_alphab\_frequency}

The file \texttt{exa\_alphab\_frequency} contains information like the following:

3 also
5 an
35 and
2 appear
3 appears
1 application
5 applications
1 approach

In other words, there are 3 tokens of the word “also”, 5 tokens of the word “an”, etc.

Exercise:
Can you see what is odd about the following frequency list?
\texttt{tr -cs 'A-Za-z' \012 < exatext1 | sort | uniq -c | more}

How would you correct this pipeline?

Solution:
The odd thing is that it counts uppercase and lowercase words separately. For example, it says there are 11 occurrences of “The” and 74 occurrences of “the” in exatext1. That is usually not what you want in a frequency list. If you look in \texttt{exa\_alphab\_frequency} you will see that that correctly gives “the” a frequency of occurrence of 85. The complete pipeline to achieve this is \texttt{tr 'A-Z' 'a-z' < exatext1 | tr -sc 'a-z' \012 | sort | uniq -c | more}

It may be useful to save a version of \texttt{exatext1} with all words in lower case. Just type
\texttt{tr 'A-Z' 'a-z' < exatext1 > exatext1\_lc}

Now that you have a list of all word types in \texttt{exatext1} and the frequency with which each word occurs, you can use \texttt{sort} to order the list according

\textsuperscript{3}The idea of using a pipeling this way (Under Unix: \texttt{sort | uniq -c | sort -nr}) to generate numerically sorted frequency lists was published in Bell System Technical Journal, 57:8, pp 2137-2154.
to frequency. The option for numerical ordering is `sort -n`; and if you add the option `-r` it will display the word list in reverse numerical order (i.e. the most frequent words first).

**Exercise:**
Generate a frequency list for `exatext`. Call it `exa_freq`.

**Solution:**
One solution is to type
```
sort -nr < exa_alphab_frequency > exa_freq
```

The complete pipeline to achieve this was
```
tr -cs 'a-z' '
012' < exatext1_lc | sort | uniq -c | sort -nr
```

To recap: first we use `tr` to map each word onto its own line. Then we `sort` the words alphabetically. Next we remove identical adjacent lines using `uniq` and use the `-c` option to mark how often that word occurred in the text. Finally, we `sort` that file numerically in reverse order, so that the word which occurred most often in the text appears at the top of the list.

When you get these long lists of words, it is sometimes useful to use `head` or `tail` to inspect part of the files, or to use the stream editor `sed`. For example, `head -12 exa_freq` or `sed 12q exa_freq` will display just the first 12 lines of `exa_freq`; `tail -5` will display the last 5 lines; `tail 14+` will display everything from line 14. `sed /indexer/q exa_freq` will display the file `exa_freq` up to and including the line with the first occurrence of the item “indexer.”

**Exercise:**
List the top 10 words in `exatext1`, with their frequency count.

**Solution:**
Your list should look as follows:
```
  85 the  34 in
  42 to  22 text
  39 of  18 for
  37 a  15 is
  35 and  14 this
```

With the files you already have, the easiest way of doing it is to say `head -10 exa_freq`.

The complete pipeline is
```
tr -cs 'a-z' '\012' < exatext1_lc | sort | uniq -c | sort -nr | head -10
```
3.2 Lemmatization

The lists of word types we’re producing now still have a kind of redundancy in them which in many applications you may want to remove. For example, in `exa_alphab_frequency` you will find the following:

```
  4 at
  2 automatic
  1 base
  2 based
  2 basic
 12 be
  3 been
  1 between
```

In other words there are 12 occurrences of the word “be”, and 3 of the word “been”. But clearly “be” and “been” are closely related, and if we are interested not in occurrences of words but word types, then we would want “be” and “been” to be part of the same type.

This can be achieved by means of lemmatisation: it takes all inflectionally related forms of a word and groups them together under a single `lemma`.

There are a number of freely available lemmatisers available.

**Note for Edinburgh readers:**
Have a look at John Carroll’s *English Lemmatiser*. It’s available as `/usr/contrib/bin/morph`. There is also a short paper there on the lemmatiser.

This lemmatiser accepts tagged and untagged text, and reduces all nouns and verbs to their base forms. Use the option `-u` if the input text is untagged. If you type `morph -u < exatext1 | more` the result will look as follows:

```
the hcrc language technology group (ltg) be a technology transfer
group work in the area of natural language engineering. it work
with client to help them understand and evaluate natural language
process method and to build language engineer solution
```

If you add the option `-s` you will see the derivations explicitly:
the hcrc language technology group (ltg) be+s a technology transfer
group work+ing in the area of natural language engineering. it work+s
with client+s to help them understand and evaluate natural language
process+ing method+s and to build language engineer+ing solution+s

Exercise:

Produce an alphabetical list of the lemmata in exatex1 and their frequencies.

Solution:

If you type

```
morph -u < exatext1 | tr 'A-Z' 'a-z' |
    tr -cs 'a-z' '\012' | sort | uniq -c > exa_lemmat_alphabet
```

the result will be a list containing the following:

- 4 at
- 2 automatic
- 3 base
- 2 basic
- 44 be
- 1 between

Note the difference with the list on page 31: all inflections of “be” have been
reduced to the single lemma “be”.

Note also that “base” and “based” have been reduced to the lemma “base”,
but “basic” wasn’t reduced to “base”. This lemmatiser only reduces nouns
and verbs to their base form. It doesn’t reduce adjectives to related nouns,
comparatives to base forms, or nominalisations to their verbs. That would
require a far more extensive morphological analysis. However, the adjectives
“rule-based” and “statistics-based” were reduced to the nominal lemma
“base”, probably an “over-lemmatisation”. Similarly, “spelling and and style
checking” is lemmatised as

**spelling and style checking**

which is strictly speaking inconsistent.

