Unix™ for Poets

Kenneth Ward Church
AT&T Research
kwc@research.att.com

Text is available like never before. Data collection efforts such as the Association for Computational Linguistics' Data Collection Initiative (ACL/DCI), the Consortium for Lexical Research (CLR), the European Corpus Initiative (ECI), ICAME, the British National Corpus (BNC), the Linguistic Data Consortium (LDC), Electronic Dictionary Research (EDR) and many others have done a wonderful job in acquiring and distributing dictionaries and corpora.[1] In addition, there are vast quantities of so-called Information Super Highway Roadkill: email, bboards, faxes. We now has access to billions and billions of words, and even more pixels.

What can we do with it all? Now that data collection efforts have done such a wonderful service to the community, many researchers have more data than they know what to do with. Electronic bboards are beginning to fill up with requests for word frequency counts, ngram statistics, and so on. Many researchers believe that they don't have sufficient computing resources to do these things for themselves. Over the years, I've spent a fair bit of time designing and coding a set of fancy corpus tools for very large corpora (eg, billions of words), but for a mere million words or so, it really isn't worth the effort. You can almost certainly do it yourself, even on a modest PC. People used to do these kinds of calculations on a PDP-11, which is much more modest in almost every respect than whatever computing resources you are currently using.

I wouldn't bring out the big guns (fancy machines, fancy algorithms, data collection committees, bigtime favors) unless you have a lot of text (e.g., hundreds of million words or more), or you are trying to count really long ngrams (e.g., 50-grams). This chapter will describe a set of simple Unix-based tools that should be more than adequate for counting trigrams on a corpus the size of the Brown Corpus. I'd recommend that you do it yourself for basically the same reason that home repair stores like DIY and Home Depot are as popular as they are. You can always hire a pro to fix your home for you, but a lot of people find that it is better not to, unless they are trying to do something moderately hard. Hamming used to say it is much better to solve the right problem naively than the wrong problem expertly.

I am very much a believer in teaching by examples. George Miller (personal communication) has observed that dictionary definitions are often not as helpful as example sentences. Definitions make a lot of sense if you already basically know what the word means, but they can be hard going if you have never seen the word before. Following this spirit, this chapter will focus on examples and avoid definitions whenever possible. In some cases, I will deliberately use new options and even new commands without defining them first. The reader is encouraged to try the examples themselves, and when all else fails consult the documentation. (But hopefully, that shouldn't be necessary too often, since we all know how boring the documentation can be.)

We will show how to solve the following exercises using only very simple utilities.

1. Count words in a text

[1] For more information on the ACL/DCI and the LDC, see http://www.cis.upenn.edu/~ldc. The CLR's web page is: http://clr.nmsu.edu/clr/CLR.html, and EDR's web page is: http://www.iijnet.or.jp/edr. Information on the ECI can be found in http://www.cogsci.ed.ac.uk/elsnet/eci_summary.html, or by sending email to eucorp@cogsci.edinburgh.ac.uk. Information on teh BNC can be found in http://info.ox.ac.uk/bnc, or by sending email to smbowie@vax.oxford.ac.uk. Information on the London-Lund Corpus and other corpora available though ICAME can be found in the ICAME Journal, edited by Stig Johansson, Department of English, University of Oslo, Norway.

2. Sort a list of words in various ways
   - ascii order
   - dictionary order
   - ‘‘rhyming’’ order
3. Compute ngram statistics
4. Make a Concordance

The code fragments in this chapter were developed and tested on a Sun computer running Berkeley Unix. The code ought to work on more or less as is in any Unix system, and even in many PC environments, running various compatibility packages such as the MKS toolkit.

1. Exercise 1: Count words in a text

The problem is to input a text file, say Genesis (a good place to start),[2] and output a list of words in the file along with their frequency counts. The algorithm consists of three steps:

1. Tokenize the text into a sequence of words (tr),
2. Sort the words (sort), and
3. Count duplicates (uniq –c).

The algorithm can be implemented in just three lines of Unix code:

```
tr -sc ’[A-Z][a-z]’ ’[\012*]’ < genesis |
sort |
uniq -c > genesis.hist
```

This program produces the following output:

```
   1
   2 A
   8 Abel
   1 Abelmizraim
   1 Abidah
   1 Abide
   1 Abimael
  24 Abimelech
 134 Abraham
  59 Abram
     ...
```

WARNING: in some Unix environments, the arguments to tr should be

```
tr -sc ’A-Za-z’ ’\012’ < genesis
```

We will try to avoid confusing the reader with the details of all the different dialects of Unix, but

[2] I am assuming that the reader has access to an electronic copy of the bible. I did a quick search in Alta Vista on Netscape and found a copy in http://arthur.cs.wwu.edu/˜phil/kjv. I’m sure this is just one of many web sites.

periodically, we’ll mention some things to try if one of the examples doesn’t happen to work in your environment.

