USING THE LEVENSHTEIN ALGORITHM FOR AUTOMATIC LEMMATIZATION IN OLD ENGLISH

by

BERNADETTE E. JOHNSON

(Under the Direction of William Kretzschmar)

This study was undertaken to develop and test an automatic lemmatization program for the Old English language utilizing the Levenshtein edit distance algorithm, stemming, and other techniques to help overcome issues such as rampant spelling irregularity and the presence of inflectional endings. The primary goal was to create a lemma list for import into text analysis software to equate Old English words with their variants, which was met with limited but promising success. The main lemmatization program is written in the Perl programming language. Other scripts in Perl and XSLT, as well as Unix command line commands and AntConc 3.2.0 corpus analysis software, were used to extract text files from the *Dictionary of Old English Corpus* XML files, to generate sorted lists of all the words in the available texts for use by the main program, and to manipulate and analyze the data.

INDEX WORDS: Levenshtein Algorithm, Levenshtein, Lemmatization, Lemma, Lemma list, Stemmer, Stemming, Old English, Text analysis, Corpus analysis, Dictionary of Old English, Dictionary of Old English Corpus, Perl, Unix, XSLT, XML, AntConc, oXygen XML Editor.

USING THE LEVENSHTEIN ALGORITHM FOR AUTOMATIC LEMMATIZATION IN OLD ENGLISH

by

BERNADETTE E. JOHNSON

B.A., The University of Georgia, 1995

A Thesis Submitted to the Graduate Faculty of The University of Georgia in Partial Fulfillment

of the Requirements for the Degree

MASTER OF ARTS

ATHENS, GA

2009

© 2009

Bernadette E. Johnson

All Rights Reserved

USING THE LEVENSHTEIN ALGORITHM FOR AUTOMATIC LEMMATIZATION IN OLD ENGLISH

by

BERNADETTE E. JOHNSON

Major Professor: William Kretzschmar

Committee: Jonathan Evans
Amanda Gailey

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
May 2009

ACKNOWLEDGEMENTS

I would like to thank my major professor, Dr. William Kretzschmar, for his friendly and valuable guidance throughout the thesis writing process and for teaching me everything I know about text analysis, Dr. Jonathan Evans for his Old English tutelage over the last few years, and Dr. Amanda Gailey for providing advice and allowing me to sit in on her Humanities Computing class, and all three for taking time out of their busy schedules to be on my committee. All of their classes have been most enjoyable, and their help with this endeavor has been invaluable.

The other people who contributed to my overall education and academic success are too numerous to list, but I would like to thank all of my wonderful teachers, my friends and my family for giving me support and guidance over the years, and Jeff for a tremendous amount of moral support throughout graduate school.

TABLE OF CONTENTS

Page

ACKNOWLEDGEMENTS..iv

LIST OF FIGURES.. vii

CHAPTER

1 INTRODUCTION..1

1. Purpose of the Study.. 1

2. How This Study is Original..4

2 USING THE LEVENSHTEIN ALGORITHM FOR AUTOMATIC LEMMATIZATION IN OLD ENGLISH...7

1. Overview of the Levenshtein Algorithm...7

2. Existing Stemming and Lemmatization Tools.................................... 11

3. Necessity for a New Lemmatization Method for Old English.......................... 15

4. Methods and Programs Used in This Study.. 16

5. Results.. 38

6. Issues and Possibilities for Future Improvement...................................50

3 CONCLUSION.. 57

WORKS CITED.. 59

APPENDICES

A Custom Scripts..61

B Code from Other Sources... 69

C Porter Stemmer.. 78

D Unicode Character Set.. 81

LIST OF FIGURES

Page

Figure 1.1: Levenshtein difference matrix between *dryhten* and *drihten*..................................... 10

Figure 1.2: Levenshtein difference matrix between *dryhtin* and *drihten*......................................10

Figure 1.3: Levenshtein difference matrix between *drihtenes* and *dryhten*..................................10

Figure 2.1: Screenshot of lemma list after importing into AntConc...46

Figure 2.2: Second page of wordlist before AntConc lemma list importation............................. 47

Figure 2.3: Second page of wordlist after AntConc lemma list importation................................ 48

Figure 2.4: First page of keyword list before AntConc lemma list importation...........................49

Figure 2.5: First page of keyword list after AntConc lemma list importation..............................50

CHAPTER 1

INTRODUCTION

1. Purpose of the Study

The purpose of this study was to make headway in performing automatic lemmatization of the Old English language using the Levenshtein edit distance algorithm to deal with issues of spelling irregularity in the language. I started the project specifically for creating lemma files that equate variant versions of the same words with each other for import into text analysis software in the hopes of generating more accurate analysis that did not require searching for and manually combining all the statistics for variants of the same words or undertaking the laborious task of taking a list of all the words in a corpus and manually creating such a lemma file.

For Modern English, it would be difficult enough to find all the related words by going over the results manually, but it is even more difficult with Old English, which has many more inflectional endings than Modern English, and a large amount of spelling irregularity from document to document (and often enough even within the same text). Simply looking down an alphabetized list of words to find related words becomes far more difficult when many letters are used interchangeably with each other and a variety of different spellings for the same word are possible. And even alphabetization poses difficulties where special characters that are not part of the Modern English alphabet are concerned.

A "lemma" is essentially the most basic form of a word, the type you would find as the headword in a dictionary as representative of all other forms of the word. The *Oxford English Dictionary* definition as regards Lexicography is "A lexical item as it is presented, usu. in a

standardized form, in a dictionary entry; a definiendum" (*OED Online* "lemma[1]"). Note that when I used the terms "lemmas," "lemma list," and "lemma file," I am referring to variants of the same word grouped together for use in text analysis, rather than to headwords. The definition of "lemmatize" illustrates the aim of this study perfectly: "To sort (words as they occur in a text) so as to group together those that are inflected or variant forms of the same word" (*OED Online*, "lemmatize").

The main program takes in a list of words generated from the *Dictionary of Old English Corpus* and outputs a "lemma file" containing related words (variants with different declensional endings and spellings) grouped together into small comma delimited lists (or as close as the current methods will get us to this goal). With some modification, the file can be imported into commonly used text analysis software such as AntConc or Wordsmith Tools, from which wordlists and keyword lists can be generated that will consider all of the lemmatized variants as part of the same word so that frequency and keyword statistics will be calculated based on all the words in each group. Without a lemma file, all the variant words and their statistics are calculated and output separately on a word-by-word basis. But with a lemma file, for instance, instead of separate entries and calculations for the word *dryhten* (meaning "ruler, king, lord, prince" (Hall 89)), and some of its declined forms such as *dryhtenes* (the genitive singular form) and *dryhtenum* (the dative plural form), these words would be grouped together with statistics reflecting all occurrences of all three words. If *dryhten* occurs three times, *drythenes* four times, and *dryhtenum* two times, the frequency will be shown as nine under one headword rather than as three separate frequencies of separate words.

Unlike Modern English (at least since the advent of the printing press and the overall standardization of spelling that eventually occurred as a result of this invention), Old English

texts exhibit a large number of spelling variations of the same words, with certain characters or groups of characters being used interchangeably, sometimes in a somewhat unpredictable manner. For instance, for "a" you might find "æ" (the Old English character "ash"), "aa," or "aæ," not to mention a number of accented versions of the letter "a." The Old English characters eth ("ð") and thorn ("þ") were very often used interchangeably, as were "y" and "i," and "c" and "k" (though "k" is somewhat rare compared to "c"). Many letters were doubled and vowels in general were quite unpredictably substituted for one another. The word *dryhten*, as an example once again, appears in *A Concise Anglo-Saxon Dictionary* under that spelling and no other (Hall 89). Its headword in the Bosworth-Toller dictionary is *drihten*, with *dryhten* listed at the end of the entry as a variant form (Bosworth 213). Though *drihten* and *dryhten* are the common dictionary headwords, it appears in the *Dictionary of Old English Corpus* in at least fourteen separate variant forms: *dræhton*, *drehtan*, *drehten*, *drehton*, *drihtan*, *drihtæn*, *dryhtæn*, *drihten*, *dryhten*, *drihtin*, *drihtyn*, *dryhtin*, *dryhtyn*, and *drihton*. I extracted this data by using the Unix command `grep "^dr.ht.n$" wordlist_sorted.txt`, which calls the grep searching program to find all occurrences matching the regular expression within quotation marks (where "^" means beginning of line, "." means any single character, and "$" means end of line) in the file wordlist_sorted.txt (a file containing every unique word in the corpus, each on a separate line). This count does not include any of the declined versions or any that might contain doubled or additional letters. This makes matching the variant forms with any built-in comparison functions, which are generally looking for exact matches, difficult to impossible.

Current Modern English lemmatizers would be ineffective for this reason and also because of the presence of inflectional endings that for the most part differ a great deal from any we find in English today. The most commonly used lemmatizers involve stemming modern

words (i.e. removing the endings from words and using their root or stem forms as the main headword) based on a strict set of rules, but this requires very predictable spelling, which is present in Modern English and most other modern languages, but is not present in Old English. This makes the Levenshtein algorithm, which measures the edit distance, or in other words the number of changes necessary to change one word into another, a good candidate for aiding in lemmatization of Old English. Rather than relying on exact spellings of words, a minimum acceptable difference between the words (or their stemmed forms) can be set in order to group like words together that are not necessarily spelled exactly alike.

My repetition of the word *dryhten* as the most frequently used example throughout the paper is the result of *dryhten* and its variants coming up as the most frequent content word in wordlist and keyword searches of the entire corpus and various smaller sections of the corpus (including the prose and poetry subsets). It was the results of the keyword lists, with every variant of every word on a separate line with a separate keyness and frequency, that made me believe that a lemma list was necessary for effective corpus analysis in Old English, and the incredibly daunting task (likely impossible within a reasonable time frame) of assembling one manually which made me think to attempt automatic lemmatization. Issues arising due to unpredictable spelling variations during my early lemmatization attempts led me to search for solutions that would match on similarity rather than exact spelling, which ultimately led me to the Levenshtein algorithm.

2. How This Study Is Original

I was unable to find any instances of the Levenshtein algorithm being used in the lemmatization of Old English, and was in fact unable to find any Old English specific lemmatizers, or even stemmers, of any sort. I found reference to Levenshtein being used to

compare Modern English proper names in order to find equivalents that were not spelled exactly the same ("Smith" versus "Smythe," for instance), and decided that it might also be of use in dealing with Old English spelling variation. I have created original programs that allow me take word lists generated by existing corpus analysis software from a large Old English corpus, sort the lists based on a sort order more appropriate to Old English than is normally allowed in existing sorting functions, and generate an at least partially accurate lemma list (which includes words that are similar enough to possibly be considered variations of the same word, grouped together in individual comma delimited lists).

Upon searching, I was also unable to find any existing Old English lemma lists, and if there are any in existence, it is doubtful that they contain very many spelling variants of each word. Using Levenshtein, it is theoretically possible to generate one with every spelling variant in existence in a particular corpus, although in practice there are some issues that need to be overcome and this study only scratches the surface.

This project includes one XSLT program used to extract text only versions of the Old English texts from the *Dictionary of Old English Corpus* XML files, two Perl programs used to do sorting and lemmatization respectively, and analysis of the resulting lemma list and its effect on analysis of the corpus. I used AntConc text analysis software for generation of the original word lists and importation and testing of the final lemma lists.

The set of texts from which all of my word lists were generated is the *Dictionary of Old English Corpus*, a corpus compiled at the University of Toronto for the Dictionary of Old English project covering English vocabulary from around 600 AD to 1150 AD. The dictionary is "based on a computerized Corpus comprising at least one copy of each text surviving in Old English" (Healey "About"). The corpus was obtained in CD form and contains copies of all of

the texts in HTML, SGML, and XML formats. The XML versions of the texts were used for this study by extracting one plain text file from each XML encoded text.

CHAPTER 2

USING THE LEVENSHTEIN ALGORITHM FOR AUTOMATIC LEMMATIZATION IN OLD ENGLISH

1. Overview of the Levenshtein Algorithm

The Levenshtein, or Edit-Distance, algorithm calculates the minimum number of operations that would be necessary to convert one string into another string, including character deletions, insertions, and substitutions (*www.levenshtein.net*; Gilleland "ld.htm"). Put simply, the algorithm moves forward through each letter of the two strings and adds 1 to the edit distance if the letters do not match and a change is necessary at any given step. If the words being compared are *cyning* and *cining*, for instance, the Levenshtein distance would be 1, because only one change ("y" to "i") is necessary to transform *cyning* into *cining.* It is one type of "fuzzy" or approximate string matching, as opposed to exact string matching (Orwant "Section 9.2"), the latter of which is what most existing string comparison functions perform. Levenshtein has been used in such applications as spell checking, speech recognition, and plagiarism detection (Gilleland "ld.htm").

The Perl implementation of the Levenshtein algorithm used in this project is from the www.wikibooks.org page "Algorithm implementation/Strings/Levenshtein distance." The Perl code at this site is a modified version of the Perl code created by Eli Bendersky at http://www.merriampark.com/ldperl.htm. The code itself can be found in Appendix B (section B2) or in Appendix A as the "levenshtein" subroutine in the main program (oe_lemma.pl, section

A3). The following paragraph is a walk through of the algorithm as implemented in the program code (note that I omitted "use strict" from the oe_lemma.pl program as it was causing problems).

The algorithm takes two arguments for comparison (any two strings you pass into the subroutine) and sets the variables $a and $b to the values of the two strings. It then sets $len1 and $len2 to the lengths of string $a and string $b respectively. If $len1 equals zero, the subroutine returns $len2 and terminates, and if $len2 equals zero, the subroutine returns $len1 and terminates (in other words, if one string is blank, the length of the other string is returned, since that is the number of changes that would be necessary to make the blank string into the non-blank string). An associative array %d is declared (where each value in %d is accessible using the variable $d{n}, “n” being the number of the location of any element in the array), and a set of nested for loops starts, the outermost looping through the variable $i starting at the value zero and going until the value of $len1 is reached, and the innermost looping through the variable $j starting at zero and going until the value of $len2 is reached. A for loop is a control structure that allows a program to iterate through a group of something, in this case the members of an associative array, generally starting with a control variable set at a fixed number and ending when that variable reaches another number, with the variable incrementing or decrementing on each iteration through the loop so that the ending condition will be reached at some point. In this case, during each iteration through either loop, the value of the $i or $j variable is incremented by one so that the loop is exited when the length of each word is reached. This set of nested loops initializes the minimum distance values of each possible pair of letters (all are set to zero except for $d{0}{$j} and $d{$i}{0} which are set to the values of $j and $i respectively). This, in effect, creates a matrix of possible edit distances, with a top “row” numbered from zero to the length of string $b and a left side “column” numbered from zero to

the length of string $a (each location containing the Levenshtein distance at that spot, and the values at the diagonal of this matrix containing the minimum edit distance up to that point in the words). In the case of, say, comparing *drihten* to *dryhtan*, both the top row and side column would consist of 0, 1, 2, 3, 4, 5, 6, and 7 since both strings are of the same length. "Row" and "column" are really just concepts here, since no literal rows or columns are returned by this particular code, but they help conceptualize the process (see Figures 1.1, 1.2, and 1.3 at end of this section for illustration). After this initialization, two arrays are created and populated with all the characters from each string, @ar1 with the characters from string $a and @ar2 with the characters from string $b. Another set of nested for loops (this time $j from 1 to $len2 within $i from 1 to $len1) go through and compare all the characters to each other and determine the Levenshtein cost between the letters being compared (i.e. the number 1 if a change is necessary and the number 0 if no change is necessary). Since the loop iterating through every value of $i is on the outside, this section will start with $i =1 and then iterate through every value of $j from 1 to the length of $b, making calculations for each comparison until the next value of $i, where it will start over with $j =1, and so on until all values of $i (the total length of string $a) are exhausted (in effect creating values for each "row" and "column"), and the final resulting Levenshtein value is returned. The following line, the first executed statement within the nested "for" loops, is somewhat obfuscated and probably needs explanation:

```
my $cost = ( $ar1[ $i - 1 ] eq $ar2[ $j - 1 ] ) ? 0 : 1;
```

The () ? : structure is actually a sort of shortcut for if/then/else, so the above means if $ar1[$i-1] equals $ar2[$j-1], then $cost equals zero, else $cost equals 1 (i.e. if the letters are equal, there is no Levenshtein cost, but otherwise there is). $min1 is then set to $d{$i-1}{$j} + 1 (i.e. the value at the location in the matrix that is immediately above the current location plus one), $min2 is set

to $d{$i}{$j-1} + 1 (i.e. the value at the location immediately to the left of the current location plus one), and $min3 is set to $d{$i-1}{$j-1} plus $cost (i.e. the value at the location diagonally above and to the left of the current location plus the cost calculated at the beginning of the loop). The value at $d{$i}{$j} (which would be the row/column location where the two letters being compared intersect) is then set to the minimum (or lowest value) of the three variables ($min1, $min2, or $min3). The increasing cost values from zero to the eventual final Levenshtein value are therefore held in the diagonal between the letters being compared all the way to $d{$len1}{$len2} (the bottom right-hand value in the matrix, which is returned after the $i for loop is exited). This final lower right-hand value is the Levenshtein distance. In the case that one string is longer than the other, it will have extra rows or columns (depending upon which string is longer) in the number of the difference between the length of the two strings, and for every extra row or column, another cost of 1 will be added to the edit distance.

		d	r	y	h	t	e	n
	0	1	2	3	4	5	6	7
d	1	0	1	2	3	4	5	6
r	2	1	0	1	2	3	4	5
i	3	2	1	1	2	3	4	5
h	4	3	2	2	1	2	3	4
t	5	4	3	3	2	1	2	3
e	6	5	4	4	3	2	1	2
n	7	6	5	5	4	3	2	1

		d	r	y	h	t	i	n
	0	1	2	3	4	5	6	7
d	1	0	1	2	3	4	5	6
r	2	1	0	1	2	3	4	5
i	3	2	1	1	2	3	4	5
h	4	3	2	2	1	2	3	4
t	5	4	3	3	2	1	2	3
e	6	5	4	4	3	2	2	3
n	7	6	5	5	4	3	3	2

		d	r	i	h	t	e	n	e	s
	0	1	2	3	4	5	6	7	8	9
d	1	0	1	2	3	4	5	6	7	8
r	2	1	0	1	2	3	4	5	6	7
y	3	2	1	1	2	3	4	5	6	7
h	4	3	2	2	1	2	3	4	5	6
t	5	4	3	3	2	1	2	3	4	5
e	6	5	4	4	3	2	1	2	3	4
n	7	6	5	5	4	1	2	1	2	3

Figures 1.1, 1.2, and 1.3 (from left to right). These show examples of Levenshtein difference matrices for comparisons between the pairs of words *dryhten* and *drihten* (with Levenshtein distance of 1), *dryhtin* and *drihten* (with Levenshtein distance of 2), and *drihtenes* and *dryhten* (with Levenshtein distance of 3) respectively. The final Levenshtein distance value is the number in the lower right-hand corner of each matrix (1, 2, and 3 in these cases). Note that in Figure 1.3, the diagonal starting at 0 in the upper left hand corner ends with 1 three spaces from the lower right, then continues to the right with 2 and 3, for a final Levenshtein distance of 3 due to a difference in word length.