It is very difficult to find lemmatisers and morphological analysers that will
do exactly what you want them to do. Developing them from scratch is
extremely time-consuming. Depending on your research project or application,
the best option is probably to take an existing one and adapt the source
code to your needs or add some preprocessing or postprocessing tools. For example, if our source text `exatext1` had been tagged, then the lemmatiser would have known that “rule-based” was an adjective and would not have reduced it to the lemma “base”.

Wheras for some data-intensive linguistics applications you want to have more sophisticated lemmatisation and morphological analysis, in other applications less analysis is required. For example, for many information retrieval applications, it is important to know that “technological”, “technologies” and “technology” are related, but there is no real need to know which English word is the base word of all these words—they can all be grouped together under the word “technologi”. This kind of reduction of related words is what stemmers do.

Again, there are a number of stemmers freely available.

**Note for Edinburgh readers:**
We’ll be using the stemmer available as

```
/projects/ltg/projects/NLSD/ir-code/stemmer/stemmer
```

There is also some limited documentation in that directory.

If you type `stemmer exatext1 | more`, the sentence

```
The HCRC Language Technology Group (LTG) is a technology transfer group working in the area of natural language engineering.
```

will come out as

```
the hcrc languag technologi group ltg i a technologi transfer group work in the area of natur languag engin
```

### 3.3 Making n-grams

To find out what a word’s most common neighbours are, it is useful to make a list of bigrams (or trigrams, 4-grams, etc)–i.e. to list every cluster of two (or three, four, etc) consecutive words in a text.

Using the UNIX tools introduced in section 3.1 it is possible to create such n-grams. The starting point is again `exa_tokens`, the list of all words in the text, one on each line, all in lowercase. Then we use `tail` to create the tail end of that list:
tail +2 exa_tokens > exa_tail2

This creates a list just like `exa_tokens` except that the first item in the new list is the second item in the old list. We now paste these two lists together:

```
paste -d ' ' exa_tokens exa_tail2 > exa_bigrams
```

`paste` puts files together “horizontally”: the first line in the first file is pasted to the first line in the second file, etc. (Contrast this with `cat` which puts files together “vertically”: it first takes the first file, and then adds to it the second file.) Each time `paste` puts two items together it puts a `tab` between them. You can change this delimiter to anything else by using the option `-d`.

If we use `paste` on `exa_tokens` and `exa_tail`, the \(-n\)-th word in the first list will be pasted to the \(-n\)-th word in the second list, which actually means that the \(-n\)-th word in the text is pasted to the \(-n+1\)-th word in the text. With the option `-d ' '`, the separator between the words will be a simple space. This is the result:

```
text in
in a
a class
class of
of its
its own
...
```

Note that the last line in `exa_bigrams` contains a single word rather than a bigram.

**Exercise:**
What are the 5 most frequent trigrams in `exatext1`.

**Solution:**
This is the list:

```
4 the human indexer
4 in the document
3 categorisation and routing
2 work on text
2 we have developed
```
For creating the trigrams, start again from `exa_tokens` and `exa_tail2` as before. Then create another file with all words, but starting at the second word of the original list:

```
tail +3 exa_tokens > exa_tail3
```

Finally paste all this together:

```
paste exa_tokens exa_tail2 exa_tail3 > exa_trigrams
```

Since all trigrams are on separate lines, you can `sort` and count them the same way we did for words:

```
sort exa_trigrams | uniq -c | sort -nr | head -5
```

**Exercise:**
How many 4-grams are there in `exatext`? How many different ones are there? (Hint: use `wc -l` to display a count of lines.)

**Solution:**
Creating 4-grams should be obvious now:

```
tail +4 exa_tokens > exa_tail4
paste exa_tokens exa_tail2 exa_tail3 exa_tail4 > exa_fourgrams
```

A `wc -l` on `exa_fourgrams` will reveal that it has 1,213 lines, which means there are 1,210 4-grams (the last 3 lines in the file are not 4-grams). When you `sort` and `uniq` that file, a `wc` reveals that there are still 1,200 lines in the resulting file, i.e. there are 1,197 different 4-grams. Counting and sorting in the usual way results in the following list:

```text
2 underlined in the text
2 the system displays the
2 the number of documents
2 the figure to the
2 should be assigned to
```

Of course, there was a simpler way of calculating how many 4-grams there were: there are 1,213 tokens in `exa_tokens`, which means that there will be 1,212 bigrams, 1,211 trigrams, 1,210 4-grams, etc.

### 3.4 Filtering: grep

When dealing with texts, it is often useful to locate lines that contain a particular item in a particular place. The UNIX command `grep` can be used for that. Here are some of the options:

```
grep 'text'          find all lines containing the word “text”
grep '^text'         find all lines beginning with the word “text”
```
CHAPTER 3. TOOLS FOR FINDING AND DISPLAYING TEXT

grep 'text$'  
find all lines ending in the word “text”

grep '^[0-9]'  
find lines containing any number

grep '[A-Z]'  
find lines containing any uppercase letter

grep '^\[A-Z\]'  
find lines starting with an uppercase

grep '[a-z]$'  
find lines ending with a lowercase

grep '^[aeiouAEIOU]'  
find lines with a vowel

grep '^[aeiouAEIOU]$'  
find lines ending with a consonant (i.e. not a vowel)

grep -i '^[aeiou]$'  
find lines ending with a vowel (ignore case)

grep -i '^[^aeiou]'  
find lines starting with a consonant (ignore case)

grep -v 'text'  
print all lines except those that contain “text”

grep -v 'text$'  
print all lines except the ones that end in “text”

The man-page for grep will show you a whole range of other options. Some examples:

grep -i '^[aeiou].*[aeiou]' exatext1  
find lines with a lowercase vowel, followed by one or more (*) of anything else (.), followed by another lowercase vowel; i.e. find lines with two or more vowels.

grep -i '^[^aeiou]*[aeiou][^aeiou]*$' exatext1  
find lines which have no vowels at the beginning or end, and which have some vowel in between; i.e. find lines with exactly one vowel (there are none in exatext1).

grep -i '[aeiou][aeiou][aeiou]' exatext1  
find lines which contain (words with) sequences of 3 consecutive vowels (it finds all lines with words like obviously, because of its three consecutive vowels).

grep -c displays a count of matching lines rather than displaying the lines that match. * means “any number of”, i.e. zero or more. In egrep (which is very similar to grep), you can also use +, which means “one or more”.