The less than symbol ‘‘<’’ indicates that the input should be taken from the file named ‘‘genesis,’’ and the greater than symbol ‘‘>’’ indicates taht the output should be redirected to a file named ‘‘genesis.hist.’’ By default, input is read from stdin (standard input) and written to stdout (standard output). Standard input is usually the keyboard and standard output is usually the active window.

The pipe symbol ‘‘|’’ is used to connect the output of one program to the input of the next. In this case, tr outputs a sequence of tokens (words) which are then piped into sort. Sort outputs a sequence of sorted tokens which are then piped into uniq, which counts up the runs, producing the desired result.

We can understand this program better by breaking it up into smaller pieces. Lets start by looking at the beginning of the input file. The Unix program ‘‘sed’’ can be used as follows to show the first five lines of the genesis file:

```
sed 5q < genesis
#Genesis
1:1 In the beginning God created the heaven and the earth.
1:2 And the earth was without form, and void; and darkness [...]
1:3 And God said, Let there be light: and there was light.
1:4 And God saw the light, that [it was] good: and God [...]
```

In the same way, we can use sed to verify that the first few tokens generated by tr do indeed correspond to the first few words in the genesis file.

```
tr -sc ’[A-Z][a-z]’ ’[\012*]’ < genesis | sed 5q

Genesis
In
the
beginning
```

Similarly, we can verify that the output of the sort step produces a sequence of (not necessarily distinct) tokens in lexicographic order.

```
tr -sc ’[A-Z][a-z]’ ’[\012*]’ < genesis |
sort | sed 5q

A
A
Abel
Abel
```

Finally, the uniq step counts up the repeated tokens.

```
tr -sc ’[A-Z][a-z]’ ’[\012*]’ < genesis |
sort | uniq -c | sed 5q

   1
   2 A
   8 Abel
   1 Abelmizraim
   1 Abidah
```

2. More Counting Exercises

This section will show three variants of the counting program above to illustrate just how easy it is to count a wide variety of (possibly) useful things. The details of the examples aren’t all that important. The point is to realize that pipelines are so powerful that they can be easily extended to count words (by almost any definition one can imagine), ngrams, and much more.

The examples in this section will discuss various (weird) definitions of what is a ‘‘word.’’ Some students get distracted at this point by these weird definitions and lose sight of the point – that pipelines make it relatively easy to mix and match a small number of simple programs in very powerful ways. But even these students usually come around when they see how these very same techiques can be used to count ngrams, which is really nothing other than yet another weird definition of what is a word/token.

The first of the three examples shows that a straightforward one-line modification to the counting program can be used to merge the counts for upper and lower case. The first line in the new program below collapses the case distinction by translating lower case to upper case:

```
tr ’[a-z]’ ’[A-Z]’ < genesis |
tr -sc ’[A-Z]’ ’[\012*]’ |
sort |
uniq -c
```

Small changes to the tokenizing rule are easy to make, and can have a dramatic impact. The second example shows that we can count vowel sequences instead of words by changing the tokenizing rule (second tr line) to emit sequences of vowels rather than sequences of alphabetic characters.

```
tr ’a-z’ ’[A-Z]’ < genesis |
tr -sc ’AEIOU’ ’[\012*]’|
sort |
uniq -c
```

The third and final counting example is just like the second, except that it counts sequences of consonants rather than sequences of words.

```
tr ’[a-z]’ ’[A-Z]’ < genesis |
tr -sc ’BCDFGHJKLMNPQRSTVWXYZ’ ’[\012*]’ |
sort |
uniq -c
```

These three examples are intended to show how easy it is to change the definition of what counts as a word. Sometimes you want to distinguish between upper and lower case, and sometimes you don’t. Sometimes you want to collapse morphological variants (does *hostage* = *hostages*). Different languages use different character sets. Sometimes I get email from Sweden, for example, where ‘‘{’’ is a vowel. The tokenizer depends on what you are trying to do. The same basic counting program can be used to count a variety of different things, depending on how you implement the definition of thing (=token).

You can find the documentation for the tr command (and many other commands as well) by saying

```
man tr
```

If you want to see the document for some other command, simply replace the ’tr’ with the name of that other command.

The man page for tr explains that tr inputs a character at a time and outputs a translation of the character, depending on the options. In a simple case like

```
tr ’[a-z]’ ’[A-Z]’
```

tr translates lowercase characters to uppercase characters. The first argument, ‘‘[a-z],’’ specifies lowercase characters and the second argument, ‘‘[A-Z],’’ specifies uppercase characters. The specification of characters is more or less standard across most Unix commands (though be warned that there are more surprises than there should be). The notation, ‘‘[x-y],’’ denotes a character between ‘‘x’’ and ‘‘y’’ in ascii order. Ascii order is a generalization of alphabetic order, including not only the standard English characters, a-z, in both upper and lower case, but also digits, 0-9, puctuation and white space. It is important to mention, though, that ascii is an American standard and does not generalize well to European character sets, let alone Asian characters. Some (but not all) Unix systems support European/Asian/wordwide standards such as Latin1, EUC and Unicode.

