MorphAdorner

A Java Library for the Morphological Adornment of English Language Texts

Version 1.0. April 30, 2009.

Table of Contents

Part One: Introduction

Poets that lasting marble seek,
Must carve in Latin or in Greek.
We write in sand, our language grows,
And like the tide, our work o'erflows.

-- Edmund Waller

Introduction to MorphAdorner

MorphAdorner is a Java command-line program which acts as a pipeline manager for processes performing morphological adornment of words in a text. We use the term "adornment" in preference to terms such as "annotation" or "tagging" which carry too many alternative and confusing meanings. Adornment harkens back to the medieval sense of manuscript adornment or illumination -- attaching pictures and marginal comments to texts.

Currently MorphAdorner provides methods for adorning text with standard spellings, parts of speech and lemmata. MorphAdorner also provides facilities for tokenizing text, recognizing sentence boundaries, and extracting names and places. You can find out more about each of these facilities, and see online demontrations of each, by choosing an item from the menu to the left.

MorphAdorner has undergone continuous development in tandem with three projects: WordHoard, Monk, and Virtual Orthographic Standardization and Part of Speech Tagging (VOSPOS), as well as smaller scale faculty research projects at Northwestern University. WordHoard and VOSPOS are complete. The Monk project ended in April 2009. While MorphAdorner has been used in these projects, it is actually a separate project in its own right.

MorphAdorner has seen its heaviest use in the Monk project. The Monk project seeks to adorn a large number of English language texts from the early Modern English period to the start of the twentieth century. We expect the total number of adorned words to reach a couple of hundred million by project end.

Our efforts to adorn English texts covering a period of over four hundred years must deal with the fact that the English language has changed significantly even since the start of the early modern period around 1470 A.D. The great vowel sound shift was only about half complete at this time. Spelling was not at all standardized. Early printed texts reflect the differences in pronunciation. In 1475 William Caxton published (in Bruges) the first book printed in English, **Recuyell of the Historyes of Troye**. That short title reveals the orthographic variety that persisted until the late eighteenth century.

Today there remain differences among British, American, and Canadian spellings. Unlike the early modern period, these differences are reasonably regular and generally easy to handle.

How MorphAdorner Works

MorphAdorner drives a text through the following stages or "pipes."

- Input
- Sentence Splitting
- Tokenization

- Spelling Standardization
- Part of Speech Tagging
- Lemmatization
- Output

Each of these stages is defined in terms of Java interfaces. Each interface has an associated factory class which MorphAdorner uses to instantiate particular implementations of the interface under control of a configuration file. This allows easy substitution of different implementations into the pipeline by changing the configuration file. A programmer can create new custom implementations of any interface and tell MorphAdorner to use the custom implementation in the configuration file. Each pipe can also be used independently of MorphAdorner.

How Do I ...

Download and install MorphAdorner?

- See Downloading and Installing MorphAdorner (page 4).

Adorn a text file with parts of speech, lemmata, and standard spellings?

- See Part Two:Adorning A Text (page 17).

Create an embedded adorner in a Java program?

- See Example One: Adorning a string With Parts Of Speech (page 114).

Find out more about the NUPOS part of speech tag set?

- See NUPOS and Morphology (page 76).

Find definitions of technical terms used this document?

- See Appendix Two: Glossary of Natural Language Processing Terms (page 156).

Know if I can use the MorphAdorner code in my own custom program?

- See the MorphAdorner License (page 12).

Get help when I have problems with MorphAdorner?

- See MorphAdorner Support (page 16).

Deal with Java "OutOfMemory" errors?

- See Java OutOfMemory Errors (page 19).

Downloading and Installing MorphAdorner

The file

morphadorner-2009-04-30.zip

contains the MorphAdorner source code, data, and libraries.

Current version: 1.0
Last update: April 30, 2009

File Layout of Morphadorner Release

File or Directory	Contents
aareadme1st.txt	Printable copy of this file in Windows text format (lines terminated by Ascii cr/lf).
bin/	Binaries for MorphAdorner.
build.xml	Apache Ant build file used to compile MorphAdorner.
data/	Data files used by MorphAdorner.
documentation/	MorphAdorner documentation.
gatelib/	Java libraries used by Gate.
javadoc/	Javadoc (internal documentation).
jetty/	Web server for running sample servlets.
lib/	Java library files.
misc/	Miscellaneous files for compiling MorphAdorner.
src/	MorphAdorner source code.

Installing and Building MorphAdorner

Extract the files from morphadorner-2009-04-30.zip, retaining the directory structure, to an empty directory. The zip file contains precompiled (with Java 1.6) versions of all of the code as well as the javadoc. You do not need to rebuild the code unless you want to make changes. If you do want to rebuild the code, make sure you have installed recent working copies of Sun's Java Development Kit and Apache Ant on your system. Move to the directory in which you extracted morphadorner-2009-04-30.zip, and type:

```
ant
```

This should build MorphAdorner successfully.

Type

```
ant doc
```

to generate the javadoc (internal documentation).

Type

```
ant clean
```

to remove the effects of compilation.

Documentation

Printable documentation, in Adobe Acrobat PDF format, appears in the documentation/morphadorner.pdf file in the MorphAdorner release. MorphAdorner documentation is also available online. The online version will generally be more up-to-date than the printable version included in the release materials. The javadoc (internal documentation) is also available online as well as in the release materials in the javadoc/ directory. The online MorphAdorner modification history describes what has changed from one release of MorphAdorner to the next.

Running MorphAdorner

MorphAdorner has run successfully on Windows, Mac OS X, and various flavors of Linux.

The sample batch file **adornncf.bat** and the corresponding Linux script **adornncf** shows how to run MorphAdorner to adorn TEI Analytics format XML files for 19th century and later works in which quote marks are not distinguished from apostrophes. Use the sample batch file **adornncfa.bat** and the script **adornncfa** for works in which quote marks are distinguished from apostrophes. Use the batch file **adornwright.bat** and script **adornwright** for Wright archive texts. The sample batch file **adorneeme.bat** and the corresponding Linux script **adorneme** shows how to run MorphAdorner to adorn TEI Analytics versions of XML files generated from the early modern English eebo/tcp collection. Please see Part Two:Adorning A Text (page 17) for more information for more information on these and other sample batch files and scripts in the MorphAdorner release.

Don't forget to mark the Unix script files as executable before using them. On most Unix/Linux systems you can use the **chmod** command to do this, e.g.:

```
chmod 755 adornncfa
```

The MorphAdorner release contains a script **makescriptsexecutable** which applies **chmod** to each of the scripts in the release. On most Unix-like systems you can execute **makescriptsexecutable** by moving to the MorphAdorner installation directory and entering

```
chmod 755 makescriptsexecutable
./makescriptsexecutable
```

or

```
/bin/sh <makescriptsexecutable
```

There are presumably lots of warts, misfeatures, bugs, missing items, and whatnot. Use MorphAdorner with caution.

Modification History

This is release 1.0 of MorphAdorner.

Main changes since v0.9.1.

(1) The list of languages recognized by the default language recognizer has been changed to:

dutch, english, french, german, italian, latin, scots, spanish, welsh

The base language recognizer has been modified to allow for setting the list of recognized languages.

(2) Minimal documentation on setting up the sample MorphAdorner servlets has been added to the online and printable documentation. The servlets are intended mainly for demonstration purposes, not production use.

(3) LGParser had problems under Unix/Linux. Those problems should be corrected.

Main changes since v0.9.

(1) The Unicode non-breaking hyphen (\u2011) is no longer treated as a separate token.

Main changes since v0.8.2.

(1) Added a facility to handle selected split words. The initial implementation handles reflexive pronouns in Early Modern English texts, e.g., "them selves" for "themselves".

(2) Added a facility to remove adornments from children of specified tags (e.g., figdesc) when writing the final adorned text.

(3) Corrected a lemmatization problem in which some words lemmatized to an empty string. The spelling is returned for such cases.

(4) Added "UnicodeReader" class to allow correct reading of files with initial BOM markers. The standard Java I/O libraries does not handle BOM markers correctly when a Reader is used.

(5) Periods before the file extension in input xml file names are now mapped to underlines when generating word IDs.

(6) Removed some GPL licensed code.

(7) Removed the following portions of GPL licensed code:
-- Lovins stemmer
-- gnu.getopt (replaced with jargs)

(8) Made MergeEnhancedBrillLexicon and CreateSuffixLexicon utilities public.

(9) Ensure selected xml tags (e.g., figdesc, sic) do not contain <w> and <c> children.

(10) Ensure vertical bars are treated as symbols when they appear in source text.

Main changes since v0.8.1.

(1) A simple XML outputter for adorned words is available as SimpleXMLAdornedWordOutputter.

(2) Added a simple EEBO name standardizer to regularize a few names.

(3) Changed the lemmatization of possessive pronouns to map to themselves, e.g., my --> my, his --> his, etc.

(4) Corrected a bug in the part of speech for words cache that caused some lexicon tags to be ignored for cached words.

(5) Combining macrons with invalid leading spaces are now joined to the preceding word in Early Modern English texts.

Main changes since v0.8.

(1) URLs are now accepted for input file names on the MorphAdorner command line. URLs may not contain wildcard characters.

(2) The Early Modern English data files have been updated. There are new abbreviations files and Latin word files.

(3) The tokenizer for Early Modern English now handles combining macrons. There is a new map for mapping words with combining macrons to their demacronized form, as well as a list of rules.

(4) Some bad memory leaks have been plugged.

(5) The English Lemmatizer handles a wider variety of unusual contractions.

(6) Old style Roman numerals of the form .romannumeral. are now treated as a single token rather then a period followed by the romannumeral followed by a period.

(7) Archaic printer's marks encoded using a sup tag (e.g., y^t for “that”) are now handled correctly.

(8) The XML validator now works again.

(9) Merging adornments into XML is now faster.

(10) The default license text has changed from the GPL to an NCSA (BSD-like) variant.

Main changes since v0.7.1.

(1) Added a simple thesaurus interface and a default implementation using Brett Spell's Java interface to WordNet. Added the jaws-bin-1.1 jar to the classpath and added a data/wordnet directory tree. The WordNet data is from v3.0 of the Princeton release.

(2) Changed the XML path ID options to add the current file name base as the first part of the generated ID.

(3) Added an interface class for part of speech tag mapping. There are no concrete implementations included yet.

(4) Added the boolean configuration parameter xmlhandler.output_pseudo_page_boundaries to add page milestone elements at word internals specified by xmlhandler.pseudo_page_size (default value 300). The xmlhandler.pseudo_page_container_div_types lists the tag types which should force the end of a pseudo-page. Default list is volume, sermon, chapter.

A standalone utility AddPseudoPages is also available.

(5) Corrected sentence and word numbering for XML files when adorner.output.word_number and/or adorner.output.sentence_number are set true. Adding word and sentence numbers requires two passes for output generation instead of one, which slows processing significantly.

(6) Added support for RelaxNG (and W3C) schema definitions.

The configuration parameter xmlhandler.xml_schema provides the name of a schema to use for XML input parsing. Default is:

http://ariadne.northwestern.edu/monk/dtds/TEIAnalytics.rng .

The ValidateXMLFiles tool has been modified to accept a relax NG or W3C schema as the first program argument. This schema is assumed to apply to all the subsequently listed XML files.

(7) Corrected numerous errors in the Early Modern English spelling map. There are a number of problems left.

(8) Externalized the word class dependent spelling map as spellingsbywordclass.txt . Currently the same map is used for Early Modern English and 19th century fiction. The "-w" configuration parameter specifies the name of the word class dependent spelling map.

(9) Configurations now operate as overrides of the default configuration file morphadorner.properties . Only configuration parameters which different from the base settings now appear in ncfa.properties, eme.properties, etc.

(10) Moved some jump tags to the hard tags list. This results in smoother word numbering.

(11) Fixed a problem with empty soft tags generating incorrectly joined words.

(12) Corrected a number of problems with the lemmatization of capitalized words. Also added two batch file and scripts for the Relemmatize tool: relemmatizencf[.bat] relemmatizes texts using the 19th century fiction data files, while relemmatizeeme[.bat] relemmatizes texts using the Early Modern English data files.

(13) Improved the indented adorned output to ensure closing tags get placed on a line of their own when they should be.

(14) The adornwithne[.bat] tool now works properly on Linux systems with Java 1.5 .

(15) Headless exceptions should no longer appear on Linux systems.

(16) Already adorned files can be readorned by reprocessing them with MorphAdorner. Previously adorned files are automatically detected. The tokenization is left unchanged, so the word IDs remain the same. Retaggers than can change the tokenization are disabled during readornment.

(17) Corrected a problem in which parts of two splits words occurring in sequence were sometimes treated as the same word.

(18) Removed the configuration option to generate sentence milestones. These cannot be generated reliably because of intrusive jump tags. Sentence number attributes are reliable and handle jump tags properly, so we may want to consider adding these as defaults.

Main changes since v0.7.

(1) Updated the English irregular forms list for the English lemmatizer.

(2) Added TabWordFileReader and TabWordInfo classes for working with XMLToTab output.

(3) Corrected some weaknesses in the CSV package.

Main changes since v0.6.1.

(1) Updated the lexicons and associated data for both 19th Century Fiction and Early Modern English.

(2) Added a TextSegmenter package. Includes modified versions of Choi's C99 linear text segmenter and Choi's Java version of Hearst's TextTiler.

(3) Fixed the Sentence Melder to not output a blank after a leading quote in a sentence.

(4) Added a TextSummarizer package. The sample summarizer is useless for literature, but serves as a simple example.

(5) Added a StopWords package to consolidate the different stopwords lists.

(6) Removed the type AdornedWordList and used generics instead.

(7) Added a WordCounts package. Very incomplete.

(8) Added xmlhandler.fix_gap_tags and xmlhandler.fix_orig_tags boolean configuration elements to allow enabling/disabling special processing of <gap> and <orig> tags, respectively. Both currently enabled by default.

Main changes since v0.6.

(1) Improved the sentence splitting heuristics for initials.

(2) Modified the name recognizer to include methods for getting and setting a part of speech tagger. Also improved the default name recognizer to use part of speech tags to extract names based upon proper noun phrases.

Main changes since v0.5.1.

(1) Corrected a bug in the part of speech guesser which resulted in the emission of a "*" for the part of speech in some cases.

(2) Added a verb inflector for English.

(3) Added a noun pluralizer for English.

(4) Added a simple relemmatizer for updating lemmata in a MorphAdorned file.

(5) Added a syllable counter for English.

(6) Made the English lemmatizer a little less conservative, to better match the lemmata in the training data.

Main changes since v0.5.0.

(1) Added an implementation of the Lovins stemmer.

(2) Fixed an error in the Lancaster stemmer which occurred when a word ended in a letter outside the range "a" through "z".

Main changes since v0.4.0 .

(1) The bad XML output caused by certain erroneous XML input should no longer occur.

(2) Fewer extraneous whitespace elements are added to adorned XML output. In particular, whitespace elements are not emitted before an ending hard or jump tag.

(3) The default categorization of XML tags into hard/soft/jump and main/side have been adjusted slightly. In particular the

<l> tag is now treated as a soft tag instead of a hard tag.

(4) Nearly all of the code has been modified to use Java 1.5 generics. This significantly improves the clarity of the code as regards to types used in maps, lists, and sets, among other things.

(5) A new "compare strings" tool produces Dunning's log-likelihood given two files containing count maps.

(6) A new AdornedWordFilter interface provides for defining filters for adorned words. A sample stop word filter based upon Martin Porter's stop word list is included as PorterStopWordFilter.

(7) Some obsolete classes have been removed, including the old "printf" class (replace by the standard Java 1.5 class) and the old pull-like Sax wrappers.

(8) The training data for nineteenth century fiction has been updated to include American fiction as well as British fiction.

(9) The standardized spellings can be adjusted to produce "British" or "American" spellings, depending upon the corpus.

(10) The new "adornwright" script and batch file may be used to adorn American nineteenth century fiction, principally the Wright archive.

Known bugs and shortcomings in snapshot v0.5.0 .

(1) The NUPos tag set remains in flux.

(2) The Hepple tagger remains broken.

(3) Not all of the MorphAdorner tools have batch files/script files yet.

MorphAdorner License

The MorphAdorner source code and data files fall under the following NCSA style license. Some of the incorporated code and data fall under different licenses as noted in the section third-party licenses below.

Developed by:
Academic and Research Technologies
Northwestern University
http://www.it.northwestern.edu/about/departments/at/

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal with the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimers.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimers in the documentation and/or other materials provided with the distribution.
3. Neither the names of Academic and Research Technologies, Northwestern University, nor the names of its contributors may be used to endorse or promote products derived from this Software without specific prior written permission.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE CONTRIBUTORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS WITH THE SOFTWARE.

Also please see the section on support on page 16.

Third-party Licenses

1. Apache Ant
   Copyright © 2000-2008 The Apache Software Foundation.
   Licensed under the Apache Software License 2.0.
   For complete license information, please see Apache Ant license.
2. Apache Log4J
   Copyright © 1999 The Apache Software Foundation. All rights reserved.
   Licensed under the Apache Software License 1.1.
   For complete license information, please see Apache Log4j license.
3. Apache Xerces2 Java Parser
   Copyright © 1999-2004 The Apache Software Foundation. All rights reserved.

Licensed under the Apache Software License 1.1.
For complete license information, please see Apache Xerces2 Java Parser license..

4. Arithmetic Utilities from Visual Numerics
   Copyright © 1997 - 1998 by Visual Numerics, Inc.
   All rights reserved.
   Some methods in the ArithUtils class written by Visual Numerics are covered by a BSD-like license. For complete license information, please see Visual Numerics license.
5. Double Metaphone
   Written by Ed Parrish.
   Licensed under an Apache license.
6. GATE (General Architecture for Text Engineering)
   Copyright © The University of Sheffield 2001-2008.
   Licensed under the GNU Lesser General Public License. This applies to the Hepple Tagger as well.
7. ISO Relax
   Copyright © 2001-2002 SourceForge ISO-RELAX Project.
   All rights reserved.
   Licensed under a BSD style license. For complete license information, please see ISO RELAX license..
8. ISO Relax JAXP Bridge
   Copyright © 2001-2002 SourceForge ISO-RELAX Project.
   All rights reserved.
   Licensed under a BSD style license. For complete license information, please see ISO RELAX license..
9. jargs command line parser
   Copyright © 2001-2003 Steve Purcell.
   Copyright © 2002 Vidar Holen.
   Copyright © 2002 Michal Ceresna.
   Copyright © 2005 Ewan Mellor.
   The jargs command line parser library is available under a BSD-style license. For complete license information, please see jargs license.
10. Jaro-Winckler String Similarity
    Copyright (c) 2003 Carnegie Mellon University.
    The Jaro-Winckler code comes from the SecondString project and is licensed under an NCSA-style license. For complete license information, please see Jaro-Winckler license.
11. Java API for WordNet Searching 1.1
    Copyright (c) 2007 by Brett Spell.
    JAWS is available under a BSD style license. For complete license information, please see JAWS license.
12. JDOM
    Copyright © 2000-2004 Jason Hunter & Brett McLaughlin.
    All rights reserved.
    JDOM is available under an Apache-style open source license, with the acknowledgment clause removed. For complete license information, please see JDOM license.
13. Jetty
    Copyright © 1995-2006 Mort Bay Consulting Pty Ltd.
    All rights reserved.

Jetty is licensed under the Apache Software License 2.0.
For complete license information, please see Jetty license and exceptions.

14. JlinkGrammar
JLinkGrammar is licensed under a BSD-like license (although early references also suggested it could be licensed under the GNU General Public license).

15. Lancaster Stemmer
The Lancaster stemmer implementation in WordHoard is based upon Java code written by Christopher O'Neill and Rob Hooper. The original code was obtained from The Lancaster Stemming Algorithm web site. The following licensing information was provided by Dr. Chris Paice.

Paice/Husk Stemmer - License Statement.

This software was designed and developed at Lancaster University, Lancaster, UK, under the supervision of Dr Chris Paice. It is fully in the public domain, and may be used or adapted by any organisation or individual. Neither Dr Paice nor Lancaster University accepts any responsibility whatsoever for its use by other parties, and makes no guarantees, expressed or implied, about its quality, reliability, or any other characteristic.

It is assumed that, as a matter of professional courtesy, anyone who incorporates this software into a system of their own, whether for commercial or research purposes, will acknowledge the source of the code.

16. Longest Common Subsequence
Copyright © 2005 Neil Jones.
All rights reserved.
The longest common subsequence code is provided AS-IS. You may use this code in any way you see fit, EXCEPT as the answer to a homework problem or as part of a term project in which you were expected to arrive at this code yourself.

17. Mersenne Twister
Liberal use license contained in the source file.
For complete license information, please see Mersenne twister license.

18. NGramJ - n-gram based text classification
Copyright © 2001- Frank S. Nestel.
Licensed under the GNU Lesser General Public License.

19. Pluralizer
Original pluralizer was code written by Tom White and licensed under the Apache Software License 2.0.

20. Sun Multischema Validator (MSV)
Copyright (c) 2001-2008 Sun Microsystems, Inc.
All rights reserved.
Licensed under a BSD style license. For complete license information, please see Sun MSV license..

21. Porter Stemmer
Martin Porter's home page says "All these encodings of the algorithm can be used free of charge for any purpose."
For complete license information, please see Martin Porter's home page at http://www.tartarus.org/~martin/PorterStemmer/.

22. SAX
    The SAX processors are in the public domain.
23. Text Segmentation
    The C99 and Text Tiling algorithms are based upon implementations written by Freddy Choi.
    Use of this code is free for academic, education, research and other non-profit making uses only.
24. XGTagger
    Licensed under the CeCILL license. For complete license information, please see CeCILL license.

MorphAdorner Support

While we are happy to hear about your experiences with MorphAdorner, our current programming commitments limit the amount of individual support we can extend to scholars or research centers using MorphAdorner. If you find an error, we would appreciate hearing about it. We cannot promise to fix all reported errors, nor can we promise to add new features as requested. Modifications and extensions to MorphAdorner are driven principally by our local needs to support Northwestern University faculty projects.

The MorphAdorner license allows to modify the code any way you please. If you create an extension that adds functionality to MorphAdorner, and you are willing to make the code available under the MorphAdorner license, please feel free to send it to us for potential inclusion in a future release of MorphAdorner.

You may contact us at pib@northwestern.edu.

Credits

MorphAdorner was designed, implemented, and documented by Philip R. "Pib" Burns of Academic and Research Technologies. MorphAdorner was partially funded by grants from The Andrew W. Mellon Foundation for WordHoard and Monk, and by ProQuest and the CIC Universities for the Virtual Orthographic Standardization project. Martin Mueller, Professor of English and Classics, was the faculty sponsor for these projects.

John L. Norstad contributed most of the section entitled “NUPos and Morphology” (page 76).

Part Two:Adorning A Text

The MorphAdorner distribution comes packaged with Windows batch files and Unix/Linux script files to execute MorphAdorner for the texts contained in the Monk collection. You may use these batch files as a basis for developing scripts to adorn other collections of texts.

The Linux/Unix scripts assume that the "java" command invokes that standard Sun Java run time environment, not the Gnu Java runtime. MorphAdorner does not run under the Gnu Java run time environment. MorphAdorner requires Sun Java v1.5 or later.

1. The **adorndocsouth.bat** Windows batch file and the **adorndocsouth** Unix shell script execute MorphAdorner using data files suitable for adorning texts from the *Documenting the American South* nineteenth century English language texts. The texts must be encoded in TEI (Text Encoding Initiative) format using the utf-8 character set.
2. The **adornncf.bat** Windows batch file and the **adornncf** Unix shell script execute MorphAdorner using data files suitable for adorning nineteenth century English language fiction texts. The texts must be encoded in TEI (Text Encoding Initiative) format using the utf-8 character set.
3. The **adornncfa.bat** Windows batch file and the **adornncfa** Unix shell script execute MorphAdorner using data files suitable for adorning nineteenth century English language fiction texts in which apostrophes are completely distinguished from left and right single quotes (e.g., the standard Unicode curly quote characters for left and right single quote are used, and the usual apostrophe character is reserved for actual apostrophes). The texts must be encoded in TEI (Text Encoding Initiative) format using the utf-8 character set.
4. The **adornecco.bat** Windows batch file and the **adornecco** Unix shell script execute MorphAdorner using data files suitable for adorning eighteenth century English language texts. The texts must be encoded in TEI (Text Encoding Initiative) format using the utf-8 character set.
5. The **adorneme.bat** Windows batch file and the **adorneme** Unix shell script execute MorphAdorner using data files suitable for adorning early modern English language texts. The texts must be encoded in TEI (Text Encoding Initiative) format or the EEBO/TCP format using the utf-8 character set.
6. The **adornplainemetext.bat** Windows batch file and the **adornplainemetext** Unix shell script execute MorphAdorner using the early modern English data files. The input texts must be plain Ascii texts encoded using the utf-8 character set.
7. The **adornplaintext.bat** Windows batch file and the **adornplaintext** Unix shell script execute MorphAdorner using the nineteenth century fiction data files. The input texts must be plain Ascii texts encoded using the utf-8 character set.
8. The **adornwright.bat** Windows batch file and the **adornwright** Unix shell script execute MorphAdorner using data files suitable for adorning nineteenth century texts from the Wright fiction archive. This script is probably suitable for other American texts of the nineteenth century. The texts must be encoded in TEI (Text Encoding Initiative) format using the utf-8 character set.

The Unix shell scripts should work with little or no modification under Mac OSX.

For example, to adorn a nineteenth century fiction text on a Windows system, open a command line prompt and move to the MorphAdorner installation directory. Then type the following command:

```
adornncf \outputdir \inputdir\mytext.xml
```

where **\outputdir** specifies the name of a directory into which to write the adorned xml output, and **\inputdir\mytext.xml** specifies the file name of the text to adorn. The output file name will be the same as the input file name. However, if a file of that name already exists in the output directory, a "versioned" file name will be created to avoid overwriting the existing file. For example, should the file "mytext.xml" already exist in the output directory, the output file name will be changed to "mytext-001.xml". More generally, the three digit version number starts at "001" and is incremented as necessary to produce a non-existing file name.

Alternatively, MorphAdorner optionally allows you to specify that texts with a matching adorned version in the current output directory should not be readorned. See the description of the *xml.adorn_existing_xml_files* configuration setting (page 22) for or more details.

You may specify more than one file to adorn, and you may specify wildcards to match more than one file. For example:

```
adornncf \outputdir \inputdir\*.xml
```

adorns all the files with the extension **.xml** in the directory **\inputdir**.

On a Unix/Linux/Mac OSX system, open a terminal window, move to the MorphAdorner installation directory, and type the following command:

```
./adornncf /outputdir /inputdir/mytext.xml
```

Don't forget to mark the **adornncf** script file as executable before using it. On most Unix/Linux systems you can use the **chmod** command to do this:

```
chmod 755 adornncf
```

If you know for certain that the text you wish to adorn distinguishes the use of the apostrophe character (') from left and right single quotes (Unicode characters 0x2018 and 0x2019 respectively), you may use **adornncfa** instead of **adornncf**.

To adorn an early modern English text, substitute **adorneme** for **adornncf** in the command line. To adorn plain text using the nineteenth century data file, substitute **adornplaintext** for **adornncf** in the command line.

MorphAdorner writes a log of its activities to standard system output, which is usually the display. You may redirect standard output to another file in the usual fashion. For example, under Windows, to redirect the MorphAdorner log output to a disk file, type:

```
adornncf \outputdir \inputdir\mytext.xml >myoutput.lis
```

where **myoutput.lis** is the name of the file to which to redirect MorphAdorner's logging output. If you have the **tee** utility installed, you can redirect the output to a file and watch the output displayed to your screen at the same time:

```
adornncf \outputdir \inputdir\mytext.xml | tee myoutput.lis
```

The **tee** utility is usually provided by default on most Unix/Linux and Mac OSX systems. The **tee** utility is not provided as a standard part of Microsoft Windows operating systems. Third party Windows implementations are available. You may download a Windows implementation of tee as tee.zip. Use your favorite unzip program to extract **tee.exe** from **tee.zip**. Place **tee.exe** in the MorphAdorner installation directory.

Java OutOfMemory Errors

Each of the batch and script files above invoke MorphAdorner with a Java virtual machine size of 720 megabytes. This means your PC needs to have a minimum of one gigabyte of memory. The 720 megabyte size is sufficient for adorning texts containing up to a quarter of a million words or so. Longer texts may require a larger Java virtual machine memory allocation. If you see the error message

```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
```

in the MorphAdorner output log, you need to specify a larger heap space setting to Java. MorphAdorner is a memory intensive program, especially when adorning large XML encoded texts.

If your system has more than a gigabyte of memory installed, you can raise the Java virtual machine size by modifying the value of the ***java -Xmx720m*** parameter in the batch file or script you are using. To specify a larger heap size, e.g., 1,500 megabytes for example, change ***java -Xmx720m*** to ***java -Xmx1500m*** . However, even when your system has more than two gigabytes of memory you may not be able to request a heap size that large on a 32 bit operating system. You will need to experiment with different heap size settings to find the maximum your particular system allows.

For large texts containing millions of words you may need to run MorphAdorner on a system with a 64 bit version of Java. For example, we found that several of the longest texts in the EEBO collection required a virtual machine size of several gigabytes, e.g., ***java -Xmx8g*** for an eight gigabyte size.

If you encounter the OutOfMemory error when running a MorphAdorner utility program, you can modify the heap size setting in the batch file or script for that program as well.

Part Three: Configuring MorphAdorner

MorphAdorner Command Line

The MorphAdorner command line takes the following form.

```
java  -Xmx640m -Xss1m edu.northwestern.at.morphadorner.MorphAdorner
      -a spellingpairs.tab
      -l lexicon.lex
      -o adornedoutput/
      -p overriding.properties
      -r contextrules.txt
      -s standardspellings.txt
      -t transitionmatrix.mat
      -u suffixlexicon.lex
      -w spellingsbywordclass.txt
      -x lexicalrules.txt
      input1 input2 ...
```

where

Parameter	Definition
a	A spelling map file. This file contains two columns separated by a tab. The first column is a variant spelling. The second column is the standard spelling. You may repeat this argument multiple times to specify more than one spelling map.
l	A word lexicon file in MorphAdorner format.
o	The directory into which adorned output files are written.
p	A MorphAdorner Configuration Settings (page 22) file. The entries in this file override the default *morphadorner.properties* file.
r	The name of a file providing contextual rules for a rule-based part of speech tagger.
s	A text file containing a list of standard spellings, one per line.
t	The part of speech tag transition probability matrix used by the probabilistic part of speech taggers.
u	A suffix lexicon file in MorphAdorner format. This should be generated from the word lexicon file specified by the *l*= parameter.
w	A spelling map file which breaks down the variant to standard spellings by word class.
x	The name of a file providing lexical rules for a rule-based part of speech tagger.
Input1 input2 ...	The input files to be adorned.

