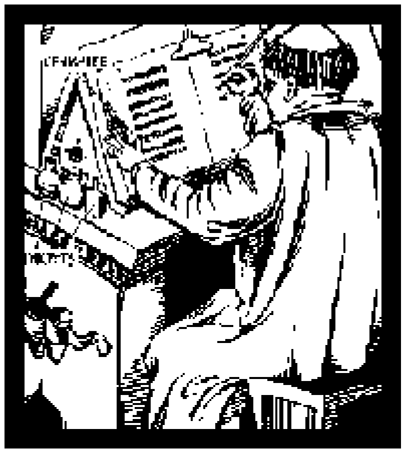
MorphAdorner

A Java Library for the Morphological Adornment of English Language Texts

Version 2.0.1. October 1, 2013.

Copyright © 2007, 2013 by Northwestern University.



– DRAFT – DRAFT – DRAFT --

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, as the scribal monk at right is doing.

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 demonstrations of each, by consulting the documentation section of the MorphAdorner web site.

MorphAdorner underwent 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. All three projects are now complete. While MorphAdorner has been used in these projects, it is actually a separate project in its own right.

MorphAdorner saw heavy use in the Monk project. The Monk project sought to adorn a large number of English language texts from the early Modern English period to the start of the twentieth century. The total number of adorned words was about 151.5 million words by project end in April 2009.

Starting in October 2012 we initiated a new MorphAdorner v2.0 project which sought to improve MorphAdorner's processing of several Text Creation Partnership corpora beyond what was attempted during the Monk project. These corpora included the Early English Books Online (EEBO) corpus, the Eighteenth Century Collections Online (ECCO), and the Evans Early American Imprint Collection. You can read more about MorphAdorner's processing of TCP texts (page 135).

We improved MorphAdorner's integration with Abbot. Abbot converts dissimilar collections of XML texts into a common interoperable form. Abbot was designed and implemented by Brian L. Pytlik Zillig, Stephen Ramsay, Martin Mueller, and Frank Smutniak.

Our goal in the Abbot and EEBO MorphAdorner collaboration is to turn the TCP texts into the foundation for a "Book of English," defined as:

- a large, growing, collaboratively curated, and public domain corpus of written English since its earliest modern form
- with full bibliographical detail
- and light but consistent structural and linguistic annotation.

We also replaced the makeshift demonstration servlets of MorphAdorner v1.0 with a separate MorphAdorner Server (page 192). The MorphAdorner Server allows access to many MorphAdorner facilities through HTTP-based web services. These services can be accessed using simple web forms or by any programming language which supports web forms and HTTP. The online examples of MorphAdorner facilities on the MorphAdorner web site use JavaScript to access the services provided by a local instance of the MorphAdorner Server.

Please see the modification history (page 7) for a general overview of the changes from MorphAdorner v1 to v2.

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 MorphAdorner Client Installation (page 4).

Cite MorphAdorner in a publication?

- See Citing MorphAdorner (page 12).

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

- See Part Two: Adorning A Text (page 13).

Tokenize a TEI XML file?

- See Tokenizing an XML Text With MorphAdorner (page 15).

Create an embedded adorner in a Java program?

- See Example One: Adorning a string With Parts Of Speech (page 152).

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

- See NUPOS and Morphology (page 94).

Find definitions of technical terms used this document?

- See Appendix Two: Glossary of Natural Language Processing Terms (page 347).

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

- See the MorphAdorner License (page 8).

Get help when I have problems with MorphAdorner?

- See MorphAdorner Support (page 12).

Deal with Java "OutOfMemory" errors?

- See Java OutOfMemory Errors (page 15).

MorphAdorner Client Installation

The file

`morphadorner-2.0.1.zip`

contains the MorphAdorner client source code, data, and libraries.

Current version: 2.0.1
Last update: September 30, 2013

The Mercurial repository

`http://bitbucket.org/pibburns/morphadorner`

contains the source code, data files, and build configuration files for generating the MorphAdorner release from scratch. The repository is intended for use by programmers who wish to modify the MorphAdorner code.

The MorphAdorner Server has its own download and installation instructions.

Quick Setup

If you downloaded the MorphAdorner release from the Mercurial repository on bitbucket.org, please go to the section "Installing and building MorphAdorner."

If you downloaded the ready-to-use morphadorner-2.0.1.zip file, proceed as follows. Expand the contents of the morphadorner-2.0.1.zip file into an empty directory. Make sure you retain the existing directory structure.

You must have the Java run-time environment installed on your machine to run MorphAdorner. If you do not, go to the section "Installing and Building MorphAdorner" for information on where to get a copy of the Java runtime.

Once you have Java installed you can proceed with running MorphAdorner.

File Layout of Morphadorner Client Release

File or Directory	Contents
README.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.
dist/	Holds generated morphAdorner.jar program file.
documentation/	MorphAdorner documentation.
gatelib/	Java libraries used by Gate.

ivy.xml	Apache Ivy dependencies definitions.
ivysettings.xml	Apache Ivy settings.
javadoc/	Javadoc (internal documentation).
lib/	Java library files.
misc/	Miscellaneous configuration files.
morphadornerlog.config	MorphAdorner logging configuration file.
src/	MorphAdorner client source code.
xslt/	XSLT stylesheets used by utilities.

Installing and Building MorphAdorner Client

Extract the files from morphadorner-2.0.1.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. The Java development kits for Windows, Mac OS X, and Linux systems may be obtained from

http://www.oracle.com/technetwork/java/javase/downloads/index.html

Alternatively, OpenJDK may be obtained from

http://openjdk.java.net/install/index.html

You must use a Java compiler which is compatible with Java 1.6 or higher.

Apache Ant may be obtained from

http://ant.apache.org

Move to the directory in which you extracted morphadorner-2.0.1.zip, and type:

```
ant
```

This should build MorphAdorner successfully. The morphadorner.jar file will be placed in the "dist" subdirectory.

Type

```
ant javadoc
```

to generate the javadoc (internal documentation) into subdirectory "javadoc".

Type

```
ant clean
```

to remove the effects of compilation.

Documentation

Printable documentation, in Adobe Acrobat PDF format, will appear in the documentation/morphadorner.pdf file in the MorphAdorner release. This documentation is still in progress and not yet available.

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.

Before running MorphAdorner on Unix-like systems you will need to mark the Unix script files as executable before using them. 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
```

The sample batch file **adornncf.bat** and the corresponding Linux script **adornncf** shows how to run MorphAdorner to adorn simple TEI 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 or the script **adornncfa** for files in which quote marks are distinguished from apostrophes.

For example, to adorn TEI XML files in directory /myfiles into the output directory /myoutputfiles on Unix-like systems, open a terminal window in the MorphAdorner directory and type

```
./adornncf /myoutputfiles /myfiles/*.xml
```

On Windows you would open a console window in the MorphAdorner directory and type

```
adornncf \myoutputfiles \myfiles\*.xml
```

Please see the documentation section "Adorning a Text" in the online web site or the printable PDF for more information on these and other sample batch files and scripts in the MorphAdorner release.

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

Modification History

Version 2.0.1. September 25, 2013.

1. Correct mishandling of some empty elements in TEI XML files.

Version 2.0.0. September xx, 2013.

Initial public release of MorphAdorner v2.0.0.

Main changes since 1.0.1.

1. Created a Mercurial repository to hold the source code and build materials: http://bitbucket.org/pibburns/morphadorner/
2. Reorganized the code base to place all the linguistics processing code under the edu.northwestern.at.morphadorner.corpuslinguistics parent package.
3. Replaced the old sample servlets with standalone MorphAdorner server. This has its own code base and release materials.
4. Multiple improvments to basic tokenization and sentence-splitting facilities, including addition of basic support for tokenizing and sentence-splitting of texts written in languages other than English.
5. Upgraded language recognition facilities with a more recent algorithm from Nakatani Shuyo.
6. Improved part of speech adornment particularly for Early Modern English. Among other changes, the suffix analysis used to select candidate parts of speech for unknown words now disallows candidate parts of speech to be assigned from closed word classes.
7. Added a utility for converting the "base" MorphAdorner adorned output to a more TEI P5-like format.
8. Added a number of support utilities for improving the processing of corpora from the Text Creation Partnership.
9. Added some extra fields to the tabular (verticalized) output from adorned files.
10. Added support for different abbreviation lists for main and paratext (important for drama texts).
11. Added implementation of the PUNKT algorithm of Tibor and Strunk for extracting potential abbreviation lists from a plain-text corpus.
12. Added generic utility to apply and XSL transformation to a set of input files.
13. Added classes to support basic text summarization, hyphenation, and syllable counting.
14. Multiple other minor bug fixes and improvements.

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.

Copyright © 2006-2013 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:

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 12.

Third-party Licenses

1. Apache Ant
Copyright © 2000-2013 The Apache Software Foundation.
Licensed under the Apache Software License 2.0. For complete license information, please see Apache Ant license.
2. Apache Commons
Copyright © 2000-2013 The Apache Software Foundation.
Licensed under the Apache Software License 2.0. For complete license information, please see Apache Ant license.
3. Apache Ivy
Copyright © 2007-2013 The Apache Software Foundation.
Licensed under the Apache Software License 2.0.

4. 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.
5. 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..
6. 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.
7. Cybozu Labs Language Detector
   Copyright (c) 2010-2011 Cybozu Labs, Inc.
   All rights reserved.
   The Cybozu Labs language detector is licensed under the Apache Software License 2.0. The version of the code used in MorphAdorner includes the original code base by Nakatani Shuyo, with modifications by Robert M. Theis as well as local Northwestern University modifications.
8. Double Metaphone
   Written by Ed Parrish.
   Licensed under an Apache license.
9. 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.
10. International Components for Unicode (ICU4J)
    Copyright © International Business Machines Corporation and others.
    Licensed under the ICU license.
11. 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..
12. 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..
13. Jackson JSON processing
    Copyright © 2007-2013 by Tatu Saloranta.
    All rights reserved.
    Licensed under the Apache Software License 2.0.
14. 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.

15. 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.
16. 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.
17. 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.
18. Jettison
Copyright 2006 Envoi Solutions LLC.
Licensed under the Apache Software License 2.0.
19. JlinkGrammar
JLinkGrammar is licensed under a BSD-like license (although early references also suggested it could be licensed under the GNU General Public license).
20. JSONIC
Licensed under the Apache Software License 2.0.
21. JUnit
JUnit is licensed under the terms of the Eclipse Public License v1.0.
22. 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.
23. 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.

24. Mersenne Twister
    Liberal use license contained in the source file. For complete license information, please see Mersenne twister license.
25. Pluralizer
    Original pluralizer was code written by Tom White and licensed under the Apache Software License 2.0.
26. 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..
27. 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/.
28. Restlet
    Of the several available licenses for Restlet, we selected the Apache Software License 2.0.
29. SAX
    The SAX processors are in the public domain.
30. Simple web server
    Simple is licensed under the Apache Software License 2.0.
31. Text Corpus Format (TCFXB)
    Licensed under the GNU Lesser General Public License.
32. TeXHyphenator-J
    Licensed under the GNU Lesser General Public License.
33. 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.
34. XGTagger
    Licensed under the CeCILL license. For complete license information, please see CeCILL license.
35. XStream
    Copyright (c) 2003-2006, Joe Walnes
    Copyright (c) 2006-2009, 2011 XStream Committers
    All rights reserved.
    Licensed under a BSD style license. For complete license information, please see the XStream 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.

The development of MorphAdorner v2.0 also received further support from The Andrew W. Mellon Foundation, the Center for Library Inititatives at the CIC, the Ford Center for Global Citizenship at Northwestern's Kellogg School of Management, the Northwestern University Library, and Proquest.

John L. Norstad contributed most of the section entitled “NUPos and Morphology” (page 94). Martin Mueller contributed parts of the section entitled “Processing Text Creation Partnership Files” (page 135).

Citing MorphAdorner

You may cite MorphAdorner as follows in a publication.

> Burns, Philip R. (2013) "MorphAdorner v2: A Java Library for the Morphological Adornment of English Language Texts."
> Evanston, IL. Northwestern University.
> Retrieved Sep 30, 2013 from
> <http://morphadorner.northwestern.edu/morphadorner/download/morphadorner.pdf> .

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 20) 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 1024 megabytes. This means your PC needs to have a minimum of one gigabyte of memory. The 1024 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 -Xmx1024m`*** 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.

Tokenizing an XML Text

If you just want to tokenize XML texts rather than fully adorn them, select the batch file/script which most nearly matches the type of corpus you have, and add the "-k" command line option to the MorphAdorner invocation. This causes MorphAdorner to emit the tokenized text -- that is, the <w>, <pc> with word IDs include along with the whitepspace marker <c> elements. The other word-level adornments are not output. See MorphAdorner Command Line Syntax (page 18) for details on the MorphAdorner command line.

As an example, let's modify the **adornncf** script so that it only tokenizes XML files rather than fully adorning them.

```
#!/bin/sh
java -Xmx1024m -Xss1m -cp .:bin/:dist/*:lib/* \
        edu.northwestern.at.morphadorner.MorphAdorner \
        -p ncf.properties \
        -l data/ncflexicon.lex \
        -t data/ncftransmat.mat \
        -u data/ncfsuffixlexicon.lex \
        -a data/ncfmergedspellingpairs.tab \
        -s data/standardspellings.txt \
        -w data/spellingsbywordclass.txt \
        -o $1 \
        $2 $3 $4 $5 $6 $7 $8 $9
```

All we need to do is to add the "-k" command line parameter and save the script to a new file, say, **tokenizencf**:

```
#!/bin/sh
java -Xmx1024m -Xss1m -cp .:bin/:dist/*:lib/* \
        edu.northwestern.at.morphadorner.MorphAdorner \
        -k \
        -p ncf.properties \
        -l data/ncflexicon.lex \
        -t data/ncftransmat.mat \
        -u data/ncfsuffixlexicon.lex \
        -a data/ncfmergedspellingpairs.tab \
        -s data/standardspellings.txt \
        -w data/spellingsbywordclass.txt \
        -o $1 \
        $2 $3 $4 $5 $6 $7 $8 $9
```

You can modify the Windows batch file similarly.

```
java -Xmx1024m -Xss1m -cp bin\;dist\*;lib\*; ^
        edu.northwestern.at.morphadorner.MorphAdorner ^
        -k
        -p ncf.properties ^
        -l data/ncflexicon.lex ^
        -t data/ncftransmat.mat ^
        -u data/ncfsuffixlexicon.lex ^
        -a data/ncfmergedspellingpairs.tab ^
        -s data/standardspellings.txt ^
        -w data/spellingsbywordclass.txt ^
        -o %1 ^
        %2 %3 %4 %5 %6 %7 %8 %9
```

For the Unix script, remember to make it executable.

```
    chmod 755 tokenizencf
```

You can now tokenize one or more TEI XML files by invoking **tokenizencf**:

```
    ./tokenizencf /mytokenizedfiles /myteifiles/*.xml
```

MorphAdorner tokenizes all the TEI XML files in the directory **/myteifiles** and writes the tokenized versions in the directory **/mytokenizedfiles** .

The equivalent Windows command is:

```
    tokenizencf \mytokenizedfiles \myteifiles\*.xml
```

Here is a brief sample of tokenized TEI XML text.

```
                <l>
                  <w xml:id="K135834_000-000990">Or</w>
                  <c> </c>
```

```
      <w xml:id="K135834_000-001000">those</w>
      <c> </c>
      <w xml:id="K135834_000-001010">whom</w>
      <c> </c>
      <w xml:id="K135834_000-001020">choice</w>
      <c> </c>
      <w xml:id="K135834_000-001030">and</w>
      <c> </c>
      <w xml:id="K135834_000-001040">common</w>
      <c> </c>
      <w xml:id="K135834_000-001050">good</w>
      <c> </c>
      <w xml:id="K135834_000-001060">ordain</w>
      <pc xml:id="K135834_000-001070" unit="sentence">.</pc>
    </l>
```

You can later fully adorn the tokenized files by inputting them to MorphAdorner using, e.g., **adornncf**.

The tokenized format is useful if you wish to edit the tokenized texts before performing full adornment. See Processing Text Creation Partnership Files (page 135) for an example of how this proved useful when processing those files.

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
      -d default.properties
      -h
      -k
      -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.
d	Default MorphAdorner properties file. Usually the *morphadorner.properties* file which appears in the main MorphAdorner installation directory.
h	Displays a brief help message to the standard output file.
k	Only tokenize XML input files. Overrides settings which produce other word-level adornments besides the word ID.
l	A word lexicon file in MorphAdorner format.
o	The directory into which adorned output files are written.
p	A MorphAdorner Configuration Settings (page 20) 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 properties file specified by the *d*= parameter (*morphadorner.properties* if no *p*= parameter appears) 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 18). 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.
ece.properties	Eighteenth Century English 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**.
abbreviations.main.abbreviations_url	Specifies the URL for an extra list of abbreviations to be used only for main text.
abbreviations.main.abbreviations_url	Specifies the URL for an extra list of abbreviations to be used only for side text, e.g., paratext.
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*.
corpus.name	The name of the corpus for this configuration. Usually a short string such as "ncf" for "nineteenth century fiction." Used by the MorphAdorner server when displaying the available configurations. The server ignores MorphAdorner configurations which do not have the *corpus.name* set.
corpus.description	Longer description the corpus for this configuration. Used by the MorphAdorner server when displaying the available configurations. The server ignores MorphAdorner configurations which do not have the *corpus.dcescription* set.
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 (page 124). 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 (page 121) 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_mileston es	*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.tokenlabel.emit	True to emit a token label which contains an image number for the current page, a letter for the current column on the page, the word number multiplied by the label spacing within the column. This is used when adorning Text Creation Partnership texts to relate words to the source page images.
xml.tokenlabel.attribute	The token label attribute name. The default is *n*.
xml.token.label.spacing	Increment value for generating token labels. The default is 10.
xml.token.label.prependworkname	Set to true to prepend the work name to the token label. The default is false; the work name will not be prepended to the token label.
xml.use_pc_to_mark_end_of_sentence	Add a *unit="sentence"* attribute to mark the end of a sentence. This is the default in MorphAdorner v2 (in v1, the *eos* was used instead).

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://morphadorner.northwestern.edu/morphadorner/schemata/TEIAnalytics.rng .

Part Four: Utilities

MorphAdorner provides a number of utility programs. The utility program names have fairly long Java classpaths, and the utility name is in mixed case. To simplify their use, all of the utilities have associated Windows batch files and Unix/Linux script files whose names are the utility name in lowercase.

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 <div> 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. Adding unclear attributes to words with gaps

Adding unclear attributes to words with gaps

AddUnclear adds a *type="unclear"* attribute to tokens containing character gaps in tokenized or adorned TEI XML files.

Usage:

```
    adduclear outputdirectory input1.xml input2.xml ...
```

where

- *outputdirectory* is the output directory containing the resultant XML files with type="unclear" attributes added to tokens containing character gaps.
- *input*.xml* are the input tokenized XML files.

Character gaps in tokens are indicated by the presence of the unicode black circle (\u25CF) in a token.

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.

Applying an XSLT transformation to XML files

ApplyXSLT applies an XSL transformation to set of input XML files to produce transformed XML files.

Usage:

```
    applyxslt outputdirectory script.xsl input1.xml
    input2.xml ...
```

where

outputdirectory	Output directory for files processed by applying the XSLT script to the input files.
script.xsl	XSLT script file.
input1.xml input2.xml ...	Input xml files.

XSL transformations, specified in XSLT files, are widely used for modifying the contents of XML files. The MorphAdorner Server (page 192) implements several of its facilities using XSL transformations, including moving notes in TEI files to the end of <div> elements, and extracting the text from a TEI file.

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

CompareAdornedFiles compares two adorned files and writes a change log indicating the differences between the two.

Usage:

```
    compareadornedfiles oldadorned.xml newadorned.xml diffs.xml
```

where

- *oldadorned.xml* is the "old" adorned TEI XML file.
- *newadorned.xml* is the "new" (modified) version of the adorned file.
- *diffs.xml* is the file name to receive the change log of the token-based differences from the old to the new adorned file.

Change Log Format

CompareAdornedFiles uses a simple XML format to contain a list of token-based changes. The format of this file is as follows.

```
<ChangeLog>
 <changeTime>The time the change file was created.</changeTime>
  <changeDescription>A description of the changes.</changeDescription>
   <changes>
    <change>
     <id>xml:id of token to be changed.</id>
     <changeType>addition, modification, or deletion.</changeType>
     <fieldType>Type of field to change: text or attribute.</fieldType>
     <oldValue>Old field value.</oldValue>
     <newValue>New field value.</newValue>
     <siblingID>xml:id of sibling word for a word being added.</siblingID>
     <blankPrecedes>true if blank precedes the token, else false.</blankPrecedes>
    </change>
      ...
      (more <change> entries)
      ...
   </changes>
</ChangeLog>
```

This simple XML formatted change file allows a file to be transformed to a corrected file using a utility in the MorphAdorner suite. A file can be "untransformed" from the corrected version to the uncorrected version using the same change file. A likely use case for the change log is an edition that wants to use long 's' and other original spellings.

Here is an example of a change log entry which records the replacement of a long "s" with a plain "s".

```
<ChangeLog>
  <changeTime>2013-07-09 13:04:17.149 CDT</changeTime>
  <changeDescription>Changes from \tokenized\K000379.000.xml to \tokenized-no-
wordbreaks\K000379.000.xml as determined by CompareAdornedFiles.</changeDescription>
  <changes>
    <change>
      <id>K000379_000-00080</id>
      <changeType>modification</changeType>
      <fieldType>text</fieldType>
      <oldValue>Addreſs'd</oldValue>
      <newValue>Address'd</newValue>
      <blankPrecedes>true</blankPrecedes>
    </change>
        ...
  </changes>
</ChangeLog>
```

A change log may be used to transform one version of an adorned file into another using the UpdateAdornedFile (page 74) utility.

Comparing Tabular 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.

Converting an adorned file to Sketch engine format

AdornedToSketch converts one or more adorned files to the verticalized input required by the Sketch or NoSketch corpus search engines.

Usage:

```
adornedtosketch sketchinput.txt corpusname adorned1.xml
adorned2.xml ...
```

where

- **sketchinput.txt** specifies the output filename of the verticalized representation required for input to the Sketch or NoSketch engines.
- **corpusname** specifies the corpus name to be used when creating the Sketch engine input.
- **adorned1.xml adorned2.xml ...** specifies the input MorphAdorned XML files from which to produce the Sketch engine input.

Known flaw: AdornedToSketch does not generate the "glue" elements which bind punctuation marks to word tokens. Searching the corpus still works fine in the Sketch or NoSketch engine, but the punctuation marks are displayed detached from any token to which they would normally be attached.

The Sketch engine, and its simpler sibling the NoSketch engine, are corpus query systems based upon the thesis work of Pavel Rychlý. The engines are products of Lexical Computing Ltd., headed by computational linguist Adam Kilgarriff.

Converting an adorned file to TCF format

AdornedToTCF04 converts one or more adorned files to the Text Corpus Format (TCF) v0.4 used by the CLARIN-D project.

Usage:

```
    adornedtotcf04 outputdirectory adorned1.xml adorned2.xml ...
```

where

- **outputdirectory** specifies the output directory to receive the TCF v0.4 formatted files.
- **adorned1.xml adorned2.xml ...** specifies the input MorphAdorned XML files from which to produce the TCF v0.4 versions.

The Text Corpus Format (TCF) is used by the European CLARIN-D project to allow interchange of corpora among different web-based services. TCF is an XML-based format which consists of a plain text representation of a work along with a series of annotation layers.

AdornedToTCF04 converts one or more MorphAdorned TEI XML files to TCF format. The text (without tags) is extracted and output, along with the following annotation layers:

- Tokens (using the MorphAdorner word IDs)
- Lemmata
- Part of speech tags
- Sentences

Converting a base adorned file to a simple TEI P5-like format

AdornedToSimpleTEIP5 converts a base-level MorphAdorner file to a simpler, more TEI P5-like format.

Usage:

```
adornedtosimpleteip5 outputdirectory [usereg|usechoice]
interpgrp.xml goodfiles.txt badfiles.txt adorned1.xml
adorned2.xml ...
```

where

- **outputdirectory** specifies the output directory for the base adorned XML files.
- **usereg** specifies that the standardized spelling should be emitted as a *reg*= attribute, while **usechoice** specifies that the standardized spelling should be emitted using TEI <*choice*> structure.
- **interpgrp.xml** specifies the file name for a section of TEI XML which defines an interpGrp element for the part of speech tags. This can be an empty file in which case the interpGrp is not added to each output TEI XML file.
- **goodfiles.txt** specifies the name of a file to receive the names of TEI XML files successfully converted to simple TEI P5 format.
- **badfiles.txt** specifies the name of a file to receive the names of TEI XML files which could not be successfully converted to simple TEI P5 format.
- **adorned1.xml adorned2.xml ...** specifies the input MorphAdorned XML files from which to produce simple TEI P5 versions.

AdornedToSimpleTEIP5 converts the base form of an adorned TEI file, which adds custom attributes to word <w> elements, to a simpler more TEI P5 compatible format as follows.

- The *pos* attribute, which specifies the part of speech, is changed to the P5 standard *ana* attribute. The part of speech is prefixed with a "#".
- The *lem* attribute, which specifies the lemma (headword) for the word, is changed to the P5 standard *lemma* attribute.
- The *reg* attribute, which specifies the modernized spelling, is handled as described below.
- The other non-standard attributes *ord*, *spe*, *tok*, etc. are dropped.

In standard TEI P5 you cannot store a standardized spelling in a *reg* attribute. One approach is to use a combination of <choice>, <orig>, and <reg> elements to make each <w> element carry its part of a double stream of original and standardized spellings, as in this adorned encoding of "wylle anone" from an early 16th century text:

```
<w xml:id ="someid1" lemma="will" ana="#vmb">
<choice>
<orig>wylle</orig>
<reg>will</reg>
</choice>
</w>
<w xml:id ="someid2" lemma="anon" ana="#av">>
<choice>
<orig>anone</orig>
<reg>anon</reg>
```

```
   </choice>
   </w>
```

Alternatively, you can customize P5 and restore a *reg* attribute that lets you encode the same phenomena in a manner that programmers -- and in particular programmers with limited skills -- are likely to find more intuitive and economical:

```
<w xml:id ="someid1" lemma="will" reg= "will" ana="#vmb">wylle</w>
<w xml:id ="someid2" lemma="anon" reg ="anon" ana="#av">anone</w>
```

For many purposes using an attribute is preferable to a choice element because the attribute leaves the token sequence undisturbed, and the added attribute value can be stored in the standard MorphAdorner change log format.

AdornedToSimpleTEIP5 allows you to use either of these two approaches.

- Select AdornedToSimpleTEIP5's **usechoice** option to store the standard spelling using a <*choice*> structure.
- Select AdornedToSimpleTEIP5's **usereg** option to store the standard spelling using a *reg* attribute.

Important: Many other MorphAdorner utilities do not yet work properly with simplified adorned texts created using the <*choice*> structure.

Defining the parts of speech using an interGrp element

Strictly speaking, a TEI interpGrp element should be added to each TEI XML output file to specify the definitions for the parts of speech used. The MorphAdorner release materials include a **nuposinterpgrp.xml** file in the release **data/** directory which defines an interpGrp for the NUPos tag set. This file can be specified as the value of AdornedToSimpleTEIP5's *interpgrp.xml* parameter.

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 77) 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 (page 77).

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 allowedpostagsfilename
```

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.
- *allowedpostagsfilename* specifies the name of a file containing a list of part of speech tags to use when constructing the suffix lexicon. Omit the tags for parts of speech for closed word classes to which new words should not be added. The MorphAdorner release provides the file *nuposallowedpostags.txt* in the release **data** directory which defines a default set of NUPos tags to use when creating a suffix lexicon.

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.

Extracting Abbreviation Using PUNKT

PunktAbbreviationDetector finds abbreviations in a set of untagged utf-8 encoded texts using the Punkt algorithm of Tibor Kiss and Jan Strunk.

The Punkt algorithm adapts collocation extraction methodology to the problem of determining when a period-terminated token is an abbreviation. For each token ending with a period, PUNKT compiles counts of the occurrences of the token with and without the trailing period. When the token appears statistically far more often with a period than without, it is a candidate abbreviation. Some additional heuristics refine the selection process.

This algorithm works well for English and other Western European languages. Its main weakness is that it fails when the collection of texts being analyzed contains many instances in which genuine abbreviations appear without the terminating period. Biblical references in early modern English texts provide a good example. Biblical book names that are abbreviated often do not end with a period. As a result, Biblical book name abbreviations in early texts will typically not be recognized as abbreviations.

Usage:

```
    punktabbreviationdetector isolangcode abbrevs.txt text1.txt
    text2.txt ...
```

where

- **isolangcode** specifies the two or three character ISO language code in which the texts to be analyzed are written.
- **abbrevs.txt** specifies the name of the output file to receive the abbreviations extracted from the texts.
- **text1 text2 ...** specify the names of utf-8 encoded text files from which to extract potential abbreviations.

Reference

Kiss, Tibor and Strunk, Jan (2006). Unsupervised Multilingual Sentence Boundary Detection. *Computational Linguistics* 32: 485-525.

Extracting text from a TEI XML file

ExtractTEIText applies an XSL transformation to an input TEI XML file to extract the text from the body of the file.

Usage:

```
    extractteitext input.xml output.xml
```

where

input.xml	The input TEI XML file.
output.txt	The output file containing the text extracted from the input TEI file.

The XSLT transformation used to extract the text is defined in the **tei2text.xsl** file in the **xslt** directory of the MorphAdorner release. This transformation works well for unadorned TEI files, not so well for adorned files. You can use the Unadorn (page 74) utility to unadorn an adorned file before extracting the text.

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.

Fixing Superscripts

SuperFixer marks "^" characters with special tags. The "^" is used to mark superscripted characters in Text Creation Partnership files.

Usage:

```
    superfixer outputdirectory input1.xml input2.xml ...
```

where

- *outputdirectory* is the output directory containing the resultant XML files.
- *input*.xml* are the input TEI XML files.

<zzzzlj>token</zzzzlj> is added to surround tokens containing "^" superscript markers. MorphAdorner removes these non-standard markers during the adornment process.

Tokens which end in ^d where "d" is a single digit are converted to the token followed by a “d”. This provides for inserting the missing targets of these apparent note references at a later editing stage.

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 Annolex corrections with adorned TEI XML
AnnoLex is a collaborative data curation tool for use with Text Creation Partnership texts. Annolex allows for the identification and correction of incompletely or incorrectly transcribed words. It can also be used for the manual correction of algorithmically applied lemmatization and part-of-speech tagging. Annolex was developed by Craig Berry and Martin Mueller.

MergeAnnolexCorrectionsIntoAdornedXML merges corrections developed in Annolex back into the source adorned TEI XML files.

Usage:

```
mergeannolexcorrectionsintoadornedxml correctionsdirectory
outputdirectory inputfiles
```

where

- **correctionsdirectory** is the input directory with Annolex correction files in tabular format.
- **outputdirectory** is the output directory for the corrected adorned TEI XML files.
- **inputfiles** contains the input adorned XML files with which to merge the AnnoLex produced corrections. These must be in the base adorned format, not the simplified TEI P5 format.

The corrections file is a tab-separated utf-8 file containing the following columns.

1. Work ID.
2. Word ID.
3. Old spelling.

4. Corrected spelling.
5. Standard spelling.
6. Corrected lemmata.
7. Corrected parts of speech.
8. Operation: 1=update, 2=insert, 3=delete, 5=delete nearest gap.

The corrected spelling, lemmata, and parts of speech may all be empty when the operation is 3 (delete).

The value of the "ord" (word ordinal) attribute for each word is adjusted to account for inserted and deleted words. The value of the "reg" (standard spelling) and "tok" attributes (original token) are generated as needed for updated and inserted words.

Whitespace markers " " are added and deleted as needed when tokens are added or deleted. In general, most added punctuation and symbols do not require added whitespace markers. When tokens are deleted, sequences of "<c> </c><c> </c> ..." are compressed to a single "<c> </c>" entry.

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 59) 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 58) 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 Annolex corrections with adorned TEI XML files

AnnoLex is a collaborative data curation tool for use with Text Creation Partnership texts. Annolex allows for the identification and correction of incompletely or incorrectly transcribed words. It can also be used for the manual correction of algorithmically applied lemmatization and part-of-speech tagging. Annolex was developed by Craig Berry and Martin Mueller.

MergeAnnolexCorrectionsIntoAdornedXML merges corrections developed in Annolex back into the source adorned TEI XML files.

Usage:

```
MergeAnnolexCorrectionsIntoAdornedXML correctionsdirectory
outputdirectory inputfiles
```

where

- **correctionsdirectory** is the input directory with Annolex correction files in tabular format.
- **outputdirectory** is the output directory for the corrected adorned TEI XML files.
- **inputfiles** contains the input adorned XML files with which to merge the AnnoLex produced corrections. These must be in the base adorned format, not the simplified TEI P5 format.

The corrections file is a tab-separated utf-8 file containing the following columns.

1. Work ID.
2. Word ID.
3. Old spelling.
4. Corrected spelling.
5. Standard spelling.
6. Corrected lemmata.
7. Corrected parts of speech.
8. Operation: 1=update, 2=insert, 3=delete, 5=delete nearest gap.

The corrected spelling, lemmata, and parts of speech may all be empty when the operation is 3 (delete).

The value of the "ord" (word ordinal) attribute for each word is adjusted to account for inserted and deleted words. The value of the "reg" (standard spelling) and "tok" attributes (original token) are generated as needed for updated and inserted words.

Whitespace markers " " are added and deleted as needed when tokens are added or deleted. In general, most added punctuation and symbols do not require added whitespace markers. When tokens are deleted, sequences of "<c> </c><c> </c> ..." are compressed to a single "<c> </c>" entry.

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.

Moving notes in TEI XML files

MoveTEINotes applies an XSL transformation to an input TEI XML file to move all the notes to a separate <div> element.

Usage:

```
    moveteinotes input.xml output.xml
```

where

input.xml	The input TEI XML file.
output.txt	The output file TEI XML file with the notes moved to a separate <div> element.

In printed source texts, <note> elements generally do not interrupt the reading order, because all the notes are either placed in the margin or at the bottom of the page. In the XML transcriptions of the source texts, the <note> elements may be encoded inline, because that is a onvenient thing to do.

Some linguists suggest that it is best to keep notes out of the flow of the text by moving them to a separate "notes only" <div> element.

MoveTEINotes reorganizes unadorned or adorned TEI XML files so that notes are moved to a <div type="notes"> in the <back> section at the end of the main <div> in which they occur. Original instances of the notes are replaced by a <ptr> element which points to the location of the relocated <note> element. An example of such a <ptr> element is:

<ptr type="note" target="nd1e8415" xml:id="rd1e8415" n="1"/>

The target= attribute gives the xml:id of the transplanted <note>. The xml:id provides the back link needed to restore the original note position give the transplanted note.