Some characters are difficult to input (because they mean different things to different programs). One of the worst of these is the newline character, since newline is also used to terminate the end of a line/command (depending on the context). To get around some of these annoyances, you can specify a character by referring to its ascii code in octal (base 8). If this sounds cryptic, don’t worry about it; it is. All you need to know is that ‘‘[\012*]’’ is a newline.

The optional flag, ‘‘–c,’’ complements the translation, so that

```
tr -c ’[a-z][A-Z]’ ’[\012*]’
```

translates any non-alphabetic character to newline. Non-alphabetic characters include puctuation, white space, control characters, etc. It is easier to refer to non-alphabetic characters by referring to their complement because many of them (like newline) are hard to refer to (without knowing the context).

The optional flag, ‘‘–s,’’ squeezes out multiple instances of the translation, so that

```
tr -c ’[a-z][A-Z]’ ’[\012*]’
```

won’t output more than one newline in a row. I probably shouldn’t have used this fancy option in the first example, because it doesn’t change the results very much (and I wanted to establish the principle that it is ok to use fancy options without explaining them first).

3. sort

The sorting step can also be modified in a variety of different ways. The man page for sort describes a number of options or flags such as:

Example	Explanation
sort –d	dictionary order
sort –f	fold case
sort –n	numeric order
sort –nr	reverse numeric order
sort –u	remove duplicates
sort +1	start with field 1 (starting from 0)
sort +0.50	start with 50th character
sort +1.5	start with 5th character of field 1

These options can be used in straightforward ways to solve the following exercises:

1. Sort the words in Genesis by frequency

```
tr -sc ’[A-Z][a-z]’ ’[\012*]’ < genesis |
sort |
uniq -c |
sort –nr
```

2. Sort them by folding case.
3. Sort them by ‘‘rhyming’’ order.

The last two examples above are left as exercises for the reader, but unlike most ‘‘exercises for the reader,’’ the solutions can be found in the appendix of this chapter.

By ‘‘rhyming’’ order, we mean that the words should be sorted from the right rather than the left, as illustrated below:

```
   ...
 1 freely
 1 sorely
 5 Surely
15 surely
 1 falsely
 1 fly
   ...
```

‘‘freely’’ comes before ‘‘sorely’’ because ‘‘yleerf’’ (‘‘freely’’ spelled backwards) comes before ‘‘yleros’’ (‘‘sorely’’ spelled backwords) in lexicographic order. Rhyming dictionaries are often used to help poets (and linguists who are interested in morphology).

Hint: There is a Unix command ‘‘rev,’’ which reverses the letters on a line:

```
echo hello world | rev
dlrow olleh

echo hello world | rev | rev
hello world
```

Thus far, we have seen how Unix commands such as tr, sort, uniq, sed and rev can be combined into pipelines with the ‘‘|,’’ ‘‘<,’’ and ‘‘>’’ operators in simple yet powerful ways. All of the examples were based on counting words in a text. The flexibility of the pipeline approach was illustrated by showing how easy it was to modify the counting program to

- tokenize by vowel, merge upper and lower case
- sort by frequency, rhyming order

4. Bigrams

The same basic counting program can be modified to count bigrams (pairs of words), using the algorithm:

1. tokenize by word
2. print $word_i$ and $word_{i+1}$ on the same line
3. count

The second step makes use of two new Unix commands, tail and paste. Tail is usually used to output the last few lines of a file, but it can be used to output all but the first few lines with the ‘‘+2’’ option. The following code uses tail in this way to generate the files genesis.words and genesis.nextwords which correspond to $word_i$ and $word_{i+1}$.

```
tr -sc ’[A-Z][a-z]’ ’[\012*]’ < genesis > genesis.words
tail +2 genesis.words > genesis.nextwords
```

Paste takes two files and prints them side by side on the same line. Pasting genesis.words and genesis.nextwords together produces the appropriate input for the counting step.

```
paste genesis.words genesis.nextwords

 ...
And    God
God    said
said   Let
Let    there
 ...
```

Combining the pieces, we end up with the following four line program for counting bigrams:

```
tr -sc ’[A-Z][a-z]’ ’[\012*]’ < genesis > genesis.words
tail +2 genesis.words > genesis.nextwords

paste genesis.words genesis.nextwords |
sort | uniq -c > genesis.bigrams
```

The five most frequent bigrams in Genesis are:

```
sort –nr < genesis.bigrams | sed 5q
372   of     the
287   in     the
192   And    he
185   And    the
178   said   unto
```

Exercise: count trigrams. The solution can be found in the appendix.

I have presented this material in quite a number of tutorials over the years. The students almost always come up with a big ‘‘aha’’ reaction at this point. Counting words seems easy enough, but for some reason, students are almost always pleasantly surprised to discover that counting ngrams (bigrams, trigrams, 50-grams) is just as easy as counting words. It is just a matter of how you tokenize the text. We can tokenize the text into words or into ngrams; it makes little difference as far as the counting step is concerned.