Check the man-page for other grep options.

Exercise:
How many words are there in exatext1 that start in uppercase?
Solution:

There are different ways of doing this. However, if you simply do `grep -c '\[A-Z\]' exatext1` then this will only tell you how many lines there are in `exatext1` that contain words with capital letters. To know how many words there are with capital letters, one should carry out the `grep -c` operation on a file that only has one word from `exatext1` per line:

`grep -c '\[A-Z\]' exa_words`.

The answer is 79—i.e. there are 79 lines in `exa_words` with capital letters, and since there is only one word per line, that means there are 79 words with capital letters in `exatext1`.

Exercise:

Can you give a frequency list of the words in `exatext1` with two consecutive vowels?

Solution:

The answer is

```
5  group
5  clients
4  tools
4  should
4  noun
3  our
...
```

If we start from a file of lowercase words, one word per line (file `exa_tokens` created earlier), then we just `grep` and `sort` as follows:

```
grep '^[^aeiou]*[aeiou][aeiou][^aeiou]*$' exa_tokens | sort | uniq -c | sort -nr
```

Exercise:

How many words of 5 letters or more are there in `exatext1`?

Solution:

The answer is 564. Here is one way of calculating this:

```
grep '[a-z][a-z][a-z][a-z][a-z]' exa_tokens | wc -l
```

Exercise:

How many different words of exactly 5 letters are there in `exatext1`?

Solution:

The answer is 50:

```
grep '^[a-z][a-z][a-z][a-z][a-z]$' exa_tokens | sort | uniq | wc -l
```
CHAPTER 3. TOOLS FOR FINDING AND DISPLAYING TEXT

Exercise:
What is the most frequent 7-letter word in `exatext1`?

Solution:
The most frequently occurring 7-letter word is “routing”; it occurs 8 times. You can find this by doing
```
grep '^[a-z][a-z][a-z][a-z][a-z][a-z][a-z]$' exa_tokens
```
and then piping it through
```
sort | uniq -c | sort -nr | head -1.
```

Exercise:
List all words with exactly two non-consecutive vowels.

Solution:
You want to search for words that have 1 vowel, then 1 or more non-vowels, and then another vowel. The “1 or more non-vowels” can be expressed using + in `egrep`:
```
egrep '^[^aeiou]*[aeiou][^aeiou]+[aeiou][^aeiou]*$' exa_tokens
```

Exercise:
List all words in `exatext1` ending in “-ing”. Which of those words are morphologically derived words? (Hint: `spell -v` shows morphological derivations.)

Solution:
Let’s start from `exa_types_alphab`, the alphabetical list of all word types in `exatext1`. To find all words ending in “-ing” we need only type
```
grep 'ing$' exa_types_alphab
```
This includes words like “string”. To see the morphologically derived “-ing”-forms, we can use `spell -v`:
```
grep 'ing$' exa_types_alphab | spell -v
```
which shows the morphological derivations.

3.5 Selecting fields

3.5.1 AWK commands

Sometimes it is useful to think of the lines in a text as records in a database, with each word being a “field” in the record. There are tools for extracting certain fields from database records, which can also be used for extracting certain words from lines. The most important for these purposes is the `awk`
programming language. This is a language which can be used to scan lines in a text to detect certain patterns of text.

For an overview of awk syntax, Aho, Kernighan and Weinberger (1988) is recommended reading. We briefly describe a few basics of awk syntax, and provide a full description of two very useful awk applications taken from the book.

To illustrate the basics of awk, consider first exatext2:

```
shop noun   41   32
shop verb   13   7
red adj     2    0
work noun   17   19
bowl noun   3    1
```

Imagine that this is a record of some text work you have done. It records that the word “shop” occurs as a noun 41 times in Text A and 32 times in Text B, “red” doesn’t occur at all in Text B, etc.

awk can be used to extract information from this kind of file. Each of the lines in exatext2 is considered to be a record, and each of the records has 4 fields. Suppose you want to extract all words that occur more than 15 times in Text A. You can do this by asking awk to inspect each line in the text. Whenever the third field is a number larger than 15, it should print whatever is in the first field:

```
awk '$3 > 15 {print $1}' < exatext2
```

This will return `shop` and `work`.

You can ask it to print all nouns that occur more than 10 times in Text A:

```
awk '$3 > 10 && $2 == "noun" {print $1}' < exatext2
```

You can also ask it to find all words that occur more often in Text B (field 4) than in Text A (field 3) (i.e. `$4 > $3`), and to print a message about the total number of times (i.e. `$3 + $4`) that item occurred:

```
awk '$4>$3 {print $1,"occurred",$3+$4,"times when used as a","$2 }' < exatext2
```

This will return:
work occurred 36 times when used as a noun

So the standard structure of an awk program is

```
awk pattern {action} < filename
```

*awk* scans a sequence of input lines (in this case from the file *filename*) one after another for lines that match the *pattern*. For each line that matches the pattern, *awk* performs the *action*. You can specify actions to be carried out before any input is processed using *BEGIN*, and actions to be carried out after the input is completed using *END*. We will see examples of both of these later.

To write the *patterns* you can use `$1`, `$2`, ... to find items in field 1, field 2, etc. If you are looking for an item in any field, you can use `$0`.