2. Existing Stemming and Lemmatization Tools

Stemmers, commonly used lemmatization tools, use rules to remove inflectional endings and reduce words down to their stem (or root) form. They often contain hard-coded lists of possible endings, along with the rules for what to do with the suffixes when they are encountered (either removal or transformation into another string ending). One distinction in meaning between stemming and lemmatization is that lemmatizers aim to reduce words down to the simplest uninflected form resulting in a word upon which all the inflected forms are based, whereas stemmers will remove both inflectional endings and derivational suffixes and the resulting stem might not even be a complete word (Mason 179), although often enough these two terms are used interchangeably. Stemmers are often used for Information Retrieval as well (Tzoukerman 531), where indexing based on a root form that may or may not be a real word can be quite useful for text search and retrieval functions (in applications such as search engines). But not all stemmers reduce words to partial non-word forms. The Porter Stemmer, for instance, makes an attempt after suffix removal in some cases to add other suffixes or final letters back to make a complete word (Porter "def.txt"). Other well-known stemmers, aside from the Porter Stemmer mentioned above, are the Lovins Stemmer and the Lancaster Stemmer, though there are a number of others in existence for use with various languages. Most of the existing lemmatizers that are not primarily stemmers appear at least to use stemmers in some way.

The Porter Stemmer was developed by Martin Porter at the University of Cambridge beginning with a paper on his stemming algorithm written in 1979 entitled "An Algorithm for Suffix Stripping," and an early programming incarnation written in BCPL, a now defunct language. It has since been re-written and distributed in a variety of programming languages by Martin Porter and others. All of the programmed versions (including the original BCPL) contain

slight improvements over the originally published algorithm. Implementations of his stemmer are available at the main Porter Stemmer web site in a variety of languages, including Perl, Java, Ruby, php, and C (Porter "Porter Stemmer").

The algorithm from Porter's original paper (available at http://tartarus.org/~martin/PorterStemmer/def.txt) lists a set of rules a word can be taken through in order to remove or translate expected suffixes to reduce it down to its most basic uninflected form. Some of the rules test for various conditions, including one or more of the following: patterns of vowels and consonants repeated a certain number of times (generally there is a minimum applied in the condition that if not met, no transformation of the word will take place), the stem ending with an “s,” the stem containing a vowel, the stem ending with a double consonant, and the stem ending consonant/vowel/consonant where the second consonant is not “w,” “x,” or “y.” Following the possible conditions, it tests for the presence of certain suffixes, and if more than one is found, only the longest matched suffix is removed or converted. The matched suffix maps to a replacement value for that suffix (which is sometimes blank, but sometimes a shorter suffix). For instance, these examples straight from the paper show one value (the suffix the program is looking for on the left) mapping to a new value (to the right of "->"):

```
SSES -> SS
IES  -> I
SS   -> SS
S    ->            (Porter "def.txt")
```

One of his examples is the word "caresses," which could match either the first or the fourth condition above. Note that the fourth suffix (an “s” at the end of a word) would result simply in the removal of the final letter. Because "-sses" is the longest match, "caresses" is converted to "caress." As well as including a large number of possible suffixes, the algorithm takes each

word through a number of steps in an attempt to get a more correct root word. For instance, if "-ed" is removed from a word and the stem now ends with "-at," "-bl," or "-iz," a subsequent step that checks for these endings will add an "-e" back on to the end of the stem, so that "conflated" becomes "conflat" and then the more correct "conflate." There are five steps described in all, the first (of which the above are examples) covering plurals and past participles, and the subsequent four including much simpler rules for removal of specific suffixes. A word with multiple suffixes will go through removal processes in multiple steps. For the word "generalizations," for example, step one will remove the "-s" making it "generalization," step two will map the "-ization" to "-ize" making it "generalize," step three will map the "-alize" to "al" reducing it to "general," and step four will map the "al" to null, reducing the word further to "gener" (Porter, "def.txt"). The last step makes it an incorrect lemma for linguistic purposes, but one likely quite useful for Information Retrieval purposes, for which the algorithm seems to have been designed originally (even though in many cases suffixes are added that result in real words). But the steps and concepts contained in the algorithm, and in the subsequent programming language versions, are quite useful for linguistic lemmatization as well. For a Perl implementation of the Porter Stemmer written by Martin Porter, see Appendix C.

An even earlier stemming algorithm is the Lovins Stemmer, which is described in Julie Beth Lovins's paper "Development of a Stemming Algorithm" published in 1968 (Snowball "The Lovins Stemming Algorithm"). It is the first published stemming algorithm, and includes an extensive list of endings, conditions, and transformation rules, and stems the longest of the possible endings found. Unlike the Porter Stemmer, it was designed to stem a word in a single pass rather than multiple steps (Lancaster "The Lovins Stemmer"), though the general concept is very similar.

Another well-known stemmer is the Lancaster (or Paice/Husk) Stemmer, developed in the late 1980s at Lancaster University. This stemmer starts with the last letter of the word and looks in a file of various conditions grouped by letter for suffixes ending in that letter and their associated rules. Each possible rule is tried until one is applied or none are found, then the next remaining ending letter is used to find the next group of rules that might apply to the remaining stem. Before application of a rule, conditions are checked to make sure the suffix removal would leave an acceptable stem (for example, one that is not too short or would still contain a vowel). Rather than just once or a certain number of times, this algorithm will iterate over a word as many times as there are still applicable rules (*Lancaster* "stemming/general/paice.htm").

Aside from their IR functions, stemmers are also incorporated into linguistics specific programs, such as MorphAdorner (http://morphadorner.northwestern.edu/morphadorner/), which is a "morphological adorning" or tagging program that takes texts and marks them up with morphological information, including "standard spelling, parts of speech, and lemmata," among other things (MorphAdorner "Home"). This application includes the Porter, Lancaster, and Lovins stemmers among its other lemmatization tools (MorphAdorner "Lemmatization").

One can see how reducing a word down to its base stem could be of great use in finding words that are inflected forms of each other and grouping them together, but the stemmers and MorphAdorner, like many such existing linguistic software tools, are written specifically for Modern English and are not very applicable to Old English. The MorphAdorner Web site's "Lemmatization" page states that its lemmatization tools work best with standardized modern spellings, and it offers tools to help take earlier Modern English spelling variants and convert them to modern standard spellings before the lemmatization process. The original forms of the three above-mentioned stemmers were all written specifically for Modern English spellings and

suffixes. Though there are currently many variations available online for other mostly modern languages which include rules specific to their intended languages, they are dependent upon the words exhibiting standard modern spellings for accurate results.

3. Necessity for a New Lemmatization Method for Old English

The existing tools mentioned previously are all quite useful in linguistic research including their uses in lemmatization, but none of them are satisfactory for stemming or lemmatizing in Old English, mainly because of the unfixed spelling that existing tools cannot handle (even if they were reworked with rules and endings specific to Old English). Because of this lack, I began work on an Old-English-specific lemmatizer that uses a list of language specific possible endings to mimic some of the work of the stemmers, but also includes a few character translations of commonly interchangeable letters, and most importantly Levenshtein comparisons of the resulting stems in order to find differently spelled but related words and group them together. The results, while not perfect, are very promising. That said, because the algorithm is used in this case to find the edit distance between similar but different spellings, this lemmatizer will not do anything to find and match up highly irregular words. Known irregular words, including the verb *beon* ("to be"), and words like the demonstrative pronouns (*seo* and its declensions, equivalent to "the"), could be included in a lemma list manually, although it might be better to hard code them into the program for comparison to words in the word list in order to find unexpected irregular spellings). They are also likely, in any sort of corpus analysis, to be included in a stop list (or list of words to be excluded from analysis) since these particular examples are function words rather than content words, so in many cases their inclusion may not be necessary. Because the program is not equipped to deal with them, the ones spelled similarly may be grouped together, but their more irregular variants will be elsewhere.

In this project, a Perl implementation of the Levenshtein algorithm, along with an Old English specific sorting program, translations of certain characters, a primitive form of stemming, and other programmatic methods, have been used to compare words in a word list generated from all the texts in the *Dictionary of Old English Corpus*, and to place words with the appropriate level of similarity with each other into groups as related lemmas. This required the use of pre-existing text analysis software (namely AntConc 3.2.0m for Macintosh OSX), along with the writing of custom scripts in XSLT and Perl.

4. Methods and Programs Used in This Study

Three custom programs were created for this project, including one XSLT program to extract text files from the *DOOE* XML files, one Perl file to sort word lists generated by AntConc, and another Perl program to generate the lemma file and two additional files (one of possible compounds and one of unmatched words). The sorting and lemmatization programs go hand in hand, as the current incarnation of the lemmatizer depends upon the words being in a particular sort order. It goes through the words sequentially, comparing the previous word to the current word until it runs out of words in the sorted word list, and then outputs the three files.

The first step was creating a small XSLT program, called multiple_files.xsl, which extracted plain text files from the 3,047 XML files in the Dictionary of Old English Corpus (see Appendix A, section A1 for the code). It was run using the oXygen XML Editor (version 9.3), software created by SyncRO Soft LTD for creating and executing XML, XSL, and other related documents and scripts. The XSLT job was run against the file corpus.xml in the *Dictionary of Old English Corpus*, a file which contains a reference to each separate XML encoded Old English work in the corpus. In the program itself, an `<xsl:output>` command sets the output method to "text" and encoding type to "UTF-8." It has been vital every step of the way to make

sure, wherever possible, that the texts are both output from programs and input into software applications as UTF-8, one possible flavor of Unicode encoding. UTF-8 (as well as other types of Unicode) contains the ASCII subset as the first 128 possible characters (from codes 0 to 127, including a number of control characters, numbers zero through nine, the Modern English alphabet, a number of punctuation marks, and other common keyboard characters), but then also includes a number of additional characters in codes 128 to 255. Default encoding types tend to be ASCII, as most files we work with do not contain characters outside of the ASCII encoding range, but the Old English texts in this corpus (along with most Old English digital texts in existence) contain a number of characters that are beyond the ASCII range but included in Unicode. For example, lowercase ash ("æ"), thorn ("þ"), and eth ("ð"), as well as their uppercase counterparts, are very common Old English characters that are not contained within the ASCII subset, but are in Unicode. UTF-8 is one of the standardized versions of Unicode (see Appendix D for the UTF-8 character set), and is fairly well integrated into Perl, so this was my encoding choice.

After setting the output parameters, the program multiple_files.xsl then executes a template that matches on the TEI.2 element in the XML document it is acting upon, which then executes an `<xsl:for-each>` statement that iterates through every matched element, sets the "filename" variable to concatenate "teiHeader/fileDesc/publicationStmt/idno," an underscore, and another statement that drills down to teiHeader/fileDesc/titleStmt/title[@type='st'] (meaning the attribute with type equals "st" within the element <title> within <titleStmt> within <fileDesc> within <teiHeader>) and does a translation of all characters "*", "/", ":", and " " to underscore. This translation was necessary for removing special characters that were wreaking havoc with Unix commands and other software. The colon character was causing issues with the

XSLT program that were keeping it from working, asterix is generally a regular expression wildcard character, the forward slash ("/") is part of a file's pathname, and space (" ") causes issues when referencing filenames in Unix.

After this, an `<xsl:result-document>` section creates a file with path text_files3/{$filename}.txt ($filename being the value of the filename variable set in the previous statements), and within the result-document statement another for-each loop iterates through every <s> (sentence) tag in the XML documents (using the statement `<xsl:for-each select="text/body/p/s">` to drill down from the `<text>` element to `<body>` to `<p>` (paragraph) to `<s>` and retrieve each sentence, outputting each sentence followed by a carriage return and line feed (the "
" and "&@x0A;" entities respectively) until the end of the file, then exiting the result-document and starting the outer for-each loop on the next file, until the folder text_files3 contains a text file for every XML Old English file in the corpus. In effect, by running the program against "corpus.xml" (which contains an entity reference to each XML file in the corpus, such as "&T01370;" which points to the *Beowulf* file "T01370.xml"), it generated a file for each separate XML OE file in the corpus, each consisting of sentences separated by carriage return and line feed.

After these files were created, the next step was to run the entire corpus through text analysis software in order to generate a list of all the unique words in the corpus. For this I used AntConc version 3.2.0m, corpus linguistics software developed by Lawrence Anthony that allows for generation of word lists, keyword lists, concordances, collocations, and word clusters, among other data useful for text analysis. AntConc allows you to select either a group of files individually or an entire directory, which was fortunate, because the individual file selection would not work due to special Old English characters in the file names. The program saw them

as multiple ASCII characters instead of single Unicode characters just in the file selection area, but a setting within the software allowed proper processing of the special characters within the files. The language encoding in AntConc had to be changed from the default "Western Europe 'Latin1' (iso-8859-1)" to "Unicode (utf8)" before proceeding. If the wrong language encoding is selected, the special characters will not be output properly.

The software allows generation of word lists either by word (in alphabetical order) or by frequency (number of occurrences in descending order). As is the case with many software applications' built-in sort functions (which I will go into in more detail in the section on the oe_sort.pl program below), the resultant word list is not sorted in an order optimal for viewing and grouping Old English words. Words starting with ash, thorn, and eth end up at the end of the list after words starting with Roman alphabetic characters "a" through "z." Words beginning with other special characters such as accented vowels, "c" with cedilla ("ç") and l-bar ("ł") end up at the end, as well.

AntConc outputs a file with three fields per line: "rank," "frequency," and "word." The rank is simply the numeric order in which the words were output (1 for the first word, 2 for the second, etc.). Frequency is the number of times the word occurs in the corpus (therefore, each word is only output once), and word is the word itself. I selected "Treat all data as lowercase," which resulted in an all-lowercase output file, to eliminate issues with case in subsequent processing. I did a sort by frequency and another by word, and saved the resulting output files as "wl_all_byfreq.txt" and "wl_all_byword.txt" respectively. Both files contain 260,557 lines, each line with one separate unique word from the corpus (mostly Old English words, but also including some Latin words and at least one Greek word).

I created two Perl programs, oe_sort.pl and oe_lemma.pl. The first one, oe_sort.pl, sorts the word list generated by AntConc into a word order more appropriate to Old English than one would get by using the built-in Perl (or Unix) sort functions. The second program, oe_lemma.pl, iterates through the sorted word list created by the former program and groups the words into sets of possible lemmas, or words that are essentially the same word but with different spellings or declensional and/or conjugational endings.

For the oe_sort.pl program, I had little success with existing sort functions (on the Unix command line or in Perl). I researched and located a CPAN (Comprehensive Perl Archive Network) module called Sort::ArbBiLex, created by Sean Burke, which was created specifically to allow users to sort using more arbitrary user-defined sort orders than the standard ASCII sort order of most built-in sort functions. This can be quite useful for languages that do not use the twenty-six letter Roman alphabet exclusively, or for languages in which some groupings of letters might be equivalent to other single letters. In the case of Old English, you really need to do things like sort “æ” and “a” as equivalent characters, and “ð” and “þ” as equivalents with each other and with “th” (though “th” is relatively rare). Using built-in system sorting methods (for example, the sort command on the Unix command line and the sort() function in Perl), the special characters such as “æ,” “ð,” and “þ” always end up sorted well after their ASCII equivalents “a” and "th," since the former are UTF-8 character codes 166, 176, and 190 respectively, whereas “a” is code 97 and “t” is code 116 (see Appendix D for the decimal values of UTF-8 characters, which also includes ASCII as a subset). If the sort program or function uses the characters’ positions in the character encoding method in use in the file or program, the characters contained in the ASCII subset up to 126 will end up before the special characters with higher decimal values in UTF-8 or any other encoding type that uses the standard ASCII

character values. Also, the UTF-8 special characters are represented by multiple bytes rather than single bytes (all ASCII characters are single bytes), which affects the way the characters are handled by a number of built-in Perl functions. The lemmatization program (oe_lemma.pl) depends upon the words being sorted in the correct order, and the program I created using the Sort::ArbBiLex module solved this issue for the most part, though it does have the shortcoming of not allowing rules for equating one letter or set of letters with other letters more than once. For instance, you cannot both equate "i" and "y," and still have "y" sort in its normal place before "z" under certain circumstances. I chose to include all the versions of the letter "y" in with all the versions of the letter "i" because this was more likely to yield correct sorting results than separating them (since these two letters were so often used interchangeably in the extant Old English texts).