5. Shell Scripts

Suppose that you found that you were often computing trigrams of different things, and you found it inconvenient to keep typing the same five lines over and over. If you put the following into a file called ‘‘trigram,’’

```
tr -sc '[A-Z][a-z]' '[\012*]' > $$words
tail +2 $$words > $$nextwords
tail +3 $$words > $$nextwords2

paste $$words $$nextwords $$nextwords2 |
sort | uniq -c

# remove the temporary files
rm $$words $$nextwords $$nextwords2
```

then you could count trigrams with a single line:

```
sh trigram < genesis > genesis.bigram
```

The ‘‘sh’’ command (pronounced ‘‘shell’’) is used to execute a shell script. PCs and DOS have basically the same concept: shell scripts are called ‘‘bat’’ or batch files, and they are invoked with the ‘‘run’’ command rather than ‘‘sh.’’ (Unix command names tend to be short and rarely contain vowels.)

This example made use of a new command, ‘‘rm,’’ that deletes (removes) files. ‘‘rm’’ is basically the same as ‘‘del’’ in DOS, but be careful! Don’t expect to be asked for confirmation. (Unix does not believe in asking lots of interactive questions.)

The shell script also introduced several new symbols. The ‘‘#’’ symbol is used for comments. The ‘‘$$’’ syntax encodes a long number (the process id) into the names of the temporary files. It is a good idea to use the ‘‘$$’’ syntax in this way so that two users (or two processes) can run the shell script at the same time, and there won’t be any confusion about which temporary file belongs to which process. (If you don’t know what a process is, don’t worry about it. A process is a job, an instance of a program that is assigned resources by the scheduler, the time sharing system.)

6. grep & egrep: An Example of a Filter

After completing the trigram exercise, you will have discovered that ‘‘the land of’’ and ‘‘And he said’’ are the two most frequent trigrams in Genesis. Suppose that you wanted to count trigrams separately for verses that contain just the phrase ‘‘the land of.’’

Grep (general regular expression pattern matcher) can be used as follows to extract lines containing ‘‘the land of.'' (WARNING: Because the ''lines'' were too long to fit the format required for publication, I have had to insert additional newlines.)

```
grep 'the land of' genesis | sed 5q

4:16 And Cain went out from the presence of the LORD, and dwelt in
the land of Nod, on the east of Eden.

10:10 And the beginning of his kingdom was Babel, and Erech, and
Accad, and Calneh, in the land of Shinar.

11:2 And it came to pass, as they journeyed from the east, that they
found a plain in the land of Shinar; and they dwelt there.

11:28 And Haran died before his father Terah in the land of his
nativity, in Ur of the Chaldees.

11:31 And Terah took Abram his son, and Lot the son of Haran his
son's son, and Sarai his daughter in law, his son Abram's wife; and
they went forth with them from Ur of the Chaldees, to go into the
land of Canaan; and they came unto Haran, and dwelt there.
```

Grep can be combined in straightforward ways with the trigram shell script to count trigrams for lines matching any regular expression including ''the land of'' and ''And he said.''

```
grep 'the land of' genesis |
sh trigram | sort -nr | sed 5q
101  the   land
 55  in    the
 35  land  of
 35  land  of
 17  all   the

grep 'And he said' genesis |
sh trigram | sort -nr | sed 5q
84  And   he
17  he    said
16  he    said
 9  said  unto
 9  said  I
```

The syntax for regular expressions can be fairly ellaborate. The simplest case is like the ''the land of'' example above, where we are looking for lines containing a string. In more ellaborate cases, matches can be anchored at the beginning or end of lines by using the ''^'' and ''$'' symbols.

Example	Explanation
grep gh	find lines containing ''gh''
grep '^con'	find lines beginning with ''con''
grep 'ing$'	find lines ending with ''ing''

With the –v option, instead of printing the lines that match, grep prints lines that do not match. In other words, matching lines are filtered out or deleted from the output.

Example	Explanation
grep –v gh	delete lines containing ‘‘gh’’
grep –v ’^con’	delete lines beginning with ‘‘con’’
grep –v ’ing$’	delete lines ending with ‘‘ing’’

The –c options counts the matches instead of printing them.

Case distinctions are ignored with the –i option, so that

```
grep ’[aeiouAEIOU]’
```

is equivalent to

```
grep -i ’[aeiou]’
```

Grep allows ranges of characters (somewhat) like tr.

Example	Explanation
grep ’[A–Z]’	lines with an uppercase char
grep ’^[A–Z]’	lines starting with an uppercase char
grep ’[A–Z]$’	lines ending with an uppercase char
grep ’^[A–Z]*$’	lines with all uppercase chars
grep ’[aeiouAEIOU]’	lines with a vowel
grep ’^[aeiouAEIOU]’	lines starting with a vowel
grep ’[aeiouAEIOU]$’	lines ending with a vowel

Warning: the ‘‘^’’ can be confusing. Inside square brackets as in ‘‘[^a-zA-Z],’’ it no longer refers to the beginning of the line, but rather, it complements the set of characters. Thus, ‘‘[^a-zA-Z]’’ denotes any non-alphabetic character.