The XSLT transformation used to move the notes is defined in the **movenotes.xsl** file in the **xslt** directory of the MorphAdorner release. This transformation is based upon one originally written by Syd Bauman.

Processing Soft Hyphens

The Text Creation Partnership (TCP) transcriptions do not record line breaks in the printed originals. They do, however record "soft" hyphens where a word straddles two lines. The pipe character or vertical bar is used to mark such line breaks as in "wind|ing".

Word breaks at line endings are not always marked with a hyphen in the printed originals. Transcribers were asked to supply missing soft hyphens with a '+' sign. Sometimes they did, sometimes they didn't. Unmarked word breaks, especially in marginal notes, are a very common feature of the TCP texts.

The soft hyphens of the SGML transcriptions of the printed texts are treated according to the following protocol after conversion to TEI XML format.

1. If a spelling with a soft hyphen occurs elsewhere in the work or corpus as an unhyphenated spelling, the soft hyphen is removed.
2. If a spelling with a soft hyphen occurs elsewhere with a hyphen, the soft hyphen is replaced with a true hyphen.
3. If a spelling with a soft hyphen does not occur elsewhere either in a hyphenated or unhyphenated form and both word parts can serve as independent words the soft hyphen is replaced with a true hyphen.
4. If a spelling with a soft hyphen does not occur elsewhere either in a hyphenated or unhyphenated form and the word parts are not independent words the soft hyphen is removed.

This replacement algorithm is implemented by a sequence of utilities after all the XML files are tokenized. This is necessary to get the complete list of tokens for determining how often a word appears with or without a real hyphen in the corpus. These utilities are applied only for TCP texts and are not particularly useful in general.

1. Count words with word breaks using `edu.northwestern.at.morphadorner.tools.tcp.CountDividedWords`.
2. Figure out which words should have word breaks using `edu.northwestern.at.morphadorner.tools.tcp.FindSoftHyphens` then `edu.northwestern.at.morphadorner.tools.tcp.ExtractSoftHyphens`.
3. Substitute real hyphens for soft hyphens in words which should be hyphenated. Other soft hyphens are removed: `edu.northwestern.at.morphadorner.tools.tcp.FixWordBreaks`.

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.

Removing cruft from TEI XML file

RemoveCruft cleans Text Creation Partnership TEI XML files by replacing long "s" characters with regular "s", removing brace-enclosed entities and certain superscripts, splitting ligatures into separate characters, and so on.

Usage:

```
removecruft outputdirectory superscriptmap.tab input1.xml
input2.xml ...
```

where

- *outputdirectory* is the output directory containing the resultant XML files.
- *superscriptmap.tab* is a two-column tab-separated file. The first column contains tokens containing tagged superscript characters. The second column contains replacement tokens with the superscript characters replaced by unicode superscript characters. This file can be empty if replacements are not wanted.
- *input*.xml* are the input TEI XML files.

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 77).
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 50).

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 58) 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 63) 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 56) 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 122 for more information on the spelling map file formats.

Unadorning adorned TEI files

Unadorn removes word level adornments from adorned files. Unadorn replaces <w>, <pc>, and <c> elements with their text contents.

Usage:

```
    unadorn outputdirectory adorned1.xml adorned2.xml ...
```

where

- **outputdirectory** specifies the output directory for the unadorned XML files.
- **adorned1.xml adorned2.xml ...** specifies the input MorphAdorned XML files from which to produce unadoned versions.

Updating an Adorned File

UpdateAdornedFile applies a change log to update or downdate an adorned file. The change log is specified in the format produced by the CompareAdornedFiles (page 40) utility.

Usage:

```
updateadornedfile   operation   oldadorned.xml   changelog.xml
newadorned.xml
```

where

- *operation* -- Update operation, either **update** to apply updates in *changelog.xml* to *srcadorned.xml* to produce *destadorned.xml*, or **revert** to undo updates in *changelog.xml* to *srcadorned.xml* to produce *destadorned.xml* .
- *srcadorned.xml* -- Source adorned file.
- *changelog.xml* -- Records token-based differences between two adorned files.
- *destadorned.xml* -- Destination adorned file.

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

Gap Filler

The text of many older works may not be clearly readable because of faded print, ink blotches, foxing, or other degradations of the printed source. Transcribers mark these unreadable sections in digital text copies using special characters or tag sequences. In TEI, the *<gap>* tag serves to mark sections of a text which cannot be transcribed because of problems in reading the original source.

It may be useful to try to repair individual damaged words by examining which letters appear in the same positions as unreadable letters across a set of related texts. In essense this is the same as trying to find the missing letters in words in crossword puzzles. In some cases there is only a single plausible reconstruction for a damaged word. More often there are several possible reconstructions.

MorphAdorner implements a "gap filler" algorithm which looks at all the words which do not contain gaps in a given lexicon and tries to find potential matches for a word containing individual letter gaps. MorphAdorner uses a trie structure to hold all the words without gaps, which supports fast searches for words contain unknown letters.

You can try MorphAdorner's gap filler online.

Hyphenator

MorphAdorner incorporates methods for hyphenating English words written by David Tolpin. The algorithms are based upon those originally written for the TeX typesetting system created by Donald E. Knuth. The rules for hyphenation differ somewhat between American English and British English. American English breaks words on sound, while British English breaks words on the origin of the word, then sound. There are also many exceptions.

MorphAdorner uses the British approach as a default since MorphAdorner has been used mostly to analyze British literature up to now. However, tables are provided which define the American rules as well.

You can try MorphAdorner's hyphenator online.

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 Nakatani Shuyo.

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 Dutch 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 recognizes over 70 languages. The longer the text, the more reliable the detection.

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 121) 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 (2007-2009), 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.

TEI-Analytics has been revised over the past few years and is now, except for the word-level adornments, a proper subset of TEI P5.

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 **<div>** 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 **<em>**. Others indicate textual milestones such as **<milestone>** or formatting such as **<lb>**. Still others mark higher level text segments such as **<rs>**.

The <w>, <pc> and <c> tags

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

> MorphAdorner v1 used **<w>** for both words and punctuation as the <pc> element was not officially adopted at the time MorphAdorner was originally developed.

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. Emitted by default in MorphAdorner v1; optional and not emitted by default in MorphAdorner v2.
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. Used in MorphAdorner v1 by default. 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. The *eos* was used by default in MorphAdorner v1, and remains an option in MorphAdorner v2. MorphAdorner v2 marks the end of a sentence by adding an *unit="sentence"* attribute to the last token in a sentence, specified either by a <w> or <pc>. In some cases an empty *<pc unit="sentence"/>* is used to mark the end of a sentence. Use of the *unit="sentence"* attribute value is more in line with TEI P5.
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.
n	Provides a location ID based upon a page image identifier and column within page. Optional;

	mostly used when adorning Text Creation Partnership after conversion from SGML to TEI XML format to maintain the tie between the original digitized page images and the text transcription.
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.

Location IDs

In addition to its xml:id MorphAdorner can generate a location ID as the 'n' attribute of <w> and <pc> elements. The purpose of this location ID is to facilitate alignment of the transcribed text with the page image, a key requirement for many forms of work with retro-digitized documents. The location ID is based on the page number of the digital scan, typically a double page. For examplem, it is referenced in the Text Creation Partnership SGML source texts as the value of the REF attribute in <PB> elements and appears as the value of the 'facs' attribute in the P5 version. Page numbers of the printed source appear in the PB elements as the value of N attributes, but not all printed pages have running page numbers. The location ID uses 'a' and 'b' to distinguish the parts of a double-paged scan.

More precisely, the location ID takes the form **facs-column-wordinpage** where *facs* comes from the attributes of the enclosing <pb> element, column is a letter starting with "a" and giving the column number on the printer page, and *wordinpage* is the ordinal of the word within the page starting at 1 multiplied by the spacing. Subsequent location ID values have a *wordinpage* value incremented by the given spacing value, which is 10 by default. Optionally the work ID (usually the base file name) can be prepended to the location ID.

Here is a typical example of a location ID.

- 2-a-0050

This refers to the first column, fifth word in page image 2 for the current work.

These can be long identifiers, but theoretically only the page-base counter needs to be recorded as an 'n' attribute. If page-based IDs are needed, they can be constructed on the fly or in a preprocessing step by concatenating the work ID, the attribute values of the <pb> element and the page counter. It may also be practical to construct an xml:id for each page by concatenating the workid with attribute values, as in <pb xml:id="A05137-025-051" facs="25" n="51" />

Marking the end of a sentence

MorphAdorner v1 used the *eos*= attribute on the <w> tag to mark a token which ends a sentence. We considered using <milestone> tags to mark sentence, but these presented many problems when sentences span jump tags. The same was true of seg-like markers such as <s>.

MorphAdorner v2 uses the *unit*= attribute with a value of "sentence" to mark the end of a sentence. This aligns with standard TEI P5 usage.

Using a word-level value -- either the *eos* attribute or the *unit*= attribute -- to mark the end of a sentence makes it easy to generate sentence information regardless of how one orders the text when dealing with jump tags. For example, Prior (part of Monk) 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:

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

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">Edward</w>
  <c> </c>
  <w eos="1" lem="saint" pos="n1" reg="St." spe="St." tok="St."
     xml:id="ancf0207-050750.1" part="I">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">t</w>
  </hi>
  <w eos="1" lem="saint" pos="n1" reg="St." spe="St." tok="St."
     xml:id="ancf0207-050750.3" part="F">.</w>
</p>
```

When an *ord* attribute appears, its 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.

Simplified TEI P5-like output

MorphAdorner v2 provides the AdornedToSimpleTEIP5 (page 45) utility which converts the non-standard word attribute values of adorned files to a simpler and more nearly standard TEI P5 format.

The simplified format emits the lemmata, the parts of speech, and the standard spelling for each token. The attribute names have changed to be compatible with TEI P5: *lem* is changed to *lemma*, *pos* is mapped to *ana* and a "#" prepended to the part of speech. The non-standard *reg* attribute can be retained or changed to a standard TEI P5 choice structure. The corrected spelling (*spe*), original token (*tok*), and word ordinal (*ord*), if any, are removed.

Here is a sample snippet showing the new adorned file format.

```
<w lemma="in" ana="#p-acp" reg="in" xml:id="A88624-000740">in</w>
<c> </c>
<w lemma="love" ana="#n1" reg="love" xml:id="A88624-000750">love</w>
<c> </c>
<w lemma="with" ana="#p-acp" reg="with" xml:id="A88624-000760">with</w>
<c> </c>
<w lemma="Ismenia" ana="#np1" reg="Ismenia" xml:id="A88624-000770">Ismenia</w>
<pc unit="sentence" xml:id="A88624-000780">.</pc>
```

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.

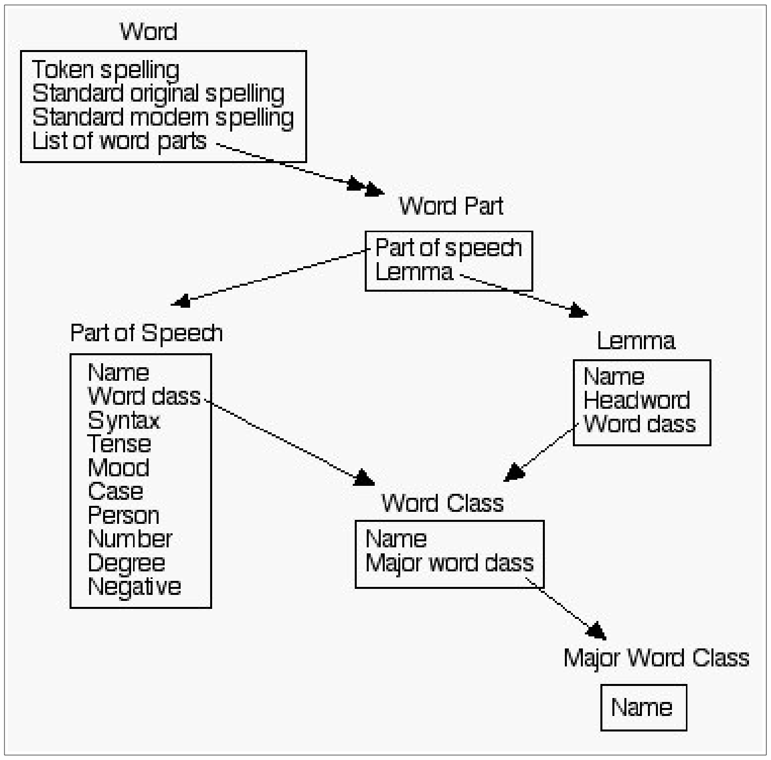
Contracted forms get two part of speech tags separated by a vertical bar, but with regard to forms like "don't', "cannot", "ain't", MorphAdorner analyzes the forms as the negative form of a verb and does not treat the form as a contraction. It uses the symbol 'x' to mark a negative part of speech tag.

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	he, she	21,000

	pronoun		
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 thow menest;	1
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 114.
- 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 94) 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‚Â£, Ã‚Â£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:

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

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. <strong>Word Tokenization</strong> (see page 132) 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. While the sentence splitter works best for English, some support is included for other languages, including those with non-Roman alphabets. Note that some languages, such as modern Japanese, provide unambiguous sentence markers. MorphAdorner uses these when present.

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)
```

Syllable Counter

MorphAdorner includes a facility for counting the number of syllables in an English word. This is useful as part of a scansion analysis.

THe printed versions of many early modern English plays were not written down by the original authors, and existed only in ephemeral scripts for use by actors. While some authors such as Ben Jonson took care to produce "official" versions of their plays for publication, many others did not, and the original scripts were lost. The printed versions were instead reconstructed by asking actors who had performed the plays to recite their parts from memory -- often years after their last performance. Surprisingly these actors were often very good at recalling the lines of their own parts. They were less accurate at recalling the lines spoken by others.

Since many of the plays were originally written in verse, usually in iambic pentameter, the quality of the reconstructions can often be assessed by simply counting the number of syllables in words in each line. Sections of the plays where the number of syllables do not match the required metrical format typically indicate misremembered lines.

You can try MorphAdorner's syllable counter online.

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.

Text Summarizer

A text summarizer attempts to produce a condensed version of text while retaining the most important parts of the original text. Some summarizers extract the entirety of important sentences. Others actually rewrite the sentences into a briefer form.

MorphAdorner contains a very simple-minded text summarizer. The summarizer accepts an English language text and produces a summary by finding the (up to) 100 most commonly used words in a text (not including stop words) and outputting the first sentence containing each common word. This works adequately for short expository articles, but rather badly for literature.

You can try MorphAdorner's text summarizer online.

Thesaurus

A thesaurus allows you to find synonyms and antonyms of a specified word. MorphAdorner includes a simple thesaurus facility based upon the venerable WordNet project. WordNet groups English words into sets of synonyms called *synsets*. WordNet also offers brief definitions for words and provides semantic relations among the synonym sets. While WordNet provides many other facilities in addition to finding synonyms and antonyms, MorphAdorner currently exposes just a simple synonym and antonym search.

WordNet only contains nouns, verbs, adjectives and adverbs. It does not include prepositions, determiners, or other parts of speech.

You can try MorphAdorner's online thesaurus based upon WordNet synsets.

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 118) 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<sup>o</sup>. 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<sup>o</sup> 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.

Processing Text Creation Partnership Files

Introduction

This page describes the sequence of steps that begin with an SGML encoded Text Creation Partnership file and transform it into a linguistically adorned file processed with Abbot and MorphAdorner.

The MorphAdorner v2.0 project seeks to capture the orthographic and morphological variety of Early Modern printed books and make their texts available in formats that both articulate and erase difference. Once Abbot has transformed the original SGML transcriptions into TEI XML, MorphAdorner tokenizes each word occurrence in a text and maps its surface form to the combination of a lemma and part of speech. A surface form like 'louyth' is mapped to the combination of the lemma 'love' and the POS tag 'vvz'. A 'lempos' or combination of lemma and POS tag can be used as the basis for a standardized spelling. On this view, linguistic adornment provides a virtual erasure of difference, which is useful for some purposes. Alternately, a lempos can also be used to look for the different surface forms in which that particular lexical and morphological phenomenon is realized. On that view, useful for other purposes, linguistic adornment provides a procedure for discovering and analyzing difference.

It is a major goal of the Abbot and EEBO MorphAdorner collaboration to turn the TCP texts into the foundation for a "Book of English," defined as:

- a large, growing, collaboratively curated, and public domain corpus of written English since its earliest modern form
- with full bibliographical detail
- and light but consistent structural and linguistic annotation.

Texts in the adorned TCP corpus will exist in more than one format so as to facilitate different uses to which they are likely to be put. In a first step, Abbot transforms the SGML source text into a TEI P5 XML format. Abbot, a software program designed by Brian Pytlik Zillig and Stephen Ramsay, can read arbitrary XML files and convert them into other XML formats or a shared format. Abbot generates its own set of conversion routines at runtime by reading an XML schema file and programmatically effecting the desired transformations.

MorphAdorner can output its results in a variety of tabular or XML based formats. Our goal is to provide output formats that can be successfully managed by scholars with moderate programming skills. We also believe that scholars working with the files will discover many instances of incompletely or incorrectly transcribed words and phrases. We want to make it easy to transmit completions or corrections back to the source files. Thus various "bread crumbs" are built into the design of MorphAdorner's routines and output formats. Linguistic adornment, coupled with appropriate analytical tools, opens up many new forms of analysis. But you should not underestimate the cumulative power of the quite humble task of discovering and fixing errors along the way.

The SGML source files

Origin and nature of the source files

The source files come from three Text Creation Partnership archives:

1. Early English Books Online files from Proquest, representing English books printed before

1700 (~45,000 files)
2. Eighteenth-Century Collection Online files from Cengage, representing books printed in the 18th century (~2,500 files)
3. The Evans collection of Early American imprints from Reddex, representing books printed in America before 1800 (~ 5,000 files)

Bibliographical data for all these files are contained in the English Short Title Catalog.

The TCP files were transcribed by various commercial vendors through a double keyboarding method. The transcriptions are based on digital scans of microfilms created in the mid- and late twentieth century. The quality of the microfilms and digital scans is variable. So is the quality of the printed original. Problems of transcription are overwhelmingly a function of what the transcriber was able to see on the digital copy of a microfilm image of a printed page.

The texts were encoded in SGML using a DTD that is a modification of the P3 TEI Guidelines. The files were encoded in ASCII and employ about 1,500 character entities to represent characters and symbols not found in the lower ASCII set of characters.

Typographical changes

The printed sources of the TCP texts use a great variety of typefaces and mix them in various ways. The TCP transcriptions ignore most of this, but use the <hi> tag to mark a change of type. An <hi> element means that the text enclosed by it is set in a different type from the text that surrounds it. This use of the <hi> tag does not provide any information about the type of the surrounding or enclosed text. In practice, text enclosed by <hi> usually means text in italics surrounded by plain text, but often this is not the case. You cannot reconstruct the "look and feel" of the printed page from the transcription alone.

Idiosyncratic features of the source files

The SGML transcriptions use some project-specific tricks to capture various features of the source files.

Line breaks

The TCP transcriptions do not record line breaks in the printed originals. They do, however record "soft" hyphens where a word straddles two lines. The pipe character or vertical bar is used to mark such line breaks as in "wind|ing".

Word breaks at line endings are not always marked with a hyphen in the printed originals. Transcribers were asked to supply missing soft hyphens with a '+' sign. Sometimes they did, sometimes they didn't. Unmarked word breaks, especially in marginal notes, are a very common feature of the TCP texts.

Superscripts and subscripts

Superscripted and subscripted alphanumerical characters are marked in the SGML transcription with a single or double 'caret', e.g. "S^t^.", "2^^3".

Decorated initial characters

Initial decorated characters in the printed texts are marked in the SGML transcriptions with a preceding underscore, as in "_T".

The interim P5 version of each file

In a first (and reversible) step we use Abbot to transform the P3 SGML version into an XML version that parses under a slightly modified version of TEI P5. The goal here is not to create the perfect P5 version but to express the structure of the SGML files in P5-like XML with minimal changes. However, Abbot is able to generate TEI XML P5 compatible versions of about 99% of the TCP SGML files.

Abbot closes unclosed tags as required by XML, maps the TEI tags to their XML "camel case" versions, changes some tag attributes to their XML format, and replaces the temporary header with the actual TEI header. The header is also converted to XML format. Abbot performs a few other changes as noted below.

Conversion of character entities

Character entities with established utf-8 code points are converted to those code points. This includes the long 's', by far the most common character entity.

Character entities with no corresponding utf-8 code points are preserved using the ad hoc devices of the TCP XML version. Thus the character entity "&abque;", which marks a printer's abbreviation for 'que' is represented by {que}, and curly braces are used in similear cases, wrapping the content of character entities in curly braces.

Line-breaking hyphens

The pipe character used for line-breaking hyphens in the SGML texts is maintained in the XML. The transcriber-supplied hyphen marked with '+' is replaced with the Unicode soft hyphen \u2011.

Superscripts and subscripts

The SGML notation for superscripts and subscripts is maintained in the intermediate P5 version. A post-processing program replaces the SGML notation with XML tags.

Decorated initial characters

The SGML notation for decorated initial characters is maintained in the intermediate P5 version. A post-processing program run after initial tokenization adds a "rend=" attribute to a token containing decorated initial characters.

Gaps

The SGML notation for gaps is modified in the intermediate P5 version. Letter, word, span-based gap extents are changed to a sequence of gap marker characters.

- The Unicode black circle ● (Unicode u25CF) replaces missing letters.
- The sequence of Unicode left-angle bracket, lozenge, right-angle bracket □ ◊ □ (\u3008\u25CA\u3009) replaces each missing word.
- The Unicode sequence left-angle bracket, horizontal ellipsis, right-angle bracket □ ? □ (\u3008\u2026\u3009) replaces a span of missing text.
- Simple foreign gaps are replaced by <seg xml:lang="unknown"> □◊ □ □◊ □</seg> .
- Foreign gap lines (enclosed by <l> tags) are replaced by a sequence of seven □ ◊ □ missing word markers enclosed in an <l xml:lang="unknown"> tag.

Post-processing the Abbot TEI files

The Abbotized TEI files are modified slightly before they are tokenized. The changes consists primarily of converting the TCP style superscripts to XML tag format.

Converting ^d to elements.

Tokens which end in ^d where "d" is a single digit are converted to the token followed by a d. This allows inserting the missing targets of these apparent note references at a later manual editing stage. Some of these may actually be incorrectly transcribed British monetary markers where the digit "1" was encoded instead of the letter "l".

Superscripts and subscripts

The ~44,000 EEBO texts include ~7,500 distinct superscript patterns with ~625,000 occurrences. All but 113 patterns (with ~700 occurrences in ~90 files) can be satisfactorily presented with current utf-8 characters for superscripts. Subscripts are much less common and mostly numerical. Some subscripts may be wrongly transcribed superscripts, e.g. "S^^r".

Although most superscripts and subscripts can be expressed literally through utf-8, it may be that for most analytical purposes superscripts add a level of complexity without corresponding benefit. There is much to be said for replacing them with plain characters wherever this can be done without creating ambiguity. Getting rid of super and subscripts in those cases removes 98% or more of all instances.

y^e, y^t, and y^u

The spellings y^e, y^t, and y^u are best seen as single brevigraphs representing a whole word. The nature of 'y' in these case is determined by the following letter and represents the thorn or 'th' rather than 'y'. Replacing these brevigraphs with 'the', 'that', and 'thou' probably makes more sense for a linguistically adorned text than keeping the original spellings, which would require special filtering if a researcher wanted to some analysis of the distribution of 'y' and 'i' spellings in 16th century texts.

Texts that have 'y^e', 'y^t', and 'y^u' are also likely to have the brevigraphs 'w^c' 'w^t' for 'with' and 'which'. These are different from the 'y' cases in the sense that the first letter stands for itself. They do not resolve comfortably to the plain spellings 'wc' or 'wt', and it seems preferable to replace them with the words they stand for.

Common superscripts

The most common superscripts in later texts are strings like M^r, M^{rs}, D^r, 2^d, and the like, which are unambigous and intelligible in their plain spellings Mr, Mrs, Dr, 2d.

Problematic superscripts

Some superscripts produce ambiguous or illegible words when written in plain type: 'Ma^{tie}' and other abbreviations for 'Majesty' are the most common examples. In these case one can fall back on using superscripts.

This fallback position cannot be used for cases where there are no appropriate utf-8 code points. There is no lower case superscript 'q', and only a limited number of upper case characters . The problem is rare: there are 150 types with 700 occurrences across 90 files. In these cases one could wrap the superscripted characters in a <hi rend = "sup">.

For the sake of simplicity, it may be preferable to extend this practice to all cases where super- or subscripts cannot be unambiguously represented as plain letters. An additional argument in favour of going this is the problematical nature of displaying utf-8 superscripted characters. They come from different Unicode ranges and do not form a coherent character family. In some type faces these differences are leveled out. In others they are not. So superscript characters are a little like long 's': not fully at home in the world of utf-8.

Converting superscripts to tag form

Because of the all the difficulties noted above, we decided to convert all superscript sequences given in the ^c^d^e form to <hi rend="superscript">cde</hi> . The intermediate XML file contains some private XML tag sequences to ensure proper spacing is maintained when tokenizing the superscript sequences, and to allow proper recognition of printer's brevigraphs.

The tokenized version

Tokenization consists of mapping the boundaries of "words" and "sentences." From a theoretical perspective, both "words" and "sentences" are highly problematic constructs. In practical terms, the consistent application of heuristics will produce results you can work with in a dependable fashion. But it is not an unambiguous matter of "carving nature at its joints" (Plato, *Phaedrus* 265e), and there are plenty of edge cases.

About tokenization

A tokenized text is a nested structure in which the text consists of an ordered sequence of sentences, and each sentence consists of an ordered sequence of words. In the XML representation of the text, each word is contained by a <w> element, and punctuation is contained by a <pc> element. These <w> and <pc> elements are the "leaf nodes" or lowest point of a hierarchical or "tree" structure that ascends on a "path" through a series of nestings. A word in a play may sit at the bottom of the path "TEI/text/div/div/sp/l/w." In the MorphAdorned text, sentences are not enclosed in <s> tags that are stages in the Xpath, because sentences often cross the discursive boundaries established by elements, especially in verse. Sentence boundaries are marked by a <pc unit="sentence"> attribute, either attached to sentence-terminating punctuation, or an empty punctuation mark. Sentences can be identified and retrieved as the sequence of words between two words or punctuation marks with <pc unit ="sentence"> elements, except where text contained by certain "jump" tags such as <note> intrudes. Sentences can still be extracted by either physically or virtually (programmatically) relocating the text contained by the "jump" tags so that it no longer intrudes. MorphAdorner includes programs which do this in extracting plain (untagged) versions of the adorned texts for use by other non-XML-aware programs.

The xml:id and its complementary location id

MorphAdorner separates the act of tokenization from the act of linguistic adornment. A tokenized text (or part of it) can be re-adorned without affecting the original tokenization. Each <w> and <pc> element has an xml:id that is composed of a work ID and a running word counter that increments by 10 so that minor corrections can be accommodated without disrupting the sequence of ID's. For instance, falsely joined words are a very common occurrence in the TCP text. The correction of such a phenomenon (e.g. 'beginwith') involves the division of one token into two. If the original token count goes like "10, 20", splitting "beginwith" into two words with the id's 10 and 15 does not affect other

IDs or their sequence.

Since the xml:ids are unique across the entire corpus they can be used to reference words in a document from other XML files, databases, or custom document types.

In addition to its xml:id MorphAdorner can generate a location ID as the 'n' attribute of <w> and <pc> elements. The purpose of this location ID is to facilitate alignment of the transcribed text with the page image, a key requirement for many forms of work with retro-digitized documents. The location ID is based on the page number of the digital scan, typically a double page. It is referenced in the SGML source text as the value of the REF attribute in <PB> elements and appears as the value of the 'facs' attribute in the P5 version. Page numbers of the printed source appear in the PB elements as the value of N attributes, but not all printed pages have running page numbers. The location ID uses 'a' and 'b' to distinguish the parts of a double-paged scan.

More precisely, the location ID takes the form **facs-column-wordinpage** where *facs* comes from the attributes of the enclosing <pb> element, column is a letter starting with "a" and giving the column number on the printer page, and *wordinpage* is the ordinal of the word within the page starting at 1 multiplied by the spacing. Subsequent location ID values have a *wordinpage* value incremented by the given spacing value, which is 10 by default. Optionally the work ID (usually the base file name) can be prepended to the location ID.

Here is a typical example of a location ID.

- 2-a-0050

This refers to the first column, fifth word in page image 2 for the current work.

These can be long identifiers, but theoretically only the page-base counter needs to be recorded as an 'n' attribute. If page-based IDs are needed, they can be constructed on the fly or in a preprocessing step by concatenating the work ID, the attribute values of the <pb> element and the page counter. It may also be practical to construct an xml:id for each page by concatenating the workid with attribute values, as in <pb xml:id="A05137-025-051" facs="25" n="51" />

Tokenization and the apostrophe

The TCP corpora use the apostrophe character (Ascii 39) to represent both the apostrophe character proper and the single quote. The apostrophe symbol presents tricky problems for tokenization when it appears before or after a word. It may be an opening or closing quotation mark or it may be part of a contracted form like "'tis" or the possessive marker of a plural noun (sailors'). In the former cases it should be replaced by opening or closing quotation marks and be identified as a separate token. In the latter cases it should be counted as part of the word. It is possible to identify contracted forms with considerable precision. Apostrophes sometimes appear as leading quotation marks at the beginning of lines of verse.

Apostrophes are rare before the late seventeenth, but their disambiguation is a non-trivial problem in texts from the late 17th and 18th century, especially play texts or texts that contain conversation or informal correspondence. The relative frequency of apostrophes is a pretty good guide to texts of this kind.

MorphAdorner uses a table of common occurrences of words beginning or ending with an apostrophe to determine when to split or retain initial or trailing apostrophes from words. This is not completely accurate but the most common occurrences of tokens with leading or trailing apostrophes are correctly handled.

Tokenization and the mdash

The mdash (\u2014) is another symbol that complicates tokenization. It is very rare in 16th and early 17th century texts. Like the apostrophe, it belongs to the world of conversation and informal correspondence. In the SGML texts the mdash character entity is used both as a punctuation mark and as the symbol for "polite elision", e.g. "d-mn" or "B—p," etc. In this second use the mdash does not mark a token boundary. You can with tolerable accuracy distinguish between these two uses through a combination of algorithms and exception lists.

The SGML texts never use the horizontal bar (\u2015), which is visually indistinguishable from the mdash. It is therefore possible to use the horizontal bar to mark polite elision. This removes existing ambiguity without creating new forms of ambiguity. The replacement of the mdash with a horizontal could be explicitly recorded in a change log, but this is not strictly necessary since in the Morphadorned TCP texts the presence of \u2015 would by definition involve its replacement of "&mdash;" in the SGML source files.

MorphAdorner attempts to distinguish the cases where the mdash is a word separator from those where it should not be (as in polite elision). This cannot be done with high accuracy and some tokenization errors remain.

Periods and abbreviations

MorphAdorner's sentence splitter uses the ICU4JBreakIterator class (from the International Components for Unicode) along with a large set of heuristics 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.

Abbreviations are a source of many tokenization errors. The TCP texts include a great many scientific, theological, and other learned texts with thousands of obscure and rarely consistent abbreviations.

MorphAdorner includes an implementation of the Punkt algorithm which treats abbreviations as a special form of collocation in which a character string habitually collocates with a final period. Running the Punkt abbreviation detection algorithm over an entire corpus provides an initial, somewhat conservative list of abbreviations. The abbreviations produced by Punkt have proved to be genuine abbreviations, or at least strings to which the trailing period should remain attached (e.g., Roman numerals). Punkt misses some abbreviations, so the initial list requires manual enhancement.

Many of the most commonly missed abbreviations are Biblical references. Punkt relies on relative occurrences of tokens with and without trailing periods in order to determine which strings are probable abbreviations. Especially in the earlier EEBO texts, an abbreviated Biblical book may appear both with and without a trailing period - e.g., Corinthians may appear as both Cor and Cor.

It is important when tokenizing some kinds of texts to use different abbreviation lists for different parts of the text. For example, we used different abbreviation lists when adorning the main part of drama texts as opposed to the paratext (stage directions, speaker labels, etc.). MorphAdorner provides for using different abbreviation lists based upon tag classes.

Roman numerals

Roman numerals in older English language texts exhibit considerably more orthographic variation than contemporary usage allows. For example, the letters "j" and "u" are often substituted for "i" and "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, as in ".XVI." Some Roman numerals are often followed by superscripted letters, as in "DCCXXV<sup>o</sup>," where the Latin inflection markers need to be stripped in order to retrieve the base form "DCCXXV". MorphAdorner attempts to recognize many of these variants.

It is sometimes difficult to distinguish algorithmically between different uses of strings such as "I." which may be a Roman numeral, an initial, or a personal pronoun. "D." may be a Roman numeral or an initial. Many of these problems occur around abbreviations in Biblical references. Disambiguating the usage is important to achieve accurate part of speech tagging.

Back-tick characters

The back-tick character ` (Ascii 96) appears in a number of texts in different contexts. When it occurs in the middle of a word, it acts as an alternative to the apostrophe. At the start of a line of verse the back-tick (or two back-ticks in sequence) functions as a kind of opening quote with no corresponding closing quote.

MorphAdorner treats two back-ticks as a single punctuation mark and splits them from the token to which they are attached. A single back-tick followed by a capital letter (ignoring any intermediate decorate tag such as <hi>) is treated as a separable token as well. This is correct more often than not. Other instances of the back-tick are left attached to the token in which they appear. The back-tick is regularized to an apostrophe when looking up spellings for purposes of lemmatization and part of speech tagging.

Edge cases of 'words' in MorphAdorned texts

The TEI Guidelines define the content of a <w> element as a "grammatical (not necessarily orthographic) word. While the blank space is the most common word boundary marker, a blank space does not always separate one word from another, and there are lexical items that may be spelled as a single word, two words, or hyphenated words. In the TCP texts there are three very common types of such lexical items: reflexive pronouns, British monetary terms, and the words 'today' , 'tomorrow', and 'yesterday'.

MorphAdorner handles these cases using pattern matching during the tokenization phase. Occurrences such as "my self" are treated as split words, and the individual parts are marked with the "part=" attribute of the <w> element to indicate this. Some special cases are handled. For example, in the phrase "from day to day" the "to" and "day" are individual words, not parts of a split word.

Reflexive pronouns

The reflexive pronouns 'myself', 'herself' etc. occur as hyphenated or single word spellings from very early on. The frequency of two-word spellings declines over time, but it is probably a mistake to use orthographic difference as a sufficient reason for analyzing "my self" as the sequence of a possessive pronoun and a noun, while 'myself' or 'my-self' are analyzed as reflexive pronouns. Mapping the different spellings to the same description and treating them as single lexemes still let an investigator pursue the question whether or how the decline of two-word spellings marks a change in the perception of these "words" as single or composite.