In oe_sort.pl, I created a function with Sort::ArbBiLex called "oe_sort" in which I equated various letters or groups of letters with each other based on manual visual scans of the word lists for evidence of possible equivalent letters. For instance, for the letter "a," I found instances of "aa," "aæ," etc. for words containing "a" and "æ," spellings like *aæfter*, *after*, and *æfter* all for the same word, and *aarwyrðe* and *arwyrð*, both variants of the word *arweorð*, meaning "honorable, venerable, revering, pious" (Hall 25). Because of this evidence, and because of the a number of accented characters also present in the corpus, I equated "a," "A," "à," "À," "á," "Á," "â," "Â," "ã," "Ã," "ä," "Ä," "å," "Å," "æ," "Æ," "aa," "aæ," "æa," "ææ," "Aa," "Aæ," "Æa," "Ææ," "AA," "AÆ," "ÆA," and "ÆÆ" with each other, then the expected "b" and "B, then "c," "C," "ç," "Ç," "k," and "K" (because, as with "i" and "y," "c" and "k" are often used interchangeably), "d" and "D," etc. Despite having output all words as lowercase from AntConc, and having included a conversion to all lowercase letters in the subsequent

oe_lemma.pl program (to be discussed later), I equated both lower and uppercase letters in the oe_sort function definition just in case there was ever a need to sort a file that includes uppercase letters. I also equated all forms and double instances of thorn and eth with each other and with all possible variations of “th.” And as with “a,” I equated all accented vowels with their unaccented counterparts (to view the full oe_sort() function and all its equated letters, see the oe_sort.pl source code in Appendix A). Some of the included accented characters are rare (or even non-existent) in the corpus, but I decided to err on the side of caution and include a few other possibilities since many accented characters are present in the text.

After the sort order definition, the program then takes the file "wl_all_byword.txt" (a complete wordlist generated from the entire *DOOE Corpus* using AntConc text analysis software) as input, splits the three fields in each line (number, frequency, and word), and outputs a new text file called wl_all_words_only.txt that includes each word on a separate line without the other information (this file is created for use within the program, but it is one of the output results of the program). It then takes the new file, sorts it using the oe_sort() function, and outputs the sorted list of words to the file wordlist_sorted.txt, which after running against wl_all_byword.txt contains 260,557 separate words just like the input file.

The second program, oe_lemma.pl, takes the sorted word list from the previous program as input (the file wordlist_sorted.txt) and outputs a file containing the equivalent words (or at least words that are very similar to each other) grouped together. The program first makes a few character conversions to conflate commonly interchangeable letters for normalization purposes. For instance, the program translates eth (“ð”) into thorn (“þ”) in every case since these characters are entirely interchangeable and really should not count toward the number of edits. It also translates ash (“æ”) into “a,” “c” into “k,” and “i” into “y” because they are similarly

interchangeable, at least in practice, throughout the corpus. I also added in translations of all accented characters (for all the vowels and the l-bar ("ƚ") and c with cedilla ("ç")) into their base characters for similar reasons (for instance, "à," "á," "â," "ã," "ä," "å," and "æ" all became "a" for stemming and comparison purposes). Despite the translations, the letters in the original words retain their accented forms for output purposes since the original remains in its own variable. Removing these accents permanently for inclusion in the lemma list would, once it was imported into text analysis software, cause any words with accented characters not to be recognized as equivalents to the same words without the accented characters. The program then stems the words (removes common endings based on a list of possible endings and their variants). And after stemming and character conversion, the program does comparisons of the first stemmed word to the second stemmed word in the list, the second to the third, and so on and so on, using the Levenshtein algorithm. It compares the stems rather than the full words since inclusion of the endings could cause the Levenshtein edit distance to be high even for words that are essentially different forms of the same word, and could therefore fail to group like words together as it should unless the allowed difference was set higher. A higher allowed difference, however, would likely cause many words to be grouped together that should not be. Because in this case the word lists were generated from the text analysis software as all lower case, case was irrelevant, but the code converts each word to lowercase using the Perl lc() function just in case this is not true of every input file (changing the source code to convert them all to uppercase instead, if the user's purpose requires it, is as easy as changing lc() to uc() in the code, and changing the cases of the hard-coded letters in the character conversion section). Some additional code might be required to handle mixed cases. For some purposes it might be best not to change the case; however, it is often essential to normalize to one case or the other, since

for the purposes of importing a list into analysis software you most likely want *Drihten* equal to *drihten*. Using multiple cases may result in incorrect results for sorting and other functions as well if you do not explicitly make your sorts and other functions case-insensitive. In the current code implementation, everything is case sensitive, but this, too, could be changed with a little added code.

The following is a close step-by-step explanation of the workings of the oe_lemma.pl program. It was necessary to include two Perl modules using the "use" command: "utf8" and “Encode,” (the latter a CPAN module developed by Dan Kogai) both to allow for proper handling of UTF-8 characters. In earlier versions, I also utilized CPAN versions of the Levenshtein algorithm (Text::Levenshtein and Text::LevenshteinXS), but found that they were not handling the special characters properly (characters such as “æ,” “þ,” and “ð” would count as two characters rather than one in the Levenshtein calculations), so I abandoned them in favor of a hard-coded Levenshtein algorithm function within the main program and tested it thoroughly with various word pairs to make sure that it was returning correct results.

The program opens the files wordlist_sorted.txt (the output of oe_sort.pl) and endings.txt (a list of common Old English inflectional endings) as input files, and opens wordlist_lemmas.txt, wordlist_unmatched.txt, and wordlist_compounds.txt as output files (the first will contain all the groups of lemmas, the second any individual words that were unmatched, and the third any suspected compounds). Each word in wordlist_sorted.txt is read into the array @words, and each ending from endings.txt is read into the array @ends. Currently the endings include the following, some taken from *Introductory Old English* by Jonathan Evans (297-309), and others emulated using spelling variants existing in the texts (in the file itself each ending is on a separate line):

a, æ, e, i, þ, ð, an, æn, as, æs, aþ, æþ, að, æð, de, en, es, eþ, eð, ie, iþ, ið, ra, re, st, um, ast, æst, ena, est, don, ede, ese, iaþ, iað, ode, dest, edon, esse, odon, edest, odest

Note that the above list has been changed many times in testing, altering the results in various ways every time, and can be modified outside of the program to best suit the user's purposes.

Variables $curr, $currshort, $currtrans, $prev, $prevshort, $prevtrans, $currlist, $lastchars, $endflag, $endflag2, and $j are all initialized for use in the main loop (most are set to blank, with the exception of $prev and $prevshort which are set to "fakeword" and "fakew," the latter of which is used in an if statement within the program to skip Levenshtein comparison of the very first word before an actual previous word exists). A little later within the main loop, the first three "curr" related values will be set to the current word being processed, a stemmed version of the current word (known endings removed), and a translated version of the current word (various characters translated into other characters for normalization purposes) respectively. The next three "prev" related variables will be set to the current word, current stem, and current translation at the end of the main loop so that during the next iteration, the last current word is the previous word. The @wordlist, @compounds, and @unmatched arrays are initialized to empty arrays (their values, as with all the initialized variables, will be filled within the main loop).

Most of the main processes of the program are held in a foreach loop, a control structure in Perl specifically for looping through every element in an array. The loop iterates through every word in the @words array, and appears as follows (the ellipsis indicates code within the loop):

```
foreach $word (@words)  {
...
}
```

The above code defines the variable $word to be the current element in the @words array and begins the loop. The first actions within the loop are used to weed out short one and two character words. The whitespace is trimmed from the current word (and set to the new variable $word_trim) and any existing newline character at the end of the word is removed using the trim() and chomp() functions (trim is a user defined function defined after the main program code, and chomp() is a built-in Perl function). $word_trim is then decoded from "UTF-8" (using the decode() function that is part of the included Encode module). This allows the special characters within the current word (if present) to be recognized as single characters rather than two bytes (due to the nature of Unicode and the built-in functions, strings have to be decoded in order for functions like length() to yield correct results), and the length of the decoded word is determined using the built-in length() Perl function. An if statement determines whether the length of the current word is greater than two, and if so, continues. Allowing these very short words caused some adverse effects, including them not matching each other properly and generally skewing the other results for the worse when using them to determine possible compounds, so I made the decision to make these words appear as unmatched. This if statement surrounds the rest of the code within the main foreach loop and, at the end, an else statement pushes any word of two or fewer characters to the @unmatched array automatically.

$endflag and $endflag2 variables are then set to "n" (for later use in setting the appropriate $currshort value and looking for compounds). The $curr variable is set to a trimmed version (whitespace removed) of $word (the current word being processed), and then chomped of any newline character. $curr is converted to lowercase using the Perl function lc() (as I noted earlier, in my testing the input file was always all lower case, and therefore case was irrelevant in this study, but this might not be the case if someone else uses the software). Then $curr is

decoded from UTF-8 using the decode() function. After decoding $curr, a variable $currlen is set to the length of $curr, using the code `$currlen = length($curr)`. $prevlen is similarly set to the length of $prev (which again is a variable that is set to $curr at the end of the main loop for use in the next comparison; in the first iteration through the loop, $prev is equal to "fakeword," the value it was initialized to before the loop). $currtrans is then set to the value of $curr and undergoes a number of transformations using the tr/// operator to prepare it for later use in stemming and in identifying compounds. The code `$currtrans =~ tr/kiæð/cyaþ/;` translates every character on the left of the second slash mark to the corresponding character to the right (i.e. "k" is changed to "c," "i" is changed to "y," "æ" is changed to "a," and "ð" is changed to "þ" in the $currtrans variable so that it is a transformed version of $curr). Similar translations are done to change all accented vowels to their unaccented counterparts, and "ç" to "c." An additional transformation uses the s/// operator, which allows you to search for the pattern to the left of the second slash mark and replace it with the pattern to the right (this can be used in a way similar to tr///, but also allows for regular expression pattern matching). The code `$currtrans =~ s/(.)\1/\1/;` finds any characters that appear twice and reduces them to a single character ("aa" will become "a," "ll" will become "l," etc.). In regular expressions, the backslash ("\") followed by a number repeats anything that was contained in parentheses that correspond to that number (i.e. if you have three different patterns contained in three sets of parentheses, "\1" matches whatever is contained in the first set, "\2" whatever is contained in the second, and "\3" whatever is contained in the third). In this case the "." matches any one character, and the "\1" repeats the character (in the first section to match the previous character and in the second to replace everything in the first section with the single character).

Another foreach loop is then started, this one iterating through every ending in the @ends array (using `foreach $end (@ends){ ... }`). This section in effect emulates a stemmer by removing any endings that are found that match one from the endings.txt file (which are stored in the @endings array). The foreach line sets each ending in the array to the variable $end. Within the loop, the current $end variable is trimmed of whitespace and chomped of its final newline character. $end is then decoded from UTF-8 to allow for proper processing, and $endlen is set to the length of $end.

An if statement checks to make sure that $currlen (the length of the current word) is greater than $endlen (the length of the current ending) before proceeding (since it would be impossible to remove an ending that was longer than the current word). If the test is true, $lastchars is set to the last characters in the current translated word ($currtrans) for the exact same number of characters as the current $end length with the following code:

```
$lastchars = substr($currtrans, ($currlen - $endlen), $endlen);
```

This is using the Perl substr() (or substring) function, whose three arguments are the string you want to reduce to a smaller subset, the starting location, and the number of characters you want to grab. Positions in Perl start at zero (which is a little confusing), so, for instance, if you did `substr('Hello', 2, 3)`, it would return "llo," because you start at position 2 (which is in reality the third position of 0, 1, 2, 3, and 4 in the word “Hello”) and go for three characters, returning what is found in those positions. With the above code, $lastchars is taking $currtrans and returning the position equal to $currlen minus $endlen for the value of $endlen characters (the length variables contain integer values equal to the length of the current word or ending). Again, because positions in Perl start with zero rather than one, all the code has to do to find the

position of the beginning of the last characters is subtract $endlen (the length of the current ending) from $currlen (the length of the current word).

Then another if statement checks to see if $end is equal to $lastchars, and if it is, $rootlen is set to $currlen minus $endlen (giving us the length of the root word or stem). $currshort is set to the substring of $currtrans starting at position zero for $rootlen characters (with the code `$currshort = substr($currtrans, 0, $rootlen);`), which in other words results in the current translated version of the word minus the ending. Then $endflag is set to "y" to indicate that a matching ending was found and $currshort was set. This could conceivably result in $currshort being set multiple times if there are multiple possible endings that match the end of the word. The endings in the file endings.txt are currently arranged from shortest to longest so that the longest match ends up being removed (since the longer endings are checked for last). The @ends loop is then exited, and an if statement checks to see if $endflag equals "n" (meaning no matching endings were found). If so, $currshort is set to $currtrans (the full translated version of the current word) rather than a shortened version.

The next lines of code start the Levenshtein comparison. The program sets $pshortlen and $cshortlen to the lengths of $prevshort and $currshort, and performs the Levenshtein comparison of $prevshort versus $currshort with the following code:

```
$lev = levenshtein($prevshort, $currshort);
```

The function levenshtein() is defined at the end of the program (and the function is described in detail above in the "Levenshtein Algorithm" section). The code passes the variables $prevshort and $currshort (the translated shortened versions of the previous and current words) into the levenshtein() function, which does the comparison and returns the edit distance between the two (i.e. the number of changes that would have to be made to turn one into the other). The short

versions are used instead of the full words because the presence of differing declensional endings would skew the results and make many of the words appear more different than they actually are. For instance *cining* and *cyningum* should be equated, and comparing the translated and stemmed short versions using Levenshtein should result in the two both being compared as *cyning*, which would result in a 100% match. The program then tests for $pshortlen greater than $cshortlen. If the condition is true, it sets $diffpercent to a ratio of $lev (the Levenshtein value calculated above) divided by $cshortlen, and if false, sets $diffpercent to $lev divided by $pshortlen. Dividing the Levenshtein value by the shorter of the lengths of the two compared words provides a percentage value that helps take into consideration the possible effect of the length of the words on the importance of the Levenshtein value. For instance, if the compared words are both only three characters long and the Levenshtein edit distance is 1, that is a much bigger deal than a Levensthain distance of 1 with two words eight characters in length. The $diffpercent value is .333 percent in the first case, whereas it is only .125 in the second case. This decimal value is then used by the program to determine whether the words are similar enough to consider essentially equal, and if they are, it groups them together as related lemmas in the code that follows.

The next section of code checks for a $diffpercent of less than 0.3. I tested with various other numbers, as well, and a smaller number would likely be better with some improvements in the sort order and code functionality. Before continuing the code explanation, here are some examples of $lev and $diffpercent calculations with certain word pairs to show what Levenshtein does and what these comparisons really mean in practice. All were output by two versions of a short program that takes a two-word file, does a Levenshtein comparison identical to the one in the main program, and prints the results to the screen (the version run on the first three examples

did a straight Levenshtein comparison, and the other version run on the fourth and fifth examples also did the character translations before comparison). The first is of two equivalent words that contain three alternate but often interchangeable letters:

Previous word: æðelcininge

Current word: aþelcyninge

Levenshtein distance: 3

Prevword length: 11

Currword length: 11

Difference percentage of shortest word: 0.272727272727273

The above results in a Levenshtein distance of 3 (which would not be the case in the actual program because of the translations, although in this case the $diffpercent result would still allow inclusion in the same @wordlist element). But below, changing the ending of the current word from -*e* to –*um* causes the difference to go above the acceptable threshold for the program:

Previous word: æðelcininge

Current word: aþelcyningum

Levenshtein distance: 5

Prevword length: 11

Currword length: 12

Difference percentage of shortest word: 0.454545454545455

The above case is an example of why the ending removal and character conversion can be quite useful in getting better matches using Levenshtein. In both of the previous cases, normalizing the previous and current word to a shortened and converted version (as the program does) yields this result:

Previous word: aþelcyning

Current word: aþelcyning

Levenshtein distance: 0

Prevword length: 10

Currword length: 10

Difference percentage of shortest word: 0

The above shows an exact match. If this happened in every case, Levenshtein would be unnecessary, but often enough there are other variations, such as substitutions of various vowels for one another or endings that are not in the list of expected endings. The following example shows a case from the wordlist_lemmas.txt result file that would not have been a 100% match but instead relied on Levenshtein to equate the words (namely *dryncfæt* and *drincfatu* with the first two vowels normalized to "y" and "a"):

Previous word: dryncfæt

Current word: drincfatu

Translated previous word: dryncfat

Translated current word: dryncfatu

Levenshtein distance: 1

Prevword length: 8

Currword length: 9

Difference percentage of shortest word: 0.125

These words end up with a difference threshold below the acceptable level of 0.3 and are equated in the lemma file (in a line including *drincfæt*, *dryncfæt*, *drincfatu*, and *dryncfatu*).