Regular expressions can be combined recursively in ellaborate ways. For example,

Example	Explanation
grep –i ’[aeiou].*[aeiou]’	lines with two or more vowels
grep –i ’^[^aeiou]*[aeiou][^aeiou]*$’	lines with exactly one vowel

A quick summary of the syntax of regular expressions is given in the table below. The syntax is slightly more general for egrep. (grep and egrep are basically the same, though egrep is often more efficient.)

Example	Explanation
a	match the letter ‘‘a’’
[a–z]	match any lowercase letter
[A–Z]	match any uppercase letter
[0–9]	match any digit
[0123456789]	match any digit
[aeiouAEIUO]	match any vowel

[ˆaeiouAEIOU]	match any letter but a vowel
.	match any character
ˆ	beginning of line
$	end of line
*x**	any number of *x*
x+	one or more of *x* (egrep only)
x|*y*	*x* or *y* (egrep only)
(*x*)	override precedence rules (egrep only)

Grep Exercises

1. How many uppercase words are there in Genesis? Lowercase? Hint: `wc -l` or `grep -c`
2. How many 4-letter words?
3. Are there any words with no vowels?
4. Find ‘‘1-syllable’’ words (words with exactly one vowel)
5. Find ‘‘2-syllable’’ words (words with exactly two vowels)
6. Some words with two orthographic vowels have only one phonological vowel. Delete words ending with a silent ‘‘e’’ from the 2-syllable list. Delete diphthongs (sequences of two vowels in a row).
7. Find verses in Genesis with the word ‘‘light.’’ How many have two or more instances of ‘‘light’’? Three or more? Exactly two?

7. sed (string editor)

We have been using

```
sed 5q < genesis
```

to print the first 5 lines. Actually, this means quit after the fifth line. We could have also quit after the first instance of a regular expression

```
sed ’/light/q’ genesis
```

Sed is also used to substitute regions matching a regular expression with a second string. For example:

```
sed ’s/light/dark/g’
```

will replace all instances of light with dark. The first argument can be a regular expression. For example, as part of a simple morphology program, we might insert a hyphen into words ending with ‘‘ly.’’

```
sed ’s/ly$/-ly/g’
```

The substitution operator can also be used to select the first field on each line by replacing everything after the first white space with nothing. (Note: the square brackets contain a space followed by a tab.)

```
sed 's/[    ].*//g'
```

sed exercises

1. Count morphs in genesis
   Hint: use `spell -v` to extract morphs,
   select first field and count

   ```
   echo darkness | spell -v
   +ness darkness
   ```

2. Count word initial consonant sequences: tokenize by word, delete the vowel and the rest of the word, and count

3. Count word final consonant sequences

8. awk

Awk is a general purpose programming language, though generally intended for shorter programs (1 or 2 lines). The name ‘‘awk’’ is derived from the names of the three authors: Alfred Aho, Peter Weinberger and Brian Kernighan.

WARNING: There are many obsolete versions of awk still in common use, especially on Sun computers. You may need to use nawk (new awk) or gawk (Gnu awk) instead of awk, if your system is still using an obsolete version of awk.

8.1 Selecting Fields by Position

Awk is especially good for manipulating lines and fields in simple ways.

Example	Explanation
awk '{print $1}'	print first field
awk '{print $2}'	print second field
awk '{print $NF}'	print last field
awk '{print $(NF-1)}'	print penultimate field
awk '{print $0}'	print the entire record (line)
awk '{print $NF}'	print the number of fields

Exercise: sort the words in Genesis by the number of syllables (sequences of vowels)

8.2 Filtering by Numerical Comparison

Example: find words that appear 300 times or more in Genesis:

```
tr -sc '[A-Z][a-z]' '[\012*]' < genesis |
sort | uniq -c |
awk '$1 > 300 {print $2}'
And
I
a
and
he
him
his
in
my
of
said
that
the
to
unto
was
```

The default action is {print $0}. Thus,

```
awk '$1 > 300 {print $0}'
```

is equivalent to

```
awk '$1 > 300'
```

Awk supports the following predicates (functions that return truth values):

>	greater than
<	less than
>=	greater than or equal
<=	less than or equal
==	equal
!=	not equal
&&	logical and
||	logical or
!	logical negation
˜	regular expression match
!˜	regular expression mismatch

Exercises:

1. find vowel sequences that appear at least 1000 times
2. find bigrams that appear exactly twice

8.3 Filtering by String Comparison

The ‘‘==’’ predicate can also be used on strings as illustrated in the following exercise.

Exercise: Find palindromes in Genesis.

```
sort -u genesis.words > genesis.types
rev < genesis.types > genesis.types.rev
paste genesis.types genesis.types.rev | awk ’$1 == $2’
A      A
I      I
O      O
a      a
deed   deed
did    did
ewe    ewe
noon   noon
s      s
```

Exercise: Find words in Genesis that can also be spelled both forwards and backwards. The list should include palindromes like ‘‘deed,’’ as well as words like ‘‘live’’ whose reverse (‘‘evil’’) is also a word, though not the same one. Hint: make a file of the words in Genesis, and a second file of those words spelled backwards. Then look for words that are in both files. One way to find words that are in both files is to sort them together with

```
sort <file1> <file2>
```

and count and look for words with a frequency of 2.