You can ask for values in fields to be greater, smaller, etc than values in other fields or than an explicitly given bit of information, using operators like `>` (more than), `<` (less than), `<=` (less than or equal to), `>=` (more than or equal to), `==` (equal to), `!=` (not equal to). Note that you can use `==` for strings of letters as well as numbers. You can also do arithmetic on these values, using operators like `+`, `-`, `*`, `^` and `/.

Also useful are assignment operators. These allow you to assign any kind of expression to a variable by saying `var = expr`.

For example, suppose we want to use *exatext2* to calculate how often nouns occur in Text A and how often in Text B. We search field 2 for occurrences of the string “noun”. Each time we find a match, we take the number of times it occurs in Text A (the value of field 3) and add that to the value of some variable `texta`, and add the value from field 4 (the number of times it occurred in Text B) to the value of some variable `textb`:

```
awk '$2 == "noun" {texta = texta + $3; textb = textb + $4}
END {print "Nouns: \"", texta, \"\" times in Text A and\", \"", textb, \"\" times in Text B\"\")}' < exatext2
```

The result you get is:

```
Nouns: 61 times in Text A and 52 times in Text B
```

Note that the variables `texta` and `textb` are automatically assumed to be 0; you don’t have to declare them or initialize them. Also note the use of *END*: 

```
```
the pattern and instruction are repeated until it doesn’t apply anymore. At that point, the next instruction (the print instruction) is executed.

You will have noticed the double quotes in patterns like `$2 == "noun"`. The double quotes mean that field 2 should be identical to the string “noun”. You can also put a variable there, in which case you don’t use the double quotes. Consider `exatext3` which just contains the following:

```
a
a
b
c
c
d
d
```

**Exercise:**
Can you see what the following will achieve?
```
awk '$1 != prev { print ; prev = $1}' < exatext3
```

**Solution:**
`awk` is doing the following: it looks in the first field for something which is not like `prev`. At first, `prev` is not set to anything. So the very first item (`a`) satisfies this condition; `awk` prints it, and sets `prev` to be `a`. Then it finds the next item in the file, which is again `a`. This time the condition is not satisfied (since `a` does now equal the current value of `prev`) and `awk` does not do anything. The next item is `b` This is different from the current value of `prev` So `b` is printed, and the value of `prev` is reset to `b` And so on. The result is the following:

```
a
b
c
d
```

In other words, `awk` has taken out the duplicates. The little `awk` program has the same functionality as `uniq`.

Another useful operator is `~` which means “matched by” (and `!~` which means “not matched by”). When we were looking for nouns in the second field we said:

```
awk '$2 == "noun"' < exatext2
```
In our example file `exatext2`, that is equivalent to saying

```
awk ' \$2 ~ /noun/ ' < exatext2
```

This means: find items in field 2 that match the string “noun”. In the case of `exatext2`, this is also equivalent to saying:

```
awk ' \$2 ~ /ou/ ' < exatext2
```

In other words, by using `~` you only have to match part of a string.

To define string matching operations you can use the same syntax as for `grep`:

```
awk ' \$0 !~ /nou/'
```

all lines which don’t have the string “nou” anywhere.

```
awk ' \$2 ~ /un$/'
```

all lines with words in the second field (`$2`) that end in `-un`.

```
awk ' \$0 ~ /^...$/'
```

all lines which have a string of exactly three characters (`^` indicates beginning, `$` indicates the end of the string, and `...` matches any three characters).

```
awk ' \$2 ~ /no|ad/'
```

all lines which have no or ad anywhere in their second field (when applied to `exatext2`, this will pick up noun and adj).

To summarise the options further:

- `^Z` matches a `Z` at the beginning of a string
- `Z$` matches a `Z` at the end of a string
- `^Z$` matches a string consisting exactly of `Z`
- `^..$` matches a string consisting exactly of two characters
- `\.$` matches a period at the end of a string
- `^\[ABC\]` matches an `A`, `B` or `C` at the beginning of a string
- `^[a-z]$` matches any character other than `A`, `B` or `C`
- `^[a-z]$` matches any character other than lowercase `a` to `z` at the end of a string
- `[the|an]` matches `the` or `an`
- `[a-z]*` matches strings consisting of zero or more lowercase characters
To produce the output we have so far only used the `print` statement. It is possible to format the output of `awk` more elaborately using the `printf` statement. It has the following form:

```plaintext
printf(format, value$_1$, value$_2$, \ldots, value$_n$)
```

`format` is the string you want to print verbatim. But the string can have variables in them (expressed as `%` followed by a few characters) which the `value` statements instantiate: the first `%` is instantiated by `value$_1$`, the second `%` by `value$_2$`, etc. The `%` is followed by a few characters, which indicate how the variable should be formatted. Here are a few examples:

- `%d` means “format as a decimal integer”—so if the value is `31.5`, `printf` will print `31`;
- `%s` means “print as a string of characters”;
- `%.4s` means “print as a string of characters, 4 characters long”—so if the value is `banana` `printf` will print `bana`;
- `%g` means “print as a digit with non-significant zeros suppressed”;
- `%7d` means “print as a decimal character, left-aligned in a field that is 7 characters wide.

For example, on page 40 we gave the following `awk`-code:

```awk
awk '$2 == "noun" {texta = texta + $3; textb = textb + $4}
      END {print "Nouns: %g times in Text A and %g times in Text B\n", texta, textb}' < exatext2
```

That can be rewritten using the `printf` command as follows:

```awk
awk '$2 == "noun" {texta = texta + $3; textb = textb + $4}
      END {printf "Nouns: %g times in Text A and %g times in Text B\n", texta, textb}' < exatext2
```

Note that `printf` does not print white lines or line breaks. You have to add those explicitly by means of the newline command `\n`.

Let us now return to our text file, `exatext1` for some exercises.

**Exercise:**

List all words from `exatext1` whose frequency is exactly 7.