Settings which appear in the default *morphadorner.properties* file will be overridden by those specified in the *p*= properties file. The other command line parameters override the settings in both properties files.

See the batch files and scripts such as **adornncf**, **adorneme**, etc. provided in the MorphAdorner release materials for examples of the use of the MorphAdorner command line parameters.

MorphAdorner Configuration Settings

The configuration settings for MorphAdorner appear in utf-8 text files. Each setting takes the form setting=value and appears on a separate line in the configuration file. The default settings file is called *morphadorner.properties*. Overriding settings may be specified by in a file named by the *p*= parameter on the MorphAdorner command line (page 20). A number of sample settings files are provided in the MorphAdorner release materials, corresponding to settings used when adorning files in the various corpora used in the Monk project.

Properties file	Corpus
docsouth.properties	Documenting the American South XML files.
eaf.properties	Early American Fiction XML files.
ecco.properties	Eighteenth Century Collections Online XML files.
eme.properties	Early Modern English XML files.
emeplaintext.properties	Early Modern English plain text files.
ncf.properties	Nineteenth Century Fiction XML files in which the apostrophe, opening single quote, and closing single quote are the same character.
ncfa.properties	Nineteenth Century Fiction XML files in which the apostrophe is distinguished from the opening and closing single quote characters,.
plaintext.properties	Plain text of nineteenth century vintage or later.
wright.properties	Wright Fiction Archive XML files.

The following table lists the setting names and their definitions, along with typical values.

MorphAdorner Configuration Settings

Setting Name	Description and Values
abbreviations.abbreviations_url	Specifies the URL for an extra list of abbreviations. Such a list for Early Modern English texts may be found in **data/emeabbreviations**.
adornedwordoutputter.class	Class which produces adorned word values. The following output classes are currently implemented in MorphAdorner. • *PrintStreamAdornedWordOutputter* writes words and their adornments as plain utf-8 text in tab-separated columns to a file. This is the default output format.

	• *ConsoleAdornedWordOutputter* writes words and their adornments as plain utf-8 text in tab-separated columns to the default system output device.
	• *ListAdornedWordOutputter* writes words and adornments to an internal list of strings. This is used when processing XML input files.
	• *SimpleXMLAdornedWordOutputter* outputs words and their adornments to a file in a simple XML format.
	
	`<words>`
	`  <word id="1">`
	`  <tok>Poets</tok>`
	`  <spe>Poets</spe>`
	`  <pos>n2</pos>`
	`  <reg>Poets</reg>`
	`  <lem>poet</lem>`
	`  <eos>0</eos>`
	`  </word>`
	`  <word id="2">`
	`  ...`
	`  </word>`
	`  ...`
	`</words>`
	
	• *ByteStreamAdornedWordOutputter* writes words and adornments to an internal byte stream.

adorner.handle_xml	*true* to use the TEI XML handler, *false* to use the ordinary text handler.
adorner.lemmatization.ignorelexiconentries	*true* to ignore lemma definitions in the current lexicon file when generating output lemmata, and use only the current lemmatizer. *false* to look at the lemma definitions in the lexicon first, and use the lemmatizer only when there is no lemma definition in the lexicon.
adorner.output.end_of_sentence_flag	*true* to output an end of sentence flag for each adorned word, *false* to not generate this flag. The attribute value is set to "1" when a word ends a sentence and "0" otherwise.
adorner.output.end_of_sentence_flag_attribute	The name of the XML word attribute for the end of sentence flag. The default value is *eos*.
adorner.output.kwic	*true* to output keyword in context (kwic) entries for

	each adorned word, *false* to not generate these entries.
adorner.output.kwic.width	The number of characters of kwic text to output. 80 is a typical value, which is split between the left and right kwic text.
adorner.output.kwic_left_attribute	The name of the XML word attribute for the kwic text appearing before a word. The default value is *kl*.
adorner.output.kwic_right_attribute	The name of the XML word attribute for the kwic text appearing after a word. The default value is *kr*.
adorner.output.lemma	*true* to output the lemma for an adorned word, *false* otherwise.
adorner.output.lemma_attribute	The name of the XML word attribute for the lemmata of an adorned word. The default value is *lem*.
adorner.output.original_token	*true* to output the original word token for an adorned word, *false* otherwise.
adorner.output.original_token_attribute	The name of the XML word attribute for the original word token of an adorned word. The default value is *tok*.
adorner.output.part_of_speech	*true* to output the part of speech for an adorned word, *false* otherwise.
adorner.output.part_of_speech_attribute	The name of the XML word attribute for the part of speech of an adorned word. The default value is *pos*.
adorner.output.running_word_numbers	*true* to output the word numbers for adorned words as continuously ascending values. *false* to restart the word numbers over for each sentence.
adorner.output.sentence_number	*true* to output the sentence number for an adorned word, *false* otherwise.
adorner.output.sentence_number_attribute	The name of the XML word attribute for the sentence number for an adorned word. The default value is *sn*.
adorner.output.spelling	*true* to output the spelling for an adorned word, *false* otherwise.
adorner.output.spelling_attribute	The name of the XML word attribute for the spelling for an adorned word. The default value is *spe*.
adorner.output.standard_spelling	*true* to output the standard spelling for an adorned word, *false* otherwise.
adorner.output.standard_spelling_attribute	The name of the XML word attribute for the standard spelling for an adorned word. The default value is *reg*.
adorner.output.word_number	*true* to output the word number for an adorned word,

	false otherwise.
adorner.output.word_number_attribute	The name of the XML word attribute for the word number for an adorned word. The default value is *wn*.
adorner.output.word_ordinal	*true* to output the word ordinal for an adorned word, *false* otherwise.
adorner.output.word_ordinal_attribute	The name of the XML word attribute for the word ordinal for an adorned word. The default value is *ord*.
initialspellingstandardizer.class	The initial spelling standardizer class. This is used when guessing parts of speech for words not present in the lexicon. *NoopSpellingStandardizer*, the default, leaves spellings unstandardized when guessing parts of speech.
lexicon.suffix_lexicon	The file containing the suffix lexicon. For the standard MorphAdorner release, the lexicon files appear in the data/ subdirectory. The 19th century fiction suffix lexicon is *data/ncfsuffixlexicon.lex* and the Early Modern English suffix lexicon is *data/emesuffixlexicon.lex*. This value may be overridden on the MorphAdorner command line by the *-u* parameter.
lexicon.word_lexicon	The file containing the word lexicon. For the standard MorphAdorner release, the lexicon files appear in the data/ subdirectory. The 19th century fiction word lexicon is *data/ncfwordlexicon.lex* and the Early Modern English word lexicon is *data/emewordlexicon.lex*. This value may be overridden on the MorphAdorner command line by the *-l* parameter.
morphadornerxmlwriter.class	The class for writing adorned XML files. *DefaultMorphAdornerXMLWriter* is the default. This should not be changed unless you implement a new XML writer class.
namestandardizer.class	The proper name standardizer class. *DefaultNameStandardizer* is the default, which implements the scheme described in Standardizing Proper Names. The *NoopNameStandardizer* class leaves names unstandardized. The *EEBOSimpleNameStandardizer* class corrects a handful of names when processing early modern English texts.
partofspeechguesser.check_possessives	*true* to check for possessive endings when guessing

	the part of speech for an unknown word, *false* otherwise. The default setting is false, which is also the recommended setting.
partofspeechguesser.class	The part of speech guesser class, which tries to determine the most likely parts of speech for an unknown word. *DefaultPartOfSpeechGuesser* is the default which is designed for English words.
partofspeechguesser.try_standard_spellings	*true* to use standard spellings when guessing the parts of speech for unknown words, *false* to use the original spellings only. The default setting is true.
partofspeechretagger.class	The class which corrects the initial part of speech tagging. The *IRetagger* class applies a short list of fixup rules to improve the tagging of **I** tokens. The *NoopRetagger* class leaves the original tagging unchanged. The *PronounRetagger* class applies a short list of fixup rules to improve the tagging of pronouns. The *DefaultPartOfSpeechRetagger* is the same as *IRetagger.*
partofspeechtagger.class	The class which perform part of speech tagging. The default *TrigramTagger* which is a hidden Markov model based trigram tagger. This is the workhorse tagger in MorphAdorner. Other taggers, mostly experimental, include: • *AffixTagger* uses an affix lexicon to assign a part of speech tag to a word based upon the prefixes or suffixes of the word. • *BigramTagger* is a hidden Markov model based bigram tagger. It is faster but less accurate than the trigram tagger. • *BigramHybridTagger* combines the bigram tagger with a second pass by a Hepple tagger to correct the initial tagging. Note: you must supply the correction rules, none are provided by default. • *HeppleTagger* is Mark Hepple's rule-based part of speech tagger modified from the version in Gate to work with the MorphAdorner lexicons, guessers, etc. • *RegexpTagger* uses regular expressions to assign a part of speech tag to a word. You must supply the regular expressions, none are provided by default. • *SimpleTagger* assigns a "noun" part of speech

	to all words, except those that appear to be numbers. Numbers are assigned a "number" part of speech. Words starting with a capital letter can be assigned a separate "proper name" part of speech. This tagger is mostly useful as a backup to a more sophisticated tagger. • *SimpleRuleBasedTagger* assigns the most commonly occurring part of speech to all words using a lexicon, and then applies a small set of contextual rules to "fix up" the tagging. This simple tagger is useful when very fast tagging without high accuracy is useful, e.g., in sentence splitting. • *TrigramHybridTagger* combines the trigram tagger with a second pass by a Hepple tagger to correct the initial tagging. Note: you must supply the correction rules, none are provided by default. • *UnigramTagger* uses a lexicon to assign the most frequently occurring part of speech tag to a word.
partofspeechtagger.transition_matrix	The file containing the tag transition probability matrix data. For the standard MorphAdorner release, these files appear in the data/ subdirectory. The 19th century fiction transition matrix file is *data/ncftransmat.mat* and the Early Modern English transition matrix file is *data/emetransmat.mat*. This value may be overridden on the MorphAdorner command line by the *-t* parameter.
pretokenizer.class	The class which applies any pretokenization corrections to the text to prepare it for initial token extraction. The default is *DefaultPreTokenizer* which ensures that characters which should always be separate tokens are surrounded by whitespace. In general this class should always be used. The *EEBOPreTokenizer* was written to correct the text for EEBO texts before those texts were modified by Abbott to conform to TEI Analytics standards.
posttokenizer.class	The class which applies any tokenization corrections to the initial token extraction. The default is *DefaultPostTokenizer*. The *EEBOPostTokenizer* was written to correct tokens extracted from EEBO texts before those texts were modified by Abbott to conform

	to TEI Analytics standards.
sentencesplitter.class	The class which determines sentence boundaries. *ICU4JBreakIteratorSentenceSplitter* uses an ICU4J BreakIterator to identify candidate sentences. Several heuristics are used to correct the initial sentence identification for English sentences. The *DefaultSentenceSplitter* is the same as *ICU4JBreakIteratorSentenceSplitter.*
spelling.spelling_pairs	The spelling data file which maps variant spellings to standard spellings. For the standard MorphAdorner release, these files appear in the data/ subdirectory. The 19th century fiction spelling map file is *data/ncfmergedspellingpairs.tab* and the Early Modern English spelling map file is *data/ememergedspellingpairs.tab*. This value may be overridden on the MorphAdorner command line by the *-a* parameter.
spelling.spelling_pairs_by_word_class	The spelling data file which maps variant spellings to standard spellings by word class. For the standard MorphAdorner release, these files appear in the data/ subdirectory. The spelling map by word class file used for all periods is *data/spellingsbywordclass.txt* . This value may be overridden on the MorphAdorner command line by the *-w* parameter.
spelling.standard_spellings	The spelling data file which list standard spellings. For the standard MorphAdorner release, this file is *data/standardspellings.txt* . This value may be overridden on the MorphAdorner command line by the *-s* parameter.
spellingmapper.class	The spelling mapper class which maps spellings from one dialect to another. The *USToBritishSpellingMapper* maps United States spellings to British spellings, while *BritishToUSSpellingMapper* maps British spellings to United States spellings.
spellingstandardizer.class	The class which maps variant spellings to standard spellings. The *DefaultSpellingStandardizer* class is the *ExtendedSimpleSpellingStandardizer* which uses spelling maps along with a few simple heuristics to find standard spellings given a variant spelling. The *SimpleSpellingStandardizer* class only uses spelling maps. The *ExtendedSearchSpellingStandardizer* implements the full scheme discussed at Spelling

	Standardization Process which can lead to exotically erroneous standard spellings in some cases.
textinputter.class	The class which reads input text for adornment. The *DefaultTextInputter* class is the *URLTextInputter* which reads utf-8 text from a URL. The *SimpleXMLTextInputter* reads utf-8 text from a TEI or EEBO XML file. The *DiskBasedXMLTextInputter* also reads utf-8 text from a TEI or EEBO XML file, but divides the file into smaller sections which are stored in temporary disk files and adorned separately. This is useful for working with large XML input files. The *FirstTokenURLTextInputter*reads only the first token in each line from a URL.
wordlists.use_latin_word_list	*true* to use an extended list of Latin words when adding part of speech tags to words, *false* to not use the extended list.
wordtokenizer.class	Class which splits a sentence into word tokens. *DefaultWordTokenizer* is the default and is suitable for English text.
xml.adorn_existing_xml_files	*true* to adorn XML files with an existing adorned version in the output directory, *false* to skip adornment for existing files. *true* is the default value. When set *true* and an existing adorned file exists, a versioned output file name is created to avoid overwriting the previous adorned version. For example, if the file "aaa.xml" is to be adorned, and the adorned version "aaa.xml" already exists in the output directory, then the file "aaa-001.xml" is created. If "aaa-001.xml" already exists, "aaa-002.xml" is created, and so on.
xml.close_sentence_at_end_of_hard_tag	*true* to force a sentence to close at the end of a hard tag, *false* to allow a sentence to cross across hard tags. In many literary texts sentences do cross hard tag (usually paragraph) boundaries, so this setting should be set false.
xml.close_sentence_at_end_of_jump_tag	*true* to force a sentence to close at the end of a jump tag, *false* to allow a sentence to cross across hard tags. This setting should generally be set to true.
xml.disallow_word_elements_in=figDesc sic	Specifies the XML elements in which to disallow generated and elements. Element names are separated by blanks. The default list is `figDesc sic`.
xml.field_delimiters	Field delimiters for adorned word output. The default is the Ascii tab character \t . This should not generally

	be changed.
xml.fix_gap_tags	*true* to fix <gap> tags in XML texts, *false* to leave them alone. In general, if the input texts are in TEI Analytics format, this setting should be false.
xml.fix_orig_tags	*true* to fix <orig> tags in XML texts, *false* to leave them alone. In general, if the input texts are in TEI Analytics format, this setting should be false.
xml.fix_split_words	*true* to fix split words in XML texts, *false* to leave them alone. The match patterns are regular expressions specified by the settings *xml.fix_split_words.match1*, *xml.fix_split_words.match2*, etc. The corresponding corrections are specified by the settings *xml.fix_split_words.replace1*, *xml.fix_split_words.replace2*, etc. These patterns may actually be used for more general purposes than splitting or joining words. Examples of these settings may be found in the *eme.properties* settings file in the MorphAdorner release.
xml.id.attribute	The name of the XML word ID attribute. The default value is *xml:id*.
xml.id.spacing	This setting gives the spacing between ID values. For example, an increment of 10 spaces reading_context_order or wordinblock values by 10. This allows new values to be interpolated for editing purposes. The default value is 10.
xml.id.type	Word IDs start with the work identifier, taken from the file name of the work. *reading_context_order* appends integer values whose order gives the reading context order defined by the classification of hard, soft, and jump tags. *word_within_page_block* appends two integer values in the the form pageblocknumber-wordinblock, where pageblocknumber is the ordinal of the current (page break) entry, and wordinblock is the number of the word within the page block (starting at 1).
xml.ignore_tag_case	*true* to ignore the case of XML tags when processing them, *false* to consider different tag case significant. The default is true.
xml.jump_tags	The list of XML jump tags, separated by blanks.

	MorphAdorner uses the following jump tags for the default TEI Analytics XML input files. `bibl figdesc figDesc figure footnote note ref stage tailnote`
xml.log	*true* to enable extended logging, *false* otherwise. The default is *false*.
xml.output_nonredundant_attributes_only	*true* to emit only non-redundant word tag attributes, *false* to emit all word attributes. A word attribute is redundant if its value can be determined from the data enclosed by the tags or from another tag value. By default MorphAdorner emits all word tag values even if redundant.
xml.output_nonredundant_token_attribute	*true* to emit only non-redundant token attributes, *false* to emit all token attributes. A redundant token attribute specifies the same text as the data enclosed by the tags. By default MorphAdorner emits all token values even if redundant.
xml.output_pseudo_page_boundary_milestones	*true* to emit XML pseudopage boundary milestone elements, *false* to not emit these milestones.
xml.output_whitespace_elements	*true* to emit whitespace elements (e.g.,) between word elements in XML, *false* to not emit these whitespace elements. This setting should be true in most cases.
xml.pseudo_page_container_div_types	The list of XML tags which close a pseudopage, separated by blanks. MorphAdorner uses the following soft tags for the default TEI Analytics XML input files. `volume chapter sermon`
xml.pseudo_page_size	The maximum length in words of a pseudopage. The default is 300 words.
xml.soft_tags	The list of XML soft tags, separated by blanks. MorphAdorner uses the following soft tags for the default TEI Analytics XML input files. `abbr add address author c cl corr date emph foreign gap`

	`hi l lb location m mentioned milestone money name num organization orig pb person phr reg rs s sb seg sic soCalled sub sup term time title unclear w zzzzsw`
xml.surround_marker	The marker character used internally for surrounding distinct segments of text. Default is Unicode character \ue501 . This should not be changed.
xml.word_delimiters	Output word delimiters for adorned word output. The default is Ascii \r\n . This should not be changed.
xml.word_tag_name	The name of the XML tag which is used to mark an adorned output word. The default is *w*.
xml.xml_schema	The name of the default scheme used for parsing an XML file when none appears in the XML text. For MorphAdorner, the default is the TEI Analytics scheme which appears at http://ariadne.northwestern.edu/monk/schemata/TEIAnalytics.rng .

Part Four: Utilities

MorphAdorner provides a number of utility programs. In the following program descriptions, the command lines for the utilities may be split across multiple lines by your web browser or document reader. They should actually all appear on a single line. If you need to split the command lines you should terminate each line before the last with the command line continuation character for your operating system's command line shell. For Windows this is the caret "^". For Unix/Linux this is usually the back slash "\".

Don't forget to mark the Unix script files as executable before using them. On most Unix/Linux systems you can use the **chmod** command to do this, e.g.:

```
chmod 755 adornncfa
```

The MorphAdorner release contains a script **makescriptsexecutable** which applies **chmod** to each of the scripts in the release. On most Unix-like systems you can execute **makescriptsexecutable** by moving to the MorphAdorner installation directory and entering

```
chmod 755 makescriptsexecutable
./makescriptsexecutable
```

or

```
/bin/sh <makescriptsexecutable
```

Adding Character Offsets

AddCharacterOffsets creates derived MorphAdorner files with character offsets to word tokens.

Usage:

```
addcharacteroffsets adornedinput.xml adornedoutput.xml
unadornedoutput.xml
```

where

adornedinput.xml	Standard MorphAdorner adorned output file.
adornedoutput.xml	Derived adorned file with character offsets added to tags.
unadornedoutput.xml	Derived unadorned file whose word offsets are given in adornedoutput.xml file.

The derived adorned output file *adornedoutput.xml* adds a *cof=* attribute to each <w> tag. The *cof=* attribute specifies the character (not byte) offset of each word in the *unadornedoutput.xml* file. The latter file removes the <w> and <c> tags from the adorned input file and outputs the word and whitespace text as specified by the <w> and <c> tags. (Note that *cof=* is not recognized by the TEI Analytics scheme.)

The source code for AddCharacterOffsets is interesting in that it shows how to process an adorned file using regular expressions instead of a full XML parser.

Adding Pseudopages

AddPseudoPages adds pseudopage milestones to an adorned file.

Usage:

```
addpseudopages input.xml output.xml pseudopagesize
pageendingdivtypes
```

where

input.xml	Input MorphAdorned XML file.
output.xml	Derived adorned file with pseudopage milestones added. N.B. Existing pseudopage milestones are deleted before the new ones are added.
pseudopagesize	The maximum number of words in each pseudopage. Default: 300 .
pageendingdivtypes	Blank separated list of types which force the closure of a pseudopage. Default: chapter volume sermon

The derived adorned output file *output.xml* has pseudopage milestone elements added approximately every *pseudopagesize* words. No distinction is made between main and paratext when generating the pseudopages. Each pseudopage starts with a milestone of the form:

```
<milestone unit="pseudopage" n="n"
position="start"></milestone>
```

and ends with a milestone element of the form:

```
<milestone unit="pseudopage" n="n"
position="end"></milestone>
```

The *n* is the pseudopage number.

Pseudopages can be used as a basis for constructing simple citation schemes and mechanisms for orienting a reader in a text which is not otherwise divided into meaningful units such as pages.

Adorning Named Entities

AdornWithNamedEntities adorns XML texts with named entities such as person, location, time, date, and organization. It is an experimental procedure based upon the Gate named entity extractor ANNIE with a few modifications to improve its utility for literary purposes.

Usage:

```
adornwithnamedentities outputdirectory input1.xml
input2.xml ...
```

where

- *outputdirectory* -- output directory to receive xml files adorned with named entities.
- *input*.xml* -- input TEI XML files.

The named entity adorner does not always recognize entities which cross soft tags. Thus "Emma Woodhouse" may be recognized as two separate entities. AdornedWithNamedEntities should be run on the input files before their submission to MorphAdorner.

Gate uses the following XML tags for marking named entities. AdornWithNamedEntities maps these to the TEI Analytics "<rs>" with a specific type= attribute value.

Gate	TEI Analytics
<Date> for a date	<rs type="date">
<Location> for a location	<rs type="location">
<Money> for an amount of money	<rs type="money">
<Organization> for an organization	<rs type="organization">
<Person> for a person	<rs type="person">
<Time> for a time	<rs type="time">

Gate seems to generate "Date" where one might expect "Time" to appear.

In addition to the named entity types generated by Gate, AdornWithNamedEntities can also generate <rs type="literary"> for literary references. This has not been fully implemented.

Comparing String Counts

CompareStringCounts compares two columnar files containing spellings and part of speech tags.

Usage:

```
comparestringcounts analysis.tab reference.tab
```

where

- *analysis.tab* is an input tab-separated file of strings and counts for an analysis text.
- *reference.tab* is an input tab-separated file of strings and counts for a reference text.

The analysis.tab and reference.tab files contain strings and counts of those strings compiled from two texts or corpora. Both files contain two tab-separated columns. The first column is a string. The second column contains the count of the number of times that string occurred in the associated text.

The output contains seven tab-separated columns, sorted in descending order by log-likelihood value. One line of output appears for each string in the analysis text.

1. The first column contains the string. This may be a spelling, a lemma, a part of speech, a spelling bigram, or any other string of interest.
2. The second column contains a "+" when the string is overused in the analysis text with respect to the reference text, a "-" when the string is underused, and a blank when the string is used the same amount in both texts.
3. The third column contains Dunning's log-likelihood value.
4. The fourth column shows the relative frequency of occurrence of the string in the analysis text as fractional parts per ten thousand.
5. The fifth column shows the relative frequency of occurrence of the string in the reference text as fractional parts per ten thousand.
6. The sixth column shows the number of times the string occurred in the analysis text.
7. The seventh column shows the number of times the string occurred in the reference text.

These results are written to the standard output file which can be redirected to another file. A brief summary of the analysis is written to the standard error file.

Statistical Background

Comparisons tell you whether there is more of this here or less of that there. Knowing that individual word forms in one text occur more or less often than in another text may help characterize some generic differences between those texts. Statistics on how often the words occur add rigor and provide a framework for judging whether the observed differences are likely or unlikely to have occurred by chance, and so deserve futher attention and interpretation.

Log-likelihood for comparing texts

CompareStringCounts allows you to compare the frequencies of word occurrences in two texts and obtain a statistical measure of the significance of the differences. CompareStringCounts uses the *log-likelihood ratio* G^2 , also known as Dunning's Log-Likelihood, as a measure of difference. To compute G^2, CompareStringCounts constructs a two-by-two *contingency table* of frequencies for each word.

	Analysis Text	Reference Text	Total
Count of word form	a	b	a+b
Count of other word forms	c-a	d-b	c+d-a-b
Total	c	d	c+d

The value of "a" is the number of times the word occurs in the analysis text. The value of "b" is the number of times the word occurs in the reference text. The value of "c" is the total number of words in the analysis text. The value of "d" is the total number of words in the reference text.

Given this contingency table, CompareStringCounts calculates the *log-likelihood ratio statistic* G^2 to assess the size and significance of the difference of a word's frequency of use in the two texts. The log-likelihood ratio measures the discrepancy of the the observed word frequencies from the values which we would expect to see if the word frequencies (by percentage) were the same in the two texts. The larger the discrepancy, the larger the value of G^2, and the more statistically significant the difference between the word frequencies in the texts. Simply put, the log-likelihood value tells us how much more likely it is that the frequencies are different than that they are the same.

The log-likelihood value is computed as the sum over all terms of the form "O * ln(O/E)" where "O" is the observed value of a contingency table entry, "E" is the expected value under a model of homogeneity for frequencies for the two texts, and "ln" is the natural log. If the observed value is zero, we ignore that table entry in computing the total. CompareStringCounts calculates the log-likelihood value G^2 for each two-by-two contingency table as follows.

```
E1=c*(a+b)/(c+d)
E2=d*(a+b)/(c+d)
G2=2*((a*ln(a/E1)) + (b*ln(b/E2)))
```

To determine the statistical significance of G^2, we refer the G^2 value to the chi-square distribution with one degree of freedom. The significance value tells you how often a G^2 as large as the one CompareStringCounts computed could occur by chance. For example, a log-likelihood value of 6.63 should occur by chance only about one in a hundred times. This means the significance of a G^2 value of 6.63 is 0.01 .

References

Ted Dunning's paper discusses the use of the log-likelihood test for general textual analysis.

- Dunning, Ted. 1993. Accurate Methods for the Statistics of Surprise and Coincidence. *Computational Linguistics*, Volume 19, number 1, pp. 61-74.

Rayson and Garside discuss the use of the log-likelihood test for comparing corpora.

- Rayson, P. and Garside, R. 2000. Comparing corpora using frequency profiling. In *Proceedings of the workshop on Comparing Corpora*, held in conjunction with the 38th annual meeting of the Association for Computational Linguistics (ACL 2000). 1-8 October 2000, Hong Kong.

Comparing Adorned Files

TagDiff compares two columnar files containing spellings and part of speech tags.

Usage:

```
tagdiff input1.tab postagcol1 input2.tab postagcol2
```

where

- *input1.tab* is an input tab-separated file containing spellings in the first column and parts of speech in the second column. Usually this is a reference (training) file in which the part of speech assignments are known to be correct.
- *postagcol1* is the column number (starting at 1) which contains the part of speech tags in the first file.
- *input2.tab* is an input tab-separated file containing spellings in the first column and parts of speech in the second column. Usually this is a file produced by MorphAdorner or some other part of speech tagger.
- *postagcol2* is the column number (starting at 1) which contains the part of speech tags in the second file.

The two files must have the exact same number of lines and the same exact spellings, in order, in column one. However, blank lines are ignored in both files.

TagDiff writes a report to the standard system output file tallying the numbers and types of differences in the part of speech assignments provided by each file. If the first file is a reference file, this allows you to see how well the part of speech tagger reproduced the reference tagging. A good part of speech tagger for English normally gets at least 96% of the tags correct.

Correcting Quote Marks

FixXMLQuotes attempts to convert straight double quotes (Ascii/Unicode 34) into "curly" left and right double quotes (Unicode 8220 and 8221 respectively). It also attempts to convert straight single quotes (Ascii/Unicode 39) into "curly" left and right single quotes (Unicode 8216 and 8217 respectively) and to distinguish these from the use of the single quote as an apostrophe. FixXMLQuotes makes mistakes, so its output should be corrected manually. FixXMLQuotes accepts XML files in TEI format as input.

Usage:

```
fixxmlquotes softtags.txt jumptags.txt outputdirectory
input1.xml input2.xml ...
```

where

- *softtags.txt* specifies a text file containing list of soft XML tags, one per line. A sample is included as part of the MorphAdorner distribution.
- *jumptags.txt* specifies a text file containing list of jump XML tags, one per line. A sample is included as part of the MorphAdorner distribution.
- *outputdirectory* specifies the output directory to receive xml files with quote marks fixed.
- *input*.xml* specifies the input TEI XML files.

For each of the input XML files, FixXMLQuotes attempts to correct the quotes and writes a corrected XML file of the same name in the specified output directory.

The companion **FixQuotes** program provides the same approach to correcting quote marks, but for plain text files instead of XML files.

Usage:

```
fixquotes input.txt output.txt
```

where

- *input.txt* specifies the input text file with quote marks to correct.
- *output.txt* specifies the output text file with quote marks fixed.

At best **fixxmlquotes** and **fixquote** correct 90% of the quotes. The remainder need to be corrected manually.

Counting Affixes in an Adorned Text

CountAffixes counts affixes (suffixes and prefixes) of adorned words by processing MorphAdorned XML output.

Usage:

```
countaffixes input.xml prefixes.tab suffixes.tab
```

where

- *input.xml* -- input XML file produced as output by MorphAdorner.
- *prefixes.tab* -- output tab-separated prefixes file described below.
- *suffixes.tab* -- output tab-separated suffixes file described below.

Both the *prefixes.tab* and *suffixes.tab* output files contain two tab-separated columns. The first column is a prefix or suffix string, respectively, and the second column contains the count of the number of times that prefix or suffix occurred in the unique words in the *input.xml* file.

Why do we care about affixes? Affixes of one kind or another are a good proxy for etymologies -- at least in English. In some ways they are better, because the affix is part of the writer's or reader's repertoire in a way in which knowledge of etymologies is not. The distribution of word etymologies -- or affixes -- offers one way of studying an author's style.

For example, R. Harald Baayen argues that 'ation' is a distinctive suffix and is characteristic of the Latinate and Johnsonian streak in Jane Austen's writing. A study of affix distributions for other authors may reveal similar interesting patterns.