Another example of equivalent words that required Levenshtein to get a match is the comparison between *dropfag* and *dropfah*:

Previous word: dropfag

Current word: dropfah

Translated previous word: dropfag

Translated current word: dropfah

Levenshtein distance: 1

Prevword length: 7

Currword length: 7

Difference percentage of shortest word: 0.142857142857143

The program, after stemming and in one case conflating a double character, correctly equates *dropfag*, *dropfaag*, *dropfagum*, and *dropfah* and puts them in their own @wordlist element. The letters "g" and "h" are used interchangeably quite a bit, but not often enough to do a permanent translation of one into the other in the program. We would not, for instance, want to translate *heap*, meaning "host, crowd, assembly, company, troop, band," (Hall 174) into *geap*, meaning "open, wide, extensive, broad, spacious," etc. (Hall 149).

In the current incarnation of the program, the next section of code uses an if statement to check for the difference percentage and continues if it is below the threshold:

```
if ($diffpercent < 0.3 && $prevshort ne "fakew") {
```

The second part after the logical "and" operator ("&&") checks to make sure that $prevshort is not equal to "fakew" (the initial fake previous shortened word, which would only happen on the first iteration through the main loop) because we do not want to do a comparison on the very first word until the next iteration through the @words array when it will be the previous word. If the Levenshtein comparison results in a value below the threshold, the $currlist variable is set to

$prev plus a comma and space plus $curr (to create a list containing previous word and current word separated by a comma). If $wordlist[$j] (which will begin as $wordlist[0], the first value of the array @wordlist, and then increase to the next element of the array each time $j is incremented) is blank, the last added value of @unmatched is removed via the command `pop(@unmatched)` (words will begin to be pushed to @unmatched later in the code, but the removal is here to remove any that were considered unmatched during the previous iteration but now appear to have a match), and $wordlist[$j] is set to $currlist. Otherwise (via the else portion of the if statement) if $wordlist[$j] is not blank, $wordlist[$j] is set to $wordlist[$j] plus comma and space plus $curr, in effect adding the current word to the end of the current $wordlist. This is how words are continuously added to each group (until a word that is too different is encountered).

The `if $diffpercent < 0.3` if statement is continued with an elsif (or “else if”) segment and then an else segment. This means that if the threshold is not below 0.3, it does the test in the elsif statement (which if true, executes the code enclosed within its brackets), and if the elsif is likewise not true, it executes the statements within the else statement’s brackets (since else is a sort of default or catchall that does not perform any sort of test). The first, the elsif, is as follows:

```
}  elsif ($currshort =~ /^${prevshort}(.+)/ || $currtrans =~
/^${prevtrans}(.+)/) {
```

The above uses the pattern-matching operation that is sometimes rendered as m//, although the “m” is optional and has been omitted. The material in between the slash marks (in both cases) is a regular expression used to look for a pattern. In this case the first matching operator is looking for a pattern in the variable $currshort. The “^” symbol indicates the beginning of the string you are searching and “${prevshort}” is simply the variable $prevshort (the curly brackets are

necessary within regular expressions in Perl to represent variables), so in this case, it is looking for the value of the variable $prevshort at the beginning of $currshort. In the next section within parenthesis, the "." means any one character, and the "+" that follows indicates one or more times. In other words, the entire first matching operator is looking for $currshort to begin with the stem of the previous word plus any other characters. The second matching operator is looking for $currtrans to match $prevtrans plus one or more characters. This line of code is the beginning of the check for possible compounds, or at least for words that begin with something like the previous word but end with characters that are not a known ending, in an attempt to root out words that might be breaking up the sort order, or rather separating words that would otherwise match (this can happen when the next word's ending begins with a character greater than that of the first word, but less than the last word's; for instance the words *drihtena*, *drihtenbealu*, and *drihtene*, where the letter "b" in –bealu, separates the first and third words because "e" comes after "b" alphabetically). The logical or operator ("||") means that if either the first or second condition is true, then the test is true and the statements within the elsif section are executed. If either is true, a new foreach loop iterates through the endings. First, $end is trimmed of whitespace, chomped of the newline character, and decoded just as before. Then an if statement checks to see if $currtrans matches a pattern beginning with $prevtrans and followed by $end or a pattern beginning with $prevshort and followed by $end via the following code:

```
if (($currtrans =~ /^(${prevtrans}){${end}}$/) || ($currtrans =~
/^(${prevshort}){${end}}$/))
```

In this case, much like "^" checks for a pattern at the beginning of the string being checked, "$" at the end of the pattern-matching operator is checking for the preceding pattern at the end of the string (this is a regular expression convention that can be a little confusing in this case since "$" also indicates a variable, which is likely why the curly brackets are required for variables within

Perl pattern matches). If either condition is met, $endflag2 is set to “y” to indicate that a match was found. After the foreach loop ends, an if statement checks to see if $endflag2 equals “n” (meaning no matching word plus ending was found) and $cshortlen is greater than $pshortlen plus two, and if both conditions are true, $curr (the current word) is pushed into the @compounds array as a possible compound, the variables $currshort, $currtrans and $curr are reset, and the “next” statement takes the program back to the beginning of the $word foreach loop (in other words, skipping over all the following code and starting at the next $word in the @words array). As stated before, the compound checking is not so much finding words that are definitely compounds, but words that begin with the previous word plus something that is not a known ending (which makes the word a likely compound candidate). The “+ 2” keeps something that begins with the previous word but ends with something that cannot be a full word (and therefore would not indicate a compound but might just be an unmatched word, meaning a new @wordlist element should be started) from being skipped over as a possible compound. If the two conditions are not met, an else section pushes the word to the @unmatched array since it is above the allowed threshold (for possible removal from @unmatched in the next iteration through the main loop depending upon how the next comparison turns out).

Regarding the compounds code above, later in the program the @compounds array will be output into a file, which can be run through the program later as the input file to generate another lemma list of just these words. These words, which may or may not be proper compounds, will be dealt with as their own lemmas for comparison to each other, and the new lemma list can be easily added to the larger lemma file.

The next `else` segment is the third and final element in the `$diffpercent < 0.3` if statement, which executes if neither the main `if ($diffpercent < 0.3 && $prevshort ne`

`"fakew")` check nor the following `elsif` check are true. It pushes $curr to @unmatched as well, and this time increments the value of $j by one (this will only happen if, by the standards of the program, the words are not matches at all and therefore a new @wordlist array element needs to be started to end the previous lemma group and start a new one). The two pushes to @unmatched might seem repetitive, but the first one is pushing a word that seems to be the previous word plus a known ending to the array just in case it does not match the next word (which is why the next time it goes through the levenshtein() check the last word added to @unmatched may be removed), and the second one is pushing words that appear not to be matches at all so that a new lemma group can be created.

The final lines of the $word `foreach` loop set $prev to $curr, $prevshort to $currshort, and $prevtrans to $currtrans, and then set $currshort, $currtrans and $curr to blank, so that the current word in the current iteration becomes the previous word in the next iteration through the main loop.

Note that @wordlist will only be populated when matches are found; therefore, if a word is not deemed to be a close match to the word previous to or following it (and is not deemed a possible compound), it will not be included in the lemma list but should be included in @unmatched. Because my purpose is to create an importable lemma list that allows text analysis software to equate words with each other, it is not necessary (and may even cause problems) to include lone words with no matches in the main file, but it may be useful to keep track of the unmatched words for other purposes including error checking, so I included the @unmatched array to keep track of these words.

Once the program has iterated through every word in the @words array, the loop is exited. A very short loop iterates through all the words in the @compounds array, encodes each

word back into UTF-8 for output, and then prints the words to the wordlist_compounds.txt file one at a time via the following code:

```
foreach $compound (@compounds) {
      $compound = encode('UTF-8', $compound);
      print MYCOMPS "$compound\n";
}
```

Note that after decoding a variable, encoding via the encode() function (also defined in the Encode module included at the beginning of the program) is necessary in order for the special characters to output to the file properly. The code `print MYCOMPS "$compound\n";` prints each element $compounds in the @compounds array to the MYCOMPS file (an alias given to the file wordlist_compounds.txt when it was initially opened). The "\n" indicates a newline character so that each word is printed on a new line.

Two nearly identical foreach loops (`foreach $unmatch (@unmatched) ( ... }` and `foreach $word (@wordlist) { ... }`) iterate through all the words in @unmatched and lists in @wordlist, encodes them, and prints them to the wordlist_unmatched.txt and wordlist_lemmas.txt files respectively. After this, all the open files are closed. The remainder of the program consists of the definitions of the trim() and levenshtein() subroutines (or user defined functions) used in the main program above them.

5. Results

The result that I hoped for was a file containing accurate groups of equivalent words that could be used as a lemma file for import into text analysis software. Unfortunately, the current results are not perfect, but they are good enough to show a great deal of promise toward reaching this goal using the Levenshtein algorithm, and the current basic lemmatizer may be a good

jumping off point for development of a more robust lemmatizer for Old English. The following is the first page of the file wordlist_lemmas.txt (27 lines/groups total):

abacæ, abacen, abaken

abaci, abacta

abacuc, abacus

abad, abæd, æbæd, abædan, abædað, abædde

abædede, abæden, abædeþ, abædeð

abal, abælgede

abær, æbær, æbæra, abarast, abarað, abære, æbære, abæred, abarede

abarian, abarim

abarn, abarndest, abarnodest, abaro, abæron, abærst, abarude

abbad, abbade

abbadissan, abbadisse, abbadyssena

abbadunæ, abbadune

abbae, abban, æbban

abbas, abbaso

abbate, abbatem, abbates

abbati, abbatis, abbatissan

abbedesse, abbedessę, abbedessen, abbedisse, abbedysse

abbendone, abbendune, abbendunes

abbod, abboda, abbodæ, abbodan, abbodas

abboddum, abboddune

abbode, abbodes, abbodessene

abbodhade, abbodhades

abbodyssa, abbodissan, abbodyssan, abbodisse, abbodysse, abbodissum

abbodrice, abbodricen, abbodrices

abbote, abboten

abbotrice, abbotrices, abbotriche

abbud, abbuda, abbudas

Each line is an element of the array @wordlist from the program. All the words in each line are very similar, though there has been some breaking up of like words, such as *abbadunæ* and *abbadune* from *abboddune*, which appears eight lines further down. Also, there are a couple of cases where words do not necessarily belong with one another (for instance *abarian* ("to lay bare, disclose" (Hall 1)) and *abarim* (uncertain definition) are likely not related unless the latter is a misspelling), which would be an issue with the Levenshtein comparison. Another later Levenshtein failure (though not the only other one) is the grouping of the three words *dryhtum*, *drihð* and *dryhð* in one list. The latter two are the same word, the present third person singular of the verb *dreogan* meaning "to lead a (certain) life, do, work, perform, fulfill…" (Hall 90), whereas the first is the dative plural of *dryht,* which means "multitude, army, company, body of retainers, nation, people" (Hall 89). There is only one different character between them (once the *-um* is stemmed), which would have resulted in a Levenshtein difference of 1, and at five characters for the stem, a difference percentage of 0.2, below the 0.3 threshold.

When the acceptable level of similarity was not met by the current word to previous word comparison, the next @wordlist element was begun and another complete @wordlist element resulted until the next dissimilar word was encountered. Any words meeting the requirements for a "compound" were pushed to the @compounds array (skipped in favor of the next possible

match) and printed to the file wordlist_compounds.txt, the first page of which is as follows (27 words total, divided into three columns left to right):

abædendre	abbachus	abbeddune
abanet	abbacuc	abbedessan
abanlake	abbacvc	abbodesbirig
abannan	abbadesse	abbodesbyrig
abanne	æbbadesse	abbodessan
abans	abbanberghe	abbudessan
abarcit	abbandune	abbutissan
abarimathia	abbandunes	abbutissena
abarimathiam	abbatibus	abdicare

The compounds file can be run through the same program, resulting in a shorter lemma list of the suspected compounds with similar words grouped together in wordlists. As stated before, the results should be easy to manually add to the wordlist_lemmas.txt lemma file before import into the text analysis software. In fact, I ran wordlist_compounds.txt through a version of the program changed only to modify the expected input and output file names, and the results were surprisingly accurate, with a few erroneous matches at first glance (*abeam* does not belong with *abeat*, *abeate*, and *abeatne* in the results below, for example). The following is the first page of the results (the first 27 lines of the file wordlist_lemmas_2.txt, generated from wordlist_compounds.txt):

abanne, abans

abarimathia, abarimathiam

abbacuc, abbacvc

abbadesse, æbbadesse

abbandune, abbandunes

abbodesbirig, abbodesbyrig

abbodessan, abbudessan, abbutissan, abbutissena

abdicare, abdicat, abdicatio, abdicatione, abdicatis, abdicatiue, abdicatus

abdicauit, abdicit, abdictis, abdictus

abdidere, abdidit

abducere, abducit

abducti, abductio, abductione, abductionem

abduxione, abduxit

abead, abeag, abeah

abealg, abealh

abeam, abeat, abeate, abeatne

abegelata, abegerant, abegerunt

abeodende, abeodenne

abigerant, abigerunt

abilgnesse, abylgnesse, æbylgnesse, æbilignesse, æbylignesse, æbilinesse,
æbylinesse, æbilinysse

abisgodne, abysgodne

abitationibus, abitationis

ablæcungum, æblæcungum

ablænd, ablændan

ablawung, ablawunge

abolgenne, abolgennes

abrerdnysse, abryrdnesse, abryrdnysse

And the third file, wordlist_unmatched.txt, should contain a list of words that were not included in the other files. The following is the first page of wordlist_unmatched.txt (27 words total, divided into three columns left to right):

a	abaddir	abaritia
æ	abæde	abaso
aa	abædon	abæst
aæ	abalhc	abat
æa	abælige	abatis
ææ	aballanus	abauctor
ab	abamita	abauditu
abacene	aban	abauia
abactus	abaredum	abauunculus

Note that there are possible compounds in the unmatched list (for example *abbedrice*, from the second page of the list). The conditional checks that result in words being added to the @compounds array are ways to weed out words that appear to be compounds beginning with the previous non-compound word, but it is not foolproof and not a valid way of getting actual compounds in the strictest sense. It is simply an expedient way of skipping what appear to be compounds that could be separating sets of like words. And some words that should have been grouped together appear in this list (such as *abæde* and *abædon*). They were likely separated in such a way that they were not near their close matches, resulting in them being pushed to @unmatched.

Dryhten, once again, is a good example of a group that ends up with both erroneous words included, and groups broken up erroneously. In the program running with a 0.3 difference, the following are the results for *dryhten* and some surrounding words:

drihst, driht, dryht, drihta, dryhta, drihtan, drihtæn, dryhtæn

drihte, drihten, dryhten, drihtena, dryhtene, drihtenes, dryhtenes, drihtenna, drihtenne, drihtennes, drihtenum, drihter

drihtin, drihtyn, dryhtin, dryhtyn, drihtine, drihtines

drihtlecu, dryhtleoð, drihtlic, dryhtlic, drihtlican, drihtlice, dryhtlice, drihtlicu, drihtlicum

drihtman, dryhtmon, drihtn, drihtna, dryhtna, drihtnas, drihtnæs, dryhtnas, dryhtnæs, drihtne, dryhtne, drihtnen, drihtnes, dryhtnes, drihtneum, drihtny, drihtnys, drihtno, drihtnum, drihton

drihtscipe, drihtscype, dryhtscipe, dryhtscype, drihtscipes, dryhtscipes

drihtsele, dryhtsele

drihtten, drihttenes

dryhtum, drihð, dryhð

drii, dryi

driyhten, driyten

One of the issues causing some of the erroneous matches and breaks is the use of a list of endings (discussed in a bit more detail in the next section), which includes *–en* and *–ena*, among others, for stemming purposes. These are not inflectional endings in words like *dryhten* and *dryhtena*, but they are being removed before comparison, causing them to match *dryht* and related words, but not match *dryhtenum* (which is stemmed to *dryhten*). Some possible solutions

are discussed in the next section. And actually, in this particular case, lowering the allowed difference ratio to 0.25 causes *dryhtenum* to be pushed to the unmatched list, leaving *drihð* and *dryhð* correctly matched. Others are broken up by unmatched words in between various groups that should match. Notice the previously mentioned *dryhtum*, *drihð*, and *dryhð* line above. *Dryhtum* should be grouped with *driht*, *dryht*, *dryhta*, *drihte*, etc., but the former are near the very beginning of this segment and *dryhtum* is near the very end, separated by many *dryhten* related words and compounds.

The program does not work perfectly and anomalies will no doubt be found in many places, but the current results show moderate success in automatic lemmatization that can be expanded upon with future research. And the lists of grouped words should be easier to go through and correct than a long list of ungrouped words.