**Solution:**

This is the list:
You can get this result by typing

```
awk '$1 == 7 {print}' < exa_freq
```

**Exercise:**
Can you see what this pipeline will produce?

```
rev < exa_types_alphab | paste - exa_types_alphab | awk '$1 == $2'
```

(Note that `rev` does not exist on Solaris, but `reverse` offers a superset of its functionality. If you are on Solaris, use `alias rev 'reverse -c'` instead in this exercise.)

Notice how in this pipe-line the result of the first UNIX-command is inserted in the second command (the `paste` command) by means of a hyphen. Without it, `paste` would not know in which order to paste the files together.

**Solution:**
You reverse the list of word types and paste it to the original list of word types. So the output is something like

```
a     a
  tuoba about
evoba above
tcartsba abstract
```

Then you check whether there are any lines where the first item is the same as the second item. If they are, then they are spelled the same way in reverse—in other words, they are the palindromes in `exatext1`. Apart from the one-letter words, the only palindromic words in `exatext1` are `deed` and `did`.

**Exercise:**
Can you find all words in `exatext1` whose reverse also occurs in `exatext1`.
These will be the palindromes from the previous exercise, but if `evil` and `live` both occurred in `exatext1`, they should be included as well.

**Solution:**
Start the same way as before: reverse the type list but then just append it to the original list of types and sort it:

```
rev < exa_types_alphab | cat - exa_types_alphab | sort > temp
```

The result looks as follows:
 Whereas in the original `exa_types_alphab` would have occurred only once, it now occurs twice. That means that it must have occurred also in `rev exa_types_alphab`. In other words, it is a word whose reverse spelling also occurs in `exatext1`. We can find all these words by just looking for words in `temp` that occur twice. We can use `uniq -c` to get a frequency list of `temp`, and then we can use `awk` to find all lines with a value of 2 or over in the first field and print out the second field:

```
uniq -c < temp | awk '{print $2}'
```

The resulting list (excluding one-letter words) is:

```
a deed did no on saw was
```

**Exercise:**

How many word tokens are there in `exatext1` ending in `-ies`? Try it with `awk` as well as with a combination of other tools.

**Solution:**

There are 6 word tokens ending in `-ies`: `agencies` (twice), `categories` (three times) and `companies` (once). You can find this by using `grep` to find lines ending in `-ies` in the file of word tokens:

```
grep 'ies$' < exa_tokens
```

Or you can use `awk` to check in `exa_freq` for items in the second field that end in `-ies`:

```
awk '$2 ~ /ies$/' < exa_freq
```

**Exercise:**

Print all word types that start with `str` and end in `-g`. Again, use `awk` as well as a combination of other tools.

**Solution:**

The only word in `exatext1` starting in `str-` and ending in `-g` is “string”:

```
awk '$2 ~ /^str.*g$/ {print $2}' < exa_freq
```

Another possibility is to use `grep`:

```
grep '^str.*g$' exa_types_alphab
```

**Exercise:**

Suppose you have a stylistic rule which says one should never have a word ending in `-ing` followed by a word ending in `-ion`. Are there any sequences like that in `exatext1`?

**Solution:**

There are such sequences, viz. `training provision`, `solving categorisation`, `existing production`, and `training collection`. You can find them by creating a file of the bigrams in `exatext1` (we did this before; the file is called `exa_bigrams`) and then using `awk` as follows:

```
awk '$1~/ing$/ && $2~/ion$/' < exa_bigrams
```
Exercise:
Map `exa_tokens_alphab` into a file from which duplicate words are removed but a count is kept as to how often the words occurred in the original file.

Solution:
The simplest solution is of course `uniq -c`. But the point is to try and do this in `awk`. Let us first try and develop this on the simpler file `exatext3`. We want to create an `awk` program which will take this file and return:

```
2 a
1 b
3 c
2 d
```

In an earlier exercise (page 41) we have already seen how to take out duplicates using `awk`:

```
awk '$1 != prev { print ; prev = $1}' < exatext3
```

Now we just have to add a counter. Let us assume `awk` has just seen an `a`; the counter will be at 1 and `prev` will be set to `a`. We get `awk` to look at the next line. If it is the same as the current value of `prev`, then we add one to the counter `n`. So if it sees another `a`, the counter goes up to 2. `awk` looks at the next line. If it is different from `prev` then we print out `n` as well as the current value of `prev`. We reset the counter to 1. And we reset `prev` to the item we’re currently looking at. Suppose the next line is `b`. That is different from the current value of `prev`. So we print out `n` (i.e. 2) and the current value of `prev` (i.e. `a`). We reset the counter to 1. And the value of `prev` is reset to `b`. And we continue as before.

We can express this as follows:

```
awk '$1==prev {n=n+1}; 
$1 != prev {print n, prev; n = 1; prev = $1}' < exatext3
```

If you try this, you will see that you get the following result:

```
2 a
1 b
3 c
```

It didn’t print information about the frequency of the `d`s. That is because it only printed information about `a` when it came across a `b`, and it only printed information about `b` when it came across `c`. Since `d` is the last element in the list, it doesn’t get an instruction to print information about `d`.

So we have to add that, once all the instructions are carried out, it should also print the current value of `n` followed by the current value of `prev`:```
3.5.2 AWK as a programming language

As you can see in the preceding exercises, the `awk` commands can easily become quite long. Instead of typing them to your UNIX prompt, it is useful to put them into files and execute them as programs. Indeed, `awk` is more than a mere UNIX tool and should really be seen as a programming language in its own right. Historically, `awk` is the result of an effort (in 1977) to generalize `grep` and `sed`, and was supposed to be used for writing very short programs. That is what we have seen so far, but modern-day `awk` can do more than this.