Exercise: Compare two files, say exodus and genesis. Find words that are in just the first file, just the second, and both. Do it with the tools that we have discussed thus far, and then do a `man` on `comm,` and do it again.

8.4 Filtering and Counting in AWK

Suppose that one wanted to find words ending with ‘‘ed’’ in genesis.hist. Then using the tools we have thus far, we could say

```
grep ’ed$’ genesis.hist
```

Alternatively, we can do the regular expression matching in awk using the ˜ operator.

```
awk ’$2 ˜ /ed$/’ genesis.hist
```

Counting can also be done using the tools introduced previously (or not). Suppose we wanted to count the number of words that end in /ed$/ by token or by type.[3] Using the tools presented thus far, we could say:

[3] The terms tokens and types are intended to clarify a possibly confusion. Do the two instances of ‘‘to’’ in ‘‘to be or not to be’’ count as one word or two? We say that the two instances of ‘‘to’’ are two different tokens, but the same type. So, if we were counting words in the phrase ‘‘to be or not to be,’’ we would say that there were six tokens and four types.

```
# by token
tr -sc '[A-Z][a-z]' '[\012*]' < genesis | grep -c 'ed$'
# by type
tr -sc '[A-Z][a-z]' '[\012*]' < genesis | sort -u | grep -c 'ed$'
```

Alternatively, in awk, we could say:

```
# by token
awk '$2 ~ /ed$/ {x = x + $1}
     END        {print x}' genesis.hist

# by type
awk '$2 ~ /ed$/ {x = x + 1}
     END        {print x}' genesis.hist
```

‘‘x’’ is a variable. For each line that matches the regular expression, we accumulate into x either the frequency of the word ($1) or just 1. In either case, at the end of the input, we print out the value of x. These examples illustrate the use of variables and the assignment operator (‘‘=’’).

It is also possible to combine the two awk programs into a single program.

```
awk '/ed$/ {token = token + $1; type = type + 1}
     END   {print token, type}' genesis.hist
```

There is some syntactic sugar for adding a constant (+=) and incrementing (++) since these are such common operations. The following program is equivalent to the previous one.

```
awk '/ed$/ {token += $1; type++}
     END   {print token, type}' genesis.hist
```

Exercises:

1. It is said that English avoids sequences of *-ing* words. Find bigrams where both words end in *-ing*. Do these count as counter-examples to the *-ing -ing* rule?

2. For comparison's sake, find bigrams where both words end in *-ed*. Should there also be a prohibition against *-ed -ed*? Are there any examples of *-ed -ed* in Genesis? If so, how many? Which verse(s)?

8.5 Memory across lines

The use of the variables, ‘‘x,’’ ‘‘type,’’ and ‘‘token’’ in examples above illustrated the use of memory across lines.

Exercise: Print out verses containing the phrase ‘‘Let there be light.’’ Also print out the previous verse as well.

The solution uses a variable, prev, which is used to store the previous line. The AWK program reads a line at a time. If there is a match, we print out the previous line as well as the current one. In any case, we reassign prev to the current line so that when the awk program reads the next line, prev will continue to contain the previous line.

```
awk '/Let there be light/ {print prev; print}
     {prev = $0}' genesis
```

Exercise: write a `uniq -c` program in `awk.` Hint: the following ‘‘almost’’ works

```
awk '$0 == prev { c++ }
     $0 != prev { print c, prev
                  c=1
                  prev=$0 }'
```

The following example illustrates the problem: the AWK program drops the last line. There should be three output lines:

```
echo a a b b c c | tr '[ ]' '[\012*]' | uniq -c
   2 a
   2 b
   2 c
```

but our program generates only two:

```
echo a a b b c c | tr '[ ]' '[\012*]' |
awk '$0 == prev { c++ }
     $0 != prev { print c, prev
                  c=1; prev=$0 }'

2 a
2 b
```

The solution is given in the appendix.

8.6 uniq1

sort morphs by freq, and list 3 examples:

```
tr -sc '[A-Z][a-z]' '[\012*]' < genesis |
spell -v | sort | uniq1 |
awk '{print NF-1, $1, $2, $3, $4}' |
sort -nr

192  +s    Cherubims  Egyptians   Gentiles
129  +ed   Blessed    Feed        Galeed
 77  +d    Cursed     abated      acknowledged
 49  +ing  Binding    according   ascending
 32  +ly   Surely     abundantly  boldly
```

We have to write uniq1

uniq1 merges lines with the same first field

input:

```
+s     goods
+s     deeds
+ed    failed
+ed    attacked
+ing   playing
+ing   singing
```

output:

```
+s     goods     deeds
+ed    failed    attacked
+ing   playing   singing

awk ’$1 == prev {list = list " " $2}
     $1 != prev {if(list) print list
             list = $0
             prev = $1}
     END        {print list}’

awk ’$1 == prev {printf "\t%s", $2}
     $1 != prev {prev = $1
                 printf "\n%s\t%s", $1, $2}
     END        {printf "\n"} ’
```