Counting Words In An Adorned Text

CountAdornedWords tabulates counts of adorned words from XMLToTab (page 60) output files.

Usage:

```
countadornedwords output.tab input.tab input2.tab ...
```

where

- *output.tab* is the output tab-separated count file.
- *input*.tab* are the input tabbed files produced by XMLToTab.

The output file is a tab-delimited utf-8 encodded text file containing the following fields, in order.

1. Short work name, formed from input file name by stripping the path and file extension.
2. The corrected original spelling.
3. The standard spelling.
4. The parts of speech.
5. The lemmata.
6. The count of the tuple (work name, corrected spelling, standard spelling, parts of speech, lemmata).

This output provides a "bag of words" for each input text which can then be input to a database or spreadsheet for further analysis.

Creating A Lexicon

CreateLexicon creates word and suffix lexicons from training data.

Usage:

```
CreateLexicon trainingdata.tab wordlexicon.lex
suffixlexicon.lex maxsuffixlength maxsuffixcount
```

where

- *trainingdata.tab* specifies the name of the file containing the part of speech training data from which the word lexicon and suffix lexicon are built.

  The word lexicon contains each spelling (and standard spellings if provided), the count for each spelling, the parts of speech for each spelling, the counts for each part of speech for each spelling, and the lemma for each part of speech for each spelling (if provided). The suffix lexicon contains a list of suffixes, their counts, and the parts of speech associated with each suffix and the count of each part of speech. Lemmata are stored as a "*' in the suffix lexicon since there are no lemmata for suffixes.

  The training data resides in a utf-8 text file. Each line contains one tab-separated spelling along with its part of speech tag and optionally its lemma and standard spelling in the form:

  ```
  spelling {tab} part-of-speech-tag {tag} lemma {tag}
  standard spelling
  ```

  where "{tab}" specifies an Ascii tab character.

  You must specify a spelling and a part of speech tag. The lemma and standard spelling are optional. If you wish to specify a standard spelling without specifying a lemma, enter the lemma as "*".

  Blanks lines are used to separate sentences. While the blank lines are not needed for creating the lexicon, they are needed for creating probability transition matrices and for part of speech tagging.

  The lexicon is built using both the spelling and the standard spelling (when provided). The lemma is also stored when present.

- *wordlexicon.lex* specifies the name of the output file to receive the word lexicon.
- *suffixlexicon.lex* specifies the name of the output file to receive tthe suffix lexicon.
- *maxsuffixlength* specifies the maximum length suffix generated for the suffix lexicon. The default is 6 characters.
- *maxsuffixcount* specifies the maximum number of times a spelling must appear in order for its suffix to be added to the suffix lexicon. The default is to include all words regardless of count.

  For some applications you may want to restrict the suffix lexicon to contain suffixes only for infrequently occurring words. Values of 10 (only include spellings which appear 10 or less times in the training data) or 1 (only include spellings which appear once in the training data) are popular choices.

Creating a Suffix Lexicon

CreateSuffixLexicon creates a suffix lexicon from a word lexicon.

Usage:

```
createsuffixlexicon inputwordlexicon.lex suffixlexicon.lex
maxsuffixlength maxsuffixcount
```

where

- *inputwordlexicon.lex* specifies the name of an input word lexicon in MorphAdorner format to receive the word lexicon.
- *suffixlexicon.lex* specifies the name of the output file to receive tthe suffix lexicon.
- *maxsuffixlength* specifies the maximum length suffix generated for the suffix lexicon. The default is 6 characters.
- *maxsuffixcount* specifies the maximum number of times a spelling must appear in order for its suffix to be added to the suffix lexicon. The default is to include all words regardless of count.

  For some applications you may want to restrict the suffix lexicon to contain suffixes only for infrequently occurring words. Values of 10 (only include spellings which appear 10 or less times in the training data) or 1 (only include spellings which appear once in the training data) are popular choices.

The suffix lexicon is used by the part of speech taggers to guess the potential parts of speech for unknown words which do not appear in the word lexicon. For each successively shorter ending substring of the unknown word, the guesser looks up that substring in the suffix lexicon. When the substring exists in the suffix lexicon, the guesser assigns its associated parts of speech to the unknown word.

Finding Languages in which a TEI Encoded Text is Written

FindTeiTextLanguage determines the language(s) in which a TEI text is written.

Usage:

```
findteitextlanguage output.tab input1.xml input2.xml ...
```

where

- *output.tab* -- output tab-separated values file described below.
- *input*.xml* -- input TEI XML files whose language is to be found.

The output file is a tab-delimited utf-8 text file containing the following fields, in order.

1. The original XML file name.
2. The length of the plain text from the TEI file, ignoring XML markup, in characters.
3. The most likely language.
4. The language recognizer score for the most likely language.
5. The second most likely language.
6. The language recognizer score for the second most likely language.
7. The third most likely language.
8. The language recognizer score for the third most likely language.

Texts which do not have at least three recognizable languages will have missing language names set to blank with a score of zero.

Language recognizer scores range from 0.0 (not a match) to 1.0 (perfect match). Documents for which the second and third languages achieve non-negligible scores indicate potential problems for processing unless the words in the secondary language are marked up in the TEI document.

Generating Tag Transition Probabilities

NGramTaggerTrainer merges the contents of multiple word list files into a single file. A word list file contains a list of words, one word on each line.

Usage:

```
ngramtaggertrainer trainingdata.tab wordlexicon.lex
transitionmatrix.mat
```

where

- *trainingdata.tab* -- input training data file.
- *wordlexicon.lex* -- input MorphAdorner lexicon.
- *transitionmatrix.mat* -- output tag transition matrix file.

The training data file is a tab-separated utf-8 file containing the part of speech training data generated from the training texts. We only use the first two columns of the training data.

1. The original token (spelling).
2. The NUPOS part of speech.

The word lexicon is a MorphAdorner format word lexicon.

The output tag transition file is a utf-8 file containing the data needed by the MorphAdorner bigram and trigram taggers.

Merging a Brill Lexicon

MergeBrillLexicon merges the contents of a Brill format lexicon with a MorphAdorner format lexicon into a combined MorphAdorner lexicon.

Usage:

```
mergebrilllexicon lexicon.lex brilllexicon.txt
mergedlexicon.lex
```

where

- *lexicon.lex* -- input MorphAdorner format word lexicon.
- *brilllexicon.txt* -- input Brill format word lexicon to be merged with MorphAdorner word lexicon.
- *mergedlexicon.lex* -- output merged lexicon in MorphAdorner format.

A Brill lexicon is a simple utf-8 formatted text file containing words and their possible part of speech tags. Each word appears on a separate line. The first token on each line is the word. The remaining tokens are the potential parts of speech for the word, separated by blanks or tab characters. The most commonly occurring part of speech should be the first one listed.

```
word pos1 pos2 pos3 …
```

This type of lexicon was popularized by Eric Brill's part of speech tagger in the early 1990s.

The Brill entries are merged with the input MorphAdorner lexicon to produce an updated output MorphAdorner format lexicon. The first part of speech for each word is added with a could of two, while the remaining words are added with a count of one. The default English lemmatizer is used to determine lemmata for the Brill words. When a word to be added already exists in the MorphAdorner lexicon, only the new parts of speech are added to the existing lexicon entry.

Brill lexicons are convenient for adding large lists of words such as proper and place names, foreign language words, and so on. Here is a small section of a sample Brill lexicon.

```
Yellott np1
Yellowby np1
Yellville np1
Yelton np1
Yelverton np1
lieu fw-fr
lieux fw-fr
lire fw-fr
lit fw-fr
literary j
livre fw-fr
livres fw-fr
```

MorphAdorner also defines an enhanced Brill lexicon which provides the lemmata for each word's parts of speech. Merging an Enhanced Brill Format Lexicon (page 48) shows how to merge an enhanced Brill lexicon into a MorphAdorner lexicon.

Merging an Enhanced Brill Format Lexicon

MergeEnhancedBrillLexicon merges the contents of an enhanced Brill format lexicon with a MorphAdorner format lexicon into a combined MorphAdorner lexicon.

Usage:

```
mergeenhancedbrilllexicon lexicon.lex
enhancedbrilllexicon.txt mergedlexicon.lex
```

where

- *lexicon.lex* -- input MorphAdorner format word lexicon.
- *enhancedbrilllexicon.txt* -- input enhanced Brill format word lexicon to be merged with MorphAdorner word lexicon.
- *mergedlexicon.lex* -- output merged lexicon in MorphAdorner format.

An enhanced Brill lexicon is a simple utf-8 formatted text file containing words and their possible part of speech tags along with the lemma for each part of speech. Each word appears on a separate line. The first token on each line is the word. The remaining tokens are a a set of pairs of potential parts of speech for the word, followed by a blank, followed by the lemma for that word and part of speech. The most commonly occurring part of speech should be the first one listed.

```
word pos1 lemma1 pos2 lemma2 pos3 lemma3 …
```

This type of lexicon is an enhancement over the simple lexicon format popularized by Eric Brill's part of speech tagger in the early 1990s. The original Brill lexicon did not provide for specifying the lemmata.

The enhanced Brill entries are merged with the input MorphAdorner lexicon to produce an updated output MorphAdorner format lexicon. The first part of speech for each word is added with a could of two, while the remaining words are added with a count of one. When a word to be added already exists in the MorphAdorner lexicon, only the new parts of speech are added to the existing lexicon entry.

Enhanced Brill lexicons are convenient for adding large lists of words such as proper and place names, foreign language words, and so on. Here is a small section of a sample enhanced Brill lexicon.

```
Chippewas np2 Chippewa
mor'n d|cs more|than
quicker'n jc|cs quick|than
y'r po22 you
you'se pn22|vbb you|be
youv'e pn22|vhb you|have
```

Merging a Brill Lexicon (page 47) show how to merge a simple Brill lexicon into a MorphAdorner lexicon. A simple Brill lexicon only provides the list of parts of speech for each word, not the lemmata.

Merging Spelling Data

MergeSpellingData merges the contents of multiple spelling map files into a single spelling map file.

A spelling map file is a utf-8 file containing two fields separated by a tab character. The first field is a variant spelling. The second field is the standardized spelling for the variant.

Usage:

```
mergespellingdata output.tab input.tab input2.tab ...
```

where

- *output.txt* -- output merged spelling map file.
- *input*.txt* -- input text files containing spelling maps to be merged.

Each input spelling map is a utf-8 file contain two fields separated by a tab character. The first field is a variant spelling. The second field is the standardized spelling for the variant.

The output file is a utf-8 text file containing the merged spelling maps from the input files. When a given variant appears more than once with different standardized spellings in the input files, the last mapping encountered is the one written to the output file.

Merging Text Files

MergeTextFiles merges (vertically concatenates) a series of text files into a single output text file.

Usage:

```
mergetextfiles output.txt input.txt input2.txt ...
```

where

- *output.txt* -- output merged text file.
- *input*.txt* -- input text files to be merged.

The output file is a utf-8 text file containing the merged content of the input utf-8 files.

Merging Word Lists

MergeWordLists merges the contents of multiple word list files into a single file. A word list file contains a list of words, one word on each line.

Usage:

```
mergewordlists output.txt input.txt input2.txt ...
```

where

- *output.txt* -- output merged word list file.
- *input*.txt* -- input text files containing word lists to be merged.

The output file is a utf-8 text file containing the merged word list from the input files. Only one copy of a word is output if it appears multiple times. The merged words appear in ascending alphanumeric order in the output file.

Relemmatizing an Adorned File

Relemmatize updates lemmata and standard spellings in MorphAdorned XML files.

Usage:

```
relemmatize lexicon.lex spellingmap.tab
spellingsbywordclass.txt standardspellings.txt
outputdirectory adornedinput.xml adornedinput2.xml ...
```

where

- *lexicon.lex* -- Input MorphAdorner lexicon file.
- *spellingmap.tab* -- Two column tab-separated spelling map file. First column is a variant spelling and the second column is the standard spelling.
- *spellingsbywordclass.tab* -- A spelling map file which breaks down the variant to standard spellings by word class.
- *standardspellings.txt* -- File containing standard known spellings.
- *outputdirectory* -- Output directory for updated MorphAdorner adorned XML files.
- *adornedinput*.xml* -- MorphAdorner adorned XML output files.

The MorphAdorner release provides two specialized versions of the relemmatize command. To relemmatize using the Early Modern English data:

```
relemmatizeeme outputdirectory adornedinput.xml
adornedinput2.xml ...
```

To relemmatize using the Nineteenth Century Fiction data:

```
relemmatizencf outputdirectory adornedinput.xml
adornedinput2.xml ...
```

The lemmata and standard spellings for each adorned word in the input XML files are updated with the most current values. The updated XML files are written to the outputdirectory directory.

The source code for Relemmatize provides an example of reading an adorned XML file and modifying it using a SAX filter.

Running The Link Grammar Parser

LGParser merges the contents of multiple word list files into a single file. A word list file contains a list of words, one word on each line.

Usage:

```
lgparser "sentence text to parse"
```

where "sentence text to parse" is the text of the sentence to parse.

The link grammar parser is a natural language parser based on link grammar theory. Given a sentence, the system assigns to the sentence a syntactic structure consisting of a set of labeled links connecting pairs of words. The parser also produces a "constituent" representation of a sentence (showing noun phrases, verb phrases, etc.).

Sampling Text Files

MorphAdorner provides two utilities for sampling lines from text files: **ExactlySampleTextFile** and **RandomlySampleTextFile**

ExactSampleTextFile usage:

```
exactlysampletextfile input.txt output.txt samplecount
```

where

- *input.txt* -- input text file to be sampled.
- *output.txt* -- output text file.
- *samplecount* -- Size of exact random sample to extract. Must be positive integer.

The output file is a text file containing the sampled text lines from the input file. Both the input and the output must be utf-8 encoded.

RandomlySampleTextFile usage:

```
randomlysampletextfile input.txt output.txt samplingpercent
```

where

- *input.txt* -- input text file to be sampled.
- *output.txt* -- output text file.
- *samplingpercent* -- sampling percent from 0 through 100.

The output file is a text file containing the sampled text lines from the input file. Both the input and the output must be utf-8 encoded.

Stripping Word Attributes

StripWordAttributes creates a derived MorphAdorner XML file with word elements stripped of attributes.

Usage:

```
stripwordattributes input.xml output.xml output.tab [/[no]id]
[/[no]trim]
```

where

input.xml	Input MorphAdorned xml file.
output.xml	Derived adorned file with word element attributes stripped.
output.tab	Tab delimited file of word element attribute values.
/id or /noid	Optional parameter indicating xml:id should be left attached to each word (<w>) element. Default is /noid which removes the xml:id attribute and value.
/trim or /notrim	Optional parameter indicating whether whitespace should be trimmed from the start and end of each XML text line. Default is /notrim, which leaves the original whitespace intact.

The derived adorned output file *output.xml* has all attributes stripped from each <w> tag.

The attribute values for each "<w>" element in the *input.xml* file are extracted and output to the tab-separated values *output.tab* file. The order of the attribute lines matches the order of appearance of the <w> elements in the XML output file. When */id* is specified the *xml:id* value in each <w> element in *output.xml* can be matched with the corresponding *xml:id* value in *output.tab* .

The first line in *output.tab* contains the attribute names for each column. Each subsequent line in the *output.tab* file contains at least the following information corresponding to a single word "<w>" element. Some adorned files may add extra word attributes, resulting in more columns.

1. xml:id -- the permanent word ID.
2. eos -- the end of sentence flag (1 if word ends a sentence, 0 otherwise)
3. lem -- the lemma.
4. ord -- the word ordinal within the text (starts at 1)
5. part -- the word part flag. "N" for a word which is not split; "I" for the first part of a split word; "M" for the middle parts of a split word; and "F" for the final part of a split word.
6. pos -- the part of speech.
7. reg -- the standard spelling.
8. spe -- the corrected original spelling.
9. tok -- The original token.

Training A Part Of Speech Tagger

MorphAdorner requires training data for the part of speech taggers. The training data consists of a utf-8 file containing tab-separated columns. Each input line contains entries corresponding to a single token (spelling, symbol, or punctuation mark) in the training text.

1. The word ID. (Not needed, but helpful.)
2. The original token (spelling).
3. The NUPOS part of speech.
4. The lemma.
5. The standardized spelling.

For some purposes we generate a derived version of the training data without the first column (the word ID).

Creating training data

Normally we generate training data as follows.

1. We MorphAdorn a suitable set of XML texts and adorn them using existing training data. The existing training data is chosen to be consonant in age with the new training texts.
2. The MorphAdorned XML is converted to verticalized tabular form using the **XMLToTab** utility (page 60).
3. We import the verticalized text into a database, spreadsheet, or column-aware editor to correct the initial tagging.
4. We export the corrected verticalized text into a tabular format text file containing the five columns listed above.
5. We run programs which check for various kind of inconsistencies (obviously mismatched parts of speech and lemmata, etc.) and produce a corrected tabular file. Part of this process includes updating the MorphAdorner definitions of the NUPOS parts of speech when new ones appear in the training data.
6. Rinse and repeat these steps until the training data is free of obvious errors.

Here are some of the checks we typically perform.

- Make sure each input line has entries for each of the fields listed above.
- Convert certain XML entity references to unicode characters. For example, the left double quote specification "“" is converted to unicode "\u201C".
- Make sure the part of speech tag for each spelling appears in the list of known NUPOS tags. Unknown tags may be valid but not yet recognized by MorphAdorner.
- Make sure the number of part of speech tags and lemmata matches for each spelling.
- Compile a list of all words marked with the "zz" (unknown) part of speech tag for further review.
- Look for mismatches between punctuation as a token and the part of speech. A punctuation

mark should have itself as its part of speech tag.

- Look for errors that have appeared in the past, such as apparent possessive words ending in "'s" that are marked as adjectives, etc.
- Check that both the spelling and the standard spelling are capitalized for proper nouns. A few proper nouns are legitimately lower case, but they are rare.
- Look for "I" marked with the "z-sy" part of speech. Some of these are legitimate, but some have been erroneously marked in the past.
- Check a list of previously encountered errors and correct them if found. Example: the the part of speech tag is "vbzx" and the lemma is "it|be", change the part of speech tag to "pn31|vbzx".

Updating the lemmatizer

The training data provides lemmata for the spellings in the training data. For spellings not in the training data, the English lemmatizer is used. The English lemmatizer uses a list of rules and a list of exceptions to lemmatize a spelling given a major word class. New training data may indicate the need for new rules or exception list entries.

Creating the lexicons

Once the training data is corrected, it is converted to the format required by the MorphAdorner **CreateLexicon** utility (page 43).

CreateLexicon creates the word and suffix lexicons from the training data. By convention the word lexicon file name takes the form {corpusname}lexicon.lex and the associated suffix lexicon takes the form {corpusname}suffixlexicon.lex .

Normally we want to merge the word lexicon produced from the training data with other word lists such as common Latin and French words, proper person and place names, and so on. These auxiliary word lists will not have frequency information, just part of speech information. For these auxiliary word lists we use the Brill lexicon format, which contains the spelling followed by a list of its possible parts of speech. The **MergeBrillLexicon** utility (page 47) merges a word list in Brill format with a MorphAdorner lexicon.

Brill lexicon entries are added with occurrence frequencies of 1.

The **MergeWordLists** utility (page 51) is helpful in merging Brill lexicons as well as other types of word lists.

Generating probability transition matrices

The bigram and trigram part of speech taggers use a Hidden Markov Model approach to tagging, which requires information about the transition probabilities from one part of speech to another. The **NgramTaggerTrainer** utility (page 46) generates the frequency entries required to compute the transition probabilities.

By convention, the ngram tagger transition matrix data file names take the form {corpusname}transmat.mat extension.

Spelling maps

MorphAdorner's spelling standardizers use a variety of rules and heuristics to map obsolete or variant spellings to standard spellings.

An important part of the spelling standardization process is the creation of the spelling map files. These contain one variant and standard spelling pair per line, separated by a tab character. By convention spelling maps take file names of the form {corpusname}mergedspellings.tab .

Some variant spellings (e.g., bee, doe) take different standard forms depending upon the word class of the original spelling. In addition to the main spelling map, a subsidiary map specifies different standardized spellings for variants depending upon word class.

See page 104 for more information on the spelling map file formats.

Validating XML Files

ValidateXMLFiles validates one or more XML files, optionally against a schema.

Usage:

```
validatexmlfiles [schemaURI] input1.xml input2.xml ...
```

where

- *schemaURI* is an optional URI for a Relax NG or W3C schema against which to validate subsequent files. The schemaURI is treated as a Relax NG schema if it ends in ".rng", and as a W3C schema if it ends in ".xsd". The schema is ignored if it ends in anything else.
- *input*.xml* are the input XML files to validate. At least one file must be specified.

Checks that the specified XML files are valid XML. For XML files referencing a DTD, checks that the XML is valid in the context of the DTD. For XML files that do not specify a DTD, the XML is validated against the optional leading Relax NG or W3C schema. If a schema file is not specified, and the XML document does not specify a DTD, the file will generally be reported as invalid.

Note: ValidateXMLFiles creates a SAX parser for each document (one at a time). This allows even large adorned files to be validated.

Verticalizing an Adorned Text

XMLToTab converts MorphAdorner XML output to tab-separated tabular form.

Usage:

```
xmltotab input.xml output.tab
```

where

- *input.xml* is the input MorphAdorned XML file.
- *output.tab* is the output tab-separated values file.

The attribute values for each **<w>** element in the input XML file are extracted and output to a tab-separated values text file. An output line contains the following information corresponding to a single word **<w>** element.

1. The attribute values for each "<w>" element in the input XML file are extracted and output to a tab-separated values text file. An output line contains the following information corresponding to a single word "<w>" element.
1. The work ID.
2. The permanent word ID.
3. The corrected original spelling.
4. The corrected original spelling reversed.
5. The standard spelling.
6. The lemma.
7. The part of speech.
8. An XPath-like path to this word. The leading work ID and trailing word number are removed from the path.
9. The previous word's original spelling.
10. The next word's original spelling.
11. Up to 80 characters of text preceding the word in the text.
12. Up to 80 characters of text following the word in the text.

This tabular representation of an adorned XML text is useful for data checking purposes. The morphological attribute values for each word <w> element appear as columns. The 80 characters (or so) of text on either side of the word allows you to focus on particular part of speech tags and pinpoint errors from the automatic adornment process. The tab separated values may also be used to construct spreadsheets or databases of the individual word information.

Part Five: Background Information

Language Recognizer

Literary texts are generally composed in one principal language with possible inclusions of short passages (letters, quotations) from other languages. It is helpful to categorize texts by principal language and most prominent secondary language, if any. MorphAdorner includes a simple statistical method based upon character ngrams and rank order statistics to determine the principal language of a text and list possible secondary languages. The method is described in a paper by William B. Cavnar and John M. Trenkle entitled **N-Gram-Based Text Categorization** which appeared in the Proceedings of SDAIR-94, 3rd Annual Symposium on Document Analysis and Information Retrieval. MorphAdorner's implementation follows one written by Frank S. Nestel.

During the Monk project we used this language recognition mechanism to help screen documents that were nominally English but in fact contained large admixtures of unmarked foreign language text. Some examples:

- EEBO document A36803 had an English introduction, a lot of Latin, and a lot of English names. It had a low English score and a non-negligible Latin score. We excluded this from the Monk corpus of EEBO texts.
- EEBO document A57469 had an English title but was classified as primarily French. It turned out to be a legal text with a lot of French and Latin. We also excluded this from the Monk corpus.
- EEBO document A34069 had a low English score (~0.7). It turned out to be an account of a trading voyage containing a lot of Duth interaction.

From these and other experiences we determined that the language recognizer test scores offered a reliable way to identify texts that might contain significant amounts of non-English text in them. The specific language labels were not quite so reliable. For example, French and Latin -- particularly in older texts -- were difficult to distinguish, but they were definitely distinguishable from English. Likewise Scots often appeared as a second choice for English texts. The Scots score was typically higher for older English texts which contain large amounts of old-fashioned variant spellings. In more modern texts a high Scots score often pointed to novels containing swaths of Scots dialect.

You can try MorphAdorner's default language extractor online. This extractor attempts to recognize only the following languages:

- dutch
- english
- french
- german
- italian
- latin
- scots
- spanish
- welsh

English Lemmatizer

Lemmatization is the process of reducing an inflected spelling to its lexical root or **lemma form**. The lemma form is the base form or head word form you would find in a dictionary. The combination of the lemma form with its word class (noun, verb. etc.) is called the **lexeme**.

In English, the base form for a verb is the simple infinitive. For example, the gerund "striking" and the past form "struck" are both forms of the lemma "(to) strike". The base form for a noun is the singular form. For example, the plural "mice" is a form of the lemma "mouse."

Most English spellings can be lemmatized using regular rules of English grammar, as long as the word class is known. MorphAdorner uses a list of about 150 such rules. Some spellings require special handling because they don't follow the general rules. These irregular forms include "strong" verbs like "to catch" and nouns like "mouse." MorphAdorner includes a list of about 1,600 irregular forms.

The lemma form of a spelling depends upon its word class. Thus the noun "bee" has "bee" as a lemma form, while "bee" as a verb has "(to) be" as a lemma form. This turns out to be a bigger problem in Early Modern English than in contemporary English because spelling was not reasonably standardized until the late eighteenth century. Using a standard spelling (page 103) helps in finding the lemma form. For example, the gerund "strykynge" is an old spelling for "striking." By transforming the old spelling to a standardized (usually modern) spelling, we can apply the standard lemmatization rules and obtain "(to) strike" as the lemma. MorphAdorner's English lemmatizer works best with standardized spellings.

Another problem area is the use of the "'s" as a possessive. Sixteenth and seventeenth century English texts generally did not use the "'s" for the possessive form. Thus a phrase like "his majesty's horses" might appear as "his majesties horses." Handling this problem requires part of speech tagging in tandem with spelling standardization.

Not so trivial is the disambiguation of homonyms like 'lie' or 'bark'. There are a few hundred (at most) such pairs in English. In the future we may be able to distinguish which homonym is meant in some situations using methods collectively called *word sense disambiguation*. That would allow more accurate lemmatization for homonyms.

You can read a more detailed description of the English lemmatization process below.

Stemming

Stemming offers a simpler alternative to lemmatization. Stemming also attempts to reduce a word to a base form by removing affixes, but the resulting stem is not necessarily a proper lemma. Such stems can be useful in information retrieval applications.

Two widely used stemmers are included in MorphAdorner.

1. The Porter stemmer, created by Martin Porter.
2. The Lancaster stemmer, created by Chris Paice and Gareth Husk.

You can try MorphAdorner's English lemmatizer online.

English Lemmatization Process

Using a lemma from the word lexicon

Given a (spelling, NUPOS part of speech) pair, MorphAdorner first checks if a lemma appears for that combination in the currently active word lexicon. If so, MorphAdorner returns the lemma specified by the lexicon

Consider the spelling pair *(striking, vvg)*. MorphAdorner's 19th English lexicon defines the lemma *strike* for this combination of spelling and NUPOS part of speech.

Word classes for lemmatization

When the (spelling, part of speech) combination is not found in the current word lexicon, MorphAdorner uses its general English lemmatizer which is based upon a list of irregular forms and grammar rules. The lemmatizer is not tied to a specific part of speech set. Instead the lemmatizer categorizes irregular forms and rules using the following major part of speech classes.

- adjective
- adverb
- compound
- conjunction
- infinitive-to
- noun, plural
- noun, possessive
- preposition
- pronoun
- verb

The NUPOS (or other) part of speech is converted to one of these major word classes for the purposes of lemmatization. In our example above, the NUPOS gerund tag *vvg* maps to the *verb* class. The lemmatizer then processes the spelling pair *(striking, verb)* by first checking the list of irregular forms, and second applying rules of detachment if needed.

Irregular forms

When the spelling pair appears in the irregular forms list, the lemmatizer returns the lemma specified in that list.

In our example, *striking* does not appear on the irregular forms list.

On the other hand, the spelling pair *(mice,noun)* does appear on the irregular forms list, which specifies that *mouse* is the lemma form for *mice*.

Rules of detachment

When the spelling pair does not appear in the irregular forms list, the lemmatizer begins a series of rule matches for the the major word class. Each rule specifies an affix pattern to match and a replacement pattern which generates the lemma form. Once a replacement has been effected, the lemmatization process is complete. These rules are often called *rules of detachment* because the affixes are *detached*

from the inflected word form to produce the lemma form.

In the case of *striking*, the first match occurs against the rule:

```
CVCing CVCe
```

which says "match a consonant, followed by a vowel, followed by a consonant, followed by **ing** at the end of the word." The replacement string says to keep the consonant followed by the vowel followed by the consonant, but replace **ing** with **e** . The result is that *striking* is lemmatized to *strike*.

Some words require the application of multiple sets of detachment rules. For example, the word "astoundingly" is an adverb formed from a present participle. The lemmatizer first applies the adverb rules to remove the "ly" producing "astounding", then applies the verb rules to produce "astound" as the lemma form.

Once a successful substitution occurs, the lemmatization process stops.

Ambiguous endings

The reduced form for some endings is ambiguous. For example, the lemma for the past tense of a verb ending in "ored" may end in "ore" (e.g., implored -> implore) or in "or" (e.g., colored -> color). To help disambiguate such cases, a lemmatization rule can specify that the resulting candidate lemma formed by applying the rule must appear in a known word list. NUPOS uses a large list of standard word forms taken from the 1911 Webster's Dictionary and other sources.

For example, consider the rule sequence:

```
+ ored ore
ored or
```

The first rule says to replace "ored" with "ore" and check that the result is a known word (that's what the "+" denotes). When the result is not a known word, the rule is bypassed, and the following rule which replaces "ored" with "or" is used instead.

Examples:

- recolored -> recolore : recolore not in dictionary, go to next rule.
- recolored -> recolor : recolor in dictionary, accept this form as the lemma.
- implored -> implore : implore in dictionary, accept this form as the lemma.

Words containing multiple parts of speech

Words containing more than one part of speech require special handling. MorphAdorner attempts to split such words at a logical point and assign a separate lemma using the process above to each word part. For example, the spelling *I'm* with a compound NUPOS part of speech *pns11|vam* (the vertical bar separates the parts of speech), is split into two pairs:

- (I,pns11)
- ('m,vam)

The first pair lemmatizes to *i* and the second pair to *be*, giving the compound lemma form *i|be*.

Certain irregular compound forms such as *gimme*, a contraction of "give me", appear under the **compound** entry in the irregular forms list. The lemma form for *gimme* is *give|i*.