To illustrate the effect on text analysis results, I ran the following sed command on the Unix command line to take the wordlist_lemma.txt file and convert the first comma in each group into "->" since that is the format expected by AntConc for the lemma file:

```
sed "s/, / -> /" wordlist_lemma.txt > wordlist_lemmafile.txt
```

To convert every comma, you would put a "g" at the end of the s// search and replace command, but in this case, finding and replacing the first instance found in each line is what I wanted. It results in a file like the following (as exemplified by a few lines from the middle of the first page; see Figure 2.1 for the lemma list as rendered by AntConc):

abær -> æbær, æbæra, abarast, abarað, abære, æbære, abæred, abarede

abarian -> abarim

abarn -> abarndest, abarnodest, abaro, abæron, abærst, abarude

abbad -> abbade

abbadissan -> abbadisse, abbadyssena

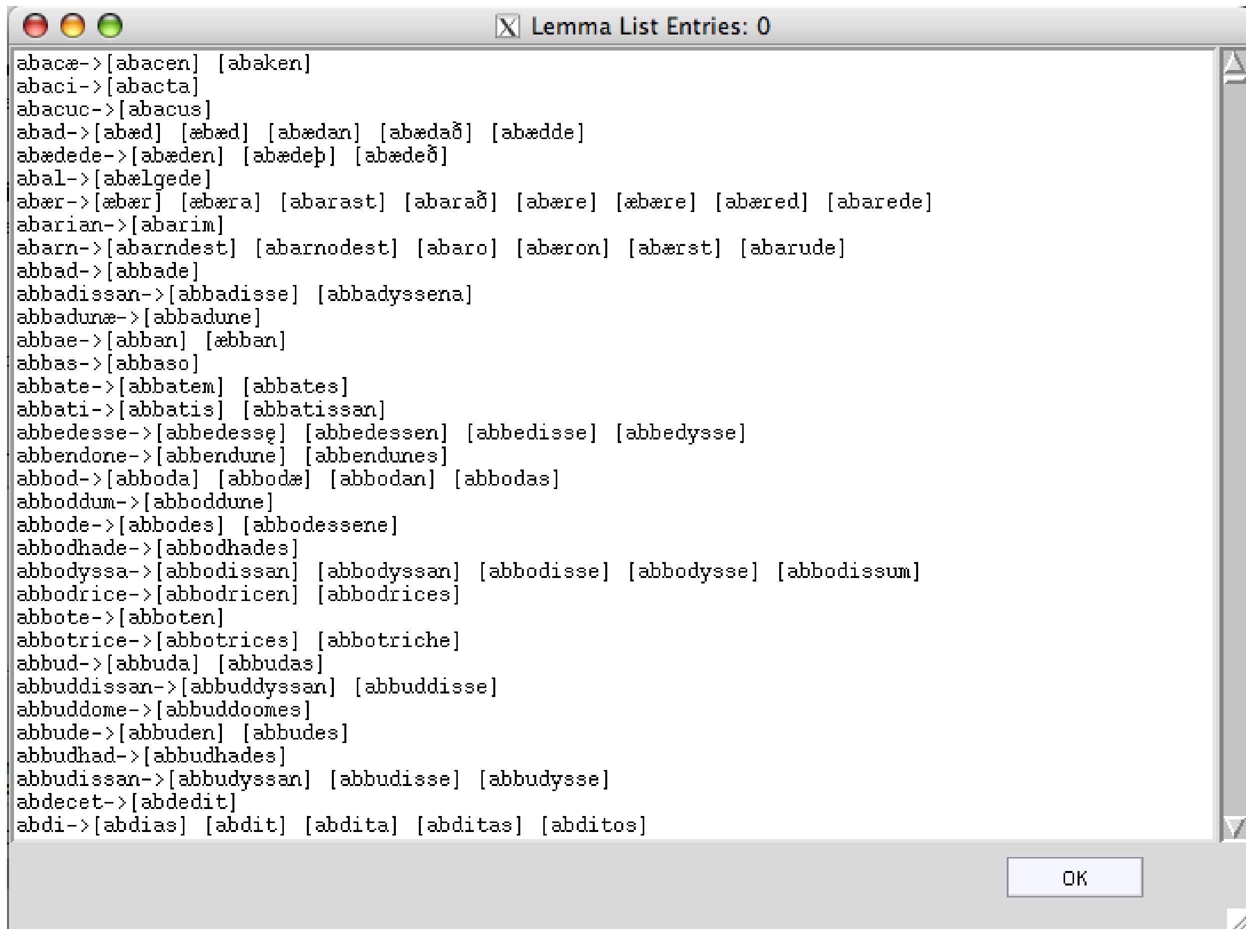

Figure 2.1. Screenshot of beginning of lemma list after importing into AntConc text analysis software.

AntConc is now ready to generate word lists and keyword lists using the lemma list. The following figures (Figures 2.2, 2.3, 2.4, and 2.5) illustrate some of the differences between the analysis results before and after importation of the lemma file (using the entire *DOOE Corpus* as the reference corpus and the poetry files as the analyzed corpus). Notice the addition of the lemma forms to the right of the words in the second wordlist image, and the conflation of *drihten*

and some other forms under *drihte* in the second, along with the increase in frequency. And in the two keyword list figures following, notice the changes, including *drihte* being moved up in rank to the fourth word with the fourth highest "keyness" value. Though some of the words are conflated incorrectly, there is such a high number of *drihten* related words that with a correct lemma list, the results would be to bump *drihten* up to one of the higher keywords, as well. Importation of the lemma file also significantly changed the makeup of the second version of the keyword list; only three words are the same on both the first list (Figure 2.4) and the second list (Figure 2.5), and the keyness values increased considerably.

Concordance | Concordance Plot | File View | Clusters | Collocates | Word List | Keyword

Hits | Total No. of Word Types: 27314 | Total No. of Word Tokens: 177701

Rank	Freq	Word	Lemma Word Form(s)
18	1082	þu	
19	995	in	
20	899	þær	
21	853	þonne	
22	811	þam	
23	808	is	
24	714	nu	
25	705	þæs	
26	688	of	
27	660	ofer	
28	657	hi	
29	653	god	
30	631	ða	
31	617	þurh	
32	584	ðe	
33	565	drihten	
34	526	ealle	

Figure 2.2. Screenshot of the second page of a word list generated by AntConc before the lemma list is imported (in descending order by word frequency).

Concordance | Concordance Plot | File View | Clusters | Collocates | Word List | Keyword

Hits 0 | Total No. of Word Types: 18461 | Total No. of Word Tokens: 177701

Rank	Freq	Lemma	Lemma Word Form(s)
18	1110	þam	ðam 154 ðæm 48 þam 811 þæm 97
19	1082	þu	
20	1031	þar	ðar 2 ðær 119 þar 11 þær 899
21	1016	þonne	ðonne 163 þonne 853
22	995	in	
23	974	þas	ðas 23 ðæs 92 þas 154 þæs 705
24	808	is	
25	777	drihte	drihte 1 drihten 565 drihtenes 6 dry
26	714	nu	
27	704	god	god 653 goda 50 godaes 1
28	688	of	
29	665	þurh	ðurh 48 þurh 617
30	660	ofer	
31	657	hi	
32	631	ða	
33	584	ðe	
34	539	biþ	bið 437 biþ 102

Figure 2.3. Screenshot of the second page of a word list generated by AntConc, this time with the lemma list. Note that scrolling to the right in the program is necessary to see all of the lemma forms.

Concordance | Concordance Plot | File View | Clusters | Collocates | Word List | Keyword

Hits	0	Keyword Types Before Cut: 27314	Keyword Types After Cut: 4067

Rank	Freq	Keyness	Keyword
1	2048	4584.994	ond
2	355	573.907	under
3	168	565.531	foldan
4	899	529.429	þær
5	159	524.827	waldend
6	194	491.510	wuldres
7	189	429.212	wide
8	137	377.496	weard
9	391	356.554	sceal
10	329	356.363	swylce
11	3317	354.639	and
12	247	333.967	wordum
13	714	316.038	nu
14	90	312.350	frean
15	91	311.995	gumena
16	165	309.656	engla
17	87	298.115	frea

Figure 2.4. First page of keyword list from AntConc with all *DOOE Corpus* texts as reference corpus and smaller poetry-only subset as analyzed corpus, before importation of lemma list.

Concordance | Concordance Plot | File View | Clusters | Collocates | Word List | Keywor

Hits | 0 | Keyword Types Before Cut: 18461 | Keyword Types After Cut: 2546

Rank	Freq	Keyness	Keyword
1	3145	15033.251	þat
2	1164	5267.016	was
3	2048	4584.994	ond
4	777	4578.011	drihte
5	1031	4079.691	þar
6	421	2525.988	sceafum
7	391	2390.216	drihtman
8	388	2371.786	swihþ
9	382	2295.773	mag
10	399	2068.312	after
11	974	1948.481	þas
12	305	1861.941	cinin
13	321	1791.382	bearm
14	289	1647.214	hafde
15	274	1627.268	hwat
16	247	1505.750	wordnum
17	269	1495.085	worcum

Figures 2.5. First page of keyword list from AntConc with all *DOOE Corpus* texts as reference corpus and smaller poetry-only subset as analyzed corpus, after importation of lemma list. Note that there are only three words on this page that were also in the pre-lemma-list results on the previous page: *ond*, *þar* (a version of *þær* from the previous page), and *wordnum* (a version of *wordum* from the previous page), and the keyness values have increased significantly.

6. Issues and Possibilities for Future Improvement

Though the results are promising, there are a number of issues that need to be addressed in order to make a more accurate automatic lemmatizer. The sequential nature of the comparisons in the current program contributes to some of the inaccuracies in the results, so perhaps in future versions, methods can be employed to compare arrays of words to one another

rather than using sequential comparison. It might be possible to use loops to compare every word to every other word, saving the pairs and Levenshtein values into another array and grouping the ones with the lowest difference percentage to each other (with some testing of the allowed difference threshold to pick an appropriate number). This would, however, require long run times and some work would have to be done to come up with the most efficient code possible for handling long word lists. Currently, with the job running against a list of 260,557 words, it is looping through every word once, with multiple loops against all the endings within each iteration. But to compare every word with every other word, within each iteration through the words another iteration through all the words would have to be performed, meaning the words would be looped through 260,557 x 260,557 times (or 67,889,950,249 times), which is a massive increase. The program takes approximately ten minutes to run against the current word list, so one can imagine that the increase in runtime would be substantial. Grouping and comparing only words that begin with the same letter might be another solution (still taking a while, but reducing the time required to compare every word to every other word), though some issues with first letter variations (such as "æ" and "e" in variants like *Edgar* and *Ædgar*) would have to be addressed. The sequential method seemed the most expedient for producing output values for every word right now, but some experimentation with comparison of larger groups of words with each other should be done to produce a better program.

Another possible solution somewhat related to the above would be comparison with a list of dictionary headwords. Comparing every word to a list of headwords, generated from a reliable source, would still be a large number of comparisons, but not nearly as large as the number of all the unique words in a large corpus, and it would have the additional advantage of allowing for a built-in first lemma. Currently for any groups of like words that are found, the

first word in the sorted list of words is the first one in the group. When imported into AntConc (or other text analysis software), this will make the first word in each group the on that shows up as representative of all the other words in the group (all statistics will show up as if they are for this word in a keyword list, for instance, although AntConc does include a list of the other conflated words to the right of the keyword). The first word is often not the best choice. The better choice would be the most basic uninflected form of the word, with the most commonly expected spelling. The current input data does not provide the program with enough data to determine this, but comparing against a fairly comprehensive list of dictionary words would be a start to solving this issue.

An additional possibility for solving the first word issue would be to use frequency (i.e. whichever version of the word appears most often in the text) as the first word in each lemma group. This could be attained by keeping a hash or array of words and their frequencies (loaded from the original word list from AntConc or other software), and for each group, doing a look up to find the highest number and moving that word to the beginning of the wordlist array. Again, this would add to run-time, but produce better results.

There are also issues with the way the way the program pushes "compounds" and unmatched words. Checking for a word starting with the previous word or stem, not containing an ending, and being greater in the length of the previous word plus two is not a very exact way to try to identify compounds, but with lots of trial and error, it yielded some promising results. Suspected compounds need to be removed because of the problem they present of dividing possible matching lemmas from one another. For instance, due to spelling irregularity, where you expect *drihten* or *dryhten*, you might actually find *drihtan*, *drihtæn*, *dryhtæn*, *dryhtin*, etc. In the sorted word list, there are variants and compounds that will separate the "a" and "æ" versions

of the word from the versions that would come later alphabetically (those with "e" and "i"), including *drihtanliccan*, *drihtealdor*, and *dryhtenbealo*, so that you end up with a list something like this (excerpt from actual oe_sort.pl results):

> drihtan, drihtæn, dryhtæn, drihtanliccan, dryhtbearn, dryhtcwen, drihte, drihtealdor, dryhtealdor, drihtealdormen, drihtealdre, drihtelican, drihtelice, drihtelicum, drihtelm, drihten, dryhten, drihtena, dryhtenbealo, dryhtenbealu, dryhtendom, drihtene, dryhtene, drihtenes, dryhtenes...

The *drihtin*, *drihtyn*, *dryhtin*, and *dryhtyn* variants end up much farther toward the end after many other compounds such as *dryhtenfolc*, *dryhtgestreona*, and *dryhtguma*, in effect separating variant versions of the same word with various compounds and non-equivalent words. And even with the compound pushing, these words are often still separated from each other when unmatched words appear within the long list of their variants (such as *drihtelm* above, which would not meet the criteria for a compound). One possible future solution would be to do a translation that removes all vowels (which exhibit much more irregularity in spelling than the consonants) and sort by the consonants only (or to both stem and remove consonants). But at the moment this is not possible with the current sorting program using Sort::ArbBiLex. The words would need to be sorted by the consonant-only version, but kept with the original intact word in a hash or array, so that the resulting sorted word list would contain only the original words.

The ending choices make a huge difference in the resultant data, as well, since some inflectional endings can also match legitimate word endings, such as the *–en* in the word *drihten* (inclusion of this ending causes *drihten* to be stemmed to *driht*, which is itself an Old English word, variant of *dryht* defined earlier, and it should not be lumped in with *drihten*). The endings are subject to the same spelling variations as the rest of the words, which causes problems with

correct stemming, as well. I included some possible variant endings in the endings.txt file, but some of those endings are also not always inflectional endings, but rather the last characters of legitimate words as in the above example. More experimentation is needed with selection of endings, and the program can also be expanded to include more rules for deciding when to stem and when not to stem. With the inclusion of a dictionary, rules could be put in place to take away the shortest possible ending (in this case -*a*) first and look for an equivalent dictionary word (or a very close "fuzzy" match), and then if it did not find a match, remove the next shortest possible ending (in this case -*na*) and look for a stem match, and then the next (-*ena*) and so on and so on. This would also allow for the possibility of removal of prefixes, which I deemed too risky given the likelihood of error in this simple lemmatizer. At present, prefixes are left on and prefixed variants of words are lemmatized together with their similarly prefixed counterparts.

There is also a possible issue with use of the *Dictionary of Old English Corpus* for generating the word lists, depending upon the user's needs. Though it is an invaluable resource in that it contains one copy of every extant text, it does not include multiple variant versions of the same texts. A corpus that includes all known versions of all texts, rather than one version of each, would be even more useful for generating an all inclusive word list that would include every variant that you might find. If a scholar used the *DOOE Corpus* to generate a lemma list, and used that list to analyze texts outside of this corpus, some words would not be properly equated with each other, resulting in some skewing of the data or a lot of manual labor to find and add all the missing variants to the lemma file.

Another issue with the lemmatization program that could not be solved (at least within the scope of this project) is with homographs, words that are spelled the same but have entirely different meanings (for example *god* in Old English, which can be either "god" or "good," one a

noun and one an adjective). Distinguishing two words spelled identically cannot be done easily in any computer automated way. One possible solution is to use inflectional endings to determine part of speech, but this would be difficult with Old English a least partially due to overlap in the endings between nouns, adjectives, adverbs, and even verbs. For instance while the endings -*est* and -*eþ* are almost always verb endings, the endings -*a* and -*e* can indicate a noun, verb or adjective, and -*um* can be either a noun or an adjective (and again, spelling irregularity makes them still harder to disambiguate). The use of an annotated corpus (one marked up with information such as part of speech of the words) is another possibility, or barring the existence of such a corpus, the production of one using an automatic tagger. The texts in the corpus itself could be encoded in such a way that the program could distinguish parts of speech of the variants of *god* and all surrounding words, and use this information to determine the likely meaning of *god* in context (although using the contextual information around the word itself could be problematic since word order was not as set in Old English as it is in Modern English). There are rule based part of speech tagging programs in existence currently for Modern English, but not Old English, unfortunately. Stringent tagging based on close reading and interpretation would likely be far more accurate, but very time consuming. As it stands, the current oe_lemma.pl program will generate lemma lists that conflate instances of the two possible meanings and parts of speech of *god*, and it will take human reading of each instance to determine where each actually belongs. Addressing semantic issues such as these is one good candidate for future study.

And as mentioned before, like the stemmers, this lemmatizer does not work for highly irregular forms, such as *beon* (the word "to be") and *seo* (the demonstrative pronoun “the”). As one solution, known irregular verbs in Old English could be entered manually into the lemma

list, though spelling irregularity would make this method somewhat ineffective at catching every instance. The irregulars could be hard coded into the software so that the variants could be compared and, if adequate matches were found, grouped with their more dissimilar forms.

CHAPTER 3

CONCLUSION

This project was undertaken in the hopes of achieving some success in Old English lemmatization in the face of a vast number of spelling variants and irregularities. It met with some success, but also many limitations, as none of the processes I employed produce correct results 100% of the time. Language rules, especially in relation to languages with as much variation as Old English, do not amount to an exact science, unfortunately. In many cases there is no way a program can know the difference between a correct and incorrect match (especially where semantic differences of very similar or related words are concerned), and whatever improvements are made on the processes in the future, human intervention will likely still be required. The aim will be to reduce further and further the amount of manual labor required to make the lemma lists correct.

The study has shown that the Levenshtein algorithm used in conjunction with sorting and other methods can be useful in trying to lemmatize the words found in Old English texts in their many forms. The common current lemmatization method of stemming alone does not work for Old English with all its variant spellings, but using Levenshtein alone does not work, either. Stemming by removal of common endings brings many of the equivalent words closer to matching, and conversion of some of the characters nearer still, so that the Levenshtein algorithm can be used to get more accurate matches.