British monetary terms

The most common way of referring to pounds, shillings, and pence is to use the Latin abreviations 'l' or 'lb', 's', and 'd' preceded by a numeral, typically Arabic. There may or may not be a space between the numeral and the abbreviation. The abbreviation is often, but not always, marked by a period. The abbreviation may also appear as a superscript (more common with 'l' or 'lb' than with 's' and 'd').

If you look for monetary terms across many texts in the corpus, it is probably helpful to treat these different spellings as single monetary expressions and contain them in a single <w> element.

MorphAdorner attempts to locate occurrences of monetary patterns and encodes the variants which contain blanks as split words. This allows recovery of the joined word as a single unit. Unfortunately a fair number of occurrences of "l" (pound) following a number are encoded as the numeral "1" instead, complicating the recognition of the monetary pattern.

Today, tomorrow, and yesterday

The spellings of 'today', 'tomorrow', and 'yesterday' are all over the map in Early Modern English. As with reflexive pronouns, there is a trend away from two word spellings.

If 'to day' is treated as a single word, you need to watch out for the phrase 'from day to day,' where 'to day' is clearly not one word. MorphAdorner has a list of such exceptional cases.

Changes in the tokenized file

The tokenized file as the basis for linguistic adornment

The tokenized file serves as the basis for linguistic adornment. Some features of the SGML source file are very unlikely to be of further use in the adorned file and make working with it harder. They are removed at this stage. Because each word in the tokenized file has a unique xml:id, it is easy to log all the changes and "park" them in a change log file. Think of it as a form of tacit stand-off markup. It is tacit in the sense that the tokenized file need not include an explicit pointer to a record in the change log. But you can ask whether a given xml:id in the tokenized file has a corresponding record in the change log.

The character of the change log

Just about anything archived in a change log could also be stored as elements or attributes in the XML file. It is not expensive to store everything in one file, but bloated files are cumbersome to manipulate. The cost of storing such files may be trivial, but the cost -- in terms of time or complexity -- of manipulating them is not. You may want to look for some kind of "off-site" storage for features that will be used rarely, if ever. It is critical that such features can be retrieved with precision and ease. It is not critical that they are retrievable "on the fly."

The changes in question always involve the content of <w> and <pc> elements, and possibly associated <c> elements which enclose word-separating whitespace.

MorphAdorner uses a simple XML format to contain a list of token-based changes. The format of this file is as follows.

<ChangeLog>

```
 <changeTime>The time the change file was created.</changeTime>
 <changeDescription>A description of the changes.</changeDescription>
 <changes>
  <change>
   <id>xml:id of token to be changed.</id>
   <changeType>addition, modification, or deletion.</changeType>
   <fieldType>Type of field to change: text or attribute.</fieldType>
   <oldValue>Old field value.</oldValue>
   <newValue>New field value.</newValue>
   <siblingID>xml:id of sibling word for a word being added.</siblingID>
   <blankPrecedes>true if blank precedes the token, else false.</blankPrecedes>
  </change>
         ...
(more <change> entries)
         ...
 </changes>
</ChangeLog>
```

This simple XML formatted change file allows a file to be transformed to a corrected file using a utility in the MorphAdorner suite. A file can be "untransformed" from the corrected version to the uncorrected version using the same change file. A likely use case for the change log is an edition that wants to use long 's' and other original spellings.

Here is an example of a change log entry which records the replacement of a long "s" with a plain "s".

```
<ChangeLog>
 <changeTime>2013-07-09 13:04:17.149 CDT</changeTime>
 <changeDescription>Changes from \tokenized\K000379.000.xml to \tokenized-no-word-
breaks\K000379.000.xml as determined by CompareAdornedFiles.</changeDescription>
 <changes>
   <change>
    <id>K000379_000-00080</id>
    <changeType>modification</changeType>
    <fieldType>text</fieldType>
    <oldValue>Addreſs'd</oldValue>
    <newValue>Address'd</newValue>
    <blankPrecedes>true</blankPrecedes>
   </change>
      ...
 </changes>
</ChangeLog>
```

Long 's'

The Unicode character for long 's' is replaced at this stage with plain 's'. For almost any conceivable inquiry, the presence of two different forms of 's' complicates analysis without compensating advantage. The word occurrences with long 's' are logged in the change log.

Soft hyphens

The soft hyphens of the SGML files are treated according to the following protocol:

1. If a spelling with a soft hyphen occurs elsewhere in the work or corpus as an unhyphenated spelling, the soft hyphen is removed.
2. If a spelling with a soft hyphen occurs elsewhere with a hyphen, the soft hyphen is replaced with a true hyphen.
3. If a spelling with a soft hyphen does not occur elsewhere either in a hyphenated or unhyphenated form and both word parts can serve as independent words the soft hyphen is replaced with a true hyphen.
4. If a spelling with a soft hyphen does not occur elsewhere either in a hyphenated or unhyphenated form and the word parts are not independent words the soft hyphen is removed.

This replacement algorithm is implemented in a post-processing step after all the XML files are tokenized. This is necessary to get the complete list of tokens for determining how often a word appears with or without a real hyphen in the corpus.

Character entities without corresponding utf-8 code points

In Michigan's display-oriented XML versions of the SGML texts, character entities without corresponding utf-8 points are represented through various workarounds, often enclosing these in curly braces, as in "cum{que}" which represents the SGML transcription "cum&abque," which represents "cum" plus a brevigraph for "que". Where the braces can be dropped without creating ambiguity or illegibility -- which is true of most cases -- they should be dropped with an appropriate record in the change log.

The horizontal bar as the marker of polite elision

As noted above, the SGML texts use the &mdash; character entity both as a punctuation mark and as a symbol for polite elision. Polite elision in the MorphAdorned files should be marked by the horizontal bar. We did not make this change in our initial conversion, but will consider doing it in the future.

Decorator characters

The underscore character identifying the initial character in a section or paragraph as decorated is removed, but logged. A rend attribute with the value *initialchardecorated* is added to the word element.

hi tags inside words

In the SGML texts <HI> tags sometimes begin or end in the middle of a word, reflecting common practices of Early Modern printing. In the possessive form of a name, the root is often italicized while the case ending is in plain type: "<hi>Caesar</hi>'s death". If the surrounding text is in italics and the name is emphasized through small caps, the SGML text is likely to represent that as "Caesar<hi>'s death</hi>. If you tokenize the text, the <w> element straddles different tags, requiring complex procedure of splitting and joining word parts.

One way of avoiding this problem in the first place is to move the information from the <HI> tag of the SGML text "atomically" into the "rend" attribute of the <w> elements. That way all information about formating is kept at the same and lowest level. If you want to ignore it you can, and it will never be in the way because there is never any data below it whose treatment depends on what you do or do not do with the formatting data.

Because most <hi> elements consist of a single word or two words, recording information about the <hi> status of a <w> element "atomically" as the value of a rend attribute does not make the file significantly more verbose. But it gives you an opportunity to use "hybrid" rend attributes to describe words, parts of which are wrapped in <hi>tags.

Consider the most common case: <hi>Caesar</hi>'s death. This is tokenized as

<w xml:id ="someid1" rend="plain_apostrophe">Caesar's</w>

<w xnk:id = "someid2">death</w>

The attribute value is part of a controlled vocabulary of about two dozen cases, and it means that a highlighted name is followed by a possessive in plain type. This preserves the formatting information as it appears in the SGML texts. The change can be performed in a post-processing step either before adorning the tokenized files, or after adornment is complete.

While we have not yet worked out all the details of hybrid hi tags, there are three basic cases with various subdivisions:

1. Two parts of a word are in hi tags, but a middle connecting string is not (rare)
2. The first part of a word is inside a <hi> tag
3. The second part of a word is inside a <hi> tag

In addition, there are a small number of cases of nested hi tags that must be handled. In a future release of MorphAdorner we expect to include a utility which replaces most hi tags in adorned or tokenized texts with rend= attributes in the word elements.

Post-processing the tokenized file

The tokenized file is post-processed to mark words containing gaps and to replace soft huphens with real hyphens or to remove them, as described above.

Adding type="unclear" to words containing gap characters

A *type="unclear"* attribute is added to any <w> element for a word or word part containing one or more gap characters ● (Unicode u25CF).

Other token-based changes

A post-tokenization program replaces the long "s" with plain "s" and removes braces surrounding brace-enclosed entities. A change can be created at this point to allow undoing these changes.

The process of linguistic adornment

Following the tokenization post-processing phase, each TEI XML file is lingistically adorned.

The pivotal position of the tokenized but not yet adorned file

The tokenized, but not yet adorned, interim P5 version of an SGML text has a pivotal position in the workflow that leads from SGML texts to linguistically adorned files. This version does not shed any data from the SGML text, but it identifies textual features that will be removed or changed, with all changes logged in a manner that allows backtracking to the SGML file.

This interim file is linked to the linguistically adorned file with its enrichments and simplifications through the stable system of xml:ids for each <w> element. There will be some changes in some xml:ids as errors are discovered and fixed, and the maintenance of the link between the first tokenized version and its adorned derivatives cannot be taken for granted. It needs attention, but it is a realistic assumption that it can be maintained.

The separation of tokenization from adornment is a key feature of MorphAdorner 2.0. It allows for work flows that are more granular and iterative, supporting cumulative improvements over time. Many data errors or opportunities for data enrichment are discovered in the process of working with data. While MorphAdorner does not by itself create a collaborative curation environment, its data structure and basic work flows are useful building blocks for such an environment.

Linguistic adornment

MorphAdorner associates every word occurrence with a lemma and a part of speech tag. From this combination, which we call the 'lempos' you can derive a standard spelling: if 'louyth' is identifed as an instance of 'love_vvz' you can algorithmically derive 'loveth' or 'loves' as the desired standardized spellings.

Errata divs

A number of TCP texts include div elements containing errata. The content of errata divs is generally not amenable to linguistic adornment. We mark all the non-punctuation tokens in errata divs with the NUPos "zz" part of speech for "unrecognizable."

Output formats

Native output

MorphAdorner produces a variety of outputs for adorned and unadorned texts as well as textual derivatives.

MorphAdorner's basic or native output format stores all its adornments as attribute values of a <w> or <pc> element. The principal nearly P5 compatible format uses the standard ana= and lemma= attributes to store the parts of speech and lemmata, respectively, adds a non-standard reg= attribute to hold the standardized spelling. Using an attribute is preferable to a choice element because the attribute leaves the token sequence undisturbed, and the added attribute value can be stored in the standard MorphAdorner change log.

Tabular output

For the purpose of reviewing and correcting data, MorphAdorner's tabular output (page 77) is very helpful. This output contains the following (among other items) as columns in a table:

- The corpus-wide xml:id
- The spelling
- The lemma
- The POS tag
- The token before
- The token after
- Up to 80 characters before
- Up to 80 characters after

- The highest level differentiating element (front, body, back)
- The parent element of <w>

Here is an example, with the spelling put between the before and after contexts:

K133535.000-052790	base	j	so	a	the twelve thousand Hessians , sold in so	b●se	a manner by their avaricious master to the	body	p

The example shows one of the several million incompletely transcribed words. In this, as in many other cases, the correct reading can be supplied by a literate reader with complete confidence and without consulting the page image. It is relatively straightforward to populate a database with tabular output containing only incomplete readings and adding a data entry capability that lets users log in and provide corrections. See, for example, Annolex, which also supports easy consultation of the page images in the many cases where that is necessary.

The main point here is that the MorphAdorner data structure provides a very robust foundation for collaborative improvement of the EEBO texts over time and by many hands. Central to this task is the maintenance of stable ID's that are the bread crumbs through which user-generate corrections can be tracked back to their source texts.

TEI compliant output

The simplest out-of-the-box version of MorphAdorned and TEI P5 compliant texts follows a format very close to the British National Corpus: the word token is the content of a <w> element. The lemma and POS tag are respectively stored in 'lemma' and 'ana' attributes. In out-of-the-box P5 you cannot store a standardized spelling in a 'reg' attribute. On the other hand, you can use a combination of <choice> , <orig> , and <reg> elements to make each <w> element carry its part of a double stream of original and standardized spellings, as in this adorned encoding of "wylle anone" from an early 16th century text:

```
<w xml:id ="someid1" lemma="will" ana="#vmb">

<choice>
<orig>wylle</orig>
<reg>will</reg>
</choice>
</w>
<w xml:id ="someid2" lemma="anon" ana="#av">>
<choice>
<orig>anone</orig>
<reg>anon</reg>
</choice>
</w>
```

Alternately, you can customize P5 and restore a 'reg' attribute that would let you encode the same phenomena in a manner that programmers -- and in particular programmers with limited skills -- are likely to find more intuitive and economical:

```
<w xml:id ="someid1" lemma="will" reg= "will" ana="#vmb">wylle</w>
<w xml:id ="someid2" lemma="anon" reg ="anon" ana="#av">anone</w>
```

Either way the linguistic adornment of consistently encoded TEI texts provides users with rich opportunities for combining lexical and morphological features with broader discursive features in their analysis of texts. MorphAdorner can generate either style of output for regular spellings.

Other output formats

MorphAdorner can also generate other types of output from adorned files, including various types of plain text, summary tabular files, and input for the Corpus Workbench, Sketch Engine, and BlackLab search engine.

NUPos interpGrp

The TEI P5 guidelines suggest including an **interpGrp** section to define the part of speech tags referenced by ana= attributes in word elements. Here is part of the **interpGrp** for the NUPOS part of speech tag set used by MorphAdorner.

```
<interpGrp type="NUPOS">
  <interp xml:id="a-acp">acp word as adverb</interp>
  <interp xml:id="av">adverb</interp>
  <interp xml:id="av-an">noun-adverb as adverb</interp>
  <interp xml:id="av-c">comparative adverb</interp>
  <interp xml:id="av-d">determiner/adverb as adverb</interp>
  <interp xml:id="av-dc">comparative determiner/adverb as adverb</interp>
  <interp xml:id="av-ds">superlative determiner as adverb</interp>
  <interp xml:id="av-dx">negative determiner as adverb</interp>
  <interp xml:id="av-j">adjective as adverb</interp>
    ...
  <interp xml:id="zz">unknown or unparsable token</interp>
</interpGrp>
```

We pondered how best to include this **interpGrp** in the adorned output files produced by MorphAdorner. A prolix approach adds the interpGrp to every adorned file in full. A sparer approach uses the *xinclude* facility to reference the same external copy from each adorned file. The question remained, where to put the full definition or include statement?

We considered placing the definition in the TEI header, or someplace in the body of the text. We finally decided to wait until the forthcoming TEI **standoff** tag becomes officially available. The **standoff** tag acts as a container for storing various kinds of standoff markup. This separates the standoff items from the actual text of the document. Hence the currently generated adorned output files do not include the NUPOS **interpGrp**.

Placement of notes

In the printed source texts, <note> elements never interrupt the reading order, because all the notes are either placed in the margin or at the bottom of the page. In the SGML texts, the <note> elements were encoded inline, because that was the most convenient thing to do.

For the purpose of maintaining continuity with the SGML source, the interim tokenized text needs to preserve their current position. The final output, however, can employ a variety of stand-off options and keep notes in special divs, whether at the end of each div or in a <back> element of each <text> element.

Consultation with several linguists suggests that there is some consensus about keeping notes out of the flow of the text and that, as Bryan Jurish put it in an email, there is much to be said for "the underlying intuition that a 'stupid' extraction of the raw text from an XML document (i.e. the concatenation all text nodes in document order) ought to return a linguistically plausible serial representation of the data."

MorphAdorner internally moves the content of <note> elements and other jump tags out of the way during adornment. This allows for proper part of speech tagging within the main text with the intrusive jump tag text getting in way of a proper reading order for words.

MorphAdorner also provides utilities for extracting the plain text of words or sentences that is cognizant of proper reading order as well. This allows for extracting random sentences, or generating input to programs such as Mallet to perform topic extraction.

MorphAdorner also contains a program which can reorganize adorned files so that notes are moved to a <div type="notes"> in the <back> section at the end of the main <div> in which they occur. Original instances of the notes are replaced by a <ptr> element which points to the location of the relocated <note> element. An example of such a <ptr> element is:

```
<ptr type="note" target="nd1e8415" xml:id="rd1e8415" n="1"/>
```

The target= attribute gives the xml:id of the transplanted <note>. The xml:id provides the back link needed to restore the original note position give the transplanted note.

Searching the corpora

Given a richly adorned corpus, the ability to search both the text and the adornments comprise an important basis for any research using that corpus. MorphAdorner itself does not provide such a search facility. Instead the assumption is that the adorned texts, or a suitable transformation of them, will comprise the input to one or more corpus search engines.

As noted in previous sections, MorphAdorner can transform the base adorned files to the input format required by some corpus search engines such as the Corpus WorkBench and the Sketch engine. The Philogic v4.0 search engine can index and search MorphAdorned XML files directly. Adorned files can also be used as input to generic XQUERY search engines such as BaseX and eXist. In addition, during the course of this project, we discovered the availability of a new corpus search engine called BlackLab, under development by the Institute of Dutch Lexicology (INL). An experimental search site built using BlackLab hosts adorned versions of Shakespeare's dramas and the TCP ECCO texts at http://devadorner.northwestern.edu/corpussearch/ .

BlackLab is a corpus retrieval engine built on top of the popular open-source search library Apache Lucene. According to its authors, BlackLab "offers fast, complex searches with accurate hit highlighting on large, tagged and annotated, bodies of text." BlackLab extends Lucene with the ability to use a variant of the Corpus Workbench search syntax to search adorned corpora for attributes such as lemma, part of speech, main versus paratext, and most any other token-level attribute one can imagine.

We heavily modified and enhanced a basic TEI corpus indexer provided by the BlackLab development group. We used this to create searchable versions of the MorphAdorner generated adorned files for all of the project corpora. The speed of both the indexing process and the searches is impressive. The BlackLab searches have allowed us to locate and correct a variety of adornment problems that would otherwise have been more difficult to find.

MorphAdorner Server

The MorphAdorner Server wraps adornment processes as web services using Rest-like interfaces. The web services can be accessed from any programming language and system which knows how to send and receive HTTP requests, or even a plain web browser. The services are automatically parallelized because of the way HTTP servers work. Many clients can access the same web service simultaneously. MorphAdorner Server uses the Restlet library to implement the web services.

Some of the TEI-based services currently provided by the MorphAdorner Server include:

- Convert an adorned TEI XML to tabular format.
- Adorn a TEI XML file.
- Apply a change log to an adorned TEI XML file.
- Compare two versions of an adorned TEI XML file and generate an XML format change log.
- Extract text from a TEI XML file.
- Extract sentences from adorned TEI XML file.
- Move notes to a special div in a TEI XML file.
- Tokenize a TEI XML file.
- Unadorn a TEI XML file.

Future directions

We hope the initial work we've done on the TCP texts will continue in subsequent projects. Aside from improving the tokenization and morphological adornment, we expect to merge meta-data from the electronic version of the English Short Title Catalog into the metadata sections of the TEI XML files, perhaps into the <keywords> sections of the TEI header. This will improve the ability of researchers to search for and create corpus subsets of interest as well as allow comparison with other corpora.

We also hope to be able to identify named entities of various types, including personal names and places. This is more difficult in literary texts than in other discursive writing because many of the names are entirely fictional. Many TCP texts contain Biblical references and references to classical authors. It would be useful to mark these using the xml:id of the tokens comprising the entities, perhaps saving them in stand-off form in auxiliary documents.

A longer term goal is the compilation of a comprehensive lexicon of spellings and variants with date information from the TCP texts. The lexicon would include frequencies of occurrence across centuries, broken down by genre and part of speech, as well as lemmata by parts of speech. Such a lexicon would allow morphological adornment processes to use a standardized lexicon ID for each word in a text.

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, 2013 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 );
    }
}

/*
Copyright (c) 2008, 2013 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 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, 2013 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 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.*;
```

```java
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, 2013 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.
*/
```

MorphAdorner Server

MorphAdorner Server, or MAServer for short, is an HTTP-based server which exposes MorphAdorner facilities over the web. The server has its own download and installation page (page 193) separate from the MorphAdorner client.

Below is the complete list of the services offered by the MorphAdorner Server. The page for each service presents the service name, query parameters, available output formats, and -- for the plain text services -- sample output. The online examples demonstrate the use of the MorphAdorner Server using web page forms. Also see "Accessing the server programmatically" (page 200) to learn about writing Javascript-enhanced web pages and Java programs to access the MorphAdorner server.

Plain text services

The plain text services work on utf-8 encoded text that has not been tagged in any way.

- Adorner for plain text (page 216)
- Corpus configurations (page 229)
- Gap filler (page 232)
- Hyphenator (page 235)
- Language recognizer (page 238)
- Lemmatizer (page 242)
- Lexicon lookup (page 246)
- Name recognizer (page 255)
- Noun Pluralizer (page 259)
- Parser (page 262)
- Sentence splitter (page 265)
- Spelling standardizer (page 275)
- Syllable counter (page 279)
- Text segmenter (page 282)
- Text summarizer (page 302)
- Thesaurus (page 307)
- Tokenizer (page 311)
- Verb conjugator (page 322)
- Version (page 326)

TEI XML services

The TEI XML services work on utf-8 encoded text tagged using the Text Encoding Initiative markup.

- Adorned TEI to tabular format (page 328)
- Adorner for TEI XML (page 330)
- Apply change file to an adorned TEI XML file (page 332)
- Compare adorned TEI XML files and generate change file (page 334)
- Extract text from TEI XML (page 336)
- Extract sentences from TEI XML (page 338)
- Move TEI notes to div (page 340)
- Tokenize TEI XML (page 342)
- Unadorn TEI XML (page 344)

MorphAdorner Server Installation

MorphAdorner Server, or MAServer for short, is an HTTP-based server which exposes selected MorphAdorner facilities over the World Wide Web.

File name:	maserver-1.0.0.zip
Current version:	1.0.0.
Last update:	September 16, 2013.

The MorphAdorner Server source code and support files, along with an issue tracker, are available as a Mercurial repository on bitbucket.org at

```
http://bitbucket.org/pibburns/morphadornerserver
```

Quick Setup

If you downloaded the MAServer release from the Mercurial repository on bitbucket.org, please go to the section "Installing and building MAServer."

If you downloaded the ready-to-use maserver-1.0.0.zip file, proceed as follows. Expand the contents of the maserver-1.0.0.zip file into an empty directory. Make sure you retain the existing directory structure.

You must have the Java run-time environment installed on your machine to run MAServer. If you do not, go to the section "Installing and Building MAServer" for information on where to get a copy of the Java runtime. Once you have Java installed you can proceed with running MAServer.

To run MAServer standalone on Windows, type

```
runmaserver.bat
```

at the command line of a console window.

On Unix-like systems, including Mac OS X, type

```
chmod 755 runmaserver
```

in a terminal window to set the shell script to execute. You only need to do this once. To run the server, type

```
./runmaserver
```

By default MAServer listens on TCP port 8182. You can change this default port number in the batch file or shell script. Both the batch file and script request 4 gigabytes of memory to run.

MAServer requires at least 2.5 gigabytes of memory to run with 4 gigabytes preferred. For best results you should run MAServer on a 64-bit operating system with a 64-bit version of the Java run-time environment installed. Your system may require more memory than these minimums. In particular, Mac OS X may require at least 3.0 gigabytes of memory to run MAServer.

You can access the test web pages once MAServer finishes initialization, which can take a couple of minutes on a slow system. Open a web browser on the system on which you are running MAServer and enter the URL

http://localhost:8182/

You should see the main MAServer test page. If you changed the default TCP port for the server, replace 8182 in the URL with your modified port number.

File Layout of MorphAdorner Server (MAServer) Release

File or Directory	**Contents**
build.properties	Build settings you can modify.
build.xml	Apache Ant build file used to compile MAServer.
conf/	Configuration files.
log4j.properties	Logging properties.
template-web.xml	Template for generating web.xml file.
wadl.xsl	Web Application Descriptor Language HTML conversion.
web-xml.properties	Settings for generating web.xml file.
data/	Data files used by server.
doc/	Documentation for using the server.
ivy.xml	Apache Ivy dependencies definitions.
ivysettings.xml	Apache Ivy settings.
lib/	Java library files used by MAServer. These are retrieved on demand during the build process using Apache Ivy.
license.txt	The MAServer license.
modhist.txt	MAServer modification history.
README.txt	Printable copy of this file in Windows text format (lines terminated by Ascii CR/LF).
runmaserver	Unix shell command file to start server in standalone mode.
runmaserver.bat	Windows batch file to start server in standalone mode.
src/	MAServer source code.
testdata/	Test data files.
webpages/	Static web pages for testing MAServer facilities.

Installing and Building MAServer

To rebuild the MAServer code, make sure you have installed recent working copies of Oracle's Java Development Kit and Apache Ant on your system. The Java development kits for Windows, Mac OS X, and Linux systems may be obtained from

http://www.oracle.com/technetwork/java/javase/downloads/index.html

Alternatively, OpenJDK may be obtained from

http://openjdk.java.net/install/index.html

http://ant.apache.org

Move to the directory into which you unzipped the MAServer release (or into which you cloned a local copy of the Mercurial repository for MAServer).

Use a plain text editor to edit the "build.properties" file. You should provide values for the following three settings.

1. The "serverdata.dir" setting should be set to the MAServer data directory for a remote installation of MAServer. This can be a local directory on your desktop if you are running MAServer under a local copy of a servlet server. This value is used only by the Ant "copyserverdata" task. If you don't intend to use that task to copy the server data, you may leave "serverdata.dir" empty.
2. The "localServerURL" setting should be set to the base URL of your local MAServer installation. The default value of

   ```
   localServerURL=http://localhost:8182/
   ```

   is fine for out-of-the-box use when you run the server using runmaserver.bat/runmaserver .
3. The "remoteServerURL" setting should be set to the base URL of your remote installation of MAServer, if any. This is needed to run tests against that server. If you only intend to run the built-in server version of MAServer (using runmaserver.bat/runmaserver), you may leave this setting empty.

   For example, if your server name is "myremotehost.com", you would enter something like:

   ```
   remoteServerURL=http://myremotehost.com/maserver/
   ```

   If you intend to run MAServer under a local copy of a servlet server such as Tomcat or Jetty on your own desktop, you can set the remoteServerURL to point to your desktop. In this case the setting will be something like:

   ```
   remoteServerURL=http://localhost:8080/maserver/
   ```

   Once you have set the above three entries in build.properties appropriately, save the build.properties file with the updated values.

To run MAServer under a servlet server such as Tomcat, you must also modify the settings in the conf/web-xml.properties file. Open this file with a plain text editor, and provide values for the following settings.

1. The "datadirectory" settings specifies the location of the MAServer data files -as seen by the servlet server-. This may differ from the value you set for "serverdata.dir" in the build.properties file.
2. The "maxunadorneduploadfilesize" specifies the maximum file size in bytes of an unadorned TEI XML file which the server will accept as an upload. The default value is "5m" or 5

megabytes.

3. The "maxadorneduploadfilesize" specifies the maximum file size in bytes of an adorned TEI XML file which the server will accept as an upload. The default value is "50m" or 50 megabytes. The larger the file size limits provided, the more memory the server requires to process the files.

Save the conf/web-xml.proerties file with the updated values.

After you have set the values in the build.properties and conf/web-xml.properties files appropriately, open a console or terminal window, move to the base directory of the MAServer release, and type:

```
ant
```

to build MAServer. If the build completes successfully, the maserver.jar and maserver.war files will be placed in the "dist" subdirectory.

You must use a Java compiler which is compatible with Java 1.6 or higher. MAserver has been successfully compiled and executed under Windows and Linux using the standard Oracle JDK 1.6 and 1.7 releases; under Linux using a recent release of OpenJDK 7; and under Mac OS X using a recent version of the standard MAC OS X Java compiler and run-time. Other Java compilers and run-times may not work.

Type

```
ant javadoc
```

to generate the javadoc (internal documentation) into subdirectory "javadoc".

Type

```
ant clean
```

to remove the effects of compilation. This does not remove the downloaded files in the lib subdirectory. To remove those as well, type

```
ant cleanlib
```

Once in a while, if you are having trouble compiling, you may need to clean your Ivy cache to make sure you have the correct library files. Type

```
ant cleancache
```

to clean the Ivy cache.

Running MAServer In A Servlet Server

To deploy MAServer in a servlet server such as Tomcat you need to do four things:

1. Copy the data directory to a location of your choosing.

   Copy the entire data/ directory along with its subdirectories to a directory somewhere on your system. By default this directory is defined as /project/maserverdata . You should change this setting in the conf/web-xml.properties file by setting the value of the "datadirectory" property to the correct directory name on your server.

If the remote server data directory is mounted so that you can access it locally, you can type

```
ant copyserverdata
```

to copy the data files to the remote directory you specified as the value of the "serverdata.dir" setting in the build.properties file.

The data directory you select, and all its subdirectories, must be readable by your chosen servlet server. The servlet server must also have permission to change to that directory while running.

2. Rebuild the maserver.war file.

   Rebuild the maserver.war file by typing

   ```
   ant war
   ```

   in a console/terminal window. The updated maserver.war file is written to dist/maserver.war .

3. Install the rebuilt maserver.war file into your servlet server.

   Different servlet servers have various methods for doing this. Consult the documentation for your particular servlet server for details.

   For example, in Tomcat, you can copy the maserver.war file to the Tomcat "webapps" subdirectory. Make sure you have configured Tomcat to deploy WAR files automatically by setting the "autoDeploy" option to "true" in the Host container element. See

   `http://tomcat.apache.org/tomcat-7.0-doc/config/host.html`

   for details.

   MAServer has been tested to work under both Tomcat (v7) and Jetty (v8).

4. Restart your servlet server.

Some servlet servers can "hot install" new web applications presented as a war file, so you may not have to restart your server. It's usually a good idea to restart the server anyway. You must restart the server if you stopped the server before installing the MAServer war file.

After you restart your servlet server, MAServer should become available within a couple of minutes under the application name "maserver". Open a web browser on the system on which you are running the server and navigate to the web page URL

`http://servername:8080/maserver/`

Replace "servername" with the name of the system on which you installed MAServer, and replace "8080" with the TCP port number for accessing your servlet server. You should see the main MAServer services web page once MAServer initialization completes.

Testing

The MAServer release contains a small set of tests which may be used to test the server's operation. These are not intended to be comprehensive.

To run the tests, make sure you've provided values for the "remoteServerURL" and/or "localServerURL" settings in build.properties, as described above.

The run the tests against a local MAServer instance, start that instance, then open a console/terminal window and type:

```
ant runlocaltests
```

To run the tests against a remote MAServer instamce, make sure the remote instance is running, and type:

```
ant runremotetests
```

Examine the output for error messages. Usually either all of the tests will run successfully, or all of them will fail (usually because the MAServer instance isn't started or is blocked by a firewall).

License

MAServer is licensed under the same NCSA style open source license as MorphAdorner. See the **license.txt** file for details of this license.

Documentation

The maserver.pdf file in the doc/ directory contains minimal documentation. At present this consists of an Adobe acrobat PDF file of the web application description language (WADL) for each MAServer service in human-legible format. Better documentation is in preparation. When ready it will appear online as part of the main MorphAdorner documentation at

http://morphadorner.northwestern.edu/

as well as in printable (PDF) format.

You may also access the WADL (web application description language) definitions for all the services using a web browser. Start the local version of the MAServer server using the runmaserver.bat (Windows) or runmaserver script (Unix and Mac OS X). Then open the following site in your web browser:

http://localhost:8182/?method=options

The WADL for an individual service can be retrieved using

http://localhost:8182/servicename?method=options

and replacing "servicename" with the name of the MAServer service for which you want the documentation. For example, the WADL for the lemmatizer service can be retrieved with:

http://localhost:8182/lemmatizer?method=options

If your system provides the curl utility, you can retrieve the XML formatted WADL descriptions for all services using curl in a console/terminal window as follows:

```
curl http://localhost:8182/?method=options
```

You can retrieve the WADL XML for a particular service -- say the lemmatizer service -- as follows:

```
    curl http://localhost:8182/lemmatizer?method=options
```

You can also retrieve the WADL descriptions from a remotely installed MAServer installation by replacing "localhost:8182" with the server name and server port of the remote server. Examples:

```
    http://myremotehost.com/maserver/?method=options
    http://myremotehost.com/maserver/lemmatizer/?method=options
```

Replace "myremotehost.com" with the name (and optionally the port number) of your remote MAServer instance.

Accessing the services

Please see “Accessing the MorphAdorner server programmatically” (page Error: Reference source not found) for details on accessing the MorphAdorner Server facilties from programs.

MorphAdorner Server: Accessing the server programmatically

How the MorphAdorner Server operates

MorphAdorner Server, or MAServer for short, is an HTTP-based server which exposes MorphAdorner facilities over the web. Communication with the server takes place using ordinary HTTP protocol GET, POST and OPTIONS requests. The input format for all services is sent to the server encoded as an HTML form. Most services return output in one of four selectable formats: JSON, XML, HTML, or plain text. The remaining services which accept XML files as input only return XML files as output.

MAServer is written using the Restlet web framework.

MAServer can be accessed using an ordinary web browser or via custom programs. Any programming language can be used as long as it supports sending HTML form data to the server over HTTP, and can receive responses over HTTP.

The simplest way to access the server is to use forms in plain web pages. Sample web forms appear on the page defining the service parameters for each service.

Common features of the services

All of the services support cross-origin resource sharing (CORS). This means you can access the services from a JavaScript script running on any client system. Most web browsers issued the past few years support CORS, allowing you to use dynamic scripting (Ajax) to access MAServer and embed the results in web pages dynamically.

Support of GET versus POST

All of the plain text services allow access using either HTTP GET or POST. POST is better when the amount of text to process is larger than a few hundred characters, as some systems do not handle large amounts of text through a GET request.

The TEI XML services that accept files as input or output only work with POST requests.

Media format of service responses

The plain text services can generate responses in four different formats.

- Plain utf-8 text. This is convenient if you want the results in a simple text format.
- HTML. This is convenient if you want to embed the results in a web page -- for example, if you intend to display the results using Ajax calls from a Javascript program.
- JSON. This is convenient for access from script languages like Python.
- XML. This is convenient if you want to postprocess the results using XSLT scripts.

The TEI XML services always generate responses in XML format. The response may optionally be sent as an attached file, which is the most convenient response format when the request is submitted from a plain web form.

Using WADL to view the service query parameters

You may also access the WADL (web application description language) definitions for all the services using a web browser. Start the local version of the MAServer server using the runmaserver.bat (Windows) or runmaserver script (Unix and Mac OS X). Then open the following site in your web browser:

```
http://localhost:8182/?method=options
```

The WADL for an individual service can be retrieved using

```
http://localhost:8182/servicename?method=options
```

and replacing "servicename" with the name of the MAServer service for which you want the documentation. For example, the WADL for the lemmatizer service can be retrieved with:

```
http://localhost:8182/lemmatizer?method=options
```

If your system provides the curl utility, you can retrieve the XML formatted WADL descriptions for all services using curl in a console/terminal window as follows:

```
curl http://localhost:8182/?method=options
```

You can retrieve the WADL XML for a particular service -- say the lemmatizer service -- as follows:

```
curl http://localhost:8182/lemmatizer?method=options
```

You can also retrieve the WADL descriptions from a remotely installed MAServer installation by replacing "localhost:8182" with the server name and server port of the remote server. Examples:

```
http://myremotehost.com/maserver/?method=options
http://myremotehost.com/maserver/lemmatizer/?method=options
```

Replace "myremotehost.com" with the name (and optionally the port number) of your remote MAServer instance.

Accessing the server from a web page

The simplest way to access the server is using an ordinary web form on a web page. Following are two examples: using the plain-text Lemmatizer service, and using the TEI XML tokenizer service.

Example: accessing the Lemmatizer service

Here is a sample form for accessing the Lemmatizer service using a POST request. You must replace "myremotehost.com" in the form *action*= parameter with the name of your MorphAdorner server.