Punctuation and Symbols

Punctuation and symbols "lemmatize" to themselves. Foreign words (marked by one of the foreign part of speech tags) and singular nouns are left untouched by MorphAdorner's lemmatizer -- the original spelling is considered the lemma form.

Ambiguous lemmata

The lemma form for some words is ambiguous. For example, "axes" is the plural form of both "axe" and "axis". NUPOS returns one of the possible forms (e.g., "axe" for "axes"). This may not be the correct form in some cases.

Lexicon Lookup

A MorphAdorner word lexicon for a corpus stores all the spellings for words which appear in the corpus, along with the lemmata and parts of speech for each spelling. Each lexicon entry also provides the number of times that spelling appears, both overall as well as broken down by part of speech. MorphAdorner currently provides two English language lexicons, one for Early Modern English, and one for Nineteenth Century Fiction.

MorphAdorner augments the lexicons with auxiliary lists of words which do not appear in the corpus. These include extensive lists of proper names, common foreign words, and combinations of existing words with parts of speech that do not appear in the corpus. These are assigned an "occurrence" count of one. These auxiliary lists improve the ability of MorphAdorner to adorn text with parts of speech and recognize proper names and places.

Lexicon File Format

Lexicon files are plain text files encoded in utf-8 format. Each line in the lexicon file takes the following form:

```
spelling countspelling pos1 lemma1 countpos1 pos2 lemma2
countpos2 ...
```

where

- *spelling* is the spelling for a word,
- *countspelling* is the number of times the spelling appears in the training data,
- *pos1* is the tag corresponding to the most commonly occurring part of speech for this spelling,
- *lemma1* is the lemma form for this spelling, <>li>*countpos1* is the number of times the *pos1* tag appeared, and
- *pos2*, *countpos2*, etc. are the other possible parts of speech and their counts and lemmata.

These fields are separated by tab characters.

The raw counts are stored rather than probabilities so that new training data can be used to update the lexicon easily, and so that individual part of speech taggers can apply different methods of count smoothing.

Following are a few lines from the nineteenth century fiction lexicon.

```
die 1660 vvi die 1164 n1 die 22 vvb die 474
die-away 2 j die-away 2
died 803 vvd die 607 vvn die 196
```

The spelling **died** appears 803 times in the training data. It appears 607 times as the part of speech **vvn** and 196 times as the part of speech **vvn**. Its lemma in both cases is **die**.

When lemmata are not available, an "*' appears in the lemma field. Suffix lexicons contain "*" for all lemmata.

You can try looking up spellings in MorphAdorner's Lexicon lookup online.

MorphAdorner XML Output

MorphAdorner can add word-level morphological adornments to XML texts encoded in two common formats, the Text Encoding Initiative (TEI) format or the Text Creation Partnership (TCP) format. Other XML formats can be accommodated using customized input methods.

MorphAdorner adds XML tags to mark words, punctuation, and whitespace. All other XML tags which appear in the input file are passed through to the output unchanged except for minor reformatting.

TEI Analytics

For the Monk project, all input texts were mapped to a common subset of TEI called *TEI Analytics*, using the Abbott framework developed by Brian Pytlik Zillig and Steve Ramsey at the University of Nebraska. TEI Analytics was jointly developed by Martin Mueller at Northwestern University and Brian Pytlik Zillig and Steve Ramsey at the University of Nebraska. TEI Analytics is the default XML input format assumed by MorphAdorner. TEI Analytics is a minor modification of the P5 TEI-Lite schema, with additional elements from the Linguistic Segment Categories to support morphosyntactic annotation and lemmatization.

XML Tag types: Hard, Soft, and Jump Tags

In order to adorn an XML formatted text properly, MorphAdorner determines the reading context of each word in the input text by constructing the reading sequence for the text. The reading context for a word depends upon the type of XML tag in which it appears as well as the text of its neighboring words.

A *hard tag* is an SGML, HTML, or XML tag which starts a new text segment but does not interrupt the reading sequence of a text. Examples of hard tags include ** ** and **<p>**.

A *jump tag* is an SGML, HTML, or XML tag which interrupts the reading sequence of a text and starts a new text segment. An example of a jump tag is **<note>**. Jump tags initiate a new reading context. The previous reading sequence continues after the end of the jump tag.

A *soft tag* is an SGML, HTML, or XML tag which does not interrupt the reading sequence of a text and does not start a new text segment. Some soft tags provide textual decoration such as **<hi>** and ****. Others indicate textual milestones such as **<milestone>** or formatting such as **<lb>**. Still others mark higher level text segments such as **<rs>**.

The <w> and <c> tags

MorphAdorner uses the **<w>** tag to enclose the text of a word, a symbol, or a punctuation mark, and the **<c>** tag to enclose whitespace.

> This may strike some TEI afficionados as an odd idea. However, treating punctuation and words the same way simplifies processing. Promoting the punctuation "meta-data" added by authors (or editors) to the same level as the words allows a consistent treatment of token transition probabilities for adornment processes such as part of speech tagging.
>
> One alternative might be to drop <w> and use specialized <seg> tags, with types indicating the nature of the enclosed token. But in a sense that is what <w> already does. We could use a sequence such as <w><c>punc</s></w>. as some others do. That seems unnecessarily redundant. We could use <c> with an

> ID, which requires more complicated programming to ensure the sequence of derived words/tokens is correct, since we could no longer pick up non-whitespace tokens by just looking for <w> elements. We would need to extract punctuation from <c> elements as well, and make sure these appear in the right sequence with sibling <w> elements. Whitespace <c> elements would be those without an ID. On balance we believe it is helpful to reserve the <c> element only for whitespace so that the tokens and whitespace are clearly separated by using different tags.

The text enclosed by the **<w></w>** tags is the original token text, which may be a complete word token, or a token fragment when the token text is split by soft or jump tags. Split words are discussed below.

MorphAdorner normalizes the whitespace in input documents, mapping all multiple blanks, tabs, and end of line characters to single blanks. The normalized whitespace is output using the **<c>** tag. Each **<c> </c>** tag pair encloses a single whitespace character.

To prevent output lines from becoming too long, MorphAdorner emits each **<w></w>** tag and each **<c></c>** tag on a separate output line. Most other XML tags are also indented and emitted on separate lines. This "pretty-printing" implies that programs which process the MorphAdorner output should ignore end of line characters and use the contents of the **<c></c>** tags to perform basic text spacing.

> One of the early decisions we made in the Monk project was that the adorned XML files should be more-or-less human readable, although in practice no human being outside of programmers would probably spend much time looking at the texts. That means that each line of output should fit, as much as possible, in the width of a typical computer screen. "Pretty-printing" the XML in this way, with indentation to show structure, introduces a great deal of extra whitespace. It is unreasonable to expect each and every program and programmer to determine what whitespace is part of the "pretty-printing" and what is part of the text. That is why we mark the textual whitespace using <c> to make it unambiguous. Whitespace which is not enclosed in <c> tags can be ignored for purposes of textual analysis or display.

<w> tag attributes

MorphAdorner defines the following attribute fields for each **<w>** tag.

xml:id	Provides a unique id for the token or token fragment. This should be treated as an opaque value. See the section on word IDs below.
ord	Specifies the ordinal of the token, beginning at 1 for the first token. The ordinal is consecutive across all XML tags. MorphAdorner assigns the same ordinal value to all parts of a token split by soft tags since these token fragments appear consecutively in the input file. Tokens split by jump tags may receive different ordinal values for non-consecutive fragments.
eos	A value of "1" indicates this token ends a sentence. A value of "0" indicates this token does not end a sentence. The *eos* value is most accurately set for ordinary text. Tokens within cells or other abbreviated entries may not be marked correctly. See below for an explanation of why we mark end of sentences this way.
lem	Provides the lemma head word form(s) of the token. For punctuation and symbols this is the same as the spelling. For words, this is the base form or head word (uninflected) form you would find in a dictionary. When a word contains more than one lemma, a vertical bar separates the lemma forms.
part	Indicates which part of a split token this token text provides.

	• A value of "N" means the token text is unsplit. • A value of "I" means the token text is the first part of a split token. • A value of "M" means the token text is some part after the first but before the last. • A value of "F" means the token text is the last part of a split token.
pos	The part of speech for the token. By default, MorphAdorner uses the NUPOS part of speech tag set. For symbols and punctuation the part of speech is the same as the token. For words containing more than one part of speech (e.g., contractions), a vertical bar separates the part of speech tags.
reg	A standardized, usually modern, version of the spelling. For obsolete words no longer in use, a representative standard form is chosen which is usually the Oxford English Dictionary headword form.
sn	The sentence number, starting at 1 and running through the text. Cognizant of sentences split by jump tags. Optional, and not used in the Monk project.
spe	The spelling. This value combines the fragments of a split word into the complete spelling. In most cases the *spe* value will match the *tok* value. However, some corpora use special metacharacters in the tokens which are not intended to be part of a word. For example, the TCP/EEBO texts use characters such as the "+" and "|" to mark various kinds of word breaks. The *tok* attribute value retains those metacharacters for archival completeness, but the *spe* value removes them.
tok	The original token text. Includes all metacharacters in the original text. The *tok* value may be a fragment of the complete token when the token text is split by soft or jump tags.
wn	The word number within a sentence, starting at 1. Cognizant of sentences split by jump tags. Optional, and not used in the Monk project.

Word IDs

MorphAdorner assigns a unique word ID to each word token in an adorned file using the *xml:id=* attribute. The principal role of word IDs is to provide a way for different programs to refer to the same words in adorned files. Without word IDs any individual program can still generate its own IDs if needed. However these IDs will differ in each program, rendering it difficult to determine when programs are referring to the same word.

The only property required of a word ID is that it be unique for each word.

MorphAdorner generates unique word IDs that start with the work identifier, taken from the file name of the work, followed by a hyphen, followed by another value which is unique within the work. MorphAdorner can generate two types of values for the within the work part of the ID: either a "reading context order" (the default) or "word within page block".

The "reading context order" appends integer values reflecting the reading context order defined by the classification of hard, soft, and jump tags. This is the default type.

The "word within page block" appends two integer values in the the form pageblocknumber-wordinblock, where pageblocknumber is the ordinal of the current <pb> (page break) entry, and wordinblock is the number of the word within the page block (starting at 1 * spacing). When the text

contains no page break elements, all words appear as part of block 0.

The spacing value provides the increment from one ID value to the next. 10 is the default spacing. Setting the spacing to a value of 10 or 20 (or larger) allows editing programs to interpolate corrections between existing words when the tokenization needs correction. This allows the word IDs to be more stable while the editing process continues. When the spacing is set to 1, adding or removing a word requires a complete resequencing in the case of reading context order IDs or a resequencing of an entire block in the case of word within page block IDs. The resequencing process is not something a human being will do, but is the province of a program such as an editing program, since not only the word IDs but the word ordinals, sentence numbers, and word numbers with sentences will require updating.

The advantage of the "reading context order" type is that a program can extract just the word elements to get the relative position of words and sentences. By sorting words by the word ID it is simple to extract sentences and n-grams without having to worry about hard, jump, and soft tags. The disadvantage is that any change in the tag structure or the classification of tags invalidates the reading context order property (but the word IDs are still valid as unique values).

The advantage of the "word within page block" type is that it provides a basis for displaying a citation position for words. Of course any individual program can generate citations without reference to word IDs, but it may be helpful to have a consistent basis for generating citations. The disadvantage is that each individual program must fully parse the XML and understand the soft, hard, and jump tag structure in order to determine the reading context order so that sentences and n-grams can be extracted.

Numerous tokenization errors remain in many digitized texts. Some errors come from the original digitization. Others come from mistakes introduced by MorphAdorner. Once these tokenization errors have been corrected, the word IDs can be resequenced and citations can be stabilized.

Marking the end of a sentence with the eos= attribute

MorphAdorner uses the *eos=* attribute on the <w> tag for marking a token which ends a sentence. We considered using <milestone> tags to mark sentence, but these present many problems when sentences span jump tags. The same is true of seg-like markers such as <s>.

Using a word-level marker for end of sentence makes it easy to generate sentence information regardless of how one orders the text when dealing with jump tags. For example, Prior and and WordHoard move jump tag content to the end of the work part. That enormously simplifies text display and operations such as collocate extraction. MorphAdorner, when requested to extract sentences, tries to leave the sentences in jump tags as close to their original location in the text. The same word-level flag supports either approach (or other approaches).

Using *sn=* to add sentence numbers is another approach.

Abbreviated attribute output

By default MorphAdorner outputs the full set of **<w>** attributes. MorphAdorner can also output an abbreviated attribute set, in which only non-redundant attribute values appear in the **<w>** tag. This produces smaller output files with no loss of information, since the omitted attribute field values can be restored from those of the other attributes or the token text.

MorphAdorner uses the following algorithm to generate the abbreviated set of **<w>** tag attributes.

1. Let the token-text be the text enclosed within the **<w></w>** tag pair.
2. When *tok* has the same value as the token-text, omit the *tok* attribute.
3. When *spe* has the same value as *tok*, omit the *spe* attribute.
4. When *reg* has the same value as *spe*, omit the *reg* attribute.
5. When *pos* has the same value as *tok*, omit the *pos* attribute.
6. When *lem* has the same value as *spe*, omit the *lem* attribute.
7. When *eos* has the value "0", omit the *eos* attribute.
8. When *part* has the value "N", omit the *part* attribute.

The following algorithm can be used to reconstruct the full set of **<w>** attributes from the abbreviated set.

1. When *tok* is missing, set its value to the text enclosed by the **<w></w>** tags.
2. When *spe* is missing, set its value to the value of *tok*.
3. When *reg* is missing, set its value to the value of *spe*.
4. When pos is missing, set its value to the value of *tok*.
5. When *lem* is missing, set its value to the value of *spe*.
6. When *eos* is missing, set its value to "0" (zero).
7. When *part* is missing, set its value to "N".

The attribute values for *xml:id* and *ord* are always present in either abbreviated or verbose output files.

Split tokens

Individual tokens in XML texts may be split by soft tags, and occasionally by jump tags. MorphAdorner assembles the fragments of a split token into a complete token and sets the *tok* and *spe* attributes of the **<w>** tag for the token fragment to contain the complete token.

The *xml:id* field for a split word adds "dot partnumber" to the end of the **<w>** tag's *xml:id* value. The *xml:id* can still be treated as an opaque object, but the part number can be extracted from the end if desired. In many cases the part number is not needed, and the value of the *part* attribute of the **<w>** tag suffices.

- *part="N"* means the token is unsplit (complete).
- *part="I"* means the token is the first part of a split token.
- *part="M"* means the token is some part after the first but before the last.
- *part="F"* means the token is the last part of a split token.

Here is an example of a split word from Austen's *Lady Susan* (ancf0207.xml). The original XML text is:

Edward S<hi rend="sup(1)">t</hi>.

The "St." token is split into three pieces by soft tags. The corresponding adorned text is:

```
<p rend="align(r)">
  <w eos="0" lem="Edward" pos="np1" reg="Edward"
     spe="Edward" tok="Edward" xml:id="ancf0207-050740" part="N"
     ord="4958">Edward</w>
  <c> </c>
  <w eos="1" lem="saint" pos="n1" reg="St." spe="St." tok="St."
     xml:id="ancf0207-050750.1" part="I" ord="4959">S</w>
  <hi rend="sup(1)">
```

```
     <w eos="1" lem="saint" pos="n1" reg="St." spe="St." tok="St."
       xml:id="ancf0207-050750.2" part="M" ord="4959">t</w>
  </hi>
   <w eos="1" lem="saint" pos="n1" reg="St." spe="St." tok="St."
     xml:id="ancf0207-050750.3" part="F" ord="4959">.</w>
</p>
```

The *ord* attribute value is the same for all three fragments of "St." . This is also the case for words split solely by soft tags. The *ord* attribute values will not be the same for words split by jump tags, as the individual word fragments can be separated by hundreds or even thousands of other words.

Named Entities

MorphAdorner contains an experimental procedure which extends the Gate facility for adding named entity tags to input texts. Each named entity is enclosed by **<rs** *type="named entity type"* **></rs>** tags. The *type=* attribute value specifies the type of the named entity, which may be one of the following.

type="date"	A date reference (e.g., March 12).
type="location"	A geographical location (e.g., England).
type="money"	An amount of money (e.g., 1 shilling).
type="organization"	An organization name (e.g., Bank of England)
type="person"	A person's name (e.g., Emma Woodhouse)
type="time"	A time reference (e.g., 12 midnight)
type="literary"	A literary reference (e.g., Ivanhoe)

Name Recognition

Literary texts are filled with names of people and places. MorphAdorner includes a simple name recognizer for extracting names to allow building lists of characters and geographical settings. MorphAdorner uses a simple noun phrase pattern recognition method to locate probable names in a text. This is not a highly accurate procedure but it provides a useful baseline for further refinement.

MorphAdorner's name extraction process is as follows.

1. Assign parts of speech to each word in the text.
2. Locate noun phrases, e.g., the longest series of nouns bracketed by non-nouns.
3. Assume noun phrases containing at least one proper noun are names.

Distinguishing a personal name from a location isn't so simple. MorphAdorner uses lists of proper names and place names, but there is considerable overlap between these in English. Even a human reader might have trouble determining in some cases whether a name refers to a place or a person. In the sentence "Chester provided arms for the mercenaries", does this refer to the Earl, the county, or another person named Chester? Even in context it might be impossible to be sure which is the referent.

You can try MorphAdorner's default name extractor online which uses the simple noun phrase method described above.

NUPOS and Morphology

This section details Martin Mueller's "NUPOS" part of speech tagset and makes explicit the structure of the tagset and other related morphology objects such as "spellings", "word classes", "lemmata", and "word parts".

As a convention, in this discussion, when we use the term "word", it means "a specific single occurrence of a word somewhere in a text." For the concept of a "word in general", we will use the terms "headword" and "lemma", which we'll define and discuss in detail later.

The full version of NUPOS can handle both Greek and English texts and part of speech tagging. Here we only describe the subset of NUPOS that deals with English. For more information, see Martin Mueller's fuller description at http://panini.northwestern.edu/mmueller/nupos.pdf .

Spellings

The first and most basic attribute of a word is its spelling. This may seem to be a simple concept, but especially for earlier texts from periods before spelling became regularized, it is useful to distinguish among several different meanings of the term "spelling". In NUPOS there are three different "spellings" for each word:

1. The "token spelling". This is the spelling of the word exactly as it appears in the original digital source for the text, including all capitalization and any typographical conventions that might be used in the source as markup for various purposes. For example, the original source for a text might contain a word token "common|lie", where the encoders used the vertical bar character "|" to mark up a soft hyphen at the end of a line. As another example, in some early printed texts, a "y" with a superscript "t" was used to represent the word "that". Such a word might be marked up as "y^t" in the source for such a text. As a final example, the token "@abper;fecit" might appear in the source for an early text. In this example "&abper;" is a symbol used in early typesetting as an abbreviation for "per" or "par".

The token spelling retains as much fidelity as possible with the original digital source. It will often contain various kinds of non-uniform markup, as used by the organizations that digitally encoded the texts. It may be of interest to some researchers, but most people will be more interested in the other two kinds of spellings.

The token spelling may be of importance in contexts where an application wishes to reproduce as much visual fidelity as possible with original printed texts when displaying the text to users.

2. The "standard original spelling". This is a version of the spelling with the typographical conventions normalized, and in most contexts is probably what one thinks of when one uses the general term "the spelling of the word". It is usually identical with the token spelling, but not always. In the examples above, the three tokens become the following "standard original spellings":

```
common|lie --> commonlie
y^t --> that
@abper;fecit --> perfecit
```

3. The "standard modern spelling". This is the standard modern orthographic form of the original spelling. But the morphological form is not modernized. Thus a spelling like "lovyth" is regularized to "loveth". "loveth" is not, however, regularized to "loves", but is rather recognized as a standard archaic

form. In the three examples above, the standard modern spellings are as follows:

```
common|lie --> commonlie --> commonly
y^t --> that --> that
@abper;fecit --> perfecit --> perfecit
```

Note that "perfecit" is a Latin word, and at no point is there an attempt made to translate foreign words into English.

For modern texts, the three spellings are nearly always identical. The main exceptions will be for words in XML texts split by decorator (soft) tags.

Word Parts

Words have spellings, as outlined above. We also want to enumerate and discuss in detail their other tagging attributes, such as word class, part of speech, and lemma. Before we can do this, however, we need to discuss a pesky complexity of texts - contractions.

Consider as an example the first word of *Hamlet*, "Who's". This is a single lexical word, and in this example all three spellings of the word are the same string "Who's".

In terms of the other attributes, however, this word is properly considered to be a lexical representation of the two separate words "who" and "is". Each part has its own word class, part of speech and lemma. In this particular example, it might also be possible to think of each part as having its own spelling or "sub-spelling", "who" and "'s", but in the general case it might be difficult to reasonably split up a spelling into its pieces, and the current version of NUPOS does not attempt to do this.

In NUPOS, this word "who's" is tagged as follows:

word part	major word class	word class	part of speech	lemma
1	wh-word	crq	q-crq	who (crq)
2	verb	va	vbz	be (va)

While we might wish that this complexity didn't exist or could be safely ignored, it can be important when analyzing texts. For example, consider the set of all words in Shakespeare which are instances of the auxiliary verb "be". In NUPOS, the first word of *Hamlet* is correctly included as a member of this set. It is also a member of the set of all words in Shakespeare which are instances of the wh-word "who".

As another example, consider the general notion of counting different kinds of words in Shakespeare. In NUPOS, the count of the total number of occurrences of the auxiliary verb "be" includes the first word of *Hamlet*, as it should, as does the count of the total number of occurrences of the wh-word "who". The first word of *Hamlet* is counted twice, once as "be" and once as "who". Consequently, the sum of the counts of the number of different kinds of words in *Hamlet* is equal to the number of word parts in *Hamlet*, not the number of words.

As a final example, consider an analysis of bigrams in Shakespeare. In NUPOS, the first word of *Hamlet* is considered to be an instance of the bigram "the lemma who (crq) followed by the lemma be (va)", as well as an instance of the bigram "word class crq followed by part of speech vaz".

In the general case, each word, while it usually only has one part, might have more than one part -- two parts in the case of most contractions, but at least conceivably perhaps even more than two parts. While it is words which possess spelling attributes, it is their parts which possess the other morphological attributes, and this is an important distinction to keep in mind.

In the normal case, when a word has only one part, we often use the simple term "word" to refer to its unique part. For example, we say "this word is a verb", when to be precise what we are really saying is "the one and only part of this word is a verb."

Word Classes

In NUPOS, each word part has a "major word class" and a "word class". These concepts provide the coarsest ways to categorize words.

There are 17 major word classes, which should be self-explanatory:

Major word classes
adjective
adv/conj/pcl/prep
adverb
conjunction
determiner
foreign word
interjection
negative
noun
numeral
preposition
pronoun
punctuation
symbol
undetermined
verb
wh-word

Major word classes are subdivided into a slightly finer categorization by "word class". There are 34 word classes in NUPOS:

Name	Description	Major Class
acp	adverb/conjunction/particle/preposition	adv/conj/pcl/prep
an	adverb/noun	noun
av	adverb	adverb
cc	coordinating conjunction	conjunction
crq	wh-word	wh-word

cs	subordinating conjunction	conjunction
d	determiner	determiner
dt	article	determiner
fo	foreign	foreign word
fr	French	foreign word
ge	German	foreign word
gr	Greek	foreign word
it	Italian	foreign word
j	adjective	adjective
jn	adjective/noun	adjective
jp	proper adjective	adjective
la	Latin	foreign word
n	noun	noun
np	proper noun	noun
nu	numeral	numeral
pf	preposition "of"	preposition
pi	indefinite pronoun	pronoun
pn	personal pronoun	pronoun
po	possessive pronoun	pronoun
pp	preposition	preposition
pu	punctuation	punctuation
px	reflexive pronoun	pronoun
sy	symbol	symbol
uh	interjection	interjection
v	verb	verb
va	auxiliary verb	verb
vm	modal verb	verb
xx	negative	negative
zz	undetermined	undetermined

Each word class has a very short string which provides a name for the word class, and each word class belongs to one and only one of the major word classes.

For example, for the major word class "verb", there are three word classes "va" (auxiliary verb), "vm" (modal verb), and "v" (verb). So in NUPOS, there are three kinds of verbs.

Parts of Speech

NUPOS has a fine-grained part of speech tagset, much finer-grained than the word classes and major word classes. There are 241 total English parts of speech in the current version of NUPOS (not counting punctuation).

Each part of speech belongs to one and only one word class, so the part of speech tagset in NUPOS represents a subdivision of the word class tagset, in the same way that the word class tagset represents a subdivision of the major word class tagset.

To continue the example of verbs, in NUPOS each of the verb word classes contains a number of parts of speech:

```
word class va (auxiliary verb): 19 parts of speech
word class vm (modal verb): 14 parts of speech
word class v (verb): 27 parts of speech
```

Each part of speech, in addition to belonging to a word class, is also characterized by, and largely defined by, how it is used in various grammatical categories. These categories and their possible values should be mostly self-explanatory to those familiar with English grammar.

```
Syntax (used as): See below.
Tense: pres, past or empty (not applicable)
Mood: ppl, inf, impt or empty (not applicable)
Case: gen, obj, subj, or empty (not applicable)
Person: 1st, 2nd, 3rd, or empty (not applicable)
Number: sg, pl, or empty (not applicable).
Degree: comp, sup, or empty (not applicable).
Negative: no, nor, not, or empty (not applicable).
```

As an example, the NUPOS part of speech "vmd2" is used for modal verbs used in the second person singular past tense. It has the following attributes in addition to its name "vmd2":

```
word class = vm (modal verb)
syntax = vm
tense = past
mood = empty
case = empty
person = 2nd
number = sg
degree = empty
negative = empty
```

An example of this part of speech occurs in Act 5, Scene 1 of *Hamlet*, where Gertrude says "I hoped thou shouldst have been my Hamlet's wife;" In this passage, the word "shouldst" is tagged with the lemma "shall (vm)" and the part of speech "vmd2". By virtue of this tagging, we know all of the following facts about this word:

```
It is an instance of the headword "shall"
It is a verb.
It is a modal verb.
It has NUPOS part of speech "vmd2".
It is in the past tense.
It is in the second person.
It is singular.
```

In a full implementation of NUPOS, any of these attributes can be used as a criterion for searching, grouping, sorting, counting, and analysis. For example, a researcher might compare the use of past tense modal verbs by one author to their use by another author, or he might do a search where he finds all uses of second person singular verbs in the works of Chaucer. Or he might find all of the verbs used in Spenser and generate a report which counts up how many times each of them are used in the various possible combinations of person and number.

The "syntax" attribute is used to specify how the part of speech is used. For example, the part of speech "av-j" is used for adjectives that are used as adverbs. The "syntax" attribute of this part of speech is "av". An example of this part of speech occurs in Act 1, Scene 1 of *Hamlet*, where Bernardo says "Long live the king!" The word "Long" in this passage in used as an adverb modifying the verb "live" and has the NUPOS part of speech "av-j". Contrast this with the word "long" in Act 3, Scene 1, where Hamlet says "That makes calamity of so long life;". In this passage, the word "long" is tagged with the part of speech "j", the part of speech for "normal" uses of adjectives. Both of the parts of speech "av-j" and "j" have the word class "j" and major word class "adjective", but "av-j" has the syntax attribute "av", while "j" has the syntax attribute "j".

Martin has also mentioned the possibility of more coarse-grained versions of NUPOS, finer grained than word classes but coarser than the full set of 238 parts of speech. These intermediate levels of NUPOS may be useful for data mining and other kinds of analysis. We have not yet worked out the details of this idea.

Another distinctive feature of NUPOS is that it offers some ambiguous wordclasses, like 'jn' for words that hover between noun and adjective or 'an' for words that hover between noun and adverb (home, tomorrow).

All of the NUPOS parts of speech are displayed at the end of this appendix.

Lemmata

A lemma is a dictionary "headword" plus its word class.

For example, consider the verb "love" in Shakespeare. This lemma has the headword "love" and the word class "v". He uses this common lemma in 41 of his 42 works, a total of 1,135 times, in a variety of contexts with quite a few different parts of speech and spellings. For example, he uses it a total of 153 times with the part of speech "vvz", which is the NUPOS part of speech tag for verbs used in the third person singular in the present tense. 150 of these uses are spelled "loves", and three of them are spelled "loveth".

There is, of course, also a noun named "love". In NUPOS, there are two separate lemmata for the headword "love", one for the noun and one for the verb. In general, headwords like "love" are used to form NUPOS lemmata based on their word class, and the word class is listed along with the headword when naming the lemma. In our example, the NUPOS names for the two "love" lemmata are "love (n)" and "love (v)".

The set of all lemmata used in a work or collection of works is called the "lexicon" for the work or collection.

MorphAdorner

MorphAdorner reads source XML texts, locates sentence and word boundaries, and marks each word

with five morphological tags -- the three spellings, the NUPOS part of speech, and the lemma headword. For contractions, MorphAdorner emits multiple parts of speech and headwords.

It's important to recall that MorphAdorner is more than just a part of speech tagger. It's also a spelling normalizer and a lemma tagger.

This tagging data emitted by MorphAdorner is sufficient to recover all of the information mentioned above for each word and word part, including the major word class, word class, part of speech category values, and lemma (headword plus major word class). Note that MorphAdorner only emits the lemma headword. The word class may be deduced from the part of speech.

Following the approach to contracted forms taken by NUPOS, Morphadorner treats contracted forms as a single token for two reasons.

1. The orthographic practice reflects an underlying linguistic reality that the tokenization should respect.
2. In Early Modern English (as in Shaw's orthographic reforms) contracted forms appear without apostrophes, as in 'noot' for 'knows not' or 'niltow' for 'wilt thou not'. It's not obvious how to split these forms. The situation is even less clear for dialectical forms.