New function: printf

Exercise: extract a table of words and parts of speech from w7.frag.

```
  ...
abacus      n
abaft       av   pp
abalone     n
abandon     vt   n
abandoned   aj
  ...
```

8.7 Arrays

Two programs for counting word frequencies:

```
tr -sc ’[A-Z][a-z]’ ’[\012*]’ < genesis |
  sort |
  uniq -c


tr -sc ’[A-Z][a-z]’ ’[\012*]’ < genesis |
awk ’
       { freq[$0]++ };
   END { for(w in freq)
           print freq[w], w }’
```

Arrays are really hashtables

- They grow as needed.
- They take strings (and numbers) as keys.

8.8 Mutual Info: An Example of Arrays

$$I(x;y) \quad = \quad \log_2 \frac{Pr(x,y)}{Pr(x) \quad Pr(y)}$$

$$I(x;y) \quad \approx \quad \log_2 \frac{N \quad f(x,y)}{f(x) \quad f(y)}$$

```
paste genesis.words genesis.nextwords |
sort | uniq -c > genesis.bigrams

cat genesis.hist genesis.bigrams |
awk ’NF == 2 { f[$2]=$1}
     NF == 3 {
print log(N*$1/(f[$2]*f[$3]))/log(2), $2, $3}
    ’ N="‘wc -l genesis.words‘"
```

8.9 Array Exercises

1. Mutual information is unstable for small bigram counts. Modify the previous prog so that it doesn’t produce any output when the bigram count is less than 5.

2. Compute *t*, using the approximation:

$$t \quad \approx \quad \frac{f(x,y) - \frac{1}{N} \quad f(x) \quad f(y)}{\sqrt{f(x,y)}}$$

   Find the 10 bigrams in Genesis with the largest *t*.

3. Print the words that appear in both Genesis and wsj.frag, followed by their freqs in the two samples. Do a `man` on `join` and do it again.

4. Repeat the previous exercise, but don’t distinguish uppercase words from lowercase words.

9. KWIC

Input:

```
All’s well that ends well.
Nature abhors a vacuum.
Every man has a price.
```

Output:

```
     Every man has a price.
     Nature abhors a vacuum.
            Nature abhors a vacuum
  All’s well that ends well.
        Every man has a price.
             Every man has a price
  Every man has a price.
       All’s well that ends well.
  Nature abhors a vacuum.
             All’s well that ends
   well that ends well.
```

10. KWIC Solution

```
awk '
{for(i=1; i<length($0); i++)
   if(substr($0, i, 1) == " ")
      printf("%15s%s\n",
        substr($0, i-15, i<=15 ? i-1 : 15),
        substr($0, i, 15))}'
```

- substr
- length
- printf
- for(i=1; i<n; i++) { ... }
- *pred* ? *true* : *false*

11. Concordance: An Example of the match function

Exercise: Make a concordance instead of a KWIC index. That is, show only those lines that match the input word.

```
awk '{i=0;
      while(m=match(substr($0, i+1), "well")){
         i+=m
         printf("%15s%s\n",
           substr($0, i-15, i<=15 ? i-1 : 15),
           substr($0, i, 15))}'

         All’s well that ends
well that ends well.
```

12. Passing args from the command-line

```
awk '{i=0;
      while(m=match(substr($0, i+1), re)) {
         i+=m
         printf("%15s%s\n",
           substr($0, i-15, i<=15 ? i-1 : 15),
           substr($0, i, 15))}}
    ' re=" [^aeiouAEIOU]"
```

```
          All's well that ends
     All's well that ends well
 well that ends well.
Nature abhors a vacuum.
          Every man has a pric
      Every man has a price.
Every man has a price.
```

- match takes regular expressions
- while (expression) { action }

Appendix: Solutions to selected exercises

Exercise: Sort words in Genesis by folding case.

```
tr -sc ’[A-Z][a-z]’ ’[\012*]’ < genesis |
sort |
uniq -c |
sort -f |
```

Note that ‘‘Beware’’ appears after ‘‘betwixt’’ and before ‘‘beyond.’’

Exercise: Sort words in Genesis by rhyming order.

```
tr -sc ’[A-Z][a-z]’ ’[\012*]’ < genesis |
sort |
uniq -c |
rev |
sort |
rev
```

Exercise: Count trigrams.

```
tr -sc ’[A-Z][a-z]’ ’[\012*]’ < genesis > genesis.words
tail +2 genesis.words > genesis.nextwords
tail +3 genesis.words > genesis.nextwords2

paste genesis.words genesis.nextwords genesis.nextwords2 |
sort | uniq -c > genesis.trigrams
```

Exercise: How many uppercase words are there in Genesis?