```
<form accept-charset="UTF-8" method="post"
      action="http://myremotehost.com/maserver/lemmatizer"
      target="_blank"
      name="lemmatizer">
<table cellpadding="0" cellspacing="5">
<tr>
<td><strong>Spelling:</strong></td>
<td><input type="text" name="spelling" size="20" value="" /></td>
</tr>
<tr>
<td>
```

```
<td><input type="checkbox" name="standardize" value="true" checked="checked"
/>Standardize spelling</td>
</td>
<td>&nbsp;</td>
</tr>
<tr>
<td><strong>Primary word class:</strong></td>
<td>
<select name="wordClass">
<option value="" selected="selected"></option>
<option value="adjective">adjective</option>
<option value="adverb">adverb</option>
<option value="compound">compound</option>
<option value="conjunction">conjunction</option>
<option value="infinitive-to">infinitive-to</option>
<option value="noun">noun</option>
<option value="noun-possessive">noun-possessive</option>
<option value="preposition">preposition</option>
<option value="pronoun">pronoun</option>
<option value="pronoun-possessive">pronoun-possessive</option>
<option value="pronoun-possessive-determiner">pronoun-possessive-
determiner</option>
<option value="verb">verb</option>
</select>
</td>
</tr>
<tr>
<td><strong>Secondary word class:</strong></td>
<td>
<select name="wordClass2">
<option value="" selected="selected"></option>
<option value="adjective">adjective</option>
<option value="adverb">adverb</option>
<option value="compound">compound</option>
<option value="conjunction">conjunction</option>
<option value="infinitive-to">infinitive-to</option>
<option value="noun">noun</option>
<option value="noun-possessive">noun-possessive</option>
<option value="preposition">preposition</option>
<option value="pronoun">pronoun</option>
<option value="pronoun-possessive">pronoun-possessive</option>
<option value="pronoun-possessive-determiner">pronoun-possessive-
determiner</option>
<option value="verb">verb</option>
</select>
</td>
</tr>
<tr>
<td valign="top">
<strong>
Lexicon:</strong>
</td>
<td>
<input type="radio" name="corpusConfig" value="eme">Early Modern English</input><br
/>
<input type="radio" name="corpusConfig" value="ece">Eighteen Century
English</input><br />
```

```
<input type="radio" name="corpusConfig" value="ncf" checked="checked">Nineteenth
Century Fiction</input>
</td>
</tr>
<tr>
<td>
&nbsp;
</td>
<td>
&nbsp;
</td>
</tr>
<tr>
<td valign="top">
<strong>Results format:</strong>
</td>
<td>
<input type="radio" name="media" value="json">JSON format</input><br />
<input type="radio" name="media" value="xml" checked="checked">XML
format</input><br />
<input type="radio" name="media" value="html">HTML format</input><br />
<input type="radio" name="media" value="text">Text format</input>
</td>
</tr>
<tr>
<td>
&nbsp;
</td>
<td>
&nbsp;
</td>
</tr>
<tr>
<td>
<input type="submit" name="lemmatize" value="Lemmatize" />
</td>
</tr>
</table>
</form>
```

After the user fills out the form and selects the **Lemmatize** button, the results will be returned in the selected format onto a new web page.

Example: accessing the Lemmatizer service using Javascript and Ajax

We can modify the form above slightly to demonstrate accessing the server and inserting the results into a <div> on the same page. We use the jQuery library to provide the Javascript and Ajax access. The jQuery libraries (**jquery-1.10.1.min.js** and **jquery.form.min.js**) should be inserted in the <head> section of the web page. We also add a custom function *ajaxifyForm* stored in **ajaxify.js** which wraps the JQuery functions to convert a standard web form into an Ajax compatible form.

```
<script src="jquery-1.10.1.min.js" type="text/javascript">
</script>
<script src="jquery.form.min.js" type="text/javascript">
</script>
<script src="ajaxify.js" type="text/javascript">
```

```
</script>
```

The contents of **ajaxify.js** is:

```
function ajaxifyForm( formID )
{
    var options =
    {
        target: '#results',            // Target element to be updated with
                                       // server response.
        error:
           function( xhr, textStatus, errorThrown )
           {
              $('#results').empty().append( "<strong>Error:</strong>&nbsp;" );
              $('#results').append(
                  "Server returned error code " + xhr.status + ": " );
              $('#results').append( textStatus + ": " + errorThrown );
           }
    };
                                       // Bind to the form's submit event.
    $( "#" + formID ).submit
    (
        function()
        {
                                       // Inside event callbacks 'this' is
                                       // the DOM element so wrap it in a
                                       // jQuery object and then invoke
                                       // ajaxSubmit.
            $(this).ajaxSubmit( options );
                                       // Return false to prevent standard
                                       // browser submit and page
                                       // navigation.
            return false;
        }
    );
}
```

The lemmatizer request form is essentially the same as the plain web form except that we force selection of the HTML result type using a hidden input form field. This is so we can insert the HTML results directly into the "results" div element.

```
<input type="hidden" name="media" value="html" />
```

Following the end of the form definition we add a jQuery script request which converts the form into an Ajax request using the *ajaxifyForm* we defined above. We then add a <div> with the name "result" which will hold the HTML formatted results return by the server.

```
<script type="text/javascript">
    $(document).ready( ajaxifyForm( 'lemmatizer' ) );
<div id="results" name="results" class="results">
</div>
```

After the user fills out the form and selects the **Lemmatize** button, the results will be returned in the *results* div below the form.

Here is the entire sample web page for requesting an "ajaxified" lemmatization. Again, you must replace "myremotehost.com" in the form *action=* parameter with the name of your MorphAdorner server.

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<title>Access Lemmatizer Service using Javascript/Ajax</title>
<script src="jquery-1.10.1.min.js" type="text/javascript">
</script>
<script src="jquery.form.min.js" type="text/javascript">
</script>
<script src="ajaxify.js" type="text/javascript">
</script>
</head>
<body>
<form accept-charset="UTF-8" method="post"
      action="http://myremotehost.com/maserver/lemmatizer"
      name="lemmatizer" id="lemmatizer">
<table cellpadding="0" cellspacing="5">
<tr>
<td><strong>Spelling:</strong></td>
<td><input type="text" name="spelling" size="20" value="" /></td>
</tr>
<tr>
<td>
<td><input type="checkbox" name="standardize" value="true" checked="checked"
/>Standardize spelling<br />
<input type="hidden" name="media" value="html" />
</td>
<td>&nbsp;</td>
</tr>
<tr>
<td><strong>Primary word class:</strong></td>
<td>
<select name="wordClass">
<option value="" selected="selected"></option>
<option value="adjective">adjective</option>
<option value="adverb">adverb</option>
<option value="compound">compound</option>
<option value="conjunction">conjunction</option>
<option value="infinitive-to">infinitive-to</option>
<option value="noun">noun</option>
<option value="noun-possessive">noun-possessive</option>
<option value="preposition">preposition</option>
<option value="pronoun">pronoun</option>
<option value="pronoun-possessive">pronoun-possessive</option>
<option value="pronoun-possessive-determiner">pronoun-possessive-
determiner</option>
<option value="verb">verb</option>
</select>
</td>
</tr>
<tr>
<td><strong>Secondary word class:</strong></td>
<td>
<select name="wordClass2">
<option value="" selected="selected"></option>
<option value="adjective">adjective</option>
<option value="adverb">adverb</option>
<option value="compound">compound</option>
```

```
<option value="conjunction">conjunction</option>
<option value="infinitive-to">infinitive-to</option>
<option value="noun">noun</option>
<option value="noun-possessive">noun-possessive</option>
<option value="preposition">preposition</option>
<option value="pronoun">pronoun</option>
<option value="pronoun-possessive">pronoun-possessive</option>
<option value="pronoun-possessive-determiner">pronoun-possessive-
determiner</option>
<option value="verb">verb</option>
</select>
</td>
</tr>
<tr>
<td valign="top">
<strong>
Lexicon:</strong>
</td>
<td>
<input type="radio" name="corpusConfig" value="eme">Early Modern English</input><br
/>
<input type="radio" name="corpusConfig" value="ece">Eighteen Century
English</input><br />
<input type="radio" name="corpusConfig" value="ncf" checked="checked">Nineteenth
Century Fiction</input>
</td>
</tr>
<tr>
<td>
<input type="submit" name="lemmatize" value="Lemmatize" />
</td>
</tr>
</table>
</form>
<script type="text/javascript">
    $(document).ready( ajaxifyForm( 'lemmatizer' ) );
</script>
<div id="results" name="results">
</div>
</body>
</html>
```

Example: accessing the Lemmatizer service using an iframe

Some users may elect to turn off JavaScript in their browsers. In this case the Ajax solution above will not work. An alternative is to display the results of the query in an internal frame or *iframe*.

The lemmatizer request form is once again mostly the same as the plain web form except that we force selection of the HTML result type using a hidden input form field and set the name of an iframe as the target to receive the lemmatizer results using *target=*. The main downside of the iframe solution is that the results may display in a different style than the rest of the text on the page. The *seamless* option on the *iframe* definition is intended to correct this, but few browsers support this yet.

The iframe approach is also useful if you want to embed a MorphAdorner server facility into another web page over which you may not have full control. For example, you can use the iframe approach to add a MorphAdorner lemmatizer form to a WordPress blog posting.

Here is a sample form for requesting an iframe version of lemmatization. The iframe target element which receives the lemmatization results is named "resultsiframe". Again, you must replace "myremotehost.com" in the form *action=* parameter with the name of your MorphAdorner server.

```
<form accept-charset="UTF-8" method="post"
      action="http://myremotehost.com/maserver/lemmatizer"
      target="resultsiframe"
      name="lemmatizer">
<table cellpadding="0" cellspacing="5">
<tr>
<td><strong>Spelling:</strong></td>
<td><input type="text" name="spelling" size="20" value="" /></td>
</tr>
<tr>
<td>
<td><input type="checkbox" name="standardize" value="true" checked="checked"
/>Standardize spelling</td>
<input type="hidden" name="media" value="html" />
</td>
<td>&nbsp;</td>
</tr>
<tr>
<td><strong>Primary word class:</strong></td>
<td>
<select name="wordClass">
<option value="" selected="selected"></option>
<option value="adjective">adjective</option>
<option value="adverb">adverb</option>
<option value="compound">compound</option>
<option value="conjunction">conjunction</option>
<option value="infinitive-to">infinitive-to</option>
<option value="noun">noun</option>
<option value="noun-possessive">noun-possessive</option>
<option value="preposition">preposition</option>
<option value="pronoun">pronoun</option>
<option value="pronoun-possessive">pronoun-possessive</option>
<option value="pronoun-possessive-determiner">pronoun-possessive-
determiner</option>
<option value="verb">verb</option>
</select>
</td>
</tr>
<tr>
<td><strong>Secondary word class:</strong></td>
<td>
<select name="wordClass2">
<option value="" selected="selected"></option>
<option value="adjective">adjective</option>
<option value="adverb">adverb</option>
<option value="compound">compound</option>
<option value="conjunction">conjunction</option>
<option value="infinitive-to">infinitive-to</option>
<option value="noun">noun</option>
<option value="noun-possessive">noun-possessive</option>
<option value="preposition">preposition</option>
```

```
<option value="pronoun">pronoun</option>
<option value="pronoun-possessive">pronoun-possessive</option>
<option value="pronoun-possessive-determiner">pronoun-possessive-
determiner</option>
<option value="verb">verb</option>
</select>
</td>
</tr>
<tr>
<td valign="top">
<strong>
Lexicon:</strong>
</td>
<td>
<input type="radio" name="corpusConfig" value="eme">Early Modern English</input><br
/>
<input type="radio" name="corpusConfig" value="ece">Eighteen Century
English</input><br />
<input type="radio" name="corpusConfig" value="ncf" checked="checked">Nineteenth
Century Fiction</input>
</td>
</tr>
<tr>
<td>
&nbsp;
</td>
<td>
&nbsp;
</td>
</tr>
<tr>
<td>
<input type="submit" name="lemmatize" value="Lemmatize" />
</td>
</tr>
</table>
</form>
<iframe id="resultsiframe" name="resultsiframe" frameborder="0"
        width="100%" scrolling="auto" seamless="true" height="500px"
        frameborder="0"
        >
</iframe>
```

Using an iframe as a fallback when JavaScript is not enabled

We can combine the Ajax/JavaScript and iframe approaches. If JavaScript is enabled, we use the Ajax approach. If JavaScript is not enabled, we fallback to the iframe approach. The <*noscript*> block which defines the iframe is only used if JavaScript is disabled or not supported. This combined approach works in the majority of modern web browsers.

Here is the entire sample web page for requesting an Ajaxified lemmatization result with a fallback to an iframe version when JavaScript is not enabled. As before, you must replace "myremotehost.com" in the form *action=* parameter with the name of your MorphAdorner server.

```
<html>
<head>
```

```
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<title>Access Lemmatizer Service using Javascript/Ajax</title>
<script src="jquery-1.10.1.min.js" type="text/javascript">
</script>
<script src="jquery.form.min.js" type="text/javascript">
</script>
<script src="ajaxify.js" type="text/javascript">
</script>
</head>
<body>
<form accept-charset="UTF-8" method="post"
      action="http://myremotehost.com/maserver/lemmatizer"
      target="resultsiframe"
      name="lemmatizer" id="lemmatizer">
<table cellpadding="0" cellspacing="5">
<tr>
<td><strong>Spelling:</strong></td>
<td><input type="text" name="spelling" size="20" value="" /></td>
</tr>
<tr>
<td>
<td><input type="checkbox" name="standardize" value="true" checked="checked"
/>Standardize spelling<br />
<input type="hidden" name="media" value="html" />
</td>
<td>&nbsp;</td>
</tr>
<tr>
<td><strong>Primary word class:</strong></td>
<td>
<select name="wordClass">
<option value="" selected="selected"></option>
<option value="adjective">adjective</option>
<option value="adverb">adverb</option>
<option value="compound">compound</option>
<option value="conjunction">conjunction</option>
<option value="infinitive-to">infinitive-to</option>
<option value="noun">noun</option>
<option value="noun-possessive">noun-possessive</option>
<option value="preposition">preposition</option>
<option value="pronoun">pronoun</option>
<option value="pronoun-possessive">pronoun-possessive</option>
<option value="pronoun-possessive-determiner">pronoun-possessive-
determiner</option>
<option value="verb">verb</option>
</select>
</td>
</tr>
<tr>
<td><strong>Secondary word class:</strong></td>
<td>
<select name="wordClass2">
<option value="" selected="selected"></option>
<option value="adjective">adjective</option>
<option value="adverb">adverb</option>
<option value="compound">compound</option>
<option value="conjunction">conjunction</option>
```

```
<option value="infinitive-to">infinitive-to</option>
<option value="noun">noun</option>
<option value="noun-possessive">noun-possessive</option>
<option value="preposition">preposition</option>
<option value="pronoun">pronoun</option>
<option value="pronoun-possessive">pronoun-possessive</option>
<option value="pronoun-possessive-determiner">pronoun-possessive-
determiner</option>
<option value="verb">verb</option>
</select>
</td>
</tr>
<tr>
<td valign="top">
<strong>
Lexicon:</strong>
</td>
<td>
<input type="radio" name="corpusConfig" value="eme">Early Modern English</input><br
/>
<input type="radio" name="corpusConfig" value="ece">Eighteen Century
English</input><br />
<input type="radio" name="corpusConfig" value="ncf" checked="checked">Nineteenth
Century Fiction</input>
</td>
</tr>
<tr>
<td>
<input type="submit" name="lemmatize" value="Lemmatize" />
</td>
</tr>
</table>
</form>
<script type="text/javascript">
    $(document).ready( ajaxifyForm( 'lemmatizer' ) );
</script>
<noscript>
<iframe id="resultsiframe" name="resultsiframe" frameborder="0"
        width="100%" scrolling="auto" seamless="true" height="500px"
        frameborder="0"
        >
</iframe>
</noscript>
<div id="results" name="results">
</div>
</body>
</html>
```

Example: accessing the Tokenize TEI file service

Here is a sample form for accessing the TEI XML tokenizer service using a POST request. You must replace "myremotehost.com" in the form *action=* parameter with the name of your MorphAdorner server.

```
<form accept-charset="UTF-8" method="post"
      action="http://myremotehost.com/maserver/teitokenizer"
      target="_blank"
      enctype="multipart/form-data" name="teitokenizer">
<table cellpadding="0" cellspacing="5">
<tr>
<td>
<strong>TEI XML file:</strong>
</td>
<td>
<input type="file" name="teifile" size="50">
</td>
</tr>
<tr>
<td>
&nbsp;
</td>
<td>
&nbsp;
</td>
</tr>
<tr>
<td>&nbsp;</td>
<td>
<input type="checkbox" name="resultsAsAttachedFile" value="true"
       checked="checked"/>
Send results as attached file
</td>
</tr>
<tr>
<td valign="top">
<strong>
Lexicon:</strong>
</td>
<td>
<input type="radio" name="corpusConfig" value="eme">Early Modern English</input><br
/>
<input type="radio" name="corpusConfig" value="ece">Eighteen Century
English</input><br />
<input type="radio" name="corpusConfig" value="ncf" checked="checked">Nineteenth
Century Fiction</input>
</td>
</tr>
<tr>
<td>
&nbsp;
</td>
<td>
&nbsp;
</td>
</tr>
<tr>
<td colspan="2">
<input type="submit" name="tokenize" value="Tokenize" />
</td>
</tr>
</table>
</form>
```

After the user fills out the form and selects the **Tokenize** button, the tokenized TEI file will be returned either as an attached file (if the "Send results as attached file" option is selected) or as an XML stream (on a separate web page). In many cases it may be simplest to force the selection of the "Send results as attached file" option.

Accessing the server from a Java program

You can access the MorphAdorner Server using any programming language which supports access to remote servers using the HTTP protocol. However, since MAServer is written using the Restlet framework, it is convenient to use the Restlet libraries to access the server from a Java program.

Example: accessing the Lemmatizer service from a Java program

Here is a simple Java program which accesses the Lemmatizer service using the Restlet libraries. As an example, we will request the lemma form of the obsolete spelling "strykynge" using the Early Modern English corpus.

```
import org.restlet.*;
import org.restlet.resource.*;
import edu.northwestern.at.morphadorner.server.*;
public class trylemmatizer
{
    public static void main( String[] args )
    {
                                //    Create lemmatizer client resource.
        ClientResource resource    =
            new ClientResource
            (
                "http://myremotehost.com/maserver/lemmatizer"
            );
                                //    Add query parameters.
        resource.addQueryParameter( "spelling" , "strykynge" );
        resource.addQueryParameter( "corpusConfig" , "eme" );
        resource.addQueryParameter( "wordClass" , "verb" );
        resource.addQueryParameter( "standardize" , "true" );
        resource.addQueryParameter( "media" , "xml" );
                                //    Get result from server.
        try
        {
            LemmatizerResult result    = resource.get( LemmatizerResult.class );
                                //    Display resultant lemma.
            System.out.println( "lemma: " + result.lemma );
        }
        catch ( Exception e )
        {
            System.out.println( "Error: " + e.getMessage() );
        }
    }
}
```

We create a Restlet client resource, add the service parameters, and perform an HTTP GET request. We request the results to be returned as XML because Restlet can automtically convert the XML to a Java object -- in this case, a *LemmatizerResult* object. We then display the resultant lemma value from the *LemmatizerResult* object, or display an error message if the server returns an error.

Example: accessing the Tokenize TEI file service

This example demonstrates sending a TEI XML file to the server for tokenization using a POST request. We request the tokenized output as XML, store it in a generic *Representation* object, and pull out the tokenized text from the *Representation* using the *getText* method. We then display the tokenized text or an error message, if any, from the server.

```
import org.restlet.*;
import org.restlet.ext.html.*;
import org.restlet.data.*;
import org.restlet.representation.*;
import org.restlet.resource.*;
import edu.northwestern.at.morphadorner.server.*;
public class tryteitokenizer
{
    public static void main( String[] args )
    {
                                //    Create client resource.
        ClientResource resource    =
            new ClientResource
            (
                "http://myremotehost.com/maserver/teitokenizer"
            );
                                //    Create form with query parameters.
        FormDataSet form    = new FormDataSet();
        form.setMultipart( true );
        form.getEntries().add( new FormData( "corpusConfig" , "ncf" ) );
        form.getEntries().add( new FormData( "media" , "xml" ) );
        form.getEntries().add(
            new FormData( "resultsAsAttachedFile" , "false" ) );
        FileRepresentation file    =
            new FileRepresentation(
                "j:/marylamb.xml" , MediaType.APPLICATION_XML );
        form.getEntries().add( new FormData(" teifile" , file ) );
                                //    Get result XML from server.
        try
        {
            Representation result    = resource.post( form );
            System.out.println( result.getText() );
        }
        catch ( Exception e )
        {
            System.out.println( "Error: " + e.getMessage() );
        }
    }
}
```

MorphAdorner Server Services: Adorn Plain Text Service

Service name:	partofspeechtagger
Service description:	Adorn words with their parts of speech.
HTTP methods allowed:	GET, POST, OPTIONS
POST accepts as input:	application/x-www-form-urlencoded
HTTP return codes:	200: service succeeded 400: service failed with an error
Query parameters	
corpusConfig	Corpus configuration name. In the standard distribution these are *ece*, *eme*, and *ncf*.
media	Result format. One of *json, xml, html, text* .
text	Text to be processed.
includeInputText	Allowed values are *true* to include the input text in the output and *false* to not include the input text.
outputReg	Output standardized spelling in TEI XML format. Allowed values are *true* to output the standard spelling, *false* to not output the standard spelling.
XML output style.	Select *outputPlainXML* for plain XML, *outputTEI* for TEI format XML, or *outputTCF* for WebLicht TCF format XML.

Sample POST form

```
<form accept-charset="UTF-8" method="post" action="partofspeechtagger"
      target="_blank"
      name="postagger">
<table cellpadding="0" cellspacing="5">
<tr>
<td><strong>Text:</strong></td>
<td colspan="2">
<textarea name="text" rows="15" cols="76"></textarea>
</td>
</tr>
<tr>
<td valign="top">
<strong>
Lexicon:</strong>
</td>
<td>
<input type="radio" name="corpusConfig" value="eme">Early Modern English</input><br
/>
<input type="radio" name="corpusConfig" value="ece">Eighteen Century
English</input><br />
<input type="radio" name="corpusConfig" value="ncf" checked="checked">Nineteenth
Century Fiction</input>
</td>
</tr>
```

```
<tr>
<td>&nbsp;</td>
<td>
<input type="checkbox" name="includeInputText" value="true"
       checked="checked"/>
Include input text in results
</td>
</tr>
<tr>
<td>
&nbsp;
</td>
<td>
&nbsp;
</td>
</tr>
<tr>
<td valign="top">
<strong>Results format:</strong>
</td>
<td>
<input type="radio" name="media" value="json">JSON format</input><br />
<input type="radio" name="media" value="xml" checked="checked">XML
format</input><br />
&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;
<input type="radio" name="xmlOutputType" value="outputPlainXML"
checked="checked">Plain XML</input><br />
&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;
<input type="radio" name="xmlOutputType" value="outputTEI">Fragmentary TEI format
XML</input><br />
&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;
<input type="checkbox" name="outputReg" value="false" />Add reg= attribute for
standard spelling<br />
&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;
<input type="radio" name="xmlOutputType" value="outputTCF">WebLicht TCF format
XML</input><br />
<input type="radio" name="media" value="html">HTML format</input><br />
<input type="radio" name="media" value="text">Text format</input>
</td>
</tr>
<tr>
<td>
&nbsp;
</td>
<td>
&nbsp;
</td>
</tr>
<tr>
<td colspan="2">
<input type="submit" name="adorn" value="Adorn" />
</td>
</tr>
</table>
</form>
```

Output

Here we adorn the first two sentences of Sarah Hale's poem "Mary had a little lamb."

> Mary had a little lamb,
> whose fleece was white as snow.
> And everywhere that Mary went,
> the lamb was sure to go.

The JSON and XML *PartOfSpeechTaggerResult* echoes the input *text* and the *corpusConfig*. The *sentences* container wraps a sequence of *sentence* entries each of which represents a single parsed sentence from the input text. Each *sentence* contains a sequence of *token* entries representing the words and punctuation in the sentence. Following this is an *adornedSentences* container which contains a sequence of *adornedSentence* entries. Each *adornedSentence* contains a sequence of *adornedWord* entries containing the morphological adornments.

For XML format output, the alternate output formats provide different formatting but the same basic information.

The HTML and text versions provide tabular versions of the adorned sentences.

JSON output

```
{
  "PartOfSpeechTaggerResult": {
    "text": "Mary had a little lamb,  whose fleece was white as snow.  And
everywhere that Mary went,  the lamb was sure to go.",
    "corpusConfig": "ncf",
    "sentences": [
      {
        "sentence": [
          {
            "token": [
              "Mary",
              "had",
              "a",
              "little",
              "lamb",
              ",",
              "whose",
              "fleece",
              "was",
              "white",
              "as",
              "snow",
              "."
            ]
          },
          {
            "token": [
              "And",
              "everywhere",
              "that",
              "Mary",
              "went",
            ",",
            "the",
            "lamb",
            "was",
            "sure",
            "to",
            "go",
            "."
          ]
        }
      ]
    }
  ],
  "adornedSentences": [
    {
      "adornedSentence": [
        {
          "adornedWord": [
            {
              "token": "Mary",
              "spelling": "Mary",
              "standardSpelling": "Mary",
              "lemmata": "Mary",
              "partsOfSpeech": "np1"
            },
            {
              "token": "had",
              "spelling": "had",
              "standardSpelling": "had",
              "lemmata": "have",
              "partsOfSpeech": "vhd"
            },
            {
              "token": "a",
              "spelling": "a",
              "standardSpelling": "a",
              "lemmata": "a",
              "partsOfSpeech": "dt"
            },
            {
              "token": "little",
              "spelling": "little",
              "standardSpelling": "little",
              "lemmata": "little",
              "partsOfSpeech": "j"
            },
            {
              "token": "lamb",
              "spelling": "lamb",
              "standardSpelling": "lamb",
              "lemmata": "lamb",
              "partsOfSpeech": "n1"
            },
            {
              "token": ",",
              "spelling": ",",
              "standardSpelling": ",",
          "lemmata": ",",
          "partsOfSpeech": ","
        },
        {
          "token": "whose",
          "spelling": "whose",
          "standardSpelling": "whose",
          "lemmata": "who",
          "partsOfSpeech": "r-crq"
        },
        {
          "token": "fleece",
          "spelling": "fleece",
          "standardSpelling": "fleece",
          "lemmata": "fleece",
          "partsOfSpeech": "n1"
        },
        {
          "token": "was",
          "spelling": "was",
          "standardSpelling": "was",
          "lemmata": "be",
          "partsOfSpeech": "vbds"
        },
        {
          "token": "white",
          "spelling": "white",
          "standardSpelling": "white",
          "lemmata": "white",
          "partsOfSpeech": "j-jn"
        },
        {
          "token": "as",
          "spelling": "as",
          "standardSpelling": "as",
          "lemmata": "as",
          "partsOfSpeech": "c-acp"
        },
        {
          "token": "snow",
          "spelling": "snow",
          "standardSpelling": "snow",
          "lemmata": "snow",
          "partsOfSpeech": "n1"
        },
        {
          "token": ".",
          "spelling": ".",
          "standardSpelling": ".",
          "lemmata": ".",
          "partsOfSpeech": "."
        }
      ]
    },
    {
      "adornedWord": [
        {
```

```
    "token": "And",
    "spelling": "And",
    "standardSpelling": "And",
    "lemmata": "and",
    "partsOfSpeech": "cc"
  },
  {
    "token": "everywhere",
    "spelling": "everywhere",
    "standardSpelling": "everywhere",
    "lemmata": "everywhere",
    "partsOfSpeech": "av"
  },
  {
    "token": "that",
    "spelling": "that",
    "standardSpelling": "that",
    "lemmata": "that",
    "partsOfSpeech": "cst"
  },
  {
    "token": "Mary",
    "spelling": "Mary",
    "standardSpelling": "Mary",
    "lemmata": "Mary",
    "partsOfSpeech": "np1"
  },
  {
    "token": "went",
    "spelling": "went",
    "standardSpelling": "went",
    "lemmata": "go",
    "partsOfSpeech": "vvd"
  },
  {
    "token": ",",
    "spelling": ",",
    "standardSpelling": ",",
    "lemmata": ",",
    "partsOfSpeech": ","
  },
  {
    "token": "the",
    "spelling": "the",
    "standardSpelling": "the",
    "lemmata": "the",
    "partsOfSpeech": "dt"
  },
  {
    "token": "lamb",
    "spelling": "lamb",
    "standardSpelling": "lamb",
    "lemmata": "lamb",
    "partsOfSpeech": "n1"
  },
  {
    "token": "was",
              "spelling": "was",
              "standardSpelling": "was",
              "lemmata": "be",
              "partsOfSpeech": "vbds"
            },
            {
              "token": "sure",
              "spelling": "sure",
              "standardSpelling": "sure",
              "lemmata": "sure",
              "partsOfSpeech": "j"
            },
            {
              "token": "to",
              "spelling": "to",
              "standardSpelling": "to",
              "lemmata": "to",
              "partsOfSpeech": "pc-acp"
            },
            {
              "token": "go",
              "spelling": "go",
              "standardSpelling": "go",
              "lemmata": "go",
              "partsOfSpeech": "vvi"
            },
            {
              "token": ".",
              "spelling": ".",
              "standardSpelling": ".",
              "lemmata": ".",
              "partsOfSpeech": "."
            }
          ]
        }
      ]
    }
  ],
  "outputTEI": false,
  "outputReg": false,
  "outputTCF": false
 }
}
```

XML output

```
<?xml version="1.0"?>
<PartOfSpeechTaggerResult>
    <text>Mary had a little lamb,  whose fleece was white as snow.  And everywhere
that Mary went,  the lamb was sure to go.</text>
    <corpusConfig>ncf</corpusConfig>
    <sentences>
        <sentence>
            <token>Mary</token>
            <token>had</token>
            <token>a</token>
            <token>little</token>
            <token>lamb</token>
            <token>,</token>
            <token>whose</token>
            <token>fleece</token>
            <token>was</token>
            <token>white</token>
            <token>as</token>
            <token>snow</token>
            <token>.</token>
        </sentence>
        <sentence>
            <token>And</token>
            <token>everywhere</token>
            <token>that</token>
            <token>Mary</token>
            <token>went</token>
            <token>,</token>
            <token>the</token>
            <token>lamb</token>
            <token>was</token>
            <token>sure</token>
            <token>to</token>
            <token>go</token>
            <token>.</token>
        </sentence>
    </sentences>
    <adornedSentences>
        <adornedSentence>
            <adornedWord>
                <token>Mary</token>
                <spelling>Mary</spelling>
                <standardSpelling>Mary</standardSpelling>
                <lemmata>Mary</lemmata>
                <partsOfSpeech>np1</partsOfSpeech>
            </adornedWord>
            <adornedWord>
                <token>had</token>
                <spelling>had</spelling>
                <standardSpelling>had</standardSpelling>
                <lemmata>have</lemmata>
                <partsOfSpeech>vhd</partsOfSpeech>
            </adornedWord>
            <adornedWord>
                <token>a</token>
                <spelling>a</spelling>
                <standardSpelling>a</standardSpelling>
                <lemmata>a</lemmata>
                <partsOfSpeech>dt</partsOfSpeech>
            </adornedWord>
            <adornedWord>
                <token>little</token>
                <spelling>little</spelling>
                <standardSpelling>little</standardSpelling>
                <lemmata>little</lemmata>
                <partsOfSpeech>j</partsOfSpeech>
            </adornedWord>
```

```
<adornedWord>
    <token>lamb</token>
    <spelling>lamb</spelling>
    <standardSpelling>lamb</standardSpelling>
    <lemmata>lamb</lemmata>
    <partsOfSpeech>n1</partsOfSpeech>
</adornedWord>
<adornedWord>
    <token>,</token>
    <spelling>,</spelling>
    <standardSpelling>,</standardSpelling>
    <lemmata>,</lemmata>
    <partsOfSpeech>,</partsOfSpeech>
</adornedWord>
<adornedWord>
    <token>whose</token>
    <spelling>whose</spelling>
    <standardSpelling>whose</standardSpelling>
    <lemmata>who</lemmata>
    <partsOfSpeech>r-crq</partsOfSpeech>
</adornedWord>
<adornedWord>
    <token>fleece</token>
    <spelling>fleece</spelling>
    <standardSpelling>fleece</standardSpelling>
    <lemmata>fleece</lemmata>
    <partsOfSpeech>n1</partsOfSpeech>
</adornedWord>
<adornedWord>
    <token>was</token>
    <spelling>was</spelling>
    <standardSpelling>was</standardSpelling>
    <lemmata>be</lemmata>
    <partsOfSpeech>vbds</partsOfSpeech>
</adornedWord>
<adornedWord>
    <token>white</token>
    <spelling>white</spelling>
    <standardSpelling>white</standardSpelling>
    <lemmata>white</lemmata>
    <partsOfSpeech>j-jn</partsOfSpeech>
</adornedWord>
<adornedWord>
    <token>as</token>
    <spelling>as</spelling>
    <standardSpelling>as</standardSpelling>
    <lemmata>as</lemmata>
    <partsOfSpeech>c-acp</partsOfSpeech>
</adornedWord>
<adornedWord>
    <token>snow</token>
    <spelling>snow</spelling>
    <standardSpelling>snow</standardSpelling>
    <lemmata>snow</lemmata>
    <partsOfSpeech>n1</partsOfSpeech>
</adornedWord>
<adornedWord>
```

```
                <token>.</token>
                <spelling>.</spelling>
                <standardSpelling>.</standardSpelling>
                <lemmata>.</lemmata>
                <partsOfSpeech>.</partsOfSpeech>
            </adornedWord>
        </adornedSentence>
        <adornedSentence>
            <adornedWord>
                <token>And</token>
                <spelling>And</spelling>
                <standardSpelling>And</standardSpelling>
                <lemmata>and</lemmata>
                <partsOfSpeech>cc</partsOfSpeech>
            </adornedWord>
            <adornedWord>
                <token>everywhere</token>
                <spelling>everywhere</spelling>
                <standardSpelling>everywhere</standardSpelling>
                <lemmata>everywhere</lemmata>
                <partsOfSpeech>av</partsOfSpeech>
            </adornedWord>
            <adornedWord>
                <token>that</token>
                <spelling>that</spelling>
                <standardSpelling>that</standardSpelling>
                <lemmata>that</lemmata>
                <partsOfSpeech>cst</partsOfSpeech>
            </adornedWord>
            <adornedWord>
                <token>Mary</token>
                <spelling>Mary</spelling>
                <standardSpelling>Mary</standardSpelling>
                <lemmata>Mary</lemmata>
                <partsOfSpeech>np1</partsOfSpeech>
            </adornedWord>
            <adornedWord>
                <token>went</token>
                <spelling>went</spelling>
                <standardSpelling>went</standardSpelling>
                <lemmata>go</lemmata>
                <partsOfSpeech>vvd</partsOfSpeech>
            </adornedWord>
            <adornedWord>
                <token>,</token>
                <spelling>,</spelling>
                <standardSpelling>,</standardSpelling>
                <lemmata>,</lemmata>
                <partsOfSpeech>,</partsOfSpeech>
            </adornedWord>
            <adornedWord>
                <token>the</token>
                <spelling>the</spelling>
                <standardSpelling>the</standardSpelling>
                <lemmata>the</lemmata>
                <partsOfSpeech>dt</partsOfSpeech>
            </adornedWord>
            <adornedWord>
                <token>lamb</token>
                <spelling>lamb</spelling>
                <standardSpelling>lamb</standardSpelling>
                <lemmata>lamb</lemmata>
                <partsOfSpeech>n1</partsOfSpeech>
            </adornedWord>
            <adornedWord>
                <token>was</token>
                <spelling>was</spelling>
                <standardSpelling>was</standardSpelling>
                <lemmata>be</lemmata>
                <partsOfSpeech>vbds</partsOfSpeech>
            </adornedWord>
            <adornedWord>
                <token>sure</token>
                <spelling>sure</spelling>
                <standardSpelling>sure</standardSpelling>
                <lemmata>sure</lemmata>
                <partsOfSpeech>j</partsOfSpeech>
            </adornedWord>
            <adornedWord>
                <token>to</token>
                <spelling>to</spelling>
                <standardSpelling>to</standardSpelling>
                <lemmata>to</lemmata>
                <partsOfSpeech>pc-acp</partsOfSpeech>
            </adornedWord>
            <adornedWord>
                <token>go</token>
                <spelling>go</spelling>
                <standardSpelling>go</standardSpelling>
                <lemmata>go</lemmata>
                <partsOfSpeech>vvi</partsOfSpeech>
            </adornedWord>
            <adornedWord>
                <token>.</token>
                <spelling>.</spelling>
                <standardSpelling>.</standardSpelling>
                <lemmata>.</lemmata>
                <partsOfSpeech>.</partsOfSpeech>
            </adornedWord>
        </adornedSentence>
    </adornedSentences>
    <outputTEI>false</outputTEI>
    <outputReg>false</outputReg>
    <outputTCF>false</outputTCF>
</PartOfSpeechTaggerResult>
```