The `awk` code from the previous exercise (page 46) can be saved in a file as follows:

```
#!/bin/nawk -f
# input: text tokens (one per line, alphabetical)
# output: print number of occurrences of each word
$n1==prev {n=n+1}; $1!=prev {print n, prev; n = 1; prev = $1}
END {print n, prev}'
< exatext3
```

The file starts with a standard first line, whose only purpose is to tell UNIX to treat the file as an `awk` program.

Then there are some further comments (preceded by the hashes); they are not compulsory, but they will help you (and others) remember what your `awk`-script was intended for. You can write any text that you like here, provided that you prefix each line with `#`. It is good practice to write enough comments to make the purpose and intended usage of the program evident, since even you will probably forget this information faster than you think.

Then there is the program proper. Note the differences from the way you type `awk` commands at the UNIX prompt—you don’t include the instructions in single quotes, and the code can be displayed in a typographical layout that makes it more readable: blank lines can be added before or after statements, and tabs and other white space can be added around operators, all
to increase the readability of the program. Long statements can be broken after commas, and comments can be added after each broken line. You can put several statements on a single line if they are separated by semicolons. And the opening curly bracket of an action must be on the same line as the pattern is accompanies. The rest of the action can be spread over several lines, as befits the readability of the program.

If you save the above file as uniqc.awk and make sure it’s executable, then the command uniqc.awk exatext3 will print out the desired result.

One of awk’s distinguishing features is that it has been tuned for the creation of text-processing programs. It can be very concise because it uses defaults a lot. For example, in general it is true that awk statements consist of a pattern and an action:

```
    pattern { action }
```

If however you choose to leave out the action, any lines which are matched by the pattern will be printed unchanged. In other words, the default action in awk is `{ print }`. This is because most of the time text processing programs do want to print out whatever they find. Similarly, if you leave out the pattern the default pattern will match all input lines. And if you specify the action as `print` without specifying an argument, what gets printed is the whole of the current input line.

For example, try the following:

```
nawk 'gsub("noun","nominal") {print}' < exatext2
```

The `gsub` function globally substitutes any occurrence of “noun” by “nominal”. The `print` action does not say explicitly what should be printed, and so just prints out all the matching lines. And if you leave off the `print` statement, it will still perform that same `print` action. `nawk 'gsub("noun","nominal")' < exatext2` gives the following result:

```
shop nominal 41 32
work nominal 17 19
bowl nominal 3 1
```

Or consider:

```
nawk '$_2=substr($2,1,3) {print}' < exatext2
```
\textit{substr} creates substrings: in the second field it will not print the entire field but a substring, starting at position 1 and lasting for 3 characters: in other words, “noun” will be replaced by “nou” and “verb” by “ver”. That is the meaning of \texttt{substr($2,1,3$)}. However, the \texttt{print} command does not have an argument and \texttt{awk} prints by default the entire line where this change has been carried out, not just the affected field, giving the following result:

\begin{verbatim}
shop nou 41 32
shop ver 13 7
red adj 2 0
work nou 17 19
bowl nou 3 1
\end{verbatim}

An important type of statement in the writing of \texttt{awk} code is the \texttt{for}-statement. Its general syntax looks as follows:

\begin{verbatim}
for (expression1; expression2; expression3) 
    statement 
\end{verbatim}

And here’s an example:

\begin{verbatim}
awk '{for(i=1; i <= NF; i++) print $i}' < exatext2
\end{verbatim}

It says: set variable \texttt{i} to 1. Then, if the value of \texttt{i} is less than or equal to the number of fields in the file (for which \texttt{awk} uses the built-in variable \texttt{NF}), then print that field and increase \texttt{i} by 1 (instead of writing \texttt{i=i+1} you can just write \texttt{i++}). In other words, this program prints all input fields, one per line.

\textbf{Exercise:}
Can you see which of the UNIX commands discussed in section 3.1 this \texttt{awk} code corresponds to?

\begin{verbatim}
nawk '{for(i=1; i <= NF; i++) print $i}' < exatext1
\end{verbatim}

\textbf{Solution:}
The output corresponds to what you get when you do

\begin{verbatim}
tr -cs 'A-Za-z' '\012' < exatext1
\end{verbatim}

Finally, it is useful to understand how \texttt{awk} deals with arrays. Like variables, arrays just come into being simply by being mentioned. For example, the following code can be used with \texttt{exatext2} to count how many nouns there are in Text A:
The program accumulates the occurrences of nouns in the array `freq`. Each time an occurrence of “noun” is found, the value associated with the array `freq` is increased by whatever number occurs in the third field ($3$). The `END` action prints the total value. The output is:

There are 61 words of category noun in Text A

To count all occurrences of all categories in Text A, you can combine this use of arrays with a `for`-statement:

```
#!/bin/nawk -f
# for use with exatext2
{freq[$2] += $3}
END { for (category in freq)
    print "There are", freq[category], "words of type", category, "in Text A"}
```

The `for`-statement says that for any category in the array `freq` (i.e. any category occurring in the second field) you increase the value for that category by whatever value is found in `$3`. So when the program looks at the first line of `exatext2`, it finds a “noun” in `$2`; an array named `freq(noun)` is created and its value is increased by 41 (the number in `$3` for that line. Next it finds a “verb” in `$2` and creates and array `freq(verb)` and increases its value from 0 to 13 (the value of `$3` on that line). When it comes across the fourth line, it finds another “noun” in `$2` and increases the value of the array `freq(noun)` by 17. When it has looked at all the lines in `exatext2` it prints for each “category” the final value for `freq[category]`:

There are 61 words of type noun in Text A
There are 2 words of type adj in Text A
There are 13 words of type verb in Text A
Exercise:
To summarise what we have seen so far about awk here is a program which counts the words in a file. It is also available as wc.awk.

```awk
1 #!/bin/nawk -f
2 # wordfreq -- print number of occurrences of each word
3 # input: text
4 # output: print number of occurrences of each word
5 { gsub(/\[.,:;!?'(){}]/, "")
6   for(i=1; i <= NF; i++)
7     count[$i]++
8 }
9 END {for (w in count)
10   print count[w],w | "sort -rn"
11 }
```

The line numbers are not part of the program, and the program will not work if they are left in, but they make it easier to refer to parts of the program. See if you can work out how the program works.