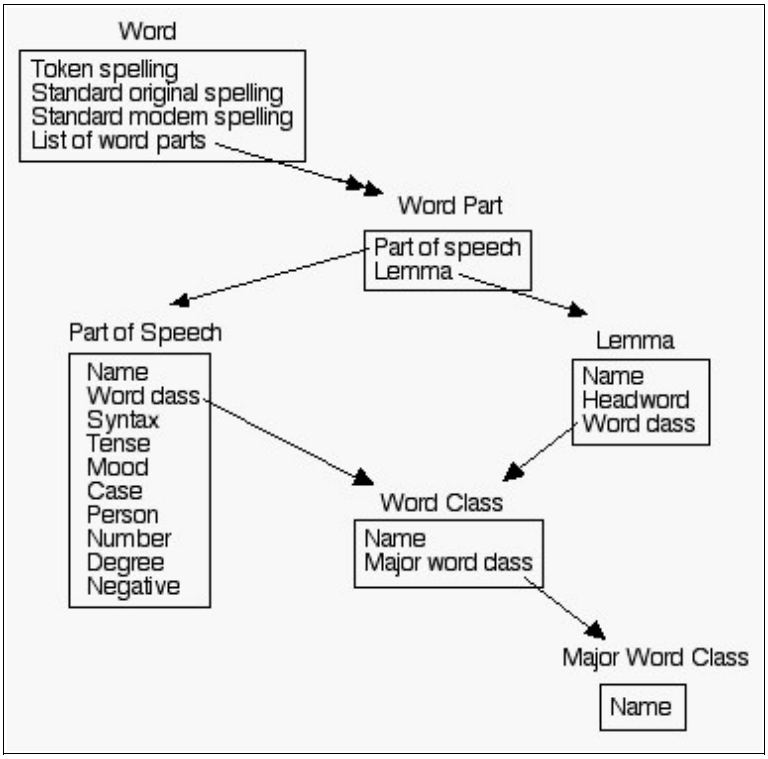
Contracted forms get two part of speech tags separated by a vertical bar, but with regard to forms like "don't', "cannot", "ain't", MorphAdorner analyzes the forms as the negative form of a verb and does not treat the form as a contraction. It uses the symbol 'x' to mark a negative part of speech tag.

Summary

NUPOS comprises the following objects, attributes, and relationships:

- Each word has three spellings: the token, standard original, and standard modern spellings.
- Each word has an ordered list of word parts, usually only one except for contractions.
- Each word part has a part of speech and a lemma.
- Each part of speech has a name, a word class, and values for the grammatical categories of syntax, tense, mood, case, person, number, degree, and negative.
- Each lemma has a name, a headword and a word class. The name of each lemma is formed from its headword and the name of its word class.
- Each word class has a name and a major word class.
- Each major word class has a name.
- In a full implementation of NUPOS, all of these objects and their attributes can be used as criteria for searching, grouping, sorting, counting, and analysis.

The following diagram is useful as a way of summarizing NUPOS. It's not a formal UML diagram, and the drawing has no particular implementation implications, other than as a way of summarizing some of the functionality that any particular full implementation of NUPOS must support. It's just an informal way of making a picture out of the objects, attributes, and relationships enumerated above and described and defined in detail in this note. The double-headed arrow is used to indicate the relationship "may have more than one of", while the single-headed arrow indicates "has one and only one of". The term "list of" in the one-to-many relationship between words and their parts indicates that the parts of a word are ordered -- there's a first one, then a second one, and so on. This is important for dealing with n-grams.



NUPOS for English

The following table lists all the non-punctuation parts of speech defined by NUPOS. The first column provides the NUPOS part of speech tag. The second column describes the tag. The third column offers an example the part of speech. The fourth column provides the count of occurrences of the tag in the NUPOS training data expressed as parts per million. That shows how commonly a tag occurs in the MorphAdorner training data. The training data consists of about six million words drawn from the following texts:

- The following table lists all the non-punctuation parts of speech defined by NUPOS. The first column provides the NUPOS part of speech tag. The second column describes the tag. The third column offers an example the part of speech. The fourth column provides a rounded count of occurrences of the tag in the NUPOS training data expressed as parts per million. That shows how commonly a tag occurs in the MorphAdorner training data. The training data consists of about six million words drawn from the following texts:
- The complete works of Chaucer and Shakespeare
- Spenser's *Faerie Queene*
- North's translation of Plutarch's Lives
- Mary Wroth's *Urania*
- Jane Austen's *Emma*
- Dickens' *Bleak House* and *The Old Curiosity Shop*
- Emily Bronte's *Wuthering Heights*
- Thackeray's *Vanity Fair*

- Mrs. Gaskell's *Mary Barton*
- Frances Trollope's *Michael Armstrong*
- George Eliot's *Adam Bede*
- Scott's *Waverley*
- Harriet Beecher Stowe's *Uncle Tom's Cabin*
- Melville's *Moby Dick*

Examples are chosen for the most part from the training data.

Tag	Explanation	Example	Occurences per million words
a-acp	acp word as adverb	I have not seen him since	9,500
av	adverb	soon	37,500
av-an	noun-adverb as adverb	go home	750
av-c	comparative adverb	sooner, rather	500
avc-jn	comparative adj/noun as adverb	deeper	8
av-d	determiner/adverb as adverb	more slowly	2,000
av-dc	comparative determiner/adverb as adverb	can lesser hide his love	1,900
av-ds	superlative determiner as adverb	most often	900
av-dx	negative determiner as adverb	no more	600
av-j	adjective as adverb	quickly	15,500
av-jc	comparative adjective as adverb	he fared worse	850
av-jn	adj/noun as adverb	duly, right honourable	1,100
av-js	superlative adjective as adverb	in you it best lies	150
av-n1	noun as adverb	had been cannibally given	2
av-s	superlative adverb	soonest	14
avs-jn	superlative adj/noun as adverb	hee being the worthylest constant	0
av-vvg	present participle as adverb	lovingly	250
av-vvn	past participle as adverb	Stands Macbeth thus amazedly	85
av-x	negative adverb	never	1,300
c-acp	acp word as conjunction	since I last saw him	14,000
cc	coordinating conjunction	and, or	42,500
cc-acp	acp word as coordinating conjunction	but	6,500
c-crq	wh-word as conjunction	when she saw	6,500
ccx	negative conjunction	nor	1,200
crd	numeral	2, two, ii	5,700
cs	subordinating conjunction	if	6,500

cst	'that' as conjunction	I saw that it was hopeless	14,000
d	determiner	that man, much money	29,500
dc	comparative determiner	less money	850
dg	determiner in possessive use	the latter's	7
ds	superlative determiner	most money	450
dt	article	a man, the man	7,000
dx	negative determiner as adverb	no money	2,500
fw-fr	French word	monsieur	500
fw-ge	German word	Herr	15
fw-gr	Greek word	kurios	15
fw-it	Italian word	signor	10
fw-la	Latin word	dominus	400
fw-mi	word in unspecified other language	n/a	50
j	adjective	beautiful	49,500
j-av	adverb as adjective	the then king	1
jc	comparative adjective	handsomer	1,500
jc-jn	comparative adj/noun	yet she much whiter	70
jc-vvg	present participles as comparative adjective	for what pleasinger then varietie, or sweeter then flatterie?	1
jc-vvn	past participle as comparative adjective	shall find curster than she	1
j-jn	adjective-noun	the sky is blue	7,000
jp	proper adjective	Athenian philosopher	800
js	superlative adjective	finest clothes	1,500
js-jn	superlative adj/noun	reddest hue	200
js-vvg	present participle as superlative adjective	the lyingest knave in Christendom	2
js-vvn	past participle as superlative adjective	deformed'st creature	3
j-vvg	present participle as adjective	loving lord	2,000
j-vvn	past participle as adjective	changed circumstances	2,500
n1	singular, noun	child	14,000
n1-an	noun-adverb as singular noun	my home	250
n1-j	adjective as singular noun	a good	4
n2	plural noun	children	35,000
n2-acp	acp word as plural noun	and many such-like "As'es" of great charge	1

n2-an	noun-adverb as plural noun	all our yesterdays	9
n2-av	adverb as plural noun	and are etcecteras no things	1
n2-dx	determiner/adverb negative as plural noun	yeas and honest kerysey noes	0
n2-j	adjective as plural noun	give me particulars	200
n2-jn	adj/noun as plural noun	the subjects of his substitute	600
n2-vdg	present participle as plural noun, 'do'	doings	50
n2-vhg	present participle as plural noun, 'have'	my present havings	1
n2-vvg	present participle as plural noun	the desperate languishings	200
n2-vvn	past participle as plural noun	there was no necessity of a Letter of Slains for Mutilation	0
ng1	singular possessive, noun	child's	2,500
ng1-an	noun-adverb in singular possessive use	Tomorrow's vengeance	6
ng1-j	adjective as possessive noun	the Eternal's wrath	1
ng1-jn	adj/noun as possessive noun	our sovereign's fall	60
ng1-vvn	past participle as possessive noun	the late lamented's house	0
ng2	plural possessive, noun	children's	350
ng2-jn	adj/noun as plural possessive noun	mortals' chiefest enemy	50
n-jn	adj/noun as noun	a deep blue	2,300
njp	proper adjective as noun	a Roman	130
njp2	proper adjective as plural noun	The Romans	1,300
njpg1	proper adjective as possessive noun	The Roman's courage	8
njpg2	proper adjective as plural possessive noun	The Romans' courage	20
np1	singular, proper noun	Paul	27,500
np2	plural, proper noun	The Nevils are thy subjects	350
npg1	singular possessive, proper noun	Paul's letter	2,600
npg2	plural possessive, proper noun	will take the Nevils' part	6
np-n1	singular noun as proper noun	at the Porpentine	260
np-n2	plural noun as proper noun	such Brooks are welcome to me	2
np-ng1	singular possessive noun as proper noun	and through Wall's chink	20
n-vdg	present participle as noun, 'do'	my doing	20
n-vhg	present participle as noun, 'have'	my having	0
n-vvg	present participle as noun	the running of the deer	1,500

n-vvn	past participle as noun	the departed	50
ord	ordinal number	fourth	2,500
p-acp	acp word as preposition	to my brother	57,000
pc-acp	acp word as particle	to do	19,000
pi	singular, indefinite pronoun	one, something	2,200
pi2	plural, indefinite pronoun	from wicked ones	50
pi2x	plural, indefinite pronoun	To hear my nothings monstered	2
pig	singular possessive, indefinite pronoun	the pairings of one's nail	35
pigx	possessive case, indefinite pronoun	nobody's	2
pix	indefinite pronoun	none, nothing	1,300
pn22	2nd person, personal pronoun	you	9,000
pn31	3rd singular, personal pronoun	it	10,500
png11	1st singular possessive, personal pronoun	a book of mine	220
png12	1st plural possessive, personal pronoun	this land of ours	35
png21	2nd singular possessive, personal pronoun	this is thine	3
png22	2nd person, possessive, personal pronoun	this is yours	100
png31	3rd singular possessive, personal pronoun	a cousin of his	200
png32	3rd plural possessive, personal pronoun	this is theirs	30
pno11	1st singular objective, personal pronoun	me	5,000
pno12	1st plural objective, personal pronoun	us	1,100
pno21	2nd singular objective, personal pronoun	thee	1,200
pno31	3rd singular objective, personal pronoun	him, her	12,000
pno32	3rd plural objective, personal pronoun	them	4,700
pns11	1st singular subjective, personal pronoun	I	14,500
pns12	1st plural subjective, personal pronoun	we	2,200

pns21	2nd singular subjective, personal pronoun	thou	2,000
pns31	3rd singular subjective, personal pronoun	he, she	21,000
pns32	3rd plural objective, personal pronoun	they	5,600
po11	1st singular, possessive pronoun	my	6,700
po12	1st plural, possessive pronoun	our	1,400
po21	2nd singular, possessive pronoun	thy	1,650
po22	2nd person possessive pronoun	your	3,000
po31	3rd singular, possessive pronoun	its, her, his	19,000
po32	3rd plural, possessive pronoun	their	3,800
pp	preposition	in	23,000
pp-f	preposition 'of'	of	29,000
px11	1st singular reflexive pronoun	myself	350
px12	1st plural reflexive pronoun	ourselves	55
px21	2nd singular reflexive pronoun	thyself, yourself	250
px22	2nd plural reflexive pronoun	yourselves	30
px31	3rd singular reflexive pronoun	herself, himself, itself	1,300
px32	3rd plural reflexive pronoun	themselves	220
pxg21	2nd singular possessive, reflexive pronoun	yourself's remembrance	1
q-crq	interrogative use, wh-word	Who? What? How?	3,000
r-crq	relative use, wh-word	the girl who ran	10,000
sy	alphabetical or other symbol	A, @	50
uh	interjection	oh!	3,000
uh-av	adverb as interjection	Well!	300
uh-crq	wh-word as interjection	Why, there were but four	500
uh-dx	negative interjection	No!	500
uh-j	adjective as interjection	Grumio, mum!	7
uh-jn	adjective/noun as interjection	And welcome, Somerset	30
uh-n	noun as interjection	Soldiers, adieu!	200
uh-v	verb as interjection	My gracious silence, hail	90
vb2	2nd singular present of 'be'	thou art	300
vb2-imp	2nd plural present imperative, 'be'	Beth pacient	10
vb2x	2nd singular present, 'be'	thow nart yit blisful	2
vbb	present tense, 'be'	are, be	3,300

vbbx	present tense negative, 'be'	aren't, ain't, beant	60
vbd	past tense, 'be'	was, were	14,000
vbd2	2nd singular past of 'be'	thou wast, thou wert	50
vbd2x	2nd singular past, 'be'	weren't	0
vbdp	plural past tense, 'be'	whose yuorie shoulders weren couered all	30
vbdx	past tense negative, 'be'	wasn't, weren't	75
vbg	present participle, 'be'	being	1,300
vbi	infinitive, 'be'	be	5,600
vbm	1st singular, 'be'	am	1,200
vbmx	1st singular negative, 'be'	I nam nat lief to gabbe	3
vbn	past participle, 'be'	been	1,800
vbp	plural present, 'be'	Thise arn the wordes	260
vbz	3rd singular present, 'be'	is	6,900
vbzx	3rd singular present negative, 'be'	isn't	100
vd2	2nd singular present of 'do'	dost	150
vd2-imp	2nd plural present imperative, 'do'	Dooth digne fruyt of Penitence	6
vd2x	2nd singular present negative, 'do'	thee dostna know the pints of a woman	2
vdb	present tense, 'do'	do	1,600
vdbx	present tense negative, 'do'	don't	500
vdd	past tense, 'do'	did	3,100
vdd2	2nd singular past of 'do'	didst	55
vdd2x	2nd singular past negative, verb	Why, thee thought'st Hetty war a ghost, didstna? 0.20	
vddp	plural past tense, 'do'	on Job , whom that we diden wo	3
vddx	past tense negative, 'do'	didn't	90
vdg	present participle, 'do'	doing	110
vdi	infinitive, 'do'	to do	1,000
vdn	past participle, 'do'	done	700
vdp	plural present, 'do'	As freendes doon whan they been met	30
vdz	3rd singular present, 'do'	does	800
vdzx	3rd singular present negative, 'do'	doesn't	20
vh2	2nd singular present of 'have'	thou hast	250
vh2-imp	2nd plural present imperative, 'have'	O haveth of my deth pitee!	1

vh2x	2nd singular present negative, 'have'	hastna	0
vhb	present tense, 'have'	have	2,500
vhbx	present tense negative, 'have'	haven't	30
vhd	past tense, 'have'	had	6,000
vhd2	2nd singular past of 'have'	thou hadst	35
vhdp	plural past tense, 'have'	Of folkes that hadden grete fames	10
vhdx	past tense negative, 'have'	hadn't	20
vhg	present participle, 'have'	having	730
vhi	infinitive, 'have'	to have	2,400
vhn	past participle, 'have'	had	220
vhp	plural present, 'have'	They han of us no jurisdiccioun,	120
vhz	3rd singular present, 'have'	has, hath	1,700
vhzx	3rd singular present negative, 'have'	Ther loveth noon, that she nath why to pleyne.	11
vm2	2nd singular present of modal verb	wilt thou	360
vm2x	2nd singular present negative, modal verg	O deth, allas, why nyltow do me deye	4
vmb	present tense, modal verb	can, may, shall, will	8,300
vmb1	1st singular present, modal verb	Chill not let go, zir, without vurther 'cagion	3
vmbx	present tense negative, modal verb	cannot; won't; I nyl nat lye	700
vmd	past tense, modal verb	could, might, should, would	8,300
vmd2	2nd singular past of modal verb	couldst, shouldst, wouldst; how gret scorn woldestow han	120
vmd2x	2nd singular present, modal verb	Why noldest thow han writen of Alceste	5
vmdp	plural past tense, modal verb	tho thinges ne scholden nat han ben doon	30
vmdx	past negative, modal verb	couldn't; She nolde do that vileynye or synne	160
vmi	infinitive, modal verb	Criseyde shal nought konne knowen me.	5
vmn	past participle, modal verb	I had oones or twyes ycould	2
vmp	plural present tense, modal verg	and how ye schullen usen hem	25
vv2	2nd singular present of verb	thou knowest	480
vv2-imp	2nd present imperative, verb	For, sire and dame, trusteth me right weel,	80
vv2x	2nd singular present negative, verb	"Yee!" seyde he, "thow nost what	1

		thow menest;	
vvb	present tense, verg	they live	17,000
vvbx	present tense negative, verb	What shall I don? For certes, I not how	30
vvd	past tense, verb	knew	33,000
vvd2	2nd singular past of verb	knewest	75
vvd2x	2nd singular past negative, verb	thou seidest that thou nystist nat	0
vvdp	past plural, verb	They neuer strouen to be chiefe	80
vvdx	past tense negative, verb	she caredna to gang into the stable	10
vvg	present participle, verb	knowing	13,700
vvi	infinitive, verb	to know	36,000
vvn	past participle, verb	known	26,200
vvp	plural present, verb	Those faytours little regarden their charge	330
vvz	3rd singular preseent, verb	knows	7,200
vvzx	3rd singular present negative, verb	She caresna for Seth.	1
xx	negative	not	7,800
zz	unknown or unparsable token	n/a	200

Parser

MorphAdorner includes a Java port of the Carnergie Mellon University link grammar parser, a syntactic parser for English. The link grammar parser is a natural language parser based on link grammar theory. Given a sentence, the system assigns to the sentence a syntactic structure consisting of a set of labeled links connecting pairs of words. The parser also produces a "constituent" representation of a sentence (showing noun phrases, verb phrases, etc.). More information is available at: http://www.link.cs.cmu.edu/link/.

Note that the parser uses the Penn Treebank part of speech tag set, not the NUPOS tag set.

You can try MorphAdorner's link grammar parser online.

Part of Speech Tagging

Part of speech tagging is the process of adorning or "tagging" words in a text with each word's corresponding part of speech. Part of speech tagging is based both on the meaning of the word and its positional relationship with adjacent words. A simple list of the parts of speech for English includes adjective, adverb, conjunction, noun, preposition, pronoun, and verb. For computational purposes, however, each of these major word classes is usually subdivided to reflect more granular syntactic and morphological structure.

MorphAdorner can adorn each spelling in a text with a part of speech. To do this MorphAdorner requires a definition of the part of speech tag set, and a training corpus containing a large swatch of text containing spellings already correctly adorned with their parts of speech. From this training data MorphAdorner can generate tagging rules, tag probability matrices, and a lexicon of known words.

MorphAdorner provides several different part of speech taggers. We expect only two will be widely used.

- The MorphAdorner trigram tagger uses a hidden Markov model and a beam-search variant of the Viterbi algorithm. We expect this will be the primary tagger. You can read a brief description of mathematical basis of the trigram tagger on page 97.
- The MorphAdorner rule-based tagger is a modified version of Mark Hepple's rule-based tagger. Hepple's tagger is a variant of Eric Brill's tagger but disallows interaction between rules. We expect the Hepple tagger to be used as a secondary tagger to correct the output of the trigram tagger.

The MorphAdorner part of speech taggers assign tags to unknown words using pattern recognition for items such as numbers and Roman numerals, and suffix analysis with successive abstraction when the pattern recognition methods fail. For example, the suffix **ly** in English often indicates the word is an adverb, while the suffix **ing** often indicates the word is a gerund (an obvious counterexample is "spring"). By looking at the statistical distribution of endings and part of speech tags in the training data, along with the sequence of previous parts of speech, MorphAdorner can often guess correctly the part of speech for a word it doesn't know. When all the pattern recognition methods fail, the word is assumed to be a noun.

You can see a detailed list of the pattern recognition methods MorphAdorner uses to assign parts of speech to unknown words on page .

Part of speech tagging of English texts from the Early Modern English period to the present raises several problems. Most part of speech tag sets for English were devised for use with modern texts. These tag sets lack the necessary tags to represent English usage that was either current at an earlier time or was archaic at its time of origin but remained current in restricted discursive environments, such as religion or poetry. The second person singular of pronouns and verb forms is the clearest example. An **-n** form that marks a plural present is much rarer but not uncommon as a deliberate archaism in Shakespeare's time.

Modern taggers rely on **'s** or **s'** to identify the possessive case. They also rely on sentence medial capitalization to extract names. These procedures don't work once you move back to the 18th century.

By default MorphAdorner uses a part of speech tag set designed by Martin Mueller. NUPOS (see page 76) as it is called, differs from modern tag sets in recognizing all morphological forms that are found in

written English from Chaucer to the present. Like the tag set used for the Brown corpus but unlike the Penn Treebank or CLAWS tag sets, NUPOS does not split the possessive case as a separate token and uses compound tags for contracted forms.

Part of speech tags tend to be somewhat inconsistent compounds of syntactic and morphological information. In NUPOS the components of each tag are kept separately and the grammatical description of each word can be easily identified at a minimal level of granularity (~20 tags) or at a maximum level (~230 tags).

MorphAdorner can use any arbitrary tag set given appropriate training data and a proper definition of the word class and major word class of each tag.

The Trigram tagger assigns the part of speech tag correctly about 96% to 97% of the time. The accuracy can be expected to improve as the training lexicon grows.

You can try MorphAdorner's trigram part of speech tagger online. This example only accepts plain text as input.

Guessing Parts of Speech for Unknown Words

A program like MorphAdorner assigns a part of speech tag to each token in an input text, e.g., this word token is a noun or this token is a period. This task is difficult since many words can take on more than one part of speech. Determining which part of speech applies to a particular word occurrence depends upon the context in which the word appears.

A set of training data specifies a large number of words along with their potential parts of speech in actual reading contexts. This combination of known words and parts of speech, along with statistical methods and/or context rules, allows a program like MorphAdorner to assign correct parts of speech to words in new texts about 97% of the time, as long as all the words in the new texts are known. That is, the words have been encountered in the training data with all their possible parts of speech, or the words appear in supplemental dictionaries along with their parts of speech.

Unfortunately many words in new texts will not have been seen in the training data and will not occur in a supplemental dictionary. This means a program like MorphAdorner must "guess" the relevant possible parts of speech for an unknown word to assign a proper part of speech tag in context.

MorphAdorner uses a variety of techniques to guess the possible parts of speech for an unknown word. The default MorphAdorner guesser applies the following methods, in order, until at least one potential part of speech is identified. A programmer can modify or replace this default guesser, and several MorphAdorner configuration settings allow you to modify the guessing process as well.

1. **Is the word punctuation?**

   Examples: period, quote mark, question mark, sequence of periods

   Assign the punctuation or punctuation class as the part of speech.

2. **Is the word a symbol?**

   Examples: A paragraph mark.

   Assign the symbol class as the part of speech.

3. **Is the word a cardinal number?**

Examples: 12, 12.5

Assign the cardinal number class as the part of speech.

4. **Is the word an ordinal number?**

   Examples: 1st, 12th

   Assign the ordinal number class as the part of speech.

5. **Is the word a currency amount?**

   Examples: $12.50, 1L, 1Ã‚£, Ã‚£10

   Assign the cardinal number class as the part of speech.

6. **Is the word a Roman numeral?**

   Examples: I, V, IX, .IX., .IX, MMM, IIIJ

   Assign the cardinal number class as the part of speech. For Roman numerals that can also be initials (I, V) or English pronouns (I), add the proper noun and appropriate pronoun classes as well.

   Note that the definition of a Roman numeral is much looser in older texts than is defined in contemporary usage.

7. **Is the word an ordinal Roman numeral?**

   Examples: xviith

   Assign the ordinal number part of speech class.

8. **Is the word hyphenated?**

   Examples: head-master, sea-serpent

   MorphAdorner extracts the part of the word after the last hyphen. If that is a known word, assign its part of speech classes.

   The following cases are treated specially.

   - a letter followed by ---*'s* is considered a possessive noun.
   - ---*'s* or ---*'S* is considered a possessive noun.
   - a letter followed by --- is considered a proper or common possessive noun, or an exclamation.

9. **Is a spelling standardizer defined?**

   If so, get the parts of speech for the standardized spelling.

   Example: "vniversitie" regularizes to "university"

   Assign the part of speech classes for "university" if known.

10. **Is the word a proper name?**

    MorphAdorner defines some auxiliary word lists containing lists of proper names for people and places. If the word appears on one of these "name" lists, assign the proper noun class.

11. **Is the word defined by an auxiliary word list?**

MorphAdorner defines some auxiliary word lists which define words and possible part of speech classes for those words. If the word appears on one of these lists, assign the associated part of speech classes defined in the lists.

12. **Is the word an abbreviation?**

    Examples: U.S., p.m.

    If the word appears to be an abbreviation, assign a proper noun class if it begins with a capital letter, or a common noun class if it does not begin with a capital letter.

13. **Is a suffix lexicon defined?**

    If so, perform the following suffix analysis.

    For each successively shorter ending substring of the word, look up that substring in the suffix lexicon. If the substring exists in the suffix lexicon, assign its part of speech classes as those of the unknown word.

    Example: reputedly

    Look up the successively shorter terminal strings:

    reputedly
    eputedly
    putedly
    utedly
    tedly
    edly
    dly
    ly
    y

    and stop at the first of those suffix strings which appears in the suffix lexicon, and use the associated part of speech classes.

14. **Is the word entirely in upper case?**

    Example: MCDOODLE

    Assign the singular proper noun part of speech class.

15. **If all else fails, assume the word is a noun.**

    If the word begins with a capital letter and ends with "s", assume it is a plural proper noun.

    If the word begins with a capital letter and does not end with "s", assume it is a singular proper noun.

    If the word does not begin with a capital letter and ends with "s", assume it is a plural common noun.

    If the word does not begin with a capital letter and does not end with "s", assume it is a singular common noun.

Trigram Tagger Mathematical Background

Assume that the part of speech tag for a word depends only upon the previous one or two tags, and that the probability of this tag does not depend upon the probabilities of subsequent tags. How do we find the most probable sequence of tags corresponding to a particular sequence of words? We can look at the sequence of part of speech tags for words as an instance of a Hidden Markov Model HMM). Finding the most probable part of speech tag sequence amounts to finding the most probable sequence of states which the Hidden Markov Model traverses for a particular sequence of words.

The Trigram Tagger in MorphAdorner seeks the most likely part of speech tag for a word given information about the previous two words. More precisely, the tag sequence t_1, t_2, ..., t_n -- corresponding to the word sequence w_1, w_2, ..., w_n -- is sought which maximizes the following expression:

$$P(t_1)P(t_2|t_1)\text{PRODUCT(i=3 to n)}\ P(t_i|t_{i-2},t_{i-1})\text{PRODUCT(i=1 to n)}\ P(w_i|t_i)$$

where

P(t)	Contextual (tag transition) probability
P(w|t)	Lexical (word emission) probability

Each $P(w_i|t_i)$ is estimated using the maximum likelihood estimator:

$$P_{MLE}(t_i|t_{i-2},t_{i-1}) = C(t_{i-2},t_{i-1},t_i) \ / \ C(t_{i-2},t_{i-1})$$

where C(x) is the observed single or joint frequency for the words or tags. To account for "holes" in the frequencies, where some possible combinations are not observed, we can compute smoothed probabilities which reduce the maximum likelihood estimates a little bit to allow a bit of the overall probability to be assigned to unobserved combinations. By default MorphAdorner uses a simple additive smoother which adds small positive values to the numerator and denominator of the probability estimate above. The numerator value is a small constant such as 0.05, while the denominator value is the numerator constant multiplied by the size of the word lexicon *L*.

$$P_{smoothed}(t_i|t_{i-2},t_{i-1}) = (C(t_{i-2},t_{i-1},t_i) + 0.05) \ / \ (C(t_{i-2},t_{i-1}) + (0.05 * L))$$

This works well when the training data size is large (as it is for MorphAdorner). For smaller training sets, deleted interpolation may prove more effective. MorphAdorner provides a deleted interpolation smoother as an option.

Each $P(w_i|t_i)$ is estimated using the maximum likelihood estimator:

$$P_{MLE}(w_i|t_i) = C(w_i\ ,\ t_i) \ / \ C(t_i)$$

Again, MorphAdorner smooths the maximum likelihood estimates using additive smoothing. This time 0.5 is used as the additive numerator value, and the denominator multiplier is K, the number of

potential part of speech tags for the word w_i. This is commonly called Lidstone smoothing.

$$P_{smoothed}(w_i|t_i) = (C(w_i , t_i) + 0.5) / (C(t_i) + (0.5 * K))$$

It is possible but inefficient to calculate the probabilities using complete enumeration of all possible values of t_1, t_2, ..., t_n. In the worst case the time complexity is n^T where *T* is the number of part of speech tags in the tag set. MorphAdorner applies the Viterbi algorithm, a dynamic programming algorithm to determine the optimal subpaths for each state in the HMM model as the algorithm traverses the model. Subpaths other than the most probable are discarded. MorphAdorner also restricts the search path even further using a beam search which discards paths whose probabilities are too small compared to a specified tolerance value to contribute significantly to the joint probability.

Pluralizer

Pluralization is the process of *inflecting* a singular noun by adding affixes or changing certain letters in the singular noun form to give the plural form.

The plural form of many nouns in English can be formed as follows.

1. Add "s" to the singular form to form the plural. Example: dog -> dogs.
2. Add "es" for singular nouns ending in a sibilant sound. Example: dress -> dresses.
3. Change the terminal "y" to "i" and then add "es" when the singular noun ends in "y" not preceded by a vowel (or is not a proper name). Example: spy -> spies.

Unfortunately there are many English nouns whose plurals do not follow the rules above. A description may be found in Damien Conway's paper at An Algorithmic Approach to English Pluralization . MorphAdorner implements Conway's pluralization procedure for nouns. MorphAdorner always produces non-classical plurals.

You can try MorphAdorner's English noun pluralizer online.

Sentence Splitting

Extracting words and sentences from a text are fundamental operations required by other language processing functions. Word Tokenization (see page 111) splits a text into words and punctuation marks. **Sentence splitting** assembles the tokenized text into sentences.

Recognizing the end of a sentence is not an easy task for a computer. In English, punctuation marks that usually appear at the end of a sentence may not indicate the end of a sentence. The period is the worst offender. A period can end a sentence but it can also be part of an abbreviation or acronym, an ellipsis, a decimal number, or part of a bracket of periods surrounding a Roman numeral. A period can even act both as the end of an abbreviation and the end of a sentence at the same time. Other the other hand, some poems may not contain any sentence punctuation at all.