HTML output (source)

```
<h3>26 words in 2 sentences found.
</h3>
<table border="0">
<tbody><tr>
<th align="left">S#</th><th align="left">W#</th><th align="left">Spelling</th><th
```

```
align="left">Pos</th><th align="left">Standard</th><th align="left">Lemma</th></tr>
<tr><td>1</td><td>1</td><td>Mary</td><td>np1</td><td>Mary</td><td>Mary</td></tr>
<tr><td>1</td><td>2</td><td>had</td><td>vhd</td><td>had</td><td>have</td></tr>
<tr><td>1</td><td>3</td><td>a</td><td>dt</td><td>a</td><td>a</td></tr>
<tr><td>1</td><td>4</td><td>little</td><td>j</td><td>little</td><td>little</td></tr
>
<tr><td>1</td><td>5</td><td>lamb</td><td>n1</td><td>lamb</td><td>lamb</td></tr>
<tr><td>1</td><td>6</td><td>,</td><td>,</td><td>,</td><td>,</td></tr>
<tr><td>1</td><td>7</td><td>whose</td><td>r-crq</td><td>whose</td><td>who</td></tr>
<tr><td>1</td><td>8</td><td>fleece</td><td>n1</td><td>fleece</td><td>fleece</td></t
r>
<tr><td>1</td><td>9</td><td>was</td><td>vbds</td><td>was</td><td>be</td></tr>
<tr><td>1</td><td>10</td><td>white</td><td>j-
jn</td><td>white</td><td>white</td></tr>
<tr><td>1</td><td>11</td><td>as</td><td>c-acp</td><td>as</td><td>as</td></tr>
<tr><td>1</td><td>12</td><td>snow</td><td>n1</td><td>snow</td><td>snow</td></tr>
<tr><td>1</td><td>13</td><td>.</td><td>.</td><td>.</td><td>.</td></tr>
<tr><td>2</td><td>1</td><td>And</td><td>cc</td><td>And</td><td>and</td></tr>
<tr><td>2</td><td>2</td><td>everywhere</td><td>av</td><td>everywhere</td><td>everyw
here</td></tr>
<tr><td>2</td><td>3</td><td>that</td><td>cst</td><td>that</td><td>that</td></tr>
<tr><td>2</td><td>4</td><td>Mary</td><td>np1</td><td>Mary</td><td>Mary</td></tr>
<tr><td>2</td><td>5</td><td>went</td><td>vvd</td><td>went</td><td>go</td></tr>
<tr><td>2</td><td>6</td><td>,</td><td>,</td><td>,</td><td>,</td></tr>
<tr><td>2</td><td>7</td><td>the</td><td>dt</td><td>the</td><td>the</td></tr>
<tr><td>2</td><td>8</td><td>lamb</td><td>n1</td><td>lamb</td><td>lamb</td></tr>
<tr><td>2</td><td>9</td><td>was</td><td>vbds</td><td>was</td><td>be</td></tr>
<tr><td>2</td><td>10</td><td>sure</td><td>j</td><td>sure</td><td>sure</td></tr>
<tr><td>2</td><td>11</td><td>to</td><td>pc-acp</td><td>to</td><td>to</td></tr>
<tr><td>2</td><td>12</td><td>go</td><td>vvi</td><td>go</td><td>go</td></tr>
<tr><td>2</td><td>13</td><td>.</td><td>.</td><td>.</td><td>.</td></tr>
</tbody>
</table>
```

HTML output (display)

26 words in 2 sentences found.

S#	W#	Spelling	Pos	Standard	Lemma
1	1	Mary	np1	Mary	Mary
1	2	had	vhd	had	have
1	3	a	dt	a	a
1	4	little	j	little	little
1	5	lamb	n1	lamb	lamb
1	6	,	,	,	,
1	7	whose	r-crq	whose	who
1	8	fleece	n1	fleece	fleece
1	9	was	vbds	was	be
1	10	white	j-jn	white	white
1	11	as	c-acp	as	as

1	12	snow	n1	snow	snow
1	13	.	.	.	.
2	1	And	cc	And	and
2	2	everywhere	av	everywhere	everywhere
2	3	that	cst	that	that
2	4	Mary	np1	Mary	Mary
2	5	went	vvd	went	go
2	6	,	,	,	,
2	7	the	dt	the	the
2	8	lamb	n1	lamb	lamb
2	9	was	vbds	was	be
2	10	sure	j	sure	sure
2	11	to	pc-acp	to	to
2	12	go	vvi	go	go
2	13	.	.	.	.

Text output

```
26 words in 2 sentences found.
S#      W#      Spelling        Pos     Standard        Lemma
1       1       Mary    np1     Mary    Mary
1       2       had     vhd     had     have
1       3       a       dt      a       a
1       4       little  j       little  little
1       5       lamb    n1      lamb    lamb
1       6       ,       ,       ,       ,
1       7       whose   r-crq   whose   who
1       8       fleece  n1      fleece  fleece
1       9       was     vbds    was     be
1       10      white   j-jn    white   white
1       11      as      c-acp   as      as
1       12      snow    n1      snow    snow
1       13      .       .       .       .
2       1       And     cc      And     and
2       2       everywhere      av      everywhere      everywhere
2       3       that    cst     that    that
2       4       Mary    np1     Mary    Mary
2       5       went    vvd     went    go
2       6       ,       ,       ,       ,
2       7       the     dt      the     the
2       8       lamb    n1      lamb    lamb
2       9       was     vbds    was     be
2       10      sure    j       sure    sure
2       11      to      pc-acp  to      to
2       12      go      vvi     go      go
2       13      .       .       .
```

MorphAdorner Server Services: CorpusConfig Service

Service name:	corpusconfig
Service description:	List available corpus configurations.
HTTP methods allowed:	GET, POST, OPTIONS
POST accepts as input:	application/x-www-form-urlencoded
HTTP return codes:	200: service succeeded 400: service failed with an error
Query parameters	
media	Result format. One of *json, xml, html, text* .

Sample POST form

```
<form accept-charset="UTF-8" method="post" action="corpusconfig"
      target="_blank"
      name="corpusconfig">
<table cellpadding="0" cellspacing="5">
<tr>
<td valign="top">
<strong>Results format:</strong>
</td>
<td>
<input type="radio" name="media" value="json">JSON format</input><br />
<input type="radio" name="media" value="xml" checked="checked">XML
format</input><br />
<input type="radio" name="media" value="html">HTML format</input><br />
<input type="radio" name="media" value="text">Text format</input>
</td>
</tr>
<tr>
<td>
<input type="submit" name="corpusconfig" value="Get configurations" />
</td>
</tr>
<tr>
</table>
</form>
```

Output

Here is sample corpus configuration output for the default set of configurations distributed with the MorphAdorner Server. In the JSON and XML output formats, the *CorpusConfigResult* contains a list of *CorpusConfigInfo* objects wrapped by a *corpusConfigs* container. Each *CorpusConfigInfo* object provides the *name* and *description* of a corpus configuration made available by the server. The HTML and text versions provide the same information in formats suitable for display.

JSON output

```
{
  "CorpusConfigResult": {
    "corpusConfigs": [
      {
        "CorpusConfigInfo": [
          {
            "name": "ece",
            "description": "Eighteenth Century English"
          },
          {
            "name": "eme",
            "description": "Early Modern English (~1475 to 1700)"
          },
          {
            "name": "ncf",
            "description": "Nineteeth Century British Fiction"
          }
        ]
      }
    ]
  }
}
```

XML output

```
<CorpusConfigResult>
    <corpusConfigs>
        <CorpusConfigInfo>
            <name>ece</name>
            <description>Eighteenth Century English</description>
        </CorpusConfigInfo>
        <CorpusConfigInfo>
            <name>eme</name>
            <description>Early Modern English (~1475 to 1700)</description>
        </CorpusConfigInfo>
        <CorpusConfigInfo>
            <name>ncf</name>
            <description>Nineteeth Century British Fiction</description>
        </CorpusConfigInfo>
    </corpusConfigs>
</CorpusConfigResult>
```

HTML output (source)

```
<h3>3 corpus configurations found.</h3>
<table border="0">
<tr>
<th align="left">Name</th>
<th align="left">Description</th>
</tr>
<tr>
<td valign="top" align="left"><strong>ece</strong></td>
<td valign="top" align="left">Eighteenth Century English</td>
</tr>
```

```
<tr>
<td valign="top" align="left"><strong>eme</strong></td>
<td valign="top" align="left">Early Modern English (~1475 to 1700)</td>
</tr>
<tr>
<td valign="top" align="left"><strong>ncf</strong></td>
<td valign="top" align="left">Nineteeth Century British Fiction</td>
</tr>
</table>
```

HTML output (display)

3 corpus configurations found.

Name	**Description**
ece	Eighteenth Century English
eme	Early Modern English (~1475 to 1700)
ncf	Nineteeth Century British Fiction

Text output

```
3 corpus configurations found.
Name    Description
ece     Eighteenth Century English
eme     Early Modern English (~1475 to 1700)
ncf     Nineteeth Century British Fiction
```

MorphAdorner Server Services: Gap Filler Service

Service name:	gapfiller
Service description:	Finds potential words matching a word containing missing characters.
HTTP methods allowed:	GET, POST, OPTIONS
POST accepts as input:	application/x-www-form-urlencoded
HTTP return codes:	200: service succeeded 400: service failed with an error
Query parameters	
corpusConfig	Corpus configuration name. In the standard distribution these are *ece*, *eme*, and *ncf*.
media	Result format. One of *json, xml, html, text* .
spelling	Spelling of a word.

Sample POST form

```
<form accept-charset="UTF-8" method="post" action="gapfiller"
      target="_blank"
      name="gapfiller">
<table cellpadding="0" cellspacing="5">
<tr>
<td><strong>Spelling:</strong></td>
<td><input type="text" name="spelling" size="20" value="" /></td>
</tr>
<tr>
<td valign="top">
<strong>
Lexicon:</strong>
</td>
<td>
<input type="radio" name="corpusConfig" value="eme">Early Modern English</input><br
/>
<input type="radio" name="corpusConfig" value="ece">Eighteen Century
English</input><br />
<input type="radio" name="corpusConfig" value="ncf" checked="checked">Nineteenth
Century Fiction</input>
</td>
</tr>
<tr>
<td>
&nbsp;
</td>
<td>
&nbsp;
</td>
</tr>
<tr>
```

```
<td valign="top">
<strong>Results format:</strong>
</td>
<td>
<input type="radio" name="media" value="json">JSON format</input><br />
<input type="radio" name="media" value="xml" checked="checked">XML
format</input><br />
<input type="radio" name="media" value="html">HTML format</input><br />
<input type="radio" name="media" value="text">Text format</input>
</td>
</tr>
<tr>
<td>
&nbsp;
</td>
<td>
&nbsp;
</td>
</tr>
<tr>
<td>
<input type="submit" name="fillgaps" value="Fill gaps" />
</td>
</tr>
</table>
</form>
```

Output

Here is sample output for the partial word "re?ate" where the "?" indicates an unknown character. We use the eme (Early Modern English) lexicon to locate potential matches. You may also use the Unicode black circle character \u25cf to specify a gap character. The output uses the black circle to display gap characters even when a "?" is used in the input word.

In the JSON and XML output formats, the *GapFillerResult* echoes the input *spelling* (with the "?" replaced by the Unicode black circle character \u25cf) and the *corpusConfig* name. The *suggestions* container wraps a list of *suggestion* entries, each of which is a single suggested gap-filled spelling. The HTML and text versions provide just these suggestions in a format suitable for display.

JSON output

```
{
  "GapFillerResult": {
    "spelling": "re●ate",
    "corpusConfig": "eme",
    "suggestions": [
      {
        "suggestion": [
          "rebate",
          "relate",
          "renate"
        ]
      }
    ]
  }
}
```

XML output

```
<GapFillerResult>
    <spelling>re●ate</spelling>
    <corpusConfig>eme</corpusConfig>
    <suggestions>
        <suggestion>rebate</suggestion>
        <suggestion>relate</suggestion>
        <suggestion>renate</suggestion>
    </suggestions>
</GapFillerResult>
```

HTML output (source)

```
<h3>3 suggestions found.</h3>
<table border="0">
<tr>
<td valign="top" align="left">rebate</td>
</tr>
<tr>
<td valign="top" align="left">relate</td>
</tr>
<tr>
<td valign="top" align="left">renate</td>
</tr>
</table>
```

HTML output (display)

3 suggestions found.

rebate
relate
renate

Text output

```
3 suggestions found.
rebate
relate
renate
```

MorphAdorner Server Services: Hyphenator Service

Service name:	hyphenator
Service description:	Hyphenate a spelling.
HTTP methods allowed:	GET, POST, OPTIONS
POST accepts as input:	application/x-www-form-urlencoded
HTTP return codes:	200: service succeeded 400: service failed with an error
Query parameters	
media	Result format. One of *json, xml, html, text* .
spelling	Spelling of a word.

Sample POST form

```
<form accept-charset="UTF-8" method="post" action="hyphenator"
      target="_blank"
      name="hyphenator">
<table cellpadding="0" cellspacing="5">
<tr>
<td><strong>Spelling:</strong></td>
<td><input type="text" name="spelling" size="20" value="" /></td>
</tr>
<tr>
<td>
&nbsp;
</td>
<td>
&nbsp;
</td>
</tr>
<tr>
<td valign="top">
<strong>Results format:</strong>
</td>
<td>
<input type="radio" name="media" value="json">JSON format</input><br />
<input type="radio" name="media" value="xml" checked="checked">XML
format</input><br />
<input type="radio" name="media" value="html">HTML format</input><br />
<input type="radio" name="media" value="text">Text format</input>
</td>
</tr>
<tr>
<td>
&nbsp;
</td>
```

```
<td>
&nbsp;
</td>
</tr>
<tr>
<td>
<input type="submit" name="hyphenate" value="Hyphenate" />
</td>
</tr>
</table>
</form>
```

Output

Here is sample hyphenated output for the spelling "coruscation." The JSON and XML formats echo the input *spelling* and provide the hyphenated version of the spelling as *hyphenatedSpelling*. The HTML and text versions provide the same information in a format suitable for display.

JSON output

```
{
  "HyphenatorResult": {
    "spelling": "coruscation",
    "hyphenatedSpelling": "co-rus-ca-tion"
  }
}
```

XML output

```
<HyphenatorResult>
    <spelling>coruscation</spelling>
    <hyphenatedSpelling>co-rus-ca-tion</hyphenatedSpelling>
</HyphenatorResult>
```

HTML output (source)

```
<h3>Hyphenation Results</h3>
<table border="0">
<tr>
<td valign="top" align="left"><strong>Spelling:</strong></td>
<td valign="top" align="left">coruscation</td>
</tr>
<tr>
<td valign="top" align="left"><strong>Hyphenated spelling:</strong></td>
<td valign="top" align="left">co-rus-ca-tion</td>
</tr>
</table>
```

HTML output (display)

Hyphenation Results

Spelling:	coruscation
Hyphenated spelling:	co-rus-ca-tion

Text output

```
Hyphenation Results
Spelling:       coruscation
Hyphenated spelling:    co-rus-ca-tion
```

MorphAdorner Server Services: Language Recognizer Service

Service name:	languagerecognizer
Service description:	Find probable languages for a text.
HTTP methods allowed:	GET, POST, OPTIONS
POST accepts as input:	application/x-www-form-urlencoded
HTTP return codes:	200: service succeeded 400: service failed with an error
Query parameters	
media	Result format. One of *json, xml, html, text* .
text	Text to be processed.
includeInputText	Allowed values are *true* to include the input text in the output and *false* to not include the input text.

Sample POST form

```
<form accept-charset="UTF-8" method="post" action="languagerecognizer"
      target="_blank"
      name="languagerecognizer">
<table cellpadding="0" cellspacing="5">
<tr>
<td><strong>Text:</strong></td>
<td colspan="2">
<textarea name="text" rows="15" cols="76"></textarea>
</td>
</tr>
<tr>
<td>&nbsp;</td>
<td>
<input type="checkbox" name="includeInputText" value="true"
       checked="checked"/>
Include input text in results
</td>
</tr>
<tr>
<td>
&nbsp;
</td>
<td>
&nbsp;
</td>
</tr>
<tr>
```

```
<td valign="top">
<strong>Results format:</strong>
</td>
<td>
<input type="radio" name="media" value="json">JSON format</input><br />
<input type="radio" name="media" value="xml" checked="checked">XML
format</input><br />
<input type="radio" name="media" value="html">HTML format</input><br />
<input type="radio" name="media" value="text">Text format</input>
</td>
</tr>
<tr>
<td>
&nbsp;
</td>
<td>
&nbsp;
</td>
</tr>
<tr>
<td colspan="2">
<input type="submit" name="recognize" value="Recognize" />
</td>
</tr>
</table>
</form>
```

Output

Here is sample language recognizer output for the traditional French song "Au claire de la lune." The JSON and XML *LanguageRecognizerResult* objects echo the input *text* and provide the most probably *languages* as a list of *language* entries with the ISO language code given as *languageCode*, the displayable language name as *languageName*, and the language probability score as *score*. The HTML and text versions do not echo the input text, and provide the languages and associated scores in a format suitable for display.

Au clair de la lune
Mon ami Pierrot
Prête-moi ta plume
Pour écrire un mot
Ma chandelle est morte
Je n'ai plus de feu
Ouvre-moi ta porte
Pour l'amour de Dieu.

JSON output

```
{
  "LanguageRecognizerResult": {
    "text": "Au clair de la lune\r\nMon ami Pierrot\r\nPr\u00eate-moi ta
plume\r\nPour \u00e9crire un mot\r\nMa chandelle est morte\r\nJe n'ai plus de
feu\r\nOuvre-moi ta porte\r\nPour l'amour de Dieu.",
    "languages": [
      {
        "language": {
          "languageCode": "fr",
          "languageName": "French",
          "score": 0.99999693379
        }
      }
    ]
  }
}
```

XML output

```
<?xml version="1.0"?>
<LanguageRecognizerResult>
    <text>Au clair de la lune
    Mon ami Pierrot
    Pr&#xEA;te-moi ta plume
    Pour &#xE9;crire un mot
    Ma chandelle est morte
    Je n'ai plus de feu
    Ouvre-moi ta porte
    Pour l'amour de Dieu.</text>
    <languages>
        <language>
            <languageCode>fr</languageCode>
            <languageName>French</languageName>
            <score>0.9999969337900039</score>
        </language>
    </languages>
</LanguageRecognizerResult>
```

HTML output (source)

```
<h3>1 language identified.</h3>
<table border="0">
<tr>
<th align="left">Language</th>
<th align="left">Score</th>
</tr>
<tr>
<td valign="top" align="left"><strong>fr (French)</strong></td>
<td valign="top" align="left">1.0000</td>
</tr>
</table>
```

HTML output (display)

1 language identified.

Language	Score
fr (French)	1.0000

Text output

```
1 language identified.
Language        Score
fr (French)     1.0000
```

MorphAdorner Server Services: Lemmatizer

Service name:	lemmatizer
Service description:	Find lemma form (dictionary headword) for a spelling.
HTTP methods allowed:	GET, POST, OPTIONS
POST accepts as input:	application/x-www-form-urlencoded
HTTP return codes:	200: service succeeded 400: service failed with an error
Query parameters	
corpusConfig	Corpus configuration name. In the standard distribution these are *ece*, *eme*, and *ncf*.
media	Result format. One of *json, xml, html, text* .
spelling	Spelling of a word.
standardize	Standardize (modernize) spelling before performing operation. Allowed values are *true* to request spelling standardization and *false* to disallow spelling standardization.
wordClass	Primary word class. One of *adjective, adverb, compound, conjunction, infinitive-to, noun, noun-possessive, preposition, pronoun, pronoun-possessive, pronoun-possessive-determiner, verb* .
wordClass2	Secondary word class. One of *adjective, adverb, compound, conjunction, infinitive-to, noun, noun-possessive, preposition, pronoun, pronoun-possessive, pronoun-possessive-determiner, verb* .

Sample POST form

```
<form accept-charset="UTF-8" method="post" action="lemmatizer"
      target="_blank"
      name="lemmatizer">
<table cellpadding="0" cellspacing="5">
<tr>
<td><strong>Spelling:</strong></td>
<td><input type="text" name="spelling" size="20" value="" /></td>
</tr>
<tr>
<td>
<td><input type="checkbox" name="standardize" value="true" checked="checked"
/>Standardize spelling</td>
</td>
<td>&nbsp;</td>
</tr>
<tr>
<td><strong>Primary word class:</strong></td>
<td>
<select name="wordClass">
<option value="" selected="selected"></option>
```

```
<option value="adjective">adjective</option>
<option value="adverb">adverb</option>
<option value="compound">compound</option>
<option value="conjunction">conjunction</option>
<option value="infinitive-to">infinitive-to</option>
<option value="noun">noun</option>
<option value="noun-possessive">noun-possessive</option>
<option value="preposition">preposition</option>
<option value="pronoun">pronoun</option>
<option value="pronoun-possessive">pronoun-possessive</option>
<option value="pronoun-possessive-determiner">pronoun-possessive-
determiner</option>
<option value="verb">verb</option>
</select>
</td>
</tr>
<tr>
<td><strong>Secondary word class:</strong></td>
<td>
<select name="wordClass2">
<option value="" selected="selected"></option>
<option value="adjective">adjective</option>
<option value="adverb">adverb</option>
<option value="compound">compound</option>
<option value="conjunction">conjunction</option>
<option value="infinitive-to">infinitive-to</option>
<option value="noun">noun</option>
<option value="noun-possessive">noun-possessive</option>
<option value="preposition">preposition</option>
<option value="pronoun">pronoun</option>
<option value="pronoun-possessive">pronoun-possessive</option>
<option value="pronoun-possessive-determiner">pronoun-possessive-
determiner</option>
<option value="verb">verb</option>
</select>
</td>
</tr>
<tr>
<td valign="top">
<strong>
Lexicon:</strong>
</td>
<td>
<input type="radio" name="corpusConfig" value="eme">Early Modern English</input><br
/>
<input type="radio" name="corpusConfig" value="ece">Eighteen Century
English</input><br />
<input type="radio" name="corpusConfig" value="ncf" checked="checked">Nineteenth
Century Fiction</input>
</td>
</tr>
<tr>
<td>
&nbsp;
</td>
<td>
&nbsp;
```

```
</td>
</tr>
<tr>
<td valign="top">
<strong>Results format:</strong>
</td>
<td>
<input type="radio" name="media" value="json">JSON format</input><br />
<input type="radio" name="media" value="xml" checked="checked">XML
format</input><br />
<input type="radio" name="media" value="html">HTML format</input><br />
<input type="radio" name="media" value="text">Text format</input>
</td>
</tr>
<tr>
<td>
&nbsp;
</td>
<td>
&nbsp;
</td>
</tr>
<tr>
<td>
<input type="submit" name="lemmatize" value="Lemmatize" />
</td>
</tr>
</table>
</form>
```

Output

Here is sample output for spelling *strykynge*, using the *eme* (early modern English) corpus configuration. We request spelling standardization and supply *verb* as the primary word class.

The JSON and XML formats echo the input *spelling*, the *corpusConfig*, the value of the *standardize* settings, and the primary and secondary word classes *wordClass* and *wordClass2* respectively. The resulting standard spelling is emitted as *standardSpelling* and the resulting lemma form appears as *lemma*. In addition, the Lancaster stemmer result appears in the *lancasterStem* output field, and the Porter stemmer result appears in the *porterStem* output field. The input query parameter field values are not emitted for the HTML or plain text output formats which are suitable for display.

JSON output

```
{
  "LemmatizerResult": {
    "spelling": "strykynge",
    "standardSpelling": "striking",
    "corpusConfig": "eme",
    "wordClass": "verb",
    "wordClass2": "",
    "lemma": "strike",
    "standardize": true,
    "lancasterStem": "strik",
    "porterStem": "strike"
  }
}
```

XML output

```
<LemmatizerResult>
   <spelling>strykynge</spelling>
   <standardSpelling>striking</standardSpelling>
   <corpusConfig>eme</corpusConfig>
   <wordClass>verb</wordClass>
   <wordClass2/>
   <lemma>strike</lemma>
   <standardize>true</standardize>
   <lancasterStem>strik</lancasterStem>
   <porterStem>strike</porterStem>
</LemmatizerResult>
```

HTML output (source)

```
<h3>Lemmatizer Results</h3>
<table border="0">
<tr>
<td valign="top" align="left"><strong>Lemma:</strong></td>
<td valign="top" align="left">strike</td>
</tr>
<tr>
<td valign="top" align="left"><strong>Lancaster stem:</strong></td>
<td valign="top" align="left">strik</td>
</tr>
<tr>
<td valign="top" align="left"><strong>Porter stem:</strong></td>
<td valign="top" align="left">strike</td>
</tr>
</table>
```

HTML output (display)

Lemmatizer Results

Lemma:	strike
Lancaster stem:	strik
Porter stem:	strike

Text output

```
Lemmatizer Results
Lemma:  strike
Lancaster stem: strik
Porter stem:    strike
```

MorphAdorner Server Services: Lexicon Lookup Service

Service name:	lexiconlookup
Service description:	Lookup spelling and related lemmata in a lexicon.
HTTP methods allowed:	GET, POST, OPTIONS
POST accepts as input:	application/x-www-form-urlencoded
HTTP return codes:	200: service succeeded 400: service failed with an error
Query parameters	
corpusConfig	Corpus configuration name. In the standard distribution these are *ece*, *eme*, and *ncf*.
media	Result format. One of *json, xml, html, text* .
spelling	Spelling of a word.

Sample POST form

```
<form accept-charset="UTF-8" method="post" action="lexiconlookup"
      target="_blank"
      name="lexiconlookup">
<table cellpadding="0" cellspacing="5">
<tr>
<td><strong>Spelling:</strong></td>
<td><input type="text" name="spelling" size="20" value="" /></td>
</tr>
<tr>
<td valign="top">
<strong>
Lexicon:</strong>
</td>
<td>
<input type="radio" name="corpusConfig" value="eme">Early Modern English</input><br
/>
<input type="radio" name="corpusConfig" value="ece">Eighteen Century
English</input><br />
<input type="radio" name="corpusConfig" value="ncf" checked="checked">Nineteenth
Century Fiction</input>
</td>
</tr>
<tr>
<td>
&nbsp;
</td>
<td>
&nbsp;
</td>
</tr>
<tr>
```

```
<td valign="top">
<strong>Results format:</strong>
</td>
<td>
<input type="radio" name="media" value="json">JSON format</input><br />
<input type="radio" name="media" value="xml" checked="checked">XML
format</input><br />
<input type="radio" name="media" value="html">HTML format</input><br />
<input type="radio" name="media" value="text">Text format</input>
</td>
</tr>
<tr>
<td>
<input type="submit" name="lookup" value="Lookup" />
</td>
</tr>
</table>
</form>
```

Output

Here is sample output for the spelling "love" in the early modern English corpus (*eme*). The JSON and XML *LexiconLookupResult* echoes the input *spelling* and *corpusConfig* values as well as a *LexiconEntry* which provides the different part of speech counts for the *lemmata* associated with the spelling "love". The output also displays other spellings in the lexicon which take the same lemma forms as "love" as a list of *relatedSpellings*. The HTML and text versions provide the results in formats suitable for display.