The code could also be modified to include more translations, and more complex rules involving the removal of endings can be implemented using a dictionary or a more heavily

marked-up group of texts or corpus (although an annotated corpus is not available for Old English at the moment). Some manual intervention with the data will be required in any case, as it is with nearly any automated stemmer, lemmatizer, or tagger. Tzoukermann, Radev, and Gale in an article on automatically tagging French corpora speak of the need to leave some words in an "ambiguous" category, and that not even the human taggers always agree with one another (57). With the current program, the user has some leeway as to what type of intervention they would find it easier to perform, at least as far as whether it is easier to separate erroneously conflated words or group erroneously separated words, by such means as changing the possible endings and modifying the allowed minimum edit distance ratio until the more desired effects are produced.

The results of the current lemmatizer are not perfect and will require manual reading and intervention to correct them for import into text analysis software in order to provide the best accuracy level possible. But using these automated methods gets us closer to having a valid lemma list far more quickly than reading through the word list one word at a time and manually grouping the words, greatly speeding up the laborious task of lemma list creation. The work done on this lemmatizer would be a good jumping off point for development of a more robust lemmatizer for use with Old English.

WORKS CITED

"Algorithm Implementation/Strings/Levenshtein Distance." Wikibooks. Jan. 2009. 9 Feb. 2009 <http://en.wikibooks.org/wiki/Algorithm_implementation/Strings/Levenshtein_distance>.

Anthony, Lawrence. "Software." Lawrence Anthony's Homepage. 30 December 2008 <http://www.antlab.sci.waseda.ac.jp/software.html>.

Bosworth, Joseph. An Anglo-Saxon Dictionary. Ed. T. Northcote Toller. Oxford: Clarendon Press, 1898. 1 March 2009 < http://beowulf.engl.uky.edu/~kiernan/BT/Bosworth-Toller.htm>.

Burke, Sean M. "Sort-ArbBiLex-4.01." CPAN. March 2004. 1 Jan. 2009 <http://search.cpan.org/dist/Sort-ArbBiLex>.

CPAN. 14 Feb 2009 <http://www.cpan.org/>.

Evans, Jonathan. *Introductory Old English*. 2006.

Gilleland, Michael. Levenshtein Distance, in Three Flavors. 9 Feb. 2009 <http://www.merriampark.com/ld.htm>.

Hall, J.R. Clark. A Concise Anglo-Saxon Dictionary. Toronto: U of Toronto P, 1960.

Healey, Antonette diPaolo. "About." The Dictionary of Old English. University of Toronto. November 2008. 14 Feb. 2009 <http://www.doe.utoronto.ca/about.html>.

Kogai, Dan. “Encode-2.30.” CPAN. 29 Jan. 2009 <http://search.cpan.org/~dankogai/Encode-2.30/Encode.pm>.

The Lancaster Stemming Algorithm. Lancaster University. 8 Feb. 2009 <http://www.comp.lancs.ac.uk/computing/researche/stemming/>.

Levenshtein. 3 Feb. 2009 <http://www.levenshtein.net/>.

"The Lovins Stemming Algorithm." Snowball. 12 Feb. 2009 <http://snowball.tartarus.org/algorithms/lovins/stemmer.html>.

Mason, Oliver. Programming for Corpus Linguistics. Edinburgh: Edinburgh U P, 2000.

MorphAdorner. Northwestern University. Dec. 2007. 9 Feb. 2009 <http://morphadorner.northwestern.edu/morphadorner/>.

Oxford English Dictionary Online. Ed. John Simpson. Oxford University Press. 2009. Feb. 28 2009 <http://dictionary.oed.com>.

Porter, Martin. The Porter Stemming Algorithm. Jan. 2006. 7 Feb. 2009 <http://tartarus.org/~martin/PorterStemmer/>.

Tzoukermann, E., D. Radev, and W. Gale. Natural Language Processing Using Very Large Corpora. Ed. Susan Armstrong, Kenneth Church, Pierre Isabelle, Sandra Manzi, Evelyne Tzoukermann, and David Yarowsky. Dordrecht: Kluwer Academic Publishers, 1999.

APPENDIX A

Custom Scripts

A1. multiple_files.xsl. XSLT file run using Oxygen XML editor to output text files from Dictionary of Old English Corpus XML files.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:tei="http://www.tei-c.org/ns/1.0"
xmlns="http://www.w3.org/1999/xhtml"
    xmlns:xs="http://www.w3.org/2001/XMLSchema" exclude-result-prefixes="tei"
version="2.0">

      <xsl:output
            method="text"
            encoding="UTF-8"
            omit-xml-declaration = "yes" />

      <xsl:template match="TEI.2">
            <xsl:for-each select=".">
                  <xsl:variable name="filename"
select="concat(teiHeader/fileDesc/publicationStmt/idno, '_',
translate(teiHeader/fileDesc/titleStmt/title[@type='st'], '*/: ', '____'))" />
                  <xsl:result-document href="text_files3/{$filename}.txt">
                  <xsl:for-each select="text/body/p/s">
                        <xsl:apply-templates
/><xsl:text>&#x0D;</xsl:text><xsl:text>&#x0A;</xsl:text>
                  </xsl:for-each>
                  </xsl:result-document>
            </xsl:for-each>
      </xsl:template>

</xsl:stylesheet>
```

A2. oe_sort.pl. Perl program that sorts the word lists generated by AntConc into an alphabetical order better suited to Old English than most program's built-in ASCII alphabetical sort order.

```
#!/usr/bin/perl

# oe_sort.pl 02/02/2009 - Latest version of sort program using Sort::ArbBiLex
# CPAN module to perform a custom Old English sort.

use Sort::ArbBiLex (
    'oe_sort' =>
    "
     a A à À á Á â Â ã Ã ä Ä å Å æ Æ aa aæ æa ææ Aa Aæ Æa Ææ AA AÆ ÆA ÆÆ
     b B
     c C ç Ç k K
     d D
     e E è È é É ê Ê ë Ë ę Ę
     f F
     g G
     h H
     i I ì Ì í Í î Î ï Ï y Y ý Ý ÿ
     j J
     l L ł Ł
     m M
     n N
     o O ò Ò ó Ó ô Ô õ Õ ö Ö ø Ø œ Œ
     p P
     q Q
     r R
     s S
     t T
     þ Þ ð Ð þþ ðþ þð ðð Þþ Ðþ Þð Ðð ÞÞ ÐÞ ÞÐ ÐÐ th Th TH
     u U ù Ù ú Ú û Û ü Ü ų Ų
     v V
     w W
     x X
     z Z
    "
  );

open(MYFILE, "<wl_all_byword.txt") or die "Cannot open wl_all_byword.txt: $!";
open(MYWORD, ">wl_all_words_only.txt") or die "Cannot open
wl_all_words_only.txt: $!";
my(@lines) = <MYFILE>;

# Takes three field per line input file (with rank, frequency, and word
# and outputs word only.

foreach $line (@lines) {
        ($num, $freq, $word) = split '\t', $line;
        chomp($word);
      print MYWORD "$word\n";
}

close MYFILE;
close MYWORD;
```

```
open(MYWORD2, "<wl_all_words_only.txt") or die "Cannot open
wl_all_words_only.txt: $!";
open(MYOUT, ">wordlist_sorted.txt") or die "Cannot open wordlist_sorted.txt:
$!";
my(@words) = <MYWORD2>;

# Performs oe_sort defined at beginning of program on file of words and
# prints to output file.

@sorted = oe_sort(@words);

foreach (@sorted) {
      print MYOUT "$_"
}
```

A3. oe_lemma.pl. Perl program that takes wordlist_sorted.txt file from oe_sort.pl and endings.txt file as input and generates lemma file (wordlist_lemmas.txt), compounds file (wordlist_compounds.txt), and file of unmatched words (wordlist_unmatched.txt) as output.

```
#!/usr/bin/perl

# oe_lemma.pl 02/22/2009 (latest of several versions)
# Attempt to automate creation of lemma list with Levenshtein algorithm
# and various character translations.  Takes output of oe_sort.pl as input.

use utf8;
use Encode qw(encode decode);

sub trim($);

open(MYFILE, "<wordlist_sorted.txt") or die "Cannot open wordlist_sorted.txt:
$!";
open(ENDFILE, "<endings.txt") or die "Cannot open endings.txt: $!";
open(MYOUT, ">wordlist_lemmas.txt") or die "Cannot open wordlist_lemmas.txt:
$!";
open(MYUNMATCHED, ">wordlist_unmatched.txt") or die "Cannot open
wordlist_unmatched.txt: $!";
open(MYCOMPS, ">wordlist_compounds.txt") or die "Cannot open
wordlist_compounds.txt: $!";
my(@words) = <MYFILE>;
my(@ends) = <ENDFILE>;

# Initialize most variables.

$curr = '';
$currshort = '';
$currtrans = '';
$prev = 'fakeword';
$prevshort = 'fakew';
$prevtrans = '';
@wordlist = ();
@compounds = ();
@unmatched = ();
$currlist = '';
$lastchars = '';
$endflag = '';
$endflag2 = '';
$j = 0;

# Loops through each word in input file wordlist_sorted.txt.

foreach $word (@words)  {
    $word_trim = trim($word);
    chomp($word_trim);
    $word_decode = decode('UTF-8', $word_trim);
    $wordlen = length($word_decode);
    if ($wordlen > 2) {
      $endflag = 'n';
      $endflag2 = 'n';
      $curr = trim($word);
      chomp($curr);
```

```
      $curr = lc($curr);
      $curr = decode('UTF-8', $curr);
      $currlen = length($curr);
      $prevlen = length($prev);
      $currtrans = $curr;
      $currtrans =~ tr/kiæð/cyaþ/;
      $currtrans =~ tr/àáâãäåæ/a/;
      $currtrans =~ tr/èéêëę/e/;
      $currtrans =~ tr/ìíîï/i/;
      $currtrans =~ tr/òóôõöøœ/o/;
      $currtrans =~ tr/ùúûüų/u/;
      $currtrans =~ tr/ýÿ/y/;
      $currtrans =~ tr/ç/c/;
      $currtrans =~ s/(.)\1/\1/;

# Loops through each ending in endings.txt file and if present at end of
# word, removes ending, sets results to $currshort, and takes $currshort
# through several translations.  If no ending found, sets $currshort to
# to translated current word.

      foreach $end (@ends)  {
            $end = trim($end);
            chomp($end);
            my $end = decode('UTF-8', $end);
            $endlen = length($end);
            if ($currlen > $endlen)  {
                  $lastchars = substr($currtrans, ($currlen - $endlen),
$endlen);
                  if ($end eq $lastchars)  {
                        $rootlen = $currlen-$endlen;
                        $currshort = substr($currtrans, 0, $rootlen);
                        $endflag = 'y';
                  }
            }
      }
      if ($endflag eq 'n') {
            $currshort = $currtrans;
      }

# Levenshtein comparison section.  Finds lengths of the previous and current
# stem, runs stems through levenshtein function, and calculates a percentage
# difference based on the Levenshtein result divided by the length of the
# shortest of the two words.

      $pshortlen = length($prevshort);
      $cshortlen = length($currshort);
      $lev = levenshtein($prevshort, $currshort);
      if ($pshortlen > $cshortlen) {
            $diffpercent = $lev/$cshortlen;
      } else {
            $diffpercent = $lev/$pshortlen;
      }

# Levenshtein results used to determine if words are close enough to be
# grouped together in @wordlist array elements.  Words that match previous
# or shortened previous word followed by something other than a known
# ending are pushed to @compounds array if current short length is greater
```

```
# than previous short length plus 2. Words that do not appear to have a
# match are pushed to the @unmatched array.

      if ($diffpercent < 0.3 && $prevshort ne "fakew") {
            $currlist = $prev . ", " . $curr;
            if ($wordlist[$j] eq '') {
                  pop(@unmatched);
                  $wordlist[$j] = $currlist;
            }  else  {
                  $wordlist[$j] = $wordlist[$j] . ', ' . $curr;
            }
      }  elsif ($currshort =~ /^${prevshort}(.+)/ || $currtrans =~
/^${prevtrans}(.+)/) {
            foreach $end (@endings) {
                  $end = trim($end);
                  chomp($end);
                  $end = decode('UTF-8', $end);
                  if ($currtrans  =~ /^(${prevtrans})(${end})$/ || $currtrans
=~ /^(${prevshort})(${end})$/) {
                        $endflag2 = 'y';
                  }

            }
            if ($endflag2 eq 'n' && $cshortlen > $pshortlen + 2) {
                  push(@compounds, $curr);
                  $currshort = '';
                  $currtrans = '';
                  $curr = '';
                  next;
            } else {
                  push(@unmatched, $curr);
            }
      }  else  {
            push(@unmatched, $curr);
            $j++;
      }
      $prev = $curr;
      $prevshort = $currshort;
      $prevtrans = $currtrans;
      $currshort = '';
      $currtrans = '';
      $curr = '';
    } else {
      push(@unmatched, $word_decode);
    }
}

# Loops throuh arrays to output compounds, unmatched and lemma files.

foreach $compound (@compounds) {
      $compound = encode('UTF-8', $compound);
      print MYCOMPS "$compound\n";
}

foreach $unmatch (@unmatched) {
      $unmatch = encode('UTF-8', $unmatch);
      print MYUNMATCHED "$unmatch\n";
```

```
}

foreach $word (@wordlist) {
      $word = encode('UTF-8', $word);
      if ($word ne '') {
            print MYOUT "$word\n";
      }
}

close MYFILE;
close ENDFILE;
close MYOUT;
close MYUNMATCHED;
close MYCOMPS;

# Perl trim function to remove whitespace from the start and end of a string
sub trim($)
{
      my $string = shift;
      $string =~ s/^\s+//;
      $string =~ s/\s+$//;
      return $string;
}

# Levenshtein algorithm function.  Expects two strings as parameters.
# Usage: levenshtein( <string1>, <string2> )
#
# Algorithm adopted from here: http://www.merriampark.com/ldperl.htm
# Code below from:
# http://en.wikibooks.org/wiki/Algorithm_implementation/Strings/
# Levenshtein_distance
sub levenshtein
{
        my ( $a, $b ) = @_;
        my ( $len1, $len2 ) = ( length $a, length $b );

        return $len2 if ( $len1 == 0 );
        return $len1 if ( $len2 == 0 );

        my %d;

        for ( my $i = 0; $i <= $len1; ++$i )
        {
                for ( my $j = 0; $j <= $len2; ++$j )
                {
                        $d{ $i }{ $j } = 0;
                        $d{ 0 }{ $j } = $j;
                }

                $d{ $i }{ 0 } = $i;
        }

        # Populate arrays of characters to compare
        my @ar1 = split( //, $a );
        my @ar2 = split( //, $b );

        for ( my $i = 1; $i <= $len1; ++$i )
```

```
	{
		for ( my $j = 1; $j <= $len2; ++$j )
		{
			my $cost = ( $ar1[ $i - 1 ] eq $ar2[ $j - 1 ] ) ? 0 :
1;

			my $min1 = $d{ $i - 1 }{ $j } + 1;
			my $min2 = $d{ $i }{ $j - 1 } + 1;
			my $min3 = $d{ $i - 1 }{ $j - 1 } + $cost;

			if ( $min1 <= $min2 && $min1 <= $min3 )
			{
				$d{ $i }{ $j } = $min1;
			}
			elsif ( $min2 <= $min1 && $min2 <= $min3 )
			{
				$d{ $i }{ $j } = $min2;
			}
			else
			{
				$d{ $i }{ $j } = $min3;
			}
		}
	}

	return $d{ $len1 }{ $len2 };
}
```

APPENDIX B

Code from Other Sources

B1. Sort::ArbBiLex CPAN module source code. Module that allows definition of arbitrary sort orders which allows sorting of the word list in a more Old English friendly order. Downloaded from CPAN (from http://cpansearch.perl.org/src/SBURKE/Sort-ArbBiLex-4.01/ArbBiLex.pm on 02/12/2009) and installed in Perl.

```
# -*-Fundamental-*-
require 5;    # Time-stamp: "2004-03-27 17:19:11 AST"
package Sort::ArbBiLex;
use strict;
use vars qw(@ISA $Debug $VERSION);
$VERSION = "4.01";
$Debug = 0;
use Carp;
use integer; # vroom vroom

BEGIN { *UNICODE = eval('chr(256)') ? sub(){1} : sub(){0} }

#POD at end
###########################################################################

sub import {
  my $class_name = shift(@_);
  my $into = scalar caller;
  return unless @_;
  croak "Argument list in 'use $class_name' must be list of pairs" if @_ % 2;
  my($sym, $spec);
  while(@_) {
    ($sym, $spec) = splice(@_,0,2);
    defined $sym  or croak "Can't use undef as the name of a sub to make";
    length $sym   or croak "Can't use \"\" as the name of a sub to make";
    defined $spec or croak "Can't use undef as a sort-order spec";
    length $sym   or croak "Can't use \"\" as a sort-order spec";
    $sym = $into . '::' . $sym unless $sym =~ m/::/ or $sym =~ m/'/;
    no strict 'refs';
    *{$sym} = maker($spec);
  }
  return;
}

#--------------------------------------------------------------------------

sub maker {
  my $subr = eval(&source_maker(@_));
  die "Compile error <$@> in eval!?!" if $@; # shouldn't be possible!
  return $subr;
}
```

```
# Implementation note:  I didn't /need/ to use eval().  I could just return
#  an appropriate closure.  But one can't do tr/$foo/$bar/ -- eval is the
#  only way to get things to (so to speak) interpolate there; and the
#  efficiency cost of requiring that Perl parse more code is offset by
#  the efficiency benefit of being able to use tr/// (instead of s///) in
#  appropriate cases.