Another problem punctuation mark is the single quote, which can introduce a quote or start a contraction such as **'tis**. Leading-quote contractions are uncommon in contemporary English texts, but appear frequently in Early Modern English texts.

Few literary texts which have already been marked up using SGML or XML recognize sentences in the markup. (The Chadwick-Healey archive of eighteenth century novels is a notable counterexample.) Sentences often cross other element boundaries. Texts without sentence markup require preprocessing to add it without disturbing the existing markup. This allows further processing of the texts, in particular, part of speech tagging, and name recognition. MorphAdorner allows pluggable input and output processors to handle reification of texts and addition of extra markup as needed.

MorphAdorner's default sentence splitter uses the ICU4JBreakIterator class along with a set of heuristics (see below) for determining if two or more sentences generated by ICU4JBreakIterator should be joined into one sentence. The heuristics include special treatment of sentence-ending brackets (right parenthesis, right bracket, and right brace), abbreviations, and interjections. The resulting sentence extraction is not perfect but is better than ICU4JBreakIterator's splitting and much better than naive splitting methods.

You can try MorphAdorner's default sentence splitter online. This example only demonstrates sentence splitting for plain text.

Sentence Splitter Heuristics

The article Finding text boundaries in Java by Rich Gillam describes the Java BreakIterator which underlies the ICU4JBreakIterator class used by MorphAdorner to obtain an initial deconstruction of text into sentences. MorphAdorner only uses ICU4JBreakIterator to provide initial sentence boundaries. MorphAdorner's word tokenizer uses its own methods for determining token boundaries within a sentence.

Abbreviations

The period ending an abbreviation may act as both a part of the abbreviation and the end of a sentence. MorphAdorner maintains a list of common abbreviations along with a flag indicating if the abbreviation usually can end a sentence. MorphAdorner will not split a sentence after an abbreviation which is not designated as a potential sentence ender.

For example, the abbreviation *Mrs.* rarely ends a sentence, so MorphAdorner does not issue sentence

splits following *Mrs.* Thus

> Mrs. Smith was here earlier.

is correctly considered a single sentence, while

> I will leave it up to the Mrs. She will know what to do.

which should be two sentences (with a split after *Mrs.*) is also treated as a single sentence by MorphAdorner. This could be handled by recognizing that *Mrs.* can end a sentence when followed by something other than a proper name.

When an abbreviation can end a sentence, MorphAdorner tries to determine if a particular use ends a sentence or not by looking for possible verbs before and after the abbreviation. MorphAdorner does not split the sentence after the abbreviation unless it has found a possible verb in the sentence preceding the abbreviation. MorphAdorner does not use detailed part of speech information during sentence splitting. However, the parts of speech for any word can be looked up in the word lexicon or determined using a part of speech guesser. That is sufficient to guide the sentence splitting algorithm in many but not all cases.

MorphAdorner splits the text

> I mailed the letter early in the a.m. The next step is to wait for a reply.

correctly into two sentences following *a.m.*, while

> I mailed the letter early in the a.m. the next day too.

is left unsplit.

MorphAdorner correctly leaves unsplit the following sentences.

> She needs her car by 5 p.m. Saturday evening.
> At 5 p.m. I had to go to the bank.
> She has an appointment at 5 p.m. Saturday afternoon.
> By 5 p.m. Sunday I have to be at home.

MorphAdorner correctly splits the following text into two sentences following *p.m.*:

> It was due Friday at 5 p.m. Saturday afternoon would be too late.

The text

> She has an appointment at 5 p.m. Saturday afternoon to get her car fixed.

should be left as a single sentence, but MorphAdorner splits it into two sentences with the split occurring after *p.m.* While both *get* and *fixed* can be verbs, neither appears in context as the the right kind of verb form to allow the text following *p.m.* to be considered a sentence.

MorphAdorner does not recognize abbreviations containing blanks, such as "U. S." for United States. However, "U.S." without the blank is recognized.

Characters not allowed to start a sentence

MorphAdorner does not allow a sentence to start with a comma, a period, or a percent sign. These characters will be attached to the previous token and/or sentence, if any. Dashes and hyphens are joined preferentially to the end of a sentence rather than the start of a sentence.

Interjections

MorphAdorner maintains a list of common interjections, These are words typically used for emphasis, and generally followed by an exclamation mark or question mark. MorphAdorner does not split the sentence following the interjection, and it leaves the question mark or exclamation point attached to the interjection word. The situation can become ambiguous when quote marks are involved.

MorphAdorner treats the following lines as single sentences.

> What! That's bad!
>
> "What! That's bad!"

On the other hand, the following line is treated as two sentences.

> "What!" "That's bad!"

"What!" is the first sentence and *"That's bad!"* is the second sentence.

Numbers

A period following a number may act as both a decimal point and the end of a sentence (in English). In general, MorphAdorner ends a sentence following a number ending in a period when the next word begins with a capital letter. The following text is considered one sentence by MorphAdorner.

> There are 12. of them.

MorphAdorner splits each of the following two lines into two sentences following *12*.

> There are 12. More would be unnecessary.
>
> There are 12. "More would be unnecessary."

Spelling Standardization

English texts of the past exhibit far greater spelling variance than contemporary texts. Texts from the seventeenth century and earlier times use conventions that differ from contemporary standards in the use of "u" and "v" and "y" and capitalization, among others. Often the same words is spelled differently even within the same work. By the eighteenth-century texts employ much more modern orthographic standards, except for capitalization.

MorphAdorner uses rules, word lists, and extended search techniques such as spelling correction methods and other heuristics to map variant spellings to their standard (usually modern) form. For obsolete words no longer in use, a representative standard form is chosen which is usually the Oxford English Dictionary headword form. Presently MorphAdorner knows a couple of hundred thousand variant spellings. Using this list, MorphAdorner can automatically determine the correct standard form for previously unseen spellings in many cases.

Sometimes a new spelling is just too different from any of the ones MorphAdorner already knows. Using the extended search facilities on such a spelling may result in a "standard spelling" which veers far from the correct form. As time goes one we hope to reduce the occurrence of such errors.

Orthographic standardization improves the quality of part of speech tagging, name recognition, and text searching. However, standardization by itself isn't sufficient to fix some other problems. These include the lack of the apostrophe to mark the possessive case and the inconsistent practices of capitalization as markers of proper nouns.

In English before 1700 the apostrophe never indicates the genitive, and "her mother's daughter" is written "her mothers daughter". An even more problematic example is "her majesty's daughter" which appears in early texts as "her majesties daughter." The use of the apostrophe as a genetive marker gained ground during the eighteenth century, and has been used as it is today since the early nineteenth century.

In the eighteenth century, the apostrophe is sometimes used as a plural marker in certain character combinations. Thus "canoe's" is much more likely to be a plural than a possessive form.

The modern practice of restricting capitalization to names, namelike entities, and certain emphatic uses is about two centuries old. In earlier English nouns are freely capitalized, and capitalization is not a reliable way of picking out proper nouns. However, proper nouns have usually been capitalized in all forms of written English since about 1550. Before that names can appear in lower case.

In poetry the first word of each line is often capitalized even when that word does not start a sentence. For purposes of part-of-speech tagging, a simple workaround is to use the lower case form of a word that does not start a sentence, except if the word appears in a list of known proper names.

You can read a more detailed description of the spelling standardization process below.

You can try MorphAdorner's spelling standardizer online.

Standardization Process

This section describes the process by which MorphAdorner maps a variant spelling to a standard (usually modern) form.

Spelling Map File Formats

Spelling maps are the key to MorphAdorner's methodology for standardizing or modernizing spelling. A spelling map is a utf-8 text file contain two fields separated by a tab character. The first field is a variant spelling. The second field is the standardized spelling for the variant.

Currently MorphAdorner uses two maps. The first is culled primarily from nineteenth century fiction texts and currently contains about 5,000 entries. The second is culled from Early Modern English texts and contains over 350,000 thousand known variants. There is also a short list of about 400 variants which are known to vary by word class.

Here are some entries from the Early Modern English spelling map showing standard spellings for forms of "advance." The first column is the variant, the second column is the standard spelling.

aduauce	advance
aduauced	advanced
aduauceing	advancing
aduaucement	advancement
aduauceth	advanceth
aduaucing	advancing
aduaucyng	advancing
aduaucynge	advancing
aduaunc'd	advanced

The file of spellings by word class is similar except that it contains multiple sections. Each is headed by a word class name by a colon. This is followed by the list of variant to standard spellings for that word class. For example, the adjectives section starts:

adjective:	
agean	again
bad	bad
blew	blue
browne	brown
chaste	chaste
christen	christian
clere	clear
cliver	clever
cold	cold
cross	cross
cumfbler	cumfortabler

while the verb section starts:

verb:	
d'	do

'm am
'old hold
's is
aint aren't
ain't aren't
allays allays
an't aren't
ar are
ar' are
arena aren't
bad bade

Some spellings map to themselves when they have different standard spellings for different word classes. The spelling "bad" is an example.

Standardization Steps

MorphAdorner attempts to standardize a spelling as follows.

1. Load the list of known standard spellings. This is a combination of entries from the 1911 Webster's Dictionary and entries verified against the Oxford English Dictionary from ongoing work with the Monk project texts.
2. Load maps of known variant spellings to modern spellings as described above.
3. Create a ternary trie of all the standard and variant spellings. A ternary trie allows very efficient extraction of strings within a specified edit distance of a given string. In other words, it allows efficient extraction of list of words whose spellings are near to any given word's spelling.
4. Load a list of modernization rules. Currently MorphAdorner defines about 70 such rules which can transform many variant spellings to their modern spellings, or come very close. The rules also provide for correcting defective spellings that contain "gap" markers reflecting illegible letters in the original text. Some sample rules include:
    - Transform the ending "me~" to "men"
    - Transform the ending "ynge" to "ing"
    - Transform "uu" to "w"
    - Transform "v" followed by a non-vowel to "u"

Now for each old spelling, perform the following steps.

1. Apply all the applicable transformation rules which results in an improved spelling. If this spelling appears in the standard spellings list, we're done. For example, applying the rules to *strykynge* directly produces the modern standard spelling *striking*.
2. See if the transformed spelling appears in the variant spellings map. If so, assign the mapped spelling value as the standard spelling. We're done. For example, applying the rules to *vniuersitie* produces *universitie* . This is not the modern spelling, but it is close. The mapped spelling list for Early Modern English provides an entry for *universitie*, giving the modern spelling as *university.*

3. Compile a list of words whose spellings are "close to" the transformed spelling by using the ternary trie to search quickly for all words within a specified edit distance of the transformed word.
4. Compute a measure of *string similarity* between each found spelling and the transformed spelling. String similarity measures how similar two strings of characters are. A similarity of 0.0 indicates two strings are completely different, while a similarity of 1.0 indicates two strings are identical. MorphAdorner uses a weighted similarity score based upon letter pair similarity, phonetic distance, and edit distance.
5. Choose the found spelling with the highest similarity as the most probable correct/standard spelling. If this spelling appears in the standard spellings list, we're done. If not, see if it appears in the mapped spellings list. if so, take the mapped spelling value as the standard spelling, and we're done. Otherwise, accept the transformed spelling as the standard spelling, with the proviso that it may not be a proper standard spelling, and requires further review.

Interactions with Part Of Speech

The standard spelling for some words cannot be determined until the part of speech for the word is known. Examples of such words include doe, bee, poor, marie, and wast. Thus "doe" is most likely "doe" a female deer when it appears as a noun, while "doe" is most likely "do" when it appears as a verb. When "marie" appears as an adjective it is probably "merry", but most likely "marry" when used as a verb.

MorphAdorner keeps a short list of variant spellings by general word class. The final standardized spelling is not assigned until a part of speech has been assigned, so these special cases can usually be disambiguated properly.

Standardizing Proper Names

Proper names can appear with a bewildering variety of spellings even within a single work. Some variants can be transformed to their modern standard forms by using the general standardization rules presented above. For example, the spellings *Syracvse* and *Vlysses*, which are the commonest variants of those proper name spellings in the TCP/EEBO version of *Plutarch's Lives*, both transform by rule to their modern spellings *Syracuse* and *Ulysses*.

Other variants are not so easily rectified. The place name *Cappadocia* appears in *Plutarch's Lives* as

CPADOCIA	1
Cappadocia	21
OHPPADOCIA	1
Coppadocia	1
CAPRADOCIA	1

where the frequency of occurrence follows each variant.

MorphAdorner currently uses the following algorithm to look for standard spelling candidates for proper names. This is a variant of the extended search algorithm for standard spellings described above. Because we know we are looking for proper names, we can do a better job by limiting the search space

to known proper names.

Proper name search algorithm

1. Collect the list of known spellings of proper names (tagged with NUPOS parts of speech np1 and np2) in the early modern English lexicon. Currently there are around 66,000 such spellings.
2. Construct a "name" ternary trie of the lowercase versions of all these names. A ternary trie allows very efficient extraction of strings within a specified edit distance of a given string.
3. Construct a "consonant" ternary trie of the lowercase versions of the names with all vowels removed. For each unique combination of consonants (in order), store the list of spellings which reduce to that consonant string.

For each unknown name, perform the following steps.

1. Find all strings in the "name" trie within a specified edit distance of the unknown name. An edit distance of 2 seems to be a good choice.
2. If any names were found in step 1, compute a measure of string similarity between each found name and the unknown name. Choose the found name with the highest similarity as the most probable correct/standard spelling. Letter-pair similarity seems to work well as a measure of string similarity, but there are many other possible choices.
3. If no names were found in step 1, find all strings in the "consonant" trie within a specified edit distance of the unknown name with vowels removed. An edit distance of 3seems to be a good choice.
4. If any consonant strings were found in step 3, perform the following steps for each consonant string.
    1. Pick up all the names which reduce to this consonant string.
    2. For each of those names, compute a measure of string similarity between the name and the unknown name (that is, between the full spellings).
    3. Keep a list of those found names with a similarity score above a reasonable threshhold. 0.75 seems to be a good choice.
    4. Choose the found name with the highest similarity as the most probable correct/standard spelling.

If no names were found by either lookup procedure, leave the unknown name alone.

Here is an example of the algorithm applied to the list of names above. In each case, only one candidate spelling (the correct one, it turns out) was found.

```
Names near CPADOCIA

cappadocia (0.75)

Names near Cappadocia

cappadocia (1.0)
```

```
Names near OHPPADOCIA

cappadocia (0.7777777777777778)

Names near Coppadocia

cappadocia (0.7777777777777778)

Names near CAPRADOCIA

cappadocia (0.7777777777777778)
```

Text Segmenter

Text Segmentation methods try to break up a text into thematically meaningful segments. MorphAdorner implements two linear segmentation methods which use measures of lexical cohesion to produce segments: Marti Hearst's TextTiler and Freddy Choi's C99. Both of these try to find those portions of a text in which the vocabulary changes from one subtopic to another. These change points mark the boundaries of the text segments.

Segmentation methods have been traditionally been applied to non-fiction discursive texts. We are interested in investigating whether segmentation methods illuminate the thematic structure of a wider span of genres in both fiction and non-fiction.

You can try MorphAdorner's linear text segmenters online.

Verb Conjugator

Conjugation is the process of *inflecting* a verb by adding affixes or changing certain letters in the base verb form to give the verb a different syntactic function. The base verb or lemma form of a verb is called the *infinitive*.

Most verbs in English can be conjugated as follows. Use the infinitive for most forms, except add "ed" for the past and past participle, add "ing" for the present participle, and add "s" for the third person singular. A few simple modifications are needed for the following cases.

1. When the infinitive ends in "e", add "d" for the past and past participle, and replace the final "e" with "ing" for the present participle.
2. When the infinitive ends in ch, s, sh, x, or z , add "es" to create the third person present.
3. When the infinitive ends in a consonant preceded by a short vowel (e.g., "chop"), double the final consonant and add "ed" to the infinitive form to create the past and past participle, double the final consonant and add "ing" to create the present participle, and add "s" to create the third person present.
4. When the infinitive ends in "y", replace the final "y" with "ied" to create the past and past participle, add "ing" to create the present participle, and replace the final "y" with "ies" to create the third person present.

There are American/British differences as regards consonant doubling. MorphAdorner maintains a list of verbs whose final consonant is typically doubled in British English, and always doubles the consonant for verbs on that list. Optionally, MorphAdorner knows how to generate the American spelling without the doubled terminal consonant for many common forms.

Verbs whose conjugations follow the rules above are called *regular verbs*. Verbs which do not follow these rules are called *irregular verbs*. English has several hundred irregular verbs, which include some of the most commonly used. MorphAdorner checks a list of irregular verb forms before applying the regular conjugation rules above. There are a few ambiguities since some common verbs take different forms depending upon their meaning. Examples include:

infinitive	**past**	**past participle**
hang (put to death)	hanged	hanged
hang (a photo)	hung	hung
lie (recline)	lay	lain
lie (tell a falsehood)	lied	lied

Some verbs can take either regular or irregular past or past participle forms (examples: shine, kneel, light, prove, wake). MorphAdorner usually generates the regular form unless the irregular form appears to be more commonly used.

You can try MorphAdorner's English verb conjugator online.

Word Tokenization

Extracting words and sentences from a text are fundamental operations required by other language processing functions. **Word tokenization** splits a text into words and punctuation marks. **Sentence splitting** (see page 100) assembles the tokenized text into sentences.

The first step in word tokenization is recognizing word boundaries. The tokenizer uses white space such as blanks and tabs as the primary cue for splitting the text into tokens. Punctuation marks are split from the initial tokens. This is not as easy as it sounds. For example, when should a token containing a hypen be split into two or more tokens? When does a period indicate the end of an abbreviation as opposed to a sentence or a number or a Roman numeral? Sometimes a period can act as a sentence terminator and an abbreviation terminator at the same time. When should a single quote be split from a word? Early modern English included many contractions such as **'tis** with a leading quote.

MorphAdorner's tokenizers use a number of heuristics and a list of common abbreviations to produce a sequence of punctuation and spellings that will be consistent with the subsequent operations of sentence boundary identification, part of speech tagging, and lemmatization. Different part of speech tag sets may require different tokenization. The Penn Treebank tag set assumes contractions should be split into separate tokens. Thus the token **can't** appears as two tokens, **can** and **'t**. The NUPOS tag set can work with tokens split this way, but at present we prefer to keep contracted forms as a single token.

Even when the text has been more-or-less correctly tokenized the individual tokens may still be erroneous. The digital text of many Early Modern English works was created using scanners and optical character recognition (OCR) software. Such digitized text frequently contains all manner of orthographic errors. Example include substitution of "~" for the letters "m" or "n" and mapping of the archaic long "s" as the letter "f". Some of these errors can be corrected automatically using heuristics and a spelling standardizer.

In the print world, a punctuation mark does not count as a word. Instead punctuation separates groups of words. In computer terms, punctuation is a kind of "meta-data", not so qualitatively different from SGML or XML markup. MorphAdorner's word tokenizers treat punctuation marks as words. This procedure is justified because the punctuation "meta-data" added by authors (or editors) lives at the same level of data as the words and allows a consistent treatment of token transition probabilities for adornment processes such as part of speech tagging.

You may be interested in reading below about some tokenization problems we encountered while processing literary texts.

You can try MorphAdorner's default word tokenizer online. The example only works with plain unmarked text.

Word Tokenization Problems

The following presents some of the problems and solutions encountered while developing the word tokenizers for MorphAdorner. One important general principle is that MorphAdorner's word tokenizer and sentence splitter iterate back and forth as needed to achieve the best possible sentence splitting and tokenization.

Commas in numbers

MorphAdorner treats a comma as a separator in all cases except when a comma appears in the middle of a number. For example, the string **1,250** represents a number (one thousand two hundred fifty). MorphAdorner leaves such number strings intact so that the part of speech taggers can treat it as a number.

Missing whitespace after a period

Many sentences in literary text transcriptions run together without a space after the period. Example:

```
systematic."How
```

Here the sentence should be split after the period and before the double quote.

For

```
systematic.'How
```

the sentence split should occur on the single quote because contractions should rarely, if ever, have a "." followed by a single quote.

Some commonly merged forms should always be split:

```
Mr.Capitalname -> Mr. Capitalname
&c.crap -> &c. crap
Mrs.Howell -> Mrs. Howell
St.Miriam
Mr.Doyce!
Dr.Mull
Mr.R.'s -> Mr. R.'s
```

Examples of other merged strings which should be split include

```
stairs.The
pleasing.How
emotions.What!
bloodthirstiness.The
on.Think
Tom.You
spring.The
it.Or
houses.But,
door.The
in.He
stairs.The
right.The
so.But
sufferable.The
dishonour.But
emotion.She
Esq.Advocate
```

Here the decision to split comes from the nature of the tokens on the left and right hand sides of the period. In each case, the token is a known word or abbreviation in its own right.

On the other hand, common abbreviations should not be split. MorphAdorner keeps a list of these. Examples:

```
i.e.
p.m.
```

It can be difficult to decide in some cases when a string is a legitimate abbreviation. For example, **e.g_** is presumably a variant of **e.g.**, but what about **etc.s**? When in doubt, MorphAdorner leaves a potential abbreviation unsplit.

Roman numerals

Roman numerals in older texts exhibit considerably more orthographic variation than contemporary usage allows. For example, the letter "j" is often used as a substitute for the letter "i" and "u" for "v". Runs of letters may exceed the nominal length, e.g., "iiiii" may be used where "v" would normally appear in current usage. Particularly in early modern texts, numerals may be preceded and/or followed by a period. Examples:

```
xviiii 19
xxc 80
.XVI. 16
```

Some Roman numerals are followed by the letter "o" or "m" in a <sup> tag, e.g., DCCXXV^o. These are Latin or quasi-Latin inflection markers for a dative or accusative form. These should be treated as a form of the word without the trailing marker characters, e.g., DCCXXV^o should be treated as DCCXXV.

MorphAdorner attempts to recognize many of these variants so that they can be assigned one of the number part of speech tags.

Part Six: Programming Examples

Example One: Adorning a string With Parts Of Speech

Adorning a string With Parts Of Speech

Suppose you have a string of text containing one or more sentences. How do you use MorphAdorner to assign part of speech tags to each word in the text?

Creating a default tokenizer and sentence splitter

First you need to break up the text into sentences and words. In MorphAdorner you use a sentence splitter and a word tokenizer to perform these tasks. You can use MorphAdorner's default sentence splitter and default word tokenizer by creating an instance of each as follows.

```
WordTokenizer wordTokenizer = new DefaultWordTokenizer();

SentenceSplitter sentenceSplitter   =
    new DefaultSentenceSplitter();
```

Use the sentence splitter and word tokenizer to split the text into a java.util.List of sentences, each of which is in turn a java.util.List of word and punctuation tokens. (Whitespace is not captured as part of the token list.) The text to split is stored in `textToAdorn`.

```
List<List<String>> sentences    =
    sentenceSplitter.extractSentences(
        textToAdorn , wordTokenizer );
```

Note that the sentence splitter requires the word tokenizer as a parameter.

Getting the parts of speech

Next, create an instance of MorphAdorner's default part of speech tagger. The default tagger is a trigram tagger using a hidden Markov model and a beam search variant of the Viterbi algorithm. The default lexicon is a combination of an extensive English name list and words found in 19th century British fiction. The default part of speech tag set is the NUPOS tag set.

```
PartOfSpeechTagger partOfSpeechTagger   =
    new DefaultPartOfSpeechTagger();
```

Now invoke the part of speech tagger to assign parts of speech to each word in the extracted sentences.

```
List<List<AdornedWord>> taggedSentences =
    partOfSpeechTagger.tagSentences( sentences );
```

Displaying the results

The part of speech tagger returns a java.util.List of java.util.list entries. Each secondary java.util.List is

a list of AdornedWord entries. Only the spelling and part of speech fields in each AdornedWord entry are guaranteed to be defined upon return from the part of speech tagger. You can display the results by extracting and printing the spelling and associated part of speech for each word.

```
        for ( int i = 0 ; i < sentences.size() ; i++ )
        {
                                //  Get the next adorned sentence.
                                //  This contains a list of adorned
                                //  words.  Only the spellings
                                //  and part of speech tags are
                                //  guaranteed to be defined.

            List<AdornedWord> sentence  = taggedSentences.get( i );

            System.out.println
            (
                "---------- Sentence " + ( i + 1 ) + "----------"
            );

                                //  Print out the spelling and part(s)
                                //  of speech for each word in the
                                //  sentence.  Punctuation is treated
                                //  as a word too.

            for ( int j = 0 ; j < sentence.size() ; j++ )
            {
                AdornedWord adornedWord = sentence.get( j );

                System.out.println
                (
                    StringUtils.rpad( ( j + 1 ) + "" , 3  ) + ": " +
                    StringUtils.rpad( adornedWord.getSpelling() , 20 ) +
                    adornedWord.getPartsOfSpeech()
                );
            }
        }
    }
```

Putting it altogether

You can peruse the Java source code for PosTagString below which puts all the above code together in a runnable sample program. You will also find the source code in the `src/edu/northwestern/at/morphadorner/examples/` directory in the MorphAdorner release.

```
package edu.northwestern.at.morphadorner.examples;

/*  Please see the license information at the end of this file. */

import java.util.*;

import edu.northwestern.at.utils.*;
import edu.northwestern.at.utils.corpuslinguistics.adornedword.*;
```

```
import edu.northwestern.at.utils.corpuslinguistics.postagger.*;
import edu.northwestern.at.utils.corpuslinguistics.sentencesplitter.*;
import edu.northwestern.at.utils.corpuslinguistics.tokenizer.*;

/** PosTagString: Adorn a string with parts of speech.
 *
 *  <p>
 *  Usage:
 *  </p>
 *
 *  <p>
 *  <code>
 *  java -Xmx256m edu.northwestern.at.morphadorner.example.PosTagString "Text to
adorn."
 *  </code>
 *  </p>
 *
 *  <p>
 *  where "Text to adorn." specifies one or more sentences of text to
 *  adorn with part of speech tags.  The default tokenizer,
 *  sentence splitter, lexicons, and part of speech tagger are used.
 *  </p>
 *
 *  <p>
 *  Example:
 *  </p>
 *
 *  <p>
 *  <code>
 *  java -Xmx256m edu.northwestern.at.morphadorner.example.PosTagString "Mary had
a little lamb.  Its fleece was white as snow."
 *  </code>
 *  </p>
 */

public class PosTagString
{
    /** Main program.
     *
     *  @param  args    Program parameters.
     */

    public static void main( String[] args )
    {
        try
        {
            adornText( args );
        }
        catch( Exception e )
        {
            e.printStackTrace();
        }
    }

    /** Adorn text specified as a program parameter.
     *
     *  @param  args    The program parameters.
```

```
     *
     *  <p>
     *  args[ 0 ] contains the text to adorn.  The text may contain
     *  one or more sentences with punctuation.
     *  </p>
     */

    public static void adornText( String[] args )
        throws Exception
    {
                                //  Get text to adorn.  Report error
                                //  and quit if none.

        if ( args.length < 1 )
        {
            System.out.println( "No text to adorn." );
            System.exit( 1 );
        }

        String textToAdorn  = args[ 0 ];

                                //  Get default part of speech tagger.

        PartOfSpeechTagger partOfSpeechTagger   =
            new DefaultPartOfSpeechTagger();

                                //  Get default word tokenizer.

        WordTokenizer wordTokenizer = new DefaultWordTokenizer();

                                //  Get default sentence splitter.

        SentenceSplitter sentenceSplitter   =
            new DefaultSentenceSplitter();

                                //  Get the part of speech
                                //  guesser from the part of
                                //  speech tagger.  Set this into
                                //  sentence splitter to improve
                                //  sentence boundary recognition.

        sentenceSplitter.setPartOfSpeechGuesser
        (
            partOfSpeechTagger.getPartOfSpeechGuesser()
        );
                                //  Split text into sentences
                                //  and words.  Here "sentences"
                                //  contains a list of sentences.
                                //  Each sentence is itself a list of words.

        List<List<String>> sentences    =
            sentenceSplitter.extractSentences(
                textToAdorn , wordTokenizer );

                                //  Assign part of speech tags to
                                //  each word in each sentence.
                                //  Here "taggedSentences" contains
```

```
                                //  a list of List<AdornedWord> entries,
                                //  one for each sentence.

        List<List<AdornedWord>> taggedSentences =
            partOfSpeechTagger.tagSentences( sentences );

                                //  Display tagged words.

        for ( int i = 0 ; i < sentences.size() ; i++ )
        {
                                //  Get the next adorned sentence.
                                //  This contains a list of adorned
                                //  words.  Only the spellings
                                //  and part of speech tags are
                                //  guaranteed to be defined.

            List<AdornedWord> sentence  = taggedSentences.get( i );

            System.out.println
            (
                "---------- Sentence " + ( i + 1 ) + "----------"
            );

                                //  Print out the spelling and part(s)
                                //  of speech for each word in the
                                //  sentence.  Punctuation is treated
                                //  as a word too.

            for ( int j = 0 ; j < sentence.size() ; j++ )
            {
                AdornedWord adornedWord = sentence.get( j );

                System.out.println
                (
                    StringUtils.rpad( ( j + 1 ) + "" , 3  ) + ": " +
                    StringUtils.rpad( adornedWord.getSpelling() , 20 ) +
                    adornedWord.getPartsOfSpeech()
                );
            }
        }
    }
}

/*
Copyright (c) 2008, 2009 by Northwestern University.
All rights reserved.

Developed by:
   Academic and Research Technologies
   Northwestern University
   http://www.it.northwestern.edu/about/departments/at/

Permission is hereby granted, free of charge, to any person
obtaining a copy of this software and associated documentation
files (the "Software"), to deal with the Software without
restriction, including without limitation the rights to use,
copy, modify, merge, publish, distribute, sublicense, and/or
```

```
sell copies of the Software, and to permit persons to whom the
Software is furnished to do so, subject to the following
conditions:

    * Redistributions of source code must retain the above copyright
      notice, this list of conditions and the following disclaimers.

    * Redistributions in binary form must reproduce the above
      copyright notice, this list of conditions and the following
      disclaimers in the documentation and/or other materials provided
      with the distribution.

    * Neither the names of Academic and Research Technologies,
      Northwestern University, nor the names of its contributors may be
      used to endorse or promote products derived from this Software
      without specific prior written permission.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE CONTRIBUTORS OR
COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR
OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS WITH THE SOFTWARE.
*/
```

Example Two: Adorning a string with lemmata and standard spellings

Let's extend the example of adorning a string with parts of speech to add lemma forms and standardized spellings for each word in the string.