JSON output

```
{
  "LexiconLookupResult": {
    "spelling": "love",
    "corpusConfig": "eme",
    "lexiconEntry": {
      "entry": "love",
      "standardEntry": "love",
      "lemmata": [
        {
          "entry": [
            {
              "string": [
                "n1",
                "love"
              ]
            },
            {
              "string": [
                "vvb",
                "love"
              ]
            },
            {
              "string": [
                "np1-n",
              "love"
            ]
          },
          {
            "string": [
              "vvi",
              "love"
            ]
          }
        ]
      }
    ],
    "entryCount": 24180,
    "categoriesAndCounts": [
      {
        "entry": [
          {
            "string": "n1",
            "MutableInteger": {
              "mutableInteger": 16227
            }
          },
          {
            "string": "vvb",
            "MutableInteger": {
              "mutableInteger": 4469
            }
          },
          {
            "string": "np1-n",
            "MutableInteger": {
              "mutableInteger": 5
            }
          },
          {
            "string": "vvi",
            "MutableInteger": {
              "mutableInteger": 3479
            }
          }
        ]
      }
    ],
    "largestCategory": "n1",
    "largestCategoryCount": 16227
  },
  "relatedSpellings": [
    {
      "relatedSpelling": [
        "vnlou'd",
        "lou'd",
        "lous",
        "louingest",
        "love's",
        "louingly",
        "lovingest",
        "lou's",
        "loveth",
        "lovd",
        "Loved",
        "lov'd",
        "{love}",
        "Lou'dst",
        "lub",
        "loues",
        "lov'dst",
        "Loued",
        "louingle",
        "loves",
        "loou'st",
        "lov'de",
        "Loves",
        "loueth",
        "lovingly",
        "lovest",
        "Loving",
        "Lov's",
        "Louing",
        "lovedst",
        "Lov'd",
        "unloved",
        "loving",
        "Lou's",
        "Loues",
        "loue",
        "lovedest",
        "Lou'd",
        "louing",
        "louiug",
        "lovingst",
        "Love",
        "Lov'st",
        "louedst",
        "lovinglie",
        "lcue",
        "louest",
        "lovst",
        "loust",
        "louinglie",
        "LOVES",
        "Loue",
        "lovinge",
        "lo'd",
        "lou`d",
        "louyng",
        "love-a",
        "loued",
        "LOVE",
        "lou'st",
        "loue'd",
        "louinge",
        "lovings",
        "lou'dst",
        "loved",
          "louynge",
          "louen",
          "lou'de",
          "louingst",
          "lovesto",
          "lovea",
          "LOue",
          "loy'st",
          "lov'st"
        ]
      }
    ]
  }
}
```

XML output

```
<LexiconLookupResult>
    <spelling>love</spelling>
    <corpusConfig>eme</corpusConfig>
    <lexiconEntry>
        <entry>love</entry>
        <standardEntry>love</standardEntry>
        <lemmata>
            <entry>
                <string>n1</string>
                <string>love</string>
            </entry>
            <entry>
                <string>vvb</string>
                <string>love</string>
            </entry>
            <entry>
                <string>np1-n</string>
                <string>love</string>
            </entry>
            <entry>
                <string>vvi</string>
                <string>love</string>
            </entry>
        </lemmata>
        <entryCount>24180</entryCount>
        <categoriesAndCounts>
            <entry>
                <string>n1</string>
                <MutableInteger>
                    <mutableInteger>16227</mutableInteger>
                </MutableInteger>
            </entry>
            <entry>
                <string>vvb</string>
                <MutableInteger>
                    <mutableInteger>4469</mutableInteger>
                </MutableInteger>
            </entry>
            <entry>
            <string>np1-n</string>
            <MutableInteger>
                <mutableInteger>5</mutableInteger>
            </MutableInteger>
        </entry>
        <entry>
            <string>vvi</string>
            <MutableInteger>
                <mutableInteger>3479</mutableInteger>
            </MutableInteger>
        </entry>
    </categoriesAndCounts>
    <largestCategory>n1</largestCategory>
    <largestCategoryCount>16227</largestCategoryCount>
</lexiconEntry>
<relatedSpellings>
    <relatedSpelling>vnlou'd</relatedSpelling>
    <relatedSpelling>lou'd</relatedSpelling>
    <relatedSpelling>lous</relatedSpelling>
    <relatedSpelling>louingest</relatedSpelling>
    <relatedSpelling>love's</relatedSpelling>
    <relatedSpelling>louingly</relatedSpelling>
    <relatedSpelling>lovingest</relatedSpelling>
    <relatedSpelling>lou's</relatedSpelling>
    <relatedSpelling>loveth</relatedSpelling>
    <relatedSpelling>lovd</relatedSpelling>
    <relatedSpelling>Loved</relatedSpelling>
    <relatedSpelling>lov'd</relatedSpelling>
    <relatedSpelling>{love}</relatedSpelling>
    <relatedSpelling>Lou'dst</relatedSpelling>
    <relatedSpelling>lub</relatedSpelling>
    <relatedSpelling>loues</relatedSpelling>
    <relatedSpelling>lov'dst</relatedSpelling>
    <relatedSpelling>Loued</relatedSpelling>
    <relatedSpelling>louingle</relatedSpelling>
    <relatedSpelling>loves</relatedSpelling>
    <relatedSpelling>loou'st</relatedSpelling>
    <relatedSpelling>lov'de</relatedSpelling>
    <relatedSpelling>Loves</relatedSpelling>
    <relatedSpelling>loueth</relatedSpelling>
    <relatedSpelling>lovingly</relatedSpelling>
    <relatedSpelling>lovest</relatedSpelling>
    <relatedSpelling>Loving</relatedSpelling>
    <relatedSpelling>Lov's</relatedSpelling>
    <relatedSpelling>Louing</relatedSpelling>
    <relatedSpelling>lovedst</relatedSpelling>
    <relatedSpelling>Lov'd</relatedSpelling>
    <relatedSpelling>unloved</relatedSpelling>
    <relatedSpelling>loving</relatedSpelling>
    <relatedSpelling>Lou's</relatedSpelling>
    <relatedSpelling>Loues</relatedSpelling>
    <relatedSpelling>loue</relatedSpelling>
    <relatedSpelling>lovedest</relatedSpelling>
    <relatedSpelling>Lou'd</relatedSpelling>
    <relatedSpelling>louing</relatedSpelling>
    <relatedSpelling>louiug</relatedSpelling>
    <relatedSpelling>lovingst</relatedSpelling>
```

```
        <relatedSpelling>Love</relatedSpelling>
        <relatedSpelling>Lov'st</relatedSpelling>
        <relatedSpelling>louedst</relatedSpelling>
        <relatedSpelling>lovinglie</relatedSpelling>
        <relatedSpelling>lcue</relatedSpelling>
        <relatedSpelling>louest</relatedSpelling>
        <relatedSpelling>lovst</relatedSpelling>
        <relatedSpelling>loust</relatedSpelling>
        <relatedSpelling>louinglie</relatedSpelling>
        <relatedSpelling>LOVES</relatedSpelling>
        <relatedSpelling>Loue</relatedSpelling>
        <relatedSpelling>lovinge</relatedSpelling>
        <relatedSpelling>lo'd</relatedSpelling>
        <relatedSpelling>lou`d</relatedSpelling>
        <relatedSpelling>louyng</relatedSpelling>
        <relatedSpelling>love-a</relatedSpelling>
        <relatedSpelling>loued</relatedSpelling>
        <relatedSpelling>LOVE</relatedSpelling>
        <relatedSpelling>lou'st</relatedSpelling>
        <relatedSpelling>loue'd</relatedSpelling>
        <relatedSpelling>louinge</relatedSpelling>
        <relatedSpelling>lovings</relatedSpelling>
        <relatedSpelling>lou'dst</relatedSpelling>
        <relatedSpelling>loved</relatedSpelling>
        <relatedSpelling>louynge</relatedSpelling>
        <relatedSpelling>louen</relatedSpelling>
        <relatedSpelling>lou'de</relatedSpelling>
        <relatedSpelling>louingst</relatedSpelling>
        <relatedSpelling>lovesto</relatedSpelling>
        <relatedSpelling>lovea</relatedSpelling>
        <relatedSpelling>LOue</relatedSpelling>
        <relatedSpelling>loy'st</relatedSpelling>
        <relatedSpelling>lov'st</relatedSpelling>
    </relatedSpellings>
</LexiconLookupResult>
```

HTML output (source)

```
<h3>love appears 24,180 times in the eme corpus training data.</h3>
<table border="0">
<tr>
<th align="left">Part of Speech</th>
<th align="left">Lemma</th>
<th align="left">Count</th>
</tr>
<tr>
<td valign="top" align="left">n1</td>
<td valign="top" align="left">love</td>
<td valign="top" align="left">16,227</td>
</tr>
<tr>
<td valign="top" align="left">vvb</td>
<td valign="top" align="left">love</td>
<td valign="top" align="left">4,469</td>
</tr>
<tr>
```

```
<td valign="top" align="left">np1-n</td>
<td valign="top" align="left">love</td>
<td valign="top" align="left">5</td>
</tr>
<tr>
<td valign="top" align="left">vvi</td>
<td valign="top" align="left">love</td>
<td valign="top" align="left">3,479</td>
</tr>
</table>
<table border="0">
<tr>
<th align="left">Related spellings:</th>
</tr>
<tr>
<td valign="top" align="left">vnlou'd, lou'd, lous, louingest, love's, louingly,
lovingest, lou's, loveth, lovd, Loved, lov'd, {love}, Lou'dst, lub, loues, lov'dst,
Loued, louingle, loves, loou'st, lov'de, Loves, loueth, lovingly, lovest, Loving,
Lov's, Louing, lovedst, Lov'd, unloved, loving, Lou's, Loues, loue, lovedest,
Lou'd, louing, louiug, lovingst, Love, Lov'st, louedst, lovinglie, lcue, louest,
lovst, loust, louinglie, LOVES, Loue, lovinge, lo'd, lou`d, louyng, love-a, loued,
LOVE, lou'st, loue'd, louinge, lovings, lou'dst, loved, louynge, louen, lou'de,
louingst, lovesto, lovea, LOue, loy'st, lov'st</td>
</tr>
</table>
```

HTML output (display)

love appears 24,180 times in the eme corpus training data.

Part of Speech	Lemma	Count
n1	love	16,227
vvb	love	4,469
np1-n	love	5
vvi	love	3,479

Related spellings:

vnlou'd, lou'd, lous, louingest, love's, louingly, lovingest, lou's, loveth, lovd, Loved, lov'd, {love}, Lou'dst, lub, loues, lov'dst, Loued, louingle, loves, loou'st, lov'de, Loves, loueth, lovingly, lovest, Loving, Lov's, Louing, lovedst, Lov'd, unloved, loving, Lou's, Loues, loue, lovedest, Lou'd, louing, louiug, lovingst, Love, Lov'st, louedst, lovinglie, lcue, louest, lovst, loust, louinglie, LOVES, Loue, lovinge, lo'd, lou`d, louyng, love-a, loued, LOVE, lou'st, loue'd, louinge, lovings, lou'dst, loved, louynge, louen, lou'de, louingst, lovesto, lovea, LOue, loy'st, lov'st

Text output

```
love appears 24,180 times in the eme corpus training data.
Part of Speech  Lemma   Count
n1      love    16,227
vvb     love    4,469
np1-n   love    5
vvi     love    3,479
Related spellings:
vnlou'd, lou'd, lous, louingest, love's, louingly, lovingest,
lou's, loveth, lovd, Loved, lov'd, {love}, Lou'dst, lub, loues,
lov'dst, Loued, louingle, loves, loou'st, lov'de, Loves, loueth,
lovingly, lovest, Loving, Lov's, Louing, lovedst, Lov'd,
unloved, loving, Lou's, Loues, loue, lovedest, Lou'd, louing,
louiug, lovingst, Love, Lov'st, louedst, lovinglie, lcue,
louest, lovst, loust, louinglie, LOVES, Loue, lovinge, lo'd,
lou`d, louyng, love-a, loued, LOVE, lou'st, loue'd, louinge,
lovings, lou'dst, loved, louynge, louen, lou'de, louingst,
lovesto, lovea, LOue, loy'st, lov'st
```

MorphAdorner Server Services: Name Recognizer Service

Service name:	namerecognizer
Service description:	Recognize names and places in text.
HTTP methods allowed:	GET, POST, OPTIONS
POST accepts as input:	application/x-www-form-urlencoded
HTTP return codes:	200: service succeeded 400: service failed with an error
Query parameters	
corpusConfig	Corpus configuration name. In the standard distribution these are *ece*, *eme*, and *ncf*.
includeInputText	Allowed values are *true* to include the input text in the output and *false* to not include the input text.
media	Result format. One of *json, xml, html, text* .
text	Text to be processed.

Sample POST form

```
<form accept-charset="UTF-8" method="post" action="namerecognizer"
      target="_blank"
      name="namerecognizer">
<table cellpadding="0" cellspacing="5">
<tr>
<td><strong>Text:</strong></td>
<td colspan="2">
<textarea name="text" rows="15" cols="76"></textarea>
</td>
</tr>
<tr>
<td valign="top">
<strong>
Lexicon:</strong>
</td>
<td>
<input type="radio" name="corpusConfig" value="eme">Early Modern English</input><br
/>
<input type="radio" name="corpusConfig" value="ece">Eighteen Century
English</input><br />
<input type="radio" name="corpusConfig" value="ncf" checked="checked">Nineteenth
Century Fiction</input>
</td>
</tr>
<tr>
<td>&nbsp;</td>
<td>
<input type="checkbox" name="includeInputText" value="true"
```

```
       checked="checked"/>
Include input text in results
</td>
</tr>
<tr>
<td>
&nbsp;
</td>
<td>
&nbsp;
</td>
</tr>
<tr>
<td valign="top">
<strong>Results format:</strong>
</td>
<td>
<input type="radio" name="media" value="json">JSON format</input><br />
<input type="radio" name="media" value="xml" checked="checked">XML
format</input><br />
<input type="radio" name="media" value="html">HTML format</input><br />
<input type="radio" name="media" value="text">Text format</input>
</td>
</tr>
<tr>
<td>
&nbsp;
</td>
<td>
&nbsp;
</td>
</tr>
<tr>
<td colspan="2">
<input type="submit" name="getnames" value="Get names" />
</td>
</tr>
</table>
</form>
```

Output

We look for names in the following short section of text taken from a Northwestern University web page discussing the early history of the University.

> In 1853, the founders purchased a 379-acre tract of land on the shore of Lake Michigan 12 miles north of Chicago. They established a campus and developed the land near it, naming the surrounding town Evanston in honor of one of the University's founders, John Evans. After completing its first building in 1855, Northwestern began classes that fall with two faculty members and 10 students.

The JSON and XML NameRecognizerResult echoes the input text. The HTML and text versions provide displayable versions of the name and place lists. Note that the implementation is rather primitive and frequently fails to distinguish names from places.

JSON output

```
{
  "NameRecognizerResult": {
    "text": "In 1853, the founders purchased a 379-acre tract of land on the  shore
of Lake Michigan 12 miles north of Chicago. They  established a campus and
developed the land near it, naming the  surrounding town Evanston in honor of one
of the University's  founders, John Evans. After completing its first building in
1855, Northwestern began classes that fall with two faculty  members and 10
students.",
    "corpusConfig": "ncf",
    "personNames": [
      {
        "@class": "tree-set",
        "personName": [
          "John Evans",
          "Lake Michigan",
          "Northwestern"
        ]
      }
    ],
    "placeNames": [
      {
        "@class": "tree-set",
        "placeName": [
          "Chicago",
          "Evanston"
        ]
      }
    ]
  }
}
```

XML output

```
<NameRecognizerResult>
    <text>In 1853, the founders purchased a 379-acre tract of land on the shore of
Lake Michigan 12 miles north of Chicago. They established a campus and developed
the land near it, naming the surrounding town Evanston in honor of one of the
University's founders, John Evans. After completing its first building in 1855,
Northwestern began classes that fall with two faculty members and 10
students.</text>
    <corpusConfig>ncf</corpusConfig>
    <personNames class="tree-set">
        <personName>John Evans</personName>
        <personName>Lake Michigan</personName>
        <personName>Northwestern</personName>
    </personNames>
    <placeNames class="tree-set">
        <placeName>Chicago</placeName>
        <placeName>Evanston</placeName>
    </placeNames>
</NameRecognizerResult>
```

HTML output (source)

```
<h3>
3 person names found.
</h3>
<table border="0">
<tr><td>John Evans</td></tr>
<tr><td>Lake Michigan</td></tr>
<tr><td>Northwestern</td></tr>
</table>
<h3>
2 place names found.
</h3>
<table border="0">
<tr><td>Chicago</td></tr>
<tr><td>Evanston</td></tr>
</table>
```

HTML output (display)

3 person names found.

John Evans

Lake Michigan

Northwestern

2 place names found.

Chicago

Evanston

Text output

```
3 person names found.
John Evans
Lake Michigan
Northwestern
2 place names found.
Chicago
Evanston
```

MorphAdorner Server Services: Noun Pluralizer Service

Service name:	pluralizer
Service description:	Find plural forms of nouns and pronouns.
HTTP methods allowed:	GET, POST, OPTIONS
POST accepts as input:	application/x-www-form-urlencoded
HTTP return codes:	200: service succeeded 400: service failed with an error
Query parameters	
american	Display American (U.S.) spellings of plural forms. Allowed values are *true* to display American spellings, *false* for British spellings.
media	Result format. One of *json, xml, html, text* .
spelling	Spelling of a word.

Sample POST form

The pluralizer service accepts a singular noun or pronoun and returns the plural form. An option allows for returning American plural forms instead of British plurals.

```
<form accept-charset="UTF-8" method="post" action="pluralizer"
      target="_blank"
      name="pluralizer">
<table cellpadding="0" cellspacing="5">
<tr>
<td><strong>Singular Noun:</strong></td>
<td><input type="text" name="singular" size = "20" value="" /></td>
</tr>
<tr>
<td>&nbsp;</td>
<td>
<input type="checkbox" name="american" value="true" />
American spellings
</td>
</tr>
<tr>
<td>
&nbsp;
</td>
<td>
&nbsp;
</td>
</tr>
<tr>
<td valign="top">
<strong>Results format:</strong>
</td>
<td>
```

```
<input type="radio" name="media" value="json">JSON format</input><br />
<input type="radio" name="media" value="xml" checked="checked">XML
format</input><br />
<input type="radio" name="media" value="html">HTML format</input><br />
<input type="radio" name="media" value="text">Text format</input>
</td>
</tr>
<tr>
<td>
<input type="submit" name="pluralize" value="Pluralize" />
</td>
</tr>
</table>
</form>
```

Output

Here we use the pluralizer service to find the plural of the noun "mouse."

JSON output

```
{
  "PluralizerResult": {
    "singular": "mouse",
    "plural": "mice",
    "american": false
  }
}
```

XML output

```
<PluralizerResult>
    <singular>mouse</singular>
    <plural>mice</plural>
    <american>false</american>
</PluralizerResult>
```

HTML output (source)

```
<h3>Pluralizer results</h3>
<table border="0">
<tbody><tr>
<td align="left" valign="top"><strong>Singular:</strong></td>
<td align="left" valign="top">mouse</td>
</tr>
<tr>
<td align="left" valign="top"><strong>Plural:</strong></td>
<td align="left" valign="top">mice</td>
</tr>
<tr>
<td align="left" valign="top"><strong>American:</strong></td>
<td align="left" valign="top">false</td>
</tr>
</tbody>
</table>
```

HTML output (display)

Pluralizer results

Singular:	mouse
Plural:	mice
American:	false

Text output

```
Pluralizer results
Singular:       mouse
Plural: mice
American:       false
```

MorphAdorner Server Services: Parser Service

Service name:	parser
Service description:	Parse sentences using Link Grammar Parser.
HTTP methods allowed:	GET, POST, OPTIONS
POST accepts as input:	application/x-www-form-urlencoded
HTTP return codes:	200: service succeeded 400: service failed with an error
Query parameters	
media	Result format. One of *json, xml, html, text* .
text	Text to be processed.
includeInputText	Allowed values are *true* to include the input text in the output and *false* to not include the input text.

Sample POST form

```
<form accept-charset="UTF-8" method="post" action="parser"
      target="_blank"
      name="parser">
<table cellpadding="0" cellspacing="5">
<tr>
<td><strong>Text:</strong></td>
<td colspan="2">
<textarea name="text" rows="15" cols="76"></textarea>
</td>
</tr>
<tr>
<td>&nbsp;</td>
<td>
<input type="checkbox" name="includeInputText" value="true"
       checked="checked"/>
Include input text in results
</td>
</tr>
<tr>
<td>
&nbsp;
</td>
<td>
&nbsp;
</td>
</tr>
<tr>
<td valign="top">
<strong>Results format:</strong>
</td>
<td>
```

```
<input type="radio" name="media" value="json">JSON format</input><br />
<input type="radio" name="media" value="xml" checked="checked">XML
format</input><br />
<input type="radio" name="media" value="html">HTML format</input><br />
<input type="radio" name="media" value="text">Text format</input>
</td>
</tr>
<tr>
<td>
&nbsp;
</td>
<td>
&nbsp;
</td>
</tr>
<tr>
<td colspan="2">
<input type="submit" name="parse" value="Parse" />
</td>
</tr>
</table>
</form>
```

Output

Here is sample parser output for the sentence "Mary had a little lamb."

JSON output

```
{
  "ParserResult": {
    "text": "Mary had a little lamb.",
    "parsedText": "\n    +---------------Xp---------------+\n    |
+-------Os------+   |\n    +---Wd--+-Ss-+   +IDDC+--Dmu-+   |\n    |       |    |
|    |      |   |\nLEFT-WALL Mary had.v a little lamb.n . \n\n\n     LEFT-WALL  Xp
<---Xp---->  Xp    .           \n(m)  LEFT-WALL  Wd    <---Wd---->  Wd    Mary
\n(m)  Mary       Ss    <---Ss---->  S     had.v       \n(m)  had.v      O     <---
Os---->  Os    lamb.n      \n(m)  little     Dmu   <---Dmu--->  D*u
lamb.n      \n(m)  a          IDDC  <---IDDC-->  IDDC  little      \n     .
RW    <---RW---->  RW    RIGHT-WALL  \n"
  }
}
```

XML output

```
<ParserResult>
<text>Mary had a little lamb.</text>
<parsedText>
    +---------------Xp---------------+
    |            +-------Os------+   |
    +---Wd--+-Ss-+   +IDDC+--Dmu-+   |
    |       |    |   |    |      |   |
LEFT-WALL Mary had.v a little lamb.n .
     LEFT-WALL  Xp    <---Xp---->  Xp    .
(m)  LEFT-WALL  Wd    <---Wd---->  Wd    Mary
(m)  Mary       Ss    <---Ss---->  S     had.v
```

```
(m)  had.v      O     <---Os---->  Os    lamb.n
(m)  little     Dmu   <---Dmu--->  D*u   lamb.n
(m)  a          IDDC  <---IDDC-->  IDDC  little
     .          RW    <---RW---->  RW    RIGHT-WALL
</parsedText>
</ParserResult>
```

HTML output (source)

```
<p>
<pre>
    +---------------Xp---------------+
    |            +-------Os------+   |
    +---Wd--+-Ss-+   +IDDC+--Dmu-+   |
    |       |    |   |    |      |   |
LEFT-WALL Mary had.v a little lamb.n .
     LEFT-WALL  Xp    <---Xp---->  Xp    .
(m)  LEFT-WALL  Wd    <---Wd---->  Wd    Mary
(m)  Mary       Ss    <---Ss---->  S     had.v
(m)  had.v      O     <---Os---->  Os    lamb.n
(m)  little     Dmu   <---Dmu--->  D*u   lamb.n
(m)  a          IDDC  <---IDDC-->  IDDC  little
     .          RW    <---RW---->  RW    RIGHT-WALL
</pre>
</p>
```

HTML output (display)

```
    +---------------Xp---------------+
    |            +-------Os------+   |
    +---Wd--+-Ss-+   +IDDC+--Dmu-+   |
    |       |    |   |    |      |   |
LEFT-WALL Mary had.v a little lamb.n .
     LEFT-WALL  Xp    <---Xp---->  Xp    .
(m)  LEFT-WALL  Wd    <---Wd---->  Wd    Mary
(m)  Mary       Ss    <---Ss---->  S     had.v
(m)  had.v      O     <---Os---->  Os    lamb.n
(m)  little     Dmu   <---Dmu--->  D*u   lamb.n
(m)  a          IDDC  <---IDDC-->  IDDC  little
     .          RW    <---RW---->  RW    RIGHT-WALL
```

Text output

```
    +---------------Xp---------------+
    |            +-------Os------+   |
    +---Wd--+-Ss-+   +IDDC+--Dmu-+   |
    |       |    |   |    |      |   |
LEFT-WALL Mary had.v a little lamb.n .
     LEFT-WALL  Xp    <---Xp---->  Xp    .
(m)  LEFT-WALL  Wd    <---Wd---->  Wd    Mary
(m)  Mary       Ss    <---Ss---->  S     had.v
(m)  had.v      O     <---Os---->  Os    lamb.n
(m)  little     Dmu   <---Dmu--->  D*u   lamb.n
(m)  a          IDDC  <---IDDC-->  IDDC  little
     .          RW    <---RW---->  RW    RIGHT-WALL
```

MorphAdorner Server Services: Sentence Splitter Service

Service name:	sentencesplitter
Service description:	Splits plain text into sentences.
HTTP methods allowed:	GET, POST, OPTIONS
POST accepts as input:	application/x-www-form-urlencoded
HTTP return codes:	200: service succeeded 400: service failed with an error
Query parameters	
corpusConfig	Corpus configuration name. In the standard distribution these are *ece*, *eme*, and *ncf*.
media	Result format. One of *json, xml, html, text* .
text	Text to be processed.
includeInputText	Allowed values are *true* to include the input text in the output and *false* to not include the input text.
langCode	ISO language code. These are two or three character codes. The default is *en*, English. You may specify *** *Detect* *** to indicate that the server should try to determine the language from the text provided.

Sample POST form

```
<form accept-charset="UTF-8" method="post" action="sentencesplitter"
      target="_blank"
      name="sentencesplitter">
<table cellpadding="0" cellspacing="5">
<tr>
<td><strong>Text:</strong></td>
<td colspan="2">
<textarea name="text" rows="15" cols="76"></textarea>
</td>
</tr>
<tr>
<td valign="top">
<strong>
Lexicon:</strong>
</td>
<td>
<input type="radio" name="corpusConfig" value="eme">Early Modern English</input><br
/>
<input type="radio" name="corpusConfig" value="ece">Eighteen Century
English</input><br />
<input type="radio" name="corpusConfig" value="ncf" checked="checked">Nineteenth
Century Fiction</input>
</td>
</tr>
<tr>
```

```
<td><strong>Language:</strong></td>
<td>
<select name="langCode">
<option value="en" selected="selected">English</option>
<option value="">*** Detect ***</option>
<option value="af">Afrikaans</option>
<option value="ak">Akan</option>
<option value="sq">Albanian</option>
<option value="am">Amharic</option>
<option value="ar">Arabic</option>
<option value="hy">Armenian</option>
<option value="as">Assamese</option>
<option value="az">Azerbaijani</option>
<option value="bm">Bambara</option>
<option value="bas">Basa</option>
<option value="eu">Basque</option>
<option value="be">Belarusian</option>
<option value="bem">Bemba</option>
<option value="bn">Bengali</option>
<option value="bs">Bosnian</option>
<option value="br">Breton</option>
<option value="bg">Bulgarian</option>
<option value="my">Burmese</option>
<option value="ca">Catalan</option>
<option value="chr">Cherokee</option>
<option value="zh">Chinese</option>
<option value="kw">Cornish</option>
<option value="hr">Croatian</option>
<option value="cs">Czech</option>
<option value="da">Danish</option>
<option value="dua">Duala</option>
<option value="nl">Dutch</option>
<option value="eo">Esperanto</option>
<option value="et">Estonian</option>
<option value="ee">Ewe</option>
<option value="ewo">Ewondo</option>
<option value="fo">Faroese</option>
<option value="fil">Filipino</option>
<option value="fi">Finnish</option>
<option value="fr">French</option>
<option value="ff">Fulah</option>
<option value="gl">Gallegan</option>
<option value="lg">Ganda</option>
<option value="ka">Georgian</option>
<option value="de">German</option>
<option value="el">Greek</option>
<option value="kl">Greenlandic</option>
<option value="gu">Gujarati</option>
<option value="ha">Hausa</option>
<option value="haw">Hawaiian</option>
<option value="iw">Hebrew</option>
<option value="hi">Hindi</option>
<option value="hu">Hungarian</option>
<option value="is">Icelandic</option>
<option value="ig">Igbo</option>
<option value="in">Indonesian</option>
<option value="ga">Irish</option>
```

```
<option value="it">Italian</option>
<option value="ja">Japanese</option>
<option value="kab">Kabyle</option>
<option value="kam">Kamba</option>
<option value="kn">Kannada</option>
<option value="kk">Kazakh</option>
<option value="km">Khmer</option>
<option value="ki">Kikuyu</option>
<option value="rw">Kinyarwanda</option>
<option value="kok">Konkani</option>
<option value="ko">Korean</option>
<option value="lv">Latvian</option>
<option value="ln">Lingala</option>
<option value="lt">Lithuanian</option>
<option value="lu">Luba-Katanga</option>
<option value="mk">Macedonian</option>
<option value="mg">Malagasy</option>
<option value="ms">Malay</option>
<option value="ml">Malayalam</option>
<option value="mt">Maltese</option>
<option value="gv">Manx</option>
<option value="mr">Marathi</option>
<option value="mas">Masai</option>
<option value="ne">Nepali</option>
<option value="nd">North Ndebele</option>
<option value="nb">Norwegian Bokm?l</option>
<option value="nn">Norwegian Nynorsk</option>
<option value="nyn">Nyankole</option>
<option value="or">Oriya</option>
<option value="om">Oromo</option>
<option value="pa">Panjabi</option>
<option value="fa">Persian</option>
<option value="pl">Polish</option>
<option value="pt">Portuguese</option>
<option value="ps">Pushto</option>
<option value="rm">Raeto-Romance</option>
<option value="ro">Romanian</option>
<option value="rn">Rundi</option>
<option value="ru">Russian</option>
<option value="sg">Sango</option>
<option value="sr">Serbian</option>
<option value="sn">Shona</option>
<option value="ii">Sichuan Yi</option>
<option value="si">Sinhalese</option>
<option value="sk">Slovak</option>
<option value="sl">Slovenian</option>
<option value="so">Somali</option>
<option value="es">Spanish</option>
<option value="sw">Swahili</option>
<option value="sv">Swedish</option>
<option value="gsw">Swiss German</option>
<option value="ta">Tamil</option>
<option value="te">Telugu</option>
<option value="th">Thai</option>
<option value="bo">Tibetan</option>
<option value="ti">Tigrinya</option>
<option value="to">Tonga</option>
```

```
<option value="tr">Turkish</option>
<option value="uk">Ukrainian</option>
<option value="ur">Urdu</option>
<option value="uz">Uzbek</option>
<option value="vai">Vai</option>
<option value="vi">Vietnamese</option>
<option value="cy">Welsh</option>
<option value="yo">Yoruba</option>
<option value="zu">Zulu</option>
</select>
</td>
</tr>
<tr>
<td>&nbsp;</td>
<td>
<input type="checkbox" name="includeInputText" value="true"
       checked="checked"/>
Include input text in results
</td>
</tr>
<tr>
<td>
&nbsp;
</td>
<td>
&nbsp;
</td>
</tr>
<tr>
<td valign="top">
<strong>Results format:</strong>
</td>
<td>
<input type="radio" name="media" value="json">JSON format</input><br />
<input type="radio" name="media" value="xml" checked="checked">XML
format</input><br />
<input type="radio" name="media" value="html">HTML format</input><br />
<input type="radio" name="media" value="text">Text format</input>
</td>
</tr>
<tr>
<td>
&nbsp;
</td>
<td>
&nbsp;
</td>
</tr>
<tr>
<td colspan="2">
<input type="submit" name="split" value="Split" />
</td>
</tr>
</table>
</form>
```

Output

Here we split a paragraph from Lincolns "Gettysburg Address" into sentences.

> 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.

The JSON and XML *WordTokenizerResult* echo the input *text*, the ISO language code *langCode*, and the *corpusConfig*. The *sentences* container wraps a sequence of *sentence* entries each of which represents a single parsed sentence from the input text. Each *sentence* contains a sequence of *token* entries representing the words and punctuation in the sentence. The *meldedSentences* container wraps a sequence of *meldedSentence* entries each of which contains a single untokenized sentence. The HTML and text versions provide displayable versions of the extracted sentences.

JSON output

```
{
  "SentenceSplitterResult": {
    "text": "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.",
    "langCode": "en",
    "corpusConfig": "ncf",
    "sentences": [
      {
        "sentence": [
          {
            "token": [
              "Now",
              "we",
              "are",
              "engaged",
              "in",
              "a",
              "great",
              "civil",
              "war",
              ",",
              "testing",
              "whether",
              "that",
              "nation",
              ",",
              "or",
              "any",
              "nation",
              ",",
              "so",
              "conceived",
              "and",
          "so",
          "dedicated",
          ",",
          "can",
          "long",
          "endure",
          "."
        ]
      },
      {
        "token": [
          "We",
          "are",
          "met",
          "on",
          "a",
          "great",
          "battle-field",
          "of",
          "that",
          "war",
          "."
        ]
      },
      {
        "token": [
          "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",
          "."
        ]
      },
          {
            "token": [
              "It",
              "is",
              "altogether",
              "fitting",
              "and",
              "proper",
              "that",
              "we",
              "should",
              "do",
              "this",
              "."
            ]
          }
        ]
      }
    ],
    "meldedSentences": [
      {
        "meldedSentence": [
          "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."
        ]
      }
    ]
  }
}
```

XML output

```
<?xml version="1.0"?>
<SentenceSplitterResult>
    <text>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.</text>
    <langCode>en</langCode>
    <corpusConfig>ncf</corpusConfig>
    <sentences>
        <sentence>
            <token>Now</token>
            <token>we</token>
            <token>are</token>
            <token>engaged</token>
            <token>in</token>
            <token>a</token>
            <token>great</token>
            <token>civil</token>
            <token>war</token>
            <token>,</token>
            <token>testing</token>
            <token>whether</token>
            <token>that</token>
            <token>nation</token>
            <token>,</token>
            <token>or</token>
            <token>any</token>
            <token>nation</token>
            <token>,</token>
            <token>so</token>
            <token>conceived</token>
            <token>and</token>
            <token>so</token>
            <token>dedicated</token>
            <token>,</token>
            <token>can</token>
            <token>long</token>
            <token>endure</token>
            <token>.</token>
        </sentence>
        <sentence>
            <token>We</token>
            <token>are</token>
            <token>met</token>
            <token>on</token>
            <token>a</token>
            <token>great</token>
            <token>battle-field</token>
            <token>of</token>
            <token>that</token>
            <token>war</token>
            <token>.</token>
        </sentence>
        <sentence>
            <token>We</token>
            <token>have</token>
            <token>come</token>
            <token>to</token>
            <token>dedicate</token>
            <token>a</token>
            <token>portion</token>
            <token>of</token>
            <token>that</token>
            <token>field</token>
            <token>,</token>
            <token>as</token>
            <token>a</token>
            <token>final</token>
            <token>resting</token>
            <token>place</token>
            <token>for</token>
            <token>those</token>
            <token>who</token>
            <token>here</token>
            <token>gave</token>
            <token>their</token>
```

```
            <token>lives</token>
            <token>that</token>
            <token>that</token>
            <token>nation</token>
            <token>might</token>
            <token>live</token>
            <token>.</token>
        </sentence>
        <sentence>
            <token>It</token>
            <token>is</token>
            <token>altogether</token>
            <token>fitting</token>
            <token>and</token>
            <token>proper</token>
            <token>that</token>
            <token>we</token>
            <token>should</token>
            <token>do</token>
            <token>this</token>
            <token>.</token>
        </sentence>
    </sentences>
    <meldedSentences>
        <meldedSentence>Now we are engaged in a great civil war, testing whether
that nation, or any nation, so conceived and so dedicated, can long
endure.</meldedSentence>
        <meldedSentence>We are met on a great battle-field of that
war.</meldedSentence>
        <meldedSentence>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.</meldedSentence>
        <meldedSentence>It is altogether fitting and proper that we should do
this.</meldedSentence>
    </meldedSentences>
</SentenceSplitterResult>
```

HTML output (source)

```
<h3>4 sentences found.</h3>
<table border="0">
<tr>
<th align="left">S#</th>
<th align="left">Sentence</th>
</tr>
<tr>
<td valign="top" align="left"><strong>1</strong></td>
<td valign="top" align="left">Now we are engaged in a great civil war, testing
whether that nation, or any nation, so conceived and so dedicated, can long
endure.</td>
</tr>
<tr>
<td valign="top" align="left"><strong>2</strong></td>
<td valign="top" align="left">We are met on a great battle-field of that war.</td>
</tr>
<tr>
```

```
<td valign="top" align="left"><strong>3</strong></td>
<td valign="top" align="left">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.</td>
</tr>
<tr>
<td valign="top" align="left"><strong>4</strong></td>
<td valign="top" align="left">It is altogether fitting and proper that we should do
this.</td>
</tr>
</table>
```

HTML output (display)

4 sentences found.

S#	Sentence
1	Now we are engaged in a great civil war, testing whether that nation, or any nation, so conceived and so dedicated, can long endure.
2	We are met on a great battle-field of that war.
3	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.
4	It is altogether fitting and proper that we should do this.

Text output

```
4 sentences found.
S#      Sentence
1       Now we are engaged in a great civil war, testing whether that nation, or
any nation, so conceived and so dedicated, can long endure.
2       We are met on a great battle-field of that war.
3       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.
4       It is altogether fitting and proper that we should do this.
```

MorphAdorner Server Services: Spelling Standardizer Service

Service name:	spellingstandardizer
Service description:	Find standard spelling for a word.
HTTP methods allowed:	GET, POST, OPTIONS
POST accepts as input:	application/x-www-form-urlencoded
HTTP return codes:	200: service succeeded 400: service failed with an error
Query parameters	
corpusConfig	Corpus configuration name. In the standard distribution these are *ece*, *eme*, and *ncf*.
media	Result format. One of *json, xml, html, text* .
spelling	Spelling of a word.
wordClass	Primary word class. One of *adjective, adverb, compound, conjunction, infinitive-to, noun, noun-possessive, preposition, pronoun, pronoun-possessive, pronoun-possessive-determiner, verb* .
wordClass2	Secondary word class. One of *adjective, adverb, compound, conjunction, infinitive-to, noun, noun-possessive, preposition, pronoun, pronoun-possessive, pronoun-possessive-determiner, verb* .
extendedSearch	Perform an extended search for standard spellings. Allowed values are *true* to perform an extended search, *false* to not perform an extended search.