Solution:
Here is what the program file has in it:

- The first line tells UNIX to treat the file as an awk program.
- There are then some comments (lines 2-4) preceded by # which indicate the purpose of the program.
- There is no BEGIN statement, because there is no need for anything to happen before any input is processed.
- There is a main body (lines 5-8) which is carried out every time awk sees an input line. Its purpose is to isolate and count the individual words in the input file: every time awk sees a line it sets up the fields to refer to parts of that line, then executes the statements within the curly braces starting at line 5 and ending on line 8. So these statements will be executed many times, but each time the fields will refer to a different line of the input file.
- There is an END statement (lines 9-11), which is executed once after the input has been exhausted. Its purpose is to print out and sort the accumulated counts.

The main body of the program (lines 5-8) does the following:

- Globally deletes punctuation (line 5), by using awk’s gsub command to replace punctuation symbols with the null string.
• Sets up a variable $i$, which is used as a loop counter in a for-loop. The for statement causes awk to execute the statements in the body of the loop (in this case just the \texttt{count[\$i]++} statement on line 7) until the exit-condition of the loop is satisfied. After each execution of the loop body, the loop counter is incremented (this is specified by the \texttt{i++} statement on line 6). The loop continues until it is no longer true that $i \leq NF$ (awk automatically sets up \texttt{NF} to contain the number of fields when the input line is read in). Taken together with the repeated execution caused by the arrival of each input line, the net effect is that \texttt{count[\$i]++} is executed once for every field of every line in the file.

Putting all this together, the effect is that the program traverses the words in the file, relying on awk to automatically split them into fields, and adding 1 to the appropriate count every time it sees a word.

Once the input has been exhausted, \texttt{counts} contains a count of word-tokens for every word-type found in the input file. This is what we wanted, but it remains to output the data in a suitable format.

The simplified version of the output code is:

\begin{verbatim}
END {for (w in count) 
    print count[w],w 
}
\end{verbatim}

This is another for loop: this time one which successively sets the the variable \texttt{w} to all the keys in the \texttt{count} array. For each of these keys we print first the count \texttt{count[w]} then the key itself \texttt{w}.

The final refinement is to specify that the output should be fed to the Unix sort command before the user sees it. This is done by using a special version of the print command which is reminiscent of the pipelines you have seen before.

\begin{verbatim}
END {for (w in count) 
    print count[w],w | "sort -rn" 
}
\end{verbatim}

Doing \texttt{wc.awk exatext1} gives the following result:

\begin{verbatim}
74 the 
42 to 
39 of 
... 
12 be 
11 on 
11 The 
10 document 
\end{verbatim}
It is not quite like what you find in `exa_freq` because in `exa_freq` we didn’t distinguish uppercase and lowercase versions of the same word. You can get exactly the same result as in `exa_freq` by doing:

```
tr 'A-Z' 'a-z' < exatext1 | wc.awk
```

**Exercise:**
What are the 10 most frequent suffixes in `exatext`? How often do they occur? Give three examples of each. (Hint: check the `man`-page for `spell` and the option `-v`.

**Solution:**
The solution looks as follows:

```
58 +s abstracters abstracts appears
16 +ed assigned called collected
11 +ing assigning checking consisting
11 +d associated based combined
10 -e+ing dividing handling incoming
10 +ly actually consequently currently
 5 +al conditional departmental empirical
 3 -y+ied classified identified varied
 3 +re representation representational research
 3 +er corner indexer number
```

A first step towards this solution is to use `spell` `-v` on all the words in `exatext1` and to sort them. We’ll store the results in a temporary file:

```
tr -cs 'a-z' '\012' < exatext1_lc | spell -v | sort > temp
```

`temp` contains the following information:

```
+able allowable
+al conditional
+al departmental
+al empirical
+al medical
+al technical
+al+ly empirically
...
```

Now we can write a little `awk` program that will take the information in `temp` and for each type of suffix collects all the occurrences of the suffixes. Let’s call this `awk`-file `suffix.awk`. Doing `suffix.awk temp` will result in:

```
+able allowable
+al conditional departmental empirical medical technical
+al+ly empirically typically
+d associated based combined compared compiled derived...
```
Then we can use awk again to print a maximum of three examples for each suffix and the total frequency of the suffix’s occurrence. For each line we first check how many fields there are. If the number of fields (NF) is 7, then we know that that line consists of a suffix in field one, followed by 6 words that have that suffix. So the total number of times the suffix occurred is NF - 1. We print that number, followed by the suffix (which is in field 1 in temp, followed by whatever is in fields 2, 3 and 4 (i.e. three examples):

```
suffix.awk temp | awk '{print NF-1, $1, $2, $3, $4}' | more
```

We can then use sort and head to display the most frequent suffixes. The total pipeline looks as follows:

```
suffix.awk temp|awk '{print NF-1,$1,$2,$3,$4}'|sort -nr|head-10
```

That just leaves the code for suffix.awk. Here is one possibility:

```
#!/bin/nawk -f
# takes as input the output of spell -v | sort
# finds a morpheme, displays all examples of it
$1==prev {printf "\t%s", $2}
$1!=prev {prev = $1
    printf "\n%s\t%s", $1, $2}
END {printf "\n"}
```

You should now have reached the point where you can work out what this awk code is doing for you.

We conclude the section with an exercise on counting bigrams instead of words. You have done this earlier using paste. It is just as easily done in awk.