#---------------------------------------------------------------------------

sub source_maker {
  no locale;
  my($decl) = $_[0];
  croak "usage: Sort::ArbBiLex::maker(DECLARATION).  See the docs."
   unless @_ == 1;

  my $one_level_mode = 0;
  my @decl;
  if(ref $decl) { # It's a rLoL declaration
    croak "Sort order declaration must be a string or a listref"
      unless ref($decl) eq 'ARRAY';
    print "rLoL-decl mode\n" if $Debug > 1;
    # Make @decl into a list of families
    @decl = @$decl;
     # and each one of the items in @decl must be a ref to a list of scalars
    foreach my $f (@decl) {
      croak "Each family must be a listref" unless ref($f) eq 'ARRAY';
      @$f = grep(defined($_) && length($_), @$f); # sanity
      foreach my $g (@$f) { # more sanity.
        croak "A reference found where a glyph was expected" if ref($g);
      }
    }

  } else { # It's a string-style declaration
    print "string-decl mode\n" if $Debug > 1;
    # Make @decl into a list of families
    if($decl =~ /[\cm\cj\n]/) { # It contains majors and minors
      @decl = grep /\S/, split( /[\cm\cj]+/, $decl );
    } else { # It's all majors, on one line
      print "Strangeness trap 1.\n" if $Debug;
      @decl = grep /\S/, split( /\s+/, $decl );
      $one_level_mode = 1;
    }

    # Now turn @decl into a list of lists, where each element is a
    #  family -- i.e., a ref to a list of glyphs in that family.

    print "Glyph map:\n", map("  {<$_>}\n", @decl) if $Debug > 1;
    foreach my $d (@decl) { # in place changing
      #print "  d $d -> ", map("<$_> ",grep($_ ne '',split(/\s+/, $d))), "\n";
      $d = [ grep($_ ne '', split(/\s+/, $d)) ];
      #print "  d $d -> ", map("<$_> ", @$d), "\n";
    }
  }

  @decl = grep( scalar(@{$_}), @decl); # nix empty families
  croak "No glyphs in sort order declaration!?" unless @decl;
```

```
  @decl = map [$_], @{$decl[0]}  if  @decl == 1;
   # Change it from a family of N glyphs into N families of one glyph each

  # Iterate thru the families and their glyphs and build the tables
  my(@glyphs, @major_out, @minor_out);
  my $max_glyph_length = 0;
  my $max_family_length = 0;
  my %seen;
  my($glyph, $minor); # scratch
  for (my $major = 0; $major < @decl; $major++) {
    print "Family $major\n" if $Debug;
    croak "Too many major glyphs" if !UNICODE and $major > 255;
    $max_family_length = @{ $decl[$major] }
     if  @{ $decl[$major] }  >  $max_family_length;

    for ($minor = 0; $minor < @{ $decl[$major] }; $minor++) {
      $glyph = $decl[$major][$minor];
      print "  Glyph ($major)\:$minor (", $glyph, ")\n" if $Debug;
      croak "Glyph <$glyph> appears twice in the sort order declaration!"
       if $seen{$glyph}++;
      croak "Too many minor glyphs" if !UNICODE and $minor > 255;

      $max_glyph_length = length($glyph) if length($glyph) >
$max_glyph_length;

      $glyph =~ s/([^a-zA-Z0-9])/_char2esc($1)/eg;
      push @glyphs,    $glyph;
      push @major_out, _num2esc($major);
      push @minor_out, _num2esc($minor);
        #  or  unpack 'H2', pack 'C', 12   or   unpack 'H2', chr 12; ?
    }
  }
  die "Unexpected error: No glyphs?!?" if $max_glyph_length == 0;  # sanity
  $one_level_mode = 1 if $max_family_length == 1;

  ###########################################################################
  # Now start building the code.

  my($prelude, $coda, $code, $minor_code, $major_code);
  if($max_glyph_length == 1) {
    # All glyphs are single characters, so we can do this all with tr's
    $prelude = "# Single character mode.";
    $coda = '';
    my $glyphs    = join '', @glyphs;
    my $major_out = join '', @major_out;
    my $minor_out = join '', @minor_out;

    $minor_code = <<"EOMN"; # contents of a FOR block mapping $$x[0] => $$x[2]
             \$x->[2] =  \$x->[0];
             \$x->[2] =~ tr[$glyphs][]cd;
             \$x->[2] =~ tr[$glyphs]
                         [$minor_out];
EOMN

  $major_code = <<"EOMJ"; # expression returning a scalar as a major key
          do { # major keymaker
```

```
              my(\$key) = \$_;
              \$key =~ tr[$glyphs][]cd;
              \$key =~ tr[$glyphs]
                         [$major_out];
              scalar(\$key);
            }
EOMJ

    # End of single-glyph stuff.

  } else {
    # There are glyphs over 2 characters long -- gotta use s's.
    # End of multi-glyph stuff.
    my $glyphs    = join ',', map "\"$_\"", @glyphs;
    my $major_out = join ',', map "\"$_\"", @major_out;
    my $minor_out = join ',', map "\"$_\"", @minor_out;

    if(!$one_level_mode) {
      $prelude = <<"EOPRELUDE";
{ # Multi-character mode.  So we need a closure for these variables.
my(\%major, \%minor);
\@major{$glyphs}
 = ($major_out);
\@minor{$glyphs}
 = ($minor_out);
my \$glyph_re = join "|", map(quotemeta,
                              sort {length(\$b) <=> length(\$a)} keys \%major);
                              # put the longest glyphs first
EOPRELUDE
    } else { #  Multi-character mode
      $prelude = <<"EOPRELUDE2";
{ # Multi-character mode.  So we need a closure for these variables.
my(\%major); # just one-level mode, tho.
\@major{$glyphs}
 = ($major_out);
my \$glyph_re = join "|", map(quotemeta,
                              sort {length(\$b) <=> length(\$a)} keys \%major);
                              # put the longest glyphs first
EOPRELUDE2
}
    $coda = "} # end of closure.";

    $minor_code = <<"EOMN2"; # contents of a FOR block mapping $$x[0] =>
$$x[2]
                 \$x->[2] = join '',
                                map \$minor{\$_},
                                    \$x->[0] =~ m<(\$glyph_re)>go;
EOMN2

  $major_code = <<"EOMJ2"; # expression returning a scalar as a major key
            join('', map \$major{\$_}, m<(\$glyph_re)>go)  # major keymaker
EOMJ2

  }

  ###
  # Now  finish cobbling the code together.
```

```
  my $now = scalar(gmtime);

if(!$one_level_mode) {  # 2-level mode
  $code = <<"EOVOODOO";
\# Generated by Sort::ArbBiLex v$VERSION at $now GMT
$prelude
# Two-level mode
sub {   # change that to "sub whatever {" to name this function
  no locale; # we need the real 8-bit ASCIIbetical sort()
  use strict;
  return
    # map sort map is the Schwartzian Transform.  See perlfaq4.
    map { \$_->[0] }
    sort {
           \$a->[1] cmp \$b->[1] ||
           do {
             foreach my \$x (\$a, \$b) {
               if( !defined(\$x->[2]) and defined(\$x->[0]) ) {
$minor_code
               }
             }
             \$a->[2] cmp \$b->[2]; # return value of this do-block
           }
         }
    map { [ \$_,
$major_code
            , undef
          ]
        }
    \@_;
}
$coda

EOVOODOO

} else { # one-level mode

  $code = <<"EOVOODOO2";
\# Generated by Sort::ArbBiLex v$VERSION at $now GMT
$prelude
# One-level mode
sub {   # change that to "sub whatever {" to name this function
  no locale; # we need the real 8-bit ASCIIbetical sort()
  use strict;
  return
    # map sort map is the Schwartzian Transform.  See perlfaq4.
    map { \$_->[0] }
    sort { \$a->[1] cmp \$b->[1] }
    map { [ \$_,
$major_code
          ]
        }
    \@_;
}
$coda
```

```
EOVOODOO2

}

  print "\nCode to eval:\n", $code, "__ENDCODE__\n\n" if $Debug;

  return $code;
}

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

sub _char2esc {
  my $in = ord( $_[0] );
  return sprintf "\\x{%x}", $in if $in > 255;
  return sprintf "\\x%02x", $in;
}

sub _num2esc {
  my $in = $_[0];
  return sprintf "\\x{%x}", $in if $in > 255;
  return sprintf "\\x%02x", $in;
}

###########################################################################

# "cmp" returns -1, 0, or 1 depending on whether the left argument is
# stringwise less than, equal to, or greater than the right argument.

sub xcmp {
  carp "usage: xcmp(\\&sorter,$a,$b)" unless @_ and ref($_[0]);
  return  0 if $_[1] eq $_[2];  # We have to trap this early.
  return  1 if $_[1] ne ( $_[0]->($_[1], $_[2]) )[0];
   # If they were switched when sorted, then the original-first was
   #  lexically GT than the original-second.
  return -1 if $_[1] eq ( $_[0]->($_[2], $_[1]) )[0];
   # If they were switched BACK when REVERSED and sorted, then the
   #  original-first was lexically LT than the original-second.
  return  0;
   # Otherwise they were lexically identical.
}

# And two actually simpler ones:

sub xlt {
  carp "usage: xlt(\\&sorter,$a,$b)" unless @_ and ref($_[0]);
  #AKA: xcmp(@_) == -1;
  return  0 if $_[1] eq $_[2];  # We have to trap this early.
  return  1 if $_[1] eq ( $_[0]->($_[2], $_[1]) )[0];
   # If they were switched BACK when REVERSED and sorted, then the
   #  original-first was lexically LT than the original-second.
  return  0;
}

sub xgt {
  carp "usage: xgt(\\&sorter,$a,$b)" unless @_ and ref($_[0]);
  #AKA: xcmp(@_) == -1;
  return  0 if $_[1] eq $_[2];  # We have to trap this early.
```

```
  return  1 if $_[1] ne ( $_[0]->($_[1], $_[2]) )[0];
   # If they were switched when sorted, then the original-first was
   #  lexically GT than the original-second.
  return  0;
}

# And then two easy ones:

sub xle {
  carp "usage: xle(\\&sorter,$a,$b)" unless @_ and ref($_[0]);
  !xgt(@_);    #AKA: xcmp(@_) <   1;
}

sub xge {
  carp "usage: xge(\\&sorter,$a,$b)" unless @_ and ref($_[0]);
  !xlt(@_);    #AKA: xcmp(@_) >  -1;
}
```

B2. Levenshtein Algorithm. This is the code incorporated into oe_lemma.pl (see Appendix A1) as the levenshtein() function. Removed "use strict;" from program as it was causing issues. From http://en.wikibooks.org/wiki/Algorithm_implementation/Strings/Levenshtein_distance.

```
# The function expects two string parameters
# Usage: levenshtein( <string1>, <string2> )
#
# Algorithm adopted from here: http://www.merriampark.com/ldperl.htm

use strict;

sub levenshtein
{
      my ( $a, $b ) = @_;
      my ( $len1, $len2 ) = ( length $a, length $b );

      return $len2 if ( $len1 == 0 );
      return $len1 if ( $len2 == 0 );

      my %d;

      for ( my $i = 0; $i <= $len1; ++$i )
      {
            for ( my $j = 0; $j <= $len2; ++$j )
            {
                  $d{ $i }{ $j } = 0;
                  $d{ 0 }{ $j } = $j;
            }

            $d{ $i }{ 0 } = $i;
      }

      # Populate arrays of characters to compare
      my @ar1 = split( //, $a );
      my @ar2 = split( //, $b );

      for ( my $i = 1; $i <= $len1; ++$i )
      {
            for ( my $j = 1; $j <= $len2; ++$j )
            {
                  my $cost = ( $ar1[ $i - 1 ] eq $ar2[ $j - 1 ] ) ? 0 : 1;

                  my $min1 = $d{ $i - 1 }{ $j } + 1;
                  my $min2 = $d{ $i }{ $j - 1 } + 1;
                  my $min3 = $d{ $i - 1 }{ $j - 1 } + $cost;

                  if ( $min1 <= $min2 && $min1 <= $min3 )
                  {
                        $d{ $i }{ $j } = $min1;
                  }
                  elsif ( $min2 <= $min1 && $min2 <= $min3 )
                  {
                        $d{ $i }{ $j } = $min2;
                  }
                  else
                  {
```

```
                        $d{ $i }{ $j } = $min3;
                    }
                }
            }

        return $d{ $len1 }{ $len2 };
}
```

APPENDIX C

Porter Stemmer

C. Perl Implementation of Porter Stemmer by Martin Porter. From http://tartarus.org/~martin/PorterStemmer/perl.txt.

```
#!/usr/bin/perl -w

# Porter stemmer in Perl. Few comments, but it's easy to follow against the
rules in the original
# paper, in
#
#   Porter, 1980, An algorithm for suffix stripping, Program, Vol. 14,
#   no. 3, pp 130-137,
#
# see also http://www.tartarus.org/~martin/PorterStemmer

# Release 1

local %step2list;
local %step3list;
local ($c, $v, $C, $V, $mgr0, $meq1, $mgr1, $_v);


sub stem
{  my ($stem, $suffix, $firstch);
   my $w = shift;
   if (length($w) < 3) { return $w; } # length at least 3
   # now map initial y to Y so that the patterns never treat it as vowel:
   $w =~ /^./; $firstch = $&;
   if ($firstch =~ /^y/) { $w = ucfirst $w; }

   # Step 1a
   if ($w =~ /(ss|i)es$/) { $w=$`.$1; }
   elsif ($w =~ /([^s])s$/) { $w=$`.$1; }
   # Step 1b
   if ($w =~ /eed$/) { if ($` =~ /$mgr0/o) { chop($w); } }
   elsif ($w =~ /(ed|ing)$/)
   {  $stem = $`;
      if ($stem =~ /$_v/o)
      {  $w = $stem;
         if ($w =~ /(at|bl|iz)$/) { $w .= "e"; }
         elsif ($w =~ /([^aeiouylsz])\1$/) { chop($w); }
         elsif ($w =~ /^${C}${v}[^aeiouwxy]$/o) { $w .= "e"; }
      }
   }
   # Step 1c
   if ($w =~ /y$/) { $stem = $`; if ($stem =~ /$_v/o) { $w = $stem."i"; } }
```

```
   # Step 2
   if ($w =~ /(ational|tional|enci|anci|izer|bli|alli|entli|eli|ousli|
ization|ation|ator|alism|iveness|fulness|ousness|aliti|iviti|biliti|logi)$/)
   { $stem = $`; $suffix = $1;
     if ($stem =~ /$mgr0/o) { $w = $stem . $step2list{$suffix}; }
   }

   # Step 3

   if ($w =~ /(icate|ative|alize|iciti|ical|ful|ness)$/)
   { $stem = $`; $suffix = $1;
     if ($stem =~ /$mgr0/o) { $w = $stem . $step3list{$suffix}; }
   }

   # Step 4

   if ($w =~ /(al|ance|ence|er|ic|able|ible|ant|ement|ment|ent|ou|ism|ate|
iti|ous|ive|ize)$/)
   { $stem = $`; if ($stem =~ /$mgr1/o) { $w = $stem; } }
   elsif ($w =~ /(s|t)(ion)$/)
   { $stem = $` . $1; if ($stem =~ /$mgr1/o) { $w = $stem; } }


   #  Step 5

   if ($w =~ /e$/)
   { $stem = $`;
     if ($stem =~ /$mgr1/o or
         ($stem =~ /$meq1/o and not $stem =~ /^${C}${v}[^aeiouwxy]$/o))
       { $w = $stem; }
   }
   if ($w =~ /ll$/ and $w =~ /$mgr1/o) { chop($w); }

   # and turn initial Y back to y
   if ($firstch =~ /^y/) { $w = lcfirst $w; }
   return $w;
}

sub initialise {

   %step2list =
   ( 'ational'=>'ate', 'tional'=>'tion', 'enci'=>'ence', 'anci'=>'ance',
'izer'=>'ize', 'bli'=>'ble',
     'alli'=>'al', 'entli'=>'ent', 'eli'=>'e', 'ousli'=>'ous',
'ization'=>'ize', 'ation'=>'ate',
     'ator'=>'ate', 'alism'=>'al', 'iveness'=>'ive', 'fulness'=>'ful',
'ousness'=>'ous', 'aliti'=>'al',
     'iviti'=>'ive', 'biliti'=>'ble', 'logi'=>'log');

   %step3list =
   ('icate'=>'ic', 'ative'=>'', 'alize'=>'al', 'iciti'=>'ic', 'ical'=>'ic',
'ful'=>'', 'ness'=>'');


   $c =    "[^aeiou]";          # consonant
   $v =    "[aeiouy]";          # vowel
   $C =    "${c}[^aeiouy]*";    # consonant sequence
```

```
   $V =    "${v}[aeiou]*";      # vowel sequence

   $mgr0 = "^(${C})?${V}${C}";                # [C]VC... is m>0
   $meq1 = "^(${C})?${V}${C}(${V})?" . '$';  # [C]VC[V] is m=1
   $mgr1 = "^(${C})?${V}${C}${V}${C}";       # [C]VCVC... is m>1
   $_v   = "^(${C})?${v}";                    # vowel in stem

}

# that's the definition. Run initialise() to set things up, then stem($word)
to stem $word, as here:


initialise();

while (<>)
{
   {  /^([^a-zA-Z]*)(.*)/ ;
      print $1;
      $_ = $2;
      unless ( /^([a-zA-Z]+)(.*)/ ) { last; }
      $word = lc $1; # turn to lower case before calling:
      $_ = $2;
      $word = stem($word);
      print $word;
      redo;
   }
   print "\n";
}

# inputs taken from the files on the arg list, output to stdout.

# As an easy speed-up, one might create a hash of word=>stemmed form, and
look up each new
# word in the hash, only calling stem() if the word was not found there.
```

APPENDIX D

Unicode Character Set

D. UTF-8 Encoding Table and Unicode Characters. From http://www.utf8-chartable.de/unicode-utf8-table.pl?utf8=dec&view=3 on 02/12/2009.

Unicode code point	character	UTF-8 (dec.)	name
U+0000		0	<control>
U+0001		1	<control>
U+0002		2	<control>
U+0003		3	<control>
U+0004		4	<control>
U+0005		5	<control>
U+0006		6	<control>
U+0007		7	<control>
U+0008		8	<control>
U+0009		9	<control>
U+000A		10	<control>
U+000B		11	<control>
U+000C		12	<control>
U+000D		13	<control>
U+000E		14	<control>
U+000F		15	<control>
U+0010		16	<control>
U+0011		17	<control>
U+0012		18	<control>
U+0013		19	<control>
U+0014		20	<control>
U+0015		21	<control>
U+0016		22	<control>
U+0017		23	<control>
U+0018		24	<control>
U+0019		25	<control>
U+001A		26	<control>
U+001B		27	<control>
U+001C		28	<control>
U+001D		29	<control>
U+001E		30	<control>
U+001F		31	<control>

U+0020		32	SPACE
U+0021	!	33	EXCLAMATION MARK
U+0022	"	34	QUOTATION MARK
U+0023	#	35	NUMBER SIGN
U+0024	$	36	DOLLAR SIGN
U+0025	%	37	PERCENT SIGN
U+0026	&	38	AMPERSAND
U+0027	'	39	APOSTROPHE
U+0028	(	40	LEFT PARENTHESIS
U+0029	)	41	RIGHT PARENTHESIS
U+002A	*	42	ASTERISK
U+002B	+	43	PLUS SIGN
U+002C	,	44	COMMA
U+002D	-	45	HYPHEN-MINUS
U+002E	.	46	FULL STOP
U+002F	/	47	SOLIDUS
U+0030	0	48	DIGIT ZERO
U+0031	1	49	DIGIT ONE
U+0032	2	50	DIGIT TWO
U+0033	3	51	DIGIT THREE
U+0034	4	52	DIGIT FOUR
U+0035	5	53	DIGIT FIVE
U+0036	6	54	DIGIT SIX
U+0037	7	55	DIGIT SEVEN
U+0038	8	56	DIGIT EIGHT
U+0039	9	57	DIGIT NINE
U+003A	:	58	COLON
U+003B	;	59	SEMICOLON
U+003C	<	60	LESS-THAN SIGN
U+003D	=	61	EQUALS SIGN
U+003E	>	62	GREATER-THAN SIGN
U+003F	?	63	QUESTION MARK
U+0040	@	64	COMMERCIAL AT
U+0041	A	65	LATIN CAPITAL LETTER A
U+0042	B	66	LATIN CAPITAL LETTER B
U+0043	C	67	LATIN CAPITAL LETTER C
U+0044	D	68	LATIN CAPITAL LETTER D
U+0045	E	69	LATIN CAPITAL LETTER E
U+0046	F	70	LATIN CAPITAL LETTER F
U+0047	G	71	LATIN CAPITAL LETTER G
U+0048	H	72	LATIN CAPITAL LETTER H
U+0049	I	73	LATIN CAPITAL LETTER I
U+004A	J	74	LATIN CAPITAL LETTER J
U+004B	K	75	LATIN CAPITAL LETTER K
U+004C	L	76	LATIN CAPITAL LETTER L

U+004D	M	77	LATIN CAPITAL LETTER M
U+004E	N	78	LATIN CAPITAL LETTER N
U+004F	O	79	LATIN CAPITAL LETTER O
U+0050	P	80	LATIN CAPITAL LETTER P
U+0051	Q	81	LATIN CAPITAL LETTER Q
U+0052	R	82	LATIN CAPITAL LETTER R
U+0053	S	83	LATIN CAPITAL LETTER S
U+0054	T	84	LATIN CAPITAL LETTER T
U+0055	U	85	LATIN CAPITAL LETTER U
U+0056	V	86	LATIN CAPITAL LETTER V
U+0057	W	87	LATIN CAPITAL LETTER W
U+0058	X	88	LATIN CAPITAL LETTER X
U+0059	Y	89	LATIN CAPITAL LETTER Y
U+005A	Z	90	LATIN CAPITAL LETTER Z
U+005B	[	91	LEFT SQUARE BRACKET
U+005C	\	92	REVERSE SOLIDUS
U+005D	]	93	RIGHT SQUARE BRACKET
U+005E	^	94	CIRCUMFLEX ACCENT
U+005F	_	95	LOW LINE
U+0060	`	96	GRAVE ACCENT
U+0061	a	97	LATIN SMALL LETTER A
U+0062	b	98	LATIN SMALL LETTER B
U+0063	c	99	LATIN SMALL LETTER C
U+0064	d	100	LATIN SMALL LETTER D
U+0065	e	101	LATIN SMALL LETTER E
U+0066	f	102	LATIN SMALL LETTER F
U+0067	g	103	LATIN SMALL LETTER G
U+0068	h	104	LATIN SMALL LETTER H
U+0069	i	105	LATIN SMALL LETTER I
U+006A	j	106	LATIN SMALL LETTER J
U+006B	k	107	LATIN SMALL LETTER K
U+006C	l	108	LATIN SMALL LETTER L
U+006D	m	109	LATIN SMALL LETTER M
U+006E	n	110	LATIN SMALL LETTER N
U+006F	o	111	LATIN SMALL LETTER O
U+0070	p	112	LATIN SMALL LETTER P
U+0071	q	113	LATIN SMALL LETTER Q
U+0072	r	114	LATIN SMALL LETTER R
U+0073	s	115	LATIN SMALL LETTER S
U+0074	t	116	LATIN SMALL LETTER T
U+0075	u	117	LATIN SMALL LETTER U
U+0076	v	118	LATIN SMALL LETTER V
U+0077	w	119	LATIN SMALL LETTER W
U+0078	x	120	LATIN SMALL LETTER X
U+0079	y	121	LATIN SMALL LETTER Y

<table>
<tr><td>U+007A</td><td>z</td><td>122</td><td>LATIN SMALL LETTER Z</td></tr>
<tr><td>U+007B</td><td>{</td><td>123</td><td>LEFT CURLY BRACKET</td></tr>
<tr><td>U+007C</td><td>|</td><td>124</td><td>VERTICAL LINE</td></tr>
<tr><td>U+007D</td><td>}</td><td>125</td><td>RIGHT CURLY BRACKET</td></tr>
<tr><td>U+007E</td><td>~</td><td>126</td><td>TILDE</td></tr>
<tr><td>U+007F</td><td></td><td>127</td><td><control></td></tr>
<tr><td>U+0080</td><td></td><td>194 128</td><td><control></td></tr>
<tr><td>U+0081</td><td></td><td>194 129</td><td><control></td></tr>
<tr><td>U+0082</td><td></td><td>194 130</td><td><control></td></tr>
<tr><td>U+0083</td><td></td><td>194 131</td><td><control></td></tr>
<tr><td>U+0084</td><td></td><td>194 132</td><td><control></td></tr>
<tr><td>U+0085</td><td></td><td>194 133</td><td><control></td></tr>
<tr><td>U+0086</td><td></td><td>194 134</td><td><control></td></tr>
<tr><td>U+0087</td><td></td><td>194 135</td><td><control></td></tr>
<tr><td>U+0088</td><td></td><td>194 136</td><td><control></td></tr>
<tr><td>U+0089</td><td></td><td>194 137</td><td><control></td></tr>
<tr><td>U+008A</td><td></td><td>194 138</td><td><control></td></tr>
<tr><td>U+008B</td><td></td><td>194 139</td><td><control></td></tr>
<tr><td>U+008C</td><td></td><td>194 140</td><td><control></td></tr>
<tr><td>U+008D</td><td></td><td>194 141</td><td><control></td></tr>
<tr><td>U+008E</td><td></td><td>194 142</td><td><control></td></tr>
<tr><td>U+008F</td><td></td><td>194 143</td><td><control></td></tr>
<tr><td>U+0090</td><td></td><td>194 144</td><td><control></td></tr>
<tr><td>U+0091</td><td></td><td>194 145</td><td><control></td></tr>
<tr><td>U+0092</td><td></td><td>194 146</td><td><control></td></tr>
<tr><td>U+0093</td><td></td><td>194 147</td><td><control></td></tr>
<tr><td>U+0094</td><td></td><td>194 148</td><td><control></td></tr>
<tr><td>U+0095</td><td></td><td>194 149</td><td><control></td></tr>
<tr><td>U+0096</td><td></td><td>194 150</td><td><control></td></tr>
<tr><td>U+0097</td><td></td><td>194 151</td><td><control></td></tr>
<tr><td>U+0098</td><td></td><td>194 152</td><td><control></td></tr>
<tr><td>U+0099</td><td></td><td>194 153</td><td><control></td></tr>
<tr><td>U+009A</td><td></td><td>194 154</td><td><control></td></tr>
<tr><td>U+009B</td><td></td><td>194 155</td><td><control></td></tr>
<tr><td>U+009C</td><td></td><td>194 156</td><td><control></td></tr>
<tr><td>U+009D</td><td></td><td>194 157</td><td><control></td></tr>
<tr><td>U+009E</td><td></td><td>194 158</td><td><control></td></tr>
<tr><td>U+009F</td><td></td><td>194 159</td><td><control></td></tr>
<tr><td>U+00A0</td><td></td><td>194 160</td><td>NO-BREAK SPACE</td></tr>
<tr><td>U+00A1</td><td>¡</td><td>194 161</td><td>INVERTED EXCLAMATION MARK</td></tr>
<tr><td>U+00A2</td><td>¢</td><td>194 162</td><td>CENT SIGN</td></tr>
<tr><td>U+00A3</td><td>£</td><td>194 163</td><td>POUND SIGN</td></tr>
<tr><td>U+00A4</td><td>¤</td><td>194 164</td><td>CURRENCY SIGN</td></tr>
<tr><td>U+00A5</td><td>¥</td><td>194 165</td><td>YEN SIGN</td></tr>
<tr><td>U+00A6</td><td>¦</td><td>194 166</td><td>BROKEN BAR</td></tr>
</table>

U+00A7	§	194 167	SECTION SIGN
U+00A8	¨	194 168	DIAERESIS
U+00A9	©	194 169	COPYRIGHT SIGN
U+00AA	ª	194 170	FEMININE ORDINAL INDICATOR
U+00AB	«	194 171	LEFT-POINTING DOUBLE ANGLE QUOTATION MARK
U+00AC	¬	194 172	NOT SIGN
U+00AD		194 173	SOFT HYPHEN
U+00AE	®	194 174	REGISTERED SIGN
U+00AF	¯	194 175	MACRON
U+00B0	°	194 176	DEGREE SIGN
U+00B1	±	194 177	PLUS-MINUS SIGN
U+00B2	²	194 178	SUPERSCRIPT TWO
U+00B3	³	194 179	SUPERSCRIPT THREE
U+00B4	´	194 180	ACUTE ACCENT
U+00B5	µ	194 181	MICRO SIGN
U+00B6	¶	194 182	PILCROW SIGN
U+00B7	·	194 183	MIDDLE DOT
U+00B8	¸	194 184	CEDILLA
U+00B9	¹	194 185	SUPERSCRIPT ONE
U+00BA	º	194 186	MASCULINE ORDINAL INDICATOR
U+00BB	»	194 187	RIGHT-POINTING DOUBLE ANGLE QUOTATION MARK
U+00BC	¼	194 188	VULGAR FRACTION ONE QUARTER
U+00BD	½	194 189	VULGAR FRACTION ONE HALF
U+00BE	¾	194 190	VULGAR FRACTION THREE QUARTERS
U+00BF	¿	194 191	INVERTED QUESTION MARK
U+00C0	À	195 128	LATIN CAPITAL LETTER A WITH GRAVE
U+00C1	Á	195 129	LATIN CAPITAL LETTER A WITH ACUTE
U+00C2	Â	195 130	LATIN CAPITAL LETTER A WITH CIRCUMFLEX
U+00C3	Ã	195 131	LATIN CAPITAL LETTER A WITH TILDE
U+00C4	Ä	195 132	LATIN CAPITAL LETTER A WITH DIAERESIS
U+00C5	Å	195 133	LATIN CAPITAL LETTER A WITH RING ABOVE
U+00C6	Æ	195 134	LATIN CAPITAL LETTER AE
U+00C7	Ç	195 135	LATIN CAPITAL LETTER C WITH CEDILLA
U+00C8	È	195 136	LATIN CAPITAL LETTER E WITH GRAVE
U+00C9	É	195 137	LATIN CAPITAL LETTER E WITH ACUTE
U+00CA	Ê	195 138	LATIN CAPITAL LETTER E WITH CIRCUMFLEX
U+00CB	Ë	195 139	LATIN CAPITAL LETTER E WITH DIAERESIS
U+00CC	Ì	195 140	LATIN CAPITAL LETTER I WITH GRAVE
U+00CD	Í	195 141	LATIN CAPITAL LETTER I WITH ACUTE
U+00CE	Î	195 142	LATIN CAPITAL LETTER I WITH CIRCUMFLEX
U+00CF	Ï	195 143	LATIN CAPITAL LETTER I WITH DIAERESIS
U+00D0	Ð	195 144	LATIN CAPITAL LETTER ETH
U+00D1	Ñ	195 145	LATIN CAPITAL LETTER N WITH TILDE
U+00D2	Ò	195 146	LATIN CAPITAL LETTER O WITH GRAVE
U+00D3	Ó	195 147	LATIN CAPITAL LETTER O WITH ACUTE

U+00D4	Ô	195 148	LATIN CAPITAL LETTER O WITH CIRCUMFLEX
U+00D5	Õ	195 149	LATIN CAPITAL LETTER O WITH TILDE
U+00D6	Ö	195 150	LATIN CAPITAL LETTER O WITH DIAERESIS
U+00D7	×	195 151	MULTIPLICATION SIGN
U+00D8	Ø	195 152	LATIN CAPITAL LETTER O WITH STROKE
U+00D9	Ù	195 153	LATIN CAPITAL LETTER U WITH GRAVE
U+00DA	Ú	195 154	LATIN CAPITAL LETTER U WITH ACUTE
U+00DB	Û	195 155	LATIN CAPITAL LETTER U WITH CIRCUMFLEX
U+00DC	Ü	195 156	LATIN CAPITAL LETTER U WITH DIAERESIS
U+00DD	Ý	195 157	LATIN CAPITAL LETTER Y WITH ACUTE
U+00DE	Þ	195 158	LATIN CAPITAL LETTER THORN
U+00DF	ß	195 159	LATIN SMALL LETTER SHARP S
U+00E0	à	195 160	LATIN SMALL LETTER A WITH GRAVE
U+00E1	á	195 161	LATIN SMALL LETTER A WITH ACUTE
U+00E2	â	195 162	LATIN SMALL LETTER A WITH CIRCUMFLEX
U+00E3	ã	195 163	LATIN SMALL LETTER A WITH TILDE
U+00E4	ä	195 164	LATIN SMALL LETTER A WITH DIAERESIS
U+00E5	å	195 165	LATIN SMALL LETTER A WITH RING ABOVE
U+00E6	æ	195 166	LATIN SMALL LETTER AE
U+00E7	ç	195 167	LATIN SMALL LETTER C WITH CEDILLA
U+00E8	è	195 168	LATIN SMALL LETTER E WITH GRAVE
U+00E9	é	195 169	LATIN SMALL LETTER E WITH ACUTE
U+00EA	ê	195 170	LATIN SMALL LETTER E WITH CIRCUMFLEX
U+00EB	ë	195 171	LATIN SMALL LETTER E WITH DIAERESIS
U+00EC	ì	195 172	LATIN SMALL LETTER I WITH GRAVE
U+00ED	í	195 173	LATIN SMALL LETTER I WITH ACUTE
U+00EE	î	195 174	LATIN SMALL LETTER I WITH CIRCUMFLEX
U+00EF	ï	195 175	LATIN SMALL LETTER I WITH DIAERESIS
U+00F0	ð	195 176	LATIN SMALL LETTER ETH
U+00F1	ñ	195 177	LATIN SMALL LETTER N WITH TILDE
U+00F2	ò	195 178	LATIN SMALL LETTER O WITH GRAVE
U+00F3	ó	195 179	LATIN SMALL LETTER O WITH ACUTE
U+00F4	ô	195 180	LATIN SMALL LETTER O WITH CIRCUMFLEX
U+00F5	õ	195 181	LATIN SMALL LETTER O WITH TILDE
U+00F6	ö	195 182	LATIN SMALL LETTER O WITH DIAERESIS
U+00F7	÷	195 183	DIVISION SIGN
U+00F8	ø	195 184	LATIN SMALL LETTER O WITH STROKE
U+00F9	ù	195 185	LATIN SMALL LETTER U WITH GRAVE
U+00FA	ú	195 186	LATIN SMALL LETTER U WITH ACUTE
U+00FB	û	195 187	LATIN SMALL LETTER U WITH CIRCUMFLEX
U+00FC	ü	195 188	LATIN SMALL LETTER U WITH DIAERESIS
U+00FD	ý	195 189	LATIN SMALL LETTER Y WITH ACUTE
U+00FE	þ	195 190	LATIN SMALL LETTER THORN
U+00FF	ÿ	195 191	LATIN SMALL LETTER Y WITH DIAERESIS