Sample POST form

```
<form accept-charset="UTF-8" method="post" action="spellingstandardizer"
      target="_blank"
      name="standardizer">
<table cellpadding="0" cellspacing="5">
<tr>
<td><strong>Spelling:</strong></td>
<td><input type="text" name="spelling" size="20" value="" /></td>
</tr>
<tr>
<td><strong>Primary word class:</strong></td>
<td>
<select name="wordClass">
<option value="" selected="selected"></option>
<option value="adjective">adjective</option>
<option value="adverb">adverb</option>
<option value="compound">compound</option>
<option value="conjunction">conjunction</option>
<option value="infinitive-to">infinitive-to</option>
<option value="noun">noun</option>
<option value="noun-possessive">noun-possessive</option>
```

```
<option value="preposition">preposition</option>
<option value="pronoun">pronoun</option>
<option value="pronoun-possessive">pronoun-possessive</option>
<option value="pronoun-possessive-determiner">pronoun-possessive-
determiner</option>
<option value="verb">verb</option>
</select>
</td>
</tr>
<tr>
<td><strong>Secondary word class:</strong></td>
<td>
<select name="wordClass2">
<option value="" selected="selected"></option>
<option value="adjective">adjective</option>
<option value="adverb">adverb</option>
<option value="compound">compound</option>
<option value="conjunction">conjunction</option>
<option value="infinitive-to">infinitive-to</option>
<option value="noun">noun</option>
<option value="noun-possessive">noun-possessive</option>
<option value="preposition">preposition</option>
<option value="pronoun">pronoun</option>
<option value="pronoun-possessive">pronoun-possessive</option>
<option value="pronoun-possessive-determiner">pronoun-possessive-
determiner</option>
<option value="verb">verb</option>
</select>
</td>
</tr>
<tr>
<td valign="top">
<strong>
Lexicon:</strong>
</td>
<td>
<input type="radio" name="corpusConfig" value="eme">Early Modern English</input><br
/>
<input type="radio" name="corpusConfig" value="ece">Eighteen Century
English</input><br />
<input type="radio" name="corpusConfig" value="ncf" checked="checked">Nineteenth
Century Fiction</input>
</td>
</tr>
<tr>
<td>&nbsp;</td>
<td><input type="checkbox" name="extendedSearch" value="true" />Perform extended
search for suggested spellings</td>
</tr>
<tr>
<td>
&nbsp;
</td>
<td>
&nbsp;
</td>
</tr>
```

```
<tr>
<td valign="top">
<strong>Results format:</strong>
</td>
<td>
<input type="radio" name="media" value="json">JSON format</input><br />
<input type="radio" name="media" value="xml" checked="checked">XML
format</input><br />
<input type="radio" name="media" value="html">HTML format</input><br />
<input type="radio" name="media" value="text">Text format</input>
</td>
</tr>
<tr>
<td>
<input type="submit" name="standardize" value="Standardize" />
</td>
</tr>
</table>
</form>
```

Output

Here is sample output for spelling *strykynge*, using the *eme* (early modern English) corpus configuration. We supply *verb* as the primary word class.

The JSON and XML formats echo the input *spelling*, the *corpusConfig*, and the primary and secondary word classes *wordClass* and *wordClass2* respectively. The resulting standard spelling **striking** is emitted as *standardSpelling*. The input query parameter field values are not emitted for the HTML or plain text output formats which are suitable for display.

JSON output

```
{
  "SpellingStandardizerResult": {
    "spelling": "strykynge",
    "standardSpelling": "striking",
    "corpusConfig": "eme",
    "wordClass": "verb",
    "wordClass2": ""
  }
}
```

XML output

```
<SpellingStandardizerResult>
    <spelling>strykynge</spelling>
    <standardSpelling>striking</standardSpelling>
    <corpusConfig>eme</corpusConfig>
    <wordClass>verb</wordClass>
    <wordClass2/>
</SpellingStandardizerResult><
```

HTML output (source)

```
<h3>Spelling Standardizer Results</h3>
<table border="0">
<tr>
<td valign="top" align="left"><strong>Standard spelling:</strong></td>
<td valign="top" align="left">striking</td>
</tr>
</table>
```

HTML output (display)

Spelling Standardizer Results

Standard spelling: striking

Text output

```
Spelling Standardizer Results
Standard spelling:      striking
```

MorphAdorner Server Services: Syllable Counter Service

Service name:	syllablecounter
Service description:	Count syllables in a word.
HTTP methods allowed:	GET, POST, OPTIONS
POST accepts as input:	application/x-www-form-urlencoded
HTTP return codes:	200: service succeeded 400: service failed with an error
Query parameters	
media	Result format. One of *json, xml, html, text* .
spelling	Spelling of a word.

Sample POST form

```
<form accept-charset="UTF-8" method="post" action="syllablecounter"
      target="_blank"
      name="syllablecounter">
<table cellpadding="0" cellspacing="5">
<tr>
<td><strong>Spelling:</strong></td>
<td><input type="text" name="spelling" size="20" value="" /></td>
</tr>
<tr>
<td>
&nbsp;
</td>
<td>
&nbsp;
</td>
</tr>
<tr>
<td valign="top">
<strong>Results format:</strong>
</td>
<td>
<input type="radio" name="media" value="json">JSON format</input><br />
<input type="radio" name="media" value="xml" checked="checked">XML
format</input><br />
<input type="radio" name="media" value="html">HTML format</input><br />
<input type="radio" name="media" value="text">Text format</input>
</td>
</tr>
<tr>
<td>
&nbsp;
</td>
<td>
```

```
&nbsp;
</td>
</tr>
<tr>
<td>
<input type="submit" name="countsyllables" value="Count Syllables" />
</td>
</tr>
</table>
</form>
```

Output

We count the syllables in the word "antidisestablishmentarianism."

JSON output

```
{
  "SyllableCounterResult": {
    "spelling": "antidisestablishmentarianism",
    "syllableCount": 11
  }
}
```

XML output

```
<SyllableCounterResult>
    <spelling>antidisestablishmentarianism</spelling>
    <syllableCount>11</syllableCount>
</SyllableCounterResult>
```

HTML output (source)

```
<h3>Syllable count results</h3>
<table border="0">
<tr>
<td valign="top" align="left"><strong>Spelling:</strong></td>
<td valign="top" align="left">antidisestablishmentarianism</td>
</tr>
<tr>
<td valign="top" align="left"><strong>Syllable count:</strong></td>
<td valign="top" align="left">11</td>
</tr>
</table>
```

HTML output (display)

Syllable count results

Spelling:	antidisestablishmentarianism
Syllable count:	11

Text output

```
Syllable count results
Spelling:       antidisestablishmentarianism
Syllable count: 11
```

MorphAdorner Server Services: Text Segmenter Service

Service name:	textsegmenter
Service description:	Break up a text into thematically meaningful segments.
HTTP methods allowed:	GET, POST, OPTIONS
POST accepts as input:	application/x-www-form-urlencoded
HTTP return codes:	200: service succeeded 400: service failed with an error
Query parameters	
corpusConfig	Corpus configuration name. In the standard distribution these are *ece*, *eme*, and *ncf*.
c99MaskSize	The C99 mask size. The default value is 11.
c99SegmentsWanted	The C99 value for the number text segmented wanted. The default value is -1, which lets the algorithm determine the number of segments.
includeInputText	Allowed values are *true* to include the input text in the output and *false* to not include the input text.
media	Result format. One of *json, xml, html, text* .
segmenterName	Text segmenter method name. The allowed values are *C99* and *Text Tiling*. Text tiling is the default.
text	Text to be processed.
tilerSlidingWindowSize	The sliding window size for the Text Tiling algorithm. The default value is 10.
tilerStepSize	The Text Tiling step size. The default value is 100.

Sample POST form

```
<form accept-charset="UTF-8" method="post" action="textsegmenter"
      target="_blank"
      name="segmenter">
<table cellpadding="0" cellspacing="5">
<tr>
<td><strong>Text:</strong></td>
<td colspan="2">
<textarea name="text" rows="15" cols="76"></textarea>
</td>
</tr>
<tr>
<td>&nbsp;</td>
<td>
<input type="checkbox" name="includeInputText" value="true"
       checked="checked"/>
Include input text in results
```

```
</td>
</tr>
<tr>
<td>
&nbsp;
</td>
<td>
&nbsp;
</td>
</tr>
<tr>
<td valign="top">
<strong>
Lexicon:</strong>
</td>
<td>
<input type="radio" name="corpusConfig" value="eme">Early Modern English</input><br
/>
<input type="radio" name="corpusConfig" value="ece">Eighteen Century
English</input><br />
<input type="radio" name="corpusConfig" value="ncf" checked="checked">Nineteenth
Century Fiction</input>
</td>
</tr>
<tr>
<td>
&nbsp;
</td>
<td>
&nbsp;
</td>
</tr>
<tr>
<td valign="top">
<strong>
Segmenter:</strong>
</td>
<td>
<input type="radio" name="segmenterName" value="C99">C99</input><br />
<table border="0">
<tr>
<td>
&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;
</td>
<td>
Mask size:
</td>
<td>
<input type="text" name="c99MaskSize" size="5" value="11" /></input>
</td>
</tr>
<tr>
<td>
&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;
</td>
<td>
Segments desired:
```

```
</td>
<td>
<input type="text" name="c99SegmentsWanted" size="5" value="-1" /></input>
</td>
</tr>
</table>
<input type="radio" name="segmenterName" value="Text Tiling"
checked="checked">Text Tiling</input><br />
<table border="0">
<tr>
<td>
&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;
</td>
<td>
Sliding window size:
</td>
<td>
<input type="text" name="tilerSlidingWindowSize" size="5" value="10" /></input>
</td>
</tr>
<tr>
<td>
&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;
</td>
<td>
Segment size:
</td>
<td>
<input type="text" name="tilerStepSize" size="5" value="100" /></input>
</td>
</tr>
</table>
</td>
</tr>
<tr>
<td>
&nbsp;
</td>
<td>
&nbsp;
</td>
</tr>
<tr>
<td valign="top">
<strong>Results format:</strong>
</td>
<td>
<input type="radio" name="media" value="json">JSON format</input><br />
<input type="radio" name="media" value="xml" checked="checked">XML
format</input><br />
<input type="radio" name="media" value="html">HTML format</input><br />
<input type="radio" name="media" value="text">Text format</input>
</td>
</tr>
<tr>
<td>
&nbsp;
```

```
</td>
<td>
&nbsp;
</td>
</tr>
<tr>
<td colspan="2">
<input type="submit" name="segment" value="Segment" />
</td>
</tr>
</table>
</form>
```

Output

Here is sample output for the text segmenter service. We use Abraham Lincoln's "Gettysburg Address" as the text. We select the default Text Tiling method.

The JSON and XML output echoes the input values. The *sentences* container wraps a sequence of *sentence* entries each of which represents a single parsed sentence from the input text. Each *sentence* contains a sequence of *token* entries representing the words and punctuation in the sentence. The *segments* container wraps a list of integer values specifying the index of the first sentence (0-origin) of each text segment. The *segmentTexts* wraps a series of *segmentText* entries each of which provides melded versions of the sentences comprising each text segments, in order. The HTML and text versions provide displayable versions of the text of the segments. The input values are not echoed.

JSON output

```
{
  "TextSegmenterResult": {
    "text": "Four score and seven years ago our fathers brought forth on
this\r\ncontinent a new nation, conceived in Liberty, and dedicated to\r\nthe
proposition that all men are created equal.\r\n\r\nNow we are engaged in a great
civil war, testing whether that\r\nnation, or any nation, so conceived and so
dedicated, can long\r\nendure. We are met on a great battle-field of that war. We
have\r\ncome to dedicate a portion of that field, as a final resting\r\nplace for
those who here gave their lives that that nation might\r\nlive. It is altogether
fitting and proper that we should do\r\nthis.\r\n\r\nBut, in a larger sense, we can
not dedicate -- we can not\r\nconsecrate -- we can not hallow -- this ground. The
brave men, living\r\nand dead, who struggled here, have consecrated it, far above
our\r\npoor power to add or detract. The world will little note, nor\r\nlong
remember what we say here, but it can never forget what\r\nthey did here. It is for
us the living, rather, to be dedicated\r\nhere to the unfinished work which they
who fought here have thus\r\nfar so nobly advanced. It is rather for us to be here
dedicated\r\nto the great task remaining before us -- that from these
honored\r\ndead we take increased devotion to that cause for which they\r\ngave the
last full measure of devotion -- that we here highly\r\nresolve that these dead
shall not have died in vain -- that this\r\nnation, under God, shall have a new
birth of freedom -- and that\r\ngovernment: of the people, by the people, for the
people, shall\r\nnot perish from the earth.",
    "corpusConfig": "ncf",
    "c99MaskSize": 11,
    "c99SegmentsWanted": -1,
    "tilerSlidingWindowSize": 10,
    "tilerStepSize": 100,
"sentences": [
  {
    "sentence": [
      {
        "token": [
          "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",
          "."
        ]
      },
      {
        "token": [
          "Now",
          "we",
          "are",
          "engaged",
          "in",
          "a",
          "great",
          "civil",
          "war",
          ",",
          "testing",
          "whether",
          "that",
          "nation",
          ",",
        "or",
        "any",
        "nation",
        ",",
        "so",
        "conceived",
        "and",
        "so",
        "dedicated",
        ",",
        "can",
        "long",
        "endure",
        "."
      ]
    },
    {
      "token": [
        "We",
        "are",
        "met",
        "on",
        "a",
        "great",
        "battle-field",
        "of",
        "that",
        "war",
        "."
      ]
    },
    {
      "token": [
        "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",
            "."
          ]
        },
        {
          "token": [
            "It",
            "is",
            "altogether",
            "fitting",
            "and",
            "proper",
            "that",
            "we",
            "should",
            "do",
            "this",
            "."
          ]
        },
        {
          "token": [
            "But",
            ",",
            "in",
            "a",
            "larger",
            "sense",
            ",",
            "we",
            "can",
            "not",
            "dedicate",
            "--",
            "we",
            "can",
            "not",
            "consecrate",
            "--",
            "we",
            "can",
            "not",
            "hallow",
            "--",
            "this",
            "ground",
            "."
          ]
        },
        {
          "token": [
            "The",
            "brave",
            "men",
        ",",
        "living",
        "and",
        "dead",
        ",",
        "who",
        "struggled",
        "here",
        ",",
        "have",
        "consecrated",
        "it",
        ",",
        "far",
        "above",
        "our",
        "poor",
        "power",
        "to",
        "add",
        "or",
        "detract",
        "."
      ]
    },
    {
      "token": [
        "The",
        "world",
        "will",
        "little",
        "note",
        ",",
        "nor",
        "long",
        "remember",
        "what",
        "we",
        "say",
        "here",
        ",",
        "but",
        "it",
        "can",
        "never",
        "forget",
        "what",
        "they",
        "did",
        "here",
        "."
      ]
    },
    {
      "token": [
        "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",
        "."
      ]
    },
    {
      "token": [
        "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",
          "."
        ]
      }
    ]
  }
],
"segments": [
  {
    "int": [
      0,
      5
    ]
  }
],
"segmenterName": "Text Tiling",
"segmentTexts": [
  {
    "segmentText": [
      "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.  "
    ]
  }
]
}
}
```

XML output

```
<TextSegmenterResult>
    <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.</text>
    <corpusConfig>ncf</corpusConfig>
    <c99MaskSize>11</c99MaskSize>
    <c99SegmentsWanted>-1</c99SegmentsWanted>
    <tilerSlidingWindowSize>10</tilerSlidingWindowSize>
    <tilerStepSize>100</tilerStepSize>
    <sentences>
        <sentence>
            <token>Four</token>
            <token>score</token>
            <token>and</token>
            <token>seven</token>
            <token>years</token>
            <token>ago</token>
            <token>our</token>
            <token>fathers</token>
            <token>brought</token>
            <token>forth</token>
            <token>on</token>
            <token>this</token>
            <token>continent</token>
            <token>a</token>
            <token>new</token>
            <token>nation</token>
            <token>,</token>
            <token>conceived</token>
            <token>in</token>
            <token>Liberty</token>
            <token>,</token>
            <token>and</token>
            <token>dedicated</token>
            <token>to</token>
            <token>the</token>
            <token>proposition</token>
            <token>that</token>
            <token>all</token>
            <token>men</token>
            <token>are</token>
            <token>created</token>
            <token>equal</token>
            <token>.</token>
        </sentence>
        <sentence>
            <token>Now</token>
            <token>we</token>
            <token>are</token>
            <token>engaged</token>
            <token>in</token>
            <token>a</token>
            <token>great</token>
            <token>civil</token>
            <token>war</token>
            <token>,</token>
            <token>testing</token>
            <token>whether</token>
            <token>that</token>
            <token>nation</token>
            <token>,</token>
            <token>or</token>
            <token>any</token>
            <token>nation</token>
            <token>,</token>
            <token>so</token>
            <token>conceived</token>
            <token>and</token>
            <token>so</token>
            <token>dedicated</token>
            <token>,</token>
            <token>can</token>
            <token>long</token>
            <token>endure</token>
            <token>.</token>
        </sentence>
        <sentence>
            <token>We</token>
            <token>are</token>
            <token>met</token>
            <token>on</token>
            <token>a</token>
            <token>great</token>
            <token>battle-field</token>
            <token>of</token>
            <token>that</token>
            <token>war</token>
            <token>.</token>
        </sentence>
        <sentence>
```

```
        <token>We</token>
        <token>have</token>
        <token>come</token>
        <token>to</token>
        <token>dedicate</token>
        <token>a</token>
        <token>portion</token>
        <token>of</token>
        <token>that</token>
        <token>field</token>
        <token>,</token>
        <token>as</token>
        <token>a</token>
        <token>final</token>
        <token>resting</token>
        <token>place</token>
        <token>for</token>
        <token>those</token>
        <token>who</token>
        <token>here</token>
        <token>gave</token>
        <token>their</token>
        <token>lives</token>
        <token>that</token>
        <token>that</token>
        <token>nation</token>
        <token>might</token>
        <token>live</token>
        <token>.</token>
    </sentence>
    <sentence>
        <token>It</token>
        <token>is</token>
        <token>altogether</token>
        <token>fitting</token>
        <token>and</token>
        <token>proper</token>
        <token>that</token>
        <token>we</token>
        <token>should</token>
        <token>do</token>
        <token>this</token>
        <token>.</token>
    </sentence>
    <sentence>
        <token>But</token>
        <token>,</token>
        <token>in</token>
        <token>a</token>
        <token>larger</token>
        <token>sense</token>
        <token>,</token>
        <token>we</token>
        <token>can</token>
        <token>not</token>
        <token>dedicate</token>
        <token>--</token>
        <token>we</token>
        <token>can</token>
        <token>not</token>
        <token>consecrate</token>
        <token>--</token>
        <token>we</token>
        <token>can</token>
        <token>not</token>
        <token>hallow</token>
        <token>--</token>
        <token>this</token>
        <token>ground</token>
        <token>.</token>
    </sentence>
    <sentence>
        <token>The</token>
        <token>brave</token>
        <token>men</token>
        <token>,</token>
        <token>living</token>
        <token>and</token>
        <token>dead</token>
        <token>,</token>
        <token>who</token>
        <token>struggled</token>
        <token>here</token>
        <token>,</token>
        <token>have</token>
        <token>consecrated</token>
        <token>it</token>
        <token>,</token>
        <token>far</token>
        <token>above</token>
        <token>our</token>
        <token>poor</token>
        <token>power</token>
        <token>to</token>
        <token>add</token>
        <token>or</token>
        <token>detract</token>
        <token>.</token>
    </sentence>
    <sentence>
        <token>The</token>
        <token>world</token>
        <token>will</token>
        <token>little</token>
        <token>note</token>
        <token>,</token>
        <token>nor</token>
        <token>long</token>
        <token>remember</token>
        <token>what</token>
        <token>we</token>
        <token>say</token>
        <token>here</token>
        <token>,</token>
        <token>but</token>
        <token>it</token>
        <token>can</token>
        <token>never</token>
        <token>forget</token>
        <token>what</token>
        <token>they</token>
        <token>did</token>
        <token>here</token>
        <token>.</token>
    </sentence>
    <sentence>
        <token>It</token>
        <token>is</token>
        <token>for</token>
        <token>us</token>
        <token>the</token>
        <token>living</token>
        <token>,</token>
        <token>rather</token>
        <token>,</token>
        <token>to</token>
        <token>be</token>
        <token>dedicated</token>
        <token>here</token>
        <token>to</token>
        <token>the</token>
        <token>unfinished</token>
        <token>work</token>
        <token>which</token>
        <token>they</token>
        <token>who</token>
        <token>fought</token>
        <token>here</token>
        <token>have</token>
        <token>thus</token>
        <token>far</token>
        <token>so</token>
        <token>nobly</token>
        <token>advanced</token>
        <token>.</token>
    </sentence>
    <sentence>
        <token>It</token>
        <token>is</token>
        <token>rather</token>
        <token>for</token>
        <token>us</token>
        <token>to</token>
        <token>be</token>
        <token>here</token>
        <token>dedicated</token>
        <token>to</token>
        <token>the</token>
        <token>great</token>
        <token>task</token>
        <token>remaining</token>
```

```
<token>before</token>
<token>us</token>
<token>--</token>
<token>that</token>
<token>from</token>
<token>these</token>
<token>honored</token>
<token>dead</token>
<token>we</token>
<token>take</token>
<token>increased</token>
<token>devotion</token>
<token>to</token>
<token>that</token>
<token>cause</token>
<token>for</token>
<token>which</token>
<token>they</token>
<token>gave</token>
<token>the</token>
<token>last</token>
<token>full</token>
<token>measure</token>
<token>of</token>
<token>devotion</token>
<token>--</token>
<token>that</token>
<token>we</token>
<token>here</token>
<token>highly</token>
<token>resolve</token>
<token>that</token>
<token>these</token>
<token>dead</token>
<token>shall</token>
<token>not</token>
<token>have</token>
<token>died</token>
<token>in</token>
<token>vain</token>
<token>--</token>
<token>that</token>
<token>this</token>
<token>nation</token>
<token>,</token>
<token>under</token>
<token>God</token>
<token>,</token>
<token>shall</token>
<token>have</token>
<token>a</token>
<token>new</token>
<token>birth</token>
<token>of</token>
<token>freedom</token>
<token>--</token>
<token>and</token>
```

```
            <token>that</token>
            <token>government</token>
            <token>:</token>
            <token>of</token>
            <token>the</token>
            <token>people</token>
            <token>,</token>
            <token>by</token>
            <token>the</token>
            <token>people</token>
            <token>,</token>
            <token>for</token>
            <token>the</token>
            <token>people</token>
            <token>,</token>
            <token>shall</token>
            <token>not</token>
            <token>perish</token>
            <token>from</token>
            <token>the</token>
            <token>earth</token>
            <token>.</token>
        </sentence>
    </sentences>
    <segments>
        <int>0</int>
        <int>5</int>
    </segments>
    <segmenterName>Text Tiling</segmenterName>
    <segmentTexts>
        <segmentText>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.  </segmentText>
        <segmentText>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.  </segmentText>
    </segmentTexts>
</TextSegmenterResult>
```

HTML output (source)

```
<h3>2 segments found using Text Tiling.</h3>
```

```
<table border="0">
<tr>
<th align="left">Segment</th>
<th align="left">Text</th>
</tr>
<tr>
<td valign="top" align="left"><strong>1</strong></td>
<td valign="top" align="left">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.</td>
</tr>
<tr>
<td valign="top" align="left"><strong>2</strong></td>
<td valign="top" align="left">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.</td>
</tr>
</table>
```

HTML output (display)

2 segments found using Text Tiling.

Segment	Text
1	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.
2	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.

Text output

```
2 segments found using Text Tiling.
Segment Text
1       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.
2       But, in a larger sense, we can not dedicate -- we can not consecrate -- we
can not hallow -
```

MorphAdorner Server Services: Text Summarizer Service

Service name:	summarizer
Service description:	Find summary sentences for a text.
HTTP methods allowed:	GET, POST, OPTIONS
POST accepts as input:	application/x-www-form-urlencoded
HTTP return codes:	200: service succeeded 400: service failed with an error
Query parameters	
corpusConfig	Corpus configuration name. In the standard distribution these are *ece*, *eme*, and *ncf*.
media	Result format. One of *json, xml, html, text* .
text	Text to be processed.
includeInputText	Allowed values are *true* to include the input text in the output and *false* to not include the input text.
maxSumSent	Maximum number of summary sentences.

Sample POST form

```
<form accept-charset="UTF-8" method="post" action="summarizer"
      target="_blank"
      name="summarizer">
<table cellpadding="0" cellspacing="5">
<tr>
<td><strong>Text:</strong></td>
<td colspan="2">
<textarea name="text" rows="15" cols="76"></textarea>
</td>
</tr>
<tr>
<td>&nbsp;</td>
<td>
<input type="checkbox" name="includeInputText" value="true"
       checked="checked"/>
Include input text in results
</td>
</tr>
<tr>
<td valign="top">
<strong>
Lexicon:</strong>
</td>
<td>
<input type="radio" name="corpusConfig" value="eme">Early Modern English</input><br
```

```
/>
<input type="radio" name="corpusConfig" value="ece">Eighteen Century
English</input><br />
<input type="radio" name="corpusConfig" value="ncf" checked="checked">Nineteenth
Century Fiction</input>
</td>
</tr>
<tr>
<td><strong>Summary sentences:</strong></td>
<td>
<input type="text" name="maxSumSent" size = "20" value="5" />
</td>
</tr>
<tr>
<td>
&nbsp;
</td>
<td>
&nbsp;
</td>
</tr>
<tr>
<td valign="top">
<strong>Results format:</strong>
</td>
<td>
<input type="radio" name="media" value="json">JSON format</input><br />
<input type="radio" name="media" value="xml" checked="checked">XML
format</input><br />
<input type="radio" name="media" value="html">HTML format</input><br />
<input type="radio" name="media" value="text">Text format</input>
</td>
</tr>
<tr>
<td>
&nbsp;
</td>
<td>
&nbsp;
</td>
</tr>
<tr>
<td colspan="2">
<input type="submit" name="summarizer" value="Summarizer" />
</td>
</tr>
</table>
</form>
```

Output

Here is sample output for the text summarizer service. We use Abraham Lincoln's "Gettysburg Address" as the text, and request a two sentence summary. The JSON and XML output optionally echoes back the input text.

JSON output

```
{
  "SummarizerResult": {
    "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.",
    "corpusConfig": "ncf",
    "maxSumSent": 2,
    "summaryText": "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. 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. "
  }
}
```

XML output

```
<SummarizerResult>
    <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.</text>
    <corpusConfig>ncf</corpusConfig>
    <maxSumSent>2</maxSumSent>
    <summaryText>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. 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.</summaryText>
</SummarizerResult>
```

HTML output (source)

```
<p>
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. 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.
</p>
```

HTML output (display)

```
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. 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.
```

Text output

```
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. 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.
```

MorphAdorner Server Services: Thesaurus Service

Service name:	thesaurus
Service description:	Find synonyms and antonyms for a spelling.
HTTP methods allowed:	GET, POST, OPTIONS
POST accepts as input:	application/x-www-form-urlencoded
HTTP return codes:	200: service succeeded 400: service failed with an error
Query parameters	
media	Result format. One of *json, xml, html, text* .
spelling	Spelling of a word.
addSynAnt	Allowed values are *true* to add the synonyms of the antonyms to antonym list and *false* to not add the antonyms.
wordClass	Word class. One of *adjective, adverb, noun, verb*, or no selection to search all four word classes.

Sample POST form

```
<form accept-charset="UTF-8" method="post" action="thesaurus"
      target="_blank"
      name="thesaurus">
<table cellpadding="0" cellspacing="5">
<tr>
<td><strong>Spelling:</strong></td>
<td><input type="text" name="spelling" size="20" value="" /></td>
</tr>
<tr>
<td><strong>Word class:</strong></td>
<td>
<select name="wordClass">
<option value="" selected="selected"></option>
<option value="adjective">adjective</option>
<option value="adverb">adverb</option>
<option value="noun">noun</option>
<option value="verb">verb</option>
</select>
</td>
</tr>
<tr>
<td>&nbsp;</td>
<td><input type="checkbox" name="addSynAnt" value="true" />Add synonyms of
antonyms</td>
</tr>
<tr>
<td>
&nbsp;
```

```
</td>
<td>
&nbsp;
</td>
</tr>
<tr>
<td valign="top">
<strong>Results format:</strong>
</td>
<td>
<input type="radio" name="media" value="json">JSON format</input><br />
<input type="radio" name="media" value="xml" checked="checked">XML
format</input><br />
<input type="radio" name="media" value="html">HTML format</input><br />
<input type="radio" name="media" value="text">Text format</input>
</td>
</tr>
<tr>
<td>
&nbsp;
</td>
<td>
&nbsp;
</td>
</tr>
<tr>
<td>
<input type="submit" name="thesaurus" value="Thesaurus" />
</td>
</tr>
</table>
</form>
```

Output

Here is a list of synonyms and the principal antonym of "hot."

JSON output

```
{
  "ThesaurusResult": {
    "spelling": "hot",
    "wordClass": "",
    "addSynAnt": false,
    "synonyms": [
      {
        "synonym": [
          "blistering",
          "hot",
          "live",
          "raging",
          "red-hot",
          "spicy"
        ]
      }
    ],
    "antonyms": [
      {
        "antonym": "cold"
      }
    ]
  }
}
```

XML output

```
<ThesaurusResult>
    <spelling>hot</spelling>
    <wordClass/>
    <addSynAnt>false</addSynAnt>
    <synonyms>
        <synonym>blistering</synonym>
        <synonym>hot</synonym>
        <synonym>live</synonym>
        <synonym>raging</synonym>
        <synonym>red-hot</synonym>
        <synonym>spicy</synonym>
    </synonyms>
    <antonyms>
        <antonym>cold</antonym>
    </antonyms>
</ThesaurusResult>
```

HTML output (source)

```
<h3>
6 synonyms found for hot.
</h3>
<table border="0">
<tr><td>blistering</td></tr>
<tr><td>hot</td></tr>
<tr><td>live</td></tr>
<tr><td>raging</td></tr>
<tr><td>red-hot</td></tr>
<tr><td>spicy</td></tr>
</table>
<h3>
1 antonym found for hot.
</h3>
<table border="0">
<tr><td>cold</td></tr>
</table>
```

HTML output (display)

6 synonyms found for hot.

blistering

hot

live

raging

red-hot

spicy

1 antonym found for hot.

cold

Text output

```
6 synonyms found for hot.
blistering
hot
live
raging
red-hot
spicy
1 antonym found for hot.
cold
```

MorphAdorner Server Services: Word Tokenizer Service

Service name:	wordtokenizer
Service description:	Split text into words and punctuation.
HTTP methods allowed:	GET, POST, OPTIONS
POST accepts as input:	application/x-www-form-urlencoded
HTTP return codes:	200: service succeeded 400: service failed with an error
Query parameters	
corpusConfig	Corpus configuration name. In the standard distribution these are *ece*, *eme*, and *ncf*.
media	Result format. One of *json, xml, html, text* .
text	Text to be processed.
includeInputText	Allowed values are *true* to include the input text in the output and *false* to not include the input text.
langCode	ISO language code. These are two or three character codes. The default is *en*, English. You may specify *** *Detect* *** to indicate that the server should try to determine the language from the text provided.

Sample POST form

```
<form accept-charset="UTF-8" method="post" action="wordtokenizer"
      target="_blank"
      name="wordtokenizer">
<table cellpadding="0" cellspacing="5">
<tr>
<td><strong>Text:</strong></td>
<td colspan="2">
<textarea name="text" rows="15" cols="76"></textarea>
</td>
</tr>
<tr>
<td valign="top">
<strong>
Lexicon:</strong>
</td>
<td>
<input type="radio" name="corpusConfig" value="eme">Early Modern English</input><br
/>
<input type="radio" name="corpusConfig" value="ece">Eighteen Century
English</input><br />
<input type="radio" name="corpusConfig" value="ncf" checked="checked">Nineteenth
Century Fiction</input>
</td>
</tr>
<tr>
<td><strong>Language:</strong></td>
```

```
<td>
<select name="langCode">
<option value="en" selected="selected">English</option>
<option value="">*** Detect ***</option>
<option value="af">Afrikaans</option>
<option value="ak">Akan</option>
<option value="sq">Albanian</option>
<option value="am">Amharic</option>
<option value="ar">Arabic</option>
<option value="hy">Armenian</option>
<option value="as">Assamese</option>
<option value="az">Azerbaijani</option>
<option value="bm">Bambara</option>
<option value="bas">Basa</option>
<option value="eu">Basque</option>
<option value="be">Belarusian</option>
<option value="bem">Bemba</option>
<option value="bn">Bengali</option>
<option value="bs">Bosnian</option>
<option value="br">Breton</option>
<option value="bg">Bulgarian</option>
<option value="my">Burmese</option>
<option value="ca">Catalan</option>
<option value="chr">Cherokee</option>
<option value="zh">Chinese</option>
<option value="kw">Cornish</option>
<option value="hr">Croatian</option>
<option value="cs">Czech</option>
<option value="da">Danish</option>
<option value="dua">Duala</option>
<option value="nl">Dutch</option>
<option value="eo">Esperanto</option>
<option value="et">Estonian</option>
<option value="ee">Ewe</option>
<option value="ewo">Ewondo</option>
<option value="fo">Faroese</option>
<option value="fil">Filipino</option>
<option value="fi">Finnish</option>
<option value="fr">French</option>
<option value="ff">Fulah</option>
<option value="gl">Gallegan</option>
<option value="lg">Ganda</option>
<option value="ka">Georgian</option>
<option value="de">German</option>
<option value="el">Greek</option>
<option value="kl">Greenlandic</option>
<option value="gu">Gujarati</option>
<option value="ha">Hausa</option>
<option value="haw">Hawaiian</option>
<option value="iw">Hebrew</option>
<option value="hi">Hindi</option>
<option value="hu">Hungarian</option>
<option value="is">Icelandic</option>
<option value="ig">Igbo</option>
<option value="in">Indonesian</option>
<option value="ga">Irish</option>
<option value="it">Italian</option>
```

```
<option value="ja">Japanese</option>
<option value="kab">Kabyle</option>
<option value="kam">Kamba</option>
<option value="kn">Kannada</option>
<option value="kk">Kazakh</option>
<option value="km">Khmer</option>
<option value="ki">Kikuyu</option>
<option value="rw">Kinyarwanda</option>
<option value="kok">Konkani</option>
<option value="ko">Korean</option>
<option value="lv">Latvian</option>
<option value="ln">Lingala</option>
<option value="lt">Lithuanian</option>
<option value="lu">Luba-Katanga</option>
<option value="mk">Macedonian</option>
<option value="mg">Malagasy</option>
<option value="ms">Malay</option>
<option value="ml">Malayalam</option>
<option value="mt">Maltese</option>
<option value="gv">Manx</option>
<option value="mr">Marathi</option>
<option value="mas">Masai</option>
<option value="ne">Nepali</option>
<option value="nd">North Ndebele</option>
<option value="nb">Norwegian Bokm?l</option>
<option value="nn">Norwegian Nynorsk</option>
<option value="nyn">Nyankole</option>
<option value="or">Oriya</option>
<option value="om">Oromo</option>
<option value="pa">Panjabi</option>
<option value="fa">Persian</option>
<option value="pl">Polish</option>
<option value="pt">Portuguese</option>
<option value="ps">Pushto</option>
<option value="rm">Raeto-Romance</option>
<option value="ro">Romanian</option>
<option value="rn">Rundi</option>
<option value="ru">Russian</option>
<option value="sg">Sango</option>
<option value="sr">Serbian</option>
<option value="sn">Shona</option>
<option value="ii">Sichuan Yi</option>
<option value="si">Sinhalese</option>
<option value="sk">Slovak</option>
<option value="sl">Slovenian</option>
<option value="so">Somali</option>
<option value="es">Spanish</option>
<option value="sw">Swahili</option>
<option value="sv">Swedish</option>
<option value="gsw">Swiss German</option>
<option value="ta">Tamil</option>
<option value="te">Telugu</option>
<option value="th">Thai</option>
<option value="bo">Tibetan</option>
<option value="ti">Tigrinya</option>
<option value="to">Tonga</option>
<option value="tr">Turkish</option>
```

```
<option value="uk">Ukrainian</option>
<option value="ur">Urdu</option>
<option value="uz">Uzbek</option>
<option value="vai">Vai</option>
<option value="vi">Vietnamese</option>
<option value="cy">Welsh</option>
<option value="yo">Yoruba</option>
<option value="zu">Zulu</option>
</select>
</td>
</tr>
<tr>
<td>&nbsp;</td>
<td>
<input type="checkbox" name="includeInputText" value="true"
       checked="checked"/>
Include input text in results
</td>
</tr>
<tr>
<td>
&nbsp;
</td>
<td>
&nbsp;
</td>
</tr>
<tr>
<td valign="top">
<strong>Results format:</strong>
</td>
<td>
<input type="radio" name="media" value="json">JSON format</input><br />
<input type="radio" name="media" value="xml" checked="checked">XML
format</input><br />
<input type="radio" name="media" value="html">HTML format</input><br />
<input type="radio" name="media" value="text">Text format</input>
</td>
</tr>
<tr>
<td>
&nbsp;
</td>
<td>
&nbsp;
</td>
</tr>
<tr>
<td colspan="2">
<input type="submit" name="tokenizer" value="Tokenize" />
</td>
</tr>
</table>
</form>
```

Output

Here we tokenize the first two sentences of Sarah Hale's poem "Mary had a little lamb."