This is a bit of a trick question. What do we mean by a word? Do we count two instances of ‘‘the’’ as two words or just one? That is, are we counting by token or by type? Either solution is fine. The exercises are intentially left vague on certain points to illustrate that the hardest part of these kinds of caculations is often deciding what you want to compute.

```
# by token
tr -sc ’[A-Z][a-z]’ ’[\012*]’ < genesis |
grep -c ’ˆ[A-Z]’
5533

# by type
tr -sc ’[A-Z][a-z]’ ’[\012*]’ < genesis |
sort -u | grep -c ’ˆ[A-Z]’
635
```

Exercise: How many 4-letter words?

```
# by token
tr -sc ’[A-Z][a-z]’ ’[\012*]’ < genesis |
grep -c ’ˆ....$’
9036

# by type
tr -sc ’[A-Z][a-z]’ ’[\012*]’ < genesis |
sort -u | grep -c ’ˆ....$’
448
```

Exercise: Are there any words with no vowels?

```
tr -sc ’[A-Z][a-z]’ ’[\012*]’ < genesis |
grep -vi ’[aeiou]’ | sort | uniq -c
   1
   4 By
  18 My
  11 Thy
   3 Why
  80 by
   4 cry
   4 dry
   1 fly
 325 my
   2 myrrh
 251 s
 268 thy
   5 why
```

The output above illustrates that there are certain technical difficulties with our definition of ‘‘word’’ and with our definition of ‘‘vowel.’’ Often, though, it is easier to work with imperfect definitions and interpret the results with a grain of salt than to spend endless amounts of time refining the program to deal with the tricky cases that hardly ever come up.

Exercise: Find ‘‘1-syllable’’ words

```
tr -sc ’[A-Z][a-z]’ ’[ 12*]’ < genesis |
grep -i ’ˆ[ˆaeiou]*[aeiou][ˆaeiou]*$’ |
sort | uniq -c | sed 5q
   2 A
   5 All
   2 Am
1251 And
   1 Ard
```

Exercise: Find ‘‘2-syllable’’ words

```
tr -sc ’[A-Z][a-z]’ ’[\012*]’ < genesis |
grep -i ’ˆ[ˆaeiou]*[aeiou][ˆaeiou]*[aeiou][ˆaeiou]*$’ |
sort | uniq -c | sed 5q
   8 Abel
  59 Abram
   1 Accad
   2 Achbor
   8 Adah
```

Exercise: How many verses have two or more instances of ‘‘light’’?

```
grep -c ’light.*light’ genesis
4
```

Three or more?

```
grep -c ’light.*light.*light’ genesis
1
```

Exactly two?

```
grep ’light.*light’ genesis | grep -vc ’light.*light.*light’
3
```

Exercise: Count morphs in genesis

```
spell -v genesis > /tmp/foo
sed ’s/     .*//g’ < /tmp/foo | sort | uniq -c
```

Exercise: Sort the words in Genesis by the number of syllables (sequences of vowels).

```
tr -sc ’[A-Z][a-z]’ ’[\012*]’ < genesis | sort -u > genesis.words
tr -sc ’[AEIOUaeiou\012]’ ’ ’ < genesis.words | awk ’{print NF}’ > genesis.syl
paste genesis.syl genesis.words | sort -nr | sed 5q
6     Jegarsahadutha
5     uncircumcised
5     peradventure
5     interpretations
5     interpretation
```

Exercise: Find vowel sequences that appear at least 1000 times

```
tr -sc ’[A-Z][a-z]’ ’[\012*]’ < genesis |
tr -sc ’AEIOUaeiou’ ’[\012*]’ |
sort | uniq -c |
awk ’$1 >= 1000’
1625 A
11164 a
15813 e
1000 ea
6015 i
7765 o
1251 ou
1849 u
```

Exercise: Find bigrams that appear exactly twice

```
awk ’$1 == 2 {print $2, $3}’ genesis.bigrams
```

Exercise: Find words in Genesis that can also be spelled both forwards and backwards.

```
rev < genesis.types > genesis.types.rev
sort genesis.types genesis.types.rev |
      uniq -c |
      awk ’$1 >= 2 {print $2}’

A
I
O
a
deed
did
draw
evil
ewe
live
no
noon
on
s
saw
ward
was
```

Exercise: Compare Exodus and Genesis. Find words that are in just the first file, just the second, and both.

Solution: sort the words in Genesis with two copies of the words in Exodus and count. Words with counts of 1 appear just in Genesis, words with counts of 2 appear just in Exodus, and words with counts of 3 appear in both.

```
tr -sc ’[A-Z][a-z]’ ’[\012*]’ < genesis | sort -u > genesis.types
tr -sc ’[A-Z][a-z]’ ’[\012*]’ < exodus | sort -u > exodus.types
sort genesis.types exodus.types exodus.types | uniq -c | head
   3
   3 A
   2 AM
   2 Aaron
   1 Abel
   1 Abelmizraim
   2 Abiasaph
   2 Abib
   1 Abidah
   1 Abide
```

Exercise: Fix the AWK implementation of uniq –c.

```
awk ’$0 == prev { c++ }
     $0 != prev { print c, prev
                  c=1
                  prev=$0 }
     END {print c, prev}’
```