**Exercise:**
Modify wc.awk to count bigrams instead of words. Hint: maintain a variable—call it prev—which contains the previous word. Note that in awk you can build a string with a space and assign it to xy in by saying xy = x " " y.

**Solution:**
Here is the awk solution. The changed lines are commented.

```awk
{ gsub(/[.,;!?()]/, "")
  for(i= 1; i <= NF; i++){
    bigram = prev " " $i # build the bigram
    prev = $i # keep track of the previous word
    count[bigram]++ # count the bigram
  }
}
```

```bash
END {for (w in count)
    print count[w],w | "sort -rn"
}
```
It is easy to verify that this gives the same results as the pipeline using `paste`.

We get the top ten bigrams from `exatext1` as follows:

```
tr 'A-Z' 'a-z' < exatext1 | bigrams.awk | head -10
```

with the result:

```
12 in the
8 to the
8 of the
6 the system
6 in a
5 the document
5 and routing
4 the text
4 the human
4 set of
```

This completes the description of `awk` as a programming language. If you like reading about programming languages, you might want to take time out to read about it in the manual. If, like me, you prefer to learn from examples and are tolerant of partial incomprehension, you could just carry on with these course notes.

### 3.6 PERL programs

All the UNIX facilities we have discussed so far are very handy. But the most widely used language for corpus manipulation is PERL. It is available free of charge and easy to install. The facilities are very similar to those of `awk` but the packaging is different. Here is the word count program re-expressed in `perl`. We’re not going to try to explain PERL in detail, because most of what you have learned about `awk` is more-or-less applicable to `perl`, and because all the evidence is that the people who need PERL find it easy to pick up. It is also available as `wc.perl`.

```perl
while(<>) {
    chop; # remove trailing newline
    tr/A-Z/a-z/; # normalize upper case to lower case
    tr/.,;:!"(){}//d; # kill punctuation
    foreach $w (split) { # foreach loop over words
        $count{$w} ++; # adjust count
    }
}
```
open(OUTPUT,"|sort -nr"); # open OUTPUT
while(($key,$value) = each %count) { # each loop over keys and values
  print OUTPUT "$value $key\n"; # pipe results to OUTPUT
}
close(OUTPUT); # remember to close OUTPUT

As in awk, when you program in PERL you don’t have to worry about declaring or initializing variables. For comparison here is the awk version repeated, with some extra comments.

```
{ gsub(/[.,;!?"(){}]/, "") # kill punctuation
  for(i= 1; i <= NF; i++) # for loop over fields
    count[$i]++ # adjust count
}
END {for (w in count) # for loop over keys
  print count[w],w | "sort -rn" # pipe output to sort process
}
```

The following are the important differences between wc.awk and wc.perl:

1. PERL uses different syntax. Variables are marked with $ and statements finish with a semi-colon. The array brackets are different too.

2. Where awk uses an implicit loop over the lines in the input file PERL uses an explicit while loop. Input lines are read in using <> . Similarly there is no END statement in PERL. Instead the program continues once the while loop is done.

3. Where awk has gsub, PERL has tr. You can see another use of tr in the line tr/A-Z/a-z/; . This is analogous to the Unix command tr which we saw earlier.

4. Where awk implicitly splits the fields of the input lines and sets NF, the PERL program explicitly calls split to break the line into fields.

5. PERL uses a foreach loop to iterate over the fields in the split line (underlyingly foreach involves an array of elements. In fact PERL has
3.6. PERL PROGRAMS

several sorts of arrays, and many other facilities not illustrated here). awk uses a C style for loop to carry out the same iteration.

6. Both programs finish off by outputting all the elements of the count array to a sort process. Where awk specifies the sort process as a kind of side-condition to the print statement, PERL opens a file handle to the sort process, explicitly closing it once its usefulness has been exhausted.

The general trend is that awk programs are more concise than sensibly written PERL programs. PERL also has a very rich infra-structure of pre-packaged libraries. Whatever you want to do, it is worth checking that there isn’t already a freely available PERL module for doing it.

awk, by contrast, is orderly and small, offering a very well chosen set of facilities, but lacking the verdant richness and useful undergrowth of PERL. The definitive awk text by Aho, Weinberger and Kernighan is 210 pages of lucid technical writing, whereas PERL has tens of large books written about it. We particularly recommend “Learning Perl” by Randal Schwartz. It is unlikely that you will ever feel constrained by PERL, but awk can be limiting when you don’t want the default behaviour of the input loop. To a great extent this will come down to a matter of personal choice. We prefer both, frequently at the same time, but for different reasons.

Exercise:
Modify wc.perl to count bigrams instead of words. You should find that this is a matter of making the same change as in the awk exercise earlier. In PERL you can build a string with a space in and assign it to $xy by saying $xy = "'$x $y'";

Solution:
The PERL solution is analogous to the awk one:

while(<>) {
    chop;
    tr/A-Z/a-z/;
    tr/.,;!?"{}//d;
    foreach $w (split) {
        $bigram = "$prev $w"; # make the bigram
        $prev = $w; # update the previous word
        $count{$bigram} ++; # count the bigram
    }
}
CHAPTER 3. TOOLS FOR FINDING AND DISPLAYING TEXT

open(OUTPUT,"|sort -nr");
while(($key,$value) = each %count) {
    print OUTPUT "$value $key\n"
}
close(OUTPUT);

You might want to think about how to generalize this program to produce trigrams, 4-grams and longer sequences.

3.7 Summary

In this chapter we have used some generally available UNIX tools which assist in handling large collections of text. We have shown you how to tokenise text, make lists of bigrams or 3-grams, compile frequency lists, etc.

We have so far given relatively little motivation for why you would want to do any of these things, concentrating instead on how you can do them. In the following chapters you will gradually get more of a feel for the usefulness of these basic techniques.

To conclude, here is a handy cheat sheet which summarises the basic UNIX operations discussed in this chapter.