> Mary had a little lamb,
> whose fleece was white as snow.
> And everywhere that Mary went,
> the lamb was sure to go.

The JSON and XML *WordTokenizerResult* echo the input *text*, the ISO language code *langCode*, and the *corpusConfig*. The *sentences* container wraps a sequence of *sentence* entries each of which represents a single parsed sentence from the input text. Each *sentence* contains a sequence of *token* entries representing the words and punctuation in the sentence. The HTML and text versions provide displayable versions of the tokenized sentences.

JSON output

```
{
  "WordTokenizerResult": {
    "text": "Mary had a little lamb,  whose fleece was white as snow.  And
everywhere that Mary went,  the lamb was sure to go.",
    "langCode": "en",
    "corpusConfig": "ncf",
    "sentences": [
      {
        "sentence": [
          {
            "token": [
              "Mary",
              "had",
              "a",
              "little",
              "lamb",
              ",",
              "whose",
              "fleece",
              "was",
              "white",
              "as",
              "snow",
              "."
            ]
          },
          {
            "token": [
              "And",
              "everywhere",
              "that",
              "Mary",
              "went",
              ",",
              "the",
              "lamb",
              "was",
              "sure",
                "to",
                "go",
                "."
              ]
            }
          ]
        }
      ]
    }
}
```

XML output

```
<WordTokenizerResult>
    <text>Mary had a little lamb,  whose fleece was white as snow.  And everywhere
that Mary went,  the lamb was sure to go.</text>
    <langCode>en</langCode>
    <corpusConfig>ncf</corpusConfig>
    <sentences>
        <sentence>
            <token>Mary</token>
            <token>had</token>
            <token>a</token>
            <token>little</token>
            <token>lamb</token>
            <token>,</token>
            <token>whose</token>
            <token>fleece</token>
            <token>was</token>
            <token>white</token>
            <token>as</token>
            <token>snow</token>
            <token>.</token>
        </sentence>
        <sentence>
            <token>And</token>
            <token>everywhere</token>
            <token>that</token>
            <token>Mary</token>
            <token>went</token>
            <token>,</token>
            <token>the</token>
            <token>lamb</token>
            <token>was</token>
            <token>sure</token>
            <token>to</token>
            <token>go</token>
            <token>.</token>
        </sentence>
    </sentences>
</WordTokenizerResult>
```

HTML output (source)

```
<h3>26 words in 2 sentences found.</h3>
<table border="0">
```

```
<tr>
<th align="left">S#</th>
<th align="left">W#</th>
<th align="left">Token</th>
<th align="left">Type</th>
</tr>
<tr>
<td valign="top" align="left"><strong>1</strong></td>
<td valign="top" align="left">1</td>
<td valign="top" align="left">Mary</td>
<td valign="top" align="left">token</td>
</tr>
<tr>
<td valign="top" align="left"><strong>1</strong></td>
<td valign="top" align="left">2</td>
<td valign="top" align="left">had</td>
<td valign="top" align="left">token</td>
</tr>
<tr>
<td valign="top" align="left"><strong>1</strong></td>
<td valign="top" align="left">3</td>
<td valign="top" align="left">a</td>
<td valign="top" align="left">token</td>
</tr>
<tr>
<td valign="top" align="left"><strong>1</strong></td>
<td valign="top" align="left">4</td>
<td valign="top" align="left">little</td>
<td valign="top" align="left">token</td>
</tr>
<tr>
<td valign="top" align="left"><strong>1</strong></td>
<td valign="top" align="left">5</td>
<td valign="top" align="left">lamb</td>
<td valign="top" align="left">token</td>
</tr>
<tr>
<td valign="top" align="left"><strong>1</strong></td>
<td valign="top" align="left">6</td>
<td valign="top" align="left">,</td>
<td valign="top" align="left">punctuation</td>
</tr>
<tr>
<td valign="top" align="left"><strong>1</strong></td>
<td valign="top" align="left">7</td>
<td valign="top" align="left">whose</td>
<td valign="top" align="left">token</td>
</tr>
<tr>
<td valign="top" align="left"><strong>1</strong></td>
<td valign="top" align="left">8</td>
<td valign="top" align="left">fleece</td>
<td valign="top" align="left">token</td>
</tr>
<tr>
<td valign="top" align="left"><strong>1</strong></td>
<td valign="top" align="left">9</td>
```

```
<td valign="top" align="left">was</td>
<td valign="top" align="left">token</td>
</tr>
<tr>
<td valign="top" align="left"><strong>1</strong></td>
<td valign="top" align="left">10</td>
<td valign="top" align="left">white</td>
<td valign="top" align="left">token</td>
</tr>
<tr>
<td valign="top" align="left"><strong>1</strong></td>
<td valign="top" align="left">11</td>
<td valign="top" align="left">as</td>
<td valign="top" align="left">token</td>
</tr>
<tr>
<td valign="top" align="left"><strong>1</strong></td>
<td valign="top" align="left">12</td>
<td valign="top" align="left">snow</td>
<td valign="top" align="left">token</td>
</tr>
<tr>
<td valign="top" align="left"><strong>1</strong></td>
<td valign="top" align="left">13</td>
<td valign="top" align="left">.</td>
<td valign="top" align="left">punctuation</td>
</tr>
<tr>
<td valign="top" align="left"><strong>2</strong></td>
<td valign="top" align="left">1</td>
<td valign="top" align="left">And</td>
<td valign="top" align="left">token</td>
</tr>
<tr>
<td valign="top" align="left"><strong>2</strong></td>
<td valign="top" align="left">2</td>
<td valign="top" align="left">everywhere</td>
<td valign="top" align="left">token</td>
</tr>
<tr>
<td valign="top" align="left"><strong>2</strong></td>
<td valign="top" align="left">3</td>
<td valign="top" align="left">that</td>
<td valign="top" align="left">token</td>
</tr>
<tr>
<td valign="top" align="left"><strong>2</strong></td>
<td valign="top" align="left">4</td>
<td valign="top" align="left">Mary</td>
<td valign="top" align="left">token</td>
</tr>
<tr>
<td valign="top" align="left"><strong>2</strong></td>
<td valign="top" align="left">5</td>
<td valign="top" align="left">went</td>
<td valign="top" align="left">token</td>
</tr>
```

```
<tr>
<td valign="top" align="left"><strong>2</strong></td>
<td valign="top" align="left">6</td>
<td valign="top" align="left">,</td>
<td valign="top" align="left">punctuation</td>
</tr>
<tr>
<td valign="top" align="left"><strong>2</strong></td>
<td valign="top" align="left">7</td>
<td valign="top" align="left">the</td>
<td valign="top" align="left">token</td>
</tr>
<tr>
<td valign="top" align="left"><strong>2</strong></td>
<td valign="top" align="left">8</td>
<td valign="top" align="left">lamb</td>
<td valign="top" align="left">token</td>
</tr>
<tr>
<td valign="top" align="left"><strong>2</strong></td>
<td valign="top" align="left">9</td>
<td valign="top" align="left">was</td>
<td valign="top" align="left">token</td>
</tr>
<tr>
<td valign="top" align="left"><strong>2</strong></td>
<td valign="top" align="left">10</td>
<td valign="top" align="left">sure</td>
<td valign="top" align="left">token</td>
</tr>
<tr>
<td valign="top" align="left"><strong>2</strong></td>
<td valign="top" align="left">11</td>
<td valign="top" align="left">to</td>
<td valign="top" align="left">token</td>
</tr>
<tr>
<td valign="top" align="left"><strong>2</strong></td>
<td valign="top" align="left">12</td>
<td valign="top" align="left">go</td>
<td valign="top" align="left">token</td>
</tr>
<tr>
<td valign="top" align="left"><strong>2</strong></td>
<td valign="top" align="left">13</td>
<td valign="top" align="left">.</td>
<td valign="top" align="left">punctuation</td>
</tr>
</table>
```

HTML output (display)

26 words in 2 sentences found.

S#	W#	Token	Type
1	1	Mary	token
1	2	had	token
1	3	a	token
1	4	little	token
1	5	lamb	token
1	6	,	punctuation
1	7	whose	token
1	8	fleece	token
1	9	was	token
1	10	white	token
1	11	as	token
1	12	snow	token
1	13	.	punctuation
2	1	And	token
2	2	everywhere	token
2	3	that	token
2	4	Mary	token
2	5	went	token
2	6	,	punctuation
2	7	the	token
2	8	lamb	token
2	9	was	token
2	10	sure	token
2	11	to	token
2	12	go	token
2	13	.	punctuation

Text output

```
26 words in 2 sentences found.
S#      W#      Token   Type
1       1       Mary    token
1       2       had     token
1       3       a       token
1       4       little  token
1       5       lamb    token
1       6       ,       punctuation
1       7       whose   token
1       8       fleece  token
1       9       was     token
1       10      white   token
1       11      as      token
1       12      snow    token
1       13      .       punctuation
2       1       And     token
2       2       everywhere      token
2       3       that    token
2       4       Mary    token
2       5       went    token
2       6       ,       punctuation
2       7       the     token
2       8       lamb    token
2       9       was     token
2       10      sure    token
2       11      to      token
2       12      go      token
2       13      .       punctuation
```

MorphAdorner Server Services: Verb Conjugator Service

Service name:	verbconjugator
Service description:	Conjugate an English verb.
HTTP methods allowed:	GET, POST, OPTIONS
POST accepts as input:	application/x-www-form-urlencoded
HTTP return codes:	200: service succeeded 400: service failed with an error
Query parameters	
american	Display American (U.S.) spellings of conjugated verbs. Allowed values are *true* to display American spellings, *false* for British spellings.
infinitive	Infinitive of an English verb. The leading "to" is not specified.
media	Result format. One of *json, xml, html, text* .
verbTense	English verb tense for which to provide conjugation. Available values are *present*, *presentParticiple*, *past, and pastParticiple.*

Sample POST form

```
<form accept-charset="UTF-8" method="post" action="verbconjugator"
      name="conjugator">
<table cellpadding="0" cellspacing="5">
<tr>
<td><strong>Infinitive:</strong></td>
<td><input type="text" name="infinitive" size="20" value="" /></td>
</tr>
<tr>
<td><strong>Verb tense:</strong></td>
<td>
<select name="verbTense">
<option value="present" selected="selected">present</option>
<option value="presentParticiple">present participle</option>
<option value="past">past</option>
<option value="pastParticiple">past participle</option>
</select>
</td>
</tr>
<tr>
<td>&nbsp;</td>
<td>
<input type="checkbox" name="american" value="true" />
American spellings
</td>
</tr>
<tr>
<td>
&nbsp;
```

```
</td>
<td>
&nbsp;
</td>
</tr>
<tr>
<td valign="top">
<strong>Results format:</strong>
</td>
<td>
<input type="radio" name="media" value="json">JSON format</input><br />
<input type="radio" name="media" value="xml" checked="checked">XML
format</input><br />
<input type="radio" name="media" value="html">HTML format</input><br />
<input type="radio" name="media" value="text">Text format</input>
</td>
</tr>
<tr>
<td>
&nbsp;
</td>
<td>
&nbsp;
</td>
</tr>
<tr>
<td>
<input type="submit" name="conjugate" value="Conjugate" />
</td>
</tr>
</table>
</form>
```

Output

Here is sample verb conjugation output for the present tense of the verb "to be". Note that only "be" is entered as the infinitive value; the "to" need not be specified.

JSON output

```
{
  "VerbConjugatorResult": {
    "infinitive": "be",
    "verbTense": "present",
    "american": false,
    "firstPersonSingular": "am",
    "secondPersonSingular": "are",
    "thirdPersonSingular": "is",
    "firstPersonPlural": "are",
    "secondPersonPlural": "are",
    "thirdPersonPlural": "are"
  }
}
```

XML output

```
<VerbConjugatorResult>
    <infinitive>be</infinitive>
    <verbTense>present</verbTense>
    <american>false</american>
    <firstPersonSingular>am</firstPersonSingular>
    <secondPersonSingular>are</secondPersonSingular>
    <thirdPersonSingular>is</thirdPersonSingular>
    <firstPersonPlural>are</firstPersonPlural>
    <secondPersonPlural>are</secondPersonPlural>
    <thirdPersonPlural>are</thirdPersonPlural>
</VerbConjugatorResult>
```

HTML output (source)

```
<h3>Conjugation of present tense for infinitive "to be"</h3>
<table border="0">
<tr>
<td valign="top" align="left"><strong>First person singular:</strong></td>
<td valign="top" align="left">am</td>
</tr>
<tr>
<td valign="top" align="left"><strong>Second person singular:</strong></td>
<td valign="top" align="left">are</td>
</tr>
<tr>
<td valign="top" align="left"><strong>Third person singular:</strong></td>
<td valign="top" align="left">is</td>
</tr>
<tr>
<td valign="top" align="left"><strong>First person plural:</strong></td>
<td valign="top" align="left">are</td>
</tr>
<tr>
<td valign="top" align="left"><strong>Second person plural:</strong></td>
<td valign="top" align="left">are</td>
</tr>
<tr>
<td valign="top" align="left"><strong>Third person plural:</strong></td>
<td valign="top" align="left">are</td>
</tr>
</table>
```

HTML output (display)

Conjugation of present tense for infinitive "to be"

First person singular:	am
Second person singular:	are
Third person singular:	is
First person plural:	are
Second person plural:	are
Third person plural:	are

Text output

```
Conjugation of present tense for infinitive "to be"
First person singular:  am
Second person singular: are
Third person singular:  is
First person plural:    are
Second person plural:   are
Third person plural:    are
```

MorphAdorner Server Services: Version Service

Service name:	version
Service description:	MorphAdorner client and server versions.
HTTP methods allowed:	GET, POST, OPTIONS
POST accepts as input:	application/x-www-form-urlencoded
HTTP return codes:	200: service succeeded 400: service failed with an error
Query parameters	
media	Result format. One of *json, xml, html, text* .

Sample POST form

```
<form accept-charset="UTF-8" method="post" action="version"
      target="_blank"
      name="version">
<table cellpadding="0" cellspacing="5">
<tr>
<td valign="top">
<strong>Results format:</strong>
</td>
<td>
<input type="radio" name="media" value="json">JSON format</input><br />
<input type="radio" name="media" value="xml" checked="checked">XML
format</input><br />
<input type="radio" name="media" value="html">HTML format</input><br />
<input type="radio" name="media" value="text">Text format</input>
</td>
</tr>
<tr>
<td>
<input type="submit" name="version" value="Get version" />
</td>
</tr>
<tr>
</table>
</form>
```

Output

Here is sample output for the version service. The MorphAdorner client library version is provided as *morphAdornerVersion*, and the MorphAdorner server version is provided as *morphAdornerServerVersion*.

JSON output

```
{
  "VersionResult": {
    "morphAdornerVersion": "2.0.1",
    "morphAdornerServerVersion": 1.0.0
  }
}
```

XML output

```
<VersionResult>
    <morphAdornerVersion>2.0.1</morphAdornerVersion>
    <morphAdornerServerVersion>1.0.0</morphAdornerServerVersion>
</VersionResult>
```

HTML output (source)

```
<h3>Program Versions</h3>
<table border="0">
<tr>
<td valign="top" align="left"><strong>MorphAdorner version:</strong></td>
<td valign="top" align="left">2.0.1</td>
</tr>
<tr>
<td valign="top" align="left"><strong>MorphAdorner Server version:</strong></td>
<td valign="top" align="left">1.0.0</td>
</tr>
</table>
```

HTML output (display)

Program Versions

MorphAdorner version:	2.0.1
MorphAdorner server version:	1.0.0

Text output

```
Program Versions
MorphAdorner version:   2.0.1
MorphAdorner Server version:    1.0.0
```

MorphAdorner Server Services: Adorned XML to Tabular File

Service name:	teiadornedtotabularformat
Service description:	Convert adorned XML to tabular file.
HTTP methods allowed:	POST, OPTIONS
POST accepts as input:	application/x-www-form-urlencoded
HTTP return codes:	200: service succeeded 400: service failed with an error
Query parameters	
media	Result format. Only *text* allowed.
teifile	TEI input file.
resultsAsAttachedFile	Allowed values are *true* to send the results as an attached file, and *false* to send the results as a data stream.

Sample POST form

```
<form accept-charset="UTF-8" method="post"
      action="teiadornedtotabularformat"
      target="_blank"
      enctype="multipart/form-data" name="teiadornedtotabularformat">
<table cellpadding="0" cellspacing="5">
<tr>
<td>
<strong>Adorned TEI XML file:</strong>
</td>
<td>
<input type="file" name="teifile" size="50">
</td>
</tr>
<tr>
<td>&nbsp;</td>
<td>
<input type="checkbox" name="resultsAsAttachedFile" value="true"
       checked="checked"/>
Send results as attached file
</td>
</tr>
<tr>
<td>
&nbsp;
</td>
<td>
&nbsp;
</td>
</tr>
<tr>
<td colspan="2">
<input type="submit" name="convert" value="Convert" />
</td>
```

```
</tr>
</table>
</form>
```

Output

The output is a columnar tab-separated utf-8 encoded text file. The column order and contents are the same as those generated by the XMLToTab utility (page 77).

MorphAdorner Server Services: Adorn a TEI XML file Service

Service name:	teiadorner
Service description:	Adorn a TEI XML file.
HTTP methods allowed:	POST, OPTIONS
POST accepts as input:	application/x-www-form-urlencoded
HTTP return codes:	200: service succeeded 400: service failed with an error
Query parameters	
corpusConfig	Corpus configuration name. In the standard distribution these are *ece*, *eme*, and *ncf*.
media	Result format. Only *xml* allowed.
teifile	TEI input file.
resultsAsAttachedFile	Allowed values are *true* to send the results as an attached file, and *false* to send the results as a data stream.
useChoice	Use TEI XML choice structure to hold standard spellings. Allowed values are *true* to use the choice structure, *false* to emit the standard spellings as reg= attributes.

Sample POST form

```
<form accept-charset="UTF-8" method="post" action="teiadorner"
      target="_blank"
      enctype="multipart/form-data" name="teiadorner">
<table cellpadding="0" cellspacing="5">
<tr>
<td>
<strong>TEI XML file:</strong>
</td>
<td>
<input type="file" name="teifile" size="50">
</td>
</tr>
<tr>
<td>
&nbsp;
</td>
<td>
&nbsp;
</td>
</tr>
<tr>
<td>&nbsp;</td>
<td>
<input type="checkbox" name="resultsAsAttachedFile" value="true"
       checked="checked"/>
```

```
Send results as attached file
</td>
</tr>
<tr>
<td valign="top">
<strong>
Lexicon:</strong>
</td>
<td>
<input type="radio" name="corpusConfig" value="eme">Early Modern English</input><br
/>
<input type="radio" name="corpusConfig" value="ece">Eighteen Century
English</input><br />
<input type="radio" name="corpusConfig" value="ncf" checked="checked">Nineteenth
Century Fiction</input>
</td>
</tr>
<tr>
<td>&nbsp;</td>
<td>
<input type="checkbox" name="useChoice" value="true"/>
Use choice structure to emit standard spelling
</td>
</tr>
<tr>
<td>
&nbsp;
</td>
<td>
&nbsp;
</td>
</tr>
<tr>
<td colspan="2">
<input type="submit" name="adorn" value="Adorn" />
</td>
</tr>
</table>
</form>
```

Output

The input file is adorned. The output file is emitted in the same simple TEI P5 format the AdornedToSimpleTEIP5 (page 45) utility produces. Setting the *useChoice* parameter to true is the same as choosing the *usechoice* setting of AdornedToSimpleTEIP5. Setting the *useChoice* parameter to false is the same as choosing the *usereg* setting of AdornedToSimpleTEIP5. The output TEI XML is returned either as an attached file if *resultsAsAttachedFile* is true or as an XML stream if *resultsAsAttachedFile* is false.

MorphAdorner Server Services: Apply changes to adorned file service

Service name:	teiapplychangestoadornedfile
Service description:	Apply changes from change log to an adorned file.
HTTP methods allowed:	POST, OPTIONS
POST accepts as input:	application/x-www-form-urlencoded
HTTP return codes:	200: service succeeded 400: service failed with an error
Query parameters	
media	Result format. Only *xml* allowed.
origadornedfile	Original adorned TEI XML file.
changelogfile	Change log file.
revertChanges	Revert changes specified in a MorphAdorner change log file. Allowed values are *true* to revert changes or *false* to apply the changes.

Sample POST form

```
<form accept-charset="UTF-8" method="post"
      action="teiapplychangestoadornedfile"
      target="_blank"
      enctype="multipart/form-data" name="teiapplychanges">
<table cellpadding="0" cellspacing="5">
<tr>
<td>
<strong>Original adorned TEI XML file:</strong>
</td>
<td>
<input type="file" name="origadornedfile" size="50">
</td>
</tr>
<tr>
<td>
<strong>XML change log file:</strong>
</td>
<td>
<input type="file" name="changelogfile" size="50">
</td>
</tr>
<tr>
<td>&nbsp;</td>
<td>
<input type="checkbox" name="revertChanges" value="true" />
Revert changes
</td>
</tr>
```

```
<tr>
<td>&nbsp;</td>
<td>
<input type="checkbox" name="resultsAsAttachedFile" value="true"
       checked="checked"/>
Send results as attached file
</td>
</tr>
<tr>
<td colspan="2">
<input type="submit" name="applychanges" value="Apply Changes" />
</td>
</tr>
</table>
</form>
```

Output

A source adorned or tokenized TEI XML file specified by *origadornedfile* is updated or downdated by applying changes specified in a MorphAdorner change log file specified by *changelogfile*. If *revertChanges* is false, the changes in the change log file are applied to update the source file. If *revertChanges* is true, the changes are reverted to downdate the source file. The resultant modified file is returned either as an attached file if *resultsAsAttachedFile* is true or as an XML stream if *resultsAsAttachedFile* is false. This service produces the same output as the UpdateAdornedFile utility (page 75). The change log format is provided in the description of the CompareAdornedFiles utility (page 40).

MorphAdorner Server Services: Compare Adorned Files Service

Service name:	teicompareadornedfiles
Service description:	Compare two adorned files and generate change log.
HTTP methods allowed:	POST, OPTIONS
POST accepts as input:	application/x-www-form-urlencoded
HTTP return codes:	200: service succeeded 400: service failed with an error
Query parameters	
media	Result format. Only *xml* allowed.
origadornedfile	Original adorned TEI XML file.
updatedadornedfile	Updated adorned TEI XML file.
resultsAsAttachedFile	Allowed values are *true* to send the results as an attached file, and *false* to send the results as a data stream.

Sample POST form

```
<form accept-charset="UTF-8" method="post" action="teicompareadornedfiles"
      target="_blank"
      enctype="multipart/form-data" name="teicompare">
<table cellpadding="0" cellspacing="5">
<tr>
<td>
<strong>Original adorned TEI XML file:</strong>
</td>
<td>
<input type="file" name="origadornedfile" size="50">
</td>
</tr>
<tr>
<td>
<strong>Updated adorned TEI XML file:</strong>
</td>
<td>
<input type="file" name="updatedadornedfile" size="50">
</td>
</tr>
<tr>
<td>&nbsp;</td>
<td>
<input type="checkbox" name="resultsAsAttachedFile" value="true"
       checked="checked"/>
Send results as attached file
</td>
</tr>
<tr>
<td colspan="2">
<input type="submit" name="compare" value="Compare" />
```

```
</td>
</tr>
</table>
</form>
```

Output

The original adorned or tokenized TEI XML file specified by *origadornedfile* is compared with the updated adorned or tokenized TEI XML file specified by *updatedadornedfile*. The changes from the original file to the updated file are written to a MorphAdorner change log file. The change log is returned either as an attached file if *resultsAsAttachedFile* is true or as an XML stream if *resultsAsAttachedFile* is false. This service produces the same output as the CompareAdornedFiles utility. The change log format is provided in the description of the CompareAdornedFiles utility (page 40).

MorphAdorner Server Services: Extract text from TEI XML file service

Service name:	teitotext
Service description:	Extract text from a TEI XML file.
HTTP methods allowed:	POST, OPTIONS
POST accepts as input:	application/x-www-form-urlencoded
HTTP return codes:	200: service succeeded 400: service failed with an error
Query parameters	
media	Result format. Only *text* allowed.
teifile	TEI input file.
resultsAsAttachedFile	Allowed values are *true* to send the results as an attached file, and *false* to send the results as a data stream.

Sample POST form

```
<form accept-charset="UTF-8" method="post" action="teitotext"
      target="_blank"
      enctype="multipart/form-data" name="teitotext">
<table cellpadding="0" cellspacing="5">
<tr>
<td>
<strong>Adorned TEI XML file:</strong>
</td>
<td>
<input type="file" name="teifile" size="50">
</td>
</tr>
<tr>
<td>&&nbsp;</td>
<td>
<input type="checkbox" name="resultsAsAttachedFile" value="true"
       checked="checked"/>
Send results as attached file
</td>
</tr>
<tr>
<td>
&&nbsp;
</td>
<td>
&&nbsp;
</td>
</tr>
<tr>
<td colspan="2">
<input type="submit" name="extract" value="Extract Text" />
```

```
</td>
</tr>
</table>
</form>
```

Output

The text in the input TEI XML file is extracted and returned. This service works the same way as the ExtractTEIText utility (page 53).

MorphAdorner Server Services: Extract Sentences Service

Service name:	teiadornedtosentences
Service description:	Extract sentences from an adorned TEI XML file.
HTTP methods allowed:	POST, OPTIONS
POST accepts as input:	application/x-www-form-urlencoded
HTTP return codes:	200: service succeeded 400: service failed with an error
Query parameters	
media	Result format. Only *text* allowed.
resultsAsAttachedFile	Allowed values are *true* to send the results as an attached file, and *false* to send the results as a data stream.
mainTextOnly	*true* to return sentences only from main text, *false* to return sentences from all of the text.
teifile	TEI input file.

Sample POST form

```
<form accept-charset="UTF-8" method="post" action="teiadornedtosentences"
      target="_blank"
      enctype="multipart/form-data" name="teiadornedtosentences">
<table cellpadding="0" cellspacing="5">
<tr>
<td>
<strong>Adorned TEI XML file:</strong>
</td>
<td>
<input type="file" name="teifile" size="50">
</td>
</tr>
<tr>
<td>&nbsp;</td>
<td>
<input type="checkbox" name="mainTextOnly" value="true"
       checked="checked"/>
Only return sentences in main text
</td>
</tr>
<tr>
<td>&nbsp;</td>
<td>
<input type="checkbox" name="resultsAsAttachedFile" value="true"
       checked="checked"/>
Send results as attached file
</td>
</tr>
<tr>
<td>
```

```
&nbsp;
</td>
<td>
&nbsp;
</td>
</tr>
<tr>
<td colspan="2">
<input type="submit" name="extract" value="Extract Sentences" />
</td>
</tr>
</table>
</form>
```

Output

The output is return as a sequence of utf-8 encoded text lines, one sentence per line. When *mainTextOnly* is true, at least the first word of a sentence must be present in the main part of the te

MorphAdorner Server Services: Move notes in TEI XML file service

Service name:	teinotesmover
Service description:	Move notes in TEI XML file.
HTTP methods allowed:	POST, OPTIONS
POST accepts as input:	application/x-www-form-urlencoded
HTTP return codes:	200: service succeeded 400: service failed with an error
Query parameters	
media	Result format. Only *xml* allowed.
teifile	TEI input file.
resultsAsAttachedFile	Allowed values are *true* to send the results as an attached file, and *false* to send the results as a data stream.

Sample POST form

```
<form accept-charset="UTF-8" method="post" action="teinotesmover"
      target="_blank"
      enctype="multipart/form-data" name="teinotesmover">
<table cellpadding="0" cellspacing="5">
<tr>
<td>
<strong>Adorned TEI XML file:</strong>
</td>
<td>
<input type="file" name="teifile" size="50">
</td>
</tr>
<tr>
<td>&nbsp;</td>
<td>
<input type="checkbox" name="resultsAsAttachedFile" value="true"
       checked="checked"/>
Send results as attached file
</td>
</tr>
<tr>
<td>
&nbsp;
</td>
<td>
&nbsp;
</td>
</tr>
<tr>
<td colspan="2">
<input type="submit" name="movenotes" value="Move Notes" />
```

```
</td>
</tr>
</table>
</form>
```

Output

The notes in the input TEI XML file are gathered and moved into a <div> at the end of the text. This service works the same way as the MoveTEINotes utility (page 64).

MorphAdorner Server Services: TEI XML Tokenizer Service

Service name:	teitokenizer
Service description:	Tokenize a TEI XML file.
HTTP methods allowed:	POST, OPTIONS
POST accepts as input:	application/x-www-form-urlencoded
HTTP return codes:	200: service succeeded 400: service failed with an error
Query parameters	
corpusConfig	Corpus configuration name. In the standard distribution these are *ece*, *eme*, and *ncf*.
media	Result format. Only *xml* allowed.
teifile	TEI input file.
resultsAsAttachedFile	Allowed values are *true* to send the results as an attached file, and *false* to send the results as a data stream.

Sample POST form

```
<form accept-charset="UTF-8" method="post" action="teitokenizer"
      target="_blank"
      enctype="multipart/form-data" name="teitokenizer">
<table cellpadding="0" cellspacing="5">
<tr>
<td>
<strong>TEI XML file:</strong>
</td>
<td>
<input type="file" name="teifile" size="50">
</td>
</tr>
<tr>
<td>
&nbsp;
</td>
<td>
&nbsp;
</td>
</tr>
<tr>
<td>&nbsp;</td>
<td>
<input type="checkbox" name="resultsAsAttachedFile" value="true"
       checked="checked"/>
Send results as attached file
</td>
</tr>
<tr>
<td valign="top">
```

```
<strong>
Lexicon:</strong>
</td>
<td>
<input type="radio" name="corpusConfig" value="eme">Early Modern English</input><br
/>
<input type="radio" name="corpusConfig" value="ece">Eighteen Century
English</input><br />
<input type="radio" name="corpusConfig" value="ncf" checked="checked">Nineteenth
Century Fiction</input>
</td>
</tr>
<tr>
<td>
&nbsp;
</td>
<td>
&nbsp;
</td>
</tr>
<tr>
<td colspan="2">
<input type="submit" name="tokenize" value="Tokenize" />
</td>
</tr>
</table>
</form>
```

Output

The input TEI XML file is tokenized and xml:id elements are added to each token. Each token is contained in either a word <*w*> or a punctuation <*pc*> element. The output TEI XML is returned either as an attached file if *resultsAsAttachedFile* is true or as an XML stream if *resultsAsAttachedFile* is false.

MorphAdorner Server Services: Unadorn an adorned TEI XML file service

Service name:	teiunadorner
Service description:	Unadorn an adorned TEI XML file.
HTTP methods allowed:	POST, OPTIONS
POST accepts as input:	application/x-www-form-urlencoded
HTTP return codes:	200: service succeeded 400: service failed with an error
Query parameters	
media	Result format. Only *xml* allowed.
teifile	TEI input file.
resultsAsAttachedFile	Allowed values are *true* to send the results as an attached file, and *false* to send the results as a data stream.

Sample POST form

```
<form accept-charset="UTF-8" method="post" action="teiunadorner"
      target="_blank"
      enctype="multipart/form-data" name="teiunadorner">
<table cellpadding="0" cellspacing="5">
<tr>
<td>
<strong>Adorned TEI XML file:</strong>
</td>
<td>
<input type="file" name="teifile" size="50">
</td>
</tr>
<tr>
<td>&nbsp;</td>
<td>
<input type="checkbox" name="resultsAsAttachedFile" value="true"
       checked="checked"/>
Send results as attached file
</td>
</tr>
<tr>
<td>
&nbsp;
</td>
<td>
&nbsp;
</td>
</tr>
<tr>
<td colspan="2">
<input type="submit" name="unadorn" value="Unadorn" />
```

```
</td>
</tr>
</table>
</form>
```

Output

The output is a TEI XML file with the word-level adornments in the input TEI XML file removed. This service produces the same output as the Unadorn utility (page 74).

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 <div> 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 94).

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 <em>.

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.