Creating a default lemmatizer and spelling standardizer

We will use the default English lemmatizer and the default spelling standardizer.

```
                            //  Get the default English
                            //  lemmatizer.

Lemmatizer lemmatizer       =  new DefaultLemmatizer();

                            //  Get the default spelling
                            //  standardizer.

SpellingStandardizer standardizer   =
    new DefaultSpellingStandardizer();
```

Adding lemmata and standardized spellings to the output

The process of adding parts of the speech is the same as in **PosTagString**. We call two new auxiliary methods to determine the lemmata and standard spelling for each part-of-speech tagged spelling.

```
        for ( int j = 0 ; j < sentence.size() ; j++ )
        {
            AdornedWord adornedWord = sentence.get( j );

                            //  Get the standard spelling
                            //  given the original spelling
                            //  and part of speech.

            setStandardSpelling
            (
                adornedWord ,
                standardizer ,
                partOfSpeechTags
            );
                            //  Set the lemma.

            setLemma
            (
                adornedWord ,
                wordLexicon ,
                lemmatizer ,
                partOfSpeechTags ,
                spellingTokenizer
            );

                            //  Display the adornments.
```

```
            System.out.println
            (
                StringUtils.rpad( ( j + 1 ) + "" , 3  ) + ": " +
                StringUtils.rpad( adornedWord.getSpelling() , 20 ) +
                StringUtils.rpad(
                    adornedWord.getPartsOfSpeech() , 8 ) +
                StringUtils.rpad(
                    adornedWord.getStandardSpelling() , 20 ) +
                adornedWord.getLemmata()
            );
        }
```

Getting the lemma form

We start by setting the lemma form to the spelling. If the spelling belongs to a word class which should not be further lemmatized, we do nothing further. We test for this by checking if the lemmatization class for the spelling's associated part of speech tag is "none" or if the language specific lemmatizer indicates that tag should not be lemmatized.

If the spelling should be lemmatized, we next check if there are multiple parts of speech in the spelling. If so, we try to find the lemma form for each part separately, and join them into a compound lemma, separating the individual pieces with the lemma form separator character. If the spelling has only a single part of speech, we find the lemma form that best fits the combination of spelling and part of speech.

```
    /** Get lemma for a word.
     *
     *  @param  adornedWord         The adorned word.
     *  @param  lexicon             The word lexicon.
     *  @param  lemmatizer          The lemmatizer.
     *  @param  partOfSpeechTags    The part of speech tags.
     *  @param  spellingTokenizer   Tokenizer for spelling.
     *
     *  <p>
     *  On output, sets the lemma field of the adorned word
     *  We look in the word lexicon first for the lemma.
     *  If the lexicon does not contain the lemma, we
     *  use the lemmatizer.
     *  </p>
     */

    public static void setLemma
    (
        AdornedWord adornedWord  ,
        Lexicon lexicon ,
        Lemmatizer lemmatizer ,
        PartOfSpeechTags partOfSpeechTags ,
        WordTokenizer spellingTokenizer
    )
    {
        String spelling     = adornedWord.getSpelling();
        String partOfSpeech = adornedWord.getPartsOfSpeech();
        String lemmata      = spelling;

                                //  Get lemmatization word class
```

```
                        //  for part of speech.
String lemmaClass   =
    partOfSpeechTags.getLemmaWordClass( partOfSpeech );

                        //  Do not lemmatize words which
                        //  should not be lemmatized,
                        //  including proper names.

if  (   lemmatizer.cantLemmatize( spelling ) ||
        lemmaClass.equals( "none" )
    )
{
}
else
{
                        //  Try compound word exceptions
                        //  list first.

    lemmata = lemmatizer.lemmatize( spelling , "compound" );

                        //  If lemma not found, keep trying.

    if ( lemmata.equals( spelling ) )
    {
                        //  Extract individual word parts.
                        //  May be more than one for a
                        //  contraction.

        List wordList   =
            spellingTokenizer.extractWords( spelling );

                        //  If just one word part,
                        //  get its lemma.

        if  (   !partOfSpeechTags.isCompoundTag( partOfSpeech ) ||
                ( wordList.size() == 1 )
            )
        {
            if ( lemmaClass.length() == 0 )
            {
                lemmata = lemmatizer.lemmatize( spelling );
            }
            else
            {
                lemmata =
                    lemmatizer.lemmatize( spelling , lemmaClass );
            }
        }
                        //  More than one word part.
                        //  Get lemma for each part and
                        //  concatenate them with the
                        //  lemma separator to form a
                        //  compound lemma.
        else
        {
            lemmata             = "";
            String lemmaPiece   = "";
```

```
                    String[] posTags    =
                        partOfSpeechTags.splitTag( partOfSpeech );

                    if ( posTags.length == wordList.size() )
                    {
                        for ( int i = 0 ; i < wordList.size() ; i++ )
                        {
                            String wordPiece    = (String)wordList.get( i );

                            if ( i > 0 )
                            {
                                lemmata = lemmata + lemmaSeparator;
                            }

                            lemmaClass  =
                                partOfSpeechTags.getLemmaWordClass
                                (
                                    posTags[ i ]
                                );

                            lemmaPiece  =
                                lemmatizer.lemmatize
                                (
                                    wordPiece ,
                                    lemmaClass
                                );

                            lemmata = lemmata + lemmaPiece;
                        }
                    }
                }
            }
        }

        adornedWord.setLemmata( lemmata );
    }
}
```

Getting the standardized spelling

We start by setting the standardized form to the original spelling. If the spelling belongs to a word class which should not be standardized, we do nothing further. This includes spellings that are tagged as numbers, proper nouns, and foreign words.

If the spelling can be standardized, we ask the spelling standardizer to give us the best standardized form it can. We try to match the case of the original spelling in the standardized form. Alternatively we could always set the standardized form to a lower case version, except possibly for proper nouns and adjectives, and the pronoun "I".

```
    /** Get standard spelling for a word.
     *
     *  @param  adornedWord     The adorned word.
     *  @param  standardizer        The spelling standardizer.
     *  @param  partOfSpeechTags    The part of speech tags.
     *
```

```
     *  <p>
     *  On output, sets the standard spelling field of the adorned word
     *  </p>
     */

    public static void setStandardSpelling
    (
        AdornedWord adornedWord  ,
        SpellingStandardizer standardizer ,
        PartOfSpeechTags partOfSpeechTags
    )
    {
                                //  Get the spelling.

        String spelling         = adornedWord.getSpelling();
        String standardSpelling = spelling;
        String partOfSpeech     = adornedWord.getPartsOfSpeech();

                                //  Leave proper nouns alone.

        if ( partOfSpeechTags.isProperNounTag( partOfSpeech ) )
        {
        }
                                //  Leave nouns with internal
                                //  capitals alone.

        else if (   partOfSpeechTags.isNounTag( partOfSpeech )  &&
                    CharUtils.hasInternalCaps( spelling ) )
        {
        }
                                //  Leave foreign words alone.

        else if ( partOfSpeechTags.isForeignWordTag( partOfSpeech ) )
        {
        }
                                //  Leave numbers alone.

        else if ( partOfSpeechTags.isNumberTag( partOfSpeech ) )
        {
        }
                                //  Anything else -- call the
                                //  standardizer on the spelling
                                //  to get the standard spelling.
        else
        {
            standardSpelling    =
                standardizer.standardizeSpelling
                (
                    adornedWord.getSpelling() ,
                    partOfSpeechTags.getMajorWordClass
                    (
                        adornedWord.getPartsOfSpeech()
                    )
                );

                                //  If the standard spelling
                                //  is the same as the original
```

```
                                //  spelling except for case,
                                //  use the original spelling.

            if ( standardSpelling.equalsIgnoreCase( spelling ) )
            {
                standardSpelling    = spelling;
            }
        }
                                //  Set the standard spelling.

        adornedWord.setStandardSpelling( standardSpelling );
    }
```

Putting it altogether

You can peruse the Java source code for **AdornAString** below which puts all the above code together in a runnable sample program. You will also find the source code in the `src/edu/northwestern/at/morphadorner/examples/` directory in the MorphAdorner release.

```
package edu.northwestern.at.morphadorner.examples;

/*  Please see the license information at the end of this file. */

import java.util.*;

import edu.northwestern.at.utils.*;
import edu.northwestern.at.utils.corpuslinguistics.adornedword.*;
import edu.northwestern.at.utils.corpuslinguistics.lemmatizer.*;
import edu.northwestern.at.utils.corpuslinguistics.lexicon.*;
import edu.northwestern.at.utils.corpuslinguistics.partsofspeech.*;
import edu.northwestern.at.utils.corpuslinguistics.postagger.*;
import edu.northwestern.at.utils.corpuslinguistics.sentencesplitter.*;
import edu.northwestern.at.utils.corpuslinguistics.spellingstandardizer.*;
import edu.northwestern.at.utils.corpuslinguistics.tokenizer.*;

/** AdornAString: Adorn a string with parts of speech, lemmata, and
 *  standard spellings.
 *
 *  <p>
 *  Usage:
 *  </p>
 *
 *  <p>
 *  <code>
 *  java -Xmx256m edu.northwestern.at.morphadorner.example.AdornAString "Text to
adorn."
 *  </code>
 *  </p>
 *
 *  <p>
 *  where "Text to adorn." specifies one or more sentences of text to
 *  adorn with part of speech tags, lemmata, and standard spellings.
 *  The default tokenizer, sentence splitter, lexicons, part of speech tagger,
```

```
 *  lemmatizer, and spelling standardizer  are used.
 *  </p>
 *
 *  <p>
 *  Example:
 *  </p>
 *
 *  <p>
 *  <code>
 *  java -Xmx256m edu.northwestern.at.morphadorner.example.AdornAString "Mary had
a little lamb.  Its fleece was white as snow."
 *  </code>
 *  </p>
 */

public class AdornAString
{
    /** Lemma separator character, */

    public static String lemmaSeparator = "|";

    /** Main program.
     *
     *  @param  args    Program parameters.
     */

    public static void main( String[] args )
    {
        try
        {
            adornText( args );
        }
        catch( Exception e )
        {
            e.printStackTrace();
        }
    }

    /** Adorn text specified as a program parameter.
     *
     *  @param  args    The program parameters.
     *
     *  <p>
     *  args[ 0 ] contains the text to adorn.  The text may contain
     *  one or more sentences with punctuation.
     *  </p>
     */

    public static void adornText( String[] args )
        throws Exception
    {
                                //  Get text to adorn.  Report error
                                //  and quit if none.

        if ( args.length < 1 )
        {
            System.out.println( "No text to adorn." );
```

```
        System.exit( 1 );
    }

    String textToAdorn  = args[ 0 ];

                            //  Get default part of speech tagger.

    PartOfSpeechTagger partOfSpeechTagger   =
        new DefaultPartOfSpeechTagger();

                            //  Get default word lexicon from
                            //  part of speech tagger.

    Lexicon wordLexicon = partOfSpeechTagger.getLexicon();

                            //  Get the part of speech tags from
                            //  the word lexicon.

    PartOfSpeechTags partOfSpeechTags   =
        wordLexicon.getPartOfSpeechTags();

                            //  Get default word tokenizer.

    WordTokenizer wordTokenizer = new DefaultWordTokenizer();

                            //  Get spelling tokenizer.

    WordTokenizer spellingTokenizer =
        new PennTreebankTokenizer();

                            //  Get default sentence splitter.

    SentenceSplitter sentenceSplitter   =
        new DefaultSentenceSplitter();

                            //  Get the part of speech
                            //  guesser from the part of
                            //  speech tagger.  Set this into
                            //  sentence splitter to improve
                            //  sentence boundary recognition.

    sentenceSplitter.setPartOfSpeechGuesser
    (
        partOfSpeechTagger.getPartOfSpeechGuesser()
    );
                            //  Get the default English
                            //  lemmatizer.

    Lemmatizer lemmatizer       =  new DefaultLemmatizer();

                            //  Get the default spelling
                            //  standardizer.

    SpellingStandardizer standardizer   =
        new DefaultSpellingStandardizer();

                            //  Split text into sentences
```

```
                        //  and words.  Here "sentences"
                        //  contains a list of sentences.
                        //  Each sentence is itself a list of words.

List<List<String>> sentences    =
    sentenceSplitter.extractSentences(
        textToAdorn , wordTokenizer );

                        //  Assign part of speech tags to
                        //  each word in each sentence.
                        //  Here "taggedSentences" contains
                        //  a list of List<AdornedWord> entries,
                        //  one for each sentence.

List<List<AdornedWord>> taggedSentences =
    partOfSpeechTagger.tagSentences( sentences );

                        //  Loop over sentences and
                        //  display adornments.

for ( int i = 0 ; i < sentences.size() ; i++ )
{
                        //  Get the next adorned sentence.
                        //  This contains a list of adorned
                        //  words.  Only the spellings
                        //  and part of speech tags are
                        //  guaranteed to be defined at
                        //  this point.

    List<AdornedWord> sentence  = taggedSentences.get( i );

    System.out.println
    (
        StringUtils.dupl( "-" , 30 ) +
        " " + ( i + 1 ) + " " +
        StringUtils.dupl( "-" , 30 )
    );
                        //  Print out the spelling and part(s)
                        //  of speech for each word in the
                        //  sentence.  Punctuation is treated
                        //  as a word too.

    for ( int j = 0 ; j < sentence.size() ; j++ )
    {
        AdornedWord adornedWord = sentence.get( j );

                        //  Get the standard spelling
                        //  given the original spelling
                        //  and part of speech.

        setStandardSpelling
        (
            adornedWord ,
            standardizer ,
            partOfSpeechTags
        );
                        //  Set the lemma.
```

```
                setLemma
                (
                    adornedWord ,
                    wordLexicon ,
                    lemmatizer ,
                    partOfSpeechTags ,
                    spellingTokenizer
                );

                                //  Display the adornments.

                System.out.println
                (
                    StringUtils.rpad( ( j + 1 ) + "" , 3  ) + ": " +
                    StringUtils.rpad( adornedWord.getSpelling() , 20 ) +
                    StringUtils.rpad(
                        adornedWord.getPartsOfSpeech() , 8 ) +
                    StringUtils.rpad(
                        adornedWord.getStandardSpelling() , 20 ) +
                    adornedWord.getLemmata()
                );
            }
        }
    }

    /** Get standard spelling for a word.
     *
     *  @param  adornedWord     The adorned word.
     *  @param  standardizer        The spelling standardizer.
     *  @param  partOfSpeechTags    The part of speech tags.
     *
     *  <p>
     *  On output, sets the standard spelling field of the adorned word
     *  </p>
     */

    public static void setStandardSpelling
    (
        AdornedWord adornedWord  ,
        SpellingStandardizer standardizer ,
        PartOfSpeechTags partOfSpeechTags
    )
    {
                                //  Get the spelling.

        String spelling         = adornedWord.getSpelling();
        String standardSpelling = spelling;
        String partOfSpeech     = adornedWord.getPartsOfSpeech();

                                //  Leave proper nouns alone.

        if ( partOfSpeechTags.isProperNounTag( partOfSpeech ) )
        {
        }
                                //  Leave nouns with internal
                                //  capitals alone.
```

```
        else if (   partOfSpeechTags.isNounTag( partOfSpeech )  &&
                    CharUtils.hasInternalCaps( spelling ) )
        {
        }
                                //  Leave foreign words alone.

        else if ( partOfSpeechTags.isForeignWordTag( partOfSpeech ) )
        {
        }
                                //  Leave numbers alone.

        else if ( partOfSpeechTags.isNumberTag( partOfSpeech ) )
        {
        }
                                //  Anything else -- call the
                                //  standardizer on the spelling
                                //  to get the standard spelling.
        else
        {
            standardSpelling    =
                standardizer.standardizeSpelling
                (
                    adornedWord.getSpelling() ,
                    partOfSpeechTags.getMajorWordClass
                    (
                        adornedWord.getPartsOfSpeech()
                    )
                );

                                //  If the standard spelling
                                //  is the same as the original
                                //  spelling except for case,
                                //  use the original spelling.

            if ( standardSpelling.equalsIgnoreCase( spelling ) )
            {
                standardSpelling    = spelling;
            }
        }
                                //  Set the standard spelling.

        adornedWord.setStandardSpelling( standardSpelling );
    }

    /** Get lemma for a word.
     *
     *  @param  adornedWord         The adorned word.
     *  @param  lexicon             The word lexicon.
     *  @param  lemmatizer          The lemmatizer.
     *  @param  partOfSpeechTags    The part of speech tags.
     *  @param  spellingTokenizer   Tokenizer for spelling.
     *
     *  <p>
     *  On output, sets the lemma field of the adorned word
     *  We look in the word lexicon first for the lemma.
     *  If the lexicon does not contain the lemma, we
```

```
     *  use the lemmatizer.
     *  </p>
     */

    public static void setLemma
    (
        AdornedWord adornedWord  ,
        Lexicon lexicon ,
        Lemmatizer lemmatizer ,
        PartOfSpeechTags partOfSpeechTags ,
        WordTokenizer spellingTokenizer
    )
    {
        String spelling     = adornedWord.getSpelling();
        String partOfSpeech = adornedWord.getPartsOfSpeech();
        String lemmata      = spelling;

                                //  Get lemmatization word class
                                //  for part of speech.
        String lemmaClass   =
            partOfSpeechTags.getLemmaWordClass( partOfSpeech );

                                //  Do not lemmatize words which
                                //  should not be lemmatized,
                                //  including proper names.

        if  (   lemmatizer.cantLemmatize( spelling ) ||
                lemmaClass.equals( "none" )
            )
        {
        }
        else
        {
                                //  Try compound word exceptions
                                //  list first.

            lemmata = lemmatizer.lemmatize( spelling , "compound" );

                                //  If lemma not found, keep trying.

            if ( lemmata.equals( spelling ) )
            {
                                //  Extract individual word parts.
                                //  May be more than one for a
                                //  contraction.

                List wordList   =
                    spellingTokenizer.extractWords( spelling );

                                //  If just one word part,
                                //  get its lemma.

                if  (   !partOfSpeechTags.isCompoundTag( partOfSpeech ) ||
                        ( wordList.size() == 1 )
                    )
                {
                    if ( lemmaClass.length() == 0 )
```

```
                    {
                        lemmata = lemmatizer.lemmatize( spelling );
                    }
                    else
                    {
                        lemmata =
                            lemmatizer.lemmatize( spelling , lemmaClass );
                    }
                }
                            //  More than one word part.
                            //  Get lemma for each part and
                            //  concatenate them with the
                            //  lemma separator to form a
                            //  compound lemma.
                else
                {
                    lemmata             = "";
                    String lemmaPiece   = "";
                    String[] posTags    =
                        partOfSpeechTags.splitTag( partOfSpeech );

                    if ( posTags.length == wordList.size() )
                    {
                        for ( int i = 0 ; i < wordList.size() ; i++ )
                        {
                            String wordPiece    = (String)wordList.get( i );

                            if ( i > 0 )
                            {
                                lemmata = lemmata + lemmaSeparator;
                            }

                            lemmaClass  =
                                partOfSpeechTags.getLemmaWordClass
                                (
                                    posTags[ i ]
                                );

                            lemmaPiece  =
                                lemmatizer.lemmatize
                                (
                                    wordPiece ,
                                    lemmaClass
                                );

                            lemmata = lemmata + lemmaPiece;
                        }
                    }
                }
            }
        }

        adornedWord.setLemmata( lemmata );
    }
}

/*
```

Example Three: Finding sentence and token offsets

You may want to locate word and sentence boundaries as a first step in text processing. Here we produce a program called **SentenceAndTokenOffsets** to find such boundaries and locate the character offsets of each sentence and word as well.

First you need to break up the text into sentences and words. In MorphAdorner you use a sentence splitter and a word tokenizer to perform these tasks. You can use MorphAdorner's default sentence splitter and default word tokenizer by creating an instance of each as follows.

```
WordTokenizer wordTokenizer = new DefaultWordTokenizer();

SentenceSplitter sentenceSplitter   =
    new DefaultSentenceSplitter();
```

Note that the sentence splitter requires the word tokenizer as a parameter.

To improve the accuracy of the sentence splitter you can create a part of speech guesser using the default word lexicon and default suffix lexicon.

```
                        //  Create part of speech guesser
                        //  for use by splitter.

PartOfSpeechGuesser partOfSpeechGuesser =
    new DefaultPartOfSpeechGuesser();

                        //  Get default word lexicon for
                        //  use by part of speech guesser.

Lexicon lexicon = new DefaultWordLexicon();

                        //  Set lexicon into guesser.

partOfSpeechGuesser.setWordLexicon( lexicon );

                        //  Get default suffix lexicon for
                        //  use by part of speech guesser.

Lexicon suffixLexicon       = new DefaultSuffixLexicon();

                        //  Set suffix lexicon into guesser.

partOfSpeechGuesser.setSuffixLexicon( suffixLexicon );

                        //  Set guesser into sentence splitter.

splitter.setPartOfSpeechGuesser( partOfSpeechGuesser );
```

Sample text: Lincoln's Gettysburg Address

Let's use Abraham Lincoln's "Gettysburg Address" as a sample text.

> Four score and seven years ago our fathers brought forth on this continent a new nation, conceived in Liberty, and dedicated to the proposition that all men are created equal.

> Now we are engaged in a great civil war, testing whether that nation, or any nation, so conceived and so dedicated, can long endure. We are met on a great battle-field of that war. We have come to dedicate a portion of that field, as a final resting place for those who here gave their lives that that nation might live. It is altogether fitting and proper that we should do this.
>
> But, in a larger sense, we can not dedicate—we can not consecrate—we can not hallow—this ground. The brave men, living and dead, who struggled here, have consecrated it, far above our poor power to add or detract. The world will little note, nor long remember what we say here, but it can never forget what they did here. It is for us the living, rather, to be dedicated here to the unfinished work which they who fought here have thus far so nobly advanced. It is rather for us to be here dedicated to the great task remaining before us—that from these honored dead we take increased devotion to that cause for which they gave the last full measure of devotion—that we here highly resolve that these dead shall not have died in vain—that this nation, under God, shall have a new birth of freedom—and that government : of the people, by the people, for the people, shall not perish from the earth.

Place that text into a utf-8 text file called gettysburg.txt . You can use a MorphAdorner utility method to read the text. You may want to convert all the whitespace characters into blanks for legibility and to avoid problems with platform specific end of line characters.

```
                            //  Load text to split into
                            //  sentences and tokens.

String sampleText   =
    FileUtils.readTextFile( inputFileName , "utf-8" );

                            //  Convert all whitespace characters
                            //  into blanks.  (Not necessary,
                            //  but makes the display cleaner below.)

sampleText  = sampleText.replaceAll( "\\s" , " " );
```

Use the sentence splitter and word tokenizer to split the text into a java.util.List of sentences, each of which is in turn a java.util.List of word and punctuation tokens.

```
List<List<String>> sentences    =
    sentenceSplitter.extractSentences(
        textToAdorn , wordTokenizer );
```

Next use the *findSentenceOffsets* method provided by the sentence splitter to get the list of sentence offsets. You can use these to find the end of each sentence as well.

```
                            //  Get sentence start and end
                            //  offsets in input text.

int[] sentenceOffsets   =
    splitter.findSentenceOffsets( sampleText , sentences );
```

Within each sentence you can use the tokenizer method *findWordOffsets* to locate the start of each token in a sentence relative to the start of the sentence.

```
                            //  Get offsets for each word token
                            //  relative to this sentence.
```

```
int[] wordOffsets   =
    tokenizer.findWordOffsets( sentence , words  );
```

Putting it altogether

You can peruse the Java source code for **SentenceAndTokenOffsets** below which puts all the above code together in a runnable sample program. You will also find the source code in the `src/edu/northwestern/at/morphadorner/examples/` directory in the MorphAdorner release.

Running the program

Executing **SentenceAndTokenOffsets** with the Gettysburg Address text as input produces the output below. Only show the first two sentences are shown. Long output lines have been folded.

Each sentence and word token is preceded with an ordinal starting at 0, followed by starting and ending character offsets in brackets. For example:

- Sentence ordinal 0 starts at character 0 and ends at character 174.
- Word ordinal 0 starts at character 0 and ends at character 3.

The word offsets are relative to the start of the sentence. Consider the word at ordinal 1 in sentence ordinal 1, "we", which starts at character position 6 relative to the start of the sentence. Its absolute character offset is 175 (the offset of sentence 1) + 6 or 181.

```
0 [0,174]: Four score and seven years ago our fathers brought forth
 on this continent a new nation, conceived in Liberty, and dedicated
 to the proposition that all men are created equal.
          0 [0,3]: Four
          1 [5,9]: score
          2 [11,13]: and
          3 [15,19]: seven
          4 [21,25]: years
          5 [27,29]: ago
          6 [31,33]: our
          7 [35,41]: fathers
          8 [43,49]: brought
          9 [51,55]: forth
          10 [57,58]: on
          11 [60,63]: this
          12 [65,73]: continent
          13 [75,75]: a
          14 [77,79]: new
          15 [81,86]: nation
          16 [87,87]: ,
          17 [89,97]: conceived
          18 [99,100]: in
          19 [102,108]: Liberty
```

```
          20 [109,109]: ,
          21 [111,113]: and
          22 [115,123]: dedicated
          23 [125,126]: to
          24 [128,130]: the
          25 [132,142]: proposition
          26 [144,147]: that
          27 [149,151]: all
          28 [153,155]: men
          29 [157,159]: are
          30 [161,167]: created
          31 [169,173]: equal
          32 [174,174]: .
1 [175,308]:   Now we are engaged in a great civil war,
 testing whether that nation, or any nation, so conceived and
so dedicated, can long endure.
          0 [2,4]: Now
          1 [6,7]: we
          2 [9,11]: are
          3 [13,19]: engaged
          4 [21,22]: in
          5 [24,24]: a
          6 [26,30]: great
          7 [32,36]: civil
          8 [38,40]: war
          9 [41,41]: ,
          10 [43,49]: testing
          11 [51,57]: whether
          12 [59,62]: that
          13 [64,69]: nation
          14 [70,70]: ,
          15 [72,73]: or
          16 [75,77]: any
          17 [79,84]: nation
          18 [85,85]: ,
          19 [87,88]: so
          20 [90,98]: conceived
          21 [100,102]: and
          22 [104,105]: so
          23 [107,115]: dedicated
          24 [116,116]: ,
          25 [118,120]: can
          26 [122,125]: long
          27 [127,132]: endure
          28 [133,133]: .
```

```
package edu.northwestern.at.morphadorner.examples;

/*  Please see the license information at the end of this file. */

import java.io.*;
import java.text.*;
import java.util.*;

import edu.northwestern.at.utils.*;
import edu.northwestern.at.utils.corpuslinguistics.lexicon.*;
import edu.northwestern.at.utils.corpuslinguistics.postagger.guesser.*;
import edu.northwestern.at.utils.corpuslinguistics.sentencesplitter.*;
import edu.northwestern.at.utils.corpuslinguistics.tokenizer.*;

/** SentenceAndTokenOffsets: Display sentence and token offsets in text.
 *
 *  <p>
 *  Usage:
 *  </p>
 *
 *  <p>
 *  <code>
 *  java -Xmx256m edu.northwestern.at.morphadorner.example.SentenceAndTokenOffsets
InputFileName
 *  </code>
 *  </p>
 *
 *  <p>
 *  where "InputFileName" specifies the name of a text file to split
 *  into sentences and word tokens.  The default sentence splitter,
 *  tokenizer, part of speech guesser, and word and suffix lexicons
 *  are used.
 *  </p>
 *
 *  <p>
 *  Example:
 *  </p>
 *
 *  <p>
 *  <code>
 *  java -Xmx256m edu.northwestern.at.morphadorner.example.AdornAString mytext.txt
 *  </code>
 *  </p>
 *
 *  <p>
 *  The output displays each extracted sentence along with its starting and
 *  ending offset in the text read from the specified input file.
 *  For each sentence, a list of the extracted tokens in that sentence
 *  is displayed along with each token's starting and ending offset
 *  relative to the start of the sentence text.
 *  </p>
 */

public class SentenceAndTokenOffsets
{
    /** Main program.
     *
```

```
     *  @param  args    Command line arguments.
     */

    public static void main( String[] args )
    {
        try
        {
            if ( args.length > 0 )
            {
                displayOffsets( args[ 0 ] );
            }
            else
            {
                System.err.println(
                    "Usage: SentenceAndTokenOffsets inputFileName" );
            }
        }
        catch ( Exception e )
        {
            e.printStackTrace();
        }
    }

    /** Display sentence and token offsets in text.
     *
     *  @param  inputFileName   Input file name.
     */

    public static void displayOffsets( String inputFileName )
        throws Exception
    {
                                //  Wrap standard output as utf-8.

        PrintStream printOut    =
            new PrintStream
            (
                new BufferedOutputStream( System.out ) ,
                true ,
                "utf-8"
            );
                                //  Load text to split into
                                //  sentences and tokens.

        String sampleText   =
            FileUtils.readTextFile( inputFileName , "utf-8" );

                                //  Convert all whitespace characters
                                //  into blanks.  (Not necessary,
                                //  but makes the display cleaner below.)

        sampleText  = sampleText.replaceAll( "\\s" , " " );

                                //  Create default sentence splitter.

        SentenceSplitter splitter   = new DefaultSentenceSplitter();

                                //  Create part of speech guesser
```

```
                        //  for use by splitter.

PartOfSpeechGuesser partOfSpeechGuesser =
    new DefaultPartOfSpeechGuesser();

                        //  Get default word lexicon for
                        //  use by part of speech guesser.

Lexicon lexicon = new DefaultWordLexicon();

                        //  Set lexicon into guesser.

partOfSpeechGuesser.setWordLexicon( lexicon );

                        //  Get default suffix lexicon for
                        //  use by part of speech guesser.

Lexicon suffixLexicon       = new DefaultSuffixLexicon();

                        //  Set suffix lexicon into guesser.

partOfSpeechGuesser.setSuffixLexicon( suffixLexicon );

                        //  Set guesser into sentence splitter.

splitter.setPartOfSpeechGuesser( partOfSpeechGuesser );

                        //  Create default word tokenizer.

WordTokenizer tokenizer = new DefaultWordTokenizer();

                        //  Split input text into sentences
                        //  and words.

List<List<String>> sentences    =
    splitter.extractSentences
    (
        sampleText ,
        tokenizer
    );
                        //  Get sentence start and end
                        //  offsets in input text.

int[] sentenceOffsets   =
    splitter.findSentenceOffsets( sampleText , sentences );

                        //  Loop over sentences.

for ( int i = 0 ; i < sentences.size() ; i++ )
{
                        //  Get start and end offset of
                        //  sentence text.  Note:  the
                        //  end is the end + 1 since that
                        //  is what substring wants.

    int start       = sentenceOffsets[ i ];
    int end         = sentenceOffsets[ i + 1 ];
```

```
                        //  Get sentence text.

            String sentence =
                sampleText.substring( start , end );

                        //  Display sentence number,
                        //  start, end, and text.

            printOut.println(
                i + " [" + start + "," + ( end - 1 ) + "]: " + sentence );

                        //  Get word tokens in this sentence.

            List words  = sentences.get( i );

                        //  Get offsets for each word token
                        //  relative to this sentence.

            int[] wordOffsets   =
                tokenizer.findWordOffsets( sentence , words  );

                        //  Loop over word tokens.

            for ( int j = 0 ; j < words.size() ; j++ )
            {
                        //  Get start and end offset of
                        //  this word token.  Note:  the
                        //  end is the end + 1 since that
                        //  is what substring wants.

                start   = wordOffsets[ j ];
                end     =
                    wordOffsets[ j ] + words.get( j ).toString().length();

                        //  Display token number,
                        //  start, end, and text.

                printOut.println
                (
                    "          " + j + " [" + start + "," +
                    ( end - 1 ) + "]: " +
                    sentence.substring( start , end )
                );
            }
        }
    }
}

/*
Copyright (c) 2008, 2009 by Northwestern University.
All rights reserved.

Developed by:
   Academic and Research Technologies
   Northwestern University
   http://www.it.northwestern.edu/about/departments/at/
```

Example Four: Using An Adorned Text

Once you have MorphAdorned a text you probably want to do something with it. The AdornedXMLReader allows you to read an adorned file and extract a list of ExtendedAdornedWord entries. In addition to the morphological information encoded in adorned files, each ExtendedAdornedWord also provides extra information including the word and sentence number, whether a word occurs in main or paratext, whether a word occurs in verse, the XML tag path, and other things. AdornedXMLReader also allows you to extract sentences easily.

Sample text

We will use Nathaniel Hawthorne's short story "The Shaker Bridal" from **Twice Told Tales** as a sample text. The adorned XML text is found in eaf434.zip.

To load the word information from an adorned file, create an AdornedXMLReader and pass the name of the adorned file to read as a parameter. Here we load *eaf434.xml* which contains the adorned XML for "The Shaker Bridal."

```
AdornedXMLReader xmlReader = new AdornedXMLReader( "eaf434.xml" );
```

To extract the list of word IDs, use the **getAdornedWordIDs** method of AdornedXMLReader.

```
List<String> wordIDs   =
    xmlReader.getAdornedWordIDs();
```

Given a word ID you can use the **getExtendedAdornedWord** method of AdornedXMLReader to obtain the word information as an ExtendedAdornedWord.

To extract the list of sentences, use the **getSentences** method of AdornedXMLReader.

```
List<List<ExtendedAdornedWord>> sentences   =
    xmlReader.getSentences();
```

Generating displayable sentences

You can regenerate displayable sentences using the SentenceMelder class, which only requires a list of ExtendedAdornedWord entries. Here we print the first five sentences of an adorned file.

```
PrintStream printStream =
        new PrintStream
        (
            new BufferedOutputStream( System.out ) ,
            true ,
            "utf-8"
        );

printStream.println();

printStream.println
(
    "The first five sentences are:"
);
```

```
printStream.println();
printStream.println( StringUtils.dupl( "-" , 70 ) );

SentenceMelder melder   = new SentenceMelder();

for ( int i = 0 ;
    i < Math.min( 5 , sentences.size() ) ; i++ )
{
                //  Get text for this sentence.

    String sentenceText =
        melder.reconstituteSentence( sentences.get( i ) );

                //  Wrap the sentence text at column 70.

    sentenceText    =
        StringUtils.wrapText(
            sentenceText, Env.LINE_SEPARATOR , 70 );

                //  Print wrapped sentence text.

    printStream.println
    (
        ( i + 1 ) + ": " +
        sentenceText
    );
}
```

Extracting individual word information

Each sentence is a list of ExtendedAdornedWord entries. For example, we can extract word information for each word in the second sentence of a text as follows.

```
List<ExtendedAdornedWord> sentence  = sentences.get( 2 );

for ( int i = 0 ; i < sentence.size() ; i++ )
{
    ExtendedAdornedWord adornedWord = sentence.get( i );

    printStream.println( "Word " + ( i + 1 ) );

    printStream.println(
        "  Word ID          : " + adornedWord.getID() );
    printStream.println(
        "  Token            : " + adornedWord.getToken() );
    printStream.println(
        "  Spelling         : " + adornedWord.getSpelling() );
    printStream.println(
        "  Lemmata          : " + adornedWord.getLemmata() );
    printStream.println(
        "  Pos tags         : " +
        adornedWord.getPartsOfSpeech() );
    printStream.println(
        "  Standard spelling: " +
```

```
            adornedWord.getStandardSpelling() );
        printStream.println(
            "  Sentence number  : " +
            adornedWord.getSentenceNumber() );
        printStream.println(
            "  Word number      : " +
            adornedWord.getWordNumber() );
        printStream.println(
            "  XML path         : " +
            adornedWord.getPath() );
        printStream.println(
            "  is EOS           : " +
            adornedWord.getEOS() );
        printStream.println(
            "  word part flag   : " +
            adornedWord.getPart() );
        printStream.println(
            "  word ordinal     : " +
            adornedWord.getOrd() );
        printStream.println(
            "  page number      : " +
            adornedWord.getPageNumber() );
        printStream.println(
            "  Main or side text: " +
            adornedWord.getMainSide() );
        printStream.println(
            "  is spoken        : " +
            adornedWord.getSpoken() );
        printStream.println(
            "  is verse         : " +
            adornedWord.getVerse() );
        printStream.println(
            "  in jump tag      : " +
            adornedWord.getInJumpTag() );
        printStream.println(
            "  is a gap         : " +
            adornedWord.getGap() );
    }
}
```

The word information for the ninth word in the third sentence of "The Shaker Bridal" is:

```
Word ID          : eaf434-00440
Token            : Father
Spelling         : Father
Lemmata          : father
Pos tags         : n1
Standard spelling: Father
Sentence number  : 3
Word number      : 9
XML path         : \eaf434\body[1]\div[1]\p[1]\w[8]
is EOS           : false
word part flag   : N
word ordinal     : 21
page number      : 8
Main or side text: MAIN
is spoken        : false
```

```
is verse          : false
in jump tag       : false
is a gap          : false
```

Word Paths

The XML word path takes the form

```
\document\struct[i]\struct2[j]\struct3[k]...\w[n]
```

where "document" is the document name (e.g., eaf434 for "The Shaker Bridal"), the "struct[]" elements are the XML tags names with numbers assigned in order of appearance in a given document subtree, and "w[]" is the word number with the current parent structural element. The path gives a flattened version of the XML ancestry for each word.

The structure numbers start at 1 (not 0) and start over for each document subtree. For example, this means that paragraph numbers (e.g., "p" element numbers) start over for each "div" .

Here is a typical word path ID:

```
\eaf434\body[1]\div[1]\p[1]\w[26]
```

In this example "eaf434" is the document name. "body[1]" is the first (and usually only) body element. div[1] corresponds to the first text division of "The Shaker Bridal" (but could be something else for another document). p[1] is paragraph 1, and w[26] is the twenty-sixth word in paragraph 1.

Generating XML

Given a list of adjacent adorned words, we can use their word paths to reconstitute an XML representation of the text for those words. We do this by using an XML element stack and pushing and popping XML elements as needed to represent the structural changes indicated by the succession of word path IDs. The XML will not match the original exactly, but is good enough for display purposes. The word range need not be confined to any specific structural element -- we can easily generate well-formed XML even when the range of words spans structural elements and indeed even if the word range does not correspond to complete sentences. This would not be true if we extracted the actual original XML corresponding to the span of word IDs.

To get the XML representation we use the **generateXML** method of AdornedXMLReader by passing the starting and ending word IDs for which we want the XML. The **generateXML** method uses the method just described to generate well-formed XML even the range of text specified by the word IDs spans XML structural boundaries.

```
String xml  =
    xmlReader.generateXML( firstWordID , secondWordID );
```

Consider the span of word IDs "eaf434-02040" through "eaf434-02780". This is a "nice" range which is wholly contained within interior structural elements. The reconstituted XML follows.

```
<body>
<div>
<p>
His brethren of the north had now courteously
```

```
invited him to be present on an occasion, when the concurrence of
every eminent member of their community was peculiarly desirable.
</p>
<p>
The venerable Father Ephraim sat in his easychair, not
only hoary-headed and infirm with age, but worn down by a
lingering disease, which, it was evident, would very soon
transfer his patriarchal staff to other hands.
</p>
</div>
</body>
```

Now consider the span of word IDs "eaf434-03630" through "eaf434-05250". These word IDs run over a paragraph boundary (marked by the XML <p> tag). The reconstituted XML follows.

```
<body>
<div>
<p>
guided my choice aright.’
</p>
<p>
Accordingly, each elder looked at the two candidates
with a most scrutinizing gaze.
The man, whose name was Adam Colburn, had a face sunburnt with
labor in the fields, yet intelligent, thoughtful, and traced with
cares enough for a whole lifetime, though he had barely reached
middle age.
There was something severe in his aspect, and a rigidity
throughout his person, characteristics that caused him generally
to be taken for a schoolmaster; which vocation, in fact, he had
formerly exercised for several years.
The woman, Martha Pierson, was somewhat above thirty, thin and
pale, as a Shaker sister almost invariably is, and not entirely
free from that corpselike appearance, which the garb of the
sisterhood is so well calculated to impart.
</p>
<p>
‘This pair are still in the summer
</p>
</div>
</body>
```

Searching word paths

The word paths can be searched using regular expression pattern matches to do things like count words that appear in particular XML nesting structures, find all sibling words in a given paragraph, and so on.

Putting it altogether

You can peruse the Java source code below for **UsingAnAdornedText** which puts all the above code together in a runnable sample program. You will also find the source code in the `src/edu/northwestern/at/morphadorner/examples/` directory in the MorphAdorner release.

```
package edu.northwestern.at.morphadorner.examples;

/*  Please see the license information at the end of this file. */

import java.io.*;
import java.util.*;

import edu.northwestern.at.morphadorner.tools.*;
import edu.northwestern.at.utils.*;
import edu.northwestern.at.utils.corpuslinguistics.sentencemelder.*;

/** Using an adorned text.
 *
 *  <p>
 *  Usage:
 *  </p>
 *
 *  <p>
 *  <code>
 *  java -Xmx256m edu.northwestern.at.morphadorner.example.UsingAnAdornedText
adornedtext.xml id1 id2 id3 id4
 *  </code>
 *  </p>
 *
 *  <p>
 *  where
 *  </p>
 *
 *  <ul>
 *  <li><em>adorntext.xml</em> is a MorphAdorned XML file</li>
 *  <li>id1 is a word ID in the adorned XML file</li>
 *  <li>id2 is a word ID in the adorned XML file which follows id1</li>
 *  <li>id3 is a word ID in the adorned XML file</li>
 *  <li>id4 is a word ID in the adorned XML file which follows id4</li>
 *  </ul>
 */

public class UsingAnAdornedText
{
    /** Adorned XML reader. */

    protected static AdornedXMLReader xmlReader;

    /** The word IDs. */

    protected static List<String> wordIDs   =
        ListFactory.createNewList();

    /** UTF-8 print stream. */

    protected static PrintStream printStream;

    /** Main program. */

    public static void main( String[] args )
    {
        try
```

```
        {
            doit( args );
        }
        catch ( Exception e )
        {
            e.printStackTrace();
        }
    }

    /** Read adorned file and perform extraction operations. */

    public static void doit( String[] args )
        throws Exception
    {
        printStream     =
            new PrintStream
            (
                new BufferedOutputStream( System.out ) ,
                true ,
                "utf-8"
            );
                                //  Read adorned input file.

        xmlReader   = new AdornedXMLReader( args[ 0 ] );

                                //  Get list of word IDs.

        wordIDs     = xmlReader.getAdornedWordIDs();

                                //  Report number of words in input.
        printStream.println
        (
            "Read " +
            StringUtils.formatNumberWithCommas( wordIDs.size() ) +
            " words from " + args[ 0 ] + " ."
        );
                                //  Get sentences.

        List<List<ExtendedAdornedWord>> sentences   =
            xmlReader.getSentences();

                                //  Report number of sentences in input.
        printStream.println
        (
            "Read " +
            StringUtils.formatNumberWithCommas( sentences.size() ) +
            " sentences from " + args[ 0 ] + " ."
        );
                                //  Display the first five sentences.
                                //  We use a sentence melder.
                                //  We also wrap the sentence text
                                //  at column 70 for display purposes.

        printStream.println();

        printStream.println
        (
```

```
        "The first five sentences are:"
    );

    printStream.println();
    printStream.println( StringUtils.dupl( "-" , 70 ) );

    SentenceMelder melder   = new SentenceMelder();

    for ( int i = 0 ; i < Math.min( 5 , sentences.size() ) ; i++ )
    {
                            //  Get text for this sentence.

        String sentenceText =
            melder.reconstituteSentence( sentences.get( i ) );

                            //  Wrap the sentence text at column 70.

        sentenceText    =
            StringUtils.wrapText(
                sentenceText, Env.LINE_SEPARATOR , 70 );

                            //  Print wrapped sentence text.

        printStream.println
        (
            ( i + 1 ) + ": " +
            sentenceText
        );
    }

    printStream.println( StringUtils.dupl( "-" , 70 ) );
    printStream.println();

                            //  Word information for words in the
                            //  third sentence.

    if ( sentences.size() > 2 )
    {
        printStream.println();

        printStream.println
        (
            "Words in the third sentence:"
        );

        printStream.println();
        printStream.println( StringUtils.dupl( "-" , 70 ) );

        List<ExtendedAdornedWord> sentence  = sentences.get( 2 );

        for ( int i = 0 ; i < sentence.size() ; i++ )
        {
            ExtendedAdornedWord adornedWord = sentence.get( i );

            printStream.println( "Word " + ( i + 1 ) );

            printStream.println(
```

```
                "  Word ID          : " + adornedWord.getID() );
            printStream.println(
                "  Token            : " + adornedWord.getToken() );
            printStream.println(
                "  Spelling         : " + adornedWord.getSpelling() );
            printStream.println(
                "  Lemmata          : " + adornedWord.getLemmata() );
            printStream.println(
                "  Pos tags         : " +
                adornedWord.getPartsOfSpeech() );
            printStream.println(
                "  Standard spelling: " +
                adornedWord.getStandardSpelling() );
            printStream.println(
                "  Sentence number  : " +
                adornedWord.getSentenceNumber() );
            printStream.println(
                "  Word number      : " +
                adornedWord.getWordNumber() );
            printStream.println(
                "  XML path         : " +
                adornedWord.getPath() );
            printStream.println(
                "  is EOS           : " +
                adornedWord.getEOS() );
            printStream.println(
                "  word part flag   : " +
                adornedWord.getPart() );
            printStream.println(
                "  word ordinal     : " +
                adornedWord.getOrd() );
            printStream.println(
                "  page number      : " +
                adornedWord.getPageNumber() );
            printStream.println(
                "  Main or side text: " +
                adornedWord.getMainSide() );
            printStream.println(
                "  is spoken        : " +
                adornedWord.getSpoken() );
            printStream.println(
                "  is verse         : " +
                adornedWord.getVerse() );
            printStream.println(
                "  in jump tag      : " +
                adornedWord.getInJumpTag() );
            printStream.println(
                "  is a gap         : " +
                adornedWord.getGap() );
        }

        printStream.println( StringUtils.dupl( "-" , 70 ) );
        printStream.println();
    }
                        //  Generate xml for selected word ranges.

    generateXML( args[ 1 ] , args[ 2 ] );
```

```
            generateXML( args[ 3 ] , args[ 4 ] );
        }

        /** Generate XML from one word ID to another.
         *
         *  @param  firstWordID     First word ID.
         *  @param  secondWordID    Second word ID.
         */

        public static void generateXML
        (
            String firstWordID ,
            String secondWordID
        )
        {
                                    //  Generate xml for selected word range.

            String xml  = xmlReader.generateXML( firstWordID , secondWordID );

                                    //  Display generated xml.

            printStream.println();

            printStream.println( "Generated XML for words " +
                firstWordID + " through " + secondWordID + ":" );

            printStream.println();
            printStream.println( StringUtils.dupl( "-" , 70 ) );
            printStream.println( xml );
            printStream.println( StringUtils.dupl( "-" , 70 ) );
            printStream.println();
        }
    }

    /*
    Copyright (c) 2008, 2009 by Northwestern University.
    All rights reserved.

    Developed by:
       Academic and Research Technologies
       Northwestern University
       http://www.it.northwestern.edu/about/departments/at/

    Permission is hereby granted, free of charge, to any person
    obtaining a copy of this software and associated documentation
    files (the "Software"), to deal with the Software without
    restriction, including without limitation the rights to use,
    copy, modify, merge, publish, distribute, sublicense, and/or
    sell copies of the Software, and to permit persons to whom the
    Software is furnished to do so, subject to the following
    conditions:

        * Redistributions of source code must retain the above copyright
          notice, this list of conditions and the following disclaimers.

        * Redistributions in binary form must reproduce the above
          copyright notice, this list of conditions and the following
```

```
      disclaimers in the documentation and/or other materials provided
      with the distribution.

    * Neither the names of Academic and Research Technologies,
      Northwestern University, nor the names of its contributors may be
      used to endorse or promote products derived from this Software
      without specific prior written permission.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE CONTRIBUTORS OR
COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR
OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS WITH THE SOFTWARE.
*/
```

Example Five: Using the Sample Servlets

The MorphAdorner web site offers simple online examples of a number of MorphAdorner facilities. These examples are written as Java servlets. The source code for these servlets is included in the MorphAdorner release materials. The servlet code appears in the `src/edu/northwestern/at/morphadorner/servlets/` directory.

You can run these servlets on your own machine if you have more than a gigabyte of memory available (e.g., your machine has two gigabytes of memory).

1. Make sure you have installed recent working copies of Sun's Java Development Kit and Apache Ant on your system.
2. Open a command line window (on Windows) or a terminal window (on Unix).
3. Move to the directory to which you installed the MorphAdorner release.
4. Type the following command to build a servlet container to hold the MorphAdorner servlets.

   ```
   ant servlets
   ```

   When this command completes, the subdirectory **jetty** will contain a tiny web site consisting of a minimal version of the Jetty servlet server, the MorphAdorner code, and supporting libraries.
5. You may now run the servlets by executing the *runservlets* batch file (Windows) or script (Unix/MacOSX) in the main MorphAdorner release directory. Wait until you see a message like

   ```
   INFO: Started SelectChannelConnector@0.0.0.0:8200
   ```

   This may take several minutes on a slow machine.
6. Start a web browser and load the page:

   ```
   http://localhost:8200/morphadorner/
   ```

   You should see a page with links to each MorphAdorner servlet. Click on any one of the links to use the corresponding servlet.
7. To stop the Jetty server, press <ctrl>C in the command line window in which you started Jetty.

You can read more about the Jetty server at http://www.mortbay.org/jetty/.

If you are already running another program which uses TCP port 8200, you can either stop that other program or use a different TCP port for the MorphAdorner servlets. To do that, edit the *runservlets.bat* or *runservlets* script file in the main MorphAdorner directory and change **8200** to an unused TCP port on your machine.

If you encounter any "out of memory" errors, you can raise the memory size allocated to the jetty server from 900k to, say, 1 gigabyte. To do that, edit the *runservlets.bat* or *runservlets* script file in the main MorphAdorner directory and change **900k** to **1g**.

All of the servlets extend the base class **BaseAdornerServlet**. Each servlet generates plain HTML output. There are no template libraries, template files, or JSP files used. All input uses plain HTML forms. All output is plain HTML. The cascading style sheet *mstyle.css* found in `jetty/webapps/morphadorner/styles/mstyle.css` sets the font and margins for the servlet output. You can modify that file to add any other kind of styling you prefer.

Appendices

Appendix One: References And Links

References

The following books provide excellent introductions to natural language processing and the technical basis of the methods implemented in MorphAdorner.

- [Ananiadou and McNaught 2005] Sophia Ananiadou and John McNaught, eds. *Text Mining for Biology and Medicine*. Boston and London: Artech House, 2005.
- [Jurafsky and Martin 2000] Daniel Jurafsky and James H. Martin. *SPEECH and LANGUAGE PROCESSING: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition.* Upper Saddle River, New Jersey: Prentice-Hall, 2000.
- [Manning and Schütze 2000] Christopher D. Manning and Hinrich Schütze. **Foundations of Statistical Natural Language Processing.** Cambridge, Massachusetts: MIT Press, 2000. Also see the companion web site with errata and links to related sites.
- [Weiss et al. 2004] Shalom Weiss, Nitin Indurkhya, Tony Zhang, and Fred Damerau. **Text Mining: Predictive Methods for Analyzing Unstructured Information.** Springer Verlag, 2004.

Links

- Christopher Manning's list offers an annotated list of resources to statistical natural language processing and corpus-based computational linguistics.
- David W. Aha's machine learning page offers links to bibliographies, books, software, group, tutorials, and much more related to machine learning.

Appendix Two: Glossary of Natural Language Processing Terms

Abbott

Abbott is a framework for converting texts encoded in disparate versions of TEI into a common format called TEI Analytics. Abbott was developed by Brian Pytlik Zillig and Steve Ramsey at the University of Nebraska.

Adorned Corpus

An *adorned corpus* is a corpus in which the words in each work in the corpus have been adorned with morphological information such as lemma and part of speech.

Adornment

Adornment is the process of adding information such as morphological information to texts. We use the term "adornment" in preference to terms such as "annotation" or "tagging" which carry too many alternative and confusing meanings. Adornment harkens back to the medieval sense of manuscript adornment or illumination performed by monks - attaching pictures and marginal comments to texts.

Affix

An *affix* is a prefix or suffix which can be added to a morpheme or word to modify its meaning.

Attribute

An *attribute* in machine learning terms is a property of an object which may be used to determine its classification. For example, one attribute of a literary work is its genre: play, novel, short story, etc.

Bayes's Rule

Bayes's rule defines the conditional probability for two events A and B as follows:

Pr(A | B) = Pr(B | A) * Pr(A) / Pr(B)

Bigram

A *bigram* is an ordered sequence of two adjacent words, characters, or morphological adornments.

Bound Morpheme

A *bound morpheme* is a prefix or suffix which is not a word but which can be attached to a free morpheme to modify its meaning. For example, the bound morpheme "un" may be attached to the free morpheme "known" to form the new morpheme/word "unknown."

Chunk

A *chunk* or work part is a part of a work residing in a corpus. A chunk consists of an ordered series of words and associated morphological information with a label. A chunk may be treated as a bag of

words or ngrams for data analysis and navigation.

Collocate

Words which appear near each other in a text more frequently than we would expect by chance are called *collocates*. Collocates may be ngrams, but may also consist of multiple words with gaps between one or more of the words.

Corpus

A *corpus* is a collection of natural language texts. The plural is corpora. Each individual text in a corpus is called a work.

Data Herding

Data herding is the process of acquiring, combining, editing, normalizing, and warehousing texts so they can be used for further analysis.

Document Coordinate System

A *document coordinate system* assign a numeric vector of coordinate values to the position of each token in a document. A typical coordinate value might consist of a pair of line and column values based upon the printed form of the text, or a character offset and length pair based upon the digitized text.

Feature

See attribute.

Free Morpheme

A *free morpheme* is the basic or root form of a word. Bound morphemes can be attached to modify the meaning.

Hard tag

A *hard tag* is an SGML, HTML, or XML tag which starts a new text segment but does not interrupt the reading sequence of a text. Examples of hard tags include and <p>.

Hidden Markov Model

A *hidden Markov model* (HMM) is a statistical model in which the system being modeled is assumed to be a Markov Process with unknown parameters. The problem is to find the unknown parameters using values of the observable model parameters.

HMM

Abbreviation for hidden Markov model.

Jump tag

A *jump tag* is an SGML, HTML, or XML tag which interrupts the reading sequence of a text and starts a new text segment. Examples of jump tags include <note> and <speaker>.

Keyword Extraction

Keyword extraction extracts "interesting" phrases which characterize a text.

Language Recognition

Language recognition attempts to determine the language(s) in which a text is written. Literary texts are generally composed in one principal language with possible inclusions of short passages (letters, quotations) from other languages. It is helpful to categorize texts by principal language and most prominent secondary language, if any. We can use statistical methods based upon character ngrams and rank order statistics to determine the principal language of a text and list possible secondary languages.

Lemma

The *lemma* form or lexical root of an inflected spelling is the base form or head word form you would find in a dictionary. A lemma can also refer to the set of lexemes with the same lexical root, the same major word class, and the same word-sense.

Lemmatization

Lemmatization is the process of reducing an inflected spelling to its lexical root or lemma form. The lemma form is the base form or head word form you would find in a dictionary.

Lexeme

A *lexeme* is the combination of the lemma form of a spelling along with its word class (noun, verb. etc.).

Lexicon

A *lexicon* is a collection of words and their associated morphological information as used in a corpus.

Machine Learning

Machine learning occurs when a computer program modifies itself or "learns" so that subsequent executions with the same input result in a different and hopefully more accurate output. Machine learning methods may be supervised, i.e., using training data, or unsupervised, without using training data.

Markov Process

A *Markov process* is a discrete state random process in which the conditional probability distribution of the future states of the process depends only upon the present state and not on any past states.

MorphAdorner

MorphAdorner is a suite of Java programs which performs morphological adornment of words in a text. A high-level description of MorphAdorner's capabilities appears on the MorphAdorner home page.

Morpheme

A *morpheme* is a minimal grammatical unit of a language. A morpheme consists of a word or meaningful part of a word that cannot be divided into smaller independent grammatical units.

Multiword Unit

A *multiword unit* is a special type of collocate in which the component words comprise a meaningful phrase.

Named Entity

A *named entity* is a multiword unit consisting of a type of name such as a personal name, corporate name, place name, or date.

Ngram

An *ngram* is an ordered sequence of n adjacent words, characters, or morphological adornments.

NUPOS

NUPOS is a part of speech tag set devised by Martin Mueller to allow part of speech tagging of English texts from all periods as well as texts in classical languages. Further information about NUPOS appears in NUPOS and Morphology (page 76).

Part of Speech

The *part of speech* is the role a word performs in a sentence. A simple list of the parts of speech for English includes adjective, adverb, conjunction, noun, preposition, pronoun, and verb. For computational purposes, however, each of these major word classes is usually subdivided to reflect more granular syntactic and morphological structure.

Part of Speech Tagging

Part of speech tagging adorns or "tags" words in a text with each word's corresponding part of speech. Part of speech tagging relies both on the meaning of the word and its positional relationship with adjacent words.

Phone

A *phone* is an acoustic pattern which speakers of a particular natural language consider distinguishable and linguistically important. Distinct phones in one language may be grouped together and treated as the same sound in another language.

Phoneme

A *phoneme* is a group of phones considered to be the same sound by speakers of a specific natural language. One or more phonemes combine to form a morpheme.

Prefix

A *prefix* consists of characters comprising one or more bound morphemes which can be added to the front of a word to modify its meaning.

Pronoun Coreference Resolution

Pronoun coreference resolution matches pronouns with the nouns to which they refer. Some pronouns may not actually refer to a specific noun. For example, in the sentence "It is not clear how to proceed" the initial pronoun "It" does not refer to any specific noun.

Pseudo-bigram

A *pseudo-bigram* generalizes the computation of bigram statistical measures to ngrams longer than two words by splitting the original multiword units into two groups of words, each treated as a single "word".

Sentence Splitting

Sentence splitting assembles a tokenized text into sentences. Recognizing sentence boundaries is a difficult task for a computer and generally requires a combination of rules and statistical methods.

Soft tag

A *soft tag* is an SGML, HTML, or XML tag which does not interrupt the reading sequence of a text and does not start a new text segment. Examples of soft tags include <hi> and .

Spelling

The *spelling* is the orthographic representation of a spoken word. Words may have more than one spelling, particularly in texts dating from earlier periods when spelling was not standardized.

Spelling Standardization

Spelling standardization is the mapping of variant, often archaic, spellings to standard modern forms.

Stemming

Stemming removes affixes from a spelling. The resulting stem is not necessarily a proper lexeme. Stemming offers a simpler alternative to lemmatization. Stemming can be useful in information retrieval applications, but is much less useful in literary applications. Popular stemmers include the Martin Porter's stemmer and the Lancaster (Paice-Husk) stemmer.

Suffix

A *suffix* consists of characters comprising one or more bound morphemes which can be added to the

end of a word to modify its meaning.

Supervised Learning

Supervised learning is a machine learning technique which predicts the value of a given function for any valid input after having been presented with training examples (i.e. pairs of input and correct output).

Tagged Corpus

See adorned corpus.

TEI

Abbreviation for Text Encoding Initiative.

TEI Analytics

TEI Analytics is a literary DTD jointly developed by Martin Mueller at Northwestern University and Brian Pytlik Zillig and Steve Ramsey at the University of Nebraska. TEI Analytics is the default XML input format assumed by MorphAdorner. TEI Analytics is a minor modification of the P5 TEI-Lite schema, with additional elements from the Linguistic Segment Categories to support morphosyntactic annotation and lemmatization.

Text Encoding Initiative

The *Text Encoding Initiative* (TEI) Guidelines "are an international and interdisciplinary standard that enables libraries, museums, publishers, and individual scholars to represent a variety of literary and linguistic texts for online research, teaching, and preservation." More information may be found at the official Text Encoding Initiative site.

Trigram

A *trigram* is an ordered sequence of three adjacent words, characters, or morphological adornments.

Unsupervised Learning

Unsupervised learning is a machine learning method which fits a model to observed data without benefit of training data.

Viterbi Algorithm

The *Viterbi algorithm* allows searching a space containing an apparently exponential number of points to be searched in polynomial time. The Viterbi algorithm is frequently used in hidden Markov model statistical part of speech tagging applications to reduce the time complexity of seaches for the best tags for a sequence of spellings in a sentence.

Word

A *word* is the basic unit of a language. Words are composed of morphemes.

Word Sense Disambiguation

Word sense disambiguation is the process of distinguishing different meanings of the same word in different textual contexts. For example, a "bank" can be both a financial institution or a geographic location next to a river.

Word Tokenization

Word tokenization splits a text into words, whitespace, and punctuation.

Work

A *work* is a single text which is a member of a corpus. Each work consist of one or more text segments called work parts or chunks.

Work Part

See